Reinforcement Learning in the Minecraft Gaming Environment

By:
Matthew Reynard

Pectora roborant cultus recti

Supervisors:
Prof. Herman Engelbrecht
Dr Herman Kamper
Dr Benjamin Rosman

Date:
March 2020

Thesis presented in (partial) fulfilment of the requirements for the degree of Master of Engineering in the Faculty of Electrical and Electronic Engineering at Stellenbosch University

Plagiarism Declaration

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

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

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

4. 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.
   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.

5. 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.
   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.

Summary

With the long-term goal of surviving the night in Minecraft, we ask whether a reinforcement learning agent learns better by first learning the skills to perform smaller tasks in a complex environment or by learning the skills in the complex environment from the start. This is investigated empirically in a non-trivial game environment. We use the premise of curriculum learning where an agent learns different skills in independent and isolated sub-environments referred to as *dojos*. The skills learned in the dojos are then used as different actions as the agent decides which skill to perform that best applies to the current game state. We evaluate this with experiments conducted in the Minecraft gaming environment. We find that our approach of Dojo learning is able to achieve better performance with faster training time in certain environments. The main benefit of this approach is that the reward functions can be finely tuned in the dojos for each action as compared to the traditional methods. However, the skills learned in the individual dojos become the limiting factor in performance as the agent is unable to combine these skills effectively when put in certain complex environments. This can be mitigated if the dojo modules are further trained to achieve similar results as a standard deep Q network.

Opsomming

Met die langtermyndoel om ’n nag in Minecraft te oorleef, vra ons of versterkingsleer beter leer deur eers die vaardighede aan te leer om kleiner take in ’n komplekse omgewing uit te voer of deur die vaardighede in die komplekse omgewing aan te leer. Dit word in ’n uitdagende spelomgewing ondersoek. Ons gebruik kurrikulumleer waar ’n agent verskillende vaardighede aanleer in onafhanklike en geïsoleerde sub-omgewings waarna as *dojos* verwys word. Die vaardighede wat in die dojos aangeleer word, word dan as verskillende aksies gebruik aangesien die agent besluit watter vaardighede hy moet uitvoer wat die beste van toepassing is op die huidige speltoestand. Ons evalueer dit eksperimenteel in die Minecraft-spelomgewing. Ons vind dat ons benadering van Dojo-leer beter vaar met ’n vinniger opleidingstyd in sekere omgewings. Die belangrikste voordeel van hierdie benadering is dat die beloningsfunksies in die dojo’s vir elke aksie fyn ingestel kan word in vergelyking met die tradisionele metodes. Die vaardighede wat in die individuele dojos aangeleer word, word egter die beperkende faktor aangesien die agent nie in staat is om hierdie vaardighede effektief te kombineer as dit in sekere komplekse omgewings geplaas word nie. Dit kan versag word as die dojo-modules verder afgerig word om soortgelyke resultate te lewer as ’n standaard diep Q-netwerk.

Acknowledgements

I would like to sincerely thank the following people for their support and assistance during the course of this project:

- My parents,
- My friends,
- The MIH Media Lab for their financial aid and allowing me the opportunity to persue my Masters degree,
- The people of the MIH Media Lab for their advice, kindness and making the work environment a pleasure to work in,
- Most importantly my supervisors Prof. Herman Engelbrecht, Dr Herman Kamper and Dr Benjamin Rosman for their guidance, feedback and utmost support throughout my Masters.

Thank you.

Contents

Nomenclature

Abbreviations

2-D	2 Dimensional
3-D	3 Dimensional
AGI	Artificial General Intelligence
AI	Artificial Intelligence
BOI	Blocks of interest
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DQN	Deep Q Network
HTN	Hierarchical Task Network
IRL	Inverse Reinforcement Learning
MDP	Markov Decision Process
ML	Machine Learning
NLP	Natural Language Processing
NN	Neural Network
RGB	Red, Green and Blue
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SMDP	Semi Markov Decision Process
TD	Temporal Difference

Symbols

α	Learning rate
β	Terminating condition
ϵ	Exploration parameter
γ	Discount factor
π	Policy
τ	Transition time

Units

m	meters
s	seconds

Definitions

agent	a reinforcement learning robot that interacts with the environment
bot	an autonomous program which interacts with a system or the user
mobs	the dangerous creatures in Minecraft which roam the environment at night and are able to attack the player
sandbox	a gaming environment which has minimal limitations on the player allowing it to roam the virtual world freely

Chapter 1

Introduction

The aim of having a robot successfully interact with a challenging environment is one that researchers are keen to solve. With the recent advancements in reinforcement learning (RL), such as the Atari 2600 from Google DeepMind in 2015 [1], Alpha Go in 2016 winning the current world champion in the board game Go [2], and OpenAI winning a 5v5 match against the top players in the world in Dota 2 [3], RL has become a powerful tool in achieving super human results in games. RL agents appear to be able to master any game, but what about a game such as Minecraft – a game that does not have a round timer or level progression? An agent cannot play the game in a few episodes nor be matched against an opponent. Minecraft is different from other games which RL has seemed to conquer with an environment comparable to a real world scenario.

1.1 Research objective

The long-term objective of our research is to use reinforcement learning (RL) to teach an agent to survive a day-night cycle in the Minecraft gaming environment. We test a new method, referred to as dojo learning and based on curriculum learning, against current methods to progress one step closer to our goal. Although Minecraft is used as a testing platform, our method could be generalised and adapted to work in any appropriate gaming environment.

There are two possible approaches to conducting this experiment. One is using RL by allowing the agent to explore the world by itself and learn the best technique to survive. The other is by acquiring survival techniques by studying and learning from real world players, also known as inverse reinforcement learning (IRL). Each approach has its own drawback, with RL needing large computing power and a long simulation time, and IRL requiring a large amount of player data to be collected in order for the agent to learn the optimal strategy.

1.2 Minecraft overview

Minecraft is a 3-D sandbox game which allows the player to do almost anything in a procedurally generated grid-world environment. The player can explore the vast open world, gather resources and build structures with a variety of blocks. The resources can also be used to craft items which can aid in exploration and survival.

The day-night cycle in Minecraft lasts for 20 minutes in real world time. During the day, the player is allowed to gather all the resources they want with little danger or consequences. However, once the sun sets the world of Minecraft is swamped with mobs – dangerous creatures which roam the environment at night. The mobs can attack the player when in range and if the player loses all their health, they will be stripped of all the gathered resources and respawn elsewhere.

There are many game modes in Minecraft. Our focus is survival mode where the mobs pose a threat and the player has to gather resources. Another is creative mode where the player has any type of block or item in their endless inventory and can build and create any sort of world they imagine. For further insight into the world of Minecraft, refer to Section 2.6.2.

Microsoft saw potential in Minecraft and purchased it from Mojang AB in 2015 for $2.5 billion (USD) [4]. Microsoft also developed Project Malmo, a research platform for machine learning (ML) algorithms and set a goal of having a reinforcement learning agent survive the night [5]. They were unsuccessful in achieving this ambitious goal initially and set their sights on smaller, more manageable sub-environments in the world of Minecraft. However, the goal remains: can an agent survive the night in Minecraft using reinforcement leanrning?

1.3 Reinforcement learning overview

Reinforcement learning (RL) is a subset of machine learning, a term coined by Arthur Samuel in 1959 [6]. A more formal definition of machine learning was later given by Tom M. Mitchell stating: “A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*" [7]. This definition introduces the basic concept of machine learning (ML), which will be elaborated on in Section 2.1. In summary, machine learning is when a computer acquires the knowledge to complete a given task through experience.

The idea behind RL is to teach a computer in a similar way dogs learn new tricks – after performing an action, it will receive either a positive or negative reward. This is also true when humans learn. The major difference between a human and a computer learning a new task is that a human will use common sense (usually from life experience or natural instinct) to determine the next approach if the first was unsuccessful. However, a computer will often perform a random action, or a slight variation of what it previously did until it achieves the goal of the task. However, the computer will attempt the task many times and will gather experience quicker than the human is able to given the same time allocation.

A simple example of this is when humans begin to drive before having previous knowledge of driving. We use logic to brake at corners and use 'common sense' to turn at the appropriate time. But this common sense is not a feature in computers and they will first have to fail at taking a corner before they learn that they cannot take a corner that fast, or drive through a wall or any other radical approach.

This sort of common sense can be programmed or hard-coded into a computer but this will limit it from learning through its own experience and can hinder the program from reaching its full potential. Sometimes the best way to succeed is to take risks, even if it goes against all common sense. This allows computers to potentially perform better than any human could, even in intellectual activities.

1.4 Significance and motivation

Artificial general intelligence (AGI) is a goal which a great deal of machine learning research is striving towards. The aim of having one robot do everything is one that has captured the imagination of society and has ingrained itself in pop culture throughout the world. Currently however, the robots we have are only good at one specific task. These algorithms may have achieved super human results when playing a specific game, or are able to drive within a crowded city while minimising the risk of an accident, but no one algorithm can play all the games a human can and drive a car. This is one main reason that the Atari 2600 by Google DeepMind [1] was such a breakthrough as it is the first case of one algorithm able to achieve super human success in multiple games on the platform. But we still have a way to go to achieve true AGI.

The idea of an RL agent exploring the world of Minecraft is a step towards AGI. The agent needs to master multiple aspects of the game to be able to explore the world, let alone fight mobs and craft items. With the research and experiments we conduct, we progress towards a better algorithm for AGI.

1.5 Experiment overview and hypothesis

The experiment set out to establish a baseline for dojo learning which is explained in Section 2.5.1. We consider two approaches: a network, which will learn from the full environment directly, and our new dojo network, which will learn the simple, primitive actions in sub-environments first before attempting the full environment. Both these approaches are explained in detail in Chapter 3.

The goal of our experiments is to determine which of the two approaches is more successful in allowing an agent to achieve a high score in a challenging environment which requires mastery of multiple tasks. Our hypothesis is that the agent with knowledge of the sub-environments will perform better in the full environment and be able to learn more sophisticated tasks within that environment than an agent that began its learning in the full environment. The latter will potentially be overwhelmed with the amount of learning needed and will not be able to learn the skills efficiently, if at all.

1.6 Contribution

Our contribution is the new approach for an RL agent to succeed in a challenging environment in the form of a dojo network. Our approach includes learning simpler tasks in sub-environments and merging them into one model which is able to do any of the sub-tasks as well as a combinations of sub-tasks. In order to make the learning efficient, a 2-D Python environment of Minecraft was also created using PyGame. This is available to the public on GitHub at `https://github.com/Matthew-Reynard/malmo`. A paper titled “Combining primitive DQNs for improved reinforcement learning in Minecraft” has also been approved for the PRASA 2020 conference.

1.7 Layout

In the next chapter we discuss many aspects and ideas related to the research in detail. Our discussion covers the basics of machine learning, to the Q-learning algorithm and the Deep Q-Network (DQN) which is used in the experiments. We also discuss what Minecraft is and why it is chosen as the RL testing environment.

The following chapter, Chapter 3, breaks down the approach taken, and the decisions and reasons behind them in order to answer the research question proposed. The first few sections of this chapter will outline the learning experience and how the final approach came to be. The classic game of Snake is used to educate the reader about the Q-learning algorithm.

The experiment chapter, Chapter 4, lays out the steps taken to achieve the results. It outlines the architecture of the neural network used, discusses the code, and explains the decisions made for the experiments and environment setup.

In the results, Chapter 5, we discuss the findings of the experiment and what information is learnt by investigating the training curves of the experiments.

The conclusion, Chapter 6, discusses the results compared to our hypothesis and whether the research objective is met. The chapter also highlights possible alternative approaches, improvements and future work that could be added to the experiment to refine the results or make them more conclusive.

Chapter 2

Background

In this chapter we delve into machine learning and specifically where reinforcement learning (RL) belongs in the broad field of computer science. The ideas and algorithms used in later chapters such as neural networks are explained, as well as the Minecraft gaming environment chosen to run the experiments is discussed in detail. Related work and similar research will be discussed in the chapter, and our approach of dojo learning is explained. First we start with the basics of machine learning and RL.

2.1 Machine learning

Machine learning can be summarised neatly by these two historical quotes from Arthur Samuel and Tom Mitchell. Samuel was one of the first to define the term in the late 1950s, while Mitchell gave it a more robust definition in 1998.

"Machine learning is a field of study that gives computers the ability to learn without being explicitly programmed." – Arthur Samuel (1959) [6]

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." – Tom M. Mitchell (1998) [7]

These definitions not only explain the concept well, but also reveal the age of the idea of machine learning. Many think of machine learning as being a recent advancement but it has been around for many decades. It has only gained traction recently due to the improved computing power available to the public.

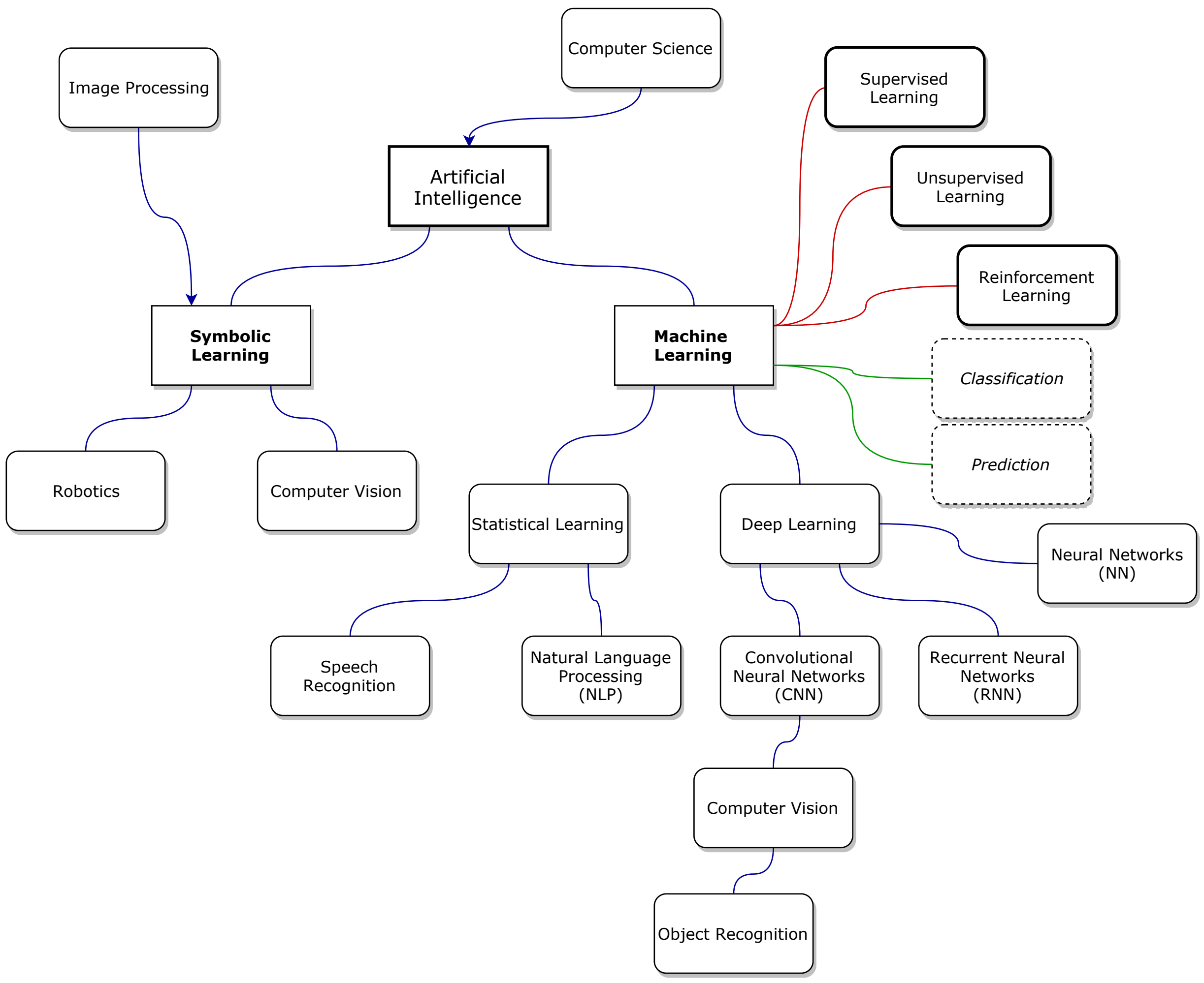


Figure 2.1: A breakdown of the artificial intelligence field and where reinforcement learning fits into the broader field of computer science. (Figure created based on content from [8].)

Artificial intelligence (AI) is a broad area of computer science. It can be split up into two main branches, symbolic learning and machine learning (ML) as shown in Figure 2.1.

Symbolic learning is an older form of AI and was popular in the 1950s until the 1980s. Since then a more advanced type of AI has grown in popularity, namely, non-symbolic AI, and for good reason. However, this is not to say that symbolic learning is inferior, there are still many practical uses for this type of machine learning and in certain situations it can perform better than modern non-symbolic AI.

Symbolic learning involves using symbols and other ‘human readable’ information that is relative to the task. It then uses this information in a type of lookup table or search algorithm by finding keywords and showing the relevant information initially provided. This allows the symbolic AI bot to be quickly set up and working for a small task, but with large data files it might take time to search and deliver a result. Whereas a non-symbolic learning AI will take a long time to set up and 'train', it will often operate more efficiently and be easily expandable [9].

Machine learning, which falls under non-symbolic AI, is used for pattern recognition. Raw data is given to the AI bot, unlike symbolic learning, and the computer learns patterns in this data either by the given labels or simply from the data itself. Machines are significantly better than humans at pattern recognition as they are able to use more data and more dimensions of data [8].

2.1.1 Classification and prediction

The two main functions of machine learning is either to classify data or predict using given data. Classification is when the labels of the data or the data itself falls into a category or class, and given the new raw data the algorithm classifies it into one of these categories. A common example of this is the MNIST dataset [10], a classic machine learning tutorial where an algorithm learns from the labeled dataset that contains handdrawn numbers from 0 to 9 that is 28x28 pixels in size. After training, the AI model needs to decide which one of the labels best fits the new given input in the test set.

With prediction, the data does not necessarily fall into a class and is more often than not continuous data. The task of the AI is to predict the unknown data given the specific input. An example of this might be of a linear regression model, where the temperature is related to the number of people at the beach. Given the temperature, the AI can predict the amount of people, and vice versa.

2.1.2 Supervised and unsupervised learning

Supervised and unsupervised learning are the two main methods of machine learning. Supervised learning is a method of training a model to map an input to an output based on labelled data, whereas unsupervised learning refers to the method of finding patterns within unlabelled data. There is also a mix of these two methods known as semi-supervised learning, a combination of the two previously described methods, where the data is partially labelled. Semi-supervised learning methods is often used in the case of large amounts

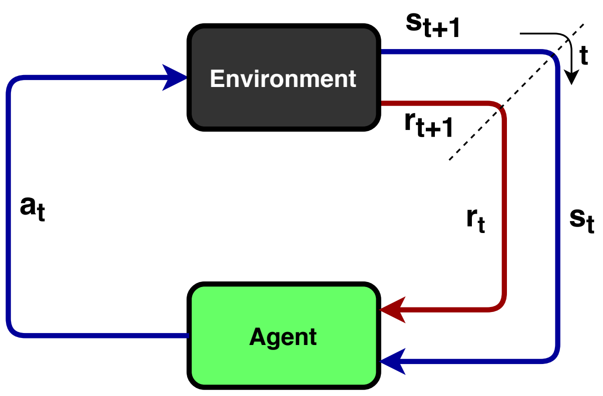


Figure 2.2: The reinforcement learning iterative interaction cycle: The agent performs an action, a_t, given the current state, s_t, of the environment at time t. The environment reacts to this action and the agent receives a scalar reward, r_t, based on the action taken, as well as receiving the next state, s_{t+1}. This cycle is repeated during the learning process.

of data and where only a small portion is of the data is labelled.

In summary, supervised learning uses labelled data and normally takes the form of linear regression (prediction) or logistic regression (classification) models, where unsupervised learning has unlabelled data and is mainly used for clustering. We do not use these two methods of ML in this investigation, and therefore will not explore them further.

2.2 Reinforcement learning

A brief overview of reinforcement learning was given in Section 1.3. In this section we will discuss RL in more detail, going over the terminology and explaining the different methods and algorithms which will be used in later sections.

The idea of reinforcement learning differs from supervised and unsupervised where instead of a computer learning through large amounts of data, the computer will learn through an agent performing actions and gathering experience in an environment.

Referring to Figure 2.2, the agent performs actions in an environment and is learning through trial and error over many episodes. An episode refers to everything the agent does within a finite number of states between the initial state and the terminal state. The environment changes accordingly and the agent receives feedback in the form of a scalar reward. The goal of RL is for the agent to maximise the expected cumulative reward, G_T, also known as the return, which is explained later in this section.

Reinforcement learning has some terminology that one needs to understand in order to discuss the other aspects more effectively. We look at the RL terms and explain each one.

History: Mathematically written as a set of all aspects up until time t that affect the agent within an episode, $H_t = \{O_1, R_1, A_1, \cdots, A_t, O_t, R_t\}$. This is a record of the observations, O, the rewards, R, and the actions taken, A, from the initial state to the terminal state.

State: The state is a function of the history $S_t = f(H_t)$. It is used to describe the configuration of the environment to the agent and based on this state, the agent will decide on an action to perform. There are two different states one can consider in RL: the agent state, which is one used for the agent to determine the next action, and the environment state, which fully defines the current environment configuration. The term state commonly refers to the agent state unless otherwise stated.

Observation: An observation is what the agent *sees*. It is a function of the state of the environment $O_t = f(S_t)$. The entire environment state often has irrelevant information for the agent, and therefore only a subset is used for decision making. If however, the agent is able to observe the entire state of the environment, we say that the environment is fully observable. Otherwise the environment is partially observable.

Action: The action taken by the agent at time step t, shown as a_t, is within the total action space of the agent $a \in A$. This is the only method for the agent to interact with the environment.

Reward: The reward is normally a function of the current state, the action taken and the next state, $R(s, a, s')$, and it defines the goal in an RL task. Each time step a reward is sent to the agent from the environment, always in the form of a scalar. The agent's objective is to maximise the total discounted reward received over a defined time period or task. It defines what are good and bad behaviours for the agent and is the primary basis for altering the policy.

Policy, π: Defines the agent's way of behaving at a given time or in a given state, $\pi(a|s)$. A policy is a mapping of the current state to the actions taken when the agent is in those states [11]. It is the core of the agent, and it alone determines the agent's behaviour. Policies may be fully stochastic or 'greedy'. A greedy policy always goes for the best immediate reward gain which limits the agent's ability to explore other possible options. We use the ϵ-greedy policy, which combats the exploration and exploitation dilemma discussed in Section 2.2.6. This policy is based on the greedy policy, but has a chance of a random action being attempted which aids in the exploration of the environment.

Value function: A reward signal shows what is a good behaviour immediately, but a value function shows the agent what is a good behaviour over time. The value of a state is the total amount of reward an agent can expect to accumulate in the future when starting from that state. The value function will constantly be trying to approximate the discounted reward for any given state. The rewards determine the immediate intrinsic desirability of the states whereas the value shows the long term desirability of the state by predicting the states that are likely to follow and their discounted rewards. Rewards remain the main objective. Without rewards there would be no values, and the reason there are values is to receive more rewards down the line. However, values are more important when making decisions. A crucial role in RL is estimating these values efficiently [11].

Transition probability: The transition probability refers to the chance of transferring from one state to another based on the current state and the action taken, $P(s'|s,a)$.

Discount factor ($\gamma \in [0,1]$): The value of the discount factor describes how much an agent values future rewards. As the discount factor, γ, approaches the value of 0 the agent is only concerned about immediate rewards, and as γ approaches the value of 1, the agent is only focussed on potential future rewards [12].

Return, G_t: Another term for return is the discounted cumulative expected reward. The overall goal for any RL algorithm is to maximise the return. The total discounted return is shown in equation 2.1, with R_t being the reward received at time t.

$$G_T = R_{t+1} + \gamma^1 R_{t+2} + \ldots = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.1}$$

Model of the environment: A model mimics the behaviour of the environment. Given the action and the state the agent resides, a model can predict the rewards and state at the next time step in order for the agent to make a calculated decision on its next action. Models are used for planning. RL methods that use models for planning are known as model-based methods and are elaborated on in Section 2.2.5. Model free methods are solely trial and error methods and use no planning [11]. The word model can also refer to the AI function used to predict the output given the input. Hereafter, the word model will refer to the AI function unless otherwise stated.

Step-size parameter or *learning rate*, α: A small positive fraction that can be reduced over time for the method to lead to convergence. This is the rate of learning, which is used in optimisation algorithms discussed in Section 2.4.4. If the parameter is not reduced to zero, the agent will remain competitive even to an evolving opponent.

To reiterate, the goal of RL is to optimise over a value function and have an agent choose an action that will maximise future return. We now discuss various aspects of RL.

2.2.1 Markov decision process

In the field of probability and statistics, the Markov property expresses that the future is independent of the past, given the present [13]. The current state completely characterises the position and condition of all entities and objects within the world.

A Markov decision process (MDP), shown in Figure 2.3, describes an RL environment and is defined by the tuple

$$(S, A, R, P, \gamma),$$

with S being the set of all possible states of the environment, and A referring to the set of all possible actions an agent can perform. R is the distribution of the rewards given and is a function of the state and action performed. P is the transition probability from one state to another and γ is the discount factor and describes how much the agent values future rewards.

There are also different order MDPs. A first order MDP is one in which the next state is only determined by the current state and the action taken. If the next state depends on two previous states it is a second order MDP, and so on.

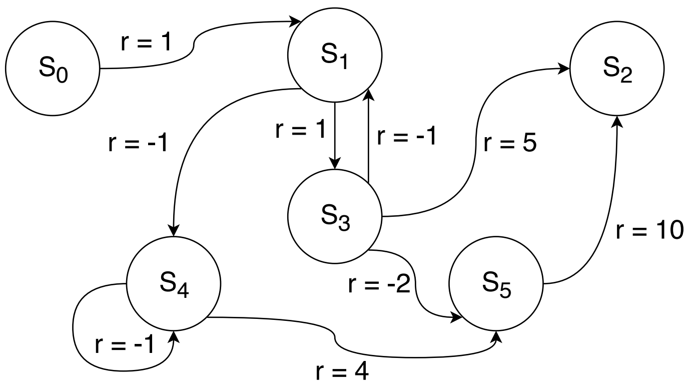


Figure 2.3: A Markov decision process diagram with initial state S_0 and terminating state S_2.

Another type of MDP is the semi Markov decision process (SMDP). In an SMDP the actions have duration, whereas in the standard MDP the actions are instantaneous. Not all actions have the same duration, and could take varying lengths of time depending on the action. This is referred to as holding time or transition time and is represented as τ. Another way to imagine it is the agent decides on an action at time t but only executes the action at time $t+\tau$.

In order to solve for the values of an MDP, one needs to use the Bellman optimality equation (2.2). This is a non-linear equation that can only be solved using iterative methods [13].

$$V^{\pi}(s) = R(s,\pi(s)) + \gamma \sum_{s'} T(s,\pi(s),s')V^{\pi}(s') \tag{2.2}$$

In words, the Bellman optimality equation states that the value of a given state under a certain policy is equal to the reward of that state plus the discounted future value of the next possible states.

2.2.2 Passive and active reinforcement learning

An agent in RL can be either passive or active, or a combination of the two. An active RL agent changes its policy, π, as it explores the environment and learns. One method of this is the greedy approach, as explained in Section 2.2. As the agent acquires more rewards, it updates the current optimal policy and then uses that policy to guide the next actions. It is known as greedy as it wants to get the best policy immediately and does not want to waste time on a sub-optimal policy.

A passive RL agent has a fixed policy and as it explores the environment it learns the reward, R, or transition model T [14]. Over time it will update the policy but only once it has stored enough rewards and learnt from the current policy.

2.2.3 Episodic and continuing tasks

Episodic tasks are tasks with a start point and an end point, or in RL terms, an initial state and a terminal state. Most games where an RL agent is asked to learn are episodic tasks. Continuous tasks go on forever and have no terminal state, such as an agent which must learn the difficult task of autonomous stock trading [15].

2.2.4 Online and offline learning

Online learning is when the model is trained as the data is gathered, whereas offline learning is training a model from a static dataset.

In terms of RL, online learning would be to update the model and train it after every step or action is executed. Each time the agent executes subsequent actions, it is using the new, updated model.

This is different from offline, where the same model is used for multiple actions and only updated after a certain number of episodes or actions. This can be done by storing the history of the agent and having a mass update of the policy.

2.2.5 Model-based and model-free learning

The difference between model-based and model-free learning is that with model-based the agent is trying to understand the entire environment and how it works, essentially creating model for the environment. This type of learning tries to learn two aspects of the world, the transition function between states, and the reward function. Using these functions, the agent is able to predict the reward it will receive and the next state, and can therefore plan accordingly.

With model free, instead of trying to understand the entire environment, the agent simply learns by trial and error and learns a policy according to the current state. The Q-learning algorithm, which is used in our experiments, is a model free method.

2.2.6 Exploration and exploitation tradeoff

Exploration is acting, be it randomly or in a deterministic manner, in order to explore unseen areas of the environment to update what is known about the environment. Exploitation, on the other hand, is acting solely in a way to maximise the return based on the current understanding of the environment. The trade off between exploration and exploitation is the problem that separates RL from supervised and unsupervised learning [11]. The ϵ-greedy policy provides a way to manage the two ideas, and allow the agent to perform well in both aspects.

2.3 Reinforcement learning methods

There are three fundamental methods of RL. Understanding these will allow one to easily grasp an understanding of the more complex methods.

Dynamic programming: This type of method is well developed mathematically but requires a complete and accurate model of the environment [11]. This incorporates model-based learning and planning methods.

Monte Carlo: These methods are simple and do not require a model of the environment. They are also not well suited for step-by-step incremental computation [11]. With Monte Carlo, the agent only receives rewards at the end of an episode. The agent looks at the total cumulative reward, G_T, to see how well it performed, and begins the new episode with that newly acquired knowledge. A general equation for Monte Carlo is shown with equation 2.3.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]. \tag{2.3}$$

Temporal difference (TD) learning: These require no model of the environment and are fully incremental methods, but are more complex to analyse than Monte Carlo methods [11]. If an agent learns at each time step, this method is known as TD(0), however, experience can be gained and learnt through a batch update with N time steps of experience being $TD(N)$. As N approaches infinity, or simply the end of the episode at time step T ($N \rightarrow T$), TD will effectively be the Monte Carlo approach. Equation 2.4 shows a general form of a TD learning equation.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1} - V(S_t))]. \tag{2.4}$$

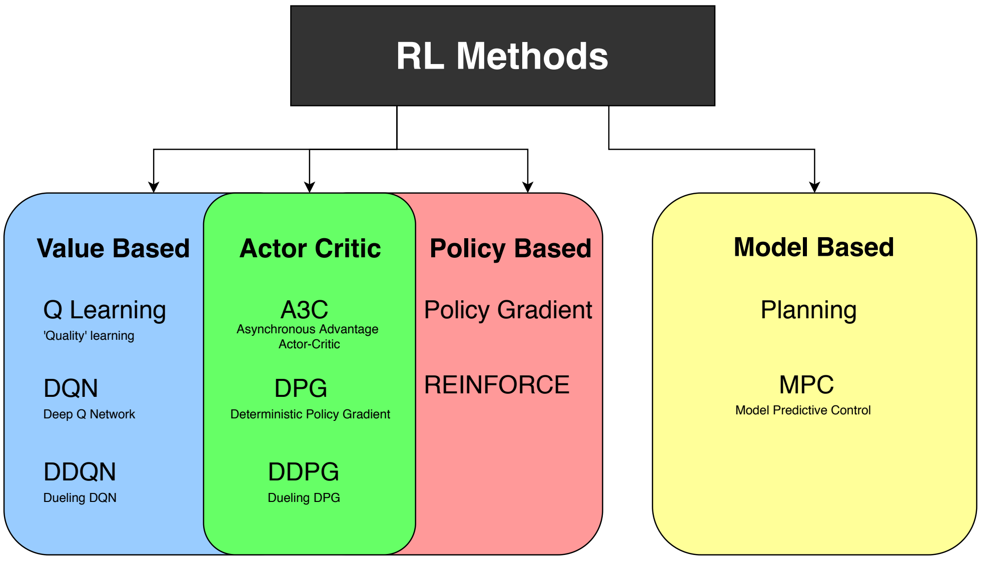


Figure 2.4: The different types of reinforcement learning methods and the most popular algorithms of each method type.

With the gained knowledge of these fundamental method types, we now discuss the specific algorithms which fall into the categories shown in Figure 2.4 of value based, policy based and actor critic. Model-based is not further discussed as we focus on model-free methods.

2.3.1 Value based

A value based method is based on TD learning, with the premise of learning the value function. The value function calculates the discounted return that an agent will receive at each given state, with the general form of equation 2.5.

$$v_\pi(s) \quad = \quad \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \ldots | S_t = s] \tag{2.5}$$

Q-learning is a value based algorithm which forms the fundamental approach to our algorithm. Being a value based algorithm means there is no need for a model of the environment and it is an iterative method used to calculate the ‘quality’ of each state-action pair.

A simple form of Q-learning could be thought of as a lookup table, or Q-matrix, where the values of the matrix is the Q values for each state-action pair. The dimensions of the matrix is $states \times actions$, with a reward matrix also defined with the same dimensions. Starting from the initial state of the environment and iterating through the formula shown in equation 2.6 (with pseudocode provided in Section 3.1.2), over many episodes, the randomly initialised Q values will begin to represent the actual Q values of each state-action pair.

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max_{a} Q'(s',a') - Q(s,a)] \tag{2.6}$$

The idea of a deep Q network (DQN), is using the iterative method of Q-learning to train a deep neural network, which is explained in Section 2.4. Effectively, a DQN is using the state of the game as the input to the neural network, and the actions of the agent as the output.

2.3.2 Policy based

Rather than trying to learn the value of an action or state, policy based methods learn the policy, π, directly. This is considered a Monte Carlo method of learning, meaning the agent first collects data from an entire episode and thereafter performs calculations and updates the model at the end of that episode [12]. Algorithms such as the policy gradient method and the REINFORCE algorithm fall under policy based methods, as both solely evaluates and updates the agent's policy.

2.3.3 Actor critic method

The actor critic method is a combination of value based and policy based as depicted in Figure 2.4. The method uses two policies to learn, a behavioural policy and an estimation policy. This is also known as an off-policy method as the policy used to perform actions is separate to the policy which is learning and improving. A method which has one policy is an on-policy method as it learns the policy while using it to generate the actions.

The critic (the estimation policy) evaluates the actor's actions and behaves similar to a value based algorithm by estimating the value function as the actor explores the environment. The actor (the behavioural policy) acts using the same policy for a given time or a certain number of episodes, learning the most from the policy and performing updates at the end from the knowledge learnt by the critic. Methods for this include the advantage actor critic (A2C) and the asynchronous A2C (A3C).

2.3.4 Genetic algorithm and neuroevolution

Another method of an agent 'learning' is to use a genetic algorithm. This does not fall under RL as it differs greatly at a fundamental level, but can however yield similar results when dealing with simple environments.

A genetic algorithm is a method where many agents are created with a variety of brain parameters known as genes, or to use a term similar to RL, policies. These agents are then all put into the same environment with the same goal and allowed to explore by themselves. Whichever agent or set of agents are closest to the goal, or are deemed to have succeeded the most, are selected and a portion or their genes is shared amongst the new generation of agents with a degree of randomness or mutation, attempting to build a stronger agent.

This algorithm is derived from Darwinian Natural Selection [16], which is colloquially termed *survival of the fittest*, and is an evolutionary method. Evolving policies do not learn while interacting with the environment and is therefore not well suited for large RL tasks in general.

Neuroevolution is a technique of combing genetic algorithms and neural networks. The neural network acts like the genes of an agent, with the stronger selected networks sharing neurons with slight mutation over multiple episodes or generations of agents. This method requires a large amount of computational power as one will have many agents acting in the same environment all evolving to achieve one task.

2.3.5 Inverse reinforcement learning

With a better understanding of RL, let's briefly explain the other possible approach to achieve our research objective, inverse reinforcement learning (IRL).

With IRL, the goal is to use expert demonstrations in order to infer a reward function [17]. Each episode that an expert performs is characterised by the states, s, and actions, a, of that expert's demonstration, grouped into the trajectory, τ, shown in equation 2.7. The full set of all experts demonstrations, D, performed under the optimal policy, π^*, is shown in equation 2.8. Using this data, the goal is to calculate the reward represented in equation 2.9.

$$\tau = \{s_1, a_1, \ldots, s_t, a_t, \ldots, s_T\}, \tag{2.7}$$

$$D = \{\tau_i\} \sim \pi^*, \tag{2.8}$$

$$R_\phi(\tau) = \sum_t r_\phi(s_t, a_t). \tag{2.9}$$

The main idea when using IRL methods, such as maximum likelihood or maximum entropy IRL, is that we cannot simply mimic the actions of the expert. One reason for this is that the expert might not be perfect or behave in an optimal manner, and therefore IRL is normally a good starting point for solving RL problems. Another reason not to mimic the expert is the environment might be stochastic and the agent might not have been exposed to a specific state before and will have to act accordingly.

The process of gathering data from experts, or crowdsourcing, is an area of concern on its own. The amount of data needed for an IRL method is less than that of an RL method, based on the fact that the agent is assumed to be operating under an optimal policy. Although, depending on the task, the optimal policy might differ, and if the optimal policies differ greatly, the agent might find it challenging to learn from the experts.

Preprocessing the data gathered is a step in the ML process that is most often overlooked or its importance underestimated. It is ensuring that the data is set up correctly in the best format with no duplicates. The data that is duplicated or redundant should be carefully discarded.

In summary, gathering data from people is an ethical and administrative challenge. Learning from experts might also limit the agent's ability to learn as it might be learning from a sub-optimal policy. Since we have chosen to use an RL approach, IRL is not discussed in depth.

2.4 Neural networks

A neural network (NN) is a massive function approximator. An input is given to the network and it provides an output. Neural networks, shown in Figure 2.5, are inspired by the way a brain works, with the values in the NN referred to as neurons or nodes, connected by weights and added biases. Each layer goes through a similar calculation in the forward pass shown in equation 2.10, with X representing the input nodes, W and B representing the weights and biases respectively, and the function, f, representing the activation function.

$$y = f(\sum_{i=1}^{n}(W_i \times X_i) + B) \tag{2.10}$$

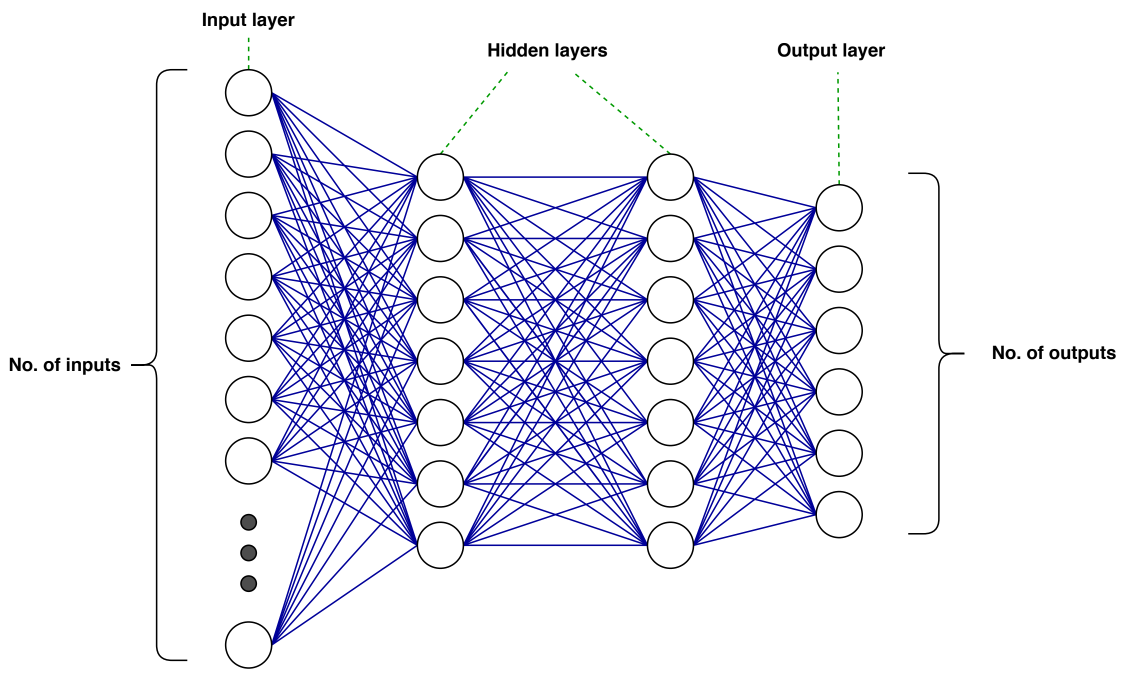


Figure 2.5: A standard fully connected neural network layout, with an input layer, two hidden layers, and an output layer.

2.4.1 Activation functions

An activation function is a non-linear function which changes any input number into a number between 0 and 1, or in some cases, between −1 and 1. This is useful for NNs as the values in the nodes work best when within this number range. When an NN has many layers, and the values go through the calculation for each layer, shown in equation 2.10, it is susceptable to enlarging to infinity or going to an extremely small value which computers cannot handle.

There are many activation functions, with the most popular ones listed from equation 2.11 - 2.15. The ReLU (Rectified Linear Unit) and Leaky ReLU functions have gained major popularity recently, however a softmax function is still widely used at the output of the NN, with the ReLU being the main choice between the layers.

Sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{2.11}$$

ReLU:

$$ReLU(x) = max(0, x), \tag{2.12}$$

Leaky ReLU:

$$ReLU_{leaky}(x) = \begin{cases} x, & \text{if } x > 0, \\ 0.01x, & \text{otherwise.} \end{cases} \tag{2.13}$$

Softmax:

$$S(x_i) \;=\; \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.14}$$

Tanh:

$$f(x) \;=\; \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{2.15}$$

2.4.2 Backpropagation

In order for a neural network to learn a given task, the network has to know how to evaluate itself. For this we have an error that the network uses called the cost function, $J(\theta)$. It is a function used to represent the difference between what the network predicted and the actual or labelled value. The larger the cost function, the worse the network is at performing that task. The goal is to minimise the value of the cost function and that is the task of backpropagation.

Backpropagation effectively determines which weights and biases of the network to change and by how much in order to minimise the given cost function. It achieves this by propagating the gradient of the cost function backwards through the network, hence the name backpropagation.

2.4.3 Architecture

There are multiple types of architectures of NNs, all of which have different use cases. The architecture depicts the number of layers, the number nodes in each layer, the input configuration and different aspects of other types of networks. Although Figure 2.5 shows the general form of a fully connected feed-forward neural network, it can take the form of more complicated structures or architectures. These include convolutional neural networks (CNNs), which are useful for scanning images or multi-dimensional matrices and recurrent neural networks (RNNs), which are useful for natural language processing (NLP) as it is able to make use of information of previous time events.

A convolutional neural network is better than a fully connected neural network at extracting features from a matrix or an image, which is a matrix of RGB pixel values. Instead of simply flattening the matrix to have an input layer such as the one shown in Figure 2.5, the data in the matrix is kept in the same relative location as a kernel or filter passes over it. This kernel, usually 3×3 in size, shifts over the input matrix by a chosen stride length and performs the mathematical operation of convolution. The goal of this convolutional layer is to extract high level features from the input matrix [18]. Between each convolutional layer is normally a max-pooling layer. This is done to downsample the matrix with the main goal of reducing parameters and computation.

2.4.4 Optimisation algorithms

Optimisation algorithms are methods used for backpropagation. There are many types of optimisation algorithms with all having advantages and disadvantages. One of the common optimisation algorithms is gradient descent due to its simplicity, while others include Adam, RMSProp and AdaGrad.

Another method for accelerating the learning process is to use batch updates. Mini-batch updates can use gradient descent to update the model for a batch of n training examples. This improves training as the gradient calculated is more accurate based on the use of a larger data set being used in each batch as opposed to a single training example.

2.4.5 Regularisation and normalisation

In order to stop the model from overfitting on the training data, one can use a technique known as regularisation. This improves the generalising capabilities of the model, and therefore allows it to predict more accurately on previously unseen data. Dropout is a regularisation technique, where random nodes are not activated during a full pass of the learning process. This allows the model to not only rely on specific nodes and creates a broader knowledge of the task which might improve generalisation.

Normalisation is usually used at the output of a NN. It scales the output to values between 0 and 1 which can be useful for the error function to remain within a certain range.

2.4.6 Deep learning

Deep learning is simply a term to illustrate that the network used has many layers and hence referred to as ‘deep’. As computational power increases, the process of deep learning becomes easier and more achievable.

2.5 Work related specifically to our approach

2.5.1 Dojo learning

The word *dojo* is a Japanese term primarily used in martial arts, meaning ‘place of the way’. It refers to a space of immersive training or learning, and the term has caught on in the field of software development. A coding dojo refers to a place for groups of programmers to practice their coding skills. The term dojo learning is chosen to define the style of learning our agent undertakes, as it learns different skills in independent and isolated training environments and uses the learnt skills in a separate environment. The method of dojo learning is further explained in the next chapter.

2.5.2 Hierarchical task network

An HTN is an approach to automated planning in AI. It mainly consists of three levels of tasks, but can contain up to N levels if the situation requires. The first level is the primitive actions or tasks. This is the building blocks of all tasks to follow. Translated to Minecraft, this is the actions being done every tick (one time step in Minecraft), for example to move forward. The next level is the compound tasks. This is a set or combination of primitive actions in some specified order to create a more intuitive task. In Minecraft, this could be to move two blocks forward. The last level is a goal or achievement task. This is the highest level of a task that can be performed autonomously and ensures that a goal is achieved before declaring the task complete. Translated to the Minecraft environment this would be to move to a specific location.

This idea of an HTN, which bears a resemblance to symbolic task acquisition [19], is beneficial when using a DQN in Minecraft. In order to make training and running the RL model a more efficient process in Minecraft, we could use an HTN. Instead of running through the NN to decide on an action every tick, it could be used every time the agent has finished executing a compound action. This would also be more intuitive to humans. Instead of learning a different action every tick, as the game state does not change much between ticks, it could learn the next compound action once it has fully completed the previous one. However, a disadvantage of an HTN is that the primitives, the compound task and the goal need to be well defined, which might be limiting in the world of Minecraft.

2.5.3 Transfer learning

Learning a new model from scratch can be time consuming and often unnecessary. The idea of transfer learning utilises the knowledge that the two models might share similar feature extractions or primitive layers in the NN, it might even be identical. Let us use an example of an MNIST digit identifier. With a convolutional layer, as described in Section 2.4.3, the filters might pick up edges and curves based on these numbers. Now suppose a new model is created to identify the first ten letters of the Greek alphabet. There is a high possibility the filters will learn the same sort of primitive lines and curves, therefore instead of retraining the new model from a blank slate, we transfer the first layer of filters to speed up the training process of the new model.

2.5.4 Curriculum learning

Curriculum learning is a special case of transfer learning in which the agent learns smaller simpler tasks and gradually builds up complexity in tasks in order to increase the performance or learning speed of a more complex task [20]. This method of learning derives directly from the human education system.

One problem with curriculum learning is the agent has a high chance of forgetting previously learnt skills. If, for example, the skill for learning how to tie a shoelace is transferred to a model which now learns how to walk, the agent might forget how to tie the shoelace. This forgetful nature is due to the model learning a new skill in a new environment on the same neural network and overwriting the previously learnt skill.

2.5.5 Options framework

The options framework stems from SMDPs, explained in Section 2.2.1, in which the transition from one state to another has a certain duration and is not instant such as MDPs. Options refer to the combination of primitive actions which may have an extended duration. It consists of a policy (π), a terminating condition (β) and an initiation set (I) [21]. Once an option is chosen, and the state is present in the initiation set, the actions of the agent are decided by that option's policy, and terminates after a reaching a terminating condition, often being a specified time duration. Thereafter, a new option is chosen, and the agent acts according to that policy, and so on, until the task is complete.

2.5.6 Limbic system

A similar idea to dojo learning in the field of psychology is the limbic system. This is a biological system to do with the emotional response in the brain. The main system comprises of the hypothalamus, amygdala, hippocampus and the thalamus. This system learns in a similar way to the dojo learning system we are investigating, by learning a reaction independently of the rest of the environment, and bringing the knowledge together when required [22].

2.6 Implementation and software

2.6.1 Machine learning libraries

There are many machine learning libraries to use when implementing an ML approach. Instead of creating a library of functions that can handle matrix manipulation efficiently and quickly, we simply use what is already developed, available and used in industry and other research fields.

An ML library is a programming library that can do matrix calculations efficiently and has NN functionality of everything discussed in Section 2.4. Many of them are high level libraries and require minimal knowledge of ML and the mathematical formulas used. This is one of the criticisms of these types of libraries, it makes it easy for the developer, but does not give them full control – unless it is built from source.

TensorFlow, which is developed by Google [23], is our library of choice. However, Keras, Theano and PyTorch are other available libraries. TensorFlow is the library most widely used in the industry and is relatively new compared to others. It is also available in many programming languages, including Python, Java and JavaScript. PyTorch, and recently TensorFlow 2.0, has good debugging functionality and is great for research purposes. Keras has a high level API, and is based on other libraries, whereas Theano is not as popular as it once was. Therefore TensorFlow is chosen based on popularity and the availability in multiple programming languages, as Minecraft is Java based.

2.6.2 Minecraft environment

Many of the current breakthroughs in ML and AI have come from the gaming world. This is no coincidence. The gaming world offers the developer a chance to encounter problems with limited amount of controls and variables. This shortens the length of time needed for simulation while still being able to optimise gaming scenarios.

Minecraft was created by the founder of Mojang AB, Markus "Notch" Persson, in 2011. It is a sandbox construction game which allows the player to roam freely and create anything at will [24]. It is a game which involves a player roaming a randomly and procedurally generated world, building structures and creating artwork. Technically speaking there is an ‘end goal’ to the game, which is accomplished by defeating the ender dragon. But few people strive to accomplish this as Minecraft has so much more to offer when creating a new world. It can be played in single player, multiplayer and creative mode, the latter of which allows the player to have unlimited resources, fly, and create anything they want. The main mode is survival, which has one main objective – to survive. During the day, the player has to hurry and gather resources in order to survive the monster infested night time. The day-night cycle is 20 minutes in total – 10 for day time and 10 for night [24].

For our purposes, it is only necessary to understand the basics of survival as the long-term goal of this research is for an agent to learn how to survive the night in Minecraft using RL techniques.

Minecraft is chosen for its simplicity of movement, relatively simple interface and grid-like environment. The crafting aspect allows for a large variety of items to be created and could pose as a problem since there are many choices. For humans, having too many choices is not always a positive, and having too many actions for an RL agent can result in a sub-optimal performance unless managed correctly.

A human in Minecraft, shown in Figure 2.6a measures in at 1.8 m, with the movement speeds found in Table 2.1. It is easy to conceptualise one meter in the virtual game world as it corresponds to one block as shown in Figure 2.6b. The time in Minecraft is also straightforward to convert between the real world and the virtual world. Dividing the real world time by 72 will calculate virtual world time and using this method it is simple to calculate the day-night cycle time shown in the following equation.

(a) Human in Minecraft (Steve). (b) One block in Minecraft.

Figure 2.6: A Minecraft human and block. Minecraft has a variety of blocks, which are not all the same size.

$$\begin{aligned}
\text{Day}_{\text{real}} &= 24 \text{ hours} \\
&= 24 \times 60 \text{ minutes} \\
&= 1440 \text{ minutes} \\
\text{Day}_{\text{virtual}} &= 1440 \div 72 \text{ minutes} \\
&= 20 \text{ minutes}
\end{aligned}$$

The world of Minecraft is split up into different dimensions which can be traversed by the use of a portal. The Overworld, shown in Figure 2.7a, is the dimension in which every player spawns and begins their Minecraft journey [24]. Other dimensions include the Nether and the End. These dimensions are not important for our purposes and hence will not be elaborated on any further.

The Overworld is subdivided into biomes which completely determine all the aspects of the virtual world, with the desert having lots of sand, the forest having an abundance of trees, etc. [24]

A chunk is a section of the map measuring 16 x 16 x 256 blocks, with a total of 65536 blocks, and is a way of splitting up the world into more manageable pieces. The render distance uses a scale of chunks and the player decides how many chunks they want to render [24]. The render distance might affect an algorithm attempting the use the pixels displayed on the screen as an input to a neural network.

The coordinate system in Minecraft has the x-axis and z-axis in the horizontal plane depicting the negative to positive direction as West to East and North to South respectively. The y-axis sits in the vertical plane with upwards

(a) The Overworld dimension in Minecraft.

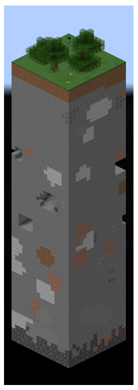

(b) One isolated Chunk in Minecraft.

Figure 2.7: The Minecraft Overworld dimension and one chunk of its landscape.

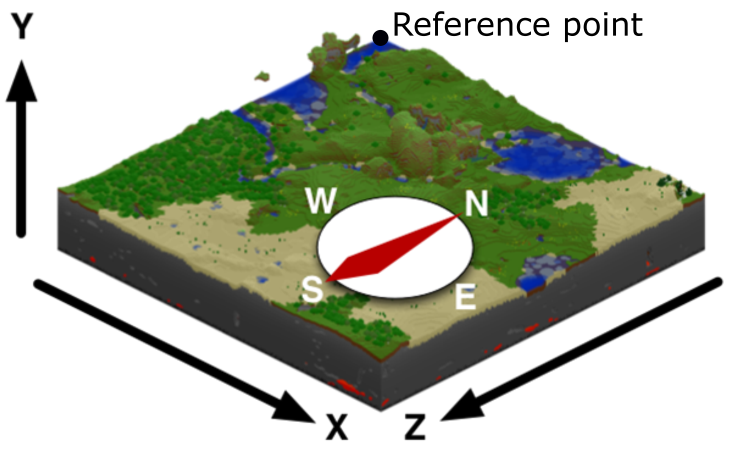


Figure 2.8: A representation of the coordinate system within Minecraft.

representing the positive direction. The reference point for each block is the most negative 3-D coordinate. This is visually represented in Figure 2.8 with the North-West corner being the reference point.

2.6.3 A* algorithm

Within Minecraft, the mobs need a method of locating the player and moving towards them. In our experiments, we create zombies that need a method of finding the agent in the environment and for this we use the A* (pronounced A-star) pathfinding algorithm. There are a many pathfinding or search algorithms in graph theory, such as breadth-first search, Dijkstra, and others, but the A* algorithm is one of the more efficient algorithms which is useful for computing many times within an episode. Without going into any detail, the A* algorithm uses heuristics to find the optimal best guess for the next step

Table 2.1: A summary of the measurements in Minecraft.

Description	**Measurement**
1 block	1 m
1 day-night cycle	20 minutes
Walking speed	4.317 m/s
Sprinting speed	5.612 m/s
Sneaking speed	1.295 m/s
Jumping height	1.252 m
Chunk size	16x16x256 m
Human height (standing)	1.8 m
Human height (sneaking)	1.65 m

with the hope of finding the target in the least number of steps [25].

2.6.4 Modifications and source code

With Minecraft being the popular game it is, creators and game developers want to add their own spin on it and modify the way they interact with the world.

A mod is a modification done on the Minecraft client side and is typically used to make the world more interesting and fun to play in, while a plugin is a modification on the server side and is typically used to make the game run smoothly and more efficiently. These are typical uses but not the rule, as mods, such as Optifine, are used to optimise the game, and plugins can also be used to make the game more fun, or modify the rules of the server for a Minecraft minigame.

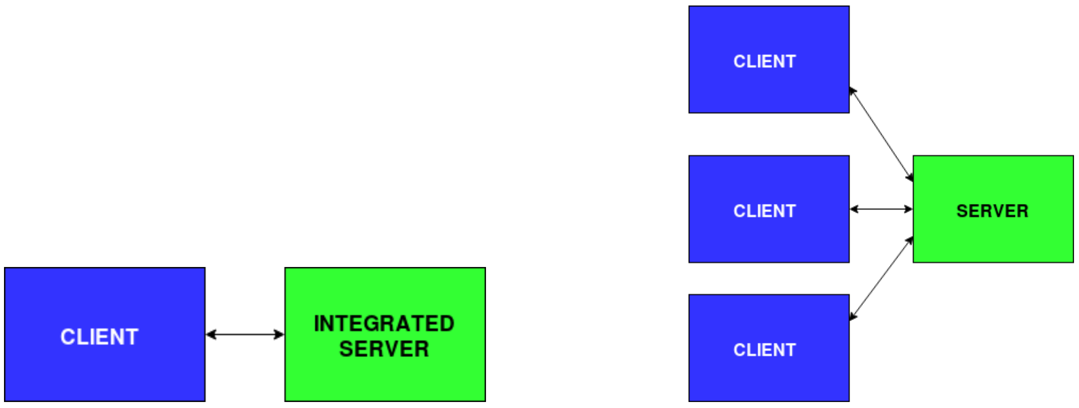
The main difference between a mod and a plugin is that a player can join a server which has a specific plugin without having to install anything on the client side, whereas to join a server with a mod, the player will need the mod installed on the client side for it to take effect.

This terminology only holds in Minecraft development, as the term mod, plugin, add-on, etc. are all interchangeable in software. This distinction in Minecraft is simply to differentiate between the two at a quick glance [26].

The three major modification platforms are Forge, Spigot, and the Minecraft coder pack (MCP). Forge is a mod tool which enables you to create a mod for the client side of Minecraft. Spigot, or Bukkit, is similar to Forge but focuses

Table 2.2: Pros and cons of the various modding tools for Minecraft.

	Forge	**Spigot**	**MCP**
Pros	Client side mod	Server side mod	Can change the source code
Cons	Cannot change source code	Cannot change source code	Not easily distributable



(a) Single player configuration (b) Multiplayer configuration

Figure 2.9: The client server relationship in Minecraft when in single player mode and multiplayer mode.

on the server side. The MCP is the decompiled source code of Minecraft and allows one to delve into the source code and alter it in any way. Table 2.2 summarises the different modding tools.

Client and server

There are four main terms used when dealing with clients and servers in Minecraft. *Physical server*, also known as the dedicated server, is a physical server out in the real world which runs the server side code of Minecraft. *Physical client*, is the client application of Minecraft, and runs on the players computer. A *logical server* is the part of the code which is responsible for all game logic, mobs and world updates. The *logical client* is the part of the code which reads the input controls from the player and relays them to the logical server [27].

The client-server connection is always present regardless of playing in single player or multiplayer mode. Figure 2.9 depicts the relationship between clients and servers in Minecraft.

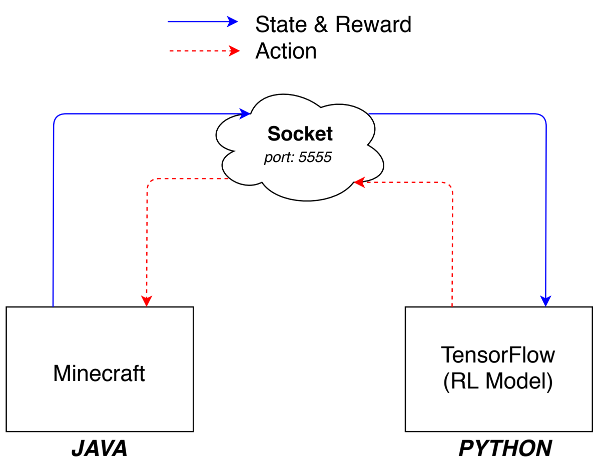


Figure 2.10: A socket server-client and Java-Python communication diagram.

Ticks

A tick is essentially one time step in Minecraft. By default the Minecraft server runs at 20 ticks per second, with 24000 ticks per Minecraft day. In order to train a Minecraft RL agent, the more times the agent can simulate a day's events the more efficient the learning process. One limit for this is the rate at which the Minecraft server runs, also known as the tick rate. Apart from the tick rate, the other limitation for simulation time is the computing hardware used for the simulations, and the efficiency of the learning algorithm.

Sockets

A socket is a connection tool that can be used to communicate between any two hosts or applications, in this case Java and Python. It creates a simple local server on localhost (IPv4 address of 127.0.0.1) and communicates via the same port. This is useful as Minecraft is Java based and most machine learning algorithms are well supported and easy to implement in Python. The only information that needs to be transferred is the state and reward from Java to Python, and the action taken from Python to Java. This is depicted in Figure 2.10.

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are two different methods of sending data over a network or through sockets. UDP is unreliable at times, and can lose data when sending across a network. To ensure that no packets are lost, TCP is a better option and is used in our socket network.

2.6.5 Synchronising code

When creating sockets and having Python and Java communicate with each other, Minecraft in Java runs at a certain rate and the socket communicates at a certain rate. When increasing the rate at which Minecraft runs using a tick rate changer mod, the Minecraft code runs significantly faster than the sockets that are used to communicate between the two languages. When the agent completes an action, it needs to send the state through to Python and receive an action back and execute it. The two rates need to be synchronised when running to minimise errors with timing.

In order to synchronise the code, the `synchronize` keyword in Java can be used with a lock on an object. The major problem is when `synchronize` is used on the main thread of code, the server thread is still running and when the main thread is notified, it simply updates the display with what happened in the background in the server thread. When using the `wait()` function on the server thread, which is where all the game logic resides, when `notify()` is used Minecraft realises that the client and server are out of sync, and simply updates the client to ‘catch up’ with the server by skipping ticks and it makes the `wait() notify()` technique null and void.

A method of pausing Minecraft while the game is in focus is possible. Pausing the game of Minecraft stops all threads from continuing the game logic. This is done by clicking the escape (ESC) key virtually using a Robot Object in Java, essentially using software to click buttons on the keyboard which is bound to actions in Minecraft. This method can synchronise Minecraft with the socket network with minor impact on the overall performance.

2.7 Chapter summary

In this chapter we presented an overview of the fundamentals of reinforcement learning and neural networks. In particular the value-based learning methods Q-learning and deep Q-learning with a DQN were discussed.

We introduced the basic concepts of Minecraft, the virtual environment in which we train our RL agents. We discussed the issues of how the deep learning software (which is Python based) will interface with the Minecraft virtual environment. Lastly we discussed different aspects related to learning in virtual environments.

Chapter 3

Approach

In this chapter we discuss the approach taken to determine the experiments and experimental setup, discussed in the next chapter. We first make a brief note of the snake experiment used to learn about the Q-learning algorithm, and the Python-Java environment setup before the use of Project Malmo. Lastly we discuss the final approach to the experiment setup, and the decisions taken to arrive at that point.

Based on the ideas of the previous chapter, Chapter 2, there are numerous ways to approach our goal of having an agent perform well in a complex environment. Reinforcement learning methods are favoured over inverse reinforcement learning as gathering real world data would not only prove challenging and time consuming, but would limit the success of the agent in the long run as it might be learning from sub-optimal policies by humans.

The goal of this chapter is to arrive at an approach for testing dojo learning. We take a step by step method of building up the complexity of a model by starting with the simple game of Snake.

3.1 Snake – a Python minigame

Before attempting to solve the Minecraft environment using RL, a snake game is developed in Python using the PyGame library [28] in order to run tests on Q-learning.

Snake is a classic game where the player takes control of the head of a snake and directs it to some food in the world using directional buttons, usually four inputs (up, down, left and right). As the snake consumes the food, it grows its tail by one unit length. The game is over when the snake bumps

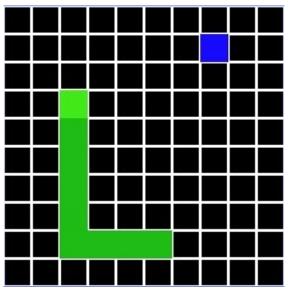


Figure 3.1: A snake minigame in Python created using the PyGame library.

into itself or into an obstacle and the player is victorious when all the blocks are occupied by the snake's body.

As discussed in Section 2.3.1, Q-learning has a Q matrix of dimensions *states* × *actions*. Without a tail and solely with the head of the snake gathering food in a ten by ten grid, the number of states is quite large at 9900. This number is calculated by the 100 possible positions for the snake's head and 100 positions for the piece of food that work out to 10000 possible arrangements, noting that the food cannot be within the same block as the head. If one includes the tail, the number of states grows to be a massive number. With the knowledge that we need to store a matrix with the number of states as the rows, using Q-learning as a lookup table is inefficient when dealing with this snake game, but it is a good place to start.

3.1.1 Objective and motivation

The objective for this minigame is to understand the fundamentals of Q-learning and the ML libraries, namely TensorFlow, available before delving into the complex world of Minecraft.

The games Snake and Minecraft are similar in that they are both grid worlds, with Snake being discrete and Minecraft being continuous. If the Minecraft controls were changed to discrete, they would have similar approaches to solve the RL environment. The idea behind using Snake is the ability to slowly grow the complexity of the method used to solve Snake, and grow the complexity of the environment of Snake to mimic Minecraft. Once a decent RL method is created and is functional in a simple game, we transfer the method over to Minecraft.

3.1.2 Q-learning table

A simple Q-learning algorithm was implemented as a baseline. Due to the large state space in a simple snake game with a grid size of 10×10 units, the game is simplified to just the head and a piece of food. This resulted in a total of approximately 10000 states (s). The number of possible actions (a) is 4, with up, down, left and right. This results in a Q matrix of dimensions (10000×4).

The rewards are -1 for every move, forcing the snake to reach the food in the shortest time, and a reward of 100 for when the head reaches the food. Since there are no obstacles or tail, there is no crashing penalty.

A time limit of 500 time steps for each episode was given, and the test ran using the ϵ-greedy policy for one million episodes with the following hyperparameters, $\alpha = 0.3$ (learning rate), $\gamma = 0.99$ (discount factor), $\epsilon = 0.1$ (probabilty of exploring a random action). This test was successful and set the benchmark for the later experiments. The code for this is shown in Listing 3.1.

3.1.3 Action space dilemma

Initially when the snake game was created it had 4 possible actions, up, down, left and right. This would seem intuitive because of the grid world nature of the game. However, when the snake begins growing a tail, 'backwards' is no longer an option.

When playing the game with the tail functionality enabled the player could go backwards and kill itself. Some code was implemented that made it impossible to go backwards and if the player chose the backwards action it would effectively do nothing, i.e. continue going forward. This created an imbalance when exploring as the agent would go forward 50% of the time and left or right, with respect to the snake's current direction, the remaining time.

When the snake game changed to have three possible actions of forward, left and right it was able to navigate the world with just these three actions, but there was a problem that was not easily noticable. The state consisted of the snake's position and the food's position. With this, the snake never had any idea which direction it was facing. This meant that with the same action in the same state, there was not a 100% certainty that it would transition to the same next state. This gives a lot of unforeseen bugs and is not a good method

```python
def main():
    RENDER_TO_SCREEN = False
    MAX_TIME = 500
    env = Environment(True, rate = 10, render = RENDER_TO_SCREEN)
    if RENDER_TO_SCREEN:
        env.prerender()
    Q = np.zeros((env.number_of_states(), env.number_of_actions()))
    alpha = 0.3  # learning rate
    gamma = 0.99  # discount factor
    epsilon = 0.1  # probability to choose random action over best action
    for episode in range(1000000):
        state, mytime = env.reset()
        done = False
        if episode % 10000 == 0:
            print(episode)
        while not done:
            if RENDER_TO_SCREEN:
                env.render()
            if np.random.rand() <= epsilon:
                action = env.sample()
            else:
                action = np.argmax(Q[env.state_index(state)])
            new_state, reward, done, myTime = env.step(action)
            Q[env.state_index(state), action] += alpha * (reward + gamma *
                np.max(Q[env.state_index(new_state)]) - Q[env.state_index(state), action])
            state = new_state
            if myTime == MAX_TIME:
                done = True
    np.savetxt("Qmatrix_random_start.txt", Q.astype(np.float), fmt='%f', delimiter = " ")
```

Listing 3.1: Q-learning main() function.

of implementing a Q-learning algorithm, as the agent can never learn what the best action is in a given state.

The actions are increased to five possible actions, up, down, left, right and do nothing. This allows the snake to always transition to the same next state regardless of its current direction, and the position of all the tail segments are added into the state which can show the snake which direction is ‘backwards’. However, the Q-value lookup table was too large for this idea, and therefore the function approximator is used. This is a single layer neural network, with no activation function.

3.1.4 Function approximator

Between Q-learning and deep Q networks reside linear function approximators. A linear approximator is an NN with one hidden layer and no activation function, $input \rightarrow f(input) \rightarrow output$.

There are various NN architectures, discussed in Section 2.4, which have different objectives, strengths and weaknesses. The architecture chosen for this design has an input layer of 300 nodes, comprising of three one-hot-type vectors flattened to represent the grid and the position of the head, food and tail respectively, one hidden layer of 128 nodes, and an output layer or five actions. Whichever output node had the highest value would be the most favourable action in that current state.

This method achieved the goal on a smaller grid size of 6 x 6. But the agent could not grow its tail more than four or five on a grid size larger than 6 x 6, therefore it was time to move to a DQN, in order to learn a more complex, non-linear function.

3.1.5 Deep neural network

With the increase in the number of hidden layers to a total of three and the creation of a deep neural network (DNN), the model works well on a 6 x 6 grid up to an 8 x 8 grid, with the hyperparameters set to 0.01 learning rate (α), 0.99 discount factor (γ) and epsilon following a linear function, shown in equation 3.1, decreasing from 0.9 to 0.1 across 50% of the training. It is evident that this small model was not working well for larger grid sizes. It was struggling to find the food in a decent time, more often than not converging on a sub-optimal local minimum and simply oscillating back and forth between two locations on the grid. This could be due to the input layer style as only one of the many input nodes represents the food block, while there are many input nodes that represent a vacant space. The training function is shown in Listing 3.2.

Hyperparameter functions

The randomness in the learning process comes from the epsilon (ϵ) value. This determines the exploration-exploitation ratio, as discussed in Section 2.2.6. One method that seems to increase performance is to have a function for the ϵ variable and decrease it over time as the agent learns. The high random value initially is to allow the agent to visit as many states as possible and let the agent have an idea of the environment and where the food is located, and slowly decrease the value to rely more on exploitation.

$$\epsilon = \begin{cases} \dfrac{-0.9}{0.5 \times episode_{total}} \times episode, & \text{if } episodes \leq 50\%\ episode_{total}, \\ 0.1, & \text{otherwise.} \end{cases} \tag{3.1}$$

```python

def train(self, model, sess):

    memory = self.memory[self.memCntr % self.memSize-1]

    output_vector = sess.run(model.q_values, feed_dict={model.input: memory[0]})

    if memory[3]:
        output_vector[:,memory[1]] = memory[2]
    else:
        # Gathering the now current state's action-value vector
        y_prime = sess.run(model.q_values, feed_dict={model.input: memory[4]})

        # Equation for training
        maxq = sess.run(model.y_prime_max, feed_dict={model.actions: y_prime})

        # RL (Bellman) Equation
        output_vector[:,memory[1]] = memory[2] + (self.GAMMA * maxq)

    _, e = sess.run([model.optimizer, model.error],
                    feed_dict={model.input: memory[0], model.actions: output_vector})

    return output_vector, e
```

Listing 3.2: DQN train() function using TensorFlow.

Normalisation

When dealing with a DNN, it is important to normalise the output data for an RL algorithm. In order for the agent to learn effectively and change action values quicker, it would be beneficial to have the action values within the same range – in this case, between 0 and 1 – with small values referring to a bad action and higher values referring to the possible good actions to take in the given state. The range of output values can be quite large if not normalised and that would affect the stability of the Q-learning equation.

3.1.6 Convolutional neural network

In order for this method to scale into the world of Minecraft, the input layers of the neural network are reworked to a CNN. The snake game consists of three layers in the CNN. Each layer is a 2-D one-hot array of positions of interest in the 10 x 10 grid – one layer representing the head of the snake, one representing the food, and the other representing the position of the tail or obstacles. These stack together to create a 3-D tensor, three layers of a 10 x 10 grid resulting in a tensor size of [10,10,3].

This tensor is the new input to the neural network with two convolutional layers, flattened to go into the two fully connected layers and finally the output layer.

3.2 Initial Minecraft environment

The Python snake game implementation is successful as it achieved its objective and helped acquire insight into Q-learning for a grid world game. In order to achieve the same success in the world of Minecraft, we train an agent in Minecraft to play a game similar to snake. The agent would spawn on a grid and given the coordinates, the agent's goal is to move to the red block. This has two large complications, one is that the action space in Minecraft is continuous and the other is that Minecraft is a Java based program while all the machine learning libraries are mainly Python based.

3.2.1 Snake in Minecraft

In order to follow a similar process to the one used in the snake game, we need to run Minecraft (Java) at the same time as we run the Python Q-learning algoritm. This is obtained by using sockets, explained in Section 2.6.4, to send the state of Minecraft in an array over to Python in order to predict the best action for the agent.

In order to optimise training time, the tick speed of Minecraft needed to be increased. This was done by using a tick changer mod mentioned in Section 2.6.4, which increases the tick speed of Minecraft on the client and server side to as fast as the hardware could handle. This was successfully changed from 20 ticks per second (TPS), which is the default, to around 2000 TPS.

Although the tick rate of Minecraft was successfully changed, the rate at which the state is sent to Python and the action sent back is not increased as the code runs on separate threads. This causes timing errors as the Python code running the model cannot inform the agent of the best action quick enough. This leads us to attempt to synchronise these two threads, the Python ML code, and the Minecraft game code.

3.2.2 Minecraft threads

Minecraft is a mesh of multiple threads all running alongside each other with the main thread, the server thread, the socket thread, and the music thread all separate. This adds a major level of complexity to Minecraft's source code.

Using the `wait()` and `notify()` functions built into every Java object, we could pause and unpause the Minecraft client successfully, but the client or main thread, does not handle the game logic of the world and when the game is unpaused, the entities in the world would simply teleport to where they were on the server thread.

The server thread is what houses the main game logic loop. This controls the positions of the entities, the blocks in the world, the health of the entities, and so on. We can pause the server thread, but when the game was unpaused, a message is displayed stating the server and client are out of sync, and the server skipped as many ticks as the client went through during the pause for the game to be synchronised again.

This makes sense in terms of gameplay. If the server lags, it simply catches up to the client and no gameplay is lost with network lag in multiplayer. However, this proved challenging to stop the game in Forge or Spigot using code to synchronise the different threads.

3.2.3 Lock step

The entire training process and running of the model relies on the synchronisation of Minecraft to the Python algorithm. This requires a mechanism known as lock step. Simply put, Minecraft sends some information to Python and it needs to wait, or be ‘locked’, and let Python run its code, or ‘step’ through its code. When Python is done, the process is mirrored on the other side, with Python sending its data to Minecraft (the sending of data normally occurs at the end of the step), and subsequently locks itself in order for Minecraft to step though its process and as a result, the program is fully synchronised.

3.2.4 Training limitation

When running the model and showing the results after the training is complete, we can run the Minecraft and Python setup easily through the socket network as this is fast enough for 20 TPS. However, when training the model a better implementation for this type of communication is necessary. The lock step method for synchronising the code on Python side and Java side worked perfectly, but is still time consuming. This lead us to use the Project Malmo, as training in Minecraft proved to be quite the challenge.

3.3 Project Malmo

The socket network and lock step method created for the communication between Python and Java had a few drawbacks which limited the training efficiency of the Minecraft world simulation. Instead of developing a communication structure between the two platforms, we use an existing one. Project Malmo is a reinforcement learning platform developed by Microsoft [29] and built on top of Minecraft. It was designed to support fundamental research in AI, mainly reinforcement learning algorithms. The platform has numerous minigame environments setup for testing various algorithms.

3.3.1 Disabling rendering

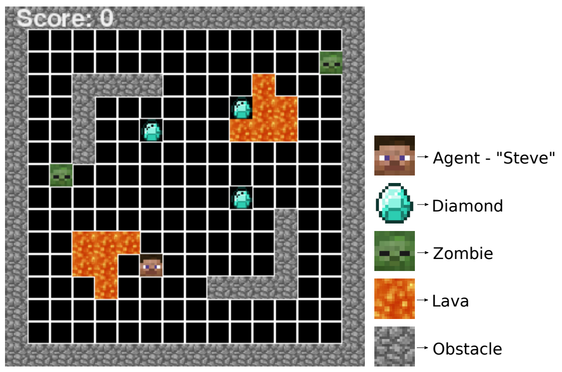
The training in Project Malmo was faster and more efficient than our own socket network communications setup, but was still significantly slower than the snake training environment which is running solely on the Python side.

In order to improve training time, Minecraft needs to run without rendering as it is a resource intensive process. We are able to run the Minecraft client without rendering it to screen using xvfb (X virtual framebuffer) and the following command in Linux, `$ xvfb-run -a -e /dev/stdout -s ’-screen 0 1400x900x24’ ./launchClient.sh`. This improved training time slightly, but would still take far too long to train for millions of episodes on our current hardware.

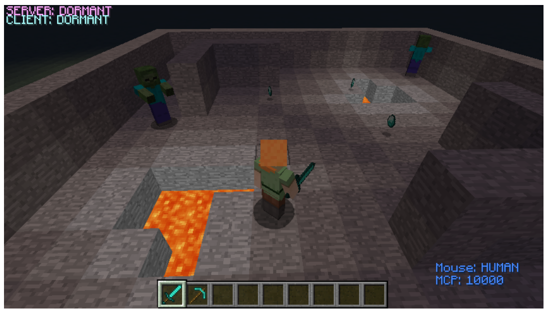
3.4 Python minigame

Although Project Malmo supports training really well, it is still resource heavy and slow in resetting the environment to the initial state at the start of each episode. With many different experiments and models that need to be run in order to test our prediction, a fast training platform is necessary to test a variety of models.

For this reason, a simple, low resource minigame was created using the Python gaming library, PyGame, to mimic the Minecraft environment. With simple discrete actions and observation information, we can train a model in the faster PyGame version, and run the model in the real Minecraft on the Project Malmo version. Figure 3.2 shows the same environment represented in the PyGame environment and Project Malmo. This environment in PyGame is available to the public on GitHub at `https://github.com/Matthew-Reynard/malmo`.



(a) Minecraft in PyGame - Python version.



(b) Minecraft in Project Malmo - Java version.

Figure 3.2: The same Minecraft environment represented in both Python's PyGame library and Microsoft's Project Malmo.

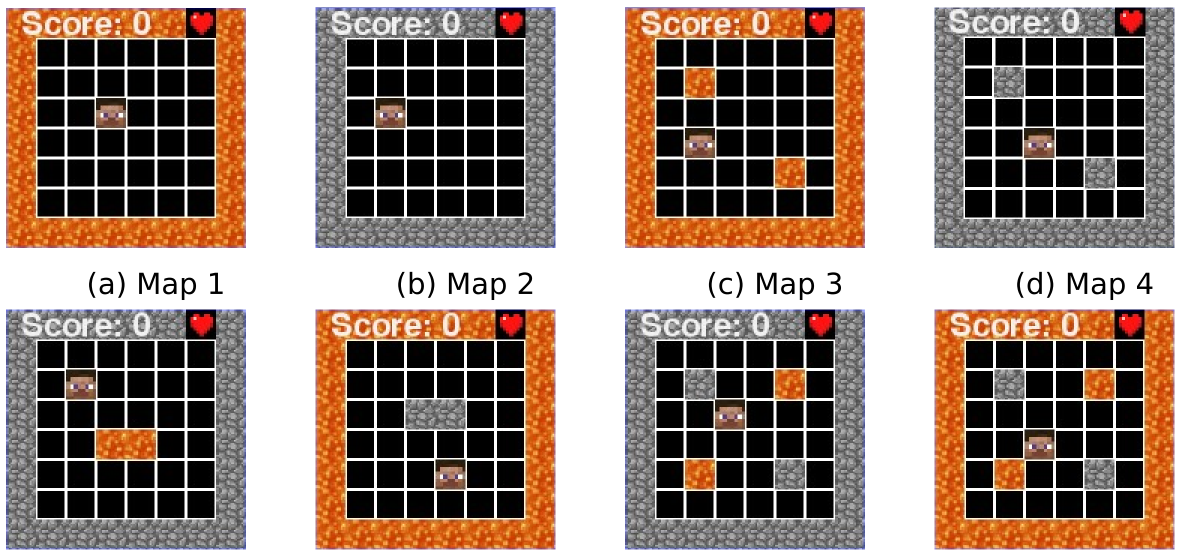
3.5 Final approach

With all the different tests complete, and various platforms and environments such as those shown in Figure 3.3 tested, we are able to create an approach to answering the research question. To recap, our goal is to test our new method of dojo learning, which poses the question whether an RL agent will achieve better performance in a challenging environment by first learning the skills needed in smaller sub-environments or by learning in the complex and challenging environment from the start.

The two networks that are compared in the experiments are that of the dojo network, and a traditional deep Q network (DQN) which will be referred to as the standard deep Q network. These methods are compared in both a simple and complex environment and are explained in the Chapter 4.

3.6 Dojo environments

The goal is to learn different skills in different isolated environments or ‘dojos’, which is similar to when a child learns different subjects at school in different classrooms. The agent will not have any knowledge of the other learned skills, and will only focus on the simple task at hand. The question we are trying to answer is whether this method of learning can be more efficient and faster than having an agent learn in a complex environment from the start. If so, how much will this improve the training and to what extent can we expand the range of skills the agent can learn.



(a) Map 1 (b) Map 2 (c) Map 3 (d) Map 4

(e) Map 5 (f) Map 6 (g) Map 7 (h) Map 8

Figure 3.3: Different 6 x 6 maps for testing environment.

3.7 Dojo network architecture

In the experiments we set out to show the feasibility of our dojo network to help an agent perform better in a complex environment. The dojo network consists of two sections which is comprised of different modules. The first section is the meta module, which decides which dojo skill the agent should reference given the current state of the environment. The second section is the different dojo modules. These modules are trained independently in a simplified environment, with each obtaining adequate results in the given sub-environments.

The specific Q network and architecture used for our experiments are simple but can easily increase in complexity if needed. We show a proof of concept here, and will explore more complex architectures and algorithms in future work, such as Dueling DQN (DDQN) and prioritised experience replay [30].

As previously mentioned, the dojo network consists of a number of dojo modules each having that same network architecture with the same five possible actions as outputs, however, these actions can also be different actions entirely. This is then combined using a meta module with the same main architecture shown in the module block in Figure 4.4 and the different dojos as its outputs. Each time a dojo is selected, the appropriate input state (i.e. a state configuration with which that dojo was trained) is then passed over to that dojo's trained module and the best action is chosen. This dojo choice occurs at every time step.

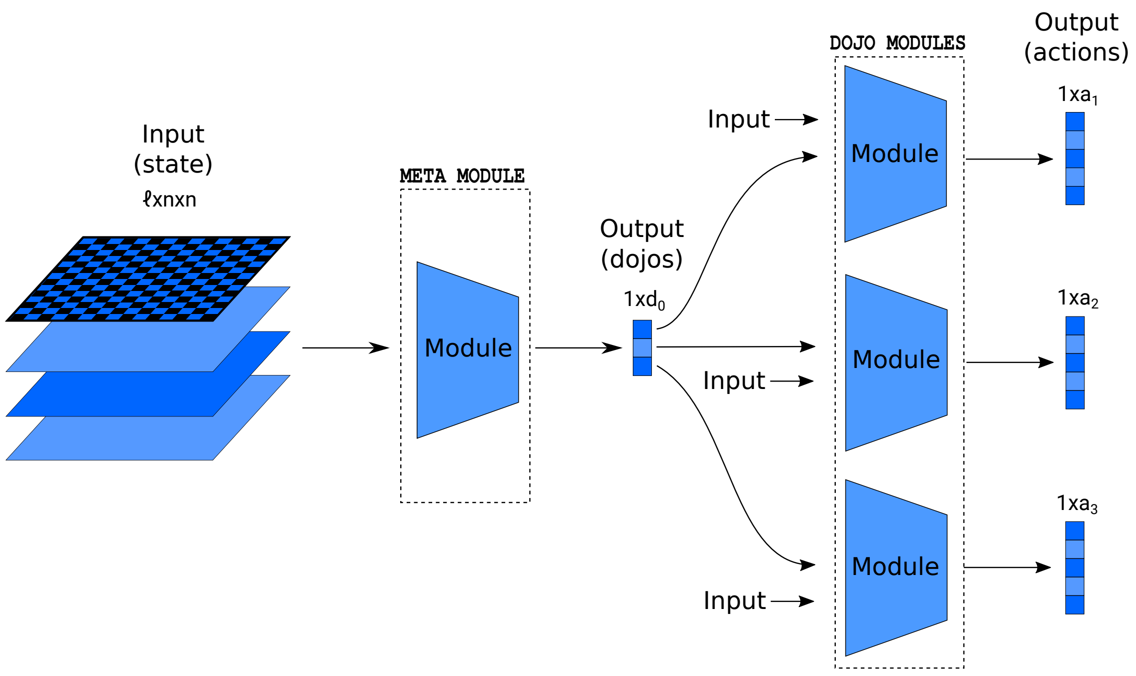


Figure 3.4: Dojo network architecture: This network has three dojo modules, each having its own set of five possible actions. The number of possible outputs for the dojo network is the number of unique actions that the dojo modules have. The meta module decides which dojo module will decide on the next action.

3.7.1 Dojo network input

The input to the dojo network can in theory be anything from the agent's coordinates to the raw RGB values of the screen, as long as the different dojo modules take in the same input with which they were trained. In our case we manually extract the positions of various blocks-of-interest (BOI) and arrange them in an easily expandable way. A binary-style grid represents the positions of the BOI, with each layer representing a different block type. The grid is fixed around the position of the agent with odd value dimensions to ensure the midpoint of the grid represents the agent.

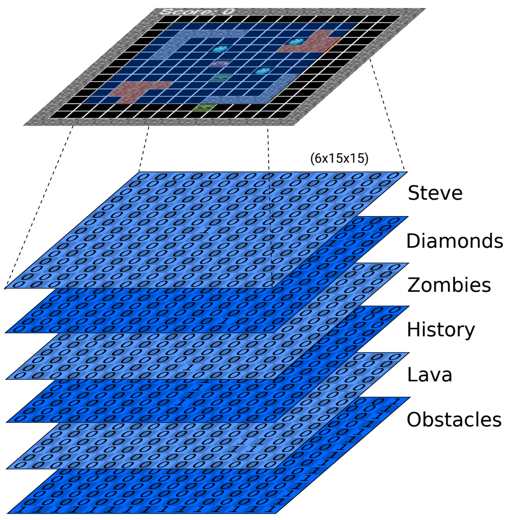


Figure 3.5: Detailed network input: Each layer of the 3-D array represents a unique block-of-interest (BOI) in the grid world environment. A ‘1’ if the BOI occupies this space and a ‘0’ otherwise. These are placed in relation to the agents position which is always represented in the center of the first layer.

3.8 Chapter summary

In summary, the main points to note in this chapter are the creation of the Python minigame environment created with PyGame in order to speed up training and still allow the model to run within the world of Minecraft, and the final approach to the experiment being the dojo network layout and architecture that will be tested in the following chapter.

Chapter 4

Experiments

In this chapter, we present the experiment conducted using the final approach laid out in the previous chapter in Section 3.5 and run the tests in different Minecraft environments. We also cover additional experiments which provide some insight into the technique of dojo learning.

4.1 Environment setup

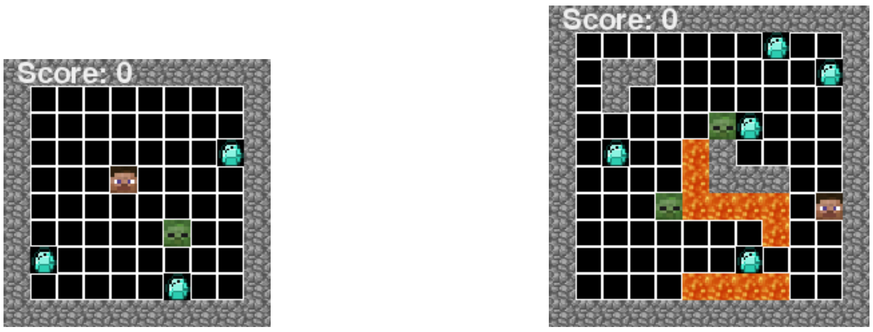
We test the dojo network in the Minecraft gaming environment as it provides a baseline for RL research as well as being able to test our network's ability to scale in complex environments.

As metioned previously, training an agent in Project Malmo is time intensive due to the platform's complexity. To speed up the research, we recreate the necessary aspects of Minecraft in a Python environment using PyGame for training, with the trained network able to transfer and run in Project Malmo in the same environmental setup, shown in Figure 3.2.

Two environments are created for these experiments, a simple environment, shown in Figure 4.1a, and a complex environment, shown in Figure 4.1b.

4.1.1 The dojos

The dojo modules, or *dojos*, shown in Figure 3.4 are trained beforehand in simpler sub-environments with just the input that affects those particular skills. In the case of collecting diamonds, the dojo is trained without a zombie present. This allows the reward function for that dojo to be refined and suited to that specific scenario. Once all the dojos are trained for 100 thousand episodes, the networks are fixed (no longer trainable) and added to the output

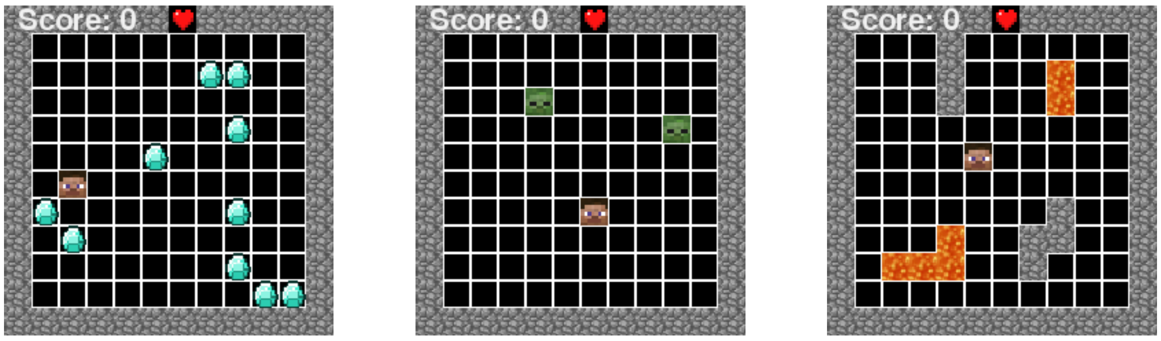


(a) Simple Environment (b) Complex Environment

Figure 4.1: The different environments tested for results to show scalability as well as simple versus complex environment behaviour.

of the meta module which will be trained for a further 100 thousand episodes.

The three different dojos used for these environments are collecting diamonds, avoiding the zombies and exploration, with each shown in Figures 4.2a, 4.2b and 4.2c respectively. An episode consists of a maximum 100 moves for the agent with ten diamonds to collect, an environment to explore and zombies that move towards the agent using the A* path finding algorithm mentioned in Section 2.6.3.



(a) Diamond dojo (b) Zombie dojo (c) Exploration dojo

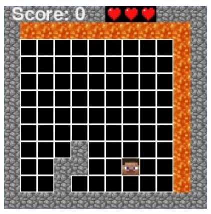
Figure 4.2: The different dojo environments in which the agent trains in this experiment.

4.1.2 Simple environment

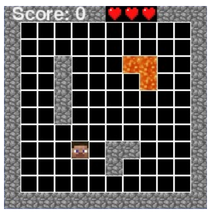
As illustrated in Figure 4.1a, the simple environment has ten diamonds to collect and one zombie to outrun. This environment was used to obtain a measure for how each model would perform in a relatively straight forward environment. It was also kept very small, to increase the simplicity. The network used in this environment also had a smaller input area than the complex environment network, with an input grid size of 9 x 9 instead of the 15 x 15 used in the complex environment.

4.1.3 Complex environment

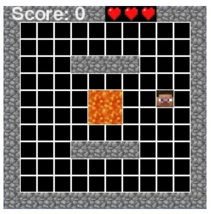
As illustrated in Figure 4.1b, the complex environment setup was a little more complicated than the simple, with ten diamonds, two zombies and the introduction of obstacles and lava to manoeuvre around. The network input for this environment had a size of 15 x 15 which allowed the agent to effectively 'see' further than that of the simple environment network, but this added an extra level of complexity as well with more information for the network to understand. The complex environment had set environment configurations or 'maps' for training, some extra maps for testing and larger maps for testing the scaling ability. All these map are shown in Figure 4.3.



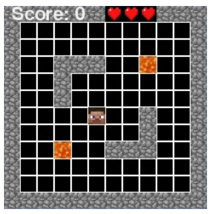
(a) Map 1



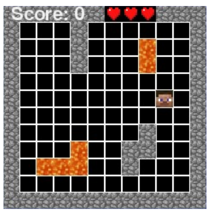
(b) Map 2



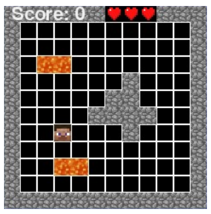
(c) Map 3



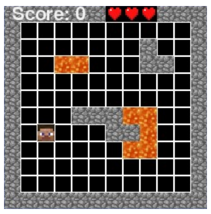
(d) Map 4



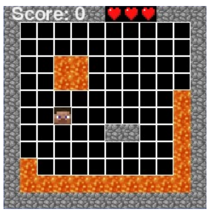
(e) Map 5



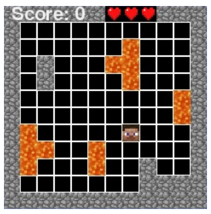
(f) Map 6



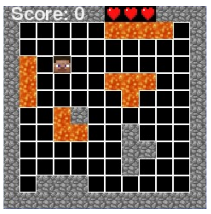
(g) Map 7



(h) Map 8



(i) Map 9



(j) Map 10

Figure 4.3: Different 10 x 10 maps used for complex environment training.

4.2 Network setup

The network's full architecture is shown in the previous chapter in Figure 3.4, with each module having an architecture as seen in Figure 4.4. It consists of two convolutional layers with two fully connected layers giving the output of five possible actions. Given that using data from raw pixels is computationally expensive, we manually extract the required features for the input state given to the network as shown in Figure 3.5. This is compiled into a one-hot positional grid that is a layered 15 x 15 grid for the complex environment and

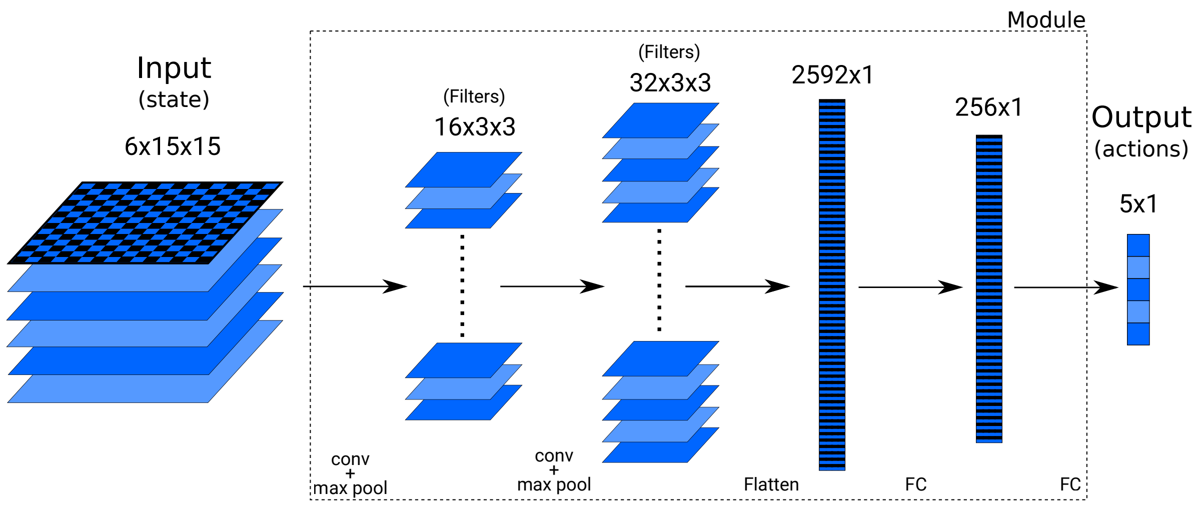


Figure 4.4: The network architecture used for the dojo modules and standard network.

a layered 9 x 9 grid for the simple environment, with the agent located in the center and points-of-interest positioned relative to it.

The dojo modules are pretrained seperately with each dojo skill learnt in the appropriate environment as shown in Figure 4.2. The choice of which dojo module is used relies on the meta module to decide based on the network's input. The meta module is only trained once the dojo modules are trained and frozen, meaning that they cannot be futher trained and their weights and biases are fixed.

For these experiments all modules are using the same network architecture, however this does not have to be the case. Each dojo module can have different architectures as well as different outputs or actions. As long as the input to each dojo module is the same format as the input that it trained with, any input will work for any dojo module. We discuss this as possible future work in Section 6.3.

4.3 Additional experiments

Based on the findings discussed in the next chapter, we conducted an additional two experiments. A 'cointoss' experiment, which kept the output of the meta module random and never pre-trained the dojo modules. This was to see that if all the weights and biases of the entire dojo network is trainable, would the results be similar to the standard network that we compare against.

The second experiment was to see the results of a zero zombie infested environment, with only the diamond dojo and exploration dojo used in the complex environment. This was compared to the standard network with the same input. These results are discussed at the end of the results section.

4.4 Chapter summary

The experiment layout is discussed in this chapter, as well as the different dojo setups (diamond, zombie and exploration) used for the experiments. The idea of simple and complex environment tests are explained, with the main difference being that of the input matrix size and the size of the environment. The simple environment does not have obstacles and the diamonds are scattered randomly in the map, while the complex environment has 10 different training environments, shown in Figure 4.3, with the diamonds scattered randomly within them. The neural network architecture for both the dojo modules and the standard DQN is shown in Figure 4.4, however, the dojo modules do not need the same architecture in the dojo network. Lastly the two additional experiments are described, the cointoss and the non-zombie environment.

Chapter 5

Results

In this chapter, we present the final results of the experiments discussed in the previous chapter. We begin with the overall results of the main experiments, to compare our dojo network to a standard DQN in the simple and complex environment setup described in Section 4.1. We then examine the training of the dojo network in more detail and explain how we arrived at the additional experiments discussed in the previous chapter in Section 4.3. Finally we present the results of the two additional experiments that were conducted, and provide a summary of the results of the experiments.

5.1 Overall results

For this experiment, the score achieved by the agent in each environment setup is the only value we use to measure the success of the different networks. However, the number of moves taken by the agent during the training and running of the model reveals a key insight as to why the standard network achieved success.

We train four networks in total, a standard DQN and a dojo network in a simple 8 x 8 environment, and another standard DQN and dojo network in a complex 10 x 10 environment with ten different environment layouts or maps shown in Figure 4.3. The complex environment training is done with a random map out of the ten chosen each training episode. The trained models are then run in different environment setups with the results of these experiments shown in Table 5.1.

The simple environment setup is an environment with ten diamonds to collect, one zombie to outrun and no obstacles or lava to avoid. For this network, the

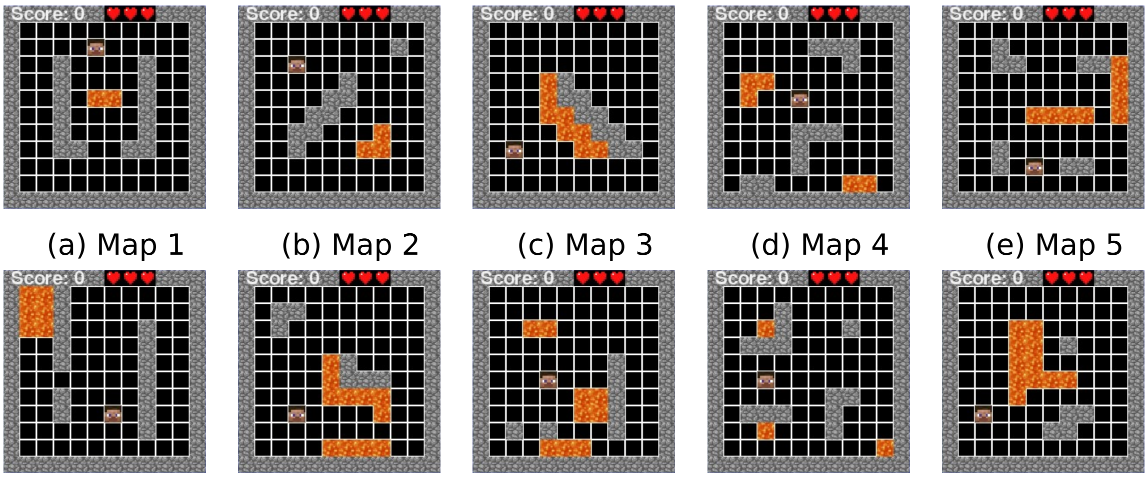
Table 5.1: Standard and dojo networks in different environments (*simple*: simple environment; *complex*: complex environment; *number*: size of grid; *new*: different environment layouts than training))

Network	Moves	Score
Standard (*simple* - 8)	27.4	7.9
Dojo (*simple* - 8)	19.9	6.2
Standard (*simple* - 16)	76.1	5.9
Dojo (*simple* - 16)	52.6	4.2
Standard (*complex* - 10)	35.2	8.1
Dojo (*complex* - 10)	30.6	7.2
Standard (*complex* - 10, new)	33.6	6.3
Dojo (*complex* - 10, new)	31.7	5.8
Standard (*complex* - 16, new)	51.3	5.5
Dojo (*complex* - 16, new)	58.0	5.2

size of the input grid is 9 x 9, representing the area around the agent which it is able to observe. The initial starting point of the diamonds, the zombie and the agent is random. The two models trained in the 8 x 8 grid simple environment are run for 10000 episodes in a 8 x 8 grid and a 16 x 16 grid simple environment. The terminating condition of the episode was either all diamonds were collected, the zombie caught the agent or the agent has taken 100 moves.

Table 5.1 shows the standard network collected more diamonds on average, i.e. had a higher score, than the dojo network in both size environments, but also taking more moves to accomplish the same task. The higher average number of moves can represent either the agent not having the goal of collecting the diamonds and just roaming the environment aimlessly, or it can represent the agent's ability to avoid the zombie for a longer period of time. With an observation size of 9 x 9, the low scores on the 16 x 16 environment is understandable, as the agent cannot see where the diamonds are located.

The complex environment setup is an environment with ten diamonds, two zombies as well as obstacles and lava placed around the environment. The initial placement of the diamonds, zombie and agent are all random with the obstacles and lava always located in the same place on a given map. One of ten environment maps are randomly chosen foreach episode of training. The two models trained in the 10 x 10 complex environment are run for 10000 episodes in three different setups, the 10 x 10 training maps, new 10 x 10



(a) Map 1 (b) Map 2 (c) Map 3 (d) Map 4 (e) Map 5

(f) Map 6 (g) Map 7 (h) Map 8 (i) Map 9 (j) Map 10

Figure 5.1: Unseen 10 x 10 maps used for complex environment testing.

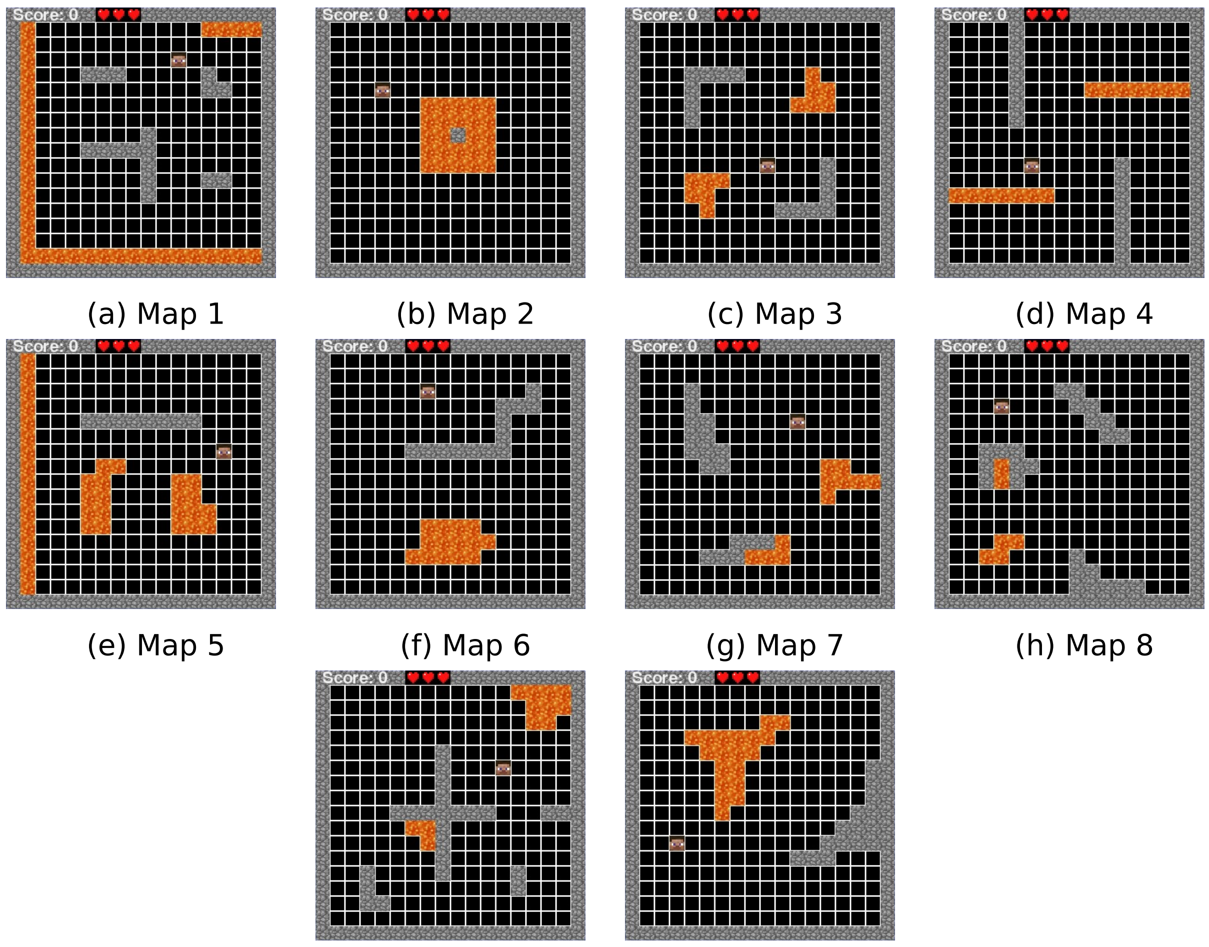
maps which the agent has never seen before, and 16 x 16 maps which were also never seen before. The new maps and the 16 x 16 maps are shown in Figures 5.1 and 5.2 respectively.

The standard DQN once again outperformed the dojo network in each of these environments by collecting more diamonds on average. This is shown in the last two rows in Table 5.1.

Having looked at the overall results, it is clear that the standard network outperforms the dojo network in these environments. We now look at the data gathered during training with some additional insight by the further training of the dojo modules after the meta module of the dojo network is trained.

5.2 Training curve

The training curves represent the average score of the agent during training after a certain number of episodes. The score represents how many diamonds the agent collected within the allocated time period. The blue dotted line in the training time graphs, such as Figure 5.3, represents the dojo network average score during each episode, and the red solid line represents the score of the standard DQN. For the training time graphs, the models were trained for 10000 episodes in the simple environment and then an additional 20000 after the dojo modules are unfrozen, or allowed to train further. The overall results, shown in Table 5.1 in the previous section, do not account for the unfrozen dojo modules. For the complex environment, the models are



(a) Map 1 (b) Map 2 (c) Map 3 (d) Map 4

(e) Map 5 (f) Map 6 (g) Map 7 (h) Map 8

(i) Map 9 (j) Map 10

Figure 5.2: Unseen 16 x 16 maps used for complex environment testing.

trained for 100000 episodes with an additional 200000 episodes after the dojo modules are unfrozen.

The first two training time graphs in Figures 5.3 and 5.4 show the results of the dojo modules being ‘blind’ to the rest of the environment as the input on which the dojo modules is trained is the exact same input to the dojo modules as used when the meta module is training. This means that the other layers in the input to dojo module were removed if the agent did not learn that dojo skill with that input layer. For example, in the case of the diamond dojo, the zombie layer represented in the input of the network is removed and the agent does not observe that zombie at all.

Although it appears promising for the dojo network within the first third of episodes in the simple environment (Figure 5.3), both the simple and complex environment networks (Figure 5.4) drop in score once the network is unfrozen at episode 10k and the dojo modules are allowed to train. The

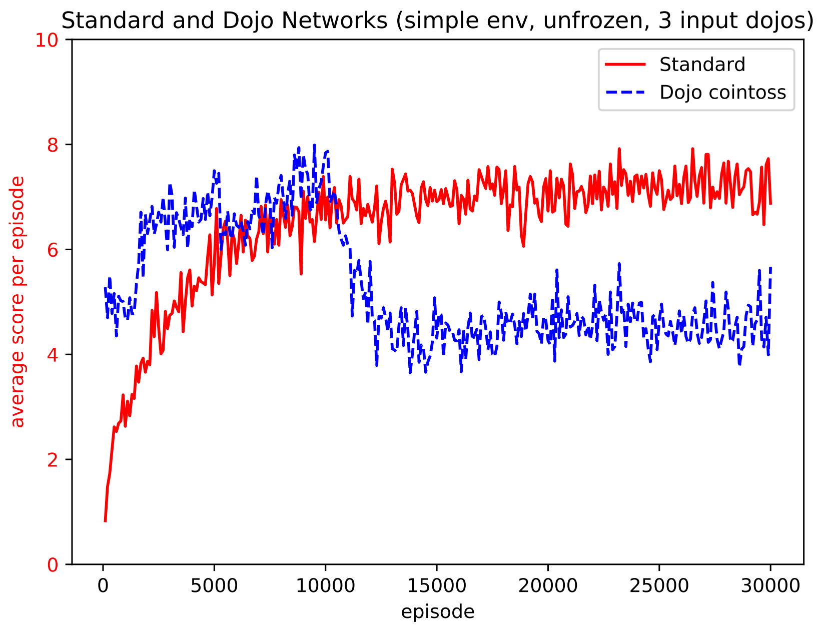


Figure 5.3: Standard and dojo networks, unfrozen at 10k, $\epsilon = 0.1$, simple environment. Dojo input states are *limited* to observations available when training dojos independently.

initial dip once unfrozen is expected as the network could have been in a local optimum, but without full awareness of the environment, the dojo network approach fails to reach the score of the standard network once unfrozen.

The next two training time graphs shown in Figures 5.5 and 5.6 show the results of the dojo networks taking in the exact same input as the meta module. This means that even if the agent does not train in a dojo with that layer, such as a zombie in a diamond dojo environment, the agent will still observe that zombie when the diamond dojo skill is chosen by the meta module. Although having low initial score for the first third of training time, once unfrozen, the score climbs to the results of the standard network, raising the question of whether the dojo modules are simply becoming three identical standard networks.

A reason for this plateau in score that can be seen in the first third of training in Figure 5.6 is due to the fact that it has a fixed dojo module as an action for the meta module and cannot make small changes to the trained dojo module once exposed to the entire complex environment. This hinders the agent's ability to adapt to the new complex environment as it has fixed skills that were trained in the dojo environment's prior.

The question posed of whether the unfrozen dojo modules in Figure 5.6

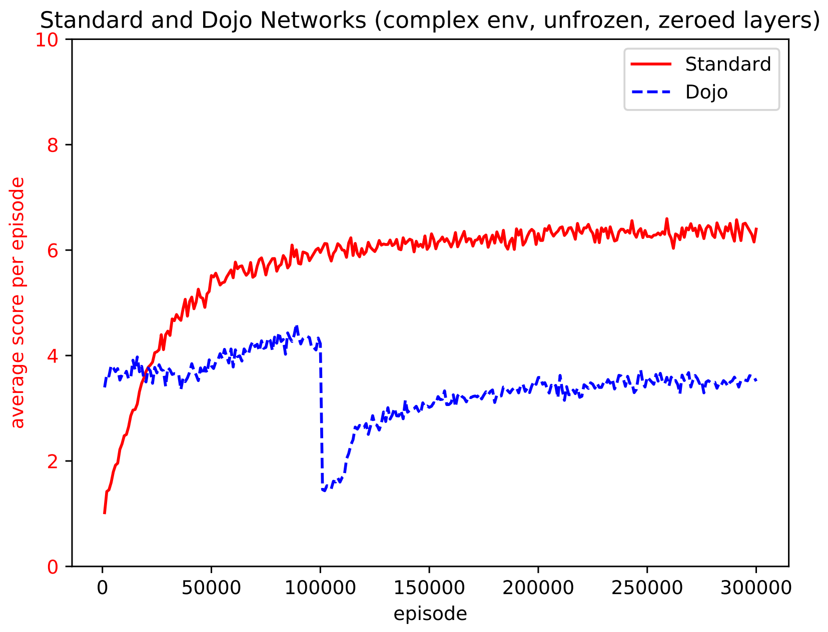


Figure 5.4: Standard and dojo networks, unfrozen at 100k, $\epsilon = 0.1$, complex environment. Dojo input states are *limited* to observations available when training dojos independently.

is simply creating three identical standard networks for our dojo modules led us to run the 'cointoss' experiment. The name is inspired by the dojo module being chosen randomly. These results are discussed in the next section.

5.3 Additional experiments

The results for the experiments are not what was expected. A deeper investigation into the training of the meta module is conducted which prompted the two additional experiments to be carried out as described in Section 4.3. One part of the investigation involved having a tally of how many times each dojo module was chosen during the training of the meta module. The results are shown as a histogram in Figure 5.7, with a surprising bias towards dojo 2 – the zombie dojo. Dojo 1 and dojo 3 are the diamond and exploration dojos respectively.

This large bias towards the zombie dojo shows that the reward system in the combined network is not balanced as the agent is focused on avoiding the zombie rather than achieving a high score and collecting diamonds. This always hinders the agent's exploration abilities as it only explores less than 10% of the time that it focuses on avoiding the zombie. The ratio between these numbers are fairly consistent between the start and end of training.

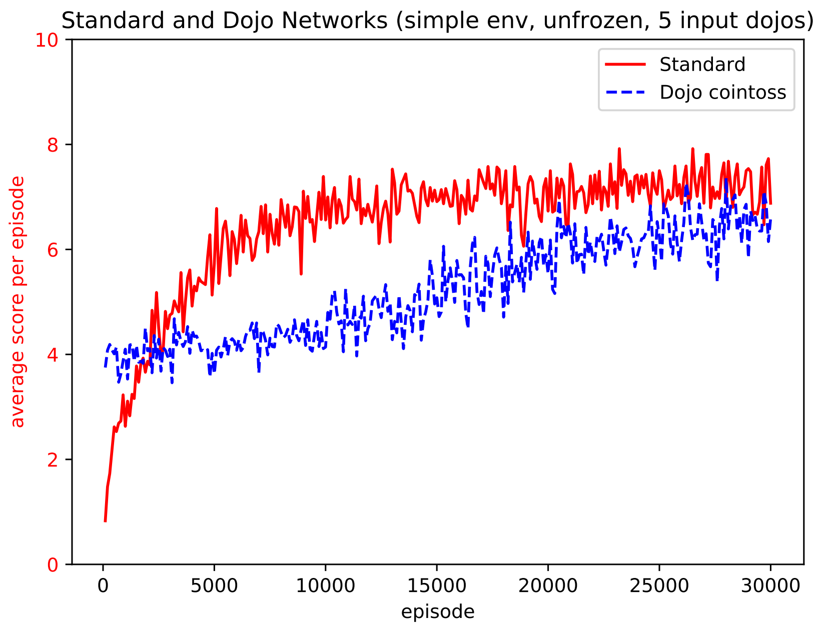


Figure 5.5: Standard and dojo networks, unfrozen at 10k, $\epsilon = 0.1$, simple environment. Dojo input states are *unlimited* and able to observe the entire environment.

We strongly expected these results, as the dojo modules have the same architecture as that of the standard network and was chosen at random which module was to train each step of the episode. It also takes longer to train based on the fact that there are three separate networks compared to one.

This prompted the second of the additional experiments where the zombies were simply taken out of the experiment all together, and we compare the two networks in a complex environment with only two dojo skills put to the test. The models in this experiment were trained in the ten training maps of the complex environment, shown in Figure 4.3. This result, with the training data shown in the graph in Figure 5.9, shows that the dojo learning method can be successful and outperform the standard DQN if in the correct environment. It shows that the dojo network started achieving better results from the beginning of training from the skills learnt in the pre-trained dojo environments. The higher score of the dojo network, shown by the blue dotted line being above the red solid line at 1 episode into training, is evident on all training curves except the cointoss experiment in Figure 5.8, since in that experiment the dojo modules were not pre-trained. Figure 5.9 shows the standard DQN (represented by the solid red line) is never able to surpass the score achieved by the dojo network. This poses another question however, in which environments is the dojo network superior to the standard DQN?

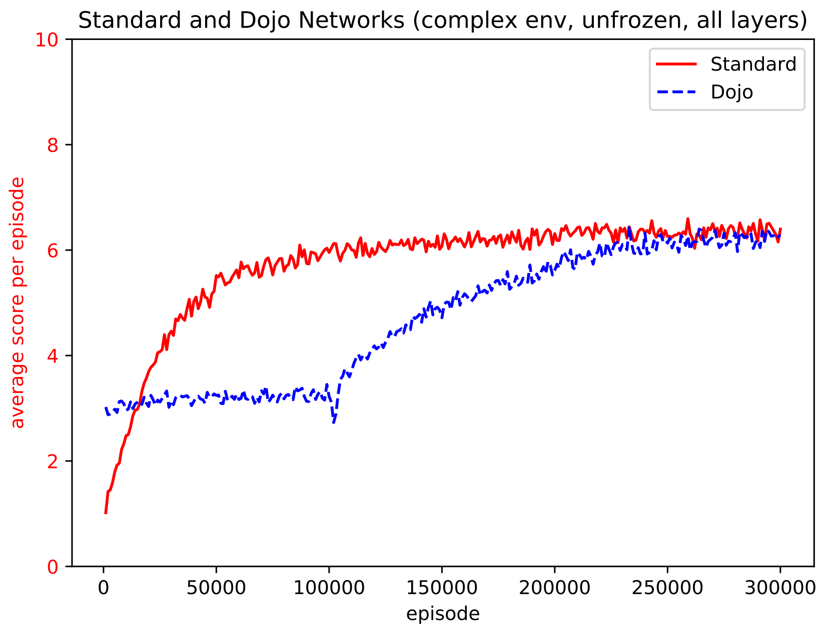


Figure 5.6: Standard and dojo networks, unfrozen at 100k, $\epsilon = 0.1$, complex environment. Dojo input states are *unlimited* and able to observe the entire environment.

5.4 Chapter summary

The results show that dojo network does not always achieve a better performance over the standard DQN in the simple nor the complex environment. It also shows that when the network is unfrozen and all modules are allowed to train, the dojo network will achieve the same result as the standard DQN. The histogram for the tally of dojo modules chosen by the meta module during training reveals a large bias towards the zombie dojo which could be a result of an unbalanced reward system. This makes the agent focus more on one skill or aspect of the complex environment and prioritises its skill over the others. Lastly, the additional experiment results reveal that the dojo network achieves a better result in certain environments with the standard DQN not being able to achieve the same results regardless of the training time.

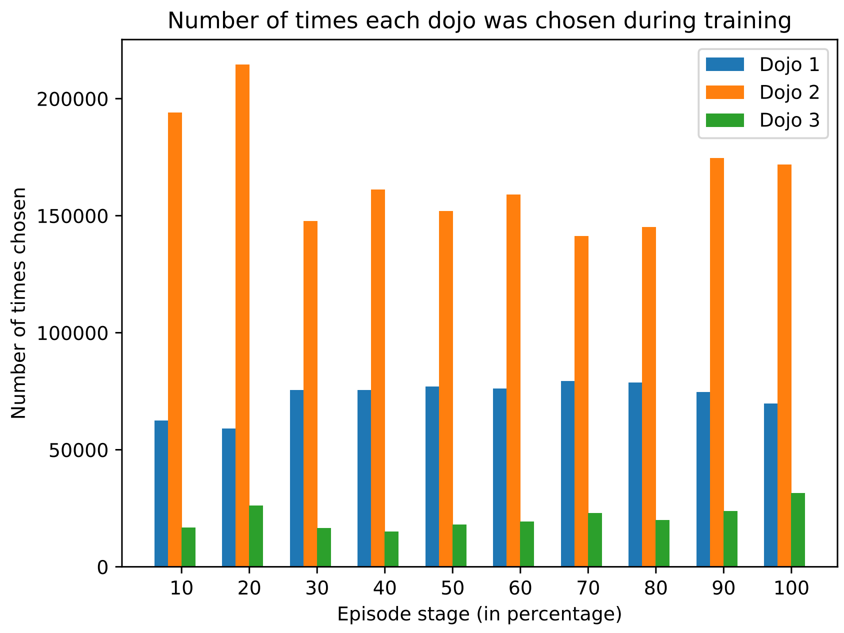


Figure 5.7: A histogram showing the number of times each dojo module was chosen during one million training episodes for the meta module. Dojo 1 (blue) is the diamond dojo, dojo 2 (orange) is the zombie dojo and dojo 3 (green) is the exploration dojo.

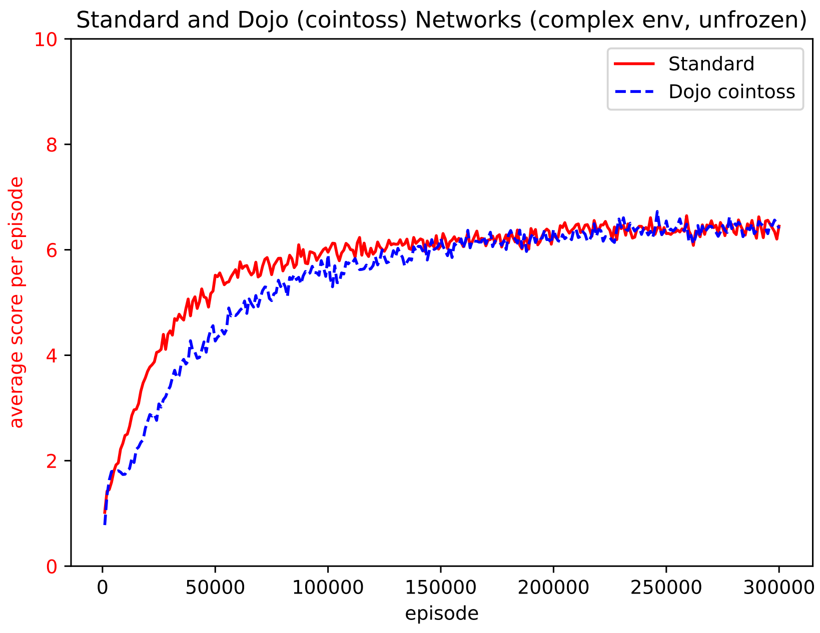


Figure 5.8: Standard network and dojo network in a *cointoss* experiment in the complex environment.

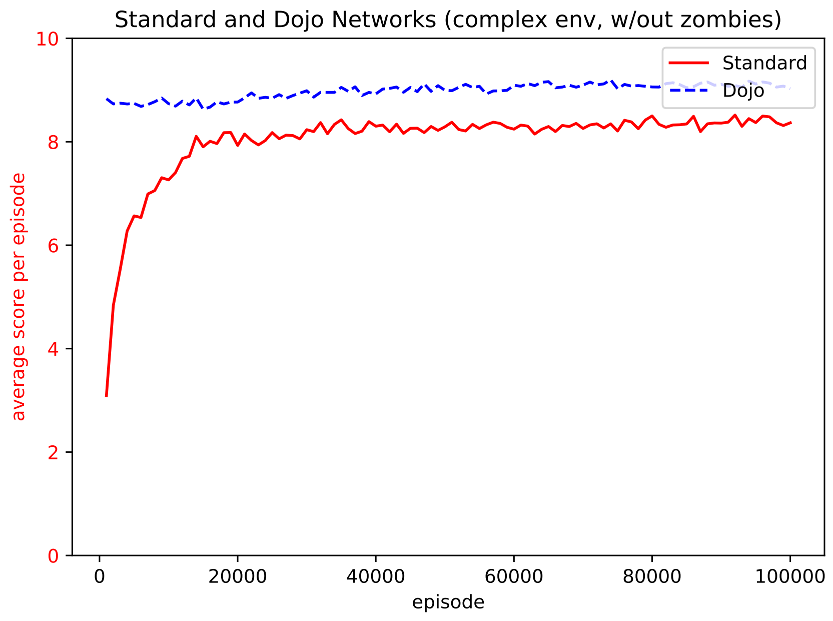


Figure 5.9: Standard network and dojo network in a comlpex environment with no zombies.

Chapter 6

Conclusions

6.1 Experiment conclusion

We set out to show that an agent that learned a subset of complex actions in dojos or sub-environments prior to being exposed to the complex environment might outperform an agent which is trained in the complex environment from the start. The evaluated models do not appear to support our hypothesis with the standard DQN outperforming our dojo network. Evaluating the results shows the dojo network is being limited by the individual modules that was previously learned in isolated dojos and when the agent is exposed to the complex environment the agent is 'stuck' performing those previously learned actions in a sub-optimal manner. It is possible to match the performance of the standard DQN by allowing the dojo modules to be further trained in the complex environment.

It has shown promise that the dojo learning method might be viable in specific environments. In which environments it performs better and why need to be explored in further research.

6.2 Improvements and recommendations

Future work should allow the meta module in the dojo network to have one more action in the form of a complex module. This module allows the agent to move and act based on the complex environment when none of the other skills learnt in the dojos are applicable. This will hopefully give it a boost in training time and performance, by not limiting the available actions.

The histogram in Figure 5.7 shows a bias towards avoiding the zombies. An improvement could be to minimise this bias by altering and balancing the reward system in training as to not make the zombie attack have an overpowering negative reward.

6.3 Future work

One of the main limitations the agent currently faces is the choice in deciding the skills learnt in the different dojos and then locking the number of skills by predetermining the number of dojos beforehand. We need to improve the agent's method of executing actions and not limit it by our choice in dojos. It will be beneficial to explore a non-fixed action structure or use the chosen actions as a stepping stone to learn about the complex environment in a more efficient manner.

Investigate the idea of time integration with the chosen actions having a duration with a specific end point, similar to the options framework discussed in Section 2.5.5. This could take the form of executing a dojo's learnt policy for multiple time steps instead of a different dojo skill every time step.

As previously mentioned, some environments seem to work better than others. We can thoroughly research which environments the dojo network outperforms the standard network, such as the complex environment with no zombies, and identify reasons for these results.

Finally, we can include a more complex algorithm, as mentioned in Chapter 3, and investigate the impact of these more advanced algorithms and methods on our dojo network approach. We could also investigate different architectures for different modules, with the actions of each dojo module being unique and fitting to that particular skill.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *NeurIPS*, 2013.

[2] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," 2016. Available at `https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf`.

[3] "OpenAI Five." Available at `https://openai.com/five/`, 2019.

[4] Microsoft, "Microsoft purchases 'Minecraft'." Available at `https://news.microsoft.com/announcement/microsoft-purchases-minecraft/`, 2015.

[5] K. Hofmann. Personal communication, 2018. Conversation at 2018 Deep Learning Indaba, Stellenbosch.

[6] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, vol. 3, pp. 210–229, 1959.

[7] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997. Available at `http://profsite.um.ac.ir/~monsefi/machine-learning/pdf/Machine-Learning-Tom-Mitchell.pdf`.

[8] R. Ramesh, "What is Artificial Intelligence? In 5 minutes." Available at `https://www.youtube.com/watch?v=2ePf9rue1Ao`, 2017. YouTube video.

[9] A. Brown, "What is the difference between the symbolic and non-symbolic approach to AI?." Available at `https://www.quora.com/What-is-the-difference-between-the-symbolic-and-non-symbolic-approach-to-AI`, 2017. YouTube video.

[10] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST database of handwritten digits." Available at `http://yann.lecun.com/exdb/mnist/`, 2007.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2018. A Bradford Book.

[12] J. Francis, “Reinforcement learning with TensorFlow.” Available at https://www.oreilly.com/ideas/reinforcement-learning-with-tensorflow, 2018.

[13] D. Silver, “RL Course by David Silver.” Available at https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLzuuYNsE1EZAXYR4FJ75jcJseBmo4KQ9-, 2015.

[14] “Reinforcement learning.” Available at https://frnsys.com/ai_notes/artificial_intelligence/reinforcement_learning.html, 2018.

[15] T. Simonini, “An introduction to Reinforcement Learning.” Available at https://www.freecodecamp.org/news/an-introduction-to-reinforcement-learning-4339519de419/, 2018.

[16] V. Mallawaarachchi, “Introduction to Genetic Algorithms — Including Example Code.” Available at https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3, 2017.

[17] C. Finn, “Deep RL Bootcamp Lecture 10B Inverse Reinforcement Learning.” Available at https://www.youtube.com/watch?v=d9DlQSJQAoI, 2017.

[18] S. Saha, “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.” Available at https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2018.

[19] K. Ogawara, J. Takamatsu, H. Kimura, and K. Ikeuchi, “Acquisition of a symbolic manipulation task model by attention point analysis,” *Advanced Robotics*, vol. 17, no. 10, pp. 1073–1091, 2003. Available at https://doi.org/10.1163/156855303322554436.

[20] S. Narvekar, “Curriculum Learning in Reinforcement Learning,” in *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 5195–5196, 2017.

[21] M. Stolle and D. Precup, “Learning Options in Reinforcement Learning,” *School of Computer Science*.

[22] Dr. C. George Boeree, “The Emotional Nervous System.” Available at https://webspace.ship.edu/cgboer/limbicsystem.html, 2009.

[23] tensorflow, “TensorFlow Release Notes.” Available at https://github.com/tensorflow/tensorflow/releases, 2017. GitHub repository.

[24] Fandom, Inc., “Official Minecraft Wiki.” Available at https://minecraft.gamepedia.com/Minecraft_Wiki, 2018. The ultimate resource for all things Minecraft.

[25] B. Roy, “A-Star (A*) Search Algorithm.” Available at https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb, 2019.

[26] jcm2606, “Mods vs Plugins, What’s the difference?.” Available at https://www.minecraftforum.net/forums/minecraft-java-edition/discussion/2575210-mods-vs-plugins-whats-the-difference, 2011.

[27] “Sides in Minecraft.” Available at https://mcforge.readthedocs.io/en/latest/concepts/sides/, 2019.

[28] PyGame community, “PyGame: Free open source Python gaming library.” Available at https://www.pygame.org, 2019.

[29] Microsoft, “Project Malmo.” Available at https://www.microsoft.com/en-us/research/project/project-malmo/, 2015. Available at https://github.com/Microsoft/malmo".

[30] T. Simonini, “Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed Q-targets.” Available at https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682, 2018.