STELLENBOSCH UNIVERSITY

UNDERGRADUATE THESIS

Creating Intelligent Agents with Reinforcement Learning

Author:
Elan VAN BILJON
Student Number: 18384439

Study leader:
Prof. JA DU PREEZ

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at the University of Stellenbosch

UNIVERSITEIT
STELLENBOSCH
UNIVERSITY

October 31, 2017

Declaration of Authorship

I, Elan VAN BILJON, declare that this thesis titled, "Creating Intelligent Agents with Reinforcement Learning" and the work presented in it are my own. I confirm that:

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is. / Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is. / I agree that plagiarism is a punishable offence because it constitutes theft.

3. Ek verstaan ook dat direkte vertalings plagiaat is. / I also understand that direct translations are plagiarism.

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is. / Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie. / I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Student Number: 18384439 Initials and surname: E. van Biljon

Signed: Date: October 31, 2017

Abstract

Creating Intelligent Agents with Reinforcement Learning

by Elan VAN BILJON

English

Reinforcement learning is a relatively new and undiscovered branch of machine learning. However, reinforcement learning has recently become very popular. Even so, very few understand what reinforcement learning is and possible applications thereof. This project report serves to give an overview of reinforcement learning and will explain some of the recently developed approaches, such as deep Q learning. Throughout this report, we build our understanding of reinforcement learning until we reach the level of deep Q learning. We then apply a deep Q network to a computer game, Code vs Zombies. While our implementation stabilised on a suboptimal policy when playing the full game, it was able to find optimal policies for constrained versions. In the process, we experiment with and optimise some of the leading approaches in reinforcement learning.

Afrikaans

Versterkingsleer is ń relatief nuwe tak van masjienleer. Maar versterkingsleer het onlangs baie gewild geraak. Tog verstaan baie min mense wat versterkingsleer is en moontlike toepassings daarvan. Hierdie projekverslag dien om ń oorsig van versterkingsleer te gee en sal sommige van die onlangs ontwikkelde tegnieke, soos diep Q leer, verduidelik. In hierdie verslag bou ons ons begrip van versterkingsleer tot ons die vlak van diep Q leer bereik. Ons pas dan ń diep Q-netwerk aan op ń rekenaarspel, Code vs Zombies. Terwyl ons implementering gestabiliseer het op ń suboptimale beleid tydens die speel van die volle speletjie nie, kon dit optimale beleide vind vir beperkte weergawes. In die proses eksperimenteer ons met, en optimaliseer sommige van, die leidende tegnieke in versterkingsleer.

Acknowledgements

I would like to thank the many people who have supported me throughout the development of this project. Without the support of my supervisor, Prof. Johan du Preez, this project would have turned out very differently and I am very thankful for his efforts. I would like to thank Benjamin Rosman and Ulrich Paquet who were quick to offer advice, insight and reassurance (when needed). On a similar note I would like to thank the organisers of the 2017 Deep Learning Indaba, Shakir Mohamed, Ulrich Paquet, Vukosi Marivate, Benjamin Rosman, Nyalleng Moorosi, Stephan Gouws, Willie Brink and Richard Klein, without whom I would be much less educated on machine learning in general. I would like to give the biggest thanks to my friends and family. They are the most understanding and supportive people one could ever hope for. Lastly, I would like to thank you, the reader, without whom this document would be meaningless.

Contents

Declaration of Authorship **iii**

Abstract **iv**

English . . . iv

Afrikaans . . . iv

Acknowledgements **v**

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Background . . . 1

1.2.1 Reinforcement Learning . . . 2

Model Based Methods . . . 3

Value Based Methods . . . 3

Policy Search Methods . . . 4

1.2.2 Q Learning . . . 4

1.2.3 Function Approximation . . . 5

Linear Combination of Basis Functions . . . 5

Neural Networks . . . 6

1.2.4 Deep Q Learning . . . 6

1.3 Literature Synopsis . . . 7

1.4 Problem Statement . . . 7

1.5 Objectives . . . 8

1.5.1 Project Objectives . . . 8

1.5.2 Project Report Objectives . . . 8

1.6 Contributions . . . 9

1.7 Overview . . . 9

2 Literature Study **14**
2.1 Abstract . . . 14
2.2 Content . . . 14
Reward Clipping . . . 15
Frame Skip . . . 15
State Rolling . . . 16
Experienced Replay . . . 16
Target Network Update Delay . . . 16
2.3 Perspective and Learnings . . . 16

3 Reinforcement Learning **17**
3.1 Introduction . . . 17
3.2 Objectives . . . 17
3.3 Content . . . 17
3.3.1 Value Based Methods . . . 21
3.3.2 Model Based Method . . . 22
3.3.3 Policy Search Method . . . 23
3.4 Conclusion . . . 24

4 Q Learning **25**
4.1 Introduction . . . 25
4.2 Objectives . . . 25
4.3 Content . . . 25
4.4 Conclusion . . . 29

5 Function Approximation **30**
5.1 Introduction . . . 30
5.2 Objectives . . . 30
5.3 Content . . . 30
5.3.1 Linear Function Approximation . . . 31
5.3.2 Neural Network . . . 33
5.4 Conclusion . . . 37

6 Deep Q Learning **38**
6.1 Introduction . . . 38
6.2 Objectives . . . 38

6.3 Content 38
6.4 Conclusion 42

7 Problem Description **43**
7.1 Introduction 43
7.2 Objectives 43
7.3 Content 43
7.3.1 Code vs Zombies 43
7.4 Conclusion 44

8 Environment Setup **45**
8.1 Introduction 45
8.2 Objectives 45
8.3 Content 45
8.3.1 Code vs Zombies 45
State 46
Reward 46
Penalty 47
Action 48
8.4 Conclusion 48

9 Software Engineering **49**
9.1 Introduction 49
9.2 Objectives 49
9.3 Content 49
9.3.1 The Program 49
9.3.2 Development Method 50
Agile Development 50
Test Driven Development 51
9.3.3 Unit testing 51
9.3.4 Version Control 51
9.3.5 Dynamic Programming 52
9.3.6 Type Checking 53
9.4 Conclusion 53

10 Implementation Issues **55**

10.1 Introduction . . . 55
10.2 Objectives . . . 55
10.3 Content . . . 55
10.3.1 Challenges . . . 55
Continuous Versus Discrete Action Spaces . . . 55
State encoding . . . 57
Large Simulation Memory Leaks . . . 58
Bugs in Code . . . 59
10.3.2 Optimisations . . . 59
Reward Normalisation . . . 59
Environment Data Carry Over . . . 60
10.3.3 Environment Generation . . . 60
10.3.4 Fixed Initial States . . . 60
Frame Skip . . . 60
State Rolling . . . 61
10.4 Conclusion . . . 61

11 Experimental Investigation **62**
11.1 Introduction . . . 62
11.2 Objectives . . . 62
11.3 Content . . . 62
11.3.1 Validation . . . 62
11.3.2 Network Architectures . . . 63
11.3.3 Reinforcement Learning Techniques . . . 66
11.3.4 Training Methods . . . 67
Experienced Replay . . . 67
Target Network Update Delay . . . 67
Network Update Delay . . . 68
Random Replay Memory Sampling . . . 70
Full Replay Memory Sampling . . . 70
11.3.5 Action Systems . . . 72
11.3.6 State Encoding Size . . . 73
11.3.7 Architecture Selection . . . 75
11.3.8 Reward Systems . . . 78
11.3.9 Chosen Architecture and Techniques Results . . . 79

11.4 Conclusion 81

12 Conclusions and Recommendations **82**
12.1 Introduction 82
12.2 Content 82
12.2.1 Learnings 82
12.2.2 Conclusions 82
12.2.3 Recommendations for follow up work 83
Policy Based Approach 83
Model Decomposition 83
Model Based Approach 84
Multiple Environments 84
General Adversarial Networks 84
Learning Approaches 85
Decision Understanding and Visualisation 85

A Project Planning Schedule **86**

B Outcomes Compliance **87**
B.1 Introduction 87
B.2 Content 87
B.2.1 ELO 1 - Problem Solving 87
What is Satisfactory Performance? 87
What Did the Candidate do to Satisfy this Outcome? 88
B.2.2 ELO 2 - Application of Scientific and Engineering Knowledge . . 88
What is Satisfactory Performance? 88
What Did the Candidate do to Satisfy this Outcome? 89
B.2.3 ELO 3 - Engineering Design 91
What is Satisfactory Performance? 91
What Did the Candidate do to Satisfy this Outcome? 92
B.2.4 ELO 4 - Investigations, Experiments and data analysis 92
What is Satisfactory Performance? 92
What Did the Candidate do to Satisfy this Outcome? 93
B.2.5 ELO 5 - Engineering Methods, Skills and Tools, Including Information Technology 94
What is Satisfactory Performance? 94

What Did the Candidate do to Satisfy this Outcome? 94
B.2.6 ELO 6 - Professional and Technical Communication 94
What is Satisfactory Performance? 94
What Did the Candidate do to Satisfy this Outcome? 95
B.2.7 ELO 9 - Independent Learning Ability 95
What is Satisfactory Performance? 95
What Did the Candidate do to Satisfy this Outcome? 96

C Code **97**

D Hyper-parameters **98**

List of Figures

1.1 The Agent Environment Loop: the cycle of the agent taking actions and the environment telling the agent how the state changed and how good the action was (reward value). 2
1.2 A graphical representation of a neural network. This computational graph like structure maps from a three dimensional input to a two dimensional output. 6
1.3 Validation scores being tracked over time during training for the two training methods. We see that network update delay seems to promote faster learning and deliver better performance. 12
1.4 Validation scores being tracked over time during training for our final model. We see that it performs significantly better than a random agent but does not find the optimal policy for this constrained version of the problem. 13

2.1 The architecture of the DQN model. It takes pixel data as input, passes this through several convolutional layers, then through a fully connected layer and gives action-value pairs as output. 15

3.1 Graphical representation of a Markov decision process. It shows that taking action a_0 while in state s_0 results in transitioning to state s_1, taking action a_3 while in state s_2 results in transitioning to state s_0 and so on. . . 19

5.1 A neuron showing the inputs x_0, x_1 and x_2 being multiplied by w_0, w_1 and w_2, respectively. This is them summed and passed to the activation function, the ReLU function in this case. The output of the activation function is then passed on as the output of the neuron, o. 34
5.2 A neural network with an input layer that consists of three neurons, an output layer consisting of two neurons and no hidden layers. As discussed later in this project report, typical input and output for our use case is the state encoding and action-value pairs, respectively. 35

7.1 Graphical representation of the Code vs Zombies environment. If a human (blue circle) enters the zombie's circle (red), it is killed. If a zombie enters the shooter's circle (orange), it is killed. 44

9.1 Abbreviated UML of the core code we developed and how it links to the most important API, Keras. It shows the inheritance structure that is present in the environment setup and the association structure that links the environment and the agent through the Interface class. 50

10.1 Two of the possible action sets. Figure 10.1a shows the direction of the possible actions for the cardinal method and Figure 10.1b shows the direction of the possible actions for the fixed point method. 57

11.1 We see two examples of logarithmic learning rate scans on different networks. We see Figure 11.1a is stable as there is a cluster of learning rates that provide similar score values. Figure 11.1b is unstable as there is very large score variation for a small learning rate change. 65
11.2 The variation in score between networks for their top learning rates. Each colour represents a different learning rate for each network. We see all networks and their variation in score, this is used to select the top performing and most stable network-learning-rate pairs (11.2a). 66
11.3 Validation scores being tracked over time during training for the two training methods. We see that network update delay seems to promote faster learning and deliver better performance. 69

11.4 The variation in score (11.4a) and time (11.4b) when comparing the network update delay method (in red) to the target network update delay method (in green). We see that network update delay seems to achieve far better scores in substantially less time. 70
11.5 Validation scores being tracked over time during training for the two sampling methods. We see that full replay sampling seems to promote slightly faster learning and deliver better performance. 71
11.6 The variation in score (11.6a) and time (11.6b) when comparing the full replay sampling method (in red) to the random replay sampling method (in green). We see that network update delay seems to achieve better scores in less time. 71
11.7 The variation in score (11.7a) and time (11.7b) when comparing the default (in red) default static (in green), larger (blue) and larger static (black) methods. We see that the default action system seems to out perform all other methods in terms of score and training time. 73
11.8 Validation scores for different state encoding sizes. We see that none of the networks were able to achieve an optimal score on the full state encoding size, 99. However, the smaller networks seem to have a higher chance of doing so. 74
11.9 Validation scores being tracked over time during training for multiple architectures for the smallest version of our problem. All networks converge to approximate optimal performance relatively quickly. 75
11.10Validation scores being tracked over time during training for multiple architectures for our problem at validation size. We see that no networks converge to the approximate optimal score. However, the larger networks certainly outperform the smaller ones. 76
11.11Validation scores being tracked over time during training for multiple architectures for our problem at validation size. We see that given more training episodes increases the likelihood of networks achieving approximate optimal performance. 77
11.12The maximum scores achieved by (Figure 11.12a) and the maximum score divided by the time taken to train (Figure 11.12b) each network. We see that larger networks reach higher maximum scores but have lower score per second return. 78

11.13Validation scores being tracked over time during training for the simple reward system. We see that our agent performs very poorly. 79
11.14Validation scores being tracked over time during training for network (512, 512, 256, 64). We see that it quickly outperforms the random score but does not achieve approximate optimal score. 80
11.15Validation scores being tracked over time during training for network (1024, 1024, 512, 512, 256, 256, 128, 128, 64, 64, 32). We see that it quickly outperforms the random score but does not achieve approximate optimal score. 81

A.1 Gantt chart showing planned and completed progress on this project. We see that most activities took longer than planned but the project was still completed on time. 86

List of Tables

11.1 Top Performing Network Architectures and Corresponding Learning Rates 66

D.1 Hyper-parameters 98

List of Algorithms

1 Q Learning 28
2 Q Update 28
3 ϵ-greedy Action Selection 29
4 Deep Q Learning 41
5 Deep Q Update 42
6 Unit Test Example 51
7 Slow Fibonacci 52
8 Fast Fibonacci - Using Dynamic Programming 53
9 Type Checking Example 53

Nomenclature

ML	**M**achine **L**earning
RL	**R**einforcement **L**earning
API	**A**pplication **P**rogram **I**nterface
MDP	**M**arkov **D**ecision **P**rocess
NN	**N**eural **N**etwork
TD	**T**emporal **D**ifference
DQN	**D**eep **Q** **N**etwork
CvZ	**C**ode **v**s **Z**ombies
UML	**U**niversal **M**odelling **L**anguage

List of Symbols

a	action
a_t	action at time step t
s	state
s'	new state
s_t	state at time step t
r	reward
r_t	reward at time step t
R_t	accumulated reward
T	transition function
R	Reward function
V	value function
V^*	optimal value function
Q	Q function / action-value function
Q^*	optimal Q function / optimal action-value function
w	weights
W	weights
$\widehat{f}$	approximation of f
q	basis function
NN	neural network function
α	learning rate / step size
γ	discount factor
π	policy / policy function
π^*	optimal policy / optimal policy function
θ	weights
ϵ	probability of acting randomly

1 Introduction

"Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed" (Samuel, 1959). We use machine learning to find underlying patterns in data and to make predictions about the world around us. Typically these models are myopic (do not take future consequences into account) and their predictions are instantaneous. have no impact on future predictions. Reinforcement learning is the branch of machine learning that addresses problems where sequential decisions need to be made. As such, current predictions do affect future predictions. Put another way, reinforcement learning is the branch of machine learning that is concerned with behaviours.

1.1 Motivation

Our entire life is a series of decisions. A science that attempts to aid us in making these decisions is, therefore, potentially very valuable. Reinforcement learning is being applied in many real life scenarios that have large impacts on our lives. The applications range from trying to decide which medications to give patients, and at what stage of their treatment, to planning how one should invest your money to ensure enough money to retire comfortably (Shin and Markey, 2006). This project may apply the techniques on a seemingly inconsequential problem, a computer game, but the learnings gained from this application are still valuable in itself and could easily be applied to real world problems.

1.2 Background

This section will give a brief overview of the knowledge needed to understand this project report. All the content covered in this section will be expanded on and explained in detail in future chapters.

1.2.1 Reinforcement Learning

As previously stated, reinforcement learning is the branch of machine learning that deals with modelling or learning behaviour. We call this learnt behaviour the policy of our agent. The agent is the entity that will be making the decisions, our reinforcement learning model. The agent is given some task that it must accomplish by acting on its environment. Environment is both the eyes of the agent and the entire world around it. The agent only knows what the environment tells it. Thus, especially in cases where the environment is stochastic, learning can be very difficult. The information between the agent and the environment is passed in the following manner: the environment tells the agent what the configuration of the environment (termed state) is, the agent performs an action and the environment then tells the agent how good its action was (a value termed the reward) and tells the agent what the new state of the environment is. This process is depicted graphically in Figure 1.1.

FIGURE 1.1: The Agent Environment Loop: the cycle of the agent taking actions and the environment telling the agent how the state changed and how good the action was (reward value).

One may notice that this sounds very similar to Markov decision processes. That is exactly what we model these interactions as. Our agent finds itself in some state, s. It performs some action, a. The environment changes by following some transition function, T. Then the environment gives feedback to the agent in the form of a reward value, r, and the next state the agent finds itself in, s'. The way the agent moves from state to state is its policy, π.

The main goal of reinforcement learning is to discover the optimal policy or the behaviour that best completes the task at hand. There are three main approaches to doing this: value based, model based and policy search methods. The goal of value based methods is to determine what the best policy by assigning a value to each state. The policy would then be to reach the states with highest value. Model based learning is the technique whereby we have our agent learn a internal approximation of

the environment. The agent can then predict how the environment will change when performing certain actions and, thus, has a mechanism to plan sequences of actions. Instead of using tools to infer the best policy, policy based approaches directly solve the policy. This is typically done by creating a parameterised function for our policy and tweaking the parameters until the optimal policy is found.

Model Based Methods

Model based methods try to use a supervised learning approach to model the transition, T, and reward, R, functions. In the real world environments are stochastic and the agent's internal approximation of the environment is never completely accurate. Thus, we model these functions probabilistically. We define our transition function to be of the form: $T(s, a, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$, where s and s_{t+1} are the states the agent found itself in at time t and $t + 1$, respectively, and a_t is the action it performed to transition between these states. As such, the transition function is the probability of transitioning to state s' from state s after performing action a. We define our reward function as follows: $R(s, a) = \mathbb{E}\{r_t | s_t = s, a_t = a\}$, where r_t is the reward received after performing action a_t. Thus, the reward function is the reward we expect to receive if we perform action a while in state s.

Value Based Methods

The value that is assigned to a state is directly linked to the rewards the agent receives for transitioning to that state. Similar model based approach, the value function also assigns value to a state probabilistically. We define the value function to be of the form:

$$V(s) = \mathbb{E}\{R_t | s_t = s\} = \mathbb{E}\left\{\sum_{t=0}^{\infty} (\gamma^t \cdot r(s_t, s_{t+1}))\right\}, \tag{1.1}$$

where V is the value function, γ is termed the discount factor, R_t is known as our accumulated reward and $r(s_t, s_{t+1})$ is the reward we get for transitioning from state s_t to state s_{t+1}. Because of our uncertainty about our environment and our model we add a coefficient that discounts the value of future states. The accumulated reward is the average payoff we can expect if we continue to interact with the environment in the same manner (under the same policy). We can now see that our value function is our expected accumulated reward.

Because we cannot solve this function using a supervised approach, we redefine our model so that it can be updated iteratively. Richard Bellman is known as the father of dynamic programming and he worked on control problems of a similar nature. He devised a way to represent these functions iteratively. We call these representations Bellman equations:

$$V(s) := R(s,a) + \gamma \cdot \sum_{s'}(T(s,a,s') \cdot V(s')). \tag{1.2}$$

We can now allow our agent to interact with the environment and use the information it gathers to update its value function. With each iteration it will get closer to the value function that gives the optimal policy.

Policy Search Methods

Researchers that using a value based technique is too indirect. The policy search approaches aim to directly solve the policy function. We define the policy function in two general forms, as a function that maps from states to actions, $\pi(s) = a$, and as a function that gives the probability of performing an action in a certain state $\pi(s,a) = P(a|s)$. Generally we favour the probabilistic approach. However, this is difficult and often impossible to solve directly since these models rarely contain closed form solutions. Thus, we parameterise the policy function and treat this as an optimisation problem:

$$\pi_\theta(s,a) = P(a|s,\theta), \tag{1.3}$$

where θ are the parameters (or weights) of the parameterised policy function. We now let our agent interact with the environment and find the set of weights that deliver the highest reward.

1.2.2 Q Learning

Because the previous form of our value function did not take the actions the agent takes into account, we are not utilising information we have access to. Q learning is a value based approach that deals with action-value functions. As the name suggests, this function assigns value to pairs of states and actions:

$$Q^{[i+1]}(s_t, a_t) := r_t + \gamma \cdot \max_a Q^{[i]}(s_{t+1}, a), \tag{1.4}$$

where Q is the action-value function, termed the Q function and the superscript $[i]$ shows that this is the i^{th} iteration of the function. This function is very similar to the value function defined previously, except for the introduction of the max operator. Simply put, this lets us link our current state value to the value of the most valuable adjacent state. This makes intuitive sense as if we are acting optimally, we will move from our current state to the one that will provide the best cumulative reward. This function lets us utilise all the knowledge at our disposal to infer the optimal policy.

1.2.3 Function Approximation

Because it is difficult and costly to solve for the true form of the previously mentioned equations, we employ techniques to approximate them instead. The classic approach is to use a linear set of basis functions to approximate our value or policy functions. However, this approach has limitations in terms of the dimensionality of the input data and with the types of functions it can approximate. Thus, newer function approximation methods, such as neural networks, have become more favourable recently and have been applied to solve a variety of interesting problems.

Linear Combination of Basis Functions

This method selects a set of functions, weight them with some coefficients and sum them to produce an approximation of the desired function:

$$f(x) \approx \widehat{f}(x, \theta)$$

$$\widehat{f}(x, \theta) = \sum_{k=0}^{n-1} w_k \cdot q_k(x), \tag{1.5}$$

where x is the input, $f(x)$ is the function we want to approximate, $\widehat{f}(x, \theta)$ is the approximation of $f(x)$, θ is the set of coefficients (or weights), n is the number of basis functions, w_k and q_k are the corresponding coefficient (or weight) and basis function, respectively (Konidaris, Osentoski, and Thomas, 2011). This method is very effective for many problems but cannot represent nonlinear relationships between basis functions. Neural networks do not have this limitation.

Neural Networks

To understand neural networks one need far more information than we can fit in this introduction. Thus, we elaborate in detail in Chapter 5 and we give a brief overview here. A Neural network is a model that takes inspiration from the structure of the human brain. Both the brain and neural networks are made out of neurons and synapses. Neurons are nodes where computation occurs and synapses are pathways (or edges) through which data flows. By connecting a series of nodes and edges together we can construct a sort of computational graph (Figure 1.2).

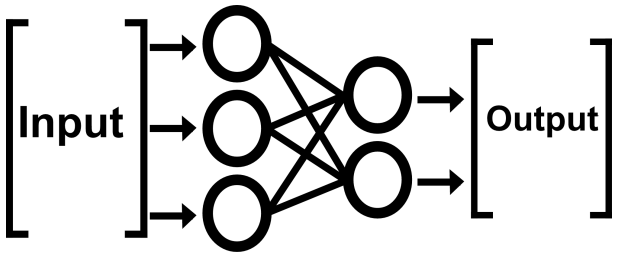


FIGURE 1.2: A graphical representation of a neural network. This computational graph like structure maps from a three dimensional input to a two dimensional output.

With this technique we can approximate any function, both linear and nonlinear. This is one of the largest reasons why neural networks have been so widely adopted.

1.2.4 Deep Q Learning

Deep Q learning is the name given to the technique whereby we approximate our Q (action-value) function using a neural network. Our approximated Q function can be expressed mathematically in the following manner:

$$\widehat{Q}(s_t, a_t, \theta^{[i+1]}) := r_t + \gamma \cdot \max_a \widehat{Q}(s_{t+1}, a, \theta^{[i]}), \tag{1.6}$$

where θ is the information about the neural network. When we use neural networks to approximate our Q function, it takes the following form:

$$\widehat{Q}(s_t, a_t, \theta^{[i]}) = NN(s_t, a_t, \theta^{[i]}), \tag{1.7}$$

where NN represents the function that the neural network applies. This has shown to be a very powerful technique with high real world capability.

1.3 Literature Synopsis

As this project report is largely a review of literature in the field of reinforcement learning, Chapter 2 discusses one of the most recent and important papers in deep Q learning to date. The paper discussed is Human-level control through deep reinforcement learning (Mnih et al., 2015). We explain the problem that they address, and optimisations and approaches they make use of. Their work is especially important in relation to this project report as it outlines the approach we followed and attempted to improve upon, in certain aspects.

In this paper Mnih et al. "develop a novel artificial agent, termed a deep Q-network (DQN), that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning". They test the agent on Atari 2600 games and show that their agent out performs all previous algorithms, and a professional human games tester, across 49 games. The structure of the model and the hyper-parameters were kept constant over all games. This shows that this is a robust approach that can generalise to, and excel in, different domains.

The paper outlines cutting edge methods and optimisations such as: frame skip, experienced replay and target network update delay.

1.4 Problem Statement

In this section we will give a brief description of the problem our program is to solve. This will be elaborated on in Chapters 7 and 8. We are tasked with creating a program that can play a computer game. The computer game in question is called Code vs Zombies. It is designed such that players write a program to control the shooter entity. This game is a top down view onto a two dimensional plane. This plane is inhabited by one shooter and a number of humans and zombies. The main objective of the game is to save as many humans from the zombies as possible by eliminating all zombies through controlling the shooter. The player receives a score dependent on the number of living humans and the number of zombies killed in each time step. The secondary objective is to maximise this score.

1.5 Objectives

There are two sets of objectives to be discussed in this project report. Firstly, the objectives that have to do with the practical aspect of the project and finally the objectives of this document itself.

1.5.1 Project Objectives

The practical aspect of the project revolves around programming a reinforcement learning model that can solve the problem described in the problem description: create an agent that can play the Code vs Zombies game. To create a program that can infer what reasonable moves could be made when only given the current configuration of the game and no rules about how the game works. To leave it at this would be too vague. Thus, we broke this objective into milestones:

- create a program that always produces valid output to the game environment
- create a program that makes moves that do not seem random to human observers
- create a program that achieves a higher score than a program that performs random moves
- create a program that will not lose a game that it is reasonably expected to win
- create a program that achieves a score higher than most human players
- create a program that achieves a score higher than traditional game AI heuristics

1.5.2 Project Report Objectives

The objectives of this document are of a different nature. This project report serves as an account of the work we have done in order to achieve the project objectives and, perhaps more importantly, give the reader the knowledge required to build on this work, should they chose to do so. This can be divided into the following objectives:

- give the reader an intuitive understanding of what reinforcement learning is and what problems it can be applied to
- give the reader an in depth understanding of the value based approach in reinforcement learning, the Q learning method specifically

- give the reader enough information to implement a value based reinforcement learning program themselves
- give the reader an extensive understanding of what function approximation is, how it can be implemented and when one would use it, all within the reinforcement learning domain
- give the reader a good understanding of how to approximate functions with neural networks and how they can be used with Q learning to form deep Q learning
- give the reader the necessary information to allow them to implement a deep Q learning reinforcement learning agent
- give the reader enough of an understanding of the field and methods we used to be able to understand our results and analysis thereof

1.6 Contributions

This section serves as an account of the work we did on this project that can be considered new, or as contributions to the field of reinforcement learning, or otherwise notable ways in which we achieved our project aims. These contributions are discussed in detail in Chapters 10 and 11 and the key aspects are mentioned in the overview section of this chapter.

- network update delay - a variation on Deep Minds target network update delay
- full replay memory sampling - a variation on the industry typical approach of sampling from replay memory
- reward normalisation - we could not find any indication of reward values typically used in the field, thus we did some investigation and have formed our approach and theory, also discussed in Chapter 8

1.7 Overview

We begin in Chapter 2 by discussing state of the art approaches to similar problems. We look at the work done by researchers at Deep Mind on deep Q learning on Atari games.

In Chapter 3 we start with the basics. We build our knowledge of reinforcement learning up from scratch so that we can understand all the steps in arriving at the deep Q learning approach. We learn that the agent's interaction with the environment can be modelled as a Markov decision process. We learn that reward functions are very important and difficult to construct. Finally, we learn about the three main approaches in reinforcement learning: model based, value based and policy search methods.

We start to delve more deeply in value based reinforcement learning approaches in Chapter 4. We learn, not only how to give value to a state, how to give value to actions performed in specific states. We also learn how to use dynamic programming techniques to solve for the optimal value function iteratively. We learn that there are limitations to this approach in terms of generalising between seen and unseen states and when dealing with high dimensional input data.

We start our exploration of more cutting edge techniques in Chapter 5 by learning about function approximation. We learn ways of approximating functions using linear combinations of basis functions and by using neural networks. We learn that by using a linear combination of basis functions we can often solve for the optimal coefficients in one step but that we cannot model nonlinear relationships between basis functions. We discover that neural networks can out perform in terms of modelling nonlinearities but we cannot solve for the optimal (or near optimal) coefficients in one step and must employ iterative techniques to do so.

In Chapter 6 we bring everything we have learnt thus far together to form the concept of deep Q learning. We discover that deep Q learning is the technique whereby we use neural networks to approximate our Q function. We discuss the mathematics behind this approach and learn how to implement it. Lastly, we learn about optimisations that can be made to the structure of our neural network to allow for assigning values to all actions associated with each state.

We go on to fully understand the Code vs Zombies problem and how its environment is constructed in Chapters 7 and 8. We learn what the specific rules of the game are that the agent must learn itself. We learn how the state of the environment is encoded, what from the agent must present its actions in and how the rewards for actions in states is calculated. We also discuss possible problems that the agent will encounter when learning from this information.

In Chapter 9 we explain how we developed the software for the agent. This includes the common software engineering practices currently used in industry. We explain the modular nature of our code, the inheritance structure for out objects and

what tools and application program interfaces (APIs) we used to develop the code for this project. We go through software engineering practices that resulted in more stable code, that contains fewer errors (one can never claim to have no errors). Finally, we explain development techniques that increased the rate of code production while keeping code functional.

We explain what implementations issues arose in the development of this project and how they were handled in Chapter 10. These issues include the fact that Q learning (and, by extension, deep Q learning) are designed for discrete action spaces and the Code vs Zombies problem has a continuous action space and the fact that the state information actually decreases as the game progresses but we must present a fixed size state encoding to the neural network. We also discuss optimisations that we made to our code throughout the development of the project. They include more effectively storing information that can be used later and generating initial game configurations in a way that our agent can learn more effectively.

Our experiments, and results there of, are explained in depth in Chapter 11. First we explain how we conducted our experiments: by tracking our agent's performance as it trained. We the explain the tests we used to find some networks than performed decently on the Code vs Zombies problem. At this point we explain a very important concept experiment and experiment. One of the things we are trained to do as engineers is to look more deeply into work others have done, to ask why they have done it in that particular way and then ask if there is a way to do it better. We did exactly this with Deep Mind's target network update delay training method.

We believed there were some optimisations to be made to their method. Thus, we trained our agent using target network update delay and we trained our agent using our method, network update delay. The details of both methods can be found in Section 11.3.4. We expected our method only to reduce training time but it offered more than that. The agent trained with network update delay also learnt faster with respect to the number of episodes played and provided better final performance (Figure 1.3).

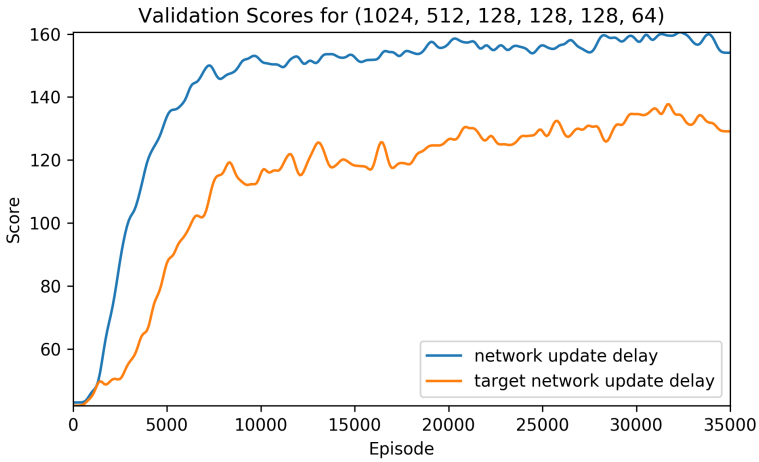


FIGURE 1.3: Validation scores being tracked over time during training for the two training methods. We see that network update delay seems to promote faster learning and deliver better performance.

We next questioned the current techniques of choosing what data we give our agents to learn from. The current standard is to select a number of data points uniformly from the data gathered while training. We noticed that this process is very computationally expensive and was the bottle neck for our system. Thus, we decided to reduce the number of data points we store during training and to train on all of the gathered data. We call our approach full replay sampling. This reduced training times and, surprisingly, increased the agents learning rate with respect to the number of seen episodes.

We go on to analyse different actions we can give our agent to use to solve the problem. We see that our original and most simple approach outperforms the rest. Similarly, we test our agent on numerous constrained versions of the Code vs Zombies problem. We discover that as the size of the state encoding increases the agent's chance of converging to the optimal solution decreases. We also see that the reward function we develop for the environment performs better than others.

After all of our experiments, we select the architecture we believe to have the best chance of solving the problem and test it on a constrained version of the problem. We see that it does not converge to the optimal policy for this constrained version (Figure 1.4). Thus, our agent is not capable of fulfilling all of the objectives laid out in Section 1.5 for the full problem. However we show that it does fulfil all stated objectives for some constrained versions of the problem.

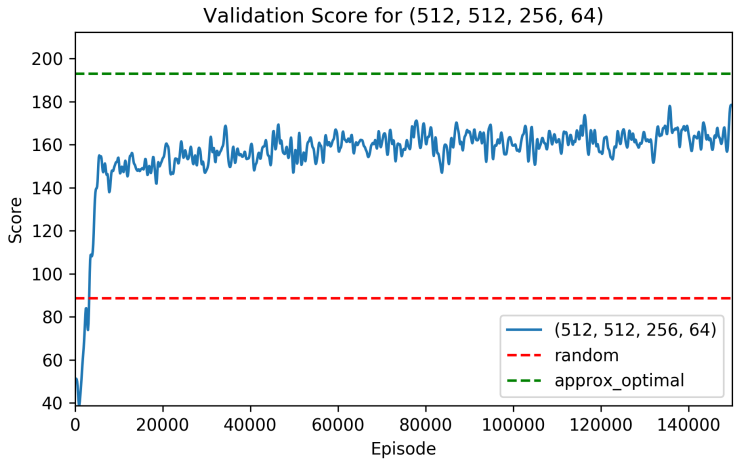


FIGURE 1.4: Validation scores being tracked over time during training for our final model. We see that it performs significantly better than a random agent but does not find the optimal policy for this constrained version of the problem.

Finally we summarise our findings and make recommendations for follow up work done on this problem in Chapter 12. We conclude that our agent can not solve the full Code vs Zombies problem. We then go on to recommend adaptions to our approach that may lead to better results. These recommendations include using a model based, policy search approach.

2 Literature Study

In this chapter we explain some of the current leading work done in reinforcement learning. We go over the basics of reinforcement learning in later chapters and work our way back to these approaches. In this chapter we discuss the paper, Human-level control through deep reinforcement learning, (Mnih et al., 2015) at a high level. We explain the problem that Mnih et al. address, and optimisations and approaches they make use of. This work is especially important in relation to this project report as it outlines the approach we followed, and attempted to improve upon in certain aspects.

2.1 Abstract

In Human-level control through deep reinforcement learning Mnih et al. "develop a novel artificial agent, termed a deep Q-network (DQN), that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning". They test the agent on Atari 2600 games and show that their agent out performs all previous algorithms, and a professional human games tester, across 49 games. The structure of the model and the hyper-parameters were kept constant over all games. This shows that this is a robust approach that can generalise to, and excel in, different domains.

2.2 Content

Mnih et al. structured their agent such that it took preprocessed pixel data as input and produced an action-value vector as output. Using the pixel data aided them in keeping the network the same across all tests. When one encodes raw information from different domains it can comprise of varying amount of data. This would lead to different sized encodings across games and different sized input layers to the network. However, all Atari 2600 games display using the same number of pixels, thus, can be encoded in a fixed size structure. The action-value vector contained an entry for each

of the actions one can perform using the Atari 2600 controls. The largest value in the action-value vector corresponds to the action the agent predicts to perform 2.1.

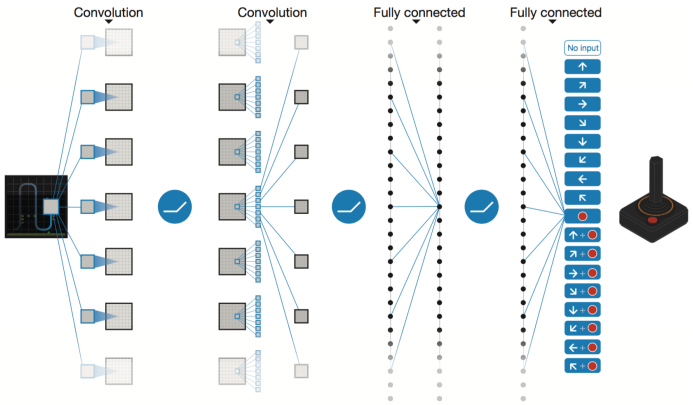


FIGURE 2.1: The architecture of the DQN model. It takes pixel data as input, passes this through several convolutional layers, then through a fully connected layer and gives action-value pairs as output.

Mnih et al. used a deep convolutional neural network, a machine learning model widely used on pixel data, to approximate the optimal action-value function. As such, they used a deep Q learning approach. The methods and optimisations they use are insightful and provide performance increases and reductions in training times. Thus, these methods are the main focus of this literature review.

Reward Clipping

"As the scale of scores varies greatly from game to game", Mnih et al. made a decision to clip all positive rewards at 1, all negative rewards at -1 and left 0 rewards untouched. This is done to limit the the magnitude of the error derivatives in the network and allows for the use of the same learning rate across multiple games. The weakness of this method is that the agent has lost any sense of which action may be better when presented with two positively rewarded actions.

Frame Skip

To allow for faster training and simulation, a simple frame-skipping technique was used. The agent is presented with every fourth frame and must select an action based on this frame. This action is then repeated for four frames and the process is repeated. Because the simulation time for the Atari 2600 emulator is negligible compared to one

forward pass of DQN, this allowed the agent to train on four times more games with a marginal time increase.

State Rolling

Because it is impossible to understand the state of some Atari 2600 games when given only one frame, Mnih et al. employed a technique we call state rolling. This is the process of keeping track of a number of sequential previous states your model has been presented with and concatenating them into a new state representation. This new state representation is then presented to the agent. This gives the agent a temporal understanding of the games.

Experienced Replay

In the most classic example of reinforcement learning the agent updates its policy only from the current observation. Experienced replay allows agents to store past observations and periodically train on a sample from this store. This is especially effective as the Bellman equation requires iterations to propagate information received from new data across the action-value function.

Target Network Update Delay

The DQN agent is comprised of two identical networks, a prediction network and a target network. The prediction network is used to assess each state and decide on actions. State, reward and action data is collected according to the prediction network's policy. This data is used to update the target network. After a number of updates of the target network, the weights are copied from the target network to the prediction network. This avoids oscillation of policies and leads to solution stability.

2.3 Perspective and Learnings

This paper provided us with an excellent source of techniques and approaches that could be employed to improve the performance and training efficiency of our agent. We go on to try and test the methods outlined in this paper on the Code vs Zombies problem.

3 Reinforcement Learning

3.1 Introduction

This chapter introduces the concept of reinforcement learning (RL) and the basic approaches that it consists of: model based, value based and policy search approaches. We gain the necessary background knowledge required to understand the model that this project report describes. We will first define what RL is and the terminology associated with it. We will then go on to compare RL to other classes of machine learning. Finally, we will go through the different approaches used in RL.

3.2 Objectives

We get an in depth understanding of value based methods and an intuitive understanding of model based and policy search methods. This will enable us to compare the model described in this project report to other approaches.

3.3 Content

We use RL in the following scenario: an entity exists that needs to interact with the environment it is in to accomplish a specific task. This is easily explained with an example. Imagine a mouse is put in a small maze. The mouse can be considered as the entity (usually called an agent), the maze would be the environment and the specific task is to find the centre of the maze and eat the piece of cheese found there. The mouse interacts with the maze by running through it, making decisions and performing actions, such as turning left or right. The reward for completing the maze is the piece of cheese. However, this reward is only received when the mouse has found the centre of the maze, thereby making the final decision. This means that most (all but one) of the mouse's decisions are made with no immediate reward being received for them. Yet

the mouse must still identify which decisions to make, to complete the task. This is the basic premise of RL. We want to create a model that acts as an agent in an environment and can complete the given task by making a series of sequential decisions, preferably the optimal set of decisions. Thus, RL is the branch of machine learning (ML) that addresses sequential decision making and behaviour modelling.

Unlike the case where there would be immediate direct feedback for each action, there are challenges that are associated with learning how to make good sequential decisions. Some include the questions of: When we get feedback, how do we know which actions in the sequence it corresponds to? How do my decisions affect the future? How do my actions change my environment?

Reinforcement learning is the branch of machine learning that exists in the grey space between supervised and non-supervised learning. In supervised learning you make one decision: "given this input, how can I best reproduce the given output?". Every input has a corresponding output to learn from, in other words, each input has a label. In the case of non-supervised learning you also make one decision: "how can I best divide the given data into meaningful groups?" or "given what I have learned from the data, which group does this observation best fit to?". In both of these cases, the decision is not based on what the models earlier predictions. Whereas, in the case of reinforcement learning the model must be able to make sequential decisions with little feedback. We seldom get a label telling our model if the decision it made was a good one and most of the time we get no label.

Before we can discuss how the agent learns to perform tasks, we must introduce some terminology and notation. The current configuration of the environment is termed the state. The way the agent decides to interact with the environment is termed the action. The agent's feedback from the environment is in the form of a scalar value, termed the reward. The state, the action performed and the reward received at time t are denoted by s_t, a_t and r_t, respectively.

Performing a_t while in state s_t will result in the environment changing and will produce a new state and reward, s_{t+1} (or s') and r_{t+1}, respectively. Thus, we can model this as a Markov decision process (Figure 3.1). This brings a large body of theory that can be used to solve the problem at hand (*Intro to reinforcement learning*).

We are now able to model this setup in the following fashion: We begin in some state, s, we make some action, a, our environment changes by some transition function, T, and we get feedback according to some reward function, R. A transition function is the equation that tells us how the environment changes when a certain action is

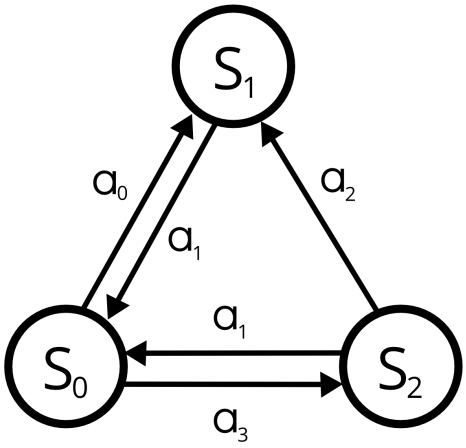


FIGURE 3.1: Graphical representation of a Markov decision process. It shows that taking action a_0 while in state s_0 results in transitioning to state s_1, taking action a_3 while in state s_2 results in transitioning to state s_0 and so on.

performed. It typically looks like this: $T(s, a, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ (*Intro to reinforcement learning*). The reward function tells us what reward we expect to get if we are in a certain state and we perform a particular action. It is typically of the form: $R(s, a) = \mathbb{E}\{r_t | s_t = s, a_t = a\}$.

If we model this setup as a Markov decision process, we can make the first order Markov assumption:

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_0, s_1, ..., s_t). \tag{3.1}$$

Intuitively, this means that our next state is only dependent on our current state, not the full history of states we have seen. This simplifies the task and makes modelling it easier.

The behaviour or strategy of our agent is termed its policy. Many things depend on our policy. For instance, the rewards we get greatly depend on our policy. We represent this mathematically in the following manner: $R^{\pi}(s, a)$, where π represents our policy function. The policy function is typically of the form $\pi(s) = a$. As such, it is a mapping from states to actions. In this case we say the reward function gives us our expected reward if we act under policy π (*Intro to reinforcement learning*).

As the creator of this agent, we need to choose a reward function that makes it very clear what the task is we want completed and how we want it to be completed. This is deceptively difficult feat to achieve. Imagine we task a computer with finding a way

of eliminating the common cold. The computer could decide that the most efficient way of doing so would be to eradicate all human life. This is clearly not the intended outcome. Anecdotes such as this have spurred conversations in artificial intelligence safety and illustrate the fact that we need to pay special attention to our choice of reward function.

Even if our reward function accurately encompasses our goal, we run into another challenge. If we leave our agent to train, it will start to accumulate some information about the environment. The challenge is knowing at what stage we should start this information. We may not know if the information we have gathered thus far is representative of the ground truth. This issue is known as the exploration vs exploitation problem. While we are gathering information and constructing notions about the environment, we are exploring. At some stage we want to start exploiting what we know to discover more by directing our exploration. We generally do this by defining a variable, ε, which represents our probability of exploring by performing a random action. We typically start with $\varepsilon = 1$ and slowly reduce it. This results in our agent initially performing entirely random actions, then slowly starting to use what it has learnt. We do not let ϵ reach zero while we are training as this may prohibit learning. Once we deploy our agent to perform the task at hand, we set ϵ to zero and thus it always performs the action it thinks is best. An algorithm that follows this approach is said to be ε-greedy (*Intro to reinforcement learning*).

Now that we have a high level idea of how to train these agents, we need to decide exactly what we want them to learn. As stated previously we want the agent to learn a policy (or behaviour) that completes the task. But more than that, we want them to complete the task in the best possible way. Thus, we want them to learn the optimal policy, the behaviour that best completes the task. We denote the optimal policy by π^*.

The next challenge to address is identifying how to learn the optimal policy. One way to do this is to construct a function that can tell us how desirable a state is to be in. We term this class of functions value functions. We measure the value of a state by predicting the rewards the agent would accumulate if it visited this state and acted under its current policy. However, we are not always certain that we can receive these rewards as the environment may be inherently stochastic. As such, we introduce the discount factor, $\gamma \in (0, 1]$, to achieve a trade off between immediate and delayed

rewards. We express the value function mathematically in the following form:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\{R_t | s_t = s\} = \mathbb{E}_{\pi}\left\{\sum_{t=0}^{\infty}(\gamma^t \cdot r_{\pi(s_t)}(s_t, s_{t+1}))\right\}. \tag{3.2}$$

In this equation we have become more specific with our reward values. Previously we simply denoted a reward received at time t by r_t. Now we denote the reward received by transitioning from state s_t to s_{t+1} while acting under policy $\pi(s_t)$ by $r_{\pi(s_t)}(s_t, s_{t+1})$. Further more, we denote the accumulated reward by R_t (*Intro to reinforcement learning*).

3.3.1 Value Based Methods

Now that we are familiar with the idea of a value function, we can move onto the class or reinforcement learning that uses them. Remember, in this approach the agent learns how to predict the value of the current state of the environment and this is used to infer a policy.

Once we have an accurate mapping of state value, we can adopt a policy that is to look at states adjacent to our current state and greedily pick the immediate best one. Thus, we shift our focus to obtaining this accurate mapping, known as the optimal value function, $V^*(s)$. Richard Bellman is known as the father of dynamic programming and he worked on control problems of a similar nature. He devised a way to represent these functions iteratively. We call these representations Bellman equations. As we cannot solve for $V^*(s)$ directly, we use a Bellman equation to rewrite the value function in recursive form so that we can solve for it iteratively. The value function now becomes:

$$V^{\pi}(s) := R(s, \pi(s)) + \gamma \cdot \sum_{s'}(T(s, \pi(s), s') \cdot V^{\pi}(s')), \tag{3.3}$$

remember our policy is of the form $\pi(s) = a$ and our transition function ($T(s, a, s')$) maps to a probability (*Intro to reinforcement learning*).

As previously stated, we want to find the optimal value function. This can be done by changing 3.3 slightly:

$$V^*(s) := \max_a\{R(s, a) + \gamma \cdot \sum_{s'}(T(s, a, s') \cdot V^*(s'))\}. \tag{3.4}$$

To explain 3.4 intuitively: we pick the action that maximises the sum of the immediate reward ($R(s,a)$) and the sum of the values of all adjacent states (those where $T(s,a,s')$ is greater than zero), weighed by γ (*Intro to reinforcement learning*).

From this optimal value function we can infer the optimal policy. We will discuss this method in more depth in later chapters as it is the focus of this project report. First, we will go over the two other main classes of reinforcement learning algorithms.

3.3.2 Model Based Method

When discussing value based methods, we saw the transition function, T, present in our value function, V. In later chapters we will show how to remove our dependance on T. However, it contains valuable information about the environment and we may well want to use it. This is what model based methods deal with, learning a model (or internal approximation) of the environment.

Having a model of our environment enables our agent to ask it questions, for instance: "How will my environment change if I perform this action?". This allows our agent to plan for the future. Not only that, the agent is able to quantify its uncertainty for any given transition between states and adjust its policy accordingly. It is more practical to use a model based approach when your agent must interact with the physical world. Imagine we are trying to teach a robot to make tea. This robot is likely very expensive and we would not want it to learn that pouring water on itself is bad by repeating it many times. Thus, we have it simulate its actions using its internal model. This way the agent can develop sensible policies with less interaction with the physical world.

This begs the question: "why did we not use a model based method in this project report?". We decided early on to use the state of the art model free (without an internal environment model) value based methods as an initial approach to the Code vs Zombies problem as they have solved the most interesting problems to date.

Let us briefly state what we already know about model based methods. We have a transition function ($T(s,a,s') = P(s_{t+1} = s' | s_t = s, a_t = a)$) that describes how our environment changes with our agents actions. We also have a reward function ($R(s,a) = \mathbb{E}\{r_t | s_t = s, a_t = a\}$) that provides feedback to our agent. In model based methods we attempt to learn the transition and reward functions. We set this up as a supervised learning problem.

We have our agent perform many random actions in its environment and collect information about the state transitions and the received rewards. We then construct a model that learns our transition function from the state transition data and similarly for our reward function, with the reward data.

The more astute reader will notice that we are missing something. We have a model of our environment that we can use but we do not have a system in place to learn the policy. We are still dependant on using a value based or policy search method in conjunction with our model based method. Since we have already discussed value based methods, let us move onto policy search methods.

3.3.3 Policy Search Method

In the previously mentioned methods, we would have our agent learn a value function ($V^{\pi}(s)$) that is dependent on our policy function ($\pi(s) = a$). In this way we can infer our policy from our value function. One might ask: "why not cut out the middle man?". The value function is a useful tool in certain applications but it is indeed possible to learn the policy directly.

As with all of the approaches we have discussed thus far, this method has advantages and disadvantages. This method is particularly effective in continuous action spaces and high dimensional spaces. An action space being the available set of actions our agent can perform, this can be discrete (for example: turn, left, move 1 unit north) or continuous (for example: move to x co-ordinate 4.2). Other methods can guarantee convergence to the optimal policy, this approach cannot.

We decided to try the method that showed state of the art performance on the most interesting problems first. Later we realised that a policy search method would probably yield better results as the environment we are dealing with has a continuous action space. It was however too late to change the focus of this project at the time of realisation. Thus, we recommend trying this approach if one decides to further this work.

The concept behind this approach is to define a set of parameters, θ, that we use to parameterise the policy function. It would then be of the form:

$$\pi_{\theta}(s, a) = P(a|s, \theta), \tag{3.5}$$

and as such, is now a probability distribution over possible actions. We can choose to model this in a number of different ways that will be explained in depth in later

chapters, but to give one example, it is possible to use a neural network. In this case θ represents the weights of the network. We would then treat this as an optimisation problem and try find the θ that would result in the optimal policy that best completes the task at hand (*Intro to reinforcement learning*).

3.4 Conclusion

From this chapter we learnt what RL is and the common approaches used in this field: model ("how does my environment change when I do this?"), value ("how good is this state?") based and policy ("what is the best action?") search methods.

4 Q Learning

4.1 Introduction

In this chapter we will explore a value based approach technique called Q learning. This concept is very important as it is the predecessor to deep Q learning, the approach we used for this project. First we will give a brief summary of value based methods, introduce the concept of temporal difference learning, action-value functions and finally we will use those concepts to define Q learning.

4.2 Objectives

We get an in depth understanding of what Q learning is and how it differs from the regular value based approach as well as what Q learning's strengths and weaknesses are.

4.3 Content

In the previous chapter we defined the following value function:

$$V(s) := R(s,a) + \gamma \cdot \sum_{s'}(T(s,a,s') \cdot V(s')), \tag{4.1}$$

where $V(s)$, $R(s,a)$ and $T(s,a,s')$ are our value, reward and transition functions, respectively. As previously noted, in this definition we are still dependent on an internal model of our environment. Ideally, we would like to remove this dependency as this would simply our agent. Luckily, our agent can interact with the actual environment to draw experience samples from it. As our agent explores the environment it gathers state, action, reward, new state data tuples, (s,a,r,s'). We can use this data to coax our value function, not to predict state transitions, to take state transitions into account.

We do this by introducing the concept of temporal difference learning. We define our recursive value function once more. This iteration contains a term we call the temporal difference error. We think of the temporal difference error as the prediction error between the current and previous iterations of our value function. We define the value function update as follows:

$$V^{[i+1]}(s) := V^{[i]}(s) + \alpha \cdot (\overbrace{\underbrace{r + \gamma \cdot V^{[i]}(s')}_{B} - V^{[i]}(s)}^{A}), \tag{4.2}$$

where A is termed the learnt value, B is the temporal difference error, r is the immediate reward, γ is the discount factor and α is the learning rate and the superscript in square brackets, $[i]$, is the number of the iteration (*Intro to reinforcement learning*). We see that if the temporal difference error is zero, we do not update our value function. This happens when the value function has converged to the optimal solution. From this reasoning, we see that the learnt value is exactly what we are trying to predict:

$$r + \gamma \cdot V^{[i]}(s') - V^{[i]}(s) = 0 \Rightarrow r + \gamma \cdot V^{[i]}(s') = V^{[i]}(s).$$

Notice that this value function does not take a very important piece of information into account, our agent's actions. To utilise this information we define the action-value function, Q:

$$Q^{[i+1]}(s_t, a_t) := Q^{[i]}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot Q^{[i]}(s_{t+1}, a_{t+1}) - Q^{[i]}(s_t, a_t)). \tag{4.3}$$

As the name suggests, this function assigns value to pairs of states and actions. In practise, we see that it is easier to extract the policy from the Q function than it is from the previous value function. It is important to note that we not only need the action the agent performed in this time step, a_t, we also need the action the agent wishes to perform in the next time step, a_{t+1}. The agent selects both of these actions by using its current policy, thus, this an on policy approach. This is apposed to an off policy approach, where actions need not be selected according to the current policy of the agent (*Intro to reinforcement learning*).

The advantage of an on policy approach is the fact that the agent learns a value function that takes future actions into account. The disadvantage is that your agent can only learn from data that corresponds to it's current policy. This does not make for

very efficient learning. For this reason, we favour an off policy approach, such as Q learning.

Using an off policy approach means that we can collect data to train on while our agent explores the environment. This also implies that we do not need direct access to the environment. Our agent simply needs observations from the environment to learn. In order for this to be an off policy approach, the Q function update must change to the following:

$$Q^{[i+1]}(s_t, a_t) := Q^{[i]}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \underbrace{\max_a Q^{[i]}(s_{t+1}, a)}_{A} - Q^{[i]}(s_t, a_t)). \tag{4.4}$$

Instead of choosing an action using our current policy, we chose the action that would lead us to the next most valuable state that our model is currently aware of, A. Note that this action does not need to be the next performed action (*Intro to reinforcement learning*).

We cannot train our agent yet as, until now, we have thought of the process of updating the Q function as infinite. Realistically there must be an end. Since our Q function is defined recursively, it does not account for this. As our Q function stands, it has no way of assigning any values to any states because we have no way of assigning a value to the last state. Thus, we make another change to the Q function:

$$Q^{[i+1]}(s_t, a_t) := \begin{cases} r_t & A \\ Q^{[i]}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q^{[i]}(s_{t+1}, a) - Q^{[i]}(s_t, a_t)) & else \end{cases},$$

where A is the condition: "if episode terminates at state s_{t+1}" (Mnih et al., 2015). An episode is a series of transitions beginning in the initial state and ending in the terminating state.

We can now formulate an implementation of Q learning. The following is a typical implementation of Q learning:

Algorithm 1 Q Learning

```
1  Q = initialise_Q()
2  num_episodes = get_number_of_training_episodes()
3  gamma = get_gamma()
4  for episode in num_episodes:
5      environment = initialise_environment()
6      while environment.episode_in_progress():
7          state = environment.get_state()
8          action = Q.choose_action(state)
9          environment.update(action)
10         new_state, reward = environment.observe()
11         is_terminating_state = environment.is_terminating_state(new_state)
12         Q.update(state, action, reward, new_state, is_terminating_state,
       gamma)
13     decay_epsilon()
```

In this sense, Q's underlying structure is very often a matrix with the same number of rows as there are states and the same number of columns as there are actions that can be performed. Thus, the Q update (line 12 of Algorithm 1) pseudocode could be the following:

Algorithm 2 Q Update

```
def update(state, action, reward, new_state, is_terminating_state, gamma):
    new_value = 0
    if is_terminating_state:
        new_value = reward
    else:
        best_action = argmax(Q[new_state])
        value_of_next_state = Q[new_state, best_action]
        new_value = Q[state, action] + alpha * (reward + gamma *
    value_of_next_state - Q[state, action])

    Q[state, action] = new_value
```

With this structure, we can also allow for ϵ-greedy decision making. We add support for this in the choose action function (line 8 in Algorithm 1):

Algorithm 3 ϵ-greedy Action Selection

```
def choose_action(state):
    epsilone = get_epsilon()
    action = None

    if random() < epsilon:
        action = get_random_action()
    else:
        action = argmax(Q.predict(state))

    return action
```

We see that this easily implementable. This implementation works well for a large variety of domains. However, the matrix required to store the values for Q, grows very quickly as the state space and action space increase in size. Set of available state configurations is termed the state space and the set of actions that can be performed in the environment is termed the action space. As the Q matrix increases in size, we need to train our agent for longer. Long enough for values to propagate through the Q matrix and stabilise.

Unfortunately, this implementation does not allow the agent to generalise well. If the agent encounters a state it had not seen in training, it will not be able to select a reasonable action. This problem, coupled with the issues surrounding high dimensional state and action spaces, lead us to use more advanced reinforcement learning techniques in this project. In the next chapter we discuss how we use function approximation to address these problems.

4.4 Conclusion

We have leant what Q learning is, how it differs from other value based approaches and what the steps in a Q learning algorithm are. We have also seen what Q learning's strengths and weaknesses are. We are well suited to understand the following chapters.

5 Function Approximation

5.1 Introduction

In the previous chapter we saw that the more traditional ways of representing functions in reinforcement learning (RL) has some fairly large flaws: they do not scale well and they do not generalise well. This is in part due to the fact that we have been trying to iterate to the exact function but this is not always necessary. In this chapter we will introduce the concept of approximating these functions in two ways. First, by using a linear combination of basis functions (termed linear function approximation) and then, by using neural networks. After this chapter we will understand all concepts that make up deep Q learning.

5.2 Objectives

We will gain an intuitive understanding of how to approximate functions with a linear function approximation and an in depth understanding of how to approximate functions using neural networks. We will acquire an intuition for when we should use each of these techniques and an understanding of the weaknesses of each.

5.3 Content

We have discussed many functions throughout this project report, such as value and policy functions. It is potentially impossible to solve for these functions' closed form solution. Thus, we need a way of approximating these functions. The most basic approach would be to choose a set of linear basis functions (such as Taylor or Fourier series) with which you will construct an approximation. The most popular method in the RL field at the moment is to use neural networks. Before we discuss what neural networks are and how we use them to approximate functions, lets discuss linear function approximation.

5.3.1 Linear Function Approximation

There are two techniques that fall under this category that we are quite familiar with: Fourier series and Taylor series expansion. The idea being: we define a set of basis functions, each with a corresponding coefficient, we multiply each basis function with its given coefficient and we sum all of them together to form an approximation of our desired function. It can be expressed mathematically by the following:

$$f(x) \approx \widehat{f}(x, \theta)$$

$$\widehat{f}(x, \theta) = \sum_{k=0}^{n-1} w_k \cdot q_k(x), \tag{5.1}$$

where x is the input, $f(x)$ is the function we want to approximate, $\widehat{f}(x, \theta)$ is the approximation of $f(x)$, θ is the set of coefficients (or weights), n is the number of basis functions, w_k and q_k are the corresponding coefficient (or weight) and basis function, respectively (Konidaris, Osentoski, and Thomas, 2011). In this case our input is a scalar and our basis functions take scalar inputs and produce scalar outputs. If we allow our input to be a vector but constrain our basis functions to accept and produce scalar inputs and outputs, respectively, we get the following equation:

$$\widehat{f}(x, \theta) = \sum_{k=0}^{n-1} \sum_{j=0}^{d-1} w_{k,j} \cdot q_k(x_j), \tag{5.2}$$

where d is the dimensionality of the input and $w_{k,j}$ is the weight that corresponds to the k^{th} basis function and the j^{th} input vector entry.

Recall from the previous chapter that our Q function is defined as follows:

$$Q^{[i+1]}(s_t, a_t) := Q^{[i]}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q^{[i]}(s_{t+1}, a) - Q^{[i]}(s_t, a_t)). \tag{5.3}$$

Using linear function approximation we can approximate our Q function as follows:

$$Q^{[i]}(s_t, a_t) \approx \widehat{Q}(s_t, a_t, \theta^{[i]})$$

$$\widehat{Q}(s_t, a_t, \theta^{[i]}) := \widehat{Q}(s_t, a_t, \theta^{[i]}) + \alpha \cdot (r_t + \gamma \cdot \max_a \widehat{Q}(s_{t+1}, a, \theta^{[i]}) - \widehat{Q}(s_t, a_t, \theta^{[i]})), \tag{5.4}$$

where $\widehat{Q}$ is of the form:

$$\widehat{Q}(s_t, a_t, \theta^{[i]}) = \sum_{k=0}^{n-1}(\sum_{j=0}^{d-1}(w^{[i]}_{s_{k,j}} \cdot q_k(s_{t_j})) + w^{[i]}_{a_k} \cdot q_k(a_t)), \tag{5.5}$$

s_{t_j} is the j_{th} item in the state vector, $w_{s_{k,j}}$ is the weight corresponding to the k^{th} basis function and the j_{th} item in the state vector and w_{a_k} is the weight corresponding to the k^{th} basis function and the action, a_t. Note that the iteration superscript has moved to the weights, this is to show that the weights change every iteration of the update.

We can see that the inclusion of the temporal difference error makes this equation cluttered and lengthy. Luckily, we have a solution. Now that we have a parameterised approximation function, we can define a loss function for our model. Recall that a loss function is, put briefly, a function that tells us how good our current weights are at approximating the original function, given the data we have. The smaller the value of our loss function, the better our approximation is. We can define our loss function in the following way such that it takes our temporal difference error into account:

$$L(\theta^{[i]}) = \mathbb{E}_{s_t, a_t, r_t, s_{t+1}}\{(r_t + \gamma \cdot \max_a \widehat{Q}(s_{t+1}, a, \theta^{[i]}) - \widehat{Q}(s_t, a_t, \theta^{[i]}))^2\} \tag{5.6}$$

where L is the loss function (Mnih et al., 2015). We note that this is actually the mean squared error (or mean squared temporal difference error) loss function. This simplifies our approximate Q function to the following:

$$\widehat{Q}(s_t, a_t, \theta^{[i+1]}) := r_t + \gamma \cdot \max_a \widehat{Q}(s_{t+1}, a, \theta^{[i]}). \tag{5.7}$$

The strength and weakness of linear function approximation lies both in its simplicity. The strength of this technique is that the derivative, with respect to the weights, of the loss function is convex and, thus, contains one easily obtainable (global) optimum. In a large portion of cases we can solve for the optimal set of weights directly by taking the derivative, with respect to the weights, of the approximated function and setting it equal to zero. In cases where we cannot do this, we can use an iterative method, such as gradient decent, to find the global optimum.

The weakness of this approach is that in many cases the number of basis functions needed for good approximations grows exponentially. If our input is d dimensional, the number of basis functions for a full p^{th} order Fourier approximation is $n = 2(p+1)^d$ (Konidaris, Osentoski, and Thomas, 2011, p. 3). Since the number of

weights required for this approximation is $nd = 2d(p+1)^d$, we see the memory needed to store the weights for high dimensional problems is too large to be practical. Not only that, the computational power needed to find the optimum may also be impractical.

Aside from this, many interesting problems require us to approximate nonlinear functions. This approach only allows for approximation of nonlinear functions if the set of basis functions contains nonlinear functions. Even so, this approximation method does not allow the modelling of nonlinear relationships between the basis functions.

As the problem this project report is addressing is high dimensional, we do not make use of this approximation method. Thus, we move onto another function approximation method that has shown excellent results when dealing with high dimensional problems.

5.3.2 Neural Network

Note: we have omitted the biases in this explanation to aid in clarity, brevity and to allow easier comparison between neural networks and linear function approximation.

A neural network is given its name as it is modelled in a similar fashion to our understanding of how the brain is structured. It is made up of a series of neurons connected by synapses. We represent this as a mathematical graph where the neurons are the nodes and the synapses are the directed edges. Computations take place in the nodes and data flows along the edges.

Neurons take each input and multiplies it with a corresponding coefficient, termed a weight. The result of all the multiplications are then summed and a function (usually nonlinear), termed the activation function, is applied to the result. The output of the activation function is the output of neuron and is passed, along the connected edges, as input to the next neuron. The mathematics of the events in the neuron are as follows:

$$z = f(\sum_{k=0}^{d-1} w_k \cdot x_k) = f(w \cdot x), \tag{5.8}$$

where o is the output of the neuron, f is the activation function, x is the input vector with dimensionality d, x_k and w_k are the k^{th} entry in the input and the corresponding weight, respectively and w is the vector of weights for this neuron (Lippmann, 1987). Figure 5.1 shows this graphically.

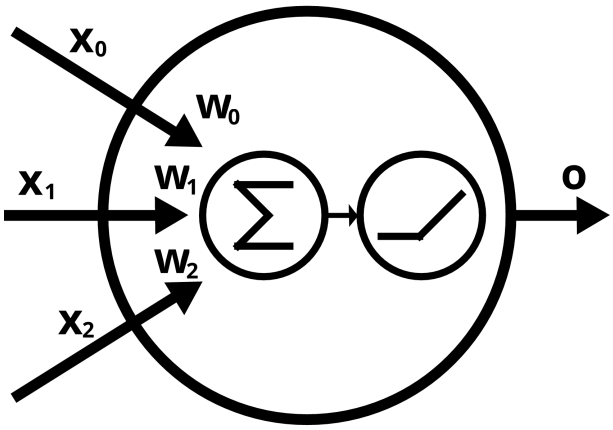


FIGURE 5.1: A neuron showing the inputs x_0, x_1 and x_2 being multiplied by w_0, w_1 and w_2, respectively. This is them summed and passed to the activation function, the ReLU function in this case. The output of the activation function is then passed on as the output of the neuron, o.

As previously stated, a neural network is made up of many neurons connected together. These neurons are connected in layers, as it is termed. The first layer, termed input layer, is where the data is fed into the network. The input layer is slightly different to the other layers as each neuron has only one input and no computation occurs within the input layers neurons. The input layer simply passes the input it receives on to every neuron in the next layer. Thus, there are the same number of neurons in the input layer as there are elements in your input.

The mathematics of the function applied by a layer can be expressed in the following manner:

$$z = Wx \leftarrow z_j = \sum_{k=0}^{d-1} w_{j,k} \cdot x_k = w_j \cdot x \tag{5.9}$$

$$h \leftarrow h_j = f(z_j), \tag{5.10}$$

where z is the output vector of this layer, W is the matrix containing all weights for this layer, x is the input vector to this layer (which is technically also the input to each neuron), z_j is the j^{th} entry in z, $w_{j,k}$ is the k^{th} entry in the weight vector for neuron j and w_j is the weight vector for neuron j (Ramchoun et al., 2016). The process of applying an input and propagating the information through the neural network to receive an output is termed a forward pass.

Typically, every neuron in a given layer is connected to both every neuron in the previous layer (directed edge for data coming into this neuron) and every neuron in the next layer (directed edge for data leaving this neuron). However, the last layer of the network, termed the output layer, is slightly different as there is no next layer to pass data to. The output of each neuron in the output layer forms an element in your output.

Thus, if we have input data $x \in \mathbb{R}^n$ and we want our network to predict output data $y \in \mathbb{R}^m$, our neural network must have n and m neurons in the input and output layers, respectively. The layers in between are termed the hidden layers. There can be any number of hidden layers and they can contain any number of neurons. The structure of the neural network is termed the architecture. Figure 5.2 shows a graphical representation of a small neural network.

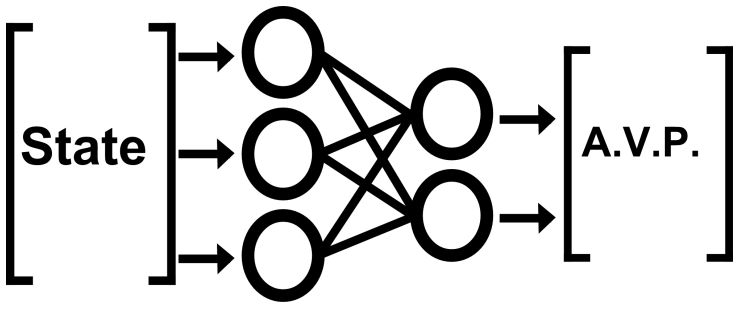


FIGURE 5.2: A neural network with an input layer that consists of three neurons, an output layer consisting of two neurons and no hidden layers. As discussed later in this project report, typical input and output for our use case is the state encoding and action-value pairs, respectively.

As such, neural networks are a way of modelling a function that maps from some input vector to some output vector. This can be adapted to work on tensors of arbitrary shape but that is outside the scope of this project report. This is represented mathematically by the following equation: $y = NN(x, \theta)$, where NN and θ are the function that the neural network applies and the collective weights of all neurons (termed the weights of the neural network) respectively.

For example, we can represent a three layer neural network (by this we mean a

neural network with one input layer, one hidden layer and one output layer) mathematically as follows:

$$\begin{aligned} y = NN(x, \theta) &= f(W_2 f(W_1 x)) \\ &= f(W_2 f(z_1)) \\ &= f(W_2 h_1) \\ &= f(z_2) \\ &= h_2, \end{aligned}$$

where x is the input to the neural network (or the input to the input layer), W_2 is the weight matrix for the output layer, f is the activation function, W_1 is the weight matrix for the hidden layer, h_1 is the output of the hidden layer and h_2 is the output of the neural network (or the output of the output layer). Note that in this case f takes in a vector and applies the activation function to every entry in the vector and outputs a vector of exactly the same dimensionality.

It can be shown that given enough layers, with enough neurons, and enough training data a neural network can approximate any function to arbitrarily high precision for a specific section of its domain (Hornik, Stinchcombe, and White, 1989). Thus, they should clearly be considered when we want to approximate a function. Naturally, for each application we must assess when enough is enough. This is not always easy.

In practice, we see that we can approximate interesting functions very well with relatively few weights. Thus, this approach has been favoured over using linear function approximation. In fact, this approach is quite similar to using linear function approximation when considering one neuron. Each neuron outputs the weighted sum of it's inputs passed through an activation function. Thus, we could make the inputs to the neuron a set of basis functions and set the activation function to be $f(x) = x$ (known as the linear activation function) and we would have an identical setup. However, we have layers of neurons, meaning that the output of the previous layer forms the new basis functions to be the input of the current layer. This, coupled with nonlinear activation functions is where the true strength of the neural network lies. Essentially, a neural network tunes its basis functions such that they best approximate the desired function. The neural network is also able to tune these basis functions to be nonlinear. That is why neural networks are so powerful. Note: neural networks can only model nonlinear functions if the activation function is nonlinear.

In the next chapter we will see how we can approximate the Q function using this technique. To do this we will have a look at how the network is trained and the loss function associated with this. First, we will briefly discuss the weakness of this approach.

The weakness of this technique is the fact that the derivative of the loss function, with respect to the weights of the network, is not convex and, thus, cannot be directly solved for. In truth, even if the derivative of the loss function is convex, it cannot always be solved for. Moreover, there is no guarantee that the global optimum will be found when using iterative methods to find the weights. When using this approach we often have to accept a local minimum that is good enough for our application.

5.4 Conclusion

We have learnt what what function approximation is and why we would want to use it. We have gained an intuitive understanding of linear function approximation and an in depth understanding of how to use neural networks to approximate a given function. We have also acquired some insight into when each of these methods would be used. Briefly, if your input data is low dimensional, linear function approximation and use neural networks otherwise.

6 Deep Q Learning

6.1 Introduction

In this chapter we use all of the previously discussed concepts to build our understanding of deep Q learning, the model this project made use of. We will first discuss what deep Q learning is by linking it to concepts we have already discussed, such as value functions, Q learning and function approximation using neural networks. We will then explain the mathematics for this approach. Finally, we discuss why we used this technique in this project and if it was the best choice to do so.

6.2 Objectives

We gain an understanding of deep Q learning by connecting it to previously discussed concepts. We gain insight into how the process works mathematically. We see what optimisations have been made to deep Q learning and we acquire intuition into when deep Q learning should be used.

6.3 Content

Simply, deep Q learning is the technique whereby a deep neural network is used to approximate the action-value function, in the Q learning technique.

In previous chapters we saw that the aim of the value based approach is to assign a value to a given state (environment configuration) and the policy is inferred from these values. We then introduced action-value functions whereby we took the agent's actions into account when evaluating states. We defined the temporal difference error and used it to remove our dependancy on the state transition function. Finally, we introduced Q learning as an off policy approach so that we could learn from any valid data.

We saw that traditional Q learning approaches do not perform well in high dimensional state or action spaces and do not generalise well. Thus, we discussed how we can overcome these problems by using function approximation. We now show the process of using a neural network to approximate the Q function.

We can use the loss function, L, and the definition of the approximated Q function, $\widehat{Q}$, from the previous chapter:

$$L(\theta^{[i]}) = \mathbb{E}_{s_t,a_t,r_t,s_{t+1}}\{(r_t + \gamma \cdot \max_a \widehat{Q}(s_{t+1}, a, \theta^{[i]}) - \widehat{Q}(s_t, a_t, \theta^{[i]}))^2\}, \tag{6.1}$$

$$\widehat{Q}(s_t, a_t, \theta^{[i+1]}) := r_t + \gamma \cdot \max_a \widehat{Q}(s_{t+1}, a, \theta^{[i]}), \tag{6.2}$$

where s, a, r, θ and γ are our state, action, neural network weights and discount factor, respectively. Note that the function is solved iteratively, thus, the $[i+1]$ and $[i]$ superscripts distinguish between iterations.

The difference between using neural networks and linear function approximation is the form of $\widehat{Q}$. When we use neural networks to approximate our Q function, it takes the following form:

$$\widehat{Q}(s_t, a_t, \theta^{[i]}) = NN(s_t, a_t, \theta^{[i]}), \tag{6.3}$$

where NN represents the function that the neural network applies. We have now defined everything we need to learn how to train our agent. However, we can make an optimisation that will make this approach more effective.

Q learning (and deep Q learning) is used to learn policies in what we call discrete, abstract action spaces. By this we mean that we label some action that can be taken in the environment as action 0, the next as action 1 and so on. Action 0 may represent any action, such as taking a step forward, the same goes for all actions. This is useful as we can then store the action-value pairs in a vector, y, in a way where the index of the vector corresponds to the action taken. For instance: y_0 would be the value associated with taking action 0 in the current state. This allows us to make an optimisation. We change the structure of our neural network to take only the state as input and to predict the action-value pairs for all actions (Mnih et al., 2015). This is achieved my having n neurons in the input layer, where n is the dimensionality of the state vector, and m neurons in the output layer of the of the neural network, where m is the dimensionality of our action space (the number of available actions). When an update is to be made, we feed the current state into the neural network and receive a vector of action-value pairs, the previously mentioned y. We then only update the value with

the corresponding action index:

$$y_{a_t} := r_t + \gamma \cdot \max_a y', \tag{6.4}$$

or, more explicitly:

$$NN(s_t, \theta^{[i+1]})_{a_t} := r_t + \gamma \cdot \max_a NN(s_{t+1}, \theta^{[}i]). \tag{6.5}$$

A small example can make this process very clear. Imagine the environment gives state encoding s_t to our agent. Our agent only has two actions it can perform (action 0 and action 1) and picks $a_t = 1$. The agent gives a_t to the environment and the environment then responds with s_{t+1} and $r_t = 0.75$. Let us say we defined $\alpha = 1$ and $\gamma = 0.9$ before we started training. We then feed s_t into our neural network and receive action-value vector $y = [\ 0.7 \quad 0.1\ ]$. We do the same with s_{t+1} to obtain $y' = [\ 0.4 \quad 0.6\ ]$. We can now substitute values into the above equations:

$$\begin{aligned} y_{a_t} &:= r_t + \gamma \cdot \max_a y' \\ y_1 &:= r_t + \gamma \cdot \max_a [\ 0.4 \quad 0.6\ ] \\ &:= 0.75 + 0.9 \cdot 0.6 \\ &:= 1.29 \\ \therefore y &:= [\ 0.7 \quad 1.29\ ]. \end{aligned}$$

We then train our neural network weights so that:

$$NN(s_t, \theta^{[i+1]}) := y := [\ 0.7 \quad 1.29\ ].$$

This structure greatly improves performance when assessing what action to take as we need only do one forward pass of the neural network to obtain all action-value pairs. Previously we would have needed to do a forward pass for each action and if we have a large action space and a large neural network this becomes very computationally expensive.

Remember the adaption made to the Q function for the final state in the episode:

$$\widehat{Q}(s_t, a_t, \theta^{[i+1]}) := \begin{cases} r_t & A \\ r_t + \gamma \cdot \max_a \widehat{Q}(s_t, a_t, \theta^{[i]}) & else \end{cases},$$

where A is the condition: "if episode terminates at state s_{t+1}" (Mnih et al., 2015).

We finally know all we need to know to train our neural network and set up a deep Q learning agent. As with linear function approximation, we have defined a loss function for our current approximation. We take the derivative of this loss function and use a technique called gradient decent to adjust the weights of our neural network to better approximate our Q function. The full nature of gradient decent is outside of the scope of this project report. Simply, gradient decent is a technique whereby we inspect the gradient of our loss function and change our network weights such that they follow the gradient towards a point where the loss function has a lower value. Thus, our approximation becomes better.

Now that we know the mathematics behind this approach, let us have a look at how it would be implemented. The following is pseudocode for a typical implementation of deep Q learning:

Algorithm 4 Deep Q Learning

```
Q = initialise_neural_network()
num_episodes = get_number_of_training_episodes()
gamma = get_gamma()
for episode in num_episodes:
    environment = initialise_environment()
    while environment.episode_in_progress():
        state = environment.get_state()
        action = Q.choose_action(state)
        environment.update(action)
        new_state, reward = environment.observe()
        is_terminating_state = environment.is_terminating_state(new_state)
        Q.update(state, action, reward, new_state, is_terminating_state,
    gamma)
  decay_epsilon()
```

As we can see, the process remains largely the same. The real difference is apparent in how we update the Q function:

Algorithm 5 Deep Q Update

```
def update(state, action, reward, new_state, is_terminating_state, gamma):
    action_value_vector = Q.predict(new_state)
    new_action_value_vector = Q.predict(state)

    if is_terminating_state:
        new_value = reward
    else:
        value_of_next_state = max(action_value_vector)
        new_value = reward + gamma * value_of_next_state

    new_action_value_vector[action] = new_value
    Q.update_with_gradient_decent(state, new_action_value_vector)
```

We can see that there are clear benefits to deep Q learning. This approach offers a way to approximate functions very well with a relatively small number of weights. It allows your approximated Q function to generalise, give plausible results for unseen states, better than that of regular Q learning. This approach has also been used, very successfully, on some of the most interesting problems to date. Thus, we thought it the best approach for our project.

Unfortunately, as previously stated, Q and deep Q learning are designed to be applied to discrete action spaces. The problem we chose to address in this project has a continuous action space and is, thus, not a typical use case for deep Q learning. We discuss why we stayed with this approach and how we worked around this problem in Chapter 11. We now realise that policy search methods would most likely have been a better approach. This is also discussed in Chapter 11.

6.4 Conclusion

We have learnt that deep Q learning is a value based approach whereby we approximate the Q function using a neural network. We have had a look at the mathematical process of this approach. We have seen that it is suited to problems with high dimensional, discrete, abstract action spaces and is not particularly suited to continuous action spaces. We noted that the problem addressed by this project has a continuous action space. The fact that deep Q learning is not suited to continuous action spaces is addressed and solution to this problem is developed in later chapters.

7 Problem Description

7.1 Introduction

In this chapter we will describe the environment in which we placed our agent. It is important to understand how the environment works when wanting to understand how the agent is to learn to react to it. First we will learn what types of entities the environment is populated with. Then we will learn how a score is allocated. Finally we will learn what the conditions are to win or lose the game.

7.2 Objectives

We gain a full understanding of the environment we have created our agent to interact with. We will discover what entities the environment is made up of, how a score is allocated and how the conditions are to end a episode.

7.3 Content

The environment was first described on the website CodinGame.com. This is a website where programers can solve algorithmic problems, program bots to play various games (some of which compete against other bots) and get in touch with other programmers on the platform.

7.3.1 Code vs Zombies

This environment is called Code vs Zombies (CvZ). The game consists of three different types of entities, namely, a shooter, humans and zombies. Humans are stationary entities. The Shooter is a human that can move a certain distance each round and will automatically kill zombies that are near enough to it. Zombies are entities that continuously move towards the closest human (remember that the shooter is also a human)

and kill any human that is close enough to it. The shooter can move a greater distance than zombies can per round. Figure 7.1 shows a graphical representation of the environment.

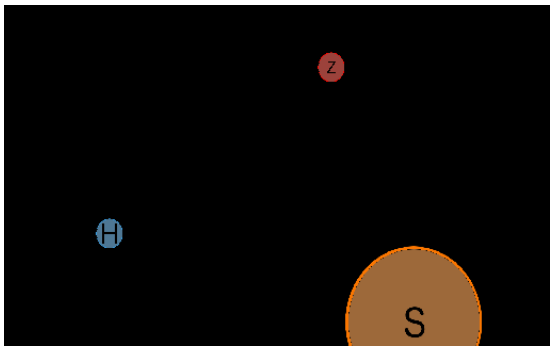


FIGURE 7.1: Graphical representation of the Code vs Zombies environment. If a human (blue circle) enters the zombie's circle (red), it is killed. If a zombie enters the shooter's circle (orange), it is killed.

The agent plays the game by giving the environment the co-ordinates to which the shooter should move towards. The objective of the game is to maximise the game score. The game score is the sum of the score for each round (the round score). The calculation for the round score is shown below. In the equation p_i is the score for round i, h_i is the number of humans alive in round i, z_i is the number of zombies the shooter has killed in round i and $Fibonacci(x)$ is the x^{th} number in the Fibonacci sequence. The score is given by:

$$p_i(h_i, z_i) = 10 \cdot h_i^2 \cdot \sum_{x=1}^{z_i} Fibonacci(2 + x). \tag{7.1}$$

It is clear from 7.1 that the best score is achieved if all zombies are killed in a single round and all humans are alive in this round. If at any point there are no longer any living humans, the game score becomes zero and the environment tells the agent it has lost the game. The environment tells the agent it has won the game when all zombies are dead. The objective is simple: maximise the game score.

7.4 Conclusion

In this chapter we learnt how the environment our agent interacts with works. What type of entities populate the environment, how the environment allocates a score to a game and what the winning and losing conditions are.

8 Environment Setup

8.1 Introduction

This chapter describes the information that is passed between the agent and the environment, specifically the form of the state, the reward and the action. The way this information is structured is very important. It has a very large impact on how the agent must be structured and how effective the agent can be. We will first form a general idea of how the environment is represented. Then we will go into detail on how the state, reward and action values are structured.

8.2 Objectives

We fully explain how the information passed between the environment and the agent looks and why it has been chosen to take this form.

8.3 Content

8.3.1 Code vs Zombies

This environment consists of a two dimensional grid with height and width of 9000 and 16000 units respectively. All entities must have integer co-ordinate values within these ranges. The top left corner is co-ordinate (0, 0) and the bottom right corner is co-ordinate (15999, 8999).

It is common practise, and has been shown to significantly improve solution stability, to normalise inputs to machine learning models. This is the reason for normalised values being used for the state and the reward.

State

The environment may have at most 99 humans and 99 zombies. Thus, the state is represented as a vector of length 398. The first two entries of this vector are the normalised x and y co-ordinate values of the shooter, respectively. The next 198 entries are the normalised x and y co-ordinate value pairs of the humans. The last 198 entries are the normalised x and y co-ordinate value pairs of the zombies. We see where the size of the state vector comes from: $398 = 2 + 2 \cdot 99 + 2 \cdot 99$, where the first term is the two values for the shooter's co-ordinates, the next term is the 198 values for the humans' co-ordinates and the last term is the 198 values for the zombies' co-ordinates.

This leaves us with a state vector that can represent states that consist of the maximum number of entities but we encounter a problem if there are less entities (such as when the shooter kills a zombie or when a zombie kills a human). This problem, and our solution for it, is explained in depth in Chapter 10. Briefly, we made an intelligent decision as to set all co-ordinate values of deceased (or non-existent) entities to -1.

An example of a state encoding that consists of one shooter in the top left corner, one human in the centre of the environment and one zombie at co-ordinate (8999, 8000) is as follows: $[0, 0, 0.5, 0.5, -1, -1, ..., 1, 0.5, -1, -1, ..., -1]$. Note: the zombie's y co-ordinate has been rounded to the first decimal place, this does not happen in practise.

Reward

As discussed previously, creating a reward function is very important. It is enormously difficult to structure rewards in a way that will produce exactly the response you expect, or want, from the agent. We have structured our reward value as follows. The reward for round i, the score achieved for round i and the maximum score that could have been achieved for this game are represented by r_i, p_i and $p_{\max}$, respectively, in the following equation: $r_i = p_i / p_{\max}$. Remember the equation for the score for a given round:

$$p_i(h, z) = 10 \cdot h^2 \cdot \sum_{x=1}^{z} Fibonacci(2 + x), \tag{8.1}$$

where h and z are the number of humans currently alive and the number of zombies killed in the last state transition, respectively. This is done with the intention of giving the agent a way to determine which actions are better than others when both give a reward, while still keeping values normalised as mentioned above.

Penalty

We call negative reward values penalties. The purpose of penalties is to teach the agent what undesirable outcomes are. We use penalties in two ways: to convey to the agent that humans dying is undesirable and that the agent should try end the game sooner rather than later. From the way the game is scored it is obvious to see why we would prevent humans dying. The more live humans, the greater the score. The second penalty is something that is typically done to shorten simulation times and to find the shortest sequence of tasks to complete the goal. In this case it is beneficial to shorten simulation time but the shortest sequence of tasks to complete the goal is not necessarily the optimal one. However, in practice with this environment, better policies are usually discovered when this penalty is in place. We speculate that the reason for this is the shooter realises that the game is ended sooner if it actively seeks out and kills zombies, instead of defending humans. This often leads to more humans being saved and, thus, higher scores.

The penalty for humans dying has been linked to their value. We have defined human value to be the following: $v(h, z, n) = p(h, z) - p(h - n, z)$, where p, h, z and n are the round score function, how many humans and zombies are currently alive and the number of humans in the group of which the worth is to be calculated, respectively. As such, the value of a human is the difference between the score achieved if all zombies were killed in this round with the current number of humans, and with one less human. The penalty is then the value divided by $-p_{\max}$: $-v(h, z, n)/p_{\max}$, this is done to ensure the penalty is also a value between zero and one. This penalty is given for all human deaths except the final one. When the last human dies, the game is ended and the shooter has lost. Thus, this death has the maximum penalty of -1 attached to it.

The second penalty is applied in every round where the reward would be zero otherwise. It is 20% of the current human death penalty. This is to ensure that should the shooter have the option of saving a human and doing another action, it is still incentivised to save the human. If this penalty is bigger than that of a human dying, there would be no incentive to save the human.

Thus, the complete reward function looks as follows:

$$r_i(h,z,n) = \begin{cases} \frac{p(h,z)}{p_{\max}} & z > 0, n = 0 \\ \frac{v(h,z,n)}{p_{\max}} & n > 0, h \neq 0 \\ -1 & h = 0 \\ 0.2 \cdot \frac{v(h,z,1)}{p_{\max}} & else \end{cases},$$

where n is the number of humans that were killed in the last state transition.

Action

The environment expects a vector of length 2 from the agent. The action is the unnormalised x and y co-ordinate values the shooter should move towards.

8.4 Conclusion

We have learnt what the form the information that is to be passed between our agent and environment and why it has been chosen to be so. We have seen that the state is a fixed length vector containing the normalised co-ordinates of entities, the reasons for this decision were briefly discussed and will be discussed on more depth in future chapters. We learnt that the action the agent must pass to the environment is a vector containing the unnormalised co-ordinates that the shooter should move towards. Finally, we saw that the reward that the environment passes to the agent is a scalar value between -1 and 1.

9 Software Engineering

9.1 Introduction

Until now we have been focused on the theory of what we did for this project, now we shift focus to how we developed this project. We outline the programs we created for this project and state which core application program interfaces (APIs) were used. As with any project that includes programming, we needed to be very methodical in our planning and execution. This chapter briefly outlines the software engineering practises we followed in the process of programming this project. We first outline the development method we used, then the version control system, the unit testing procedure and finally dynamic programming and type checking methods.

9.2 Objectives

We go through the basic structure of the code we developed and explain why we used certain tools and APIs. We get a basic idea of the common software engineering practices such as agile development, unit testing, version control, dynamic programming and type checking and why they have been used.

9.3 Content

9.3.1 The Program

The practical aspect of this project consisted of programming a reinforcement learning agent. Figure 9.1 shows an abbreviated universal modelling language (UML) class diagram for the core of our project. We developed an environment for the Code vs Zombies problem, a deep Q learning agent and an interface that bridges these two. As such, one could present our interface with any environment and agent (provided they follow a similar structure) and our interface will facilitate communication between them.

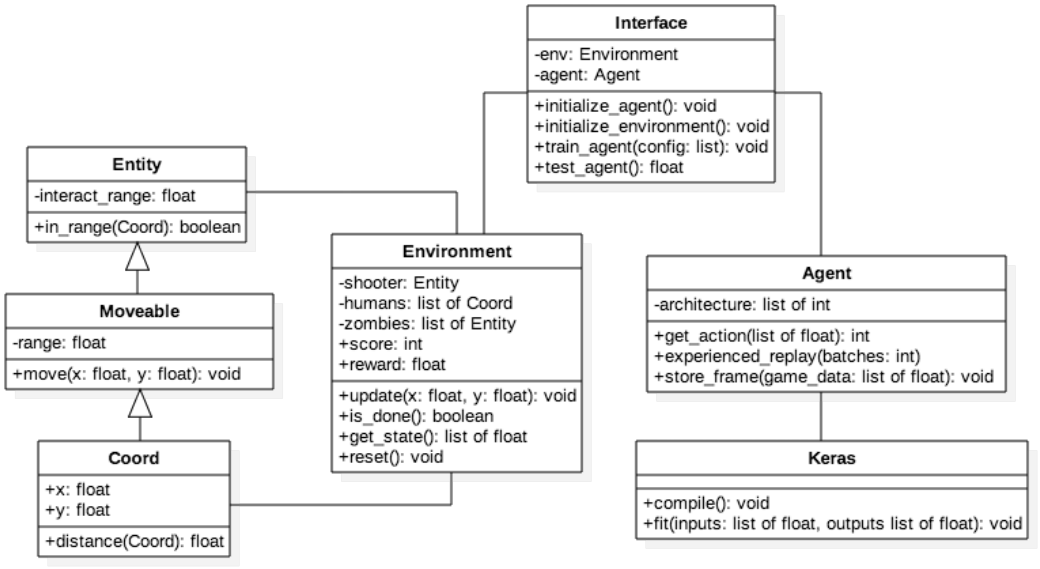


FIGURE 9.1: Abbreviated UML of the core code we developed and how it links to the most important API, Keras. It shows the inheritance structure that is present in the environment setup and the association structure that links the environment and the agent through the Interface class.

The development of all the resources necessary in the development of our program could take a life time. Thus, we used a number of APIs that provided a platform for us to concentrate on the reinforcement learning task. The Keras API is used to facilitate the creation and training of our Q network. The Keras API uses the Tensorflow API as a backend. Tensorflow is a highly optimised machine learning API that automatically parallelises user code. Numpy is a numerical computation API that allows us to perform complicated mathematical operations easily and quickly and was used extensively. A link to our full code can be found in Appendix C.

9.3.2 Development Method

Agile Development

Throughout this project we followed a development method known as agile development. Agile development is a set op principles and practices that allow software teams to develop quickly and respond to change (Martin and Martin, 2006). A good example of an agile programming methodology is test driven development.

Test Driven Development

We used test driven development for a large portion of this project. Test driven development is when one writes tests for a piece of code to pass before the piece of code is written (Astels, 2003). One then writes the code such that it passes all the tests. This helped us develop more quickly, as we had a clear goal of what our code needed to achieve. This approach also kept the quality of our code high as it needed to follow specifications set by the tests. For example, we wrote the unit tests, discussed in the next section, before we wrote the code they are meant to test.

9.3.3 Unit testing

Unit testing is a method by which small blocks (or units) of code tested by running multiple tests where input is fed to this block of code and the output is compared to the output we expect (Sen, Marinov, and Agha, 2005). This unit is only considered to be working if it produces the correct output for all tests. Unit testing also forms part of the test driven development methodology. We wrote extensive unit tests for our code to reduce the number of errors present in our code. This also allowed for faster debugging and it ensured that future code changes do not lead to older code not working as intended. We used the UnitTest Python library, an example of this follows:

Algorithm 6 Unit Test Example

```
def test_distance(self):
    point_a = Coord(x=1, y=1)
    point_b = Coord(x=0, y=1)
    self.assertEqual(1, point_a.distance(point_b))
```

Note: point_a and point_b are co-ordinates, the distance function calculates the distance between the two points and the assertEqual function is a function in the UnitTest library that raises an error if the two arguments are not equal.

9.3.4 Version Control

We used Git version control extensively throughout this project. “Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later” (Chacon and Straub, 2014). Using Git enabled us to backup

our code (and this report) and have easy access to it from any other internet connected computers. Git also allowed us to try different programming approaches in a contained environment.

9.3.5 Dynamic Programming

Dynamic programming is a core element of this project. Dynamic programming is method of solving complex problems by breaking them into smaller problems and storing the results of the smaller problems for future use (Bellman, 2013). A simple example of dynamic programming in our code is the function that calculates the score for the Code vs Zombies environment. The score involves calculating the Fibonacci sequence, which is a classic introductory problem to dynamic programming. Instead of recalculating the sequence every time we need one of the numbers:

Algorithm 7 Slow Fibonacci

```
def slow_fibonacci(n):
    previous_number = 0
    current_number = 1

    for i in range(n-2):
        next_number = previous_number + current_number
        previous_number = current_number
        current_number = next_number

    return current_number
```

We rather store what we have already calculated and carry on where we left of (if we need to):

Algorithm 8 Fast Fibonacci - Using Dynamic Programming

```
fibonacci_seq = [0, 1]

def fast_fibonacci(n):
  if n <= len(fibonacci_seq):
    return fibonacci_seq[n-1]

  for i in range(n - len(fibonacci_seq)):
    fibonacci_seq.append(fibonacci_seq[-1] + fibonacci_seq[-2])

  return fibonacci_seq[-1]
```

This greatly improves simulation speeds as we do not waste time recalculating values. Note: there are corner cases that would lead the above functions not to work as intended, for brevity and clarity we chose not to cater to those cases in this example.

9.3.6 Type Checking

The vast majority of the code we wrote was written in Python. Since python is a dynamically-typed language (meaning that variable types are assigned at execution time and can be changed), we must be careful to ensure that we know which types will result in proper execution of our functions. We greatly reduced the number of errors in our code by whitelisting certain types to be passed in our functions and raising errors should a type be received that is not on the whitelist. A simple example of this follows:

Algorithm 9 Type Checking Example

```
def func(x):
  if not isinstance(x, (int, float)):
    raise TypeError("x must be an integer or a float")
```

9.4 Conclusion

We went through the core of our program and the main software engineering practices what we used throughout the development of this project, namely: agile development,

unit testing, version control, dynamic programming and type checking. These all reduced the number of errors present in our code, optimised our development process and overall attributed to a better project.

10 Implementation Issues

10.1 Introduction

This chapter aims to outline the implementation difficulties that were encountered throughout the development of this project and how they were overcome. This is especially important to do as it reinforces this information in our minds. This chapter also serves as a record for others, should they meet the same difficulties. We will also discuss any optimisations we made throughout the project but we leave the testing, results and analysis thereof for the following chapter.

10.2 Objectives

We will explain why deep Q learning is not suited to the problem described in Chapters 7 and 8 and how we overcame this. We then discus complications that the environment introduced and how overcame them. Finally, we will gain an understanding of the optimisations that were made on the agent.

10.3 Content

10.3.1 Challenges

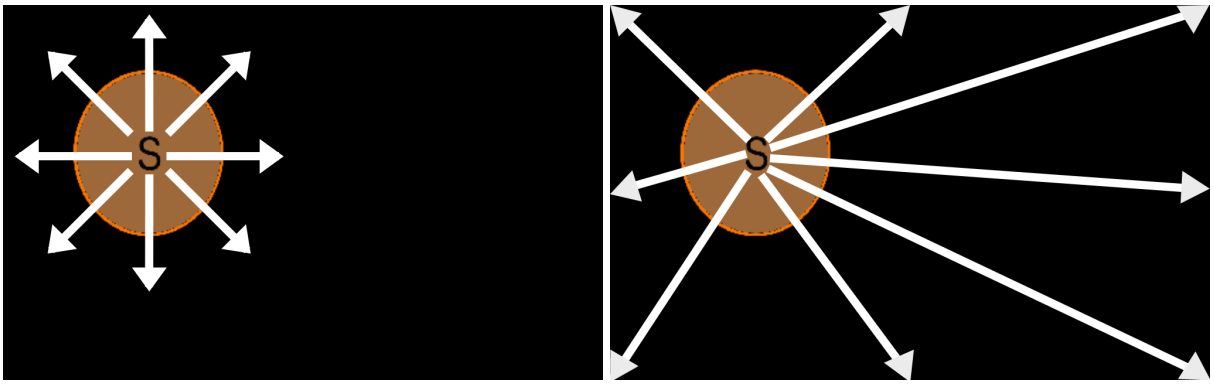
Continuous Versus Discrete Action Spaces

The first, and perhaps, the largest issue we were faced with is the fact that we decided to use deep Q learning for this project. We started our research on reinforcement learning methods by identifying those currently used to solve the most interesting problems. We quickly found deep Q learning and thought it a good approach. We developed an understanding of it from reading various papers and implementing it on simple environments. Our implementation performed well and we understood the

approach from the perspective of using it on this simple environment. However, the problem this report sought to address and the simple environment had a very important difference that we failed to detect. As previously discussed, deep Q learning is designed for use on environments with discrete action spaces. The simple environment had a discrete action space and the Code vs Zombies problem has a continuous action space.

We felt it was too late to change the problem and the approach, thus, we decided to devise some solution. We decided that the easiest solution would be to discretise the action space that our agent sees. There are numerous ways one could do this. One could have the agents actions defined as moving a fixed distance in one of the cardinal directions (north, east, south or west). One could expand this to include the inter-cardinal (north east, north west, etc.). One could take both of the previously mentioned methods and double the set of actions by allowing for two fixed distances the agent could choose to move in. The list of possibilities goes on.

We decided to define the agent's actions as moving a set distance towards predetermined and fixed points in the environment. For example: action 0 would be to move a fixed distance towards the point (0, 0). Our first thought was to use the points (0, 4500), (8000, 0), (8000, 8999) and (15999, 4500), the points that would point in the cardinal directions if the agent is placed in the centre of the environment, (8000, 4500). However, this setup makes it impossible for the agent to reach every point in the environment. For instance, the corners, (0, 0), (0, 8999), (15999, 0) and (15999, 8999), cannot be reached. To solve this problem, we select the corners to be our points to which the agent will move towards. In Figure 10.1 we see two possible action sets: the cardinal method and an extended version of the fixed point system described in this paragraph.



(A) Cardinal Direction Method (B) Fixed Point Method

FIGURE 10.1: Two of the possible action sets. Figure 10.1a shows the direction of the possible actions for the cardinal method and Figure 10.1b shows the direction of the possible actions for the fixed point method.

We feel that this method is better than the previously mentioned cardinal direction method as it allows full exploration on the environment with only four actions and in general allows for shorter paths between points in the environment. Furthermore, by the way the environment is setup, is more computationally efficient to chose fixed points to travel towards instead of calculating ones based on the agent's current position.

This setup successfully discretises the action space and allows us to use deep Q learning for this problem. The fact that we have done this will undoubtably have an adverse affect on the performance of the agent. As such, if one used an approach that is designed for continuous action spaces, we would expect a fairly large increase in performance.

State encoding

The second difficulty we were presented with is the fact that we must give a fixed size state encoding to the neural network. As previously stated, our state encoding is the co-ordinate values of the shooter, the humans and the zombies. However, as time progresses, both humans and zombies die and, thus, we no longer have co-ordinates for them. This means that the actual state representation reduces in size. However, we just noted that we must present a fixed length state encoding to our neural network.

We decided to solve this in the following way: every human and zombie is allocated an ID number, this ID corresponds to their position in the state encoding. While the human or zombie is alive, their normalised co-ordinate values appear at the location

corresponding to their ID. When they die each of their co-ordinates are replaced with -1. For example, the state encoding s_0 depicts a state that contains one shooter, one human and one zombie, and the state encoding s_1 depicts a state where the zombie has died:

$$s_0 = [\ 0.25 \quad 0.2 \quad 0.1 \quad 0.1 \quad 0.3 \quad 0.25\ ]$$
$$s_1 = [\ 0.25 \quad 0.2 \quad 0.1 \quad 0.1 \quad -1 \quad -1\ ].$$

This solves the issue and gives us a fixed length state encoding. This does imply that the maximum number of humans and zombies that can exist at once must be defined before the network is trained. This size encoding must be used throughout training and when the agent is deployed to complete the task. We replace the co-ordinates with -1 because this is the normalised point farthest away from all other normalised co-ordinates (recall that our environment only consist of positive normalised co-ordinates). The hope is that the agent either realises that -1 means that the human or zombie is dead, or the agent thinks that this entity is far enough away that it need not be paid attention to. We asked international acclaimed industry professionals what they would have done in this situation and they confirmed that this was the approach they would have followed.

The downside of this solution is that our state encoding is usually quite informationally sparse. This makes it more difficult for our agent to learn good policies and increases the time it takes to for propagation through our neural network as the input layer must be the same size as the state encoding. In Chapter 11 we critically assess this decision.

Large Simulation Memory Leaks

Originally we used a single Python script to train many networks to find architectures and learning rates that provided good performance. However, we quickly noticed that the models were taking much longer to train than expected and that the memory being used by the script was far more than was to be expected. We guessed that a memory leak was taking place and noted that if the Python interpreter is exited regularly, these problems vanish. Thus, we changed the Python script to accept command line arguments that determine the network architecture. We then wrote a bash script to initiate

the Python script with the necessary command line arguments required to run all training sessions. Since the Python script was being called from the bash script, the Python interpreter would terminate and restart between each call and this issue was resolved. The original approach did not finish all training sessions when given a whole week. The fixed version finished all training sessions in just over one day.

Bugs in Code

Despite good software engineering practices, errors are present in our code. A particularly devastating error had to do with the way the previously mentioned zombie IDs mapped to the state encoding. The error resulted in the zombies usually not being mapped to the state encoding at all. This bug went unnoticed for weeks. In the mean time we were already getting fairly promising results from our agent that was able to learn optimal policies on lower dimensional encodings in roughly ten minutes. Once this bug was noticed and fixed, the agent was able to learn better policies on the same encodings in less than two minutes. Our agent was essentially blind to zombies for much of the project. We are quite pleasantly surprised that we were able to achieve some decent results with the blind version. However, we are far more pleased with the results of the corrected version.

10.3.2 Optimisations

Throughout the development of the project we pushed to decrease agent training time. Thus, we made a few significant optimisations, to the common practices in the field. They are briefly described below and are analysed in more depth in the next chapter.

Reward Normalisation

We discuss experiments in which we assess the performance of our reward scheme as laid out in Chapter 8. However, we noted that following the same scheme without the normalising terms lead to the agent taking longer to converge to the optimal policy or never converged to the optimal policy. This realisation lead us to develop the theory that reward values should be normalised.

Environment Data Carry Over

Originally we designed our environment to compute the Fibonacci numbers, needed to compute the score, at the time the environment is initialised. However, since the maximum number of entities in the environment stays constant over all simulations we realised that we need not recalculate this every time we initialise an environment. Thus, we changed the environment to carry over the Fibonacci information when it is reinitialised. This meant that the Fibonacci information would only be calculated once per training session. This lead to a large reduction in training time.

10.3.3 Environment Generation

Originally we generated fixed numbers of humans and zombies to form the initial state of our episodes. This resulted in our agent not generalising well. For instance: as the level progresses more humans die than zombies (especially at the beginning of training), thus, our agent will learn on these scenarios. For agents trained this way we observed the agent acting erratically when the number of humans became greater than the number of zombies. To solve this we randomly generated the number of humans and zombies for initial states.

10.3.4 Fixed Initial States

We observed that if we always provide the same initial state to our agent in training, it learns the optimal policy for this configuration very quickly. This setup is similar to the problem described in Human-level control through deep reinforcement learning, in Chapter 2 as many Atari 2600 games always start in a particular state. If we constrain our Code vs Zombies problem to do this we get considerably better results. However, we lacked the time to adequately explore this approach.

Frame Skip

To allow for faster training and simulation, we use the frame-skipping technique described in Chapter 2.

State Rolling

Because it is possible to determine exactly what will happening the following game state if given the current game state, we did not employ state rolling. However, functionality for it is integrated into our code.

10.4 Conclusion

During the development of this project, we encountered some implementation issues, such as needing a fixed size state encoding and the fact that deep Q learning is not suited to the Code vs Zombies problem. We discussed our solutions to these and other implementation issues. We also made optimisations that improved performance of our agent and reduced the time it needed to train.

11 Experimental Investigation

11.1 Introduction

In this chapter we will test what we have discussed in previous chapters on the problem described by Chapters 7 and 8. We will explain what we are testing, why it should be tested and how it will be tested. We will then view and discuss the results of the test. This chapter is arguably the most important chapter in this project report as it confirms (or denies) what we have reasoned about throughout this report. We start by explaining how our tests were performed and move onto specific tests related to finding a good network architecture, the training of our agent, the actions our agent can perform, how the scale of the Code vs Zombies problem affects performance and then we present and test our final model.

11.2 Objectives

We will use evidence to show: that network update delay outperforms target network update delay, that full replay sampling outperforms random replay sampling, the default action system performs better than all others and our reward scheme is more effective than others, for our given problem.

11.3 Content

11.3.1 Validation

During training, we periodically stopped the agent and validated it's current policy. This is a tricky process to do properly, it is subject to high variance and is time consuming to do accurately. With this in mind, we designed a validation test that would rather be less accurate (subject to higher variance) but would be relatively quick to complete.

We did not want to increase the time taken up by the training process. Thus, we performed, what we call, light weight validation during training and after the model was trained we performed validation that was more accurate and time consuming.

Light weight validation consisted of taking the agent's average score over 100 episodes of randomly generated initial states. The more accurate validation is the same but over 1000 episodes.

When selecting which model architectures to focus on, we looked at validation scores (to see how well the model can perform) and validation scores divided by the time it took to train the agent (to see how quickly the model learns). Naturally, we look for a good balance between pure performance and performance per time interval.

For most of the experiments done below, we used a constrained version of our environment. The idea being that we need to try solve a small version of the problem at hand before we can move onto the actual problem, a dynamic programming principle. We used environments with a maximum of three humans and three zombies (and one shooter - or agent). We also reduced the state encoding size to fit only these entities. Thus, the state encodings where vectors of $2 + 2 * 3 + 2 * 3 = 14$ in length, apposed to the full problem which uses state encoding vectors of length $2 + 2 * 99 + 2 * 99 = 398$. We decided to use this constrained version of the problem as a prototype as it contains the complex interactions with the scoring of the game while being small enough to allow for more simulations to be run.

Each of the below experiments where repeated between three and twenty times, depending on computational cost of the test, and figures are averaged over all data for the give test. Thus, we can claim that the results are reproducible. However, even a set of twenty tests does not form enough data to make a definitive claim as to whether these results are truly representative. This must be taken into consideration when reading through the rest of this chapter.

Note: the curves of some graphics have been smoothed to allow us to analyse the underlying information more easily and accurately.

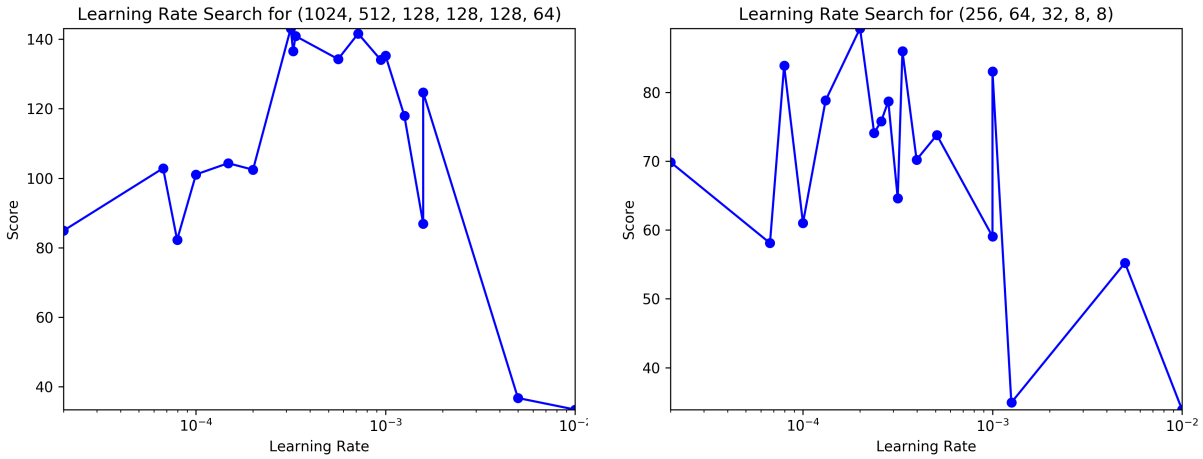
11.3.2 Network Architectures

Perhaps the most common question is machine learning is: "how big should the model be?". This is not an easy question to answer. The affect of network architecture on model performance is somewhat of a mystery and findings do not necessarily generalise to other cases. Thus, we pseudo-randomly generated network architectures and

tested them on our problem to see if patterns could be found. By pseudo-randomly generated, we mean that we used insights we have gained from practical experience with neural networks to ensure that we generated plausible candidates. The largest insight used being that the layers generally decrease in size the deeper into the network they are.

The process was as follows: generate a random number, l, to represent the number of hidden layers, sample l random numbers from a set of powers of two and sort these numbers in descending order. We sample from a set of powers of two because generally this has the result of our layers exponentially reducing in size, allowing us to have deeper networks. This sequence of numbers then represents the number of neurons in each hidden layer. We train networks with these architectures via a logarithmic scan over learning rates to find one that performs well for each architecture. We label networks by the sequence of hidden layer sizes.

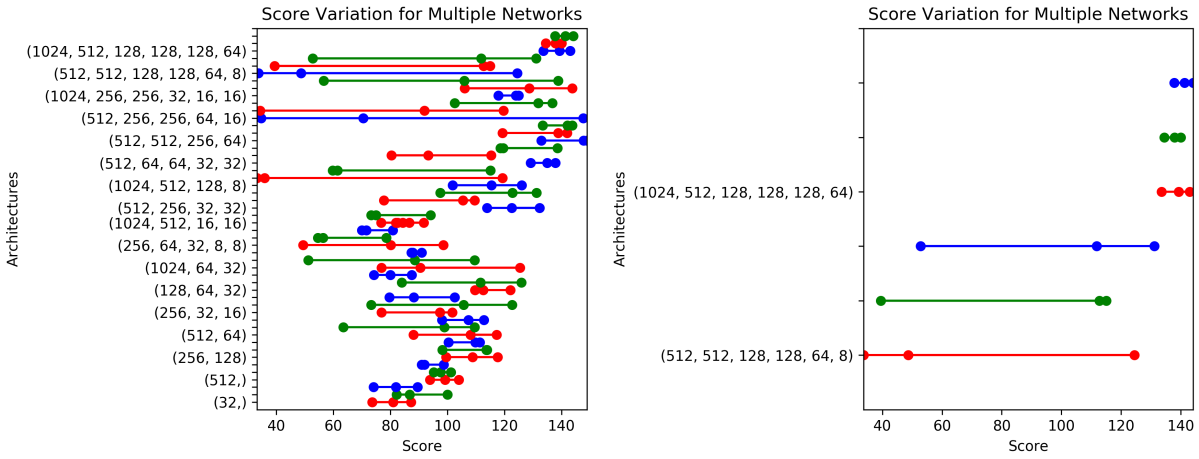
We require our networks to perform well and do so consistently. Figure 11.1 shows us that the learning rate of a network has a great deal of influence over the performance of the network. We also see that some architectures seem inherently unstable. Figure 11.1b shows drastic changes in score from very small changes in the learning rate. Thus, this architecture is too inconsistent and we would not select it for our model. On the other hand, Figure 11.1a shows consistently good performance for a large range of learning rates. The network associated with Figure 11.1a is a candidate for our model going forward.



(A) Stable Network (B) Unstable Network

FIGURE 11.1: We see two examples of logarithmic learning rate scans on different networks. We see Figure 11.1a is stable as there is a cluster of learning rates that provide similar score values. Figure 11.1b is unstable as there is very large score variation for a small learning rate change.

We performed this test for seventeen different architectures. We then selected the three learning rates that performed best for each network and retrained our agent with each of them. We did this to ensure that we only continued our investigation with architectures that perform consistently well. In Figure 11.2 we see final scores for different networks and their respective top three learning rates (each learning rate is a different colour). For instance, we rather select (1024, 512, 128, 128, 128, 64) as it performs better and is more consistent than (512, 512, 128, 128, 64, 8) (Figure 11.2b).



(A) All Networks (B) Example Networks

FIGURE 11.2: The variation in score between networks for their top learning rates. Each colour represents a different learning rate for each network. We see all networks and their variation in score, this is used to select the top performing and most stable network-learning-rate pairs (11.2a).

Table 11.1 shows the networks and their respective learning rates we selected to continue our investigation with.

TABLE 11.1: Top Performing Network Architectures and Corresponding Learning Rates

Architecture	**Learning Rate**
(512,)	0.0012574334296829354
(256, 128)	0.0012574334296829354
(256, 32, 16)	0.0012574334296829354
(512, 512, 256, 64)	0.001
(512, 64, 64, 32, 32)	0.001
(1024, 512, 128, 128, 128, 64)	0.0003353628856001657

11.3.3 Reinforcement Learning Techniques

Initially we tried to address the Code vs Zombies problem with Q learning. However, it was simply not feasible. The agent found itself in states it had never seen before, and was unable to generalise from seen states, far too often. Consequently, we quickly moved away from this method without gathering any performance results.

Moving to deep Q learning showed promising results and allowed us to do experimental analysis of many other aspects within the approach. The first of which we analyse is how the agent is trained.

11.3.4 Training Methods

The way we train our model has a large impact on its performance. Thus, we try find the best way of doing so. Training methods can broadly be broken up into two classes, online and offline learning. Online learning is when we update our model weights after every state, with this state information. Offline learning is when we gather state information over a number of episodes and update our model periodically with the gathered data. Online learning was not feasible for our problem as updating the model at every time step is computationally expensive and time consuming. We compared methods that somewhat bridge the gap between offline and online learning, such as experienced replay.

Experienced Replay

Experienced replay is the process whereby we gather state information and update our model on some of the gathered data at every time step (Lin, 1993). Typically we limit the number of data points we gather. This is because of memory limitations and because older state information may not be as valuable as the newer state information (as our agent is constantly improving it's policy and moving towards more valuable states). We call the store of the state information the replay memory. This is an effective method and produces decent results. However, experienced replay is time consuming and models tend to get stuck in local optima when using this method, a common problem with online learning methods.

Target Network Update Delay

As previously discussed, the paper Human-level control through deep reinforcement learning, describes a method we call target network update delay. This method is closer to offline learning as we only update our model after multiple episodes. In this approach we keep two separate networks, a target and a prediction network. The prediction network is the model that the agent uses to predict how valuable a state is and choose actions. Every time step the target network is then updated with the

information gathered by using the prediction network. As we are updating the target network, not the prediction network, we are not changing the policy of the agent after every update. Only after a number of updates to the target network, do we copy the weights from the target network over to the prediction network and, thus, update the policy. This prevents models from getting stuck in local optima and increases solution stability. We call it target network update delay as a target network is used to delay the updating of the prediction network.

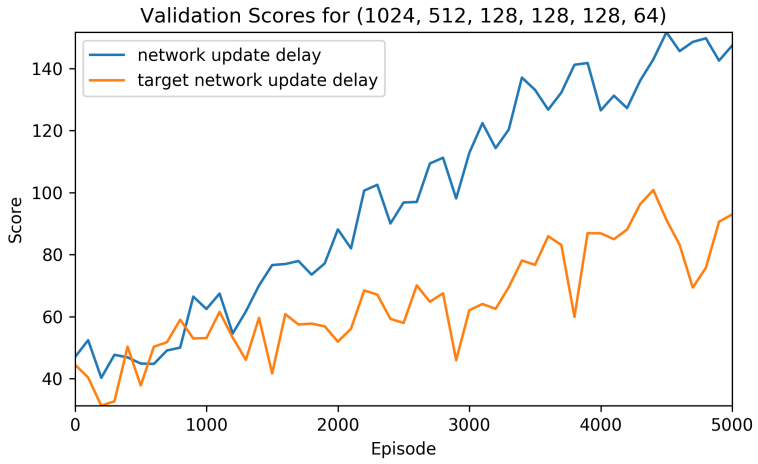
Network Update Delay

Target network update delay produces decent results. However, we felt that there were optimisations that could be made. Target network update delay requires one to use two separate networks, meaning that one must store both sets of weights in memory at the same time and one must periodically take time to copy the weights from one network to the other. We decided to adapt target network update delay to use only one network. We call our method network update delay.

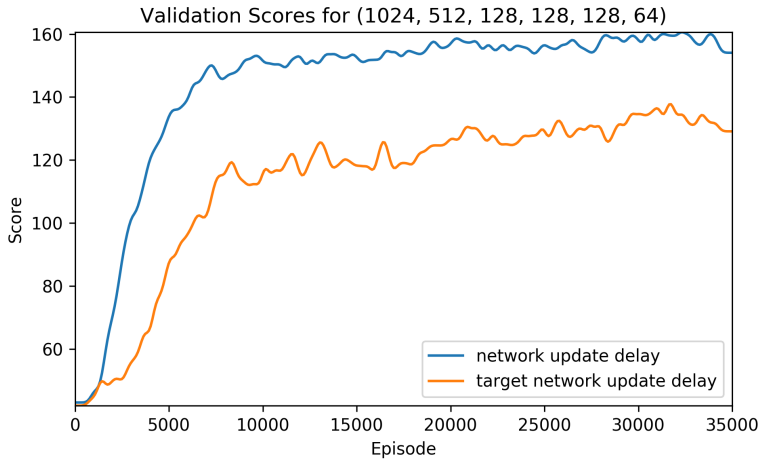
In our approach we store the state information, gathered using our single network, in replay memory over multiple episodes. After a number of episodes (see Appendix D) we then train our single network using the replay memory. This deceptively simple method gives us all the benefits of target network update delay while using approximately half the memory and saving computation time by not having to copy weights from one network to another.

To compensate for the fewer number of updates to our network, we train using more information from our replay memory. Training with more information at once can often lead to more effective and faster training. Many computers are "starved" of data and have more potential to parallelise the training, effectively taking the same amount of time to train on more data.

From comparing the validation scores over a training session, we see that network update delay seems to allow the model to learn more quickly and achieve a higher end score than target network update delay (Figure 11.3). This suggests that our approach may be better but we cannot make this claim yet.



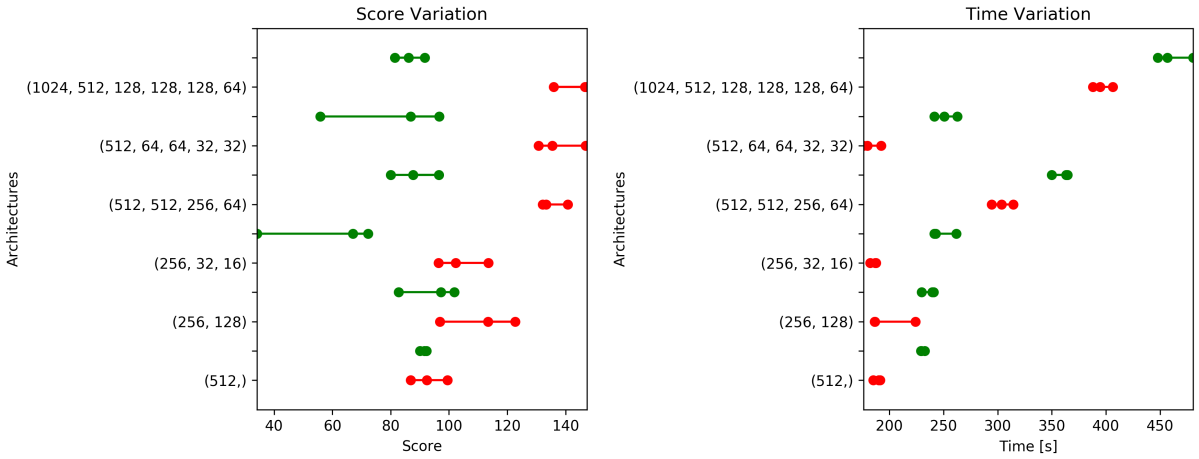
(A) Validation score over 5000 episodes



(B) Validation score over 35000 episodes

FIGURE 11.3: Validation scores being tracked over time during training for the two training methods. We see that network update delay seems to promote faster learning and deliver better performance.

Our claim becomes much stronger when comparing the two update methods for multiple network architectures (Figure 11.4) For every network we tested, our approach achieved a higher end score and trained in less time. We also see that the performance is somewhat more consistent, as stated previously, this is a desirable trait.



(A) Scores (B) Training Times

FIGURE 11.4: The variation in score (11.4a) and time (11.4b) when comparing the network update delay method (in red) to the target network update delay method (in green). We see that network update delay seems to achieve far better scores in substantially less time.

Naturally, many more tests must be done and other domains need to be used to assess whether our approach is truly better. However, for our problem, we can be reasonably certain that network update delay outperforms target network update delay.

Random Replay Memory Sampling

We call the data actually used to train the model the actual replay memory. Typically one samples the replay memory with a uniform random chance to form the actual replay memory and then updates the network with the actual replay memory. By an informal inspection, we noticed that the random sampling of the replay memory took up a large portion of the training time. Thus, we thought of seeking ways of forming the actual replay memory that were less computationally expensive.

Full Replay Memory Sampling

Our first thought was simply to reduce the size of the replay memory and to take the replay memory and use it as the actual replay memory, thereby training on all of the data in it. We call this approach full replay memory sampling.

Our approach seems to slightly out perform random replay sampling (Figure 11.5).

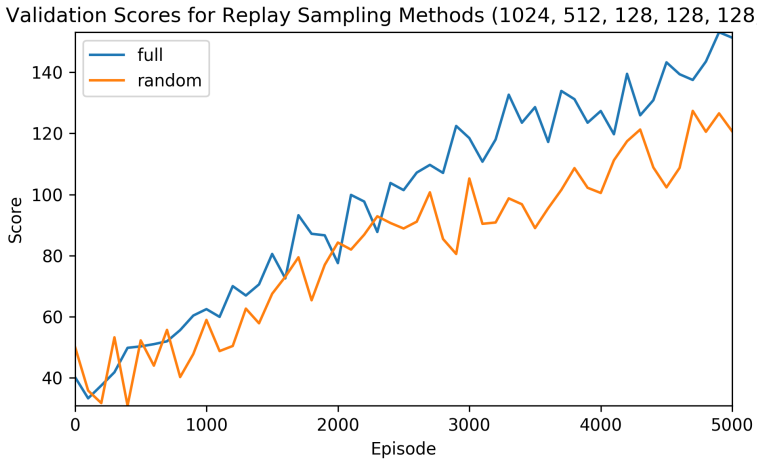
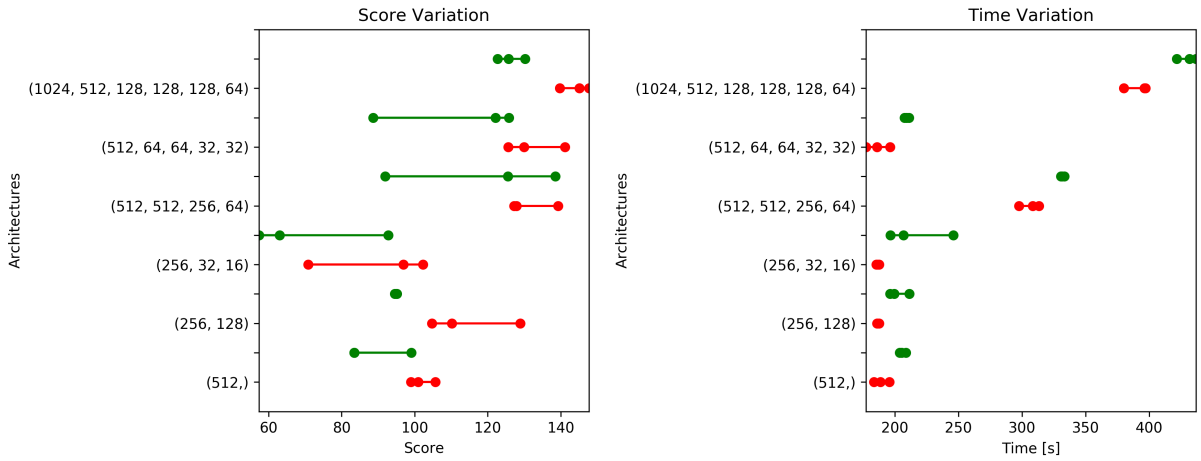


FIGURE 11.5: Validation scores being tracked over time during training for the two sampling methods. We see that full replay sampling seems to promote slightly faster learning and deliver better performance.

The benefits of full replay sampling over random replay sampling boast lower variation in scores and training times as well as consistently achieving better scores with less time spent training.



(A) Scores (B) Training Times

FIGURE 11.6: The variation in score (11.6a) and time (11.6b) when comparing the full replay sampling method (in red) to the random replay sampling method (in green). We see that network update delay seems to achieve better scores in less time.

There is strong evidence that full replay sampling is better than random replay sampling for our problem. More research would need to be done to determine if our approach is better in other domains.

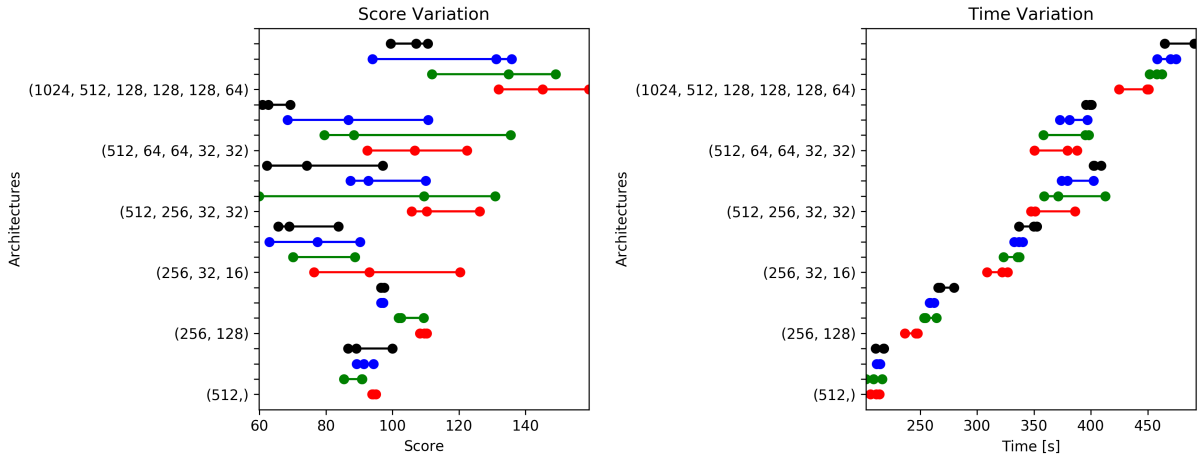
Note: we ensured the actual replay memory was the same size for both approaches, thus, we increased the replay memory size for the random replay memory sampling approach test.

11.3.5 Action Systems

Now we come to the section that tells us if we discretised our action space in an intelligent way. As discussed previously, deep Q learning is designed for discrete actions spaces. Thus, we gave our agent a discrete representation of the action space in the form of four actions, moving towards each of the corners of the rectangular environment space. We refer to this as the default action system. This seems like a very important decision to make and one that would impact the overall performance of the agent.

To know if this was a good decision or not we tested it against three other approaches. The first alternative approach is the same as the default except for one thing, we give the agent the option of remaining stationary. We refer to this as the default static action system. The idea behind this decision being that there are scenarios where it is the optimal action not to move and our agent was otherwise incapable of performing this action. The second alternative method was to give the agent eight points it could move towards, the four corners and the points in between adjacent corners. We refer to this approach as the larger action system. The thought behind this method being that the agent now has far more control in which direction it can move, allowing it to choose shorter paths between targets. The final approach we tried was the same as the larger action system but we added the action to be stationary. We call this the larger static action system.

In general, we see that the default action system offers the best scores and does so in the least training time. It is interesting to note that the larger static action system seems to offer the most consistent performance. Training times and scores seem to vary much less when using the larger static action system. While this is desirable, the default action system seems to offer a large chance that the lowest score achieved will be higher than the highest score achieved by the larger static action system (Figure 11.7).



(A) Scores (B) Training Times

FIGURE 11.7: The variation in score (11.7a) and time (11.7b) when comparing the default (in red) default static (in green), larger (blue) and larger static (black) methods. We see that the default action system seems to out perform all other methods in terms of score and training time.

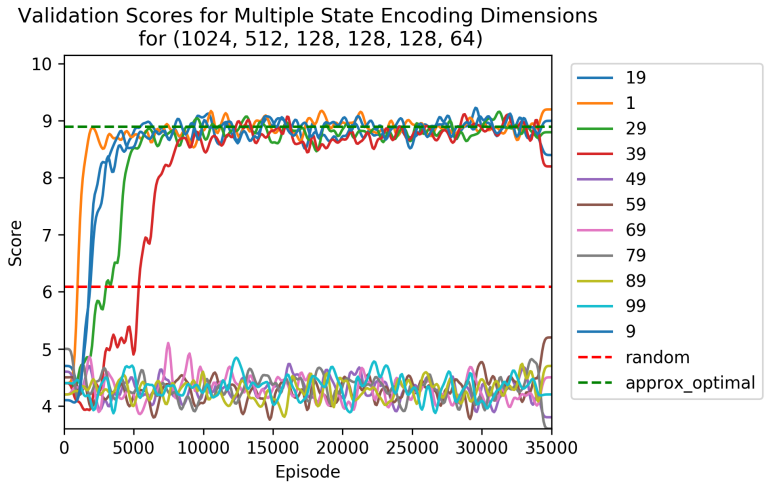
These results indicate that we discretised our action space in an intelligent manner. This is reassuring as it shows that deep Q learning may not have been as ill suited to this problem as we once thought.

11.3.6 State Encoding Size

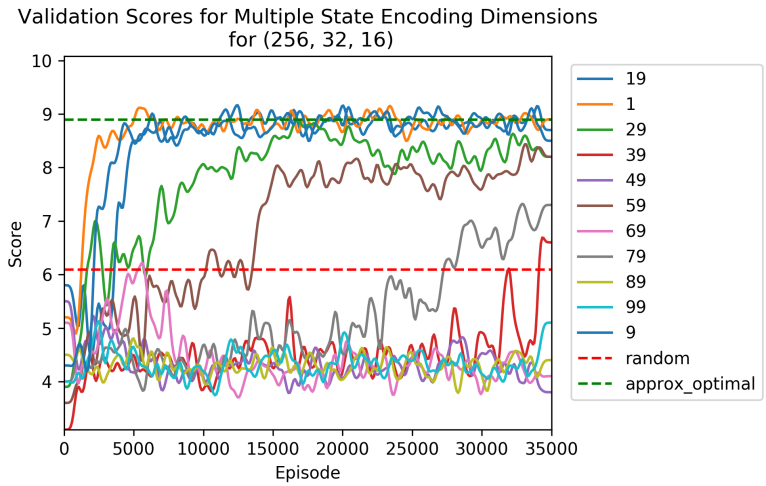
A large concern throughout the development of this project was weather the agent would be able to solve the problem when given the full state encoding. We saw from previous tests that the agent performs quite well when we give it smaller state representations but we wondered how the size of these encodings affected performance. Thus, we set up tests where linearly increased the size of the state encodings and observed the performance of the agent. These tests become very computationally expensive, thus, we only generated environments containing one human and one zombie (and a shooter - or agent). We label the sate encoding as 99, for example, if there are 99 places for humans and 99 places for zombies in the state encoding.

We see that larger networks either achieve optimal performance or perform very poorly. The network that achieved the highest number of optimal performances on different encoding sizes was (1024, 512, 128, 128, 128, 64), with optimal performance up to state encoding 39 (Figure 11.8a). The smaller networks were able to achieve

performance that was above random and below optimal. The network that achieved above random score on the most different state encodings was (256, 32, 16), with above random performance up to state encoding 79. However, we see that network (256, 32, 16) was not able to achieve score above random level for state encoding 69 (Figure 11.8b).



(A) Most tests at optimal



(B) Most tests above random

FIGURE 11.8: Validation scores for different state encoding sizes. We see that none of the networks were able to achieve an optimal score on the full state encoding size, 99. However, the smaller networks seem to have a higher chance of doing so.

This test does not bode well. It seems that the networks are unable to detect single co-ordinate pairs as the size of the state encoding grows. This means that our agent

will not be able to make use of all the state information it is presented with and will not be able to solve the problem at full scale.

11.3.7 Architecture Selection

We now take all we have learnt from the previous experiments and attempt to determine which of the architectures we have been testing is the best for our problem. To do so we compare the architectures with each other and with two other agents. We compare with an agent that performs random actions and an agent that uses a predetermined heuristic. These two serve as our random and approximate optimal threshold bands. Ideally we would like our agent to perform on the level of, or better, than the approximate optimal agent. If an architecture does not perform better than random it must be immediately discarded and our approach may need to be reconsidered.

When looking at the smallest version of our problem (the case where there is one human, one zombie and state encodings are of length 6) we see that the larger networks initially learn much faster than the smaller ones. However, almost all networks converge to the approximate optimal policy at the same point (Figure 11.9). This makes it difficult to definitively say which architecture is best. We must move onto testing larger versions of our problem.

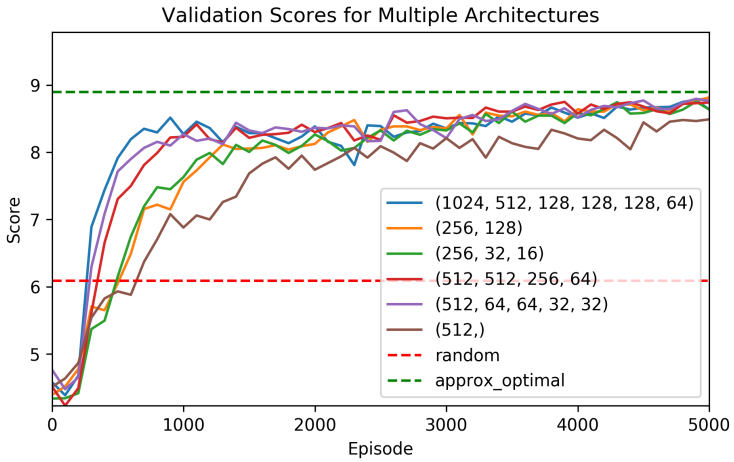


FIGURE 11.9: Validation scores being tracked over time during training for multiple architectures for the smallest version of our problem. All networks converge to approximate optimal performance relatively quickly.

In the validation sized version of our problem we see that the performance differences between the various architectures become clear. Networks (1024, 512, 128, 128,

128, 64) and (512, 512, 256, 64) perform similarly and clearly outperform the rest of the networks. The smaller networks only just do better than average, which is a large concern. Another concern is that non of the networks comes near to the approximate optimal performance. Before discarding these networks, it is important to note that it seems that near the end of training all networks were still climbing in performance (Figure 11.10). Thus, we must try the same test with more training episodes.

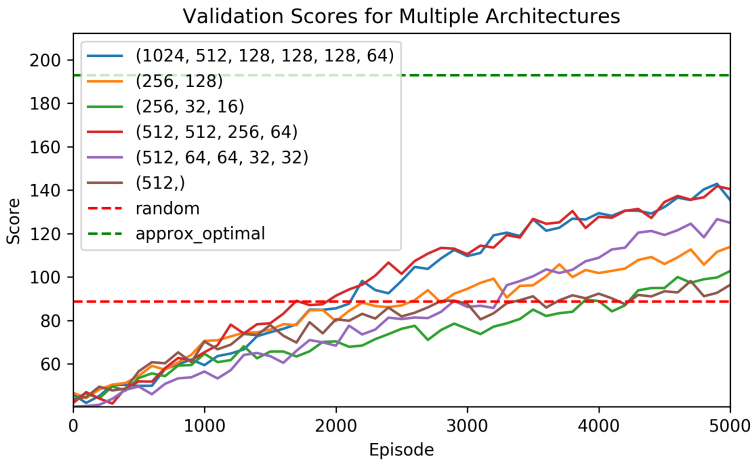


FIGURE 11.10: Validation scores being tracked over time during training for multiple architectures for our problem at validation size. We see that no networks converge to the approximate optimal score. However, the larger networks certainly outperform the smaller ones.

We see a large contrast in results when increasing the number of training episodes. All networks now do substantially better than the random agent. However, the smaller networks seem to stabilise on a suboptimal score. It seems that networks (1024, 512, 128, 128, 128, 64) and (512, 512, 256, 64) will converge to the approximate optimal score if given enough training data. However, learning becomes very slow after the the thousandth episode (Figure 11.11). From this we can assume that the next course of action would be to try a larger architecture. From what we have seen in the past, the larger architectures learn quicker, in terms of number of episodes, and achieve higher final scores. Before we can confidently move onto larger networks, we have to take more information into account.

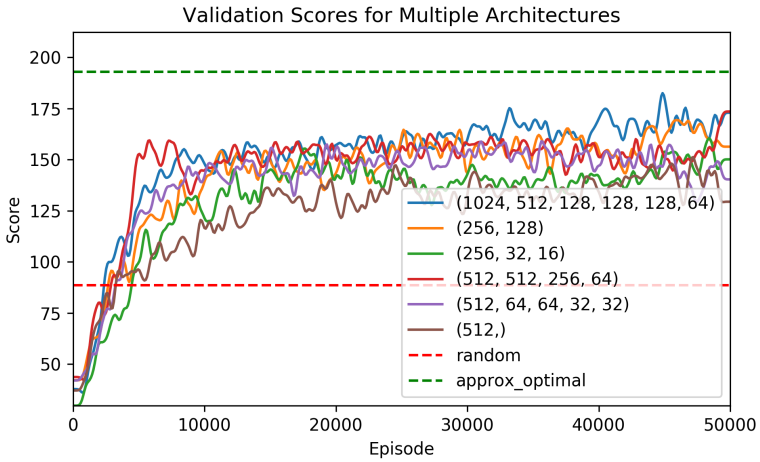
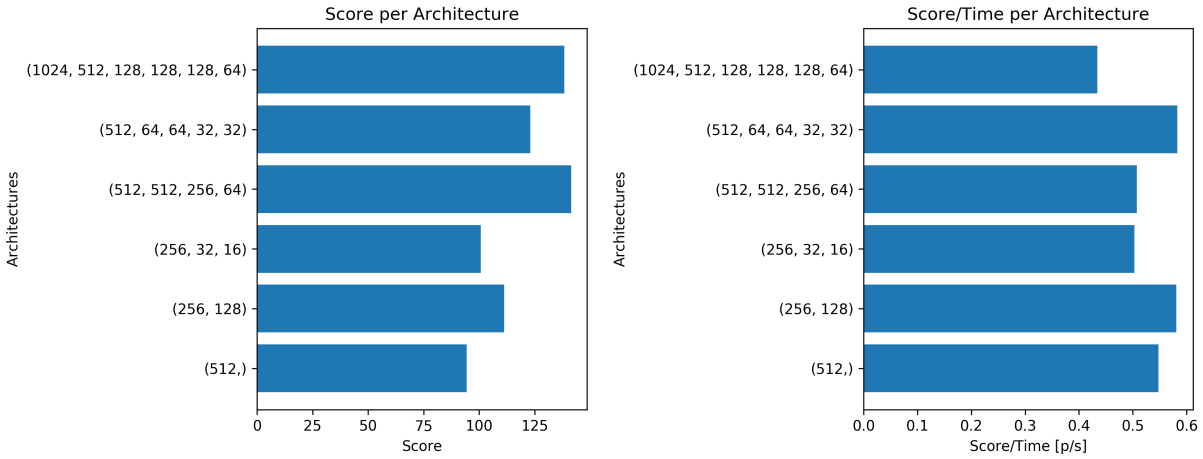


FIGURE 11.11: Validation scores being tracked over time during training for multiple architectures for our problem at validation size. We see that given more training episodes increases the likelihood of networks achieving approximate optimal performance.

We need to take maximum score and time taken to train into account. Thus, we look at score divided by time to get a score per second value. We see that the (512, 512, 256, 64) network outperforms the (1024, 512, 128, 128, 128, 64) network in terms of maximum score reached and score per second return (Figure 11.12). This indicates that the process of choosing an architecture is not as simple as "bigger is better".



(A) Scores (B) Score Divided By Training Time

FIGURE 11.12: The maximum scores achieved by (Figure 11.12a) and the maximum score divided by the time taken to train (Figure 11.12b) each network. We see that larger networks reach higher maximum scores but have lower score per second return.

It seems that the larger the network, the more information we must give it before its validation score stabilises, its score usually stabilises at a higher value but it takes longer to train. Not only that, some networks that are reasonably smaller perform better than the bigger ones.

11.3.8 Reward Systems

Before finalising our architecture we wanted to conduct one final test. We wanted to ensure we have selected a good reward scheme. A reward scheme that promotes quick learning. Until now we had used the reward scheme defined in Chapter 8. For this test we decided to use the scheme described in Chapter 2. Thus, we gave rewards only consisting of +1 (for a round where a zombie is killed), -1 (for a round when a human is killed) and zero otherwise. We call this the simple reward system.

We see that all networks trained using this scheme performed exceptionally poorly (Figure 11.13). This reward system does not convey the nuances of the problem and, thus, the agent is unable to learn a good policy from it.

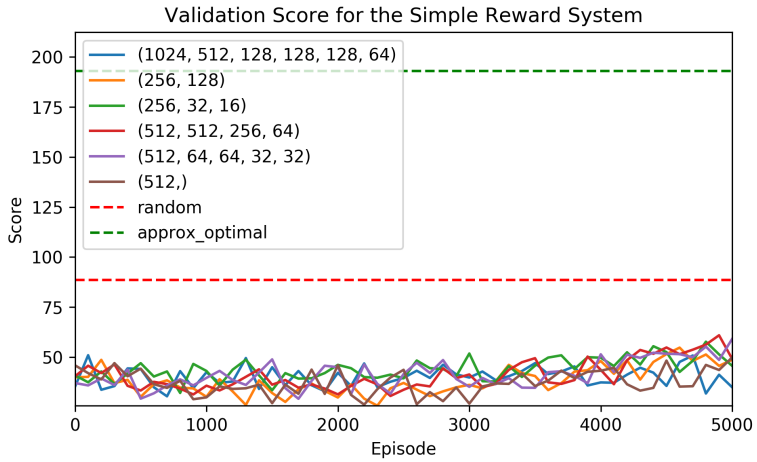


FIGURE 11.13: Validation scores being tracked over time during training for the simple reward system. We see that our agent performs very poorly.

This reinforces the idea that reward functions are extremely important. Our agent's learning ability is only as good as the information conveyed to it.

11.3.9 Chosen Architecture and Techniques Results

We now take all techniques from our previous tests that outperformed the others and apply them to see how well our agent performs. For our final agent we select network (512, 512, 256, 64) as it outperforms all other networks in terms of high performance with low training time. We left this agent to train on 150 000 episodes on the validation size problem and recorded the results. We see that network (512, 512, 256, 64) does not achieve approximate optimal score in this time. It seems as though it make tend towards it but this may take an infeasible amount of time (Figure 11.14).

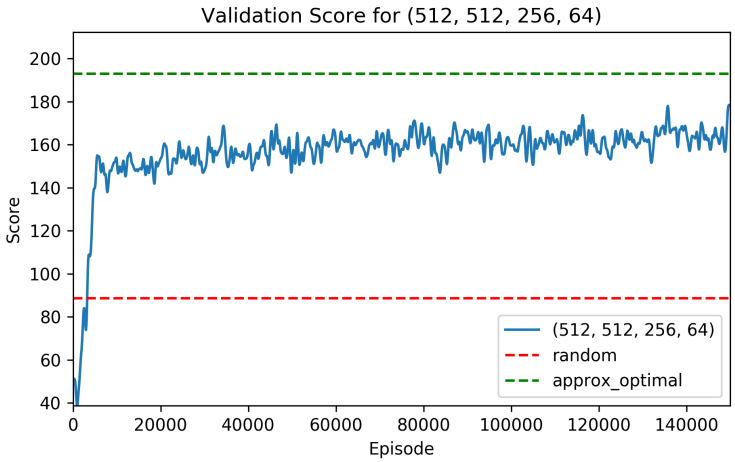


FIGURE 11.14: Validation scores being tracked over time during training for network (512, 512, 256, 64). We see that it quickly outperforms the random score but does not achieve approximate optimal score.

This is a disappointing result. If our agent cannot achieve approximate optimal score on a constrained version of the Code vs Zombies problem, it will not be able to achieve optimal score on the full problem. Reluctant to admit defeat, we made one last effort. We constructed a very large architecture in the hope that it could overcome the limitations of all the smaller ones that we have been dealing with up to thus far. We a network with hidden layers (1024, 1024, 512, 512, 256, 256, 128, 128, 64, 64, 32) for 150 000 episodes and recorded the results. We see that, while it learnt faster (per episode), it did not do any better than network (512, 512, 256, 64) (Figure 11.15).

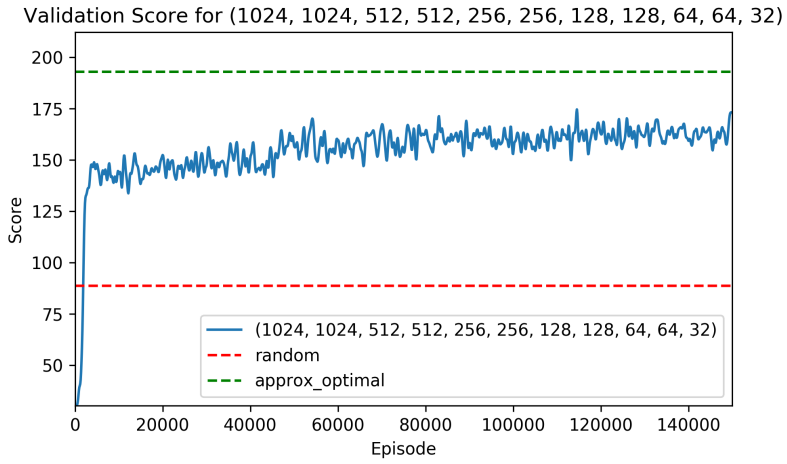


FIGURE 11.15: Validation scores being tracked over time during training for network (1024, 1024, 512, 512, 256, 256, 128, 128, 64, 64, 32). We see that it quickly outperforms the random score but does not achieve approximate optimal score.

We are forced to concede and admit that our agent cannot solve the Code vs Zombies problem as it cannot solve all constrained versions thereof. With this test we are reminded: "bigger is not always better".

11.4 Conclusion

We went through comprehensive experiments and their results and can now draw our conclusions. We used evidence to show: that network update delay outperforms target network update delay, that full replay sampling outperforms random replay sampling, the default action system performs better than all others and our reward scheme is effective, for our given problem. We also showed that our agent cannot achieve optimal performance on some constrained versions of the Code vs Zombies problem and, thus, cannot solve the full Code vs Zombies problem.

12 Conclusions and Recommendations

12.1 Introduction

We have gone through a large amount of information in this project report. Thus, this chapter serves as a brief overview of what was learnt, discussed and what further work could be done on this topic. First we will go through the main learnings: what reinforcement learning is, what approaches exist in reinforcement learning and what deep Q learning is. Then we will discuss the conclusions we can draw from the report, mostly surrounding the effectiveness of deep Q learning on the Code vs Zombies problem. Finally, we discuss follow on work, such as taking a policy search approach, adding a internal model of the environment and testing this project's agent on other problems.

12.2 Content

12.2.1 Learnings

We have learnt that reinforcement learning is the branch of machine learning that deals with sequential decision making and behaviour learning. We discussed the three main approaches in reinforcement learning: value based (deriving policies based on the desirability of states), model based (learning an internal model of the environment) and policy search (directly learning behaviours). We saw that deep Q learning is a value based approach where we use a neural network to approximate the action-value function, Q.

12.2.2 Conclusions

We saw that despite deep Q learning being designed for a discrete action space, it performs quite well on our Code vs Zombies problem once we have discretised the action

space. We noted that network update delay outperforms target network update delay, that full replay sampling outperforms random replay sampling, the default action system performs better than all others and our reward scheme is effective, for our given problem. We also showed that our agent cannot achieve optimal performance on some constrained versions of the Code vs Zombies problem and, thus, cannot solve the full Code vs Zombies problem. As such, our program completes all objectives for some constrained versions of the problem (the case where there is one zombie and one human is one example there of). However, our program only completes the most basic objective on the full problem as it always produces valid actions.

12.2.3 Recommendations for follow up work

Reinforcement learning is a very active field at the moment. While working on this project there were multiple large milestones achieved by the community, including the creation of AlphaGo Zero, the best Go playing program in the world. Thus, there are many ways one could extend the work done for this project.

Policy Based Approach

As previously discussed, a natural progression of this work would be to try a policy search based approach instead of a value based approach as policy search approaches allow continuous action spaces. This is due to the fact that policy search methods perform well on high dimensional environments and continuous actions spaces, two key areas where our agent is lacking. We recommend looking at the paper Continuous control with deep reinforcement learning (Lillicrap et al., 2015). That paper describes state of the art policy search methods.

Model Decomposition

The agent this report discusses uses a single neural network that must form some understanding of the state vectors that are presented to it and predict action-value pairs based on those. Once could try splitting the model into two neural networks: a feature extractor and an agent. It would be the job of the feature extractor to understand how the state vectors work and to present a lower dimension encoding to the agent network. The agent network can then be much smaller than the network used by this report and can solely focus on action-value pair prediction. We recommend using an

auto-encoder for the feature extractor as it is a type of neural network that is designed to do exactly that (Hinton and Salakhutdinov, 2006). The approach to create the agent network would not change from the approach followed in this project report at all, the only difference is the agent network takes the output of the feature extractor as its input (as its state).

Model Based Approach

One could also try build an internal environment model that the agent uses to plan sequences of actions. This couples quite nicely with using a feature extractor as one would prefer to learn a transition function for a lower dimensional representation of the state. We recommend looking at the paper Imagination-Augmented Agents for Deep Reinforcement Learning (Weber et al., 2017). With this one can achieve state of the art results.

Multiple Environments

One could take the agent that was created for this project and try to apply it to other problems or environments. It would be interesting to see if this agent generalises to other domains. If we had more time we would have done so. If this agent does not adapt well to another problem, one could investigate why it is so and what changes could be made to allow the agent to solve both problems. We recommend looking for problems on the CodingGame.com website, there are many to choose from and they are all quite interesting.

General Adversarial Networks

Generative adversarial networks have been producing state of the art results in interesting domains. One could try apply them to this problem and agent. The setup could look something like the following: the agent remains essentially the same as the one created in this project, the difference being that we introduce a model (the adversarial network) that generates game configurations for the agent to learn from (Halbritter, 2017). As training progresses the agent scores higher on the generated levels but the adversarial network attempts to generate more difficult configurations. Thus, the difficulty level is somewhat regulated such that it matches the agents capabilities and promotes efficient learning.

Learning Approaches

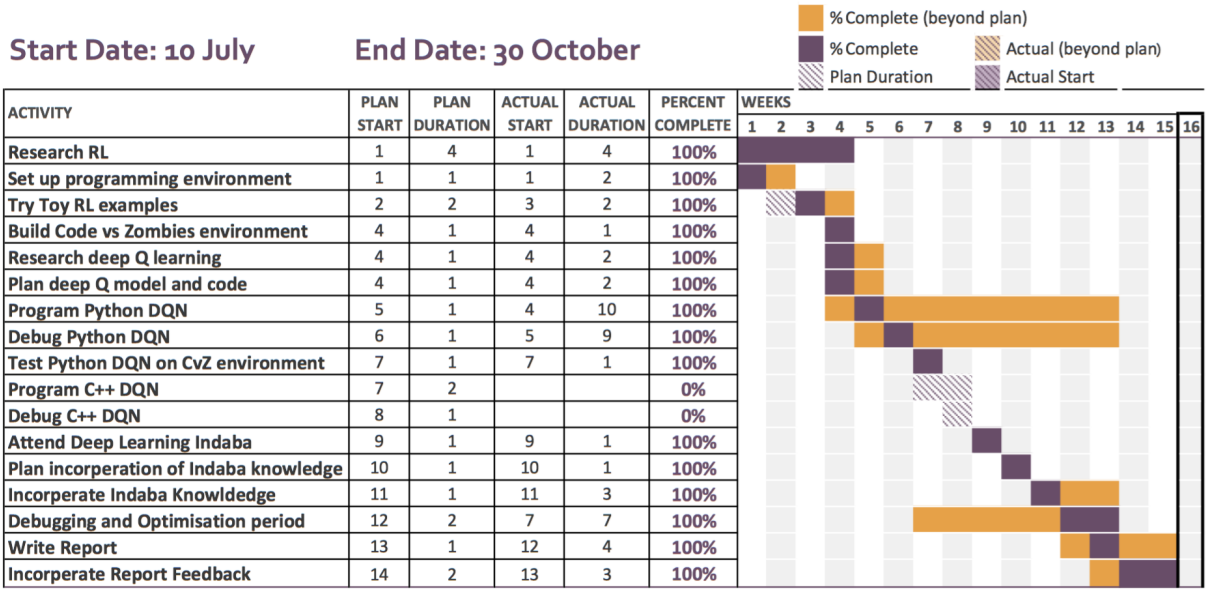
One could try more advanced learning techniques such as curriculum learning (starting with a basic example of the problem that the agent learns to solve and then progressively making the problem more complex as the agents policy becomes better) (Bengio et al., 2009), hierarchical learning and symbolic task acquisition (learning higher level actions, for instance: instead of learning the action "walk towards the zombie", the agent learns the action "go and kill the zombie") (Andersen and Konidaris, 2017) and inverse reinforcement learning (the agent learns by watching the behaviour of an expert solving the given problem) (Ng and Russell, 2000).

Decision Understanding and Visualisation

One could try understand what it is that the agent has learnt. One way to go about this would be to create a visual representation of the neural network and the weights and animate it alongside the environment as the agent interacts with it. We are at a stage in reinforcement learning where we understand that these approaches work but we do not always understand why or what was learnt by the agent. It would be very beneficial to the community if real progress was made on methods to gain that sort of insight.

A Project Planning Schedule

This appendix serves the purpose of showing the examiners the planned against the actual activities and durations for this project. We can see that all but two of the planned activities were completed. The two that were not completed were deemed to add nothing extra too the project and would have hindered other areas. We can see that most tasks took much longer than expected. This would have lead to the project not being finished had it not been for the tasks that started earlier than planned. We can see that once we started work on the code, we did not stop making changes and optimisations until very close to the end of the project (Figure A.1).



ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Research RL	1	4	1	4	100%
Set up programming environment	1	1	1	2	100%
Try Toy RL examples	2	2	3	2	100%
Build Code vs Zombies environment	4	1	4	1	100%
Research deep Q learning	4	1	4	2	100%
Plan deep Q model and code	4	1	4	2	100%
Program Python DQN	5	1	4	10	100%
Debug Python DQN	6	1	5	9	100%
Test Python DQN on CvZ environment	7	1	7	1	100%
Program C++ DQN	7	2			0%
Debug C++ DQN	8	1			0%
Attend Deep Learning Indaba	9	1	9	1	100%
Plan incorperation of Indaba knowledge	10	1	10	1	100%
Incorperate Indaba Knowldedge	11	1	11	3	100%
Debugging and Optimisation period	12	2	7	7	100%
Write Report	13	1	12	4	100%
Incorperate Report Feedback	14	2	13	3	100%

FIGURE A.1: Gantt chart showing planned and completed progress on this project. We see that most activities took longer than planned but the project was still completed on time.

B Outcomes Compliance

B.1 Introduction

This appendix serves the purpose of informing the examiners of the Engineering Council of South Africa (ECSA) outcomes that must be assessed in this project and will "state explicitly how each of the relevant ECSA outcomes were achieved during the execution of this project" (Botha, 2017).

B.2 Content

B.2.1 ELO 1 - Problem Solving

What is Satisfactory Performance?

Using the assessment material and opportunities, the student must show that he/she applied a systematic problem solving method to a complex engineering problem which required specialised engineering knowledge at a level consistent to that which a graduate would participate in an employment situation shortly after graduation. In his approach, the student must show that he/she understands and can follow a systematic technique which includes the following steps:

1. analysis of the problem
2. identification of the criteria for an acceptable solution, necessary information, and required engineering skills and knowledge
3. generation and formulation of possible approaches to the solution of the problem
4. modelling, analyses and evaluation of possible solution(s), and selection of the best solution
5. formulation and presentation of the solution in an appropriate form

(Botha, 2017)

What Did the Candidate do to Satisfy this Outcome?

1. Chapters 7 and 8 show an in depth analysis of the problem. The candidate identifies the underlying goals of the problem, how the information is to be encoded and how the nature of the problem affects the techniques one would use.

2. In Chapter 1 the candidates discusses possible milestones for the performance level of the agent. In Chapter 11 the candidate then updates his notions of what is realistically reachable by investigating the limiting factors and showing performance of his program against the lower milestones set in Chapter 1. The candidate develops and investigates the engineering knowledge needed to solve the problem in Chapters 3, 4, 5 and 6 by looking at the mathematical formulations of the approaches.

3. The candidate largely discusses this in chapter 3 by discussing the different approaches in reinforcement learning and what their strengths and weaknesses are.

4. In Chapter 11 the candidate conducts numerous experiments on multiple possible solutions and discusses the performance of each of them. This is largely done by experimental analysis of different network architectures and their performance.

5. The candidate presents the final model approach in Chapter 6 and presents the finer details of the approach in Chapter 11. He does so by describing what deep Q learning is, using the knowledge we gathered throughout the project report, and then defining the specifics of the deep Q learning model and techniques he used. The presentation of the final hyper-parameters can be found in Appendix D.

B.2.2 ELO 2 - Application of Scientific and Engineering Knowledge

What is Satisfactory Performance?

Using the assessment material and opportunities, the student must show that he/she has applied mathematical, scientific and engineering knowledge systematically to a

problem at a level consistent to that which a graduate would participate in an employment situation shortly after graduation. The student must show that he/she:

1. used mathematical techniques and/or numerical analysis and/or statistical knowledge and methods on engineering problems by:
   (a) applying formal analysis and modelling of engineering components, systems or processes
   (b) communicating concepts, ideas and theories with the aid of mathematics
   (c) reasoning about and conceptualising engineering components, systems or processes using mathematical concepts
   (d) and/or dealing with uncertainty and risk through the use of probability and statistics
2. used physical laws and knowledge of the physical world as a foundation for the engineering sciences and the solution of engineering problems by:
   (a) applying formal analysis and modelling of engineering components, systems or processes using principles and knowledge of the basic sciences
   (b) reasoning about and conceptualising engineering problems, components, systems or processes using principles of the basic sciences
3. used the techniques, principles and laws of engineering science at a fundamental level and in at least one specialist area to:
   (a) identify and solve open-ended engineering problems
   (b) identify and pursue engineering applications
   (c) and/or work across engineering disciplinary boundaries through cross disciplinary literacy and shared fundamental knowledge

(Botha, 2017)

What Did the Candidate do to Satisfy this Outcome?

1. used mathematical techniques and/or numerical analysis and/or statistical knowledge and methods on engineering problems by:

(a) In chapter 3 the candidate explains that we model the interaction between the environment and the agent as Markov decision processes and explains the benefits it brings in terms of analysis.

(b) The candidate formulates concepts such as the reward, transition and value functions in Chapter 3 using mathematics and then explains these formulations so that we gain an intuitive understanding of them.

(c) The candidate formulates the first order Markov assumption mathematically in Chapter 3. This very assumption reasons that our next state is dependent on only our current state and not all previous states.

(d) In chapter 3 the candidate formulates functions (such as the value and transition functions) stochastically so that our agent learns how to deal with uncertainty in our systems.

2. used physical laws and knowledge of the physical world as a foundation for the engineering sciences and the solution of engineering problems by:

   (a) The candidate largely does this in the code he has written but briefly explains in Chapter 9 that he was required to write a model of an environment that took physical laws into account. The environment facilitated certain actions, between entities in the environment, that take place when the distance between them is small enough. This changes entity trajectories and the state of the environment.

   (b) In chapter 10 the candidate gives an outline of engineering problems he encountered in the development of the project. He goes on to reason about, and give solutions to, them. An example of this is when he was required to make optimisations to code so that it would execute in a reasonable amount of time. This takes scientific knowledge of the fundamental operations the programming language and the computer support.

3. used the techniques, principles and laws of engineering science at a fundamental level and in at least one specialist area to:

   (a) The candidate identifies and solves a problem with how the state of the game is to be represented in Chapters 8, 10 and 11. This is a problem that few in the reinforcement learning community have come across and fewer have tried to solve.

(b) The candidate identified a number of problems that were easily solved by engineering applications. An example thereof is in Chapter 9 when the candidate applies dynamic programming to create a function that efficiently calculates Fibonacci numbers.

(c) A large portion of the work done in the reinforcement learning field is done by computer scientists and mathematicians. Thus, the candidate war required to enter those spheres and study the literature. This is outlined in Chapters 3, 4, 5 and 6.

B.2.3 ELO 3 - Engineering Design

What is Satisfactory Performance?

Using the assessment material and opportunities, the student must show that he/she has performed design and synthesis of components or systems at a level consistent to that at which a graduate would participate in an employment situation shortly after graduation. The student must show that he/she:

1. Identified and formulated the design problem to satisfy user needs, applicable standards, codes of practice and legislation

2. Planned and managed the design process – focusing on important issues, while recognising and dealing with constraints

3. Acquired and evaluated the requisite knowledge, information and resources, applied correct principles, and evaluated and used design tools

4. Performed design tasks including analysis, quantitative modelling and optimisation

5. Evaluated alternatives and preferred solutions, exercised judgment, and tested implementation ability

6. Assessed impacts and benefits of the design in terms of social, legal, health, safety, and environmental aspects (in the case of Project 448)

7. Communicated the design logic and information

(Botha, 2017)

What Did the Candidate do to Satisfy this Outcome?

1. In Chapter 7 the candidate identifies the problem and in Chapter 8 he formulates it such that it meets the needs of the techniques and standards involved.

2. The project plan is shown in Appendix A. It shows the original plan and the actual events. The candidate also discusses design processes, choices and constraints in Chapter 10. A specific example there of is the fact that the candidate acknowledges and works around the fact that deep Q learning is designed for a discrete action space.

3. The candidate cites many sources throughout the report and makes specific references to the application program interfaces and design tools used for the development of this project in Chapter 9.

4. The candidate was required to design software and has shown his abbreviated design in Chapter 9. He also shows the steps in designing the end model by taking the reader through the results from the experimental analysis in Chapter 11.

5. In Chapters 10 and 11 the candidate discusses alternative models or approaches, judgements made, preferred solutions and tested implementations of different networks.

6. Impacts society of reinforcement learning are discussed in Chapters 1 and 3. He comments that it is already being used to improve health care but there is still much discussion on the safety of artificial intelligence that is on going.

7. Chapters 10 and 11 largely communicate the candidate's design logic and the information thereof. However, this report serves as a design guide itself.

B.2.4 ELO 4 - Investigations, Experiments and data analysis

What is Satisfactory Performance?

Using the assessment material and opportunities, the student must show that he/she has designed and conducted investigations and experiments at a level consistent to that which a graduate would participate in an employment situation shortly after graduation. The student must show that he/she:

1. Planned and conducted investigations and experiments
2. Conducted a literature search and critically evaluated material
3. Performed necessary analyses
4. Selected and used appropriate equipment or software
5. Analysed, interpreted and derived information from data
6. Drew conclusions based on evidence
7. Communicated the purpose, process and outcomes verbally (in Design 314) or verbally and in a technical report (in Project 448)

(Botha, 2017)

What Did the Candidate do to Satisfy this Outcome?

1. The candidate has shown his project plan in Appendix A and described his investigations and experiments in 11.
2. Chapter 2 serves as a study of a specific piece of literature and Chapters 3, 4, 5 and 5 cover a wide range of other literature on the topic of reinforcement learning.
3. The candidate performed a great deal of analysis on the problem in Chapter 8 and fully analysed the results of experiments in Chapter 11.
4. In Chapter 9 the candidate discusses the software tools and application program interfaces he used.
5. The candidate analyses, interprets and derives information from the raw data acquired from experiments and this is discussed in Chapter 11.
6. In Chapter 11 the candidate draws many conclusions from based on the test results. An example of a conclusion he makes is that network update delay works better than target network update delay on the Code vs Zombies problem.
7. The candidate included a written account of objectives that this project and project report are to meet in Chapter 1.

B.2.5 ELO 5 - Engineering Methods, Skills and Tools, Including Information Technology

What is Satisfactory Performance?

Using the assessment material and opportunities, the student must show that he/she has designed and conducted investigations and experiments at a level consistent to that which a graduate would participate in an employment situation shortly after graduation. The student must show that he/she:

1. Used methods, skills or tools effectively by appropriate selection, proper application and critical assessment of the results
2. Created computer applications as required

(Botha, 2017)

What Did the Candidate do to Satisfy this Outcome?

1. The candidate discusses methods used to make optimisations in Chapter 10. He also discusses skills used in development of the project in Chapter 9. Finally, he uses application program interfaces to present the data in an effective manner in Chapter 11.
2. In Chapter 9 the candidate outlines the core computer applications he developed and in Appendix C he outlines all programs he created for this project.

B.2.6 ELO 6 - Professional and Technical Communication

What is Satisfactory Performance?

Using the assessment material and opportunities, the student must show that he/she can generate a long professional project report (10000-15000 words) and can defend the quality of his/her work during an oral examination. For written work, the student must provide evidence of:

1. The use of appropriate structure, style and language for purpose and audience
2. The use of effective graphical support

3. Application of technologically advanced methods of providing information
4. Meeting the requirements of the target audience

(Botha, 2017)

What Did the Candidate do to Satisfy this Outcome?

1. The intended audience for this project report is researchers and other intellectuals and the candidate has structured his language accordingly throughout the entire report.
2. The candidate has used figures and graphics extensively throughout the report but most helpfully in Chapter 11 when visualising experiment data.
3. Novel ways of graphically representing experimental results were used in Chapter 11. An example of this is the score and time variation plots.
4. The candidate recognised that that target audience may not have an extensive machine learning background and training. Thus, he explained all concepts starting with relatively basic ones in Chapter 1 and expanded on those same concepts throughout the report. The more basic elaborations are found in Chapter 3.

B.2.7 ELO 9 - Independent Learning Ability

What is Satisfactory Performance?

Using the assessment material and opportunities, the student must show that he/she has developed the ability to acquire knowledge in an independent fashion, apply such knowledge, and take responsibility for learning requirements. The student must show that he/she can:

1. Reflect on own learning and determine learning requirements and strategies
2. Source and evaluate information
3. Access, comprehend and apply knowledge acquired outside formal instruction
4. Critically challenge assumptions and embraces new thinking

(Botha, 2017)

What Did the Candidate do to Satisfy this Outcome?

The candidate would like to preface the specific answers given below by informing the examiner that there are no courses offered at his university that cover reinforcement learning and there are exceptionally few researchers at his university that deal with reinforcement learning. All knowledge pertaining to reinforcement learning in this project report was obtained by the candidate through self research and study. The candidate even attended a conference to obtain more knowledge on the field.

1. The candidate reflects on choices made and knowledge gathered in Chapter 12, especially with regard to the approach of policy search, which would most likely have been a better approach to the Code vs Zombies problem.

2. Chapters 3, 4, 5 and 6 contain many citations of the information the candidate found during development of this project.

3. In Chapter 6 especially, the candidate outlines his understanding of deep Q learning and application there of. This knowledge was gained outside of formal instruction.

4. The candidate critically challenged some of the methods discussed in Chapter 2 and devises his own alternative approaches. These approaches and their results are given in Chapter 11.

C Code

Many programs were created for this project. This includes:

- a simulation environment for the Code vs Zombies problem
- a deep Q learning agent
- an interface object that facilitates communication between the environment and the agent
- many scripts to run and record the results for the experimental investigation
- many scripts to plot the results of the experimental investigation

The code for the project can be found at: `https://github.com/ElanVB/cvz_env`

D Hyper-parameters

TABLE D.1: Hyper-parameters

Hyper-parameter	**Value**
test episodes	1000
validate episodes	100
batch size	32
replay memory size	3200
network update frequency	100
gamma	0.99
frame skip rate	4
optimizer	Nadam
initial epsilon	1.0
final epsilon	0.1
epsilon decay	0.000045
activation	ReLU
architecture	(512, 512, 256, 64)
learning rate	0.001

Bibliography

Andersen, G. and G. Konidaris (2017). "Active Exploration for Learning Symbolic Representations". In: *ArXiv e-prints*. arXiv: 1709.01490 [cs.AI].

Astels, Dave (2003). *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference.

Bellman, Richard (2013). *Dynamic programming*. Courier Corporation.

Bengio, Yoshua et al. (2009). "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning*. ACM, pp. 41–48.

Botha, Prof MM (2017). *PE448 2017 Study Guide*.

Chacon, Scott and Ben Straub (2014). *Pro git*. Apress.

Halbritter, Stephan (2017). "Generative Adversarial Networks". In:

Hinton, Geoffrey E and Ruslan R Salakhutdinov (2006). "Reducing the dimensionality of data with neural networks". In: *science* 313.5786, pp. 504–507.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5, pp. 359–366.

Konidaris, George, Sarah Osentoski, and Philip S Thomas (2011). "Value function approximation in reinforcement learning using the Fourier basis." In: *AAAI*. Vol. 6, p. 7.

Lillicrap, Timothy P. et al. (2015). "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971. URL: http://arxiv.org/abs/1509.02971.

Lin, Long-Ji (1993). *Reinforcement learning for robots using neural networks*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Lippmann, R. (1987). "An introduction to computing with neural nets". In: *IEEE ASSP Magazine* 4.2, pp. 4–22. ISSN: 0740-7467. DOI: 10.1109/MASSP.1987.1165576.

Martin, Robert C and Micah Martin (2006). *Agile principles, patterns, and practices in C*. Pearson Education.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518, pp. 529–542.

Ng, Andrew Y, Stuart J Russell, et al. (2000). "Algorithms for inverse reinforcement learning." In: *Icml*, pp. 663–670.

Ramchoun, Hassan et al. (2016). "Multilayer Perceptron: Architecture Optimization and Training". In: *International Journal of Interactive Multimedia and Artificial Intelligence* 4.1, pp. 26–30.

Rosman, Benjamin and Vukosi Marivate. *Intro to reinforcement learning*. http://www.deeplearningindaba.com/uploads/1/0/2/6/102657286/introtoreinforcementlearning.pdf.

Samuel, A. L. (1959). "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3, pp. 210–229. ISSN: 0018-8646. DOI: 10.1147/rd.33.0210.

Sen, Koushik, Darko Marinov, and Gul Agha (2005). "CUTE: a concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM, pp. 263–272.

Shin, Hyunjin and Mia K Markey (2006). "A machine learning perspective on the development of clinical decision support systems utilizing mass spectra of blood samples". In: *Journal of Biomedical Informatics* 39.2, pp. 227–248.

Weber, Theophane et al. (2017). "Imagination-Augmented Agents for Deep Reinforcement Learning". In: *CoRR* abs/1707.06203. URL: http://arxiv.org/abs/1707.06203.