Hierarchical Reinforcement Learning in Minecraft

by

Francois Armand Rossouw

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

Supervisor: Prof. H.A. Engelbrecht

March 2021

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

March 2021

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / *Plagiarism Declaration*

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

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

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

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

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

Studentenommer / *Student number*	Handtekening / *Signature*
F.A. Rossouw Voorletters en van / *Initials and surname*	Datum / *Date*

Abstract

Hierarchical Reinforcement Learning in Minecraft

F.A. Rossouw

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MEng (Electronical)

Humans have the remarkable ability to perform actions at various levels of abstraction. In addition to this, humans are also able to learn new skills by applying relevant knowledge, observing experts and refining t hrough e x p erience. M any c urrent r einforcement learning (RL) algorithms rely on a lengthy trial-and-error training process, making it infeasible to train them in the real world. In this thesis, to address sparse, hierarchical problems we propose the following: (1) an RL algorithm, Branched Rainbow from Demonstrations (BRfD), which combines several improvements to the Deep Q-Networks (DQN) algorithm, and is capable of learning from human demonstrations; (2) a hierarchically structured RL algorithm using BRfD to solve a set of sub-tasks in order to reach a goal. We evaluate both of these algorithms in the 2019 MineRL challenge environments. The MineRL competition challenged participants to find a D iamond i n M inecraft—a 3 D, o pen-world, procedurally generated game. We analyse the efficiency of several improvements implemented in the BRfD algorithm through an extensive ablation study. For this study, the agents are tasked with collecting 64 logs in a Minecraft forest environment. We show that our algorithm outperforms the overall winner of the MineRL challenge in the TreeChop environment. Additionally, we show that nearly all of the improvements impact the performance either in terms of learning speed or rewards received. For the hierarchical algorithm, we segment the demonstrations into the respective sub-tasks. The algorithm then trains a version of BRfD on these demonstrations before learning from its own experiences in the environment. We then evaluate the algorithm by inspecting the proportion of episodes in which certain items were obtained. While our algorithm is able to obtain iron ore, the current state-of-the-art algorithms are capable of obtaining a diamond.

Uittreksel

Hiërargiese Versterkingsleer in Minecraft

("Hierarchical Reinforcement Learning in Minecraft")

F.A. Rossouw

Departement Elektries en Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid-Afrika.

Tesis: MIng (Elektronies)

Mense het die uitsonderlike vermoë om op verskillende vlakke van abstraksie verskeie take uit te voer. Verder kan nuwe vaardighede aangeleer word deur relevante kennis toe te pas, kundiges waar te neem en deur verfyning van ondervinding. Verskeie bestaande versterkingsleer-algoritmes vertrou op omslagtige probeer-en-tref opleidingsprosesse wat dit nie lewensvatbaar maak in die praktyk nie. In hierdie tesis, om die beperkte rangorde van belangrikheid aan te spreek, stel ons die volgende voor: (1) 'n versterkingsleer-algoritme, *"Branched Rainbow from Demonstrations (BRfD)"*, wat verskeie verbeterings in die *"Deep Q-Networks (DQN)"* algoritme kombineer wat deur menslike demonstrasie leer; (2) 'n hiërargiesgestruktureerde versterkingsleer-algoritme wat deur middel van BRfD verskeie subtake kan oplos. Ons ontleed beide die bovermelde algoritmes in die 2019 *"MineRL"* omgewing. Die *"MineRL"* kompetisie het deelnemers uitgedaag om 'n Diamant te vind in *"Minecraft"*. *"Minecraft"* is 'n driedimensionele, *"open-world"*, progressief gegenereerde rekenaarspeletjie. Verskeie verbeteringe wat in die BRfD-algoritme toegepas is deur omvangryke ablasiestudiemetodes word ontleed. Vir die studie is die agente opdrag gegee om 64 *"logs"* in 'n *"Minecraft"* woud omgewing bymekaar te maak. Ons toon dat hierdie algoritme die algehele wenner in die *"Treechop"* omgewing van die 2019 *"MineRL"* uitdaging klop. Verder toon ons dat byna alle verbeterings 'n positiewe impak het ten opsigte van leerspoed of vergoeding ontvang. Vir die hiërargiese algoritme is die demonstrasies opgebreek in hulle verskeie subopdragte. Die algoritme leer dan 'n weergawe van BRfD deur middel van hierdie demonstrasies gebaseer op sy eie ondervinding in die omgewing. Ons evalueer dan die algoritmes deur 'n ondersoek te doen na die proporsie van episodes waar sekere items verkry is. Ons algoritme kon slegs ystererts vind in teenstelling met die huidige moderne algoritmes wat 'n diamant vind.

Acknowledgements

I would like to express my deepest gratitude to the following people:

- My supervisor, Prof. H.A. Engelbrecht, for his support and guidance throughout this work.
- The students from the Media Lab, especially Cobus Louw, who entered the 2019 MineRL competition with me and continued to provide insights.
- My parents, without whom none of this would have been possible. This work is a testimony to their love and support.
- My brother, Werner, who offered advice when required and always inspired me to improve.
- My wife, Liesl, whose unwavering love and support carried me throughout my studies. In the good days and the bad days, I could always count on her for support and motivation.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

2D	2-Dimensional
Adam	Adaptive Moment estimation
BC	Behavioural Cloning
BDQ	Branching Duelling DQN
BRfD	Branched Rainbow from Demonstrations
CNN	Convolutional Neural Network
DDQN	Double Deep Q-Networks
DQfD	Deep Q-learning from Demonstrations
DQN	Deep Q-Networks
FFNN	Feed-forward Neural Network
ForgER	Forgetful Experience Replay
FQF	Fully Parameterized Quantile Function
HDQfD	Hierarchical Deep Q-Network from Demonstrations
HRL	Hierarchical Reinforcement Learning
IL	Imitation Learning
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IMPALA	Importance Weighted Actor-Learner Architecture
IQN	Implicit Quantile Networks
KL	Kullback-Leibler
LAE	Least Absolute Error

LSE	Least Squares Error
MAE	Mean Absolute Error
MC	Monte Carlo
MDP	Markov Decision Process
MSE	Mean Squared Error
NN	Neural Network
PER	Prioritised Experience Replay
POfD	Policy Optimisation with Demonstrations
POMDP	Partially Observable Markov Decision Process
POV	Point of View
QR-DQN	Quantile Regression DQN
ReLU	Rectified Linear Unit
RGB	Red, Green and Blue
RL	Reinforcement Learning
RMSProp	Root Mean Squared Propagation
RNG	Random Number Generator
RNN	Recurrent Neural Network
SE	State Estimator
SGD	Stochastic Gradient Descent
SOTA	State-of-the-art
TD	Temporal-Difference

Symbols

α	Learning Rate
$\mathcal{A}$	Finite set of Actions
$\mathcal{B}$	Finite set of Belief States
$\mathcal{T}^{\pi}$	Bellman Operator
$\mathcal{T}$	Bellman Operator
γ	Discount Factor
$\mathcal{G}$	Finite set of Options
$\mathcal{L}$	Scalar Loss
$\mathcal{N}$	Normal Distribution
Ω	Finite set of Observations
$\mathcal{O}$	Observation Transition Probability Matrix
π	Agent Policy
π_*	Optimal Policy
$\mathcal{R}$	Reward Function
$\mathcal{D}$	Replay Memory
$\mathcal{S}$	Finite set of States
$\mathcal{P}$	State Transition Probability Matrix
τ	Belief State Transition Probability Matrix
δ	TD Error
q_{π}	Action-value Function
q_*	Optimal Action-value Function
v_*	Optimal State-value Function
v_{π}	State-value Function

1 Introduction

In light of recent advancements in reinforcement learning (RL), there is an ongoing search for an algorithm capable of performing in real world scenarios at levels on par with or exceeding human performance. Many of these advancements focus on solving a specific task or small subset of subsequent tasks, and would have trouble when presented with a hierarchy of tasks to discover and traverse. While many solutions have been proposed to attempt to solve these hierarchical problems, the environments evaluated are often fully observable and predictable. Real-world problems on the other hand are strictly partially observable, mostly unpredictable and often involve a hierarchy of tasks to be solved.

1.1 Motivation and Topicality

Model-free RL algorithms rely on a lengthy trial and error training process. This leads one to question the safety of training algorithms with no prior experience in real world environments. Therefore, the need for a fast simulation environment arises to train RL algorithms quickly and safely. While the environment would ideally also need be accurate, providing a simulation capable of placing the algorithm in constant never-before-seen scenarios would suffice as an early indication of its viability.

A common approach to reducing trial and error interactions in the environment is by leveraging human demonstration data. Providing the algorithm with examples of how a human would solve a task can significantly reduce training time and increase performance. However, this comes at the cost of potentially discovering alternative methods of solving a task—methods a human would not necessarily consider.

This work attempts to solve a complex subset of hierarchical tasks (each of which may require solving another subset of tasks) to reach a final goal. By incorporating demonstration data we hope to significantly reduce reliance on trial and error training. Through an ablation study we also evaluate the impact of each enhancement on the final model on a single task to determine the performance impact relative to computational cost.

1.2 Background

Before we can discuss the contributions of this work and a synopsis of relevant literature, we need to cover the background relevant to the work performed in this thesis. We provide high-level descriptions of RL, neural networks (NNs) and the MineRL environments used for experimentation in this work.

1.2.1 Reinforcement Learning

Reinforcement learning is the process of learning to map states to actions by maximising the expected future reward through interaction with an environment [1, p. 1–2]. A reward in the context of RL refers to a signal received from the environment for displaying certain desired behaviour. We typically frame RL problems as Markov Decision Processes (MDPs).

An MDP is a sequential environment where future states are independent from past states given the current state [10]. This property is referred to as the Markov Property. An MDP is defined by the following components:

- A finite set of all possible states of the environment ($\mathcal{S}$).
- A finite set of actions to choose from ($\mathcal{A}$).
- A state transition probability matrix to describe the probability of moving from the current state to the next state given a certain action ($\mathcal{P}$).
- A reward function that provides a reward given the action taken in a certain state ($\mathcal{R}$).

These four components respectively form the MDP tuple: $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$. At every time step in an MDP, the agent receives the current state of the environment and selects an action according to its behaviour policy. The environment then adapts and returns a new state and reward according to the selected action. This feedback loop is depicted in Figure 1.1 below. We refer to a sequence of these steps as a trajectory. When a trajectory includes the initial and terminal states, it can also be called an episode.

By rewarding the agent for behaviour which leads to desired outcomes, the agent can refine its policy to maximise the expected future reward. However, achieving a desired outcome is not always possible when only considering the short term rewards; the agent needs to account for the ramifications in the long term.

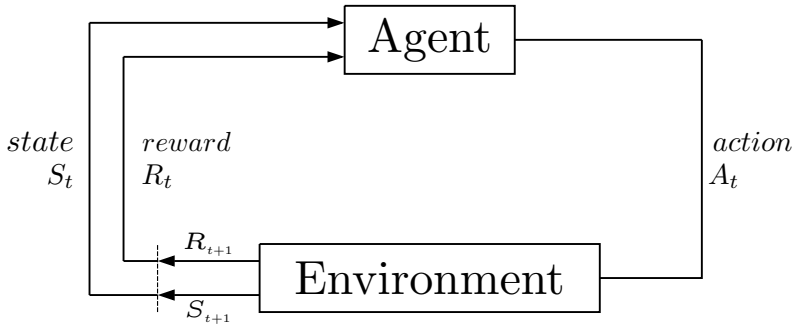

Figure 1.1: Typical reinforcement learning loop. The environment provides a state and a reward which agent uses to select an action to interact with the environment. This then leads to a new state and reward from the environment [1, p. 48].

RL problems can be solved with either model-free methods or model-based methods. Model-based methods use planning and a model of the environment to solve an RL problem. Model-free methods assumes the model is unknown and attempts to approximate the model by way of trail-and-error [1]. Thus, due to the complexity of the natural world or a procedurally generated world we focus on model-free methods.

The trail-and-error methodology also introduces the issue of exploration versus exploitation. On the one hand, we want the agent to maximise the expected future reward. On the other hand, our agent should not be limited in its initial understanding of the environment. Limiting the exploration of an agent inevitably leads to decreased generalisation, and the agent may not discover alternative behaviours resulting in even greater rewards. A common solution to the problem is to start by exploring and gradually move towards exploiting knowledge gained.

When an agent needs to select an action, it either consults a policy for action probabilities or directly evaluates the expected future reward of each action. These are respectively known as the policy- and value functions. The Q-learning algorithm popularised by Mnih *et al.* [11] is an example of a value function where the algorithm computes the value of each action in each state. These concepts are explored further in Chapter 2.

Hierarchical Reinforcement Learning. In non-hierarchical RL, an agent typically behaves according to a single policy. This can limit the performance of an agent in complex environments that require long-term planning and involve solving several sub-tasks. On the other hand, hierarchical reinforcement learning (HRL) agents learn a set of sub-policies, each to solve a specific sub-task. During an episode, another, higher-level policy then dictates which sub-policy is in control of selecting primitive actions. This model is more akin to how humans behave. Humans act and decide with various levels of abstraction. For example, when humans want to grasp an item, they do not have to consciously control

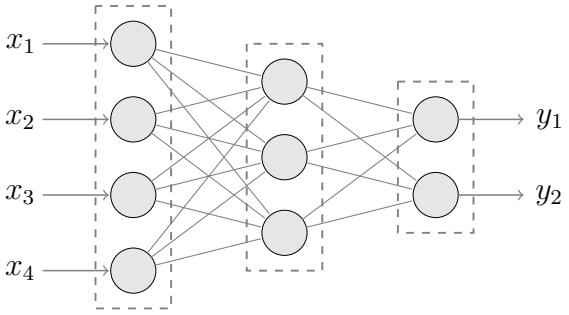

Figure 1.2: Simple feed-forward fully connected neural network. Each circle represents a single neuron and each edge a weighted connection. Layers are demarcated by dashed lines.

each muscle involved in the action. HRL aims to learn similar behaviour. In Section 2.8 we discuss this topic in more detail.

1.2.2 Neural Networks

In small toy problems, we often represent the Q-function with a table mapping the values of all possible actions in all possible states. However, we quickly find ourselves faced by the curse of dimensionality. The curse of dimensionality is the exponential rate of growth of a space as the dimensionality increases [12].

In our particular case, the environment is observed as a raw image. If each distinct image is regarded as a state, a 64×64, 24-bit image can represent over $2.8 \times 10^{29\,592}$ possible states. This is of course a gross overestimation, since all possible states would not necessarily be observable in an environment, but it serves as a good justification to explore alternative solutions to the curse of dimensionality.

With the above-mentioned in mind, we investigate a way of approximating Q-functions: artificial neural networks. Note that for the purposes of this work, all references to neural networks henceforth can be assumed to be artificial. Figure 1.2 depicts a simple neural network which may be used to approximate function f in the equation $\boldsymbol{y} = f(\boldsymbol{x})$. The learning process of the neural network attempts to find the ideal weight for each edge in the depiction based on example inputs ($\boldsymbol{x}$) and outputs ($\boldsymbol{y}$).

Unfortunately, processing an image is not as simple. Images may not necessarily represent information in fixed patterns. For example, an important object to detect may be obfuscated, rotated, translated or have different appearances.

Convolutional neural networks. Convolutional neural networks (CNNs) attempt to overcome the issues in processing image data. They apply learnable sliding filters to detect spatial features. Lower level CNN layers generally detect basic directional edges. With each additional layer the network is able to detect higher level features. For example, in the task of facial recognition, a second layer may detect facial features such as eyes. A subsequent layer may then detect full faces [13].

In their publication, Mnih *et al.* [3] successfully utilised CNN layers to achieve human level performance on the Atari 2600 games suite. They proposed three CNN layers followed by two fully connected layers. In their work they also evaluated the effectiveness of using CNN layers as opposed to a single linear layer. They found the CNN architecture to vastly outperform a linear layer. We explore their CNN design further in Section 6.10 by evaluating performance against other proposed architectures.

1.2.3 MineRL

In 2019, Guss *et al.* [14] proposed the MineRL challenge for the NeurIPS 2019 conference. The goal of the challenge was to solve Minecraft by obtaining a diamond. Minecraft is a 3D, first person, open-world, procedurally generated game where a player is tasked to survive. Although there are no defined goals in the game itself, certain items can only be obtained if a set of prerequisites is met. Thus, we can pick an arbitrary item based on the complexity of acquiring it, and set it as the goal. For example, obtaining a block of cobblestone requires a player to break the block with any pick-axe. A basic wooden pick-axe requires sticks and planks in the player inventory in addition to the presence of a crafting table nearby. All of the items above require blocks of wooden logs to be crafted into blocks of planks.

The MineRL competition challenged participants to develop a generalised algorithm capable of obtaining a diamond in Minecraft. To that end, they released the ObtainDiamond environment based on a Minecraft experimentation platform, Project Malmö [15]. The player is placed in a Minecraft survival world with the goal of obtaining a diamond. A game is terminated upon reaching the goal, dying or reaching the time limit of 18000 steps (which translates to 15 minutes of normal play). The sequence of items required to obtaining a diamond in Minecraft is shown in Figure 1.3. Note that all items are not obtained by simply breaking a block. Certain items require crafting in either the crafting table or the player inventory. Other items require a furnace with fuel and another resource to be smelted.

Aside from the primary environment of the competition, the MineRL team also released

auxiliary environments with different goals. Among the auxiliary environments, we focus on two specific environments.

Firstly, the Treechop environment places the player in a moderately dense forest. The environment is considered solved when the player obtains 64 or more log blocks in an episode. Episode termination conditions are similar to that of the ObtainDiamond environment; the difference being the time limit is 8000 steps or roughly 7 minutes of play. We use this environment in the ablation study to evaluate performance impact on a more consistent environment.

Secondly, the ObtainIronPickaxe environment only shifts the goal of ObtainDiamond back to an iron pick-axe. The time limit also decreases to 6000 steps (5 minutes). Although we do not use this environment directly for evaluation, we consider obtaining an iron pick-axe as the goal for our agents.

Obtaining an iron pick-axe is already considered to be a complex exploration problem. First a player needs to locate and collect a log. Upon collecting the required materials for crafting a wooden pick-axe, the player then needs to find stone to mine for crafting a furnace and stone pick-axe. After crafting a stone pick-axe, the player then needs to locate three iron ore blocks. The player should then have the required materials to craft an iron pick-axe. Finally, the player can then explore the world to find a diamond. However this exploration is an extremely challenging task without domain knowledge of the world. A diamond can only be found between levels 0 and 16 (where one level is a layer of blocks). For reference, the sea level is at around 62 and a player is spawned anywhere above this level.

In addition to the challenge proposal, the team behind MineRL also released a large dataset of human demonstrations [16]. In their retrospective analysis of the competition, they noted that of the top nine teams, the top seven leveraged the demonstration data [17].

1.2.3.1 Observation Space

The observation space of the MineRL environment consists of an image and a vector. The image is a snapshot in time of the agent's current POV. This image is provided in the form of a 64×64 red, green and blue (RGB) image. Figure 1.4 shows a sample POV observation.

The observation vector is only relevant in environments where the goal involves obtaining a certain item. This vector provides an agent with information about the contents of its inventory. Each component represents quantities of specific items present in the agent's

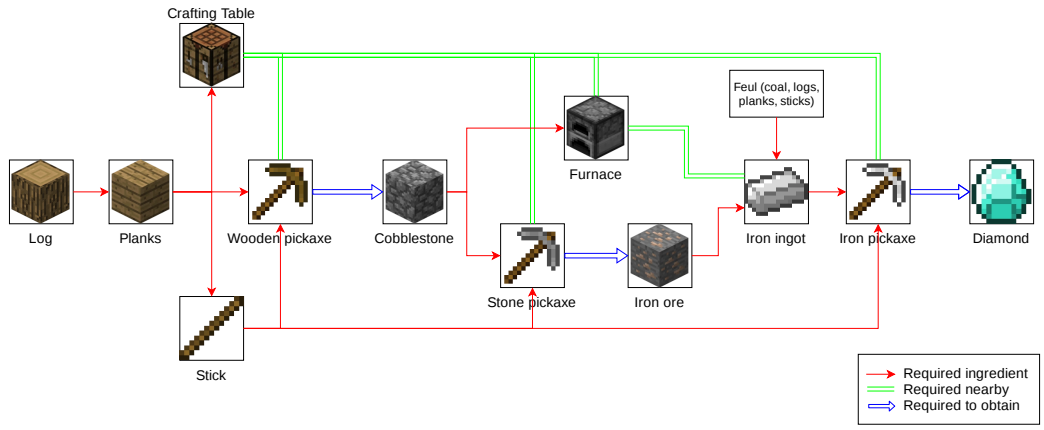

Figure 1.3: The hierarchy of items that a player needs to traverse to obtain a diamond in Minecraft. Note that a higher level pick-axe is capable of mining lower-level items.

Figure 1.4: Sample of 64×64 POV observation in the Treechop environment.

inventory. Additionally the vector also contains characteristics of the item currently equipped in the agent's main hand. Table 1.1 lists the components along with descriptions and possible value ranges.

1.2.3.2 Action Space

We can categorise the action space into three types of actions, i.e. movement, camera and crafting. Movement actions include actions such as 'forward' or 'back'. These actions are generally used to traverse the environment an agent finds itself in. Although we include 'attack' in the movements category, it is mainly used to break blocks or damage entities in the environment.

Table 1.1: MineRL Obtain environments inventory observation vector with value range limitations.

Item	Value range	Item	Value range
Equipped item damage	$[-1, 1562]$	Iron pick-axe	$[0, 2304]$
Equipped item maxDamage	$[-1, 1562]$	Log	$[0, 2304]$
Equipped item type	$[0, 9]$	Planks	$[0, 2304]$
Coal	$[0, 2304]$	Sticks	$[0, 2304]$
Cobblestone	$[0, 2304]$	Stone	$[0, 2304]$
Crafting table	$[0, 2304]$	Stone axe	$[0, 2304]$
Dirt	$[0, 2304]$	Stone pick-axe	$[0, 2304]$
Furnace	$[0, 2304]$	Torch	$[0, 2304]$
Iron axe	$[0, 2304]$	Wooden axe	$[0, 2304]$
Iron ingot	$[0, 2304]$	Wooden pick-axe	$[0, 2304]$
Iron ore	$[0, 2304]$		

Camera actions shift the view of the agent's POV along the pitch and yaw axes. A change in camera yaw changes the direction that the agent is facing. This change will then impact the direction of movement relative to before the change since movement occurs relative to the absolute direction that the agent is facing.

We categorise the remaining actions as crafting actions since they involve manipulating items in the inventory. In Minecraft a player has a small 2×2 grid area in the inventory interface to craft simple items like planks, sticks, crafting tables and torches. Crafting tables can then be placed on the surface of a nearby block. A player within reach can use the crafting table with a larger 3×3 grid to craft more items. Organising items in a specific pattern would then result in another (ideally more valuable) item. If the item is a tool, the player can equip the tool in the main hand. The tool in a player's hand affects the speed of block breaking depending on the block and the tool. It may also determine whether or not the broken block can be collected. For example, breaking a stone block without a pick-axe will not allow the player to collect a cobblestone block.

The Malmö project simplified the above-mentioned actions to single actions performed in a single step. The 'craft' action uses player inventory to craft the simple items. If the agent needs a tool to collect a block, the tool can be equipped with the 'equip' action given that the tool exists in the agent's inventory. To use a crafting table the agent needs to be within reach (roughly 4 blocks) and simply use the 'nearbyCraft' action to select an item to craft. If the agent possesses the required materials the item will be crafted and placed into the agent's inventory. The 'nearbySmelt' action functions in a similar manner for a nearby furnace. Lastly, the agent can use the 'place' action to place a furnace or crafting table on a nearby block. See Table 1.2 for valid value ranges for each action. And Table 1.3 shows the mapping of discrete values to inventory actions for the crafting

Table 1.2: MineRL action space with range of possible actions. For the Treechop environment the crafting actions are not relevant.

Movement (Discrete)		**Crafting (Discrete)**		**Camera (Continuous)**	
Action	Value range	Action	Value range	Action	Value range
Attack	$[0, 1]$	Craft	$[0, 4]$	Camera pitch	$[-180, 180]$
Back	$[0, 1]$	Equip	$[0, 7]$	Camera yaw	$[-180, 180]$
Forward	$[0, 1]$	Nearby Craft	$[0, 7]$		
Jump	$[0, 1]$	Nearby Smelt	$[0, 2]$		
Left	$[0, 1]$	Place	$[0, 6]$		
Right	$[0, 1]$				
Sneak	$[0, 1]$				
Sprint	$[0, 1]$				

Table 1.3: Mappings of discrete values for crafting actions to Minecraft actions.

Values	**Actions**				
	Craft	Equip	Nearby Craft	Nearby Smelt	Place
0	None	None	None	None	None
1	Crafting table	Air	Furnace	Coal	Cobblestone
2	Planks	Axe (I)	Axe (I)	Iron ore	Crafting table
3	Torch	Pick-axe (I)	Pick-axe (I)		Dirt
4	Sticks	Axe (S)	Axe (S)		Furnace
5		Pick-axe (S)	Pick-axe (S)		Stone
6		Axe (W)	Axe (W)		Torch
7		Pick-axe (W)	Pick-axe (W)		

category. Note that in the interest of space, we used a notation of (*) to indicate the material of the axe or pick-axe—where (I) is iron, (S) is stone and (W) is wooden.

1.2.3.3 Rewards

To help RL agents with solving this problem, the MineRL environments provide additional auxiliary rewards for obtaining items required to obtain a diamond. These rewards are tabulated in Table 1.4. For the MineRL Obtain environments (non-dense), the agent receives each reward only the first time that an item is collected. Alternatively, the ObtainDense environments reward the agent for all required items collected. For the Treechop environment the agent simply receives a reward of 1 for each log collected.

Table 1.4: Rewards for items obtained in the MineRL Obtain environments.

Item	**Reward**
Log	1
Plank	2
Stick	4
Crafting table	4
Wooden pick-axe	8
Cobblestone	16
Stone pick-axe	32
Furnace	32
Iron ore	64
Iron ingot	128
Iron pick-axe	256
Diamond	1024

1.3 Problem Statement

In this thesis, we attempt to solve the problems proposed by the 2019 MineRL challenge [14]. The main problem involves solving a complex, hierarchical, and sparse simulation environment with a generalised RL algorithm. To improve the sample-efficiency of possible solutions, the organisers of MineRL also released a dataset of human demonstrations showing how to solve the environments [16]. With the above-mentioned in mind, we find ourselves facing the following problems:

- In order to obtain a diamond in Minecraft, our agent needs to obtain the items shown in Figure 1.3 in varying quantities. Each of these items can be viewed as a sub-task that needs to be completed to reach the final goal. Additionally, these sub-tasks may not be clearly defined.
- Sparse environments requiring long-term planning often involve generating a huge number of samples to train from. This is done through a lengthy trial and error training process that may never reach a solution to a complex problem such as this.
- While the sub-tasks may not present formidable challenges individually, selecting which sub-task to solve in a given scenario can be very challenging. Ideally, the algorithm selecting sub-tasks should be generalisable to other problems.

1.4 Objectives

In order to address the problems described in Section 1.3 above, we set out to design an HRL algorithm. In designing this algorithm, we aim to address the following objectives.

- Design an algorithm to act as a low-level policy. This algorithm should be capable of efficiently learning to solve a given task. To that end, we investigate various components of the algorithm through an ablation study.
- Effectively leverage the human demonstration data to reduce the reliance on a lengthy trail-an-error training process.
- Discover and navigate the hierarchy of sub-tasks in a generalised manner.

1.5 Literature Synopsis

We review an RL algorithm using NNs to approximate the values of actions. This algorithm is called Deep Q-Networks (DQN) and was proposed by Mnih *et al.* [3]. In the years following the creation of this algorithm, several improvements have been proposed to address various flaws or shortcomings of the original DQN algorithm. The authors of the Rainbow DQN algorithm combined some of these improvements to achieve state-of-the-art (SOTA) performance at the time. We investigate the findings of the authors to determine the effectiveness of their algorithms.

Since we have access to a dataset of demonstrations, we need a way of effectively leveraging them to learn a good prior policy. To that end, we investigate behavioural cloning (BC), an imitation learning method using only observations and actions to learn a policy. However, the authors of Hester *et al.* [18] raised valid concerns on the effectiveness of imitation learning approaches that ignore the reward signal. They proposed Deep Q-learning from Demonstrations (DQfD), a DQN based algorithm capable of leveraging demonstration data. We discuss their work along with BC as two ways of learning a prior.

To provide context on existing solutions to the problem we face, we also review the two winning submissions of the 2019 MineRL challenge. Amiranashvili *et al.* [9] proposed an imitation learning approach. Besides inventory- and action space shaping, they investigated alternative CNN designs and data augmentation techniques. They found a more complicated CNN architecture based on the work of Nichol [19] to outperform the original DQN CNN architecture proposed by Mnih *et al.* [3]. Additionally, their results indicate

that flipping the demonstration data horizontally is the only data augmentation technique to provide any significant benefits. They found their solution outperforms both DQN and DQfD [18].

Skrynnik *et al.* [20] on the other hand, proposed a hierarchical DQfD algorithm. They examined expert trajectories to construct a chain of sub-tasks to be completed in order to achieve the goal. Each of these sub-tasks are then assigned a sub-policy which assumes control until the task is completed. While this approach proved successful, it is not necessarily generalisable to new problems where the items may be obfuscated in one way or another. They also proposed a replay buffer framework where the proportion of expert samples versus agent samples in a training batch is controlled with a dynamic variable. Lastly, they augmented the demonstration data for each sub-policy with data from other sub-policies. This helps the agent to learn behaviour generally beneficial across all policies. Ultimately, the solution proposed by Skrynnik *et al.* [20] managed to solve the ObtainDiamond environment in one evaluation episode—an impressive feat nonetheless.

1.6 Contributions

In this work we set out to design an HRL algorithm capable of discovering and navigating a complex set of hierarchical sub-tasks to accomplish a goal. We started by designing the agent on a higher level and investigating SOTA RL algorithms capable of solving the lower level tasks. The successes of the DQfD and Rainbow DQN algorithms served as motivation to investigate a solution based on a combination of these algorithms. We also investigated Branching Duelling DQN (BDQ) (an algorithm that supports simultaneous action selection) to address the complications introduced by an action space where simultaneous actions are possible and need to be interpreted from expert demonstrations.

The resulting low-level agent created from a combination of DQfD, Rainbow DQN and BDQ proved capable of solving the Treechop environment. It successfully leveraged the demonstration data to learn good starting behaviour, even when faced with unseen states. Additionally, it outperformed both the baselines and the 2019 MineRL winning submission, Forgetful Experience Replay (ForgER), in the Treechop environment. To discover the most effective variant, we also performed an extensive ablation study on the DQN enhancements implemented in this work. Our findings revealed certain enhancements provide little to no impact on performance. When comparing our results to that of Hessel *et al.* [7], we concluded that the addition of demonstration data decreases the impact of the distributional, duelling and prioritised replay enhancements.

Unfortunately, our HRL architecture failed to deliver when tasked with solving the

ObtainDiamond environment. We believe the unstructured hierarchy is a source of confusion for both high- and low-level policies. With demonstration data following a wide variety of trajectories and traversing the item hierarchy in differing orders, the agent is not able to generalise to unseen environment seeds.

We also observed that the small probability at which the low-level policies acted randomly impacted performance. Interestingly, we found that when the agent was unable to select actions randomly from the crafting actions, it performed considerably worse. However, in a problem where resources are required for progression, crafting randomly may unnecessarily consume these valuable resources.

1.7 Overview

In Chapter 2 we provide some background on RL. We discuss the building blocks to help understand the Q-learning algorithm. Additionally, we discuss a way of approaching hierarchical problems from an RL perspective.

Continuing from the previous chapter, we also provide background on NNs in Chapter 3. The DQN algorithm uses NNs to process the observation and approximate action values. The topics we discuss in this chapter provide the reader the relevant knowledge to understand how the algorithm learns.

In Chapter 4, we investigate the existing literature on algorithms that are relevant to solving the problem. We look at the original DQN algorithm along with several improvements proposed by other researchers. Particularly, we focus on the improvements included in the Rainbow DQN algorithm. We then look at imitation learning algorithms to effectively leverage the human demonstrations from the MineRL dataset. Lastly, we investigate existing solutions to the MineRL problem.

With the foundations of existing literature and background, we discuss the details of our implementation in Chapter 5. This chapter also describes the training procedure for our experiments. In implementing our solution, we came across many issues harming the performance of our algorithm. We also discuss these issues and our solutions in Chapter 5.

In Chapter 6 we perform an extensive ablation study on the components of our algorithm. We analyse the training curves of each ablation on the Treechop environment to determine their effectiveness. We also discuss the cost of implementing the ablations.

Finally, in Chapter 7, we evaluate our HRL algorithm on the MineRL ObtainDiamond environment. We measure the performance as the percentage of evaluation episodes in

which the agent obtained a given item. We also compare our results to a non-hierarchical version of our algorithm, as well as the winning submission of MineRL 2019.

In Chapter 8, we present our conclusions based on the findings of the previous two chapters. We then revisit our objectives stated in Section 1.4 and whether or not we were able to meet them. We also provide several recommendations for future work on this problem.

2 Reinforcement Learning

To solve problems such as the MineRL environments, we need to investigate methods of learning how to behave appropriately in familiar and especially unfamiliar scenarios. We can think of the MineRL problem in terms of a simple daily task like making a cup of tea. For adult humans raised in a tea drinking society, this task should be trivial. However, without any domain knowledge, it can be quite a complicated task. In this example, the initiation and termination conditions are simple: we start when a desire for tea arises, and stop when the tea is finished. We are then rewarded with the desired beverage. Unfortunately, not all tasks yield rewards in such a short term. For example, investments often require one to be patient in order to reap the accumulated rewards at a later date.

The question then becomes: how do we know which actions in the present will yield the greatest reward over time? In the real world, we can almost never be 100% sure. However, we can make informed decisions based on personal past experience or information obtained from external sources. Achieving the same level of informed decision-making with an algorithm is not an easy task. A computer algorithm does not have access to past experience or external sources of information. It starts with no knowledge of the environment it finds itself in. Like a toddler learning to navigate its surroundings and communicating, we provide positive reinforcement for accomplishing certain goals. In RL, we formalise this idea mathematically with an MDP framework. In the rest of this chapter, we discuss MDP frameworks and methods of solving them.

2.1 Markov Decision Process

An MDP formalises a sequential decision making process to a mathematical control process. At a given time-step t an agent performs an action A_t according to a certain policy π. The policy selects the action from a finite set of actions ($\mathcal{A}$) based on the state S_t the agent finds itself in. The environment then updates to a new state S_{t+1} from a finite set of states ($\mathcal{S}$). The agent is also provided with a reward R_{t+1} from a reward function ($\mathcal{R}$). A state transition probability matrix ($\mathcal{P}$) describes the probability of each state occurring given

the current state and selected action. The process described above is typically depicted as a feedback loop—as shown in Figure 1.1.

The environment dynamics can therefore be described by the four fundamental components mentioned above as a four-tuple: $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$. We recognise that many other literature include the discount factor (γ) in the MDP model. The discount factor is used to scale the importance of future rewards. For example, even though an investment may pay off in the long term, an investor may value immediate returns more. We believe the discount factor does not form part of the environment dynamics. Rather, it should be considered as another hyperparameter to be fine-tuned.

To formulate our problem as an MDP, the set of states ($\mathcal{S}$) needs to satisfy the Markov property. The Markov property requires all states to contain all information pertaining to past interactions with the environment that influence the future. For example, in the MineRL Obtain environments, the POV observation and inventory contents provides the agent with sufficient information to infer the past and make decisions that impact the future [1, p. 49].

Unfortunately, the POV image provided by the MineRL environments is only a view of the much larger environment. In these types of problems, we define states and observations as separate sets. The state represents all of the environment, whereas the observation represents only a small section within the environment. In these cases we classify the MDP as a Partially Observable Markov Decision Process (POMDP) [2].

Partially Observable Markov Decision Process. A POMDP is an MDP where the full state of the environment is not observable at all times. This introduces two new components to our four-tuple:

- A finite set of observations (Ω) representing all possible observations of $\mathcal{S}$.
- A observation transition probability matrix ($\mathcal{O}$) describing the probability of transitioning to an observation given the selected action, the state of the environment and the current observation.

A POMDP can thus be represented by a six-tuple: $\langle \mathcal{S}, \Omega, \mathcal{A}, \mathcal{P}, \mathcal{O}, \mathcal{R} \rangle$. While the state is not fully observable in a POMDP, the agent maintains a belief of the state. The belief state represents a probability distribution that the environment is in a certain state within $\mathcal{S}$ given an observation.

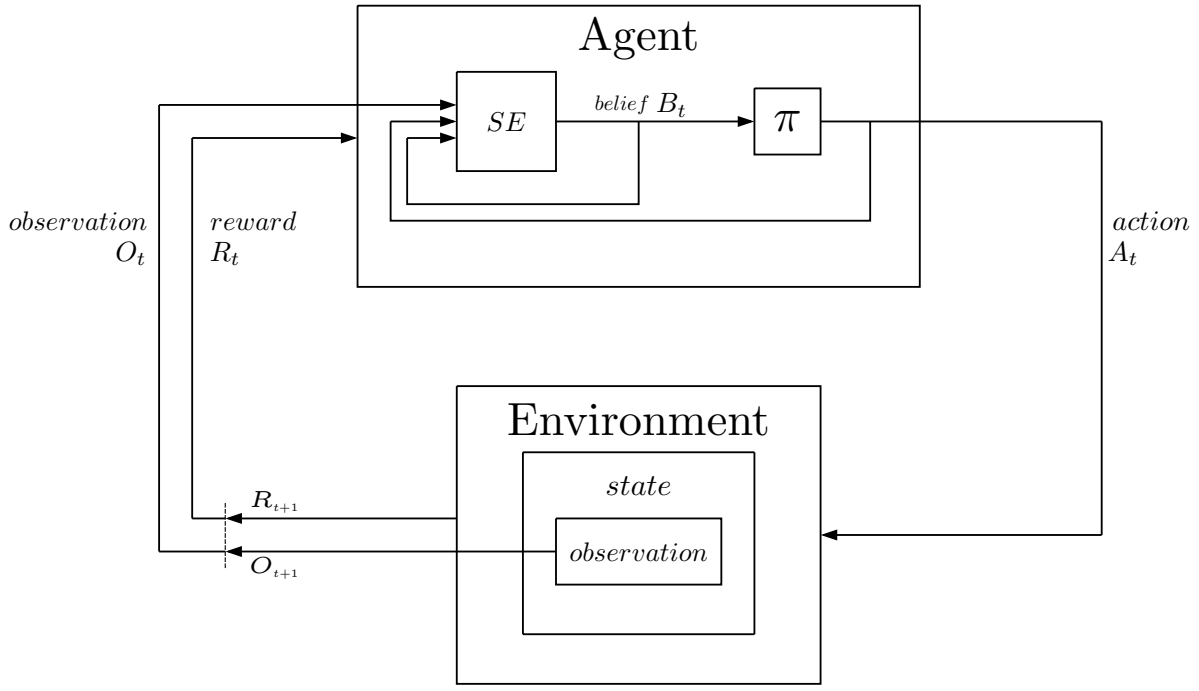

Figure 2.1: POMDP controller. The agent's state estimator (SE) produces a belief state based on the new observation and the previous belief state and action [2].

By introducing a finite set of belief states ($\mathcal{B}$), we can effectively revert to the normal form of an MDP. We replace the set of states $\mathcal{S}$ with a set of belief states $\mathcal{B}$. The state transition probability matrix becomes a belief state transition probability matrix (τ). The original four-component MDP representation is then $\langle \mathcal{B}, \mathcal{A}, \tau, \mathcal{R} \rangle$. Figure 2.1 shows the MDP controller modified to represent a belief MDP. With the belief MDP defined we can now discuss the components involved in solving an MDP.

2.2 Model of Environment

In the previous section we mentioned the idea of a policy that determines the behaviour of an agent. This behaviour is influenced by a reward signal received from the environment for various reasons. The policy and reward signal are two of the four sub-elements of an RL system [1, p. 6–7]. The remaining elements are the value function (see Section 2.3 below) and a model of the environment. However, a model is not a required element. Methods that utilise a model of the environment are described as model-based methods. Alternatively, model-free methods attempt to learn a policy without access to a model.

A model of the environment allows an agent to infer the next state and reward given the current state and action selected. With an accurate model, an agent can plan for the future with a certain amount of confidence. This can lead to a very sample efficient algorithm.

For model-based methods to be effective, an accurate model is required. Unfortunately, obtaining a model of the environment that is sufficiently accurate can be a challenging task. This is especially true for visual POMDP problems [21]. Therefore, in this work we only consider model-free methods.

Model-free methods attempt to find the optimal policy (π_*) through trial and error. By experiencing a state, the value function can be updated to reflect the value of being in that state. Without a model of the environment, the agent needs to experience nearly all possible states and actions to find the optimal policy.

2.3 Value Function

Ultimately, the objective of an MDP is to find the optimal policy that yields the greatest expected return. The return is defined as the discounted sum of rewards over time [1, p. 55]. The equation

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

defines the expected return. Notably, the discounted reward from time t can be defined in terms of the reward at time $t+1$ and the discounted reward from time $t+1$ onward. We exploit this recursive relationship in the equation

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots + \gamma^{T-1} R_T \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots + \gamma^{T-2} R_T \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.2}$$

to obtain G_t in terms of G_{t+1}. The expected return is used to determine the value of being in each state. This is known as the state-value function (v_π). Formally, the state-value function calculates the expected future return G_t for a given state S_t while following a certain policy π from step t onward. The Bellman expectation equation for the value function is defined by

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi \left(S_{t+1} \right) \mid S_t = s \right], \ \forall s \in \mathcal{S}. \end{aligned} \tag{2.3}$$

Note that the π subscript for the expectation is used to indicate that the agent is following policy π.

Thus far, we have only considered the value of being in a certain state. However, the future reward can vary wildly depending on behaviour of the policy. Thus, we also define

the value of each action in a given state from policy π as

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t \mid S_t = s,\ A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi \left(S_{t+1}, A_{t+1} \right) \mid S_t = s, A_t = a \right],\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A}. \end{aligned} \tag{2.4}$$

This is known as the action-value function (q_π). Consider a game of tic-tac-toe. If an agent wins a round, it should learn to recognise preceding states as valuable. Similarly, it should also learn to select the actions in those states that resulted in the victory. The state-value function provides the agent with a sense of how beneficial the current state of the environment is, whereas the action-value function attempts to predict the expected return of taking each action possible in the current state.

An agent can improve the return by acting greedily and updating its value functions with past experiences. We define greedy action as

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, q_\pi(s,a). \tag{2.5}$$

Theoretically, an agent should converge to the optimal policy for a given problem by iteratively updating its state-value function. This iterative update process is defined by

$$v_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}(a,s) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s' \mid s,a) v_k\left(s'\right) \right), \tag{2.6}$$

where s' is the next state and therefore $\mathcal{T}(s' \mid s,a)$ defines the probability of the next state given the current state and performed action.

If an optimal policy is reached, it must follow that both the state-value and action-value functions are also optimal [10]. They are denoted as v_* and q_* respectively. These optimal functions are defined as the maximum value functions over all policies [10]:

$$v_*(s) = \max_\pi v_\pi(s),\ \forall s \in \mathcal{S} \text{ and} \tag{2.7}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a),\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A}. \tag{2.8}$$

While acting greedily seems like a good approach towards finding these optimal functions, the reality is that without a model of the environment, an agent will always select the first-best policy. Without experiencing the full state-action space, we cannot be certain that the agent has discovered the optimal policy. Therefore, we need some way to force the agent into experiencing the full state-action space.

2.4 ϵ-greedy

A major problem in the RL field is the trade-off between exploitation and exploration. If an agent exploits its knowledge of the environment before exploring nearly all state-action

pairs, we cannot be certain that the agent is following the optimal policy. However, exploring the full space may be entirely impossible to do.

A good approach is to force the agent to explore while it is still learning about the environment [1]. As the agent then learns more about the environment, we gradually decrease the amount of exploration done. This is known as ϵ-greedy. At each time step the agent acts either randomly with a probability of ϵ or greedy otherwise.

2.5 Monte Carlo Learning

Monte Carlo (MC) learning solves reinforcement learning problems with empirical sample returns [1]. An agent interacts with the environment until the episode terminates. Upon termination, the agent updates its value function and policy based on the actual states, actions and rewards experienced in the episode. Since the update is done with the unbiased estimate of v_π, MC learning has zero bias. However, updating with a full episode of random transitions means that MC learning also has high variance. This update is defined by [10]

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t)). \tag{2.9}$$

Note the difference in notation, here we use V to indicate an estimate of v_π. Continuing this update for $t \to \infty$ would result in v_π[1]. Ultimately, the value function should be able to accurately estimate the discounted reward G_t. Therefore, G_t is referred to as the target of the update. Note that since an episode is merely a sample trajectory of the environment, we scale the update with α to track a running mean of the updates. Over time, the value function should forget older episodes.

2.6 Temporal-Difference Learning

Unlike Monte Carlo learning, Temporal-Difference (TD) learning updates the value function with incomplete episodes by bootstrapping. Therefore, the target G_t is not directly available. By using Equation 2.2 and estimating G_{t+1} with our value function, TD learning introduces some bias. However, since TD learning updates after a single step, the estimations have much lower variance than MC learning. Both the TD error (δ) and TD learning algorithms are defined by

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \text{ and} \tag{2.10}$$

$$V(S_t) \leftarrow V(S_t) + \alpha \delta_t. \tag{2.11}$$

[1] This is known as the law of large numbers [10].

Equation 2.11 is the simplest form of the TD learning algorithm. However, TD updates are not constrained to a single step. We can recursively expand the target G_t to n steps to give the n-step return,

$$\begin{aligned} G_t^{(1)} &= R_{t+1} + \gamma V(S_{t+1}) \\ G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\ &\vdots \\ G_t^{(n)} &= \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n V(S_{t+n}). \end{aligned} \tag{2.12}$$

This reduces potential for estimation errors by the value function. We then substitute the 1-step target in Equation 2.11 with the n-step target to obtain the update function

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(n)} - V(S_t) \right). \tag{2.13}$$

One advantage of using TD learning over MC learning, is that it exploits the Markov property. The reader may have noticed that if we expand n in Equation 2.12 to span an entire episode, we are effectively doing MC learning. Therefore, we need to select the value of n to effectively balance the variance and bias between TD learning and MC learning. TD learning can also be used on continuous problems, whereas MC learning is limited to episodic problems.

2.7 Q-Learning

Thus far, we have only considered learning to estimate the state-value function with samples from the environment. However, agents need control over their actions to interact with the environment and improve the rewards received. Watkins [22] introduced Q-Learning: a model free, off-policy TD control algorithm.

The Q-learning algorithm learns to estimate the action-value function by evaluating a target policy π while following a behaviour policy μ. This algorithm is defined by

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right), \tag{2.14}$$

where $s = S_t$, $a = A_t$, $s' = S_{t+1}$, $a' = A_{t+1}$ and $r = R_{t+1}$. Note that a and a' are sampled from policies μ and π respectively. As shown in Equation 2.14, Q-Learning sets the TD target to the Q-value of the greedily selected action from the target policy. This is done because the Q-Learning algorithm considers the target policy to be the optimal behaviour. As an agent obtains and learns from more samples through control, the target network converges to q_*. Therefore an agent behaves independent from its approximation of q_*. It only relies on the behaviour policy for collecting samples.

Off-policy methods like Q-learning are useful when we want to learn a policy from human demonstration data. The demonstration data can be used to bring the target policy closer to q_*. The behaviour policy can then still be used to experience the environment without affecting the learning process. Additionally, it also allows us to reuse old experiences generated by previous policies.

2.8 Hierarchical Reinforcement Learning

In HRL, we divide an RL problem into a hierarchical set of sub-tasks. Each of these sub-tasks can then be viewed as RL problems where a policy learns to maximise its own return before termination. Together these policies should help the agent to reach its final goal and solve the original problem.

To gain a better understanding of why a hierarchical set of policies may be necessary, we can turn to the previously described example of making a cup of tea. The sub-tasks can then be the following:

- Fill the kettle with water and start boiling.
- Collect a cup.
- Collect and add tea to the cup.
- Add hot water and allow time to brew.
- Collect completed beverage.

Each of these sub-tasks represents a macro-action that requires several micro-actions to solve. Notice that certain sub-tasks do not need to occur in the specified order. It is not required to boil the water before collecting the other items, it is merely more efficient to do so since waiting for the kettle to boil wastes time. Other sub-tasks however, require previous tasks to be completed before even being attempted.

While each sub-task may seem easy, controlling a physical entity to execute these tasks is by no means a simple task. Moving in the correct direction and grasping the required items may involve manipulating complicated hardware. Additionally, all policies may not need to learn all states and actions to solve their respective sub-tasks leading to more specialised policies.

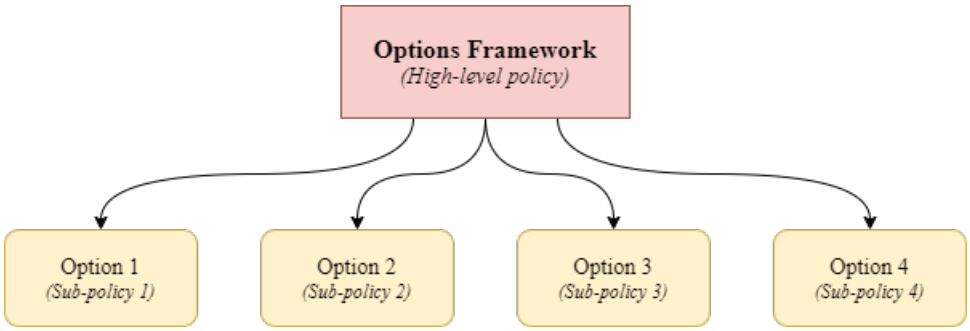

Figure 2.2: Block diagram of an Options Framework. The framework selects an option to control the agent until that option terminates. Unless the episode is also terminated, the next option is selected.

2.8.1 Options Framework

One HRL method we consider in this work is the options framework. In Equation 2.13 we can see that the TD error is not constrained to a single step in the environment. It can be taken over an arbitrary finite series of sequential steps. The options framework exploits this by defining each sub-task as a single step on the path towards reaching the final goal. These steps are otherwise referred to as temporally-extended actions. At each of these steps a low-level policy is selected to solve the next sub-task. Upon solving the sub-task or reaching the final goal, the framework selects another option to solve the next sub-task.

Each option is defined by a tuple of three components: $\langle I_\omega(s), \pi_\omega(a|s), \beta_\omega(s)\rangle$ [23]. These are the initiation function, low-level policy and termination function, respectively. The initiation function defines a set of conditions in which a given option can initiate. The termination function defines the conditions upon which an option can terminate. A block diagram depicting an options framework can be seen in Figure 2.2.

As stated previously, an option is a temporally extended action. We can train the high-level policy upon the termination of an option by slightly modifying Equation 2.14 to use options as actions. The value function update for a high-level policy is defined by

$$Q\left(s,o\right) \leftarrow Q\left(s,o\right) + \alpha\left(r + \gamma^k \max_{o'} Q\left(s',o'\right) - Q\left(s,o\right)\right), \tag{2.15}$$

where $o = O_t$, $s' = S_{t+k}$, $o' = O_{t+k}$ and O is an option from finite set of options ($\mathcal{G}$). Using this equation, we can train the high-level policy with Q-learning.

In the case of MineRL for example, we can define an option for collecting wood. The initiation function I_ω would trigger when the agent runs out of wood and requires more to reach the final goal. The agent would then follow the sub-policy π_ω of the option until enough wood is collected and the option is terminated by the termination function β_ω or episode termination. Theoretically, the options framework should be flexible enough for

this to occur as the agent requires the items, rather than a fixed series of sub-tasks. Before we can apply this algorithm to a problem, we need a strategy to discover the options.

Option Discovery. In their paper proposing the options framework, Sutton *et al.* [23] addressed the issue of long-term planning and learning. However, they did not address the issue of discovering these options. This is a particularly challenging task since an option is very loosely defined—it can involve collecting a log or all tasks required to craft a wooden pick-axe. Logically the options should be selected to maximise the expected reward. In Section 5.2 we discuss our solution to this problem.

2.9 Summary

In this chapter, we introduced background information on RL pertaining to this work. We discussed MDPs and why RL problems need to satisfy the Markov property. In cases where the full state of the environment is not always observable, we also discussed the idea of a POMDP. This finally lead to the belief MDP relevant to approaching a solution to our problem.

We then covered value functions, Monte Carlo learning and TD learning. With an understanding of these concepts, we described the Q-learning algorithm used in this work. Lastly, we discussed HRL (specifically the options framework) as a method of splitting the task into several sub-tasks to be solved by a sub-policy.

3 Neural Networks

Although traditional tabular RL methods are effective at solving low dimensional tasks, the complexity increases exponentially in proportion to the dimensionality of the observation- and action space. To overcome the problem of exploding dimensions, Mnih *et al.* [3] developed the DQN algorithm. They combined NNs with RL methods to create an artificial agent capable of learning directly from visual data. Their DQN algorithm proved capable of performance comparable to that of professional game testers.

Given the observation space of the MineRL environment, attempting to solve the problem using tabular methods would not be possible. Following the work of Mnih *et al.* [3] and Hessel *et al.* [7], we implement neural network architectures in our solutions to form lower dimensional representations of raw images. This theoretically enables an agent to process the contents of a POV observation to select the best action. In this chapter we discuss the theory required to implement a neural network architecture for an agent.

3.1 Feed-Forward Neural Networks

In Figure 3.1 we show a simple NN. Each node in the network is known as an artificial neuron. These nodes represent a mathematical function applied to a set of input values. In Figure 3.2, we show a single artificial neuron with an input vector $\boldsymbol{x}$, a weight vector $\boldsymbol{w}$, a bias b, an activation function g and an output y.

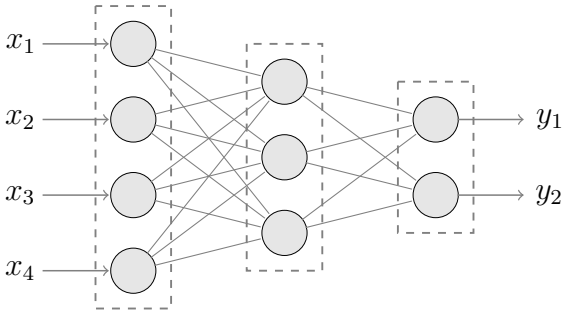

Figure 3.1: Simple feed-forward fully connected neural network. Each circle represents a single neuron and each edge a weighted connection. Layers are demarcated by dashed lines.

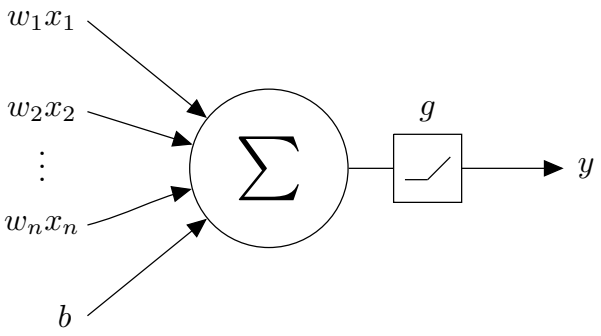

Figure 3.2: Depiction of an artificial neuron within a neural network. The input vector $\boldsymbol{x}$ is multiplied with a similarly sized weight vector W. The components are then summed and passed through a certain non-linear function g.

The neuron calculates the weighted sum of the input values and applies an activation function to the result. The result y is defined by

$$y = g\left(\sum_{k=1}^{n} w_k x_k + b\right). \tag{3.1}$$

However, this becomes a costly operation when we arrange thousands of neurons in parallel to calculate a vector $\boldsymbol{y}$. We can overcome this issue by extending the weights- and bias vectors to include all neurons in a layer of parallel neurons. The calculation for a layer can then be represented with a matrix multiplication as defined by

$$\boldsymbol{z}(\boldsymbol{x}) = W^T\boldsymbol{x} + \boldsymbol{b} \text{ and} \tag{3.2}$$

$$\boldsymbol{a}(\boldsymbol{z}) = g(\boldsymbol{z}). \tag{3.3}$$

Note that we separate the application of an activation function to a new equation. For an input vector of size m and a layer with n neurons, we have: $\boldsymbol{z} \in \mathbb{R}^{n\times 1}$, $W^T \in \mathbb{R}^{n\times m}$, $\boldsymbol{x} \in \mathbb{R}^{m\times 1}$ and $\boldsymbol{b} \in \mathbb{R}^{n\times 1}$. Now that we have a more efficient way of calculating the resulting vector from a layer of neurons, we can arrange several layers to create an NN. If the layers are ordered sequentially from input to output, it is known as a feed-forward neural network (FFNN).

Figure 3.3 shows an FFNN with three layers. The layer of green neurons is known as the input layer. This layer does not perform any operation and its presence is merely to indicate the inputs visually in the network. The next layer of red neurons is referred to as the hidden layer. It performs Equation 3.3 on the output of the preceding layer. While Figure 3.3 only depicts a single hidden layer, we can add as many as the hardware can handle[1]. The final layer of blue neurons provides us with the output of the network after again applying Equation 3.3 to the output of the hidden layer.

Upon a full pass through the network, we need a metric for comparing the actual output $\boldsymbol{y}$ to the desired output $\hat{\boldsymbol{y}}$ (ground truth) given a certain input. This metric will then

[1]Note that adding additional layers or more neurons is not always beneficial.

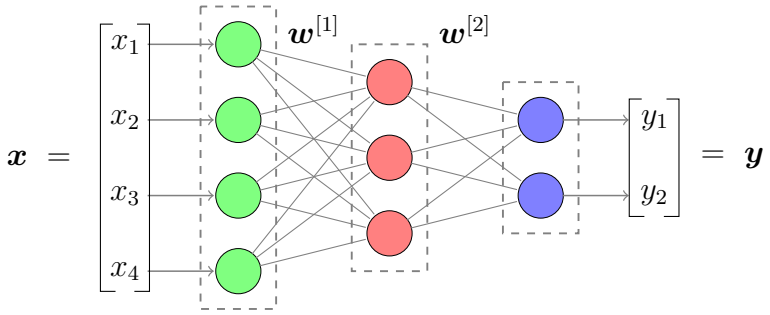

Figure 3.3: Example of a feed-forward neural network with 3 layers of neurons. The green layer represents the input to the network. The edges connecting the nodes represent the neural network's layers. The final layer then outputs the result.

determine how we modify the weight vector $\boldsymbol{w}$ during the backward pass over the network. The metrics we use for this purpose are known as loss functions.

3.2 Loss Functions

A loss function provides a metric for modifying the weights of a network to yield outputs closer to what we expect. The metric used as a loss function depends entirely on the goal of the neural network. For regression, the least absolute error (LAE) or least squares error (LSE) functions deliver good performance. For classification, the cross-entropy loss can be used. Although there are many other loss functions, we only focus on the above-mentioned and variants of them.

Note that in machine learning, we typically process a batch of samples in parallel for training. We therefore also define the cost of each of the loss functions.

Least absolute error simply calculates the absolute distance between the network output and the ground truth. As the LAE decreases, the network outputs should become closer to the expected ground truths. However, convergence to zero may be a difficult task since the gradient is consistent for both large and small errors. When we expand the LAE loss function to a batch of samples, we obtain the mean absolute error (MAE) cost function. These functions are defined by

$$L_{LAE}(x, y) = |x - y| \text{ and} \tag{3.4}$$

$$J_{MAE}(x, y) = \frac{1}{N} \sum_{k=1}^{N} L_{LAE}(x_k, y_k). \tag{3.5}$$

Least squares error attempts to address the gradient issue of LAE at lower errors. Instead of the absolute difference, LSE squares the difference. The squared difference introduces a smooth step size in the gradient for errors smaller than one. It also heavily penalises errors larger than one when compared to LAE. This unfortunately leads to LSE being more sensitive to large outlier errors. By expanding the LSE loss function to a batch, we obtain the mean squared error (MSE) cost function. The LSE loss and MSE cost functions are defined by

$$L_{LSE}(x, y) = (x - y)^2 \text{ and} \tag{3.6}$$

$$J_{MSE}(x, y) = \frac{1}{N} \sum_{k=1}^{N} L_{LSE}(x_k, y_k). \tag{3.7}$$

Huber loss combines LAE and LSE to address the issues of both. It uses the smoother gradients of LSE for errors smaller than one and the more stable gradients of LAE otherwise. Huber loss is therefore not as sensitive to large errors and has smoother gradients close to zero. We can follow the same approach as shown in Equations 3.6 and 3.7 above to obtain the Huber cost function. The Huber loss function is defined by

$$L_{HUBER}(x, y) = \begin{cases} 0.5\,(x - y)^2\,, & \text{if } |x - y| < 1 \\ |x - y| - 0.5, & \text{otherwise.} \end{cases} \tag{3.8}$$

Cross-entropy loss treats each element of the output vector as the log-odds of being the correct label in a classification problem. The ground truth for these problems is then a one-hot encoded vector of the correct class. The cross-entropy loss should therefore reduce the likelihood of incorrect predictions while increasing correct prediction likelihoods. The cross-entropy loss is defined by

$$L_{CE}(x, y) = -\sum_{k=1}^{N_{ce}} y_k \log\left(x_k\right). \tag{3.9}$$

Again, we can expand the cross-entropy loss for a batch of samples by simply taking the mean of all the elements.

3.3 Activation Function

The activation function is a critical feature of an NN. If a non-linear activation function is used, it allows a network to compute more complicated functions. If the activation function is excluded or linear, the network will be limited to learning linear functions. To

better understand why this is, we can look at

$$\begin{aligned} a^{[1]} &= W^{T^{[1]}}x + b^{[1]} \\ a^{[2]} &= W^{T^{[2]}}a^{[1]} + b^{[2]} \\ a^{[2]} &= W^{T^{[2]}}W^{T^{[1]}}x + W^{T^{[2]}}b^{[1]} + b^{[2]} \\ a^{[2]} &= W^{'}x + b^{'} \end{aligned} \tag{3.10}$$

for the example network shown in Figure 3.3 [24]. Note that we use $a^{[\ell]}$ to denote the output of the activation function of layer ℓ.

Looking at final answer in Equation 3.10, we find that we end with a familiar linear function. The new variables $W^{'}$ and $b^{'}$ are simply new constants created from multiplying the weights and biases of previous layers in a specific way. This demonstrates the importance of including a non-linear activation function in our neural network. In this work we only consider the rectified linear unit (ReLU) and the softmax function. The ReLU- and softmax activation functions are defined by

$$g(z) \;=\; \max(0,\; z) \;\text{ and} \tag{3.11}$$

$$g(z) \;=\; \frac{e^{z}}{\sum_{j} e^{z_j}}, \tag{3.12}$$

respectively. Due to its simplicity in both forward- and backward propagation, we use the ReLU function for nearly all hidden layers in our networks. We use the softmax function to normalise the output of the network to a probability distribution.

3.4 Backpropagation

During the training phase of our network, we iteratively update the weights and biases of the network to come closer to approximating the underlying function. To update the network, we first perform a full forward pass and loss calculation of a batch of data. The scalar loss ($\mathcal{L}$) is then propagated backward through the network using gradient descent. However, updating the network with the full value of the loss gradients would result in updates too large for the network to converge. The resulting network would deliver sub-optimal performance and would not be capable of generalising across a variety of scenarios. Therefore we scale the updates with a parameter called the learning rate (α).

The learning rate is typically limited between 0 and 1. A small learning rate will extend the training time and potentially get stuck in a local minimum loss. A large learning rate will learn faster, at the cost of generalisation and missing the global minimum loss entirely. Selecting the learning rate is therefore a trade-off between performance, training time and generalisation.

We define the weight and bias update rules as

$$W_{new} = W_{old} - \alpha \frac{\partial L}{\partial W_{old}} \text{ and} \tag{3.13}$$

$$b_{new} = b_{old} - \alpha \frac{\partial L}{\partial b_{old}}. \tag{3.14}$$

Using these update rules, we can calculate the new weights and biases of each layer. To obtain $\frac{\partial L}{\partial W}$, we start by calculating the loss with the output of the last layer ($a^{[2]}$). We then derive backwards through the network with the loss value to update the weights and biases of each layer. This is done by using the chain rule shown in Equation 3.15. The components of the chain rule are calculated in Equations 3.16 and 3.17 and when substituted back in, results in Equation 3.18 [24]. The bias derivative can be calculated in a similar fashion to yield Equation 3.19.

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial W^{[2]}} \tag{3.15}$$

$$\frac{\partial L}{\partial a^{[2]}} = \frac{\partial}{\partial a^{[2]}} \frac{1}{2} \left(y - a^{[2]}\right)^2 = a^{[2]} - y \tag{3.16}$$

$$\frac{\partial a^{[2]}}{\partial W^{[2]}} = \frac{\partial}{\partial W^{[2]}} \left(W^{T^{[2]}} a^{[1]} + b^{[2]}\right) = a^{[1]} \tag{3.17}$$

$$\therefore \frac{\partial L}{\partial W^{[2]}} = (a^{[2]} - y) \cdot a^{[1]} \tag{3.18}$$

$$\frac{\partial L}{\partial b^{[2]}} = (a^{[2]} - y) \cdot 1. \tag{3.19}$$

We can follow a similar approach with the preceding layers. However, to simplify calculations we introduce additional applications of the chain rule as shown in Equation 3.20. Equations 3.21 to 3.24 below calculate each of the components of the chain rule [24]. The result is then shown in Equation 3.25.

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial W^{[1]}} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial W^{[1]}} \tag{3.20}$$

$$\frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} = (a^{[2]} - y) \cdot 1 \tag{3.21}$$

$$\frac{\partial z^{[2]}}{\partial a^{[1]}} = W^{T^{[2]}} \tag{3.22}$$

$$\frac{\partial a^{[1]}}{\partial z^{[1]}} = \frac{\partial}{\partial z^{[1]}} \text{ReLU}(z^{[1]}) = 1 \ , \quad \text{for } z^{[1]} > 0 \tag{3.23}$$

$$\frac{\partial z^{[1]}}{\partial W^{[1]}} = a^{[0]} = x \tag{3.24}$$

$$\therefore \frac{\partial L}{\partial W^{[1]}} = (a^{[2]} - y) \cdot W^{T^{[2]}} x \ , \quad \text{for } z^{[1]} > 0. \tag{3.25}$$

3.5 Gradient Descent Optimisers

During the gradient descent update step, we update the weights and biases of our network with the rules defined in Equations 3.13 and 3.14. With batch gradient descent, the update step is done on the full dataset. This can be a slow and costly operation depending on the size of the dataset. Stochastic gradient descent (SGD) attempts to address this by updating after each sample. However, the frequent updates make for a very noisy descent towards the minimum point in the cost function. Mini-batch gradient descent balances the two above-mentioned issues by splitting the dataset into smaller, more manageable mini-batches. After each forward pass of a mini-batch through the network, the weights and biases of the network are updated with gradient descent. This leads to a faster and less noisy descent towards the minimum.

In this work we used the Adam optimiser. The Adam optimiser combines several other improvements to the original SGD algorithm. In the rest of this section we briefly discuss each of these improvements leading up to the final Adam optimiser.

Momentum gradient descent modifies the update rules by replacing the derivatives $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$ with their respective moving averages. The moving averages assist to counter the noisy updates introduced by SGD. Combined with mini-batch gradient descent, momentum brings the SGD gradient descent closer to the full batch gradient descent. The new update rules are defined by

$$v_w = \beta_1 v_w + (1 - \beta_1)\frac{\partial L}{\partial W} \tag{3.26}$$

$$W = W - \alpha v_w \text{ and} \tag{3.27}$$

$$v_b = \beta_1 v_b + (1 - \beta_1)\frac{\partial L}{\partial b} \tag{3.28}$$

$$b = b - \alpha v_b. \tag{3.29}$$

Root mean squared propagation (RMSProp) gradient descent implements a similar approach to the momentum algorithm described above. However, the moving average becomes exponentially weighted. The idea is then to dampen the effects of large updates particularly in directions not leading to the minimum point. Note that we introduce a small constant ϵ to prevent the denominator from being zero. The update rules for

RMSProp optimisation are defined by [25]

$$s_w = \beta_2 s_w + (1 - \beta_2) \left(\frac{\partial L}{\partial W} \right)^2 \tag{3.30}$$

$$W = W - \alpha \frac{\partial L}{\partial W} \frac{1}{\sqrt{s_w} + \epsilon} \text{ and} \tag{3.31}$$

$$s_b = \beta_2 s_b + (1 - \beta_2) \left(\frac{\partial L}{\partial b} \right)^2 \tag{3.32}$$

$$b = b - \alpha \frac{\partial L}{\partial b} \frac{1}{\sqrt{s_b} + \epsilon}. \tag{3.33}$$

Adaptive moment estimation (Adam) combines all the above improvements discussed above. By combining RMSProp and Adam with SGD and mini-batches, they are able to effectively reduce oscillation during gradient descent. Ng [25] describes the Adam algorithm as very effective for a wide variety of problems and architectures. The update rules for the Adam optimiser are defined by

$$W = W - \alpha \frac{v_w}{\sqrt{s_w} + \epsilon} \text{ and} \tag{3.34}$$

$$b = b - \alpha \frac{v_b}{\sqrt{s_b} + \epsilon}. \tag{3.35}$$

Note that v_w, v_b, s_w and s_b are defined above in Equations 3.26, 3.28, 3.30 and 3.32 respectively.

3.6 Convolutional Neural Networks

Unfortunately, fully-connected NNs do not perform well when presented with a flattened 2-dimensional (2D) visual input, as reducing images to a single dimension may obscure crucial spatial information contained within the images. To overcome this issue, CNNs use learnable sliding filters to extract desirable information from an image. Figure 3.4 shows a 3×3 filter sliding over a 7×7, single channel 2D image with a stride of 2 to produce a 3×3 output feature map. This technique is similar to many computer vision techniques where fixed filters perform, among others: edge detection, blurring and sharpening. Figure 3.5 below depicts a sample 2D RGB image input.

CNNs can stack several $n \times m$ filters in the channel dimension to detect useful features from the input. For example, the filters of the first convolutional layer may detect edges in various directions. The filters of the following convolutional layers can then combine these low level features to detect higher level features such as eyes, faces and people [25].

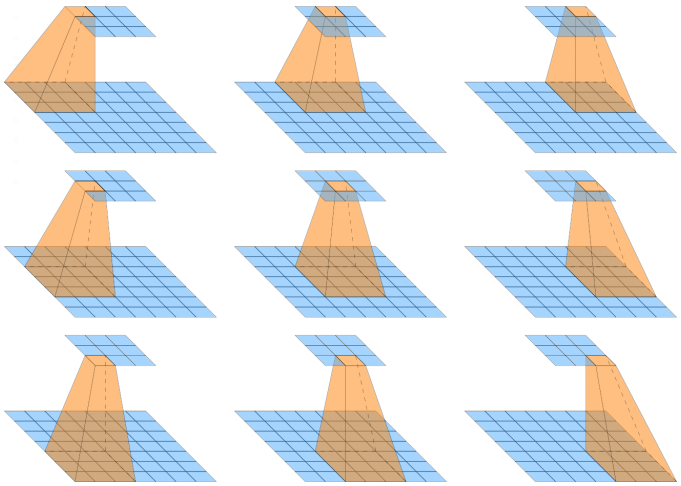

Figure 3.4: Representation of a 3×3 convolutional filter sliding over a single channel 2D input with a stride of 2. The filter is applied on a 7×7 input to produce a 3×3 output in 9 steps.

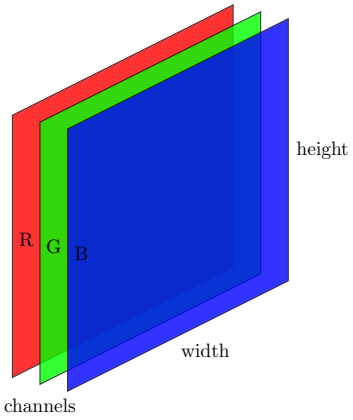

Figure 3.5: Sample 2D RGB image input to CNN network.

In the yearly ImageNet Large Scale Visual Recognition Challenge (ILSVRC), participants are challenged to classify a large dataset of labelled images. In ILSVRC 2012, Krizhevsky *et al.* [26] proposed using CNNs to augment deep NNs. They reported significant advances in classification performance of the ImageNet dataset. Since their breakthrough, recent advances has reduced the top 5 error rate from around 30% [26] to 1.3% in 2020 [27].

Inspired by the work of Mnih *et al.* [3] and many others after them, we use CNNs to extract features from the POV observation obtained from the MineRL environment. Ideally, the network would learn to detect trees, stone, iron ore, etc. These features are then processed by several layers of feed-forward linear layers before producing an estimate of the Q-values.

3.7 Summary

In this chapter, we covered the relevant background information of NNs. We discussed the artificial neuron, the building block of NNs. From there we briefly looked at the loss and activation functions used by the NNs in this work. We then described the backpropagation process and various gradient descent optimisers. Finally, we discussed the CNNs required to process visual observations.

4 Literature Review

At the conclusion of the MineRL challenge, the top two teams documented their submissions in technical reports and accompanying GitHub repositories. In this chapter, we critically review the work of these teams. We also determine whether or not their solutions include techniques beneficial to our implementation. However, before doing so, we need to discuss the various algorithms which they build on. This includes BC, DQN and various improvements to DQN.

4.1 Deep Q-Networks

In 2015, Mnih *et al.* [3] developed an algorithm capable of human-level performance in the Atari 2600 games suite [11]. They limited the observation space to the raw visual data as observed by human players. Instead of using a lookup table for the Q function, the DQN algorithm uses a deep neural network to approximate the value of each action for a given state. Mnih *et al.* [3] describes two modifications, namely experience replay and a target network, that enable training large NNs without diverging. In addition to these two methods, they also implemented frame-skipping and frame stacking to address problems like partial observability, motion detection and experience generation.

Experience Replay. One problem we face when using NNs to estimate the Q-values, is the correlation of subsequent states. If we simply train our network on the most recent data, the network would become biased in favour of the correlated data. For this reason, the authors introduced experience replay.

Experience replay involves training the Q-function NN on past experiences observed by the agent, thereby training on uncorrelated data. For each transition observed, we store a tuple $\langle s, a, r, s' \rangle$ in a buffer called the replay memory ($\mathcal{D}$). In a training step, we then sample past experiences from the memory at a uniform rate. These experiences are then used to update the online network with the Bellman equation defined in Equation 2.14.

Target Network. In Equation 2.14, the action used for the TD target is sampled from another policy. Mnih *et al.* [3] used a separate network cloned from the online network to compute the TD target. The target network helped to stabilise training by fixing the TD target rather than calculating the prediction error of the network with respect to itself. Every C steps, they update the target network weights θ^- with the online network weights θ. Mnih *et al.* [3] defines the DQN algorithm as

$$Q\left(S_t, A_t; \theta\right) \leftarrow Q\left(S_t, A_t; \theta\right) + \alpha\delta_t \text{ and} \tag{4.1}$$

$$\delta_t = R_{t+1} + \gamma \max_{a'} Q\left(S_{t+1}, a'; \theta^-\right) - Q\left(S_t, A_t; \theta\right). \tag{4.2}$$

Frame-skipping. Mnih *et al.* [3] implemented the frame-skipping technique to increase the rate at which new experiences are generated. The authors stated that it is much more computationally expensive to select an action than it is to perform a step in the environment. For every transition seen by the agent, we repeat the last action for k environment steps. Since we only select an action every k steps, we can experience roughly k times more games in a similar time span. In addition to the aforementioned, frames immediately following the current frame tend to be very similar (only differing by a couple of pixels in some cases). Skipping k frames means that we are more likely to observe a frame which is less identical.

Frame stacking. Since the observation received from the environment is only a single snapshot in time of the agent's POV, we may not be able to detect motion. For example in the Atari game of Pong, if we receive an observation with the ball in the middle of the field, how do we know which direction the ball is moving? Or at what speed? To help overcome both partial observability and an inability to detect motion, Sutton and Barto [1] suggests stacking the last m frames observed. This also makes the problem more Markovian.

In Figure 4.1 we show the Nature CNN architecture design by Mnih *et al.* [3] for the DQN algorithm. They employed three CNN layers in sequence to process image inputs. The architecture layers are specified in Table 4.1. Unless specified otherwise, we used the ReLU activation function for all layers except the output layer. Since we do not modify this design, we depict the CNN tail as a block module. In all subsequent figures showing modifications to the head of the network, the reader may assume the head is preceded by the CNN tail module. The default DQN head can then be seen in Figure 4.2.

Although the DQN algorithm delivered human-level performance in the Atari games suite, it still suffered from various flaws or shortcomings. In the rest of this section, we discuss several methods of addressing the issues with the DQN algorithm.

Table 4.1: Layer specifications for the nature CNN architecture proposed by Mnih *et al.* [3].

Layer	Channels	Filter size	Stride	Activation
1	32	8×8	4	ReLU
2	64	4×4	2	ReLU
3	32	3×3	1	ReLU

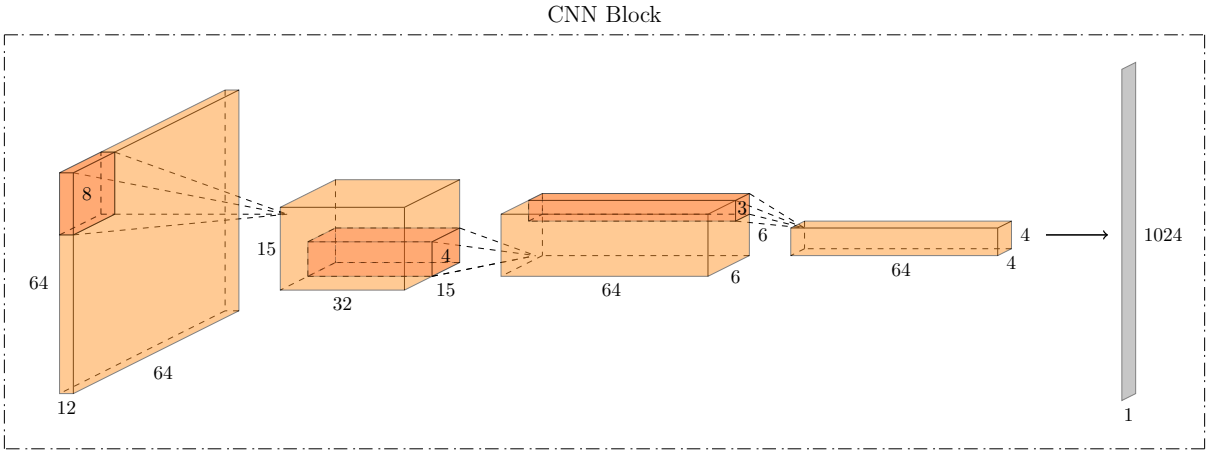

Figure 4.1: Nature CNN architecture based on the work of Mnih *et al.* [3]. The CNN is surrounded by a border to indicate that this model will be used as a module in other depictions.

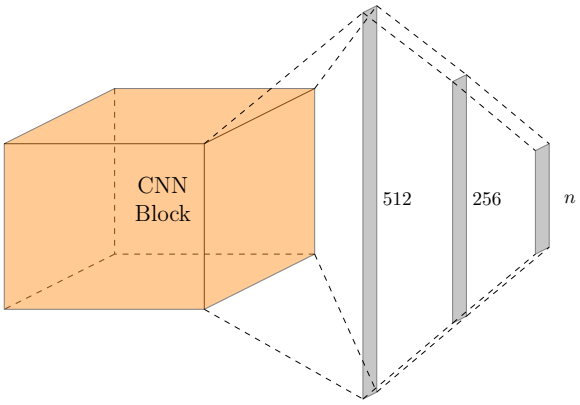

Figure 4.2: DQN network head based on the work of Mnih *et al.* [3]. This model continues from the final layer of the CNN block shown in Figure 4.1. Each component in the output predicts the Q-value for a certain action.

4.1.1 Double DQN

As noted by van Hasselt *et al.* [28], the original DQN algorithm is prone to overestimating action values in certain situations. They attribute the overestimation to the usage of the same Q-values for both action selection and evaluation. They proposed decomposing the selection and evaluation of the action.

Double Deep Q-Networks (DDQN) modify Equation 4.2 by selecting the TD target action with the online network rather than the target network. The evaluation is then still done on the target network Q-values. Formally, the TD error for the DDQN algorithm is defined by

$$\delta_t^{\text{DoubleQ}} = R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname*{argmax}_a Q\left(S_{t+1}, a; \theta\right); \theta^-\right) - Q\left(S_t, A_t; \theta\right). \tag{4.3}$$

van Hasselt *et al.* [28] found their improvement to the DQN algorithm to successfully reduce the overestimation problem faced by the DQN algorithm. Additionally, their evaluations on the Atari benchmark showed that DDQN performed better or at least on par with DQN.

4.1.2 Prioritised Experience Replay

In the original DQN algorithm, the authors used an experience replay buffer to store past experiences. When updating the online network, they uniformly sampled experiences from the replay buffer. However, some experiences are more valuable than others. Sampling at a uniform rate means we are just as likely to sample unimportant experiences. This is especially problematic in a sparse reward environment, where valuable experiences are not experienced very often. Schaul *et al.* [29] proposed prioritising the sampling of valuable experiences from the buffer. They measure the value of an experience based on the TD error calculated during the update step. The probability of sampling transition i from the buffer is defined by

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{4.4}$$

$$p_i = |\delta_i| + \epsilon. \tag{4.5}$$

In Equations 4.4 and 4.5 above, α controls the amount of prioritisation used and ϵ adds a small positive constant to the probability to ensure a non-zero value. However, in changing the distribution of sampling, they introduce some bias in the estimation of an expected value. Schaul *et al.* [29] corrected the bias with importance-sampling weights,

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta, \tag{4.6}$$

where β anneals from a certain starting value β_0 to 1 as the agent nears the end of training.

The Prioritised Experience Replay (PER) improvement proved to be a valuable addition to the DQN algorithm. Schaul *et al.* [29] found prioritised sampling to speed up learning by a factor of 2 in addition to improved performance. At the time, PER produced SOTA performance on the Atari benchmark.

4.1.3 Duelling DQN

In certain scenarios, the agent finds itself in a state where the action selected has little to no consequence on the future reward. In these cases, the difference between the Q-values of the actions is minimal compared to the magnitude of the Q-values. Wang *et al.* [4] proposed a solution to this problem since estimating the state values is considered important when using bootstrapping based algorithms. Their solution involved computing the value of the state and the impact of each action separately.

They accomplished the separation by splitting the DQN NN head into two branches: one to calculate the state value (V^π) and another to calculate the advantage of each action (A^π). Therefore, instead of directly calculating the Q-value of each action, they calculate the value of the state and the advantage of each action on the state-value separately. The two branches are then combined to calculate the Q-value of each action. This split is illustrated in Figure 4.3. The new action value function is defined by

$$\begin{aligned} Q(s,a;\theta) &= V^\pi(s;\theta) + \left(A^\pi(s,a;\theta) - \frac{1}{|\mathcal{A}|} \sum_{a'} A^\pi(s,a';\theta) \right) \\ &= V^\pi(s;\theta) + \left(A^\pi(s,a';\theta) - mean(A^\pi(s,a';\theta)) \right), \end{aligned} \tag{4.7}$$

where $|\mathcal{A}|$ is the number of possible actions. Wang *et al.* [4] found their improvement to further refine the DQN algorithm and produce SOTA performance in the Atari benchmark at the time. Additionally, they also showed their algorithm can be applied alongside the two improvements previously discussed in this chapter. Figure 4.3 shows how this improvement modifies the NN head.

4.1.4 Categorical DQN

Thus far we have only considered estimating the expected future reward. However, estimating the expectation directly obfuscates the intrinsic randomness of the reward function. For example, in MineRL, when a block breaks it falls in a random direction. Since the agent cannot be certain whether or not it will collect the block in the next step, the expected reward will therefore be some value smaller than one. In reality the reward is either one or zero, we will never receive a reward $0 < r < 1$. Instead of taking the expectation, Bellemare *et al.* [5] proposed modelling the distribution of the return. They define Q as the expectation over a certain distribution for random variable Z. This distribution is then used to define the distributional Bellman equation:

$$Z(s,a) = R(s,a) + \gamma Z(s',a'). \tag{4.8}$$

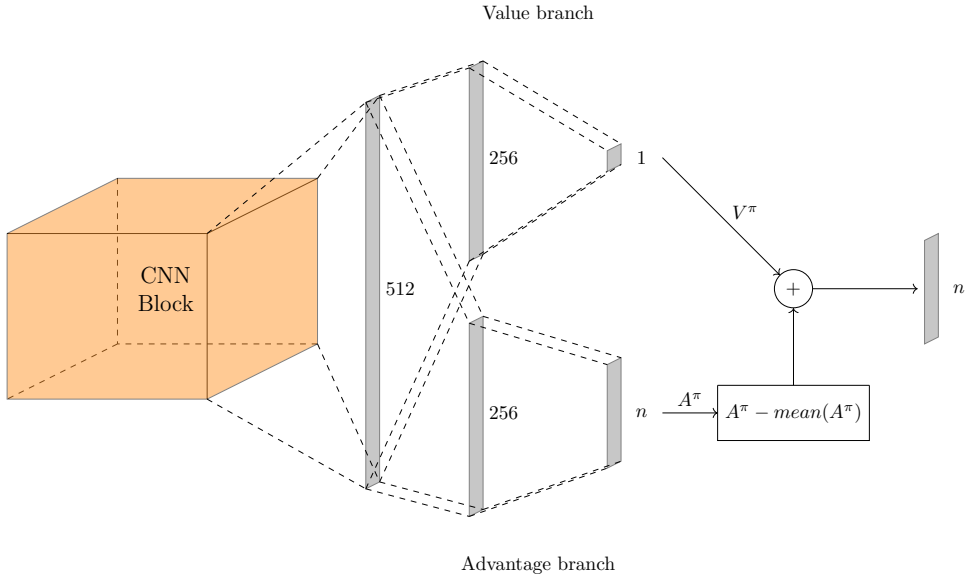

Figure 4.3: Duelling DQN network head based on the work of Wang *et al.* [4]. The network separates the predictions for the state values and action advantages. The two streams then combine after removing the mean of the action advantages to obtain the action values.

In the work of Bellemare *et al.* [5], the authors defined the Bellman operator ($\mathcal{T}^\pi$) as a contraction mapping. The contraction mapping theorem states that an operator converges to a unique fixed point at a rate of at least γ [10]. Theoretically, applying the operator $\mathcal{T}^\pi$ infinitely to some initial Q would result in q^π. Similarly, applying the Bellman operator ($\mathcal{T}$) infinitely to Q would result in q^*. Note that the two aforementioned statements are only true if the operator is a contraction mapping and Q is a closed space under the operator. In the equation

$$\mathcal{T}^\pi Z(s,a) = R(s,a) + \gamma P^\pi Z(s',a'), \tag{4.9}$$

the authors attempted to apply the Bellman operator to the distributional value function Z, where P^π is the transition operator that defines transitioning from the current state to each next state. Unfortunately, they proved that unlike the normal Bellman operator, the distributional Bellman operator is not a contraction. This means that convergence to a stationary optimal policy is not guaranteed. They did however prove convergence to a non-stationary set of optimal value distributions—albeit slow and not guaranteed.

Bellemare *et al.* [5] modelled Z with a discrete distribution—conveniently allowing an NN to estimate the values in the distribution. This distribution is clamped between maximum- and minimum values V_{max} and V_{min}, respectively. They then create a support vector for the distribution by dividing the range between V_{max} and V_{min} into equal parts according to the number of atoms N_{atoms} selected. An atom is defined as the probability that the future

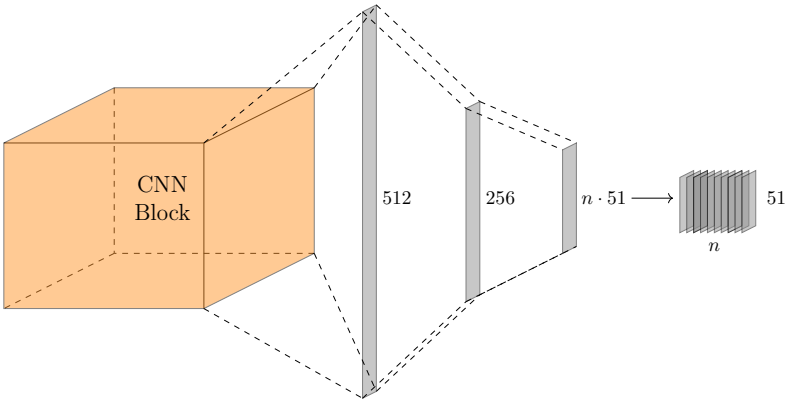

Figure 4.4: C51 DQN network head based on the work of Bellemare *et al.* [5]. Instead of estimating the expected value, the network predicts a value distribution for each action. The expectations can then be obtained directly from the distributions.

reward will correspond to a certain value z_i. Each atom in the support vector is defined by

$$\begin{aligned} z_i &= V_{\min} + i\Delta z\,,\ 0 \leq i < N\,;\ i \in \mathbb{N} \\ \Delta z &= \frac{V_{\min} - V_{\min}}{N_{\text{atoms}} - 1}. \end{aligned} \tag{4.10}$$

The estimated distribution would then describe the probability of the rewards corresponding to each atom's value. Interestingly, their findings indicate a selection of 51 atoms to perform exceedingly well. The algorithm has therefore been referred to as C51.

Bellemare *et al.* [5] then used the result of Equation 4.9 as the target for the network. They minimise the Kullback-Leibler (KL) divergence between the estimated- and target distributions. However, to calculate the KL divergence loss, both distributions need to have the same support. They address this issue by projecting the target distribution back on to the support of the estimated distribution with linear interpolation. The loss is then defined by

$$L(s,a) = D_{\text{KL}}\left(\Phi\mathcal{T}Z(s,a;\theta^-)\,||\,Z(s,a;\theta)\right), \tag{4.11}$$

where Φ is the projection that is applied to the target distribution. The resulting algorithm, C51, outperformed a combination of all the previous DQN improvements, delivering SOTA performance on the Atari benchmark at the time. In their conclusion, the authors noted several possible reasons to learn a distribution, namely: increased stability by averaging the different distributions, a richer set of predictions and well behaved optimisation through a KL divergence loss. In addition to these reasons, a distribution also enables an algorithm to manage risk (see Appendix B). Figure 4.4 shows how C51 modifies the NN head. For each action, we compute the value of all 51 atoms.

4.1.5 Noisy Networks for Exploration

Noisy Networks were designed to replace the traditional ϵ-greedy exploration method. Fortunato *et al.* [30] proposed modifying feed-forward layers in the NN by adding noise sampled from normal distributions. Specifically, they add noise, scaled by learnable parameters[1], to Equation 3.2 to obtain Equation 4.12. The modified equation is defined by

$$\boldsymbol{z}(\boldsymbol{x}) \;=\; \left(W^T + \boldsymbol{\sigma}^{\boldsymbol{w}} \odot \boldsymbol{\epsilon}^{\boldsymbol{w}}\right) \boldsymbol{x} + \left(\boldsymbol{b} + \boldsymbol{\sigma}^{\boldsymbol{b}} \odot \boldsymbol{\epsilon}^{\boldsymbol{b}}\right), \tag{4.12}$$

where $\odot$ represents element-wise multiplication and $\boldsymbol{\epsilon}$ is sampled from a normal distribution ($\mathcal{N}$): $\boldsymbol{\epsilon} \sim \mathcal{N}(0,\, 1)$.

Ideally, the network would learn to minimise the learnable parameters to zero to effectively ignore the noise. They found their approach to deliver substantially increased performance over ϵ-greedy exploration. However, in this work we have access to a large dataset of human demonstrations that partly negates the need for full exploration.

One would also question the point at which we introduce the noise. Introducing the noise while training on demonstration data seems rather senseless since the agent does not actively explore the environment for new experiences. On the other hand, waiting for the agent to start training on experience gathered from the environment may affect the NN weights learned from the demonstrations. Therefore, we use the ϵ-greedy method since NN weights are not suddenly modified.

4.1.6 Rainbow DQN

All the DQN improvements previously discussed in this chapter delivered SOTA performance at the time. Additionally, they all address a different problem of the DQN algorithm. Hessel *et al.* [7] proposed combining all of these improvements with n-step DQN.

They performed an extensive ablation study on all DQN improvements in the Atari benchmark. Interestingly, they found all but one of the improvements to affect performance positively. Their results indicate that DDQN provides little to no benefit to the performance of the algorithm. They hypothesised that clipping the C51 value distribution to [-10, +10] leads to an underestimation of Q-values rather than the expected overestimation countered by DDQN.

[1]Note that the learnable parameters, $\boldsymbol{\sigma}^{\boldsymbol{w}}$ and $\boldsymbol{\sigma}^{\boldsymbol{b}}$, and noise variables, $\boldsymbol{\epsilon}^{\boldsymbol{w}}$ and $\boldsymbol{\epsilon}^{\boldsymbol{b}}$ are shaped according to the corresponding vector denoted in the superscript.

Ultimately, Hessel *et al.* [7] found the various improvements to be complementary to the original DQN algorithm. The resulting algorithm, Rainbow DQN, again delivered SOTA at the time, improving on all other combinations of the improvements.

4.1.7 Branching DQN

One problem we face with the DQN algorithm is that the agent is limited to selecting a single action per step. While this is usually not problematic from a pure RL perspective, we run into the issue when attempting to learn from demonstrations in a generalised manner. This is because the human demonstrators select actions simultaneously, e.g. moving the camera while sprinting and jumping. A rather simple approach to solving this issue involves combining possible simultaneous actions to an extra discrete action. However, this solution requires the combined action space to be redesigned for a new action space.

A more viable, generalised solution proposed by Tavakoli *et al.* [6] suggests computing Q-values for each set of actions in the action space. They build off the Duelling DQN network architecture by splitting the advantage branch into several action branches, one for each simultaneous action. They then perform the DQN learning algorithm for each branch with a global TD target defined by

$$y = r + \gamma \frac{1}{N_d} \sum_d Q_d \left(s', \operatorname*{argmax}_a Q_d \left(s', a_d, \theta \right), \theta^- \right), \tag{4.13}$$

where d represents an action branch that selects an action a_d from a subset of actions $\mathcal{A}_d$ and N_d is defined as the number of branches.

During backpropagation, the gradients of all the branches are accumulated at the layer before the branching. This means that the shared network would be updated with much larger gradients than the action branches. Tavakoli *et al.* [6] address this issue by scaling the gradients of each branch with $\frac{1}{N_d+1}$ before they are accumulated.

While BDQ did not deliver extraordinary performance, it proved capable of competing with algorithms better suited to selecting simultaneous actions. For the purposes of this work, we calculated the target for each branch individually rather than computing a global target. We believe using a global TD target when combining BDQ with C51 would be problematic since C51 estimates the distributions directly. Figure 4.5 provides a visual representation of how the branching architecture modifies the NN head. Note that the BDQ algorithm uses the Duelling network architecture with action advantages (A^π) being estimated for each branch.

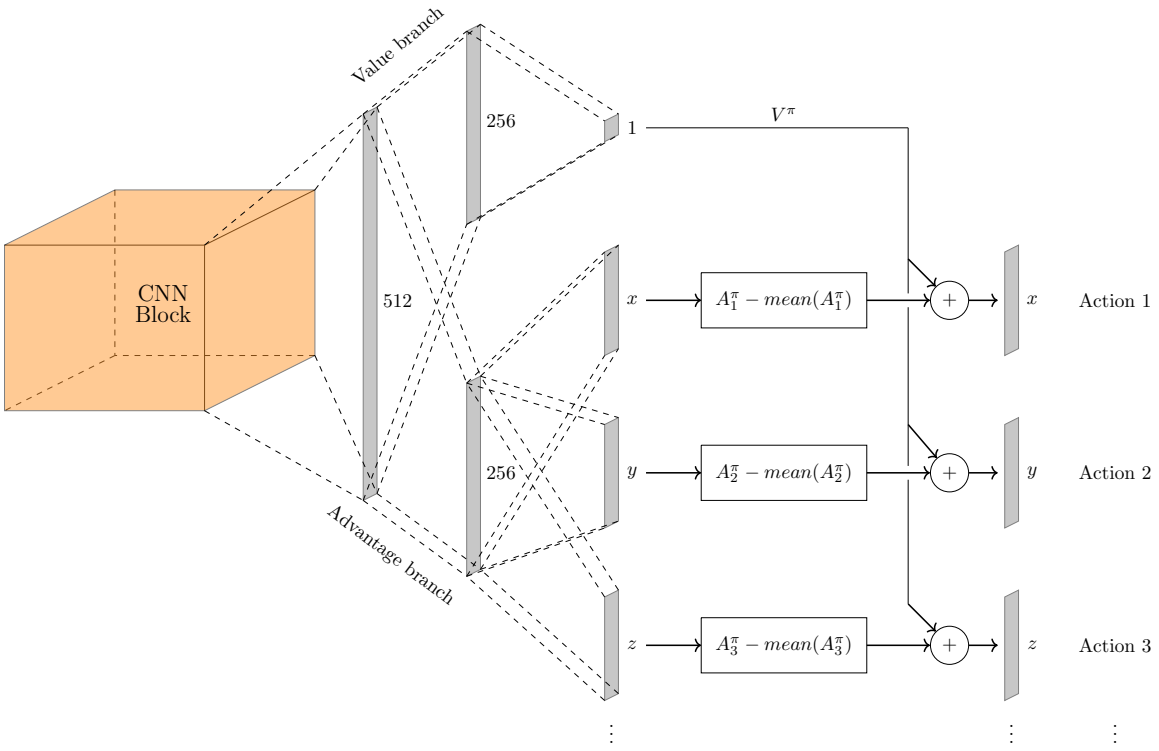

Figure 4.5: Branching DQN network head based on the work of Tavakoli *et al.* [6]. The network simply combines several DQNs and shares the CNN block and value branch.

4.2 Imitation Learning

The RL methods discussed in the previous section can be very effective in solving simulated problems. However, they often require millions of environment steps before learning a remotely useful policy[2]. This raises two concerns with pure RL methods: computational power and real-world viability. We can address the concerns by using methods that allow the algorithm to observe and learn from expert demonstrators, i.e. to mimic human behaviour. These are categorised as imitation learning (IL) methods. In this section we discuss two IL methods used in this work.

4.2.1 Behavioural Cloning

Behavioural cloning (BC) treats the problem as a supervised learning classification problem. When provided with recordings of actions performed by experts in a variety of situations, the network learns to map the actions to the states—effectively classifying which action an expert would select in a certain state [31]. To learn the expert action for a given state, a BC algorithm typically minimises the cross-entropy loss. The cross-entropy loss is defined

[2]Agent57, the current SOTA algorithm for Atari, was trained for anywhere up to 100×10^9 steps.

by

$$L_{CE}(x, y) \;=\; -\sum_{k=1}^{N} y_k \log\left(x_k\right), \tag{4.14}$$

where x is the predicted score and y is a one-hot vector with the expert action selected. Minimising the cross-entropy loss maximises the distance between the scores of the correct- and incorrect actions.

This approach does not consider the reward received from the environment, only the actions performed in the states observed. It also forgoes any interaction with the environment and relies on the demonstrators to act according to an optimal policy. Therefore, the upper performance limit of BC algorithms will always be limited to that of the demonstration data.

4.2.2 Deep Q-learning from Demonstrations

To address the issues of BC described above, Hester *et al.* [18] proposed the DQfD algorithm. Their main contribution,

$$J = J_{DQ} + \lambda_1 J_n + \lambda_2 J_E + \lambda_3 J_{L2}, \tag{4.15}$$

combines 1-step and n-step TD losses with a margin loss based on the expert action and an $L2$ regularisation loss. The $L2$ loss is included to prevent overfitting, especially when training on demonstration data. The margin loss (J_E) forces the expert action in a demonstration state to be least a margin larger than the other actions. The margin loss is defined by

$$J_E(s, a, a_E) = \max_a \left[\, Q(s, a) + l(a_E, a) \,\right] - Q(s, a_E), \tag{4.16}$$

where $l(a_E, a)$ is a simple margin function that is 0 for the expert selected action and a positive value otherwise. Note that the margin loss is not applied to experiences collected by the agent. They split the learning into a pre-training phase followed by learning from environment interactions mixed with demonstration data. The demonstration data is kept in a separate buffer and sampled at a slightly higher rate than agent data.

The results obtained by Hester *et al.* [18] indicate that their approach is a more viable IL solution than BC. They showed that DQfD successfully learned a good starting policy from the demonstration data. It also does not necessarily rely solely on the data being expert demonstrations.

4.3 MineRL 2019 Top Algorithms

Now that we have established the foundations, we can discuss the algorithms of the top 2019 MineRL challenge submissions. Their algorithms were trained in the ObtainDiamond environment for 8×10^6 steps (or 4 days real-time) before being evaluated for 100 episodes. The two teams achieved mean rewards of 42.41 and 61.61 respectively. In the rest of this section we review their algorithms.

4.3.1 Scaling Imitation Learning in Minecraft

In the 2019 MineRL competition, Amiranashvili *et al.* [9] proposed a solution using IL alone. Their approach was inspired by the successes of previous work that relied on traditional supervised learning [32; 33; 34; 35]. They evaluated IL against the DQfD proposed by Hester *et al.* [18].

The authors found the IL approach to consistently outperform DQfD in all their evaluations. They did however find that the use of a deterministic argmax[3] output policy seemed to deteriorate the performance of IL. In the following sections we investigate the methods which the authors describe as beneficial to their success.

4.3.1.1 Inventory Multi-hot

An observation in the MineRL environment consists of a raw image as well as the contents of the agent's inventory. These items in order are: coal, cobblestone, crafting table, dirt, furnace, iron axe, iron ingot, iron ore, iron pick-axe, log, planks, stick, stone, stone axe, stone pick-axe, torch, wooden axe and wooden pick-axe. The items are represented in an 18 dimensional vector of integers representing the number of an item in the inventory.

The authors applied n-hot encoders on a selection of the items. For the handheld item, they applied a one-hot encoding. For items that can potentially reach quantities too large to feasibly encode with multi-hot, they simply normalised the value to a small float based on the mean of quantities observed in the demonstration data. All other items were simply multi-hot encoded. For example, two wooden pick-axes would be encoded as $[1\,1\,0\,0\,0\,0]$, where the size of the vector is determined by a maximal value chosen by the authors. Concatenating the encoded data results in a vector with 216 components.

[3]The argmax function returns the index of the largest component for a given vector.

The inventory shaping described in this section could be considered hand-crafting features for the network. However, we acknowledge that Amiranashvili *et al.* [9] may have selected the above-mentioned techniques to simplify the resulting vector rather than specifically hand-crafting. For example choosing 128 as an arbitrary maximum value for encoding results in an extremely sparse vector with 2312 components. Compared to a dense vector of length 216, the sparse vector may degrade network performance and efficiency.

4.3.1.2 Action Space Shaping

The action space of MineRL consists of movement, camera and crafting actions. Some of these actions can be executed simultaneously. This would normally not be considered problematic, since we could simply continue selecting a single action per step. However, given the prevalence of IL techniques present in the top 10 of the competition [17], handling simultaneous actions of human recordings proved crucial to the success.

Amiranashvili *et al.* [9] combined several actions to form a new vector of actions to choose from. Considering every possible combination of actions (disregarding the utility) leads to a total of 1280 possibilities. However, many of these combinations are redundant, for example combining left and right or forward and back in single actions will result in no movement. Removing redundant actions and limiting simultaneous actions to three, resulted in a final vector of 130 actions. For the Treechop environment it can be reduced even further to 112 action combinations.

Their action shaping approach seemed to succeed in its goal of simplifying the original action space. Although the reduction in possible actions may have been beneficial to performance, it is not a viable generalised solution. If the environment or action space changes, the action combination space would need to be redesigned.

4.3.1.3 CNN Architecture

In their paper, Mnih *et al.* [11] proposed a CNN architecture to overcome spatial variations in images. Their design involved three simple sequential CNN layers. Due to the popularity and simplicity of their design, their CNN architecture became known as the 'nature CNN' in the RL field.

Aside from the nature CNN architecture, Amiranashvili *et al.* [9] evaluated three additional designs. The first design was inspired by the work done by Espeholt *et al.* [8] on the Importance Weighted Actor-Learner Architecture (IMPALA). The second design was from the winning submission of Nichol [19] in the 2019 Unity Obstacle Tower Challenge. Nichol

Table 4.2: Results of CNN architecture comparison done by Amiranashvili *et al.* [9] on the MineRL ObtainIronPickaxe environment.

CNN Architecture	Reward	Network Parameters
Nature [3]	27.9 ± 2.3	94.37 k
IMPALA	39.2 ± 5.2	391.33 k
Fixup IMPALA	49.7 ± 2.0	540.38 k
Double Deep Fixup IMPALA	53.4 ± 7.5	2.15 M

[19] modified the IMPALA design by implementing Fixup initialisation [36]. The last design was simply a modification of the second where all CNN channels were doubled.

In their evaluation on the ObtainIronPickaxe environment, Amiranashvili *et al.* [9] found the performance of the nature CNN to be inferior. The last design mentioned outperformed the rest in the environment. However, given that it only scored 7% higher than the original version at the cost of four times the parameters and nearly three times the variation in performance, it may not be very efficient in terms of computational cost. The results of each of the CNN architectures are shown in Table 4.2.

4.3.1.4 Data Augmentation

In addition to alternative CNN architectures, Amiranashvili *et al.* [9] also investigated data augmentation techniques traditionally used in supervised learning tasks. They applied and evaluated horizontal flipping, rectangle removal & brightness, sharpness, contrast and posterisation augments. These augments are not considered hand crafting since they can be applied to any situation where visual data is used. They also only modify the dataset and not the RL algorithm.

Horizontal flipping involves reversing the columns in a 2D vector. This transformation can be applied to an image to create what is effectively a new data entry. However, in our case the augmentation needs to be paired with a modification in the recorded actions of the human data. If a human moved the camera to the left, flipping the POV observation horizontally would require the camera movement to flip to the right. Additionally, any actions moving left would need to be flipped to right and vice versa. They included horizontal flipping in all their evaluations of the other augmentations.

The rectangle removing augment involved selecting a random region in an image and replacing it with a constant value (in this case zero). They found this augmentation to degrade performance when applied. We postulate that the removal process may have removed crucial regions of observations leading to unrealistic input data. Similarly, reducing the colour bit-depth (posterisation) also degraded performance; possibly due to the same

reason.

Modifying the contrast likely seemed to be a beneficial augmentation. However, they found the performance to drop slightly. Perhaps the contrast adjustments are not a necessary addition with the given data.

The only augmentation that provided additional performance (even though slightly), was sharpness. While the mean performance may have increased by roughly 2%, the variation in returns also increased. That means the agent's behaviour is slightly more unstable.

While most of these augments may not have yielded desirable results in the MineRL environment, they may be beneficial when tasked to operate with real world scenarios. In the real world we have to deal with cameras that lose focus occasionally and a wide variety of lighting situations. These situations may benefit from an algorithm trained on a wider variety of scenarios.

4.3.1.5 Discussion

Ultimately, Amiranashvili *et al.* [9] designed an impressive IL agent capable of reaching 2^{nd} place in the 2019 MineRL competition. While their inventory shaping technique is specific to this environment scenario, we believe they could have used a constant encoding throughout the vector but chose to simplify the resulting vector and speed up the training process.

Their action shaping technique is also only applicable to this specific scenario. Placing an agent in a different environment would break the actions. Instead they could have investigated a branched approach, similar to what we propose.

The investigation of alternative CNN architectures seemed to be beneficial to performance. However, their agent is simply a supervised learning algorithm trained on human demonstrations. We investigate their best performing CNN architecture in Section 6.10.

They also investigated several data augmentation techniques traditionally used for supervised learning tasks. Unfortunately, they only found image flipping and sharpness modifications to be beneficial. Independent of their work we also implemented the flipping technique to effectively double the demonstration data. We investigate the effect of the augment in Section 6.11.

4.3.2 Forgetful Experience Replay in Hierarchical Reinforcement Learning from Demonstrations

Inspired by the work of Hester *et al.* [18] on DQfD, Skrynnik *et al.* [20] proposed the Hierarchical Deep Q-Network from Demonstrations (HDQfD) algorithm. They combined DQfD with HRL to create the winning submission for the 2019 MineRL competition. In their later work, Forgetful Experience Replay in Hierarchical Reinforcement Learning from Demonstrations [37], they updated HDQfD with a heuristically modified hierarchy of sub-tasks extracted from a sample demonstration.

ForgER proved to be the best solution applied to the MineRL competition problem to date. The final version of ForgER, referred to as ForgER++, managed to obtain a diamond in one episode during evaluation. Although their results are impressive, they deviated from the competition rules by removing limitations placed on training steps and time. In the following sections we investigate the methods which the authors describe as beneficial to the success of ForgER.

4.3.2.1 Hierarchy Extraction

In their earlier work, Skrynnik *et al.* [20] described their hierarchy extraction method as examining all expert trajectories to build a chronological timeline of the order in which items are collected and which crafting actions are performed. In their later work [37], they extracted the sequence from a single selected trajectory. This timeline is then used as a fixed sequence of sub-tasks to achieve. Each sub-task tracks specific items in the inventory. If sufficient resources are available, the relevant crafting action is then performed.

They mentioned a specific example of a sequence of sub-tasks. For each item required to collect a diamond (as demonstrated in Figure 1.3), a selected quantity must be obtained to move to the next item. While this solution does seem to outperform all other approaches, it requires monitoring the inventory vector and may not generalise to other hierarchical environments with demonstrations.

4.3.2.2 Replay Buffer Forgetting

In their DQfD algorithm, Hester *et al.* [18] combined Q-learning with a supervised margin loss based on expert demonstration data. They added the demonstration data to the replay buffer and prevented the algorithm from overwriting any expert observations. The demonstration data is then sampled at a higher probability than agent observations.

However, depending on the proportion of the replay buffer dedicated to the demonstration data, the proportion of demonstrations in a batch can vary. The demonstration data is also not always perfect.

While the demonstrations may act as a guide to the agent, certain trajectories may be inefficient or adversarial. The data may appear disjointed from agent collected observations. This is especially true for cases where continuous actions need to be discretised. ForgER proposed a way of addressing this issue by adding a decreasing variable controlling the proportions in a training batch. Slowly reducing the proportion of demonstrations used for training would allow agents to overcome the issue of imperfect trajectories by relying on its own observations.

In their evaluations on multiple environments, Skrynnik *et al.* [37] found ForgER to outperform the DQfD algorithm after transitioning to agent collected experience alone. Later work done by Paine *et al.* [38] came to a slightly different conclusion. They found the presence of demonstration data (although in small amounts) to be crucial to the performance of their algorithm. They suggested treating the proportion of demonstrations in a training batch as another hyper-parameter that requires fine-tuning.

We adopted a middle ground between the work of Skrynnik *et al.* [37] and Paine *et al.* [38]. As the agent collects experience we can decrease the proportion of demonstration data utilised for training. This prevents the agent from over-fitting on a limited number of initial experiences by slowly introducing agent experiences in training batches.

4.3.2.3 Action Space Shaping

The MineRL action space allows a player or agent to select multiple actions in a single step. The DQN algorithm was only designed to select a single action per step. For the movement actions, Skrynnik *et al.* [20] implemented a combined action space of 10 actions. Each action represents multiple selected actions to be performed by the agent. For example, having an action to perform 'jump' and 'forward' in the same step seems like a logical option.

To further simplify the action space, they fixed the 'attack' action to always be selected for all the combined actions. Additionally, keeping the 'attack' action selected may also be to ensure the agent does not get interrupted by an epsilon-selected action during the breaking of a block. Missing attack actions while breaking a block can lead to the progress being reset. They also discretised the camera action space to $[-5, 0, 5]$.

To utilise the human demonstration data, they needed to categorise demonstration actions

as single combined actions. However, this can be quite a challenging task due to humans moving the camera independently from movement actions. Their solution involved multiple steps applied in a certain order. Firstly, the camera actions are aggregated over four steps to attempt a frame-skipping effect. If the camera delta values are larger than a certain threshold value, the relevant combined action is selected in order of the largest magnitude first. Secondly, if the camera deltas do not exceed the threshold, the movement actions are selected based on their frequency in four sequential demonstration steps. The combined action which most resembles the demonstration action is then selected. Finally, all crafting actions are handled by an item agent which performs actions based on the contents of the agent's inventory.

Based on their evaluation results, their action space shaping seems to be a successful implementation. However, combining certain actions and fixing actions to always be active requires some human domain knowledge. We opted to use an alternative solution proposed by Tavakoli *et al.* [6] as discussed in Section 4.1.7.

4.3.2.4 Data Augmentation

Although we found an implementation of the flipping augment discussed in Section 4.3.1.4 in the code that was released by [37], we found no instance of usage. Based on the findings of Amiranashvili *et al.* [9], we believe the implementation of this augment could have provided additional performance at a small cost.

In their hierarchical solution, Skrynnik *et al.* [20] discussed utilising data from other sub-tasks when training a specific sub-task. They nullified rewards from additional data and did not consider them to be expert in order to prevent the agent from learning behaviour designated to solving another sub-task. The agent may learn certain behaviours that are beneficial in all sub-tasks, such as surviving in a body of water.

The utilisation of all data in all sub-tasks seems like a beneficial addition to hierarchical RL agents where certain behaviour is beneficial across all sub-tasks. We adopt this approach in our solution by keeping all the data in a single replay buffer and prioritising the sampling of data from the current sub-task.

4.3.2.5 Discussion

HDQfD and ForgER both deliver SOTA performance on the MineRL problem. To date, ForgER is the only algorithm capable of fully solving the MineRL environment by obtaining a diamond. However, it is worth noting that the MineRL environment is a novel

problem posed in 2019 with limitations on computational resources and algorithm designs. Nevertheless, we investigated the algorithms proposed by Skrynnik *et al.* [37] and found some of their work to be beneficial.

By examining expert trajectories, Skrynnik *et al.* [20] extracted a timeline of crafting actions based on the changes observed in the player's inventory. In their ForgER algorithm, they expanded on this idea by selecting a specific trajectory to follow. While the solution proved successful in solving MineRL, we opted to take a more generalised approach to extracting the hierarchy.

In their ForgER algorithm they introduced a dynamically decreasing variable to control the proportion of expert observations used for training. Their experimental evaluation revealed the method to be generalisable to a variety of other environments and yielding increased performance over DQfD and Policy Optimisation with Demonstrations (POfD) [39]. We adopted this approach to control the introduction of agent experiences to the training batches and reduce reliance on expert observations.

In their HDQfD algorithm, Skrynnik *et al.* [20] implemented multiple methods to reduce the action space to a single vector representing combined actions. While their action space reduction method was clearly successful, we are in search of a more generalised approach that does not rely on human domain knowledge to design a combined action space.

Finally, they augmented the data available to a given policy with data from other policies. We found this augmentation to be an interesting way to efficiently use all the data available to the agent.

4.4 Summary

In this chapter we investigated the original DQN algorithm along with several improvements proposed by other researchers. We also briefly discussed IL methods, namely BC and DQfD. Lastly, we analysed the work of the top two participants of the 2019 MineRL challenge. Both of these algorithms delivered impressive results. However, in search of a more generalised algorithm, we were unable to implement several of their methods; particularly methods related to action space shaping and sub-task discovery and navigation. Their methods also limit the agent to a single action per step, thereby necessitating action space shaping.

5 Implementation Details

In this chapter we discuss our solutions to the MineRL Treechop- and ObtainDiamond environments. The MineRL environments were released in a Python package with a wrapper for the OpenAI Gym [40] interface to simplify interaction with a Minecraft client [14]. We used PyTorch, a deep learning library available in Python, to train our networks [41]. The code for our solution to the MineRL Treechop environment is available at `github.com/Frana-0/minerl-treechop`.

5.1 Branched Rainbow from Demonstrations

We implemented all the improvements to the DQN algorithm in some way in our final agent—all but noisy networks for exploration. Specifically, we implemented a Branched Rainbow algorithm capable of learning from demonstrations (BRfD). We were unable to implement a working version of the DQfD margin loss for the distributional Q-values produced by C51. We attempted to compute the margin loss for the distributional case by using the KL divergence shown in Equation 4.11. Unfortunately, we were unsuccessful in this attempt. We believe the reason behind this failure to be related to PyTorch not supporting backpropagation on the operations to interpolate the target distribution back to the original support. Usually, in TD learning we do not backpropagate the gradients through the target. However, we found DQfD unable to learn if we remove these gradients from the margin loss.

Fortunately, we found success by replacing the margin loss with a simple BC loss. While the BC loss does not account for the reward received, the TD losses ensure that the Q-value estimates are reasonable. The BC loss on the other hand ensures that value estimations of expert actions are higher than that of non-expert actions. We calculate the BC loss with the Q-values estimated by the NN. However, since the NN outputs a value distribution, we first calculate the expectation. This is done by multiplying the atoms from the support vector with its probabilities. These probabilities are calculated by applying the softmax

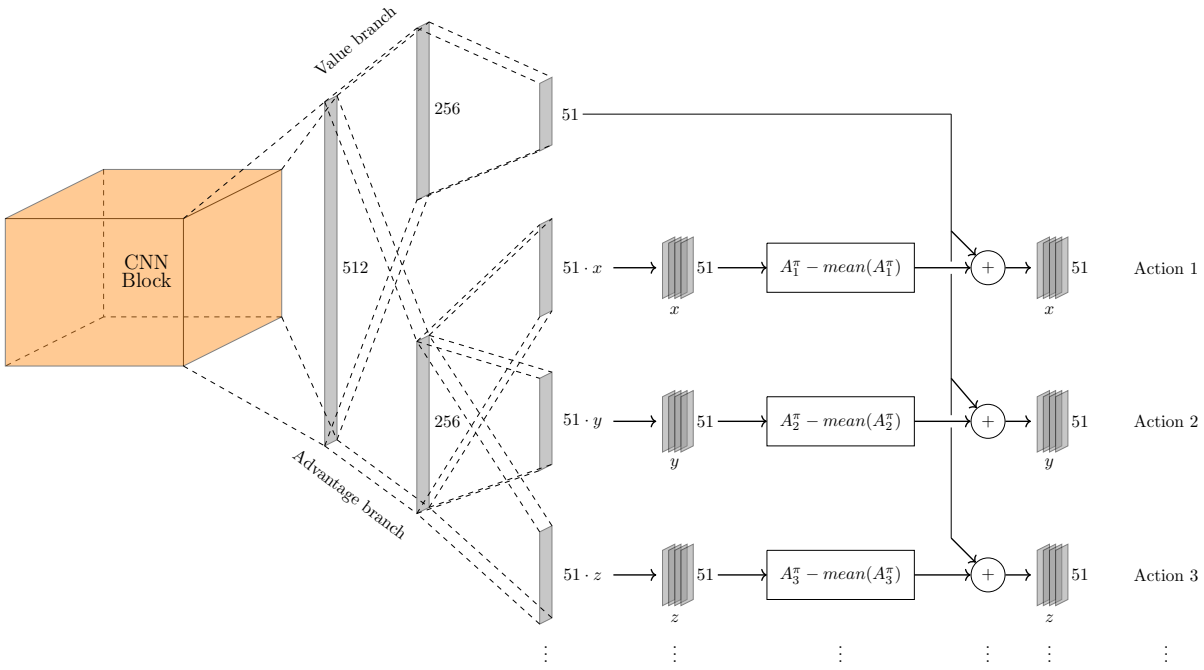

Figure 5.1: Branched Rainbow DQN network head based on the work of Hessel *et al.* [7] and Tavakoli *et al.* [6]. The network combines the action branching network from Figure 4.5, with the distributional network from Figure 4.4. The resulting network can be much larger than depicted. In the MineRL Treechop environment, the network has 9 branches.

function (defined in Equation 3.12) to the output of the NN. The output is then defined by

$$\begin{aligned} p_i(s,a) &= \text{softmax}\left(\theta_i(s,a)\right) \\ Q(s,a) &= \sum_{i=0}^{N} z_i p_i(s,a). \end{aligned} \tag{5.1}$$

If we combine all the modifications to the NN head, we obtain Figure 5.1. We also use the nature CNN architecture proposed by Mnih *et al.* [3]. The nature CNN block is depicted in Figure 4.1 and its specifications are defined in Table 4.1.

We also implemented experience replay from demonstrations with forgetting as proposed by Skrynnik *et al.* [37]. However, instead of completely removing the demonstrations, we reduced the proportion of sampled demonstrations to a value selected as a hyperparameter.

Lastly, we implemented the flipping augment to the demonstration data. For every transition we stored the flipped action and a reference to the frames object. The flipped action maps move left to move right, and vice versa. It also flips the discrete camera yaw action. When a flipped transition is sampled, the frames are flipped and used for learning. Using this method increases training time marginally, but uses just slightly more memory.

5.1.1 Hyperparameters

For all experiments in the Treechop environment we used the hyperparameters shown in Table 5.1. Nearly all the values selected were inspired by the relevant literature. Alternatively, in cases like the memory capacity, we were limited by hardware limitations.

We did however select the following hyperparameters: C51 $V_{\max}$, pre-train batch size and minimum forget proportion. We selected $V_{\max}$ as 50 to balance the distribution resolution and estimation accuracy. The pre-train batch size was used to optimise the pre-training process. For the Treechop environment, we used a minimum forget proportion of 0.5 since the demonstration data was of high quality.

Table 5.1: Hyperparameters for our algorithms in all ablation study experiments.

Hyperparameter	**Value**
Deep learning	
Learning rate (α)	62.5×10^{-6}
Batch size	32
DQN	
Discount factor (γ)	0.99
Replay frequency	4
Learn start	5×10^3
Frame skip	4
Frame stack	4
Target update	2×10^3
n-step	10
e-greedy	0.025
PER	
Memory capacity	3×10^6
Prioritisation exponent	0.6
Importance-sampling weights	$0.4 \rightarrow 1$
C51	
Atoms	51
Max value	50
Min value	0
DQfD	
1-step loss scale (λ_0)	1.0

n-step loss scale (λ_1)	1.0
Supervised loss scale (λ_2)	1.0
Regularisation loss scale (λ_3)	1.5×10^{-5}
Agent experience bonus priority	1×10^{-3}
Demonstration data bonus priority	1.0
Pre-train batches to train for	80×10^3
Batch size during pre-training	128
Memory dedicated to demonstrations	30%
ForgER	
Minimum forget	0.5
Final forget decrease step	125×10^3

5.2 Hierarchical Implementation

With the BRfD algorithm to act as a low-level policy, we can now discuss our hierarchical implementation. As described in Section 2.8.1, an option is defined by three components: an initiation function (I_ω), a policy (π_ω) and a termination function (β_ω). However, the termination function of one option corresponds to the initiation function of another. We can therefore define a high-level policy that chooses from $\mathcal{G}$ to solve a certain sub-task. But we first need to discuss how we selected the options.

5.2.1 Option Discovery

As mentioned in Section 2.8.1, the authors of the options framework did not address the issue of discovering the options. Fortunately, the organisers of MineRL released two versions of the ObtainDiamond environment and dataset. These are the dense and non-dense environments. In the dense environments, the agent receives a reward for all relevant items obtained. In the non-dense environments, a reward is received only for the first time a relevant item is obtained.

With the hierarchical structure of MineRL Obtain rewards (see Table 1.4), we can extract a $\mathcal{G}$ with trajectories from the non-dense demonstration dataset. For example, with the following sequence of rewards over time:

$$\text{Trajectory rewards} = [\underbrace{0, 0, 0, 1}_{\text{Option 0}}, \underbrace{0, 0, 2}_{\text{Option 1}}, \underbrace{0, 0, 0, 0, 4}_{\text{Option 2}}, \underbrace{0, 0, 4}_{\text{Option 2}}, \underbrace{0, 0, 8}_{\text{Option 3}}, \underbrace{0, 0, 0, 0, 16}_{\text{Option 4}}, \dots]$$

We can take the $\log_2$ of the rewards to easily classify the item obtained as the goal of an option. The steps leading up to the reward can then be used to learn how to achieve the goal of the option.

This method is simple to implement and should apply to most problems with a hierarchical reward structure and access to demonstration data. However, it reduces the flexibility of our agent in terms of its capacity to select an option when needed, as opposed to the sequence in which options were observed in the demonstration data. For example, if the agent runs out of logs, it should be able to switch to option 0 to look for a tree. This method does not necessarily allow for the case described in the example. Unfortunately, we were not able to devise another strategy of discovering the options in a generalised manner.

5.2.2 High-level Policy

Since we extract the set of option trajectories from the demonstration data rewards, we can use the inventory vector to select the next option. Although it would be possible (and more generalised) to use the POV observation as well, we believe the inventory vector contains all the necessary information pertaining to option selection. Given that we are looking for a generalised solution, accessing the inventory directly to select the next option based on specific items is not allowed. Therefore we need to find another solution to select policies.

Another possible solution is to simply treat the high-level policy as a classifier. For each option, it would output a likelihood of that option being the next option given the input. We can train the high-level policy NN with the option trajectories extracted from the demonstration data. Whenever the agent's inventory contents change, the high-level policy would predict the next option. This has the advantage of potentially detecting items that are not required to move to another option.

Unfortunately the classifier method has several flaws. Firstly, it forces a decision at each step where a change in the inventory is detected. Given that it would be trained with supervised learning, and that we expect to observe a much wider variety of possible vectors for the inventory, forcing the network to select an option may lead to behaviour not matching the task at hand. Secondly, if the agent selects the wrong option, it may remain stuck with that option for the rest of the episode. For example, if an agent selects the option to collect logs in a cave, its inventory may not change again. Lastly, it does not account for rewards received. As observed in our ablation study, IL methods tend to perform better when considering the reward received. However, we opted to not use the

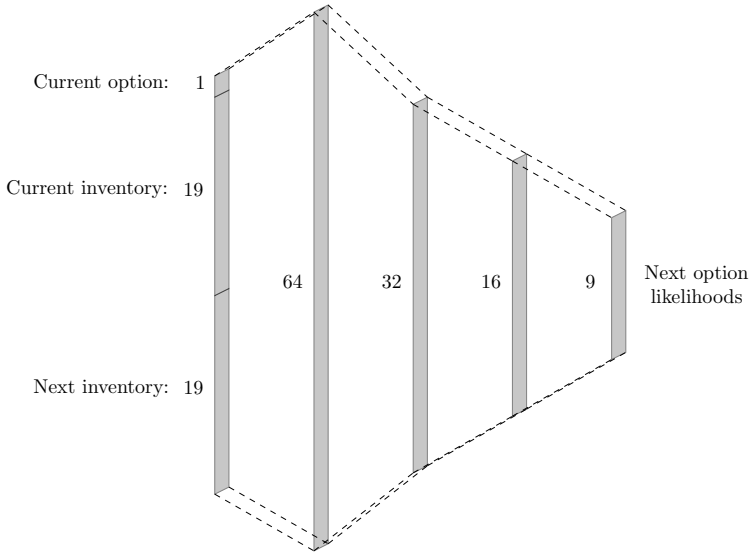

Figure 5.2: High-level policy that selects the next option from a concatenated vector. The vector consists of the current option, current inventory and next inventory.

rewards since we train in the Dense environment where rewards vary wildly in magnitude. While these flaws can certainly be detrimental, we were not able to address them in this work.

For the input of this classifier we concatenate the current policy and the current- and next inventories into a single vector. The idea is that the NN learns to associate changes in the inventory with the current option in order to predict the option that typically follows in the demonstration data. This network is shown in Figure 5.2.

5.2.3 Low-level Policies

We use the reward trajectories extracted from the demonstration data to train the low-level policies (options). These policies are then only allowed to train when they are in control. In other words, the policies only learn when new experiences relevant to their sub-tasks become available. To adapt our BRfD algorithm to the ObtainDiamond problem, we implemented several methods and modifications.

Inventory input. Unlike the Treechop environment, the observation space of the ObtainDiamond environment includes a vector representing inventory contents. If we do not consider this vector, the MDP of our problem is no longer Markovian. Therefore, we process the inventory with a single FFNN layer before concatenating the result with the flattened output of the nature CNN block. This is shown in Figure 5.3. Note that we concatenate the current policy and equipped item type to form the 20 component vector.

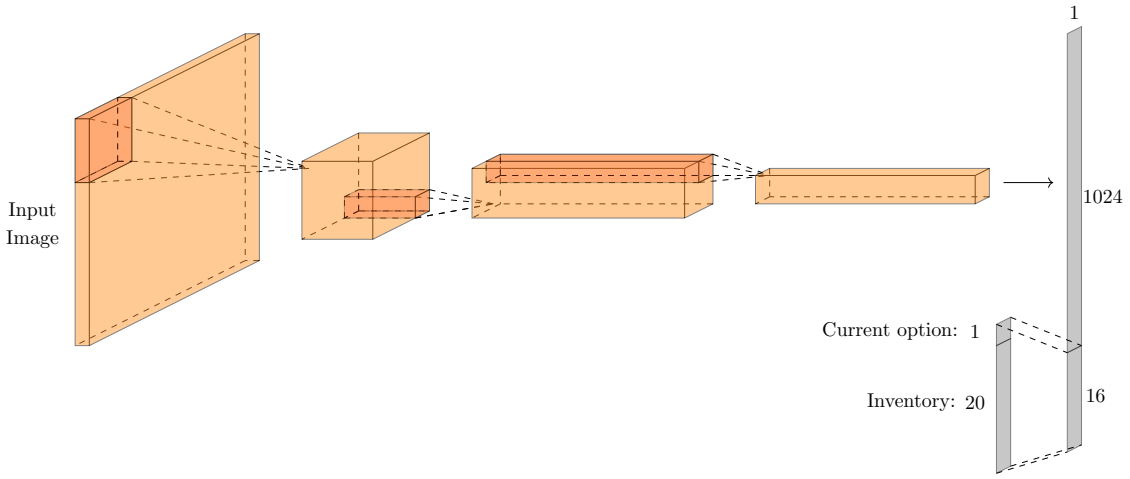

Figure 5.3: For the Obtain environments we concatenate the contents of the inventory to the flattened output of the CNN network. This model is then followed by the full head shown in Figure 5.1.

Reward scaling. In their paper proposing the original DQN algorithm, Mnih *et al.* [3] clipped the rewards of the Atari games between $+1$ and -1. They stated that a varied scale of rewards would lead to large estimation errors. Unlike the Treechop environment where the agent receives a reward of 1 for each log obtained, the rewards returned by the Obtain environments increase exponentially based on the item obtained. Therefore, we applied the logarithm function to the reward received: $r = \log_2(1 + r)$[1]. This reduced the possible maximum reward in a single step from 1024 to 10.

Additional environment data. Although the MineRL dataset includes a large amount of transitions, not all of these are in the ObtainDiamond environment. Additionally, not all demonstrations were successful in solving the task. For that reason, we attempted to supplement the data from ObtainDiamond with both the Treechop- and the ObtainIronPickaxe data. Unfortunately, limited by hardware memory, we were only able to add the Treechop data to reach a total of nearly 1 million transitions (2 million with the flipping augment).

Augmenting option data. As proposed by Skrynnik *et al.* [37], we augmented the data available to each option with data from the other options. However, instead of separating the replay buffers of each option, we separated the priorities of each option on a global replay buffer. All options therefore have access to the full replay buffer. To ensure that task-specific transitions are sampled at a higher rate for a certain option, we scaled the priorities for transitions from other options with 0.025. To counteract this when calculating the importance-sampling weights, we re-scaled the priorities to their original values.

[1]Base 2 works well with the reward structure defined in Table 1.4.

Transfer knowledge. When pre-training the options, the order in which we train them does not matter. Nevertheless, we trained the low-level policies in the order of smallest sub-task reward to the largest. This method allowed us to exploit the fact that many low-level policies face similar situations to the immediately preceding option. We therefore transferred the weights from each low-level policy to the next upon the completion of pre-training for the policy.

Frame-skipping single craft. While the frame-skipping technique allows us to generate more episodes of experiences, it introduces a flaw in the Obtain environments. If we perform an action, the technique repeats the action k times when skipping. This may lead to repeated craft actions consuming valuable, limited resources. To address this issue, we simply limited craft actions to a single execution per k steps.

5.2.4 Hyperparameters

For the experiments performed in the Obtain environments, we also used the hyperparameters defined in Table 5.1. However, in Table 5.2 we show the additional hyperparameters as well as any updates to the previous values.

We used ϵ-greedy for some exploration at the start of RL training. The relevant values are shown in Table 5.2. Additionally, we reduce the minimum forget proportion to 0.25 since the demonstration data contained various flaws. We also used a value to scale the priorities of transitions from alternative options. This value was selected to be very small to ensure that options are still mostly trained on relevant data. Lastly, we trained the high-level policy for 150×10^3 steps, or until its accuracy on the demonstration data crossed the 99.8% threshold.

Table 5.2: Hyperparameters for HRL agents in the MineRL Obtain environments. This table continues or updates the hyperparameters listed in Table 5.1.

Hyperparameter	**Value**
DQN	
Target update	10×10^3
ϵ start value	0.2
ϵ final value	0.05
ϵ decay per episode	0.99
DQfD	
Memory dedicated to demonstrations	50%
Target update during pre-training	2×10^3
ForgER	
Minimum forget	0.25
Final forget decrease step	1×10^6
Other policy data scale value	0.025
HRL	
Training steps for high-level policy	150×10^3

5.3 Implementation Issues

During the implementation of our solution we were faced with many issues preventing us from directly applying an existing algorithm. In this section we describe these issues and our approaches towards solving them.

5.3.1 Human Demonstrations

While the MineRL team released a large dataset of human demonstration data, the data is not without flaws. We addressed many of these flaws by simply filtering out episodes where the humans were unsuccessful[2]. Additionally, we also skip transitions where the state does not change, the player has selected no action and no reward is recorded.

Experiences obtained by RL agents when interacting with the environment does not at first appear similar to the provided demonstration data. To overcome this we employ several techniques which we will now discuss.

[2]These flaws include anomalies, server errors and adversarial behaviour by humans [42].

5.3.1.1 Frame skipping

The first DQN agent to perform at human level in Atari games, utilised frame-skipping [11]. The authors argued that the technique allows for more games to be played since it is rather computationally expensive to select an action compared to generating a new transition. In our case it also helps to address another issue regarding the continuous camera actions from the demonstration data. Unlike the agent providing a camera movement action in degrees, human camera movement involves acceleration and deceleration. Using frame skipping allows us to use the last k frames' camera actions to approximate the degree of observed camera movement. One drawback in applying this technique to the demonstration data is the reduction from N to $\frac{N}{k}$ observations. We circumvent this issue by splitting each trajectory into k separate, frame-skipping trajectories. For example a trajectory with steps $\{0, 1, 2, 3, ..., N\}$, becomes k separate trajectories with frame-skipping:

$$\begin{aligned}
\text{Trajectory } 1 &= \{0,\ k,\ 2k,\ ...,\ N-(k-1)\} \\
\text{Trajectory } 2 &= \{1,\ k+1,\ 2k+1,\ ...,\ N-(k-2)\} \\
\text{Trajectory } 3 &= \{2,\ k+2,\ 2k+2,\ ...,\ N-(k-3)\} \\
&\vdots \\
\text{Trajectory } k &= \{k-1,\ 2k-1,\ 3k-1,\ ...,\ N\}
\end{aligned}$$

Figure 5.4 illustrates this technique on a sample trajectory with 16 frames.

Additionally, we need to modify expert actions to represent the new step size k, i.e.: which combination of actions (when repeated k times) best represent how the expert reached state S_{t+k} from state S_t. We investigate two techniques to approximate expert actions. The first technique tracks the frequency of each action taken between steps t and $t+k$. These frequencies are then used to select the top m actions, where m is an integer in the range $0 < m < n_$actions. The second technique is a more greedy approach where all actions taken between step t and step $t+k$ are used.

The first technique could unfortunately lead to crafting actions being ignored due to the presence of more frequent actions. We avoid this by finding and storing any crafting actions before counting other actions. While the second technique would not suffer from the same flaw, it could perhaps select actions too greedily and repeat an action (which appeared only once) k times.

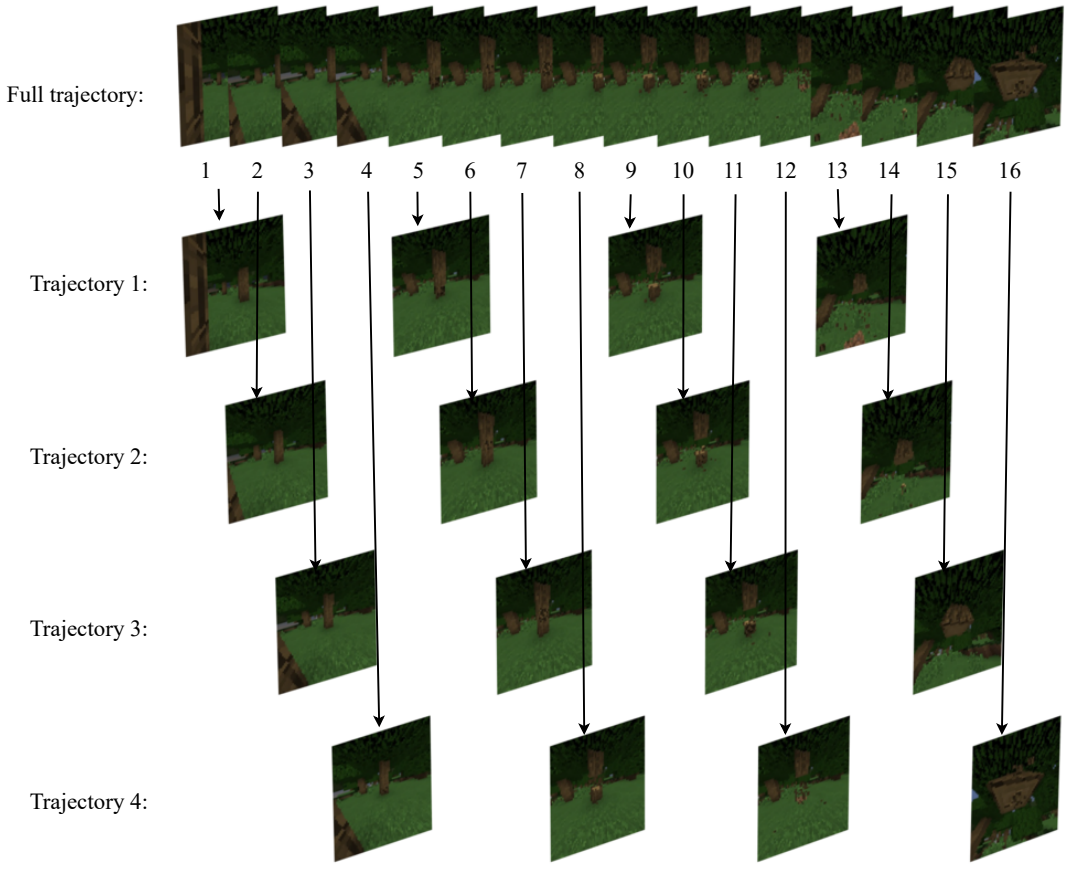

Figure 5.4: Illustration of frame-skipping applied to the demonstration data to show how we constructed k separate trajectories.

5.3.1.2 Crafting roll-out

The MineRL team also released the MineRL-v0 dataset [14]. Unfortunately, the dataset is not flawless. The team mentions possible feature anomalies, server errors or adversarial behaviour in the documentation [42].

While many adversarial demonstrations and server errors can be avoided by evaluating the total received reward in the non-dense case, feature anomalies can be present either way. We identified three anomalies that could impact an agent's performance negatively. These are:

- Players can craft multiple items in a single step. This is not possible for an agent.
- Placing an iron ore into a furnace and retrieving it before it is finished smelting is rewarded.
- Players tend to wait for all iron ore to finish smelting, resulting in more iron ingots

for single crafting actions.

The second anomaly can be attributed to a flaw in the environment rewarding system where an agent is rewarded if a relevant item appears in its inventory. This can be exploited by human players by removing (dropping, placing in the environment or in an interface) and retrieving a relevant item.

Fortunately, dropping and placing in an interface is not part of an agent's action space. We decided to not modify the reward received for breaking and picking up a placed item since it may help the agent to keep a single crafting table or furnace.

The first- and last anomalies are addressed by employing a roll-out technique. While loading a trajectory, we track the number of crafting ingredients and the crafted items for a specific item. If a craft would not be possible for an agent, we roll the N crafts out over $N \times k$ steps. This technique is demonstrated in Table 5.3 and also fits well with the frame-skipping method since we can space actions accordingly.

Table 5.3: Demonstration of a craft roll-out. An agent can craft only once per step. In comparison, a player can craft batches in a single step. To address this, we roll the player action out. Note that frame-skip $k = 4$ in this case.

Before Rollout												
Step (t)	1	2	3	4	5	6	7	8	9	10	11	12
Plank craft action		3										
Inventory logs	3	3	0	0	0	0	0	0	0	0	0	0
Inventory planks	0	0	12	12	12	12	12	12	12	12	12	12
Dense reward		24										
After Rollout												
Step (t)	1	2	3	4	5	6	7	8	9	10	11	12
Plank craft action		1				1				1		
Inventory logs	3	3	2	2	2	2	1	1	1	1	0	0
Inventory planks	0	0	4	4	4	4	8	8	8	8	12	12
Dense reward		8				8				8		

5.3.2 Discretisation

Aside from the camera, all other actions are discrete. The camera action is a two-component vector representing changes in the camera pitch and yaw respectively. The delta value is ranged between -180 and 180 degrees. As described in Section 5.3.1.1, we aggregate the delta in camera movement over k steps. Unfortunately, the DQN algorithm is not capable

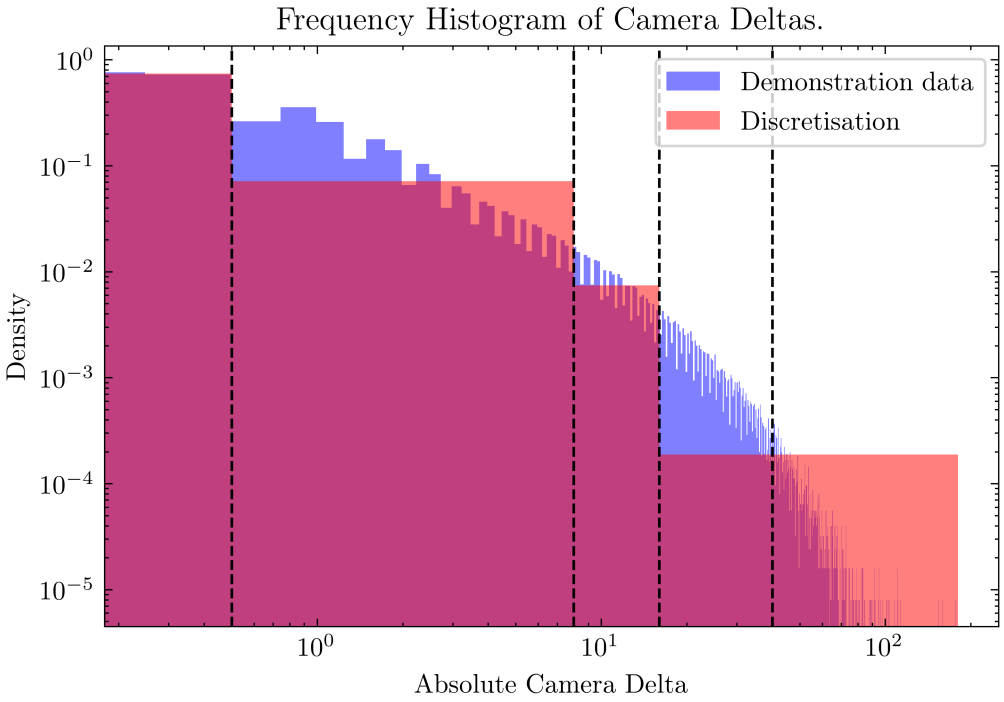

Figure 5.5: Probability density histogram of human camera delta actions. We include binned absolute camera pitch and yaw deltas. The selected discretisation densities are superimposed on the plot.

of handling continuous actions. Therefore we chose to discretise the camera action space. Selecting effective discrete values is a challenging task. Values too large will prevent our agent from making fine movements when needed. Alternatively, selecting only small values will limit the range of motion of our agent. Fortunately, we have access to 60 million frames of human demonstrations to assist in selecting effective discretisation values. In Figure 5.5 we plot the probability density of human selected camera deltas. Superimposed on these densities, we also plot the density values of the bins we selected for discretisation. Considering the need for low- and high bin values we selected the following discretisation values:

$$[-40.0,\ -16.0,\ -8.0,\ -0.5,\ 0.0,\ 0.5,\ 8.0,\ 16.0,\ 40.0]\,.$$

These values allow the agent to select between small, moderate and large camera deltas. Before discretisation, we clamp all camera deltas to $-40.0 \leq x \leq 40.0$. Positive values are rounded to the next bin, whereas negative values round to the previous bin. For example, 0.1 will round to 0.5 (bin 6) and -1.0 will round to -8.0 (bin 3). Unless otherwise specified, all experiments of our algorithm are performed using the above-mentioned discretisation values.

5.4 Summary

In this chapter we described the implementation details for our BRfD algorithm along with a hierarchical implementation thereof. For both of these algorithms, we provided the relevant hyperparameters used in our experiments.

Our BRfD algorithm combines the Rainbow DQN algorithm with BDQ and DQfD to enable simultaneous action selection and learning from demonstrations respectively. However, instead of the margin supervised loss proposed by the authors of DQfD, we implemented a BC loss. We also implemented forgetful experience replay to slowly introduce agent-collected experiences to the training batches. Lastly, we augmented the demonstration data with horizontal flipping to effectively double the data at a very low cost.

We then used the BRfD algorithm as low-level policies for a hierarchical implementation. With a large dataset of demonstration data available, we extracted the options from the non-dense rewards received by the experts. We trained a classifier to act as a high-level policy based on the rewarding transitions. The rewards also allowed us to segment a demonstration trajectory into separate trajectories for each option. We then used these trajectories to train the low-level policies. Lastly, we augmented the option demonstration data with data from other options as proposed by Skrynnik *et al.* [20].

Finally, we discussed the various issues we faced when implementing our solutions. These were largely comprised of the inconsistencies between how humans act and how the agent acts. The DQN algorithm is also limited to discrete outputs, therefore we discretised the camera action space based on the frequencies observed in the demonstration data.

6 Ablation Study

To determine the performance impact offered by each improvement in our algorithm, we performed an extensive ablation study on our BRfD algorithm in the Treechop environment. Although this study is similar to the one performed by Hessel *et al.* [7], we enhanced their Rainbow DQN algorithm with action branching and IL. Therefore, we are mostly interested by how these improvements affect the performance of the Rainbow algorithm. In analysing the results, we hope to address the first two objectives described in Section 1.4:

- Design an effective algorithm to act as the low-level policies.
- Reduce the reliance on trail-and-error training by leveraging human demonstration data.

For comparison, we also performed experiments on four additional algorithms to further motivate certain design decisions. As a preface to the work done in this chapter, we realise that these results are merely indicative of performance in the MineRL Treechop environment. Nevertheless, we believe these results are sufficient to draw conclusions about performance in unseen scenarios.

6.1 Training Procedure

We evaluate the learning curves of all the algorithms after pre-training on the demonstration data. The algorithms were trained in the Treechop environment for 300 episodes at a time. During these 300 episodes, we seeded the environment and nearly all other sources of randomness with a random number generated by NumPy[1]. In turn, the numpy random number generator (RNG) was seeded with a constant value for each experiment. We performed the experiment described above three times with three different seeds for the numpy RNG. Therefore, each algorithm was trained on a fixed set of 900 environment- and RNG seeds.

[1]NumPy is a Python package used for processing high dimensional data. It can also generate random numbers.

We would like to note that while we attempted to keep these seeds consistent across all the experiments, we were unable to control all sources of randomness. Specifically, for MineRL, the environments generate a world according to the seed. However, this seed does not control all the sources of randomness[2]. This means that all the experiments on a given seed observed the same world with slightly different POVs.

We show the results of the experiments by averaging across the three experiments and plotting the 20-episode rolling mean and standard deviations. On every plot we also include the results of our full BRfD algorithm for comparison. All our algorithms applied the methods described in Section 5.3 to address the relevant issues. Unless specified otherwise, we used the hyperparameters shown in Table 5.1.

6.2 Branched Rainbow from Demonstrations

Firstly, we investigate the experiments performed with our BRfD algorithm. As described in Section 5.1, we combined BDQ with Rainbow DQN and implemented an additional BC loss based on the DQfD algorithm. We then use the results of BRfD as a baseline for the ablation experiments performed in the rest of this chapter. For comparison, we also include the learning curve of ForgER, the 2019 MineRL challenge winning submission.

We expect BRfD to outperform ForgER considering its more flexible simultaneous action selection, distributional value estimation and data augmentation. We trained ForgER with the hyperparameters suggested by the authors [20]. It is however worth noting that while our algorithm constantly acted according to ϵ-greedy with $\epsilon = 0.025$, ForgER reduced ϵ from 0.1 to 0.01 by a factor of 0.99 after each episode.

6.2.1 Results

In Figure 6.1, we plot the learning curve of BRfD compared to ForgER. Investigating this plot, we can observe a few interesting trends. Firstly, when the ForgER algorithm started learning, it performed poorly. After reaching a minimum reward of 4, its performance started increasing drastically. At its best, ForgER achieved a reward of 60.

Secondly, our algorithm started learning with a rather good policy. In the first 20 episodes, we observe a minimum reward of 52. In the next 40 episodes, its performance increased before stabilising at around a reward of 60. At this point, the standard deviation also decreased. BRfD achieved a best reward of just below 64.

[2]These sources include, among other, clouds and particle effects.

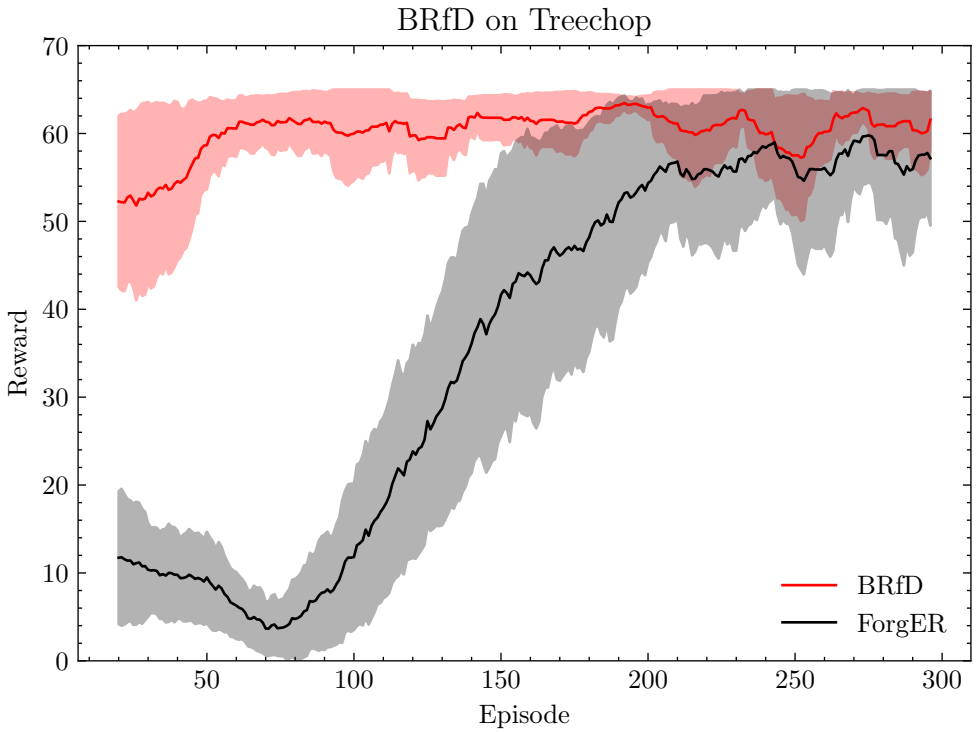

Figure 6.1: BRfD learning curve compared to the 2019 MineRL winning submission, ForgER, on the Treechop environment.

6.2.2 Discussion

It is clear that BRfD delivered better performance across the 300 episodes. The standard deviations indicate that it also performed more consistently across the three sets of seeds trained on. Notably, our algorithm also performed well directly after pre-training on the demonstration data.

A rather interesting observation is the opposing trends of the two algorithms when starting to learn from the environment. BRfD started with a high reward and increased to its stabilisation point just below the maximum possible reward. ForgER on the other hand, started poorly and trended downward to its minimum point before trending upwards. We believe this phenomenon can be explained by the differing views of the authors on the demonstration data after pre-training.

Generally speaking, we view the RL as supplementary to the IL. We use the demonstration data as a guide to keep the agent from deviating too far from the expert trajectories. In contrast, the authors of ForgER view the IL as supplementary to the RL. They treat the demonstration data as inherently imperfect—only to be used to learn a starting policy in a challenging environment. Therefore, while BRfD supplements its knowledge with environment experience, ForgER linearly reduces its reliance on the demonstration data to nothing.

Another interesting observation is the differences in standard deviation between the algorithms. BRfD performed more consistently after gaining some experience on the environment. We believe there may be several possible reasons for this. Firstly, BRfD is not limited to a fixed set of combined actions to select from. Instead, it can perform each action simultaneously. Secondly, ForgER is limited to two discretisations in opposing directions for the two camera actions. For BRfD, we analysed the demonstration data to create a rich selection of 8 discretisations for each camera. These reasons could also explain why ForgER delivered poor performance directly after pre-training.

6.3 Deep Q-Network

Although this ablation study is done to measure the performance impact of each improvement to our final algorithm, it would also be useful to know how they compare to the base DQN algorithm. Unfortunately, the very basic DQN algorithm failed to learn a remotely successful policy. Instead, we performed experiments on a branched version of DQfD[3] without n-step loss, Duelling- or Double DQN. Naturally, we expect BRfD to outperform the simplified DQfD.

6.3.1 Results

In Figure 6.2 we show results of the experiments on DQfD. Similar to BRfD, DQfD started with a somewhat good policy. From its starting minimum reward of 48, the DQfD algorithm improved to a local maximum of 53. For the remaining 230 episodes, the reward remained relatively consistent, albeit with a slight upward trend.

6.3.2 Discussion

As expected, BRfD outperformed DQfD with a rolling mean reward of 5. Keep in mind that BRfD is bound by the maximum reward. We observe the same phenomenon of drastically increasing performance in the first 70 episodes. Unlike BRfD, the reward of DQfD does not approach the ceiling reward of 64. Therefore, we can more clearly observe the point at which we stopped decreasing the presence of demonstration samples in training batches. We linearly decrease the proportion of demonstration samples in training batches

[3]Similar to BRfD without any Rainbow improvements.

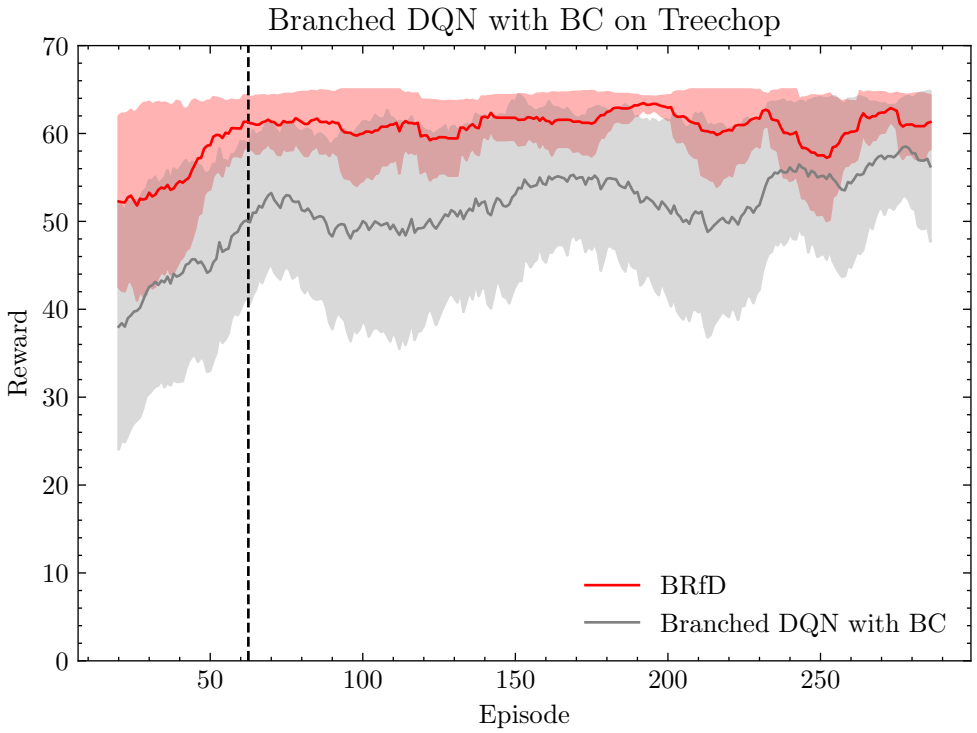

Figure 6.2: DQN ablation learning curve on Treechop environment after pre-training. This ablation shows the performance drop of removing all Rainbow improvements.

from 1.0 to 0.5 in 125×10^3 steps[4]. Assuming max episode length, 125×10^3 amounts to about 62.5 episodes.

Finally, the performance of DQfD shows a slow upward trend. It is possible that, given more time, DQfD could have reached the performance of BRfD. In their extensive ablation study on the Atari benchmark, Hessel *et al.* [7] found that DQN is capable of performing on par with Rainbow in a few games. However, DQN was almost never able to learn faster. We believe BRfD should continue to perform better and more consistently across a variety of seeds when compared to DQfD.

6.4 Behavioural Cloning

In their paper proposing DQfD, Hester *et al.* [18] attributes their success partly to the fact that they consider the reward received. BC ignores the reward received and treats the problem like a supervised learning problem—with the states and actions as inputs and ground truths. Since BC cannot use agent collected experience, we fix the network after pre-training and evaluate for the 300 episodes. For BRfD without BC, we used noisy

[4]Note that whenever we mention a number of steps taken by our agent, we exclude skipped steps. Therefore, 125×10^3 steps by the agent translates to 500×10^3 environment steps.

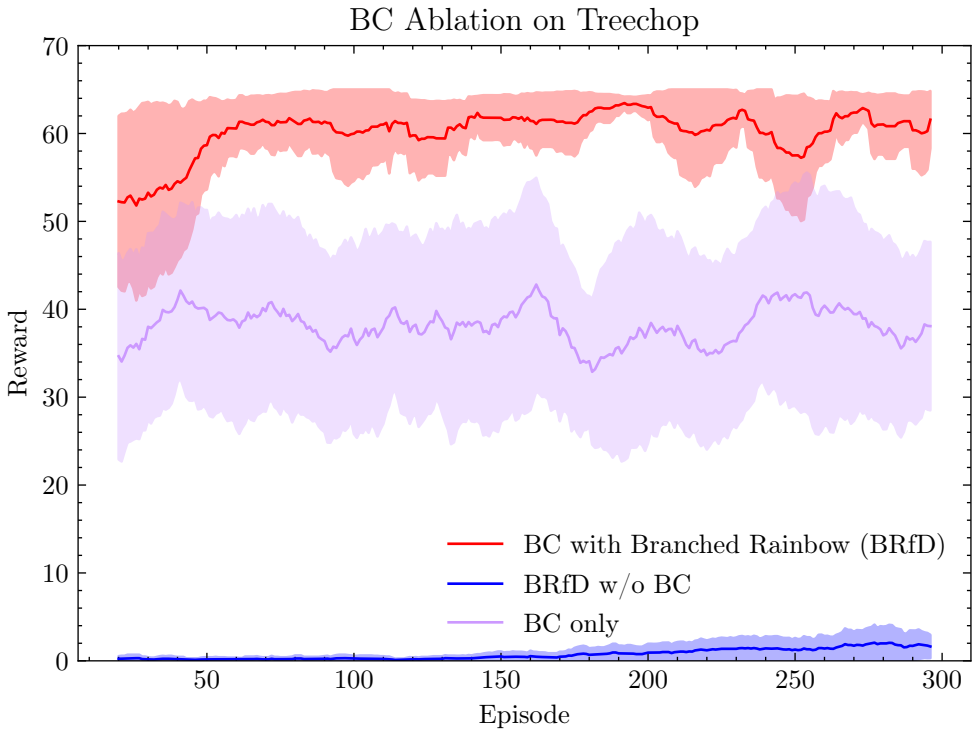

Figure 6.3: BC performance on Treechop environment after supervised training on demonstration data. This ablation shows the results of excluding RL methods.

networks for exploration. Considering the findings of Hester *et al.* [18], we expect BRfD to outperform both pure BC and BRfD without BC. Note that we used a branched DQN NN architecture for BC (similar to Section 6.3). Additionally, we implemented all the relevant modifications for pre-training described in Section 5.3.

6.4.1 Results

In Figure 6.3 we can see the results of this experiment. The BC algorithm starts with a poor reward of 35. Of course, since we do not learn on agent collected experiences, BC did not improve over the course of 300 episodes. On average, performance varied around a reward of 40. Without BC, our Branched Rainbow algorithm was barely able to reach a reward of 2. This roughly coincides with the findings of Guss *et al.* [14].

6.4.2 Discussion

It is clear that BRfD significantly outperforms both BC only and BRfD without BC. We do however concede that BC is an extremely simple algorithm incapable of learning from experience, and without IL the Treechop problem is rather challenging. With a

standard deviation of around 12, the performance of BC is also inconsistent throughout. For reference, the largest standard deviation in the performance of BRfD is below 10 and decreases to around 4. When looking at the learning curve of BRfD with and without BC, it is clear that IL determines the success of BRfD—albeit with limited training time.

6.4.3 Efficiency

BC is significantly more computationally efficient when compared to BRfD. However, it comes at a great cost of performance and generalisation. Unless the demonstration data is perfect and explores the full state- and action space, BC will almost always under-perform compared to approaches that combine IL with RL. On the other hand, RL alone usually requires a huge number of experiences gathered by training for days at a time. As can be seen from the results, the addition of a simple BC loss function reduces this training time significantly.

6.5 Double DQN

The DDQN improvement attempts to address the overestimation of action values by the original DQN algorithm. While DDQN succeeded in its goal and delivered improved performance, Hessel *et al.* [7] found the improvement to provide no benefit when combined with C51. They hypothesised that the clipping of action values to a maximum value (usually less than the maximum possible return) led to underestimation rather than overestimation. We expect a similar result since none of the additional improvements we introduce to Rainbow addresses this problem.

6.5.1 Results

Figure 6.4 shows the results of the experiments for BRfD without DDQN. While it started slightly better than BRfD, we can observe an immediate decrease in performance to a minimum reward of 47. From episode 40 onward, its performance started to increase gradually to match BRfD.

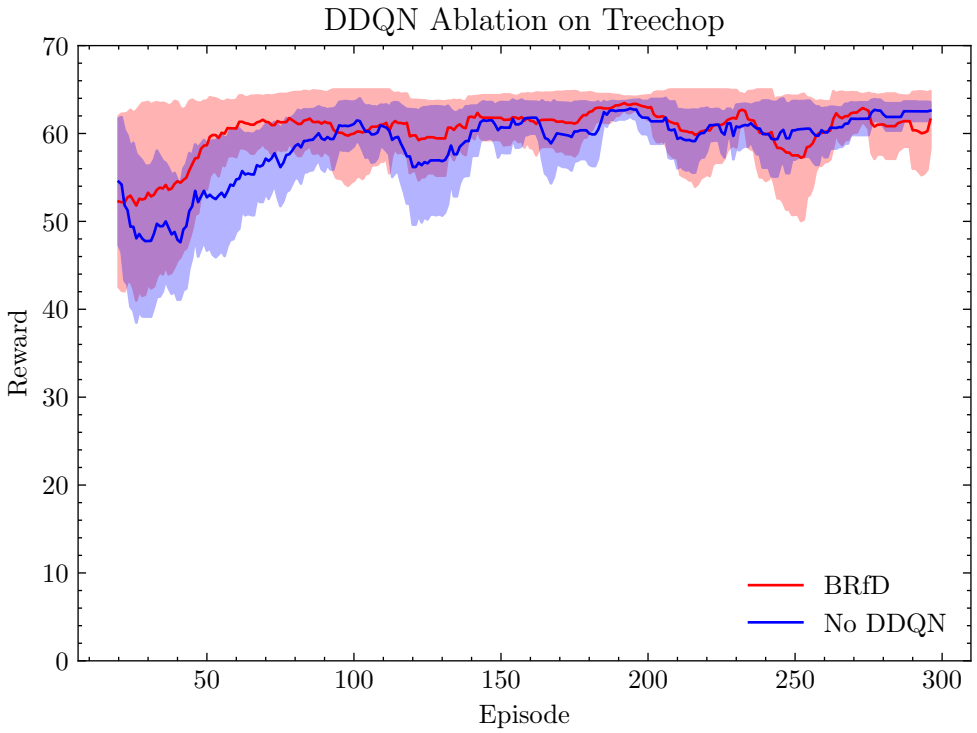

Figure 6.4: Double DQN ablation learning curve on Treechop environment after pre-training.

6.5.2 Discussion

Unsurprisingly, our findings corroborate the findings of Hessel *et al.* [7]. However, we do observe a slight initial downward trend of performance when starting learning. While this could have been entirely arbitrary, we hypothesise that the supervised loss used in BRfD blindly maximised expert selected actions without much regard for the reward. This again led to an overestimation. Note that in our case, we use $V_{\max} = 50$ for C51; using a higher $V_{\max}$ would mean our resolution for the value parametric distribution is larger than one. As shown in Equation 6.1, if we assume a near perfect episode where a reward is received every step for 64 steps, the maximum return for an episode remains within the limits of our value distribution. This leaves very little room for overestimation. Based on our findings, DDQN is not a necessary addition to our algorithm.

$$\begin{aligned} G_t &= \sum_{k=0} \gamma^k R_{t+1} \\ &= 47.44 \\ G_t &\leq V_{\max}. \end{aligned} \tag{6.1}$$

6.5.3 Efficiency

When training the agent, DDQN requires an extra forward pass through the NN. Depending on the size of the NN, this can become a costly operation to perform on every learning step. On the other hand, DDQN has no additional computational cost when evaluating the agent. Therefore, removing DDQN would only slightly speed up the training process at the cost of potentially overestimating the Q-values.

6.6 Duelling DQN

The Duelling DQN architecture was designed to improve generalisation in cases where the action taken has little to no impact on the expected return. For example, in the Treechop environment, the value branch may learn to only pay attention to trees. On the other hand, the advantage branch may also learn to pay attention to parts of the environment that are difficult to navigate. Wang *et al.* [4] found that the architecture performs as intended, and delivered increased performance. However, Hessel *et al.* [7] found the addition of the duelling architecture to be either beneficial or detrimental depending on the game. We hypothesise that the Duelling DQN improvement provides a small bonus to the performance of BRfD.

6.6.1 Results

We show the plot for this ablation in Figure 6.5. In the first 50 episodes, a slight drop in performance can be observed. The agent reached a minimum reward of 51 before trending upwards. While it is slightly unclear at which point the reward stabilises, we can observe the standard deviation trend flattening out at around episode 120. We therefore infer that performance stabilised at a reward of 60. The agent reached a maximum reward of 62.

6.6.2 Discussion

It seems our hypothesis was incorrect. The Duelling DQN ablation produced a relatively similar learning curve to that of BRfD. We can however observe that BRfD learned slightly faster directly after pre-training. As with the DDQN ablation, we notice the same initial drop in performance in the first 20 episodes. In this case, the drop was much less severe and may have resulted from decreased generalisation mentioned in the work of [4].

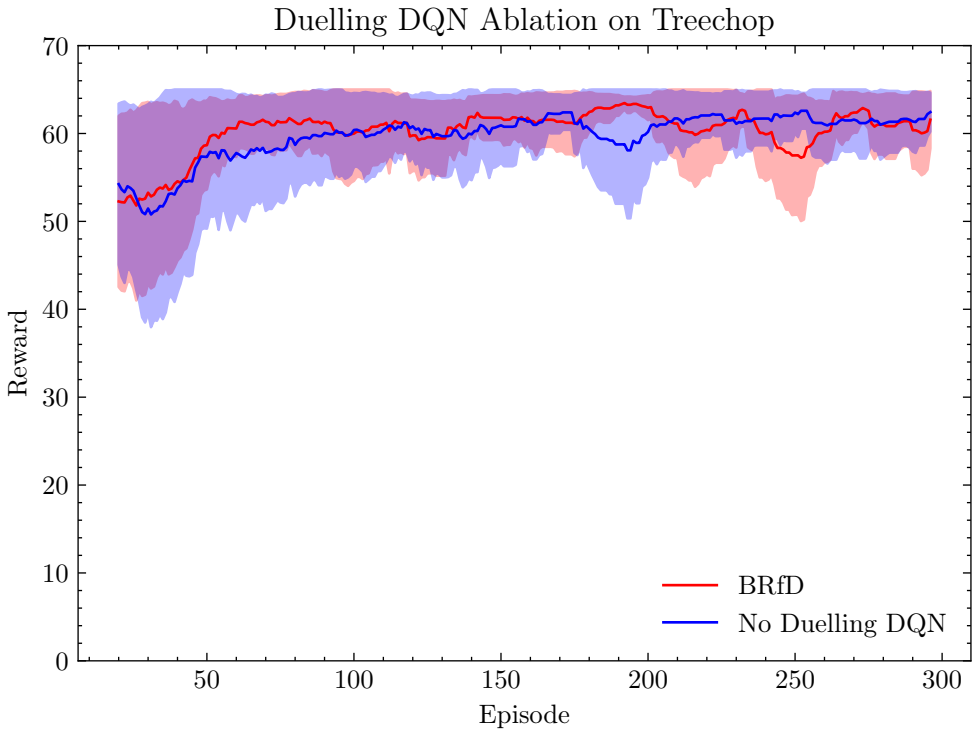

Figure 6.5: Duelling DQN ablation learning curve on Treechop environment after pre-training.

6.6.3 Efficiency

The Duelling DQN improvement modifies the NN architecture of an agent. Therefore, it increases the computational cost of both training and evaluation. This cost depends entirely on the size of the network at the output layer. For example, with C51 and BDQ, our output is both branched and much larger. If computational cost is of extreme importance in one (or both) of these cases, we would remove Duelling DQN from BRfD.

6.7 Prioritised Experience Replay

The original DQN algorithm trained on samples of previous experiences. The algorithm sampled the buffer at a uniform rate. In the pure RL case, this may lead to important experiences not being sampled for training. PER addresses this issue by sampling experiences based on the TD error. The algorithm delivered improved performance over the base DQN algorithm, especially in terms of learning speed. However, in our case, this problem may be redundant since we have access to a set of expert demonstrations. We therefore hypothesise that PER provides almost no benefit to the BRfD algorithm.

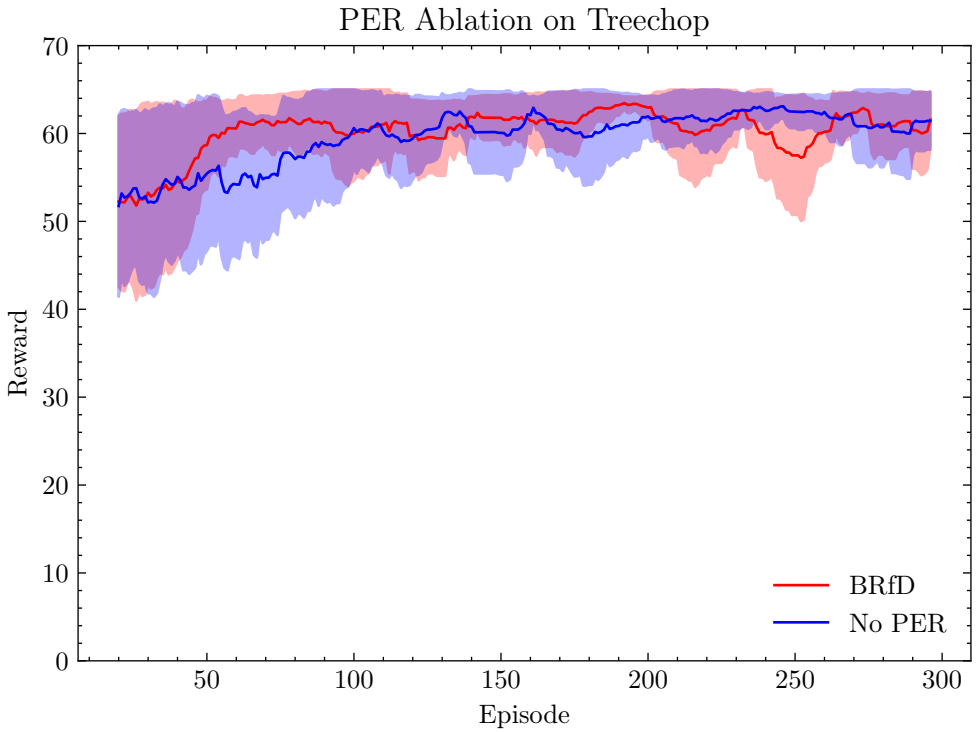

Figure 6.6: PER ablation learning curve on Treechop environment after pre-training.

6.7.1 Results

In Figure 6.6 we show the learning curve when ablating PER. In the figure, we can observe a slow upward trend in performance up until episode 110. From there, the learning curve is very similar to that of BRfD.

6.7.2 Discussion

From the results, we can confirm that after 300 episodes of training, PER had little to no impact in the Treechop environment. However, sampling with priority did seem to help with phasing in RL. We believe that discrepancies between demonstration data and agent collected experiences may have led to larger TD errors in the latter case. That means PER started to learn from agent collected experiences much earlier and faster. Therefore, we can conclude that our hypothesis was incorrect and PER helps the agent to learn faster from the new set of unseen data, but does not result in an improvement in the overall performance once the agent is trained.

6.7.3 Efficiency

Similar to DDQN, PER comes at no cost during evaluation. During training however, PER requires sampling experiences with certain priorities. This can be a rather computationally expensive implementation. It also requires additional data storage for the priorities of samples. In uniform sampling, we can generate a random index (from a uniform distribution) and fetch the sample with constant time complexity. The priority update can also be performed in constant time complexity. When we need to sample with priorities, the best we can do is logarithmic time complexity[5]. Considering the above-mentioned (and in contrast to our hypothesis) we believe PER to be a crucial addition to the BRfD algorithm. On more challenging tasks where the demonstration data does not provide a good starting policy and rewards are much more sparse and difficult to achieve, PER may determine the success of the algorithm.

6.8 Multi-step

Based on the findings of Hessel *et al.* [7] and Silver [10], we believe multi-step (previously referred to as n-step) to be the most important addition to BRfD. Specifically, there are two possible reasons for this. Firstly, as discussed in Section 2.6, multi-step is a middle ground between MC- and TD learning. The selection of n allows us to balance the amount of variance and bias between the two methods. Secondly, multi-step reduces the possible error in estimation by the NN by partially grounding the estimation in samples from reality. For those reasons, we expect the multi-step improvement to be a valuable addition to BRfD.

6.8.1 Results

In Figure 6.7 we present the learning curve of BRfD without multi-step learning. Similar to full BRfD, it starts with a rather good reward of 50. For the rest of the 300 episodes, its performance trends only slightly upwards. Inspecting the figure, we can see that it achieved a maximum reward of 53. Excluding multi-step learning also seemed to greatly reduce the stability of the algorithm across a variety of seeds.

[5]This is achieved with a sum-tree data structure. See Schaul *et al.* [29].

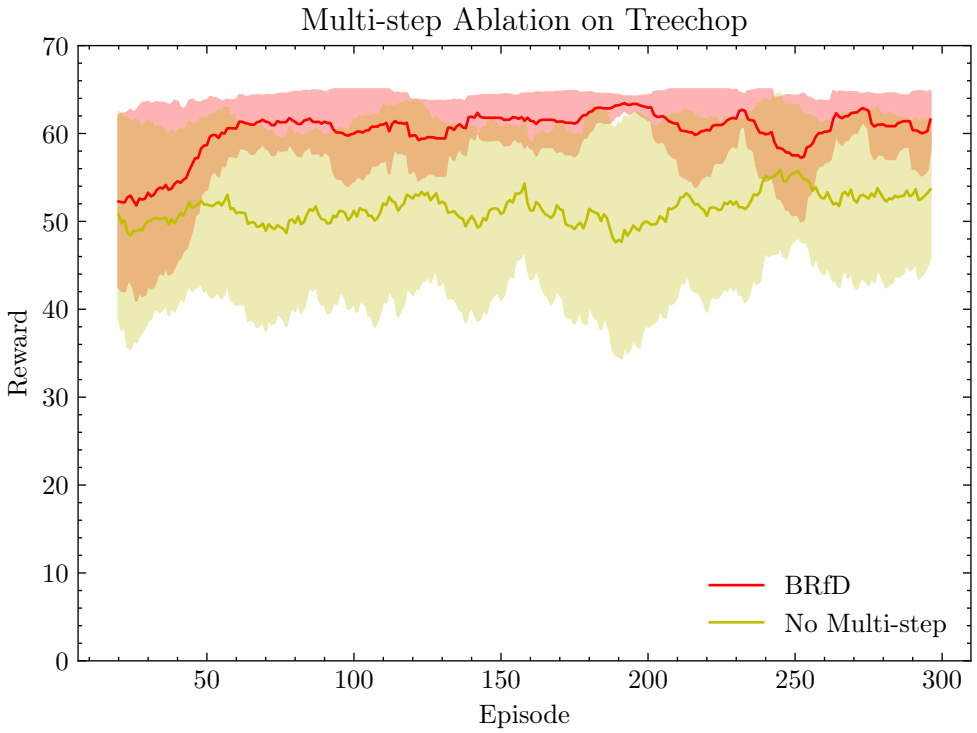

Figure 6.7: Multi-step DQN ablation learning curve on Treechop environment after pre-training.

6.8.2 Discussion

As expected, multi-step learning drastically impacted performance. Without it, the algorithm seems unable to improve upon the policy learned from pre-training. It also has trouble generalising across a variety of environment seeds. Later, in Section 6.12 we will compare the improvements to each other. We are particularly interested in how this ablation compares to the simplified Branched DQfD ablation.

6.8.3 Efficiency

Switching from 1-step to n-step learning comes at a low cost. Instead of storing $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$, we use $g_n = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1}$ to calculate the return for the next n steps. We then store a new tuple $\langle s_t, a_t, g_n, s_{t+n} \rangle$ for multi-step learning. However, in our algorithm we implement the four-component loss of DQfD defined in Equation 4.15. This means that we calculate both the 1-step and n-step losses. We believe that BRfD should work without the 1-step loss since Hester *et al.* [18] use it to ground the margin loss in realistic values. Considering the performance impact, the cost of multi-step learning is negligible.

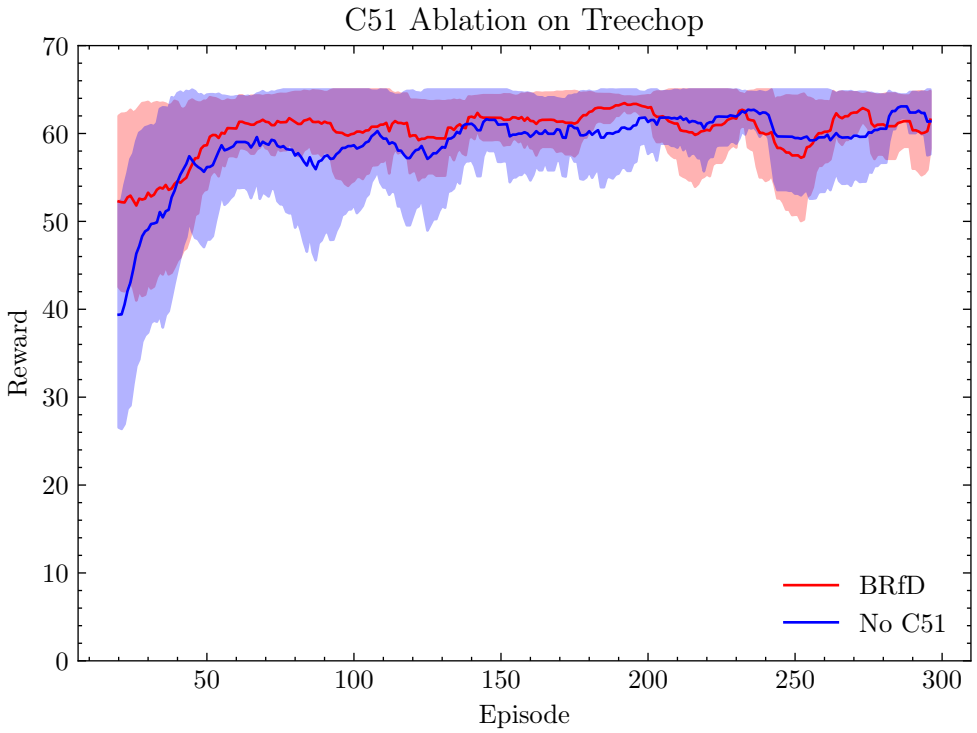

Figure 6.8: Categorical DQN ablation learning curve on Treechop environment after pre-training.

6.9 Categorical DQN

As discussed in Section 4.1.4, taking the expectation of the action-value function may not necessarily capture the dynamics of the environment effectively. C51 attempts to address this by directly modelling a parametric distribution for the action-value function. The authors of Rainbow DQN perceive C51 to be one of the more important additions to the algorithm. They found that the benefits of C51 increase as the agent converges to the best policy. However, it is difficult to predict the effects of pre-training on distributional RL. Based on the findings of Hessel *et al.* [7] and Bellemare *et al.* [5], we hypothesise that the C51 did have some impact on the performance of BRfD.

6.9.1 Results

We show the results of ablating C51 in Figure 6.8. The ablation achieved a moderate reward of 40 directly after pre-training. Thereafter, its performance rapidly increased to 58 in the span of 20 episodes. At the end of 300 episodes of learning, the ablation nearly matched the performance of BRfD.

6.9.2 Discussion

We found these results to be rather interesting. While it is clear that C51 is beneficial to our algorithm, we did not expect the ablation to match the performance of BRfD near the end of training. Based on the standard deviations, C51 also seems to deliver more consistent performance.

Directly after pre-training, the performance of the C51 ablation drastically increased. This increase coincides with the gradual introduction of agent-collected experiences into training batches. However, this phenomenon is not present in any of the previous ablations on the Rainbow DQN algorithm. Although this may be entirely arbitrary, we suspect that C51 was able to learn a better starting policy from the demonstration data.

6.9.3 Efficiency

The C51 improvement increases the NN size of BRfD by roughly 22%. In their paper, Bellemare *et al.* [5] mentioned that C51 trains at roughly 75% the speed of DQN. While this metric may not hold across all systems, it gives an idea of how computationally expensive the improvement is relative to base DQN. Considering the fact that BRfD is bounded by the maximum reward attainable as well as the findings of previous work [5; 7] we believe the C51 improvement to be a moderately important addition to BRfD.

6.10 Convolutional Neural Networks

As discussed in Section 4.3.1.3, the authors of the algorithm that placed second in the 2019 MineRL challenge investigated several CNN architectures. We implemented the most successful architecture of these: Double Deep Impala with Fixup initialisation [8; 36; 19]. Anecdotally, in our experience we have found the nature CNN to perform better than large, complicated CNN architectures in pure RL problems. It is therefore difficult to predict the outcome of these experiments. However, based on the findings of Amiranashvili *et al.* [9], we hypothesise that the Double Deep Impala CNN architecture enhanced BRfD.

6.10.1 Results

Figure 6.9 shows the learning curve of the Double Deep Impala CNN applied to BRfD. It starts with an impressive reward of 60. After a minor drop in the next 20 episodes, it

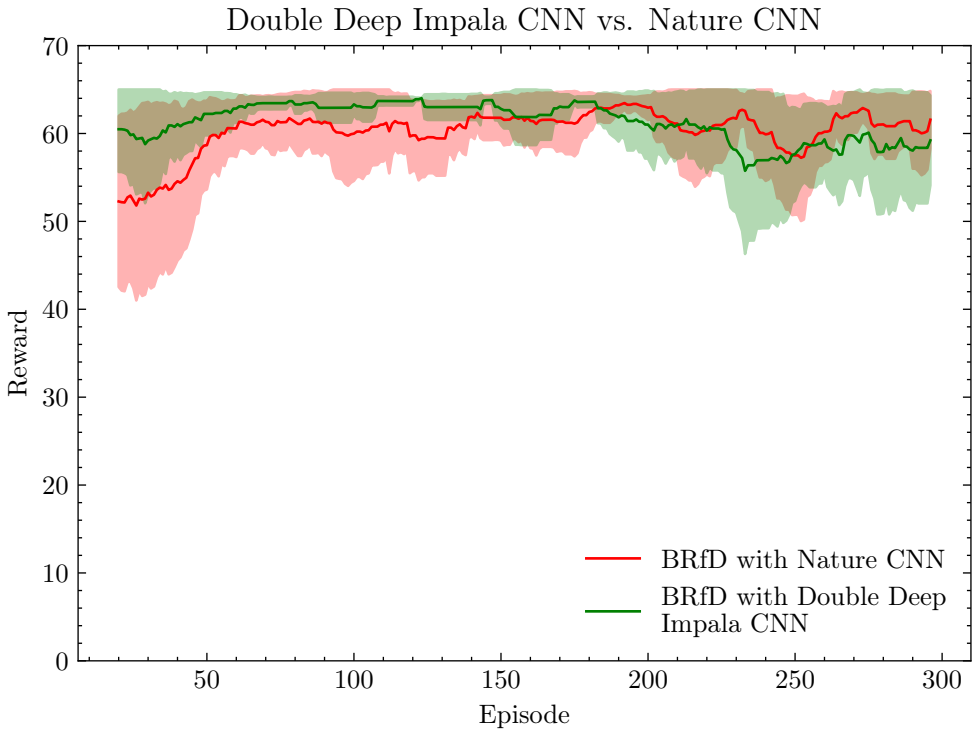

Figure 6.9: Alternative CNN architecture learning curve on Treechop environment after pre-training. This architecture was taken from the work of Espeholt *et al.* [8] and modified with increased channels. We evaluated this architecture based on the findings of Amiranashvili *et al.* [9].

increases to a maximum reward of 64. It also delivered a stable performance and reached a minimum standard deviation of near-zero at episode 121. At around episode 182 however, performance drops to below that of BRfD.

6.10.2 Discussion

The results show that the Double Deep Impala CNN architecture delivered improved performance over the nature CNN. However, at around episode 182, we notice the performance dropping to slightly below that of BRfD. This is especially interesting considering the fact that at around episode 62, the training batches were fixed at 50/50[6]. As mentioned previously, in our experience, larger and more complicated CNN architectures tend to have trouble in pure RL problems compared to the nature CNN. In this case, the Double Deep Impala architecture performed exceedingly well in the pre-training, but near the end of the 300 episodes its performance dropped. This evidence is not sufficient to conclude that either of the CNN architectures are the better design choice.

[6]50% demonstration data, 50% agent collected experiences.

6.10.3 Efficiency

Although the Double Deep Impala CNN architecture significantly outperforms the nature CNN after pre-training, it comes at a great cost. The former architecture has 2.15 M parameters compared to 94.37 k for the latter. That is a staggering 2178% increase. However, Amiranashvili *et al.* [9] implemented a scaled down version of this network. The Deep Impala CNN architecture halves all the CNN filter channels to reduce the number of parameters to 540.38 k. This is still 472% larger than the nature CNN. Therefore, we retained the nature CNN architecture for BRfD.

6.11 Data Augmentation

As described in Section 4.3.1.4, we applied the horizontal flipping data augmentation technique to effectively double the demonstration data. The augment can be implemented efficiently by only applying the flip to the original POV image when it is required. Therefore, we store every observation only once in memory, but twice in the replay buffer. Since we flipped the POV observation, the camera yaw actions performed in the demonstrations did not correspond to the observation. This was easily addressed by copying the action and flipping the camera yaw action. We expect the data augment to provide some benefit to the BRfD algorithm.

6.11.1 Results

Figure 6.10 shows the results of removing the flip augment. It starts with a moderately good reward of 38. From there, its performance trends upward with a similar slope to that of BRfD. For the remaining 240 episodes, we see another slight increase before the performance stabilises at a reward of 54. Inspecting the standard deviation of the plot, we also notice that the augment leads to more consistent performance in BRfD.

6.11.2 Discussion

Unsurprisingly, the results indicate that the augment enhanced the performance of BRfD. We did not however expect it to be with such a large margin. We hypothesise that the flip augment helped the NN to generalise better. Instead of a single example of when to turn left/right, the augment provides an additional example for the opposing direction. This

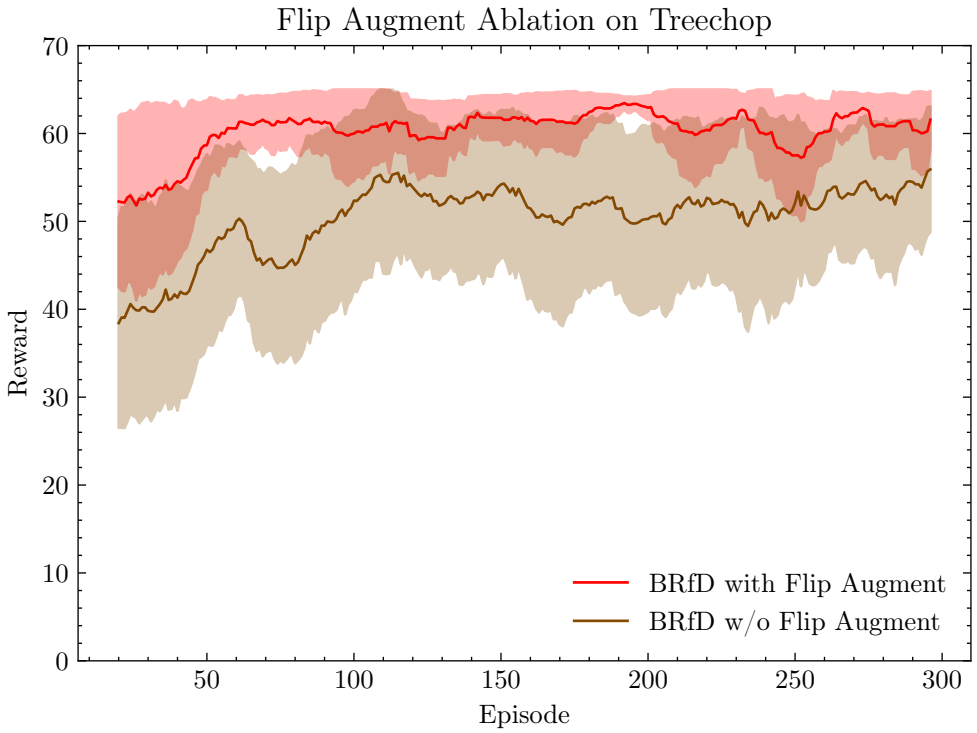

Figure 6.10: Data flipping augmentation ablation learning curve on Treechop environment after pre-training. We evaluated this augmentation to determine the effectiveness of augmenting the data with flipped observations.

means that the agent may have learned a better policy pertaining to selecting the camera yaw action.

6.11.3 Efficiency

This augment is a relatively inexpensive addition to BRfD in terms of computation cost and storage. For a rather large performance increase, we only need to store another reference to the observation. When the observation is sampled to be trained on, we simply flip the image horizontally. It also only affects the performance during training.

6.12 Full Comparison

To finish off the ablation study, we superimpose the results of all the experiments on a single graph shown in Figure 6.11. Note that we excluded standard deviations in favour of readability. Investigating the plot, we notice some interesting trends in the ablations.

Firstly, our pure RL Branched Rainbow algorithm seems unable to learn any useful behaviour in 300 episodes. Our experiments show that the algorithm is barely able to

obtain a single log in an episode. While we did not run the experiment for more than 300 episodes, we believe that a pure RL algorithm like Agent57 may be capable of solving the Treechop environment, albeit in considerably more episodes [43].

Secondly, the BC algorithm delivers average performance. Since it does not learn from the experience gathered by the agent, it does not improve over the course of 300 episodes. However, when we consider the starting point of all the other ablations (especially the Double Deep Impala CNN architecture) we can conclude that Hester *et al.* [18] was correct in their assessment that the consideration of the reward is crucial when learning behaviour from demonstrations for RL problems.

Lastly, the reader may have noticed nearly all the results trending slightly upwards throughout the 300 episodes of learning. This indicates that most of the improvements can be individually removed without damaging evaluation performance too much. It may however require many more episodes of training to reach comparable performance.

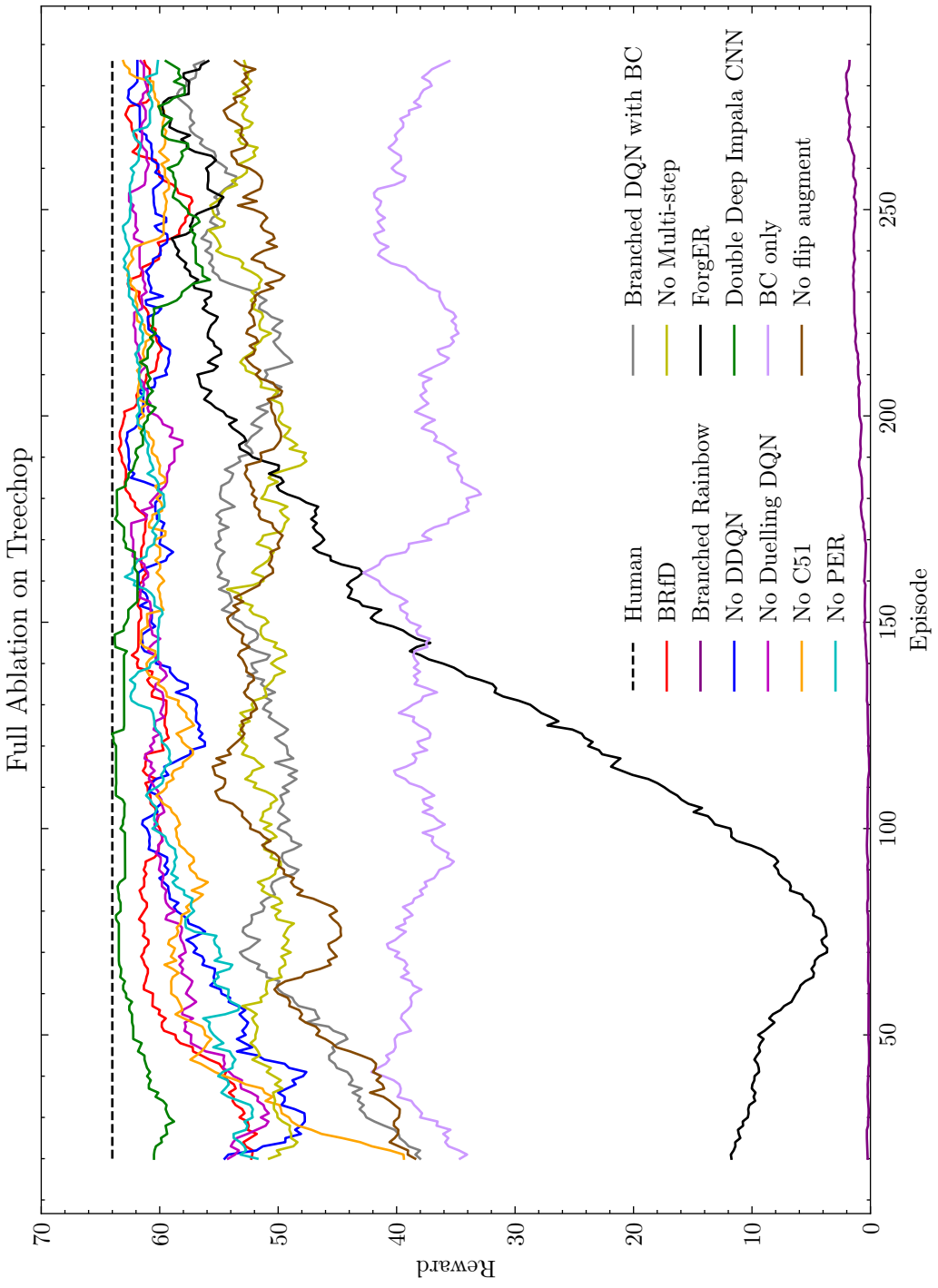

Figure 6.11: Learning curves of all ablations evaluated on the Treechop environment. This puts all the experiments into perspective for comparative analysis.

6.13 Summary

In this chapter, we performed a thorough ablation study on our BRfD algorithm. We trained all the algorithms for 3 sets of 300 episodes with different seeds. The results of these experiments then allowed us to evaluate the impact of each component of our algorithm.

With the experiment on BRfD in the Treechop environment, we showed that our algorithm was not only able to solve the environment, but also the issue of trial and error training. Additionally, our algorithm outperformed ForgER, the winning submission of the 2019 MineRL challenge in this environment.

The learning curves of BC only and Branched Rainbow without BC showed that the best approach is a combined one. With BC only, the agent was not capable of improving based on new or unseen experiences. Without BC (only RL), the problem was too difficult to find a solution within 300 episodes. Therefore, we included a BC loss in the RL training process for BRfD.

Based on the results of the DDQN ablation and the findings of Hessel *et al.* [7], we believe the improvement to be inconsequential compared to others. However, at the relatively low cost of an extra forward pass through the network during training, we opted to include the improvement since it only seemed to have a positive impact on BRfD.

Similar to the DDQN ablation, the ablation of Duelling DQN had little impact on the performance. As noted by the authors of the algorithm, it is most beneficial in cases where the action has little to no impact on the future value. It is therefore possible that the inclusion of BC and data augmentation may have negated the need for Duelling DQN. We opted to include the improvement to keep performance more consistent.

The PER improvement was also rendered nearly obsolete by BC. The results showed that the ablation learns slightly slower with slightly less stable performance. However, due the fact that we use the priorities to augment option data, we included PER in our final algorithm.

The multi-step improvement proved to be the most impactful DQN improvement included in our algorithm. The improvement was able to successfully reduce the prediction errors of the NN. Considering the performance impact is rather inexpensive, we retained multi-step learning in BRfD.

Ablating C51 from BRfD had little impact on performance. Aside from the poor start, the ablation was able to match the performance of BRfD after 150 episodes. We would like

to note that the performance of BRfD was bounded by the maximum reward attainable in this environment. We believe that the C51 improvement may be beneficial in the more complex ObtainDiamond environment. Therefore, C51 was included in the BRfD algorithm.

Replacing the nature CNN proposed by Mnih *et al.* [3] with a deeper network seemed to deliver increased performance at first. However, after a period of RL training, the performance dropped to below that of the nature CNN model. Although the results showed promise for pure IL approaches, they remain limited by the quality of demonstrations. Based on the results, we used the nature CNN architecture.

Augmenting the demonstration data with horizontally flipped frames delivered drastically increased performance. At the low cost of flipping the input image and camera yaw action, it is clear that the augment is a valuable addition to the algorithm.

While we decided to retain nearly all the methods evaluated in this chapter, we believe that our findings may assist future research in deciding which components are negligible considering the cost. Specifically, our findings in Section 6.10 showed that a deep, complicated CNN architecture greatly benefits IL methods. On the other hand, it harms performance of RL methods. We also showed that certain data augmentation methods can improve the performance if RL algorithms.

7 Hierarchical Reinforcement Learning Experiments

Now that we have established that BRfD successfully solves the Treechop environment, we can start implementing an HRL agent to face the ObtainDiamond environment. As mentioned in Section 1.2.3, finding a diamond in Minecraft can be an incredibly difficult task—even for humans. We therefore lower the goal for our experiments from a diamond to an iron pick-axe. If the agent crafts the iron pick-axe, it only needs to explore to find a diamond. In this chapter we attempt to address the final objective of this work: to discover and navigate the hierarchy of sub-tasks in a generalised manner. We implement a slightly modified version of the options framework proposed by Sutton *et al.* [23]. For each option, we use our BRfD algorithm as the policy π_ω. By separating the policies for each sub-task, we hope to prevent the forgetting of behaviour pertaining to solving the sub-task.

7.1 Training Procedure

Similar to the ablation study on Treechop, we performed experiments in three sets of seeds. However in this case, the learning curve is not a particularly useful metric. We found the reward curves to reveal little information about the items obtained. We therefore trained the agents for the full 8 million environment steps. Thereafter, we evaluated the agents on three sets of 100 seeds in the non-dense ObtainDiamond environment. In these evaluations we infer the items obtained by inspecting the rewards received. For example, a reward of 35 means the agent obtained at least one of each of the following (rewards in parentheses): log (1), planks (2), sticks (4), crafting table (4), wooden pick-axe (8) and cobblestone (16). Using this method we were unfortunately unable to differentiate between similarly valued items[1].

We show the results by calculating and plotting the proportion of episodes in which the agent obtained a given item. This is done by inspecting the logged rewards and inferring

[1] A reward of 11 could mean either sticks or crafting tables were obtained.

the obtained items for the three sets of 100 episodes. Note that during the evaluations, we observed some anomalous rewards. Specifically, in the game of Minecraft, it is possible to obtain sticks without using planks to craft them. In a few evaluation episodes the agents managed to obtain a stick and nothing else[2]. We classified these episodes as anomalies (or as 'no items obtained') since the agent likely only stumbled upon it by chance rather than actively searching for the items.

7.2 Hierarchical Agent Results

In Figure 7.1 we show the results of our HRL algorithm. The agent managed to obtained a log in about 63% of episodes. In the majority of these episodes, the agent had little trouble crafting planks and either sticks or crafting tables. However, we can observe a steep decline in frequency for crafting both sticks and a crafting table. We believe this may have occured in cases where the agent only collected a single log before crafting planks. It then only had sufficient resources to craft either sticks or a crafting table.

In these episodes, the agent needed to place the crafting table somewhere nearby and perform the craftNearby action for the wooden pick-axe. From Figure 7.1 we can see that the agent found this to be an incredibly difficult task. It succeeded in crafting a wooden pick-axe in a mere 5 episodes. Interestingly, in four of these episodes the agent managed to find cobblestone. In three of those four episodes, the agent managed to use a crafting table (either by crafting and placing a new one or using an existing one) to craft either a stone pick-axe or a furnace. In only one of these three episodes did the agent craft both of those items. Unfortunately the HRL model did not obtain an iron ore (and by extension an iron pick-axe and diamond) during our evaluations. Anecdotally, we observed our algorithm collecting an iron ore in a single instance.

[2]A stick can be obtained by breaking a dead bush or leaves.

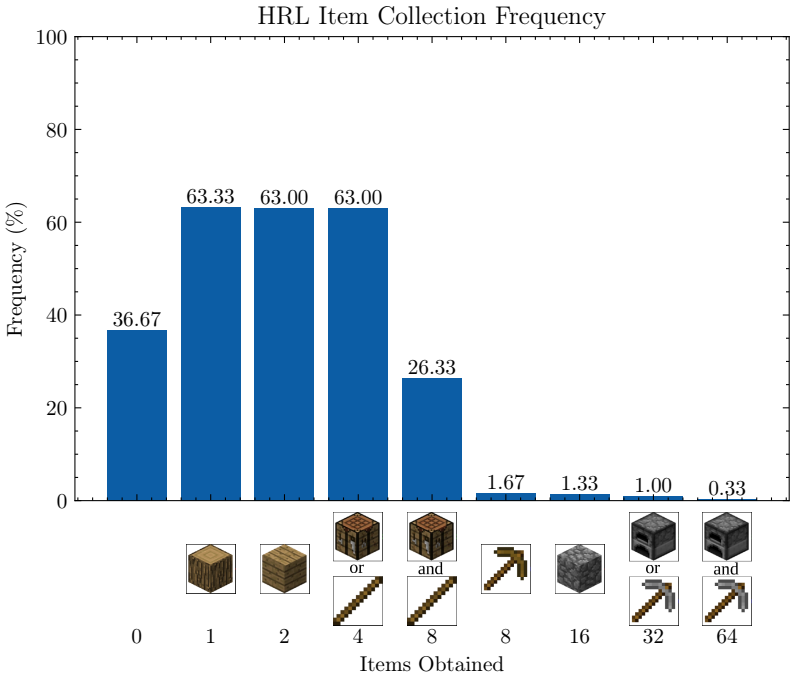

Figure 7.1: Proportion of 300 evaluation episodes in which our HRL algorithm was able to obtain a given item at least once. The first bar indicates the proportion where no relevant items were obtained.

7.3 Single Policy Agent Results

We contrast the previous experiment by performing the same training and evaluations on a single BRfD policy with no hierarchical structure. The results of this experiment are shown in Figure 7.2. Compared to the HRL it seems to be much more successful at finding and collecting logs. For all but one episode, the agent crafted planks and either sticks or a crating table. The same drop in frequency can be seen for crafting both of these items. However, it still managed to collect both more frequently.

In addition to the above-mentioned, BRfD also managed to craft more wooden pick-axes in the 300 episodes. However, it did not obtain a single stone pick-axe or furnace. These results show that an RL algorithm helps to prevent the network from forgetting behaviour pertaining to solving tasks later in the item hierarchy. Interestingly, the BRfD algorithm seemed much more capable of finding the items to obtain cobblestone. When it came to actually obtaining the cobblestone however, it was not able to do so at the same rate as the HRL algorithm.

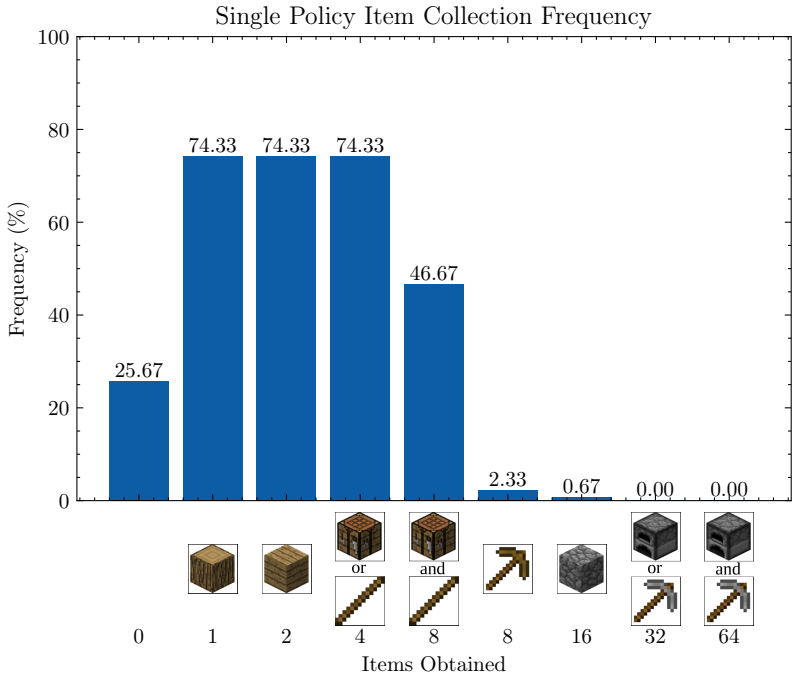

Figure 7.2: Proportion of 300 evaluation episodes in which a single BRfD policy was able to obtain a given item at least once. The first bar indicates the proportion where no relevant items were obtained.

7.4 ForgER Results

To place our algorithms into perspective, we include the results of the winning submission for the MineRL 2019 challenge. Note that these results were extracted from a table in the work of Skrynnik *et al.* [37]. As described in Section 4.3.2.5 they followed a chain of sub-goals derived from the demonstration data. For each sub-goal, a specific quantity of the relevant item is required to move to the next sub-goal in the chain[3]. They trained their agent without step or time restrictions and evaluated on 1000 episodes.

In Figure 7.3 we show the results of their algorithm on the non-dense ObtainDiamond environment. From these results, it is clear that their algorithm is much more successful than both our versions at finding and collecting logs. Interestingly, their results show that the agent does not always craft planks for all episodes where it finds at least a log. This is due to their hierarchical model only switching to the next policy if a certain amount of a specific item is present in the inventory. Specifically, in this case the agent was not able to collect 6 logs in all the episodes. However, if their agent did manage to find at least 6 logs and craft 24 planks, it also crafted sticks, crafting tables and wooden pick-axes.

If the ForgER++ agent crafted a wooden pick-axe in an episode, it also very likely managed to mine cobblestone and craft a stone pick-axe. The algorithm did seem to have trouble

[3]For example: 6 logs, 24 planks, 1 crafting table, etc.

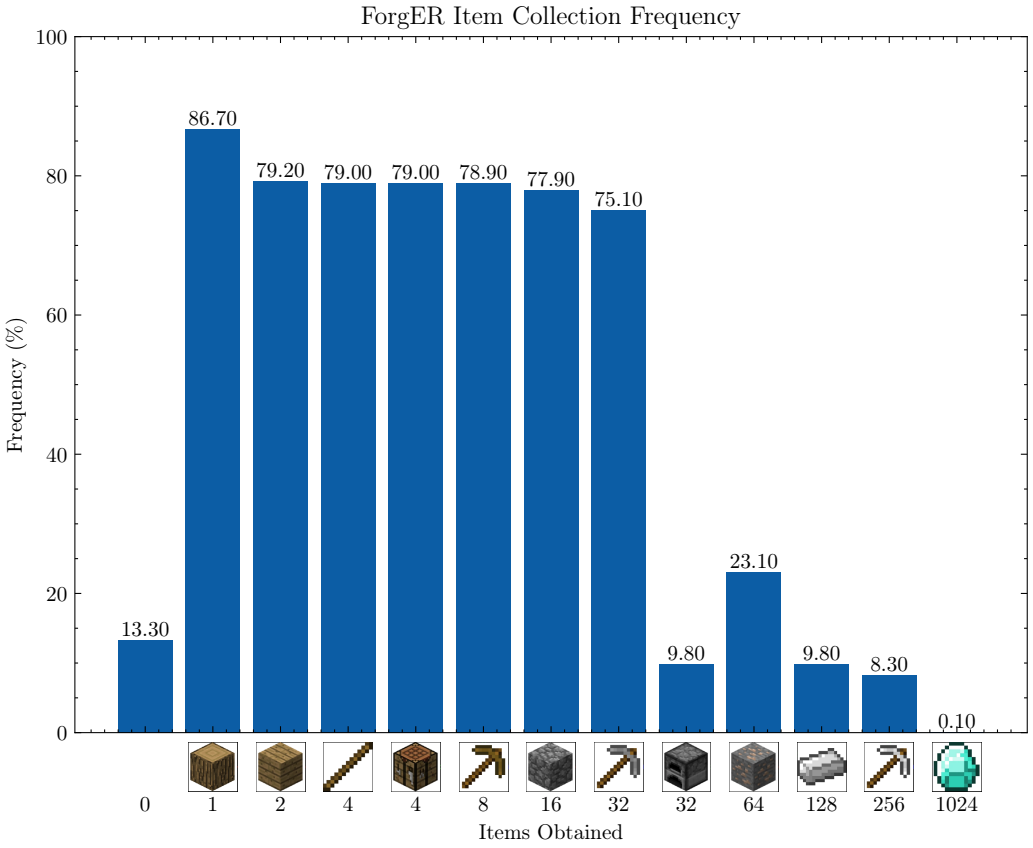

Figure 7.3: Proportion of 1000 evaluation episodes in which ForgER++ managed to obtain each item at least once. The first bar indicates the proportion where no relevant items were obtained.

with crafting a furnace. We believe that this is partly because most demonstrators found iron ore before crafting a furnace. Thus, the agent first looks for three iron ores, thereby decreasing the time available in an episode to craft a furnace or simply not finding three.

Another interesting observation is that ForgER++ successfully obtained iron ingot in all the episodes where it crafted a furnace. For roughly 85% of these episodes, it also managed to craft an iron pick-axe. Finally, the ForgER++ algorithm successfully obtained a single diamond in 1000 episodes of evaluation. We believe their success can be attributed to the features described in Section 4.3 and the increased training time.

7.5 Summary

Based on the results shown in this chapter, we can conclude that a hierarchically structured agent is able to retain its knowledge of later sub-tasks. Although the single policy agent was more proficient at solving the early sub-tasks and crafting wooden pick-axes, it was barely able to find cobblestone in a single episode. We found this result to be particularly

interesting. Theoretically, the hierarchical agent should learn better policies for each sub-task given its limited scope. And considering the fact that the agents almost always crafted planks if logs were obtained, we believe the HRL agent has the potential to progress even further.

It is clear that both our agents had trouble with crafting a wooden pick-axe. The wooden pick-axe is the first item in the hierarchy that has several requirements the agent needs to meet in order to craft. Not only does the agent have to craft and place a crafting table nearby, it also needs sufficient resources to craft a wooden pick-axe (two sticks and three planks). And considering the wooden pick-axe requires 9 planks to craft all the required items, it first needs to collect at least three logs to craft 12 planks. However, if even some of these planks are erroneously used to craft unnecessary items, the agent may not be able to progress past that point if more logs are not obtained.

Unfortunately, ForgER++ outperformed all versions of our algorithms. As noted previously, we extracted the results from their paper (table 3 [37]) where they trained the agent for longer and evaluated on 1000 episodes. Not only was ForgER++ able to obtain an iron pick-axe more often than we were able to obtain a wooden pick-axe, ForgER++ managed to obtain a diamond during evaluation. It was also much more proficient at finding and collecting logs.

7.5.1 Objectives Achieved

At the start of this chapter we mentioned the remaining three objectives we aimed to address. Firstly, we discovered the hierarchy of sub-tasks by evaluating rewards received by the human demonstrators in the non-dense ObtainDiamond environment. With this knowledge we trained a classifier to identify the best low-level policy to take control based on inventory content changes—thereby navigating the hierarchy of sub-tasks. Although this method is one possible solution to the problem, it was not very effective. Therefore, while we found a way to discover and navigate the hierarchy of sub-tasks, we believe our method is limited and not the most effective.

Secondly, we were also able to segment the demonstrations into separate trajectories for each sub-task using the hierarchical reward structure. We then trained the low-level policies mostly on trajectories relating to the specific sub-task. This method successfully reduced the forgetting of behaviour in a long-term task. Unfortunately it seems that it also affected the performance of earlier sub-tasks. The reason for this is unclear and requires further investigation.

Lastly, our algorithms were unable to compete against the ForgER++ algorithm on the MineRL ObtainDiamond environment. We would like to note that we attempted to keep our solution as generalised as possible. We did not access the inventory contents to track specific items for any other reason than logging.

8 Conclusion

In this thesis we proposed a hierarchical reinforcement learning (HRL) agent to solve the 2019 MineRL challenge. Our agent consisted of a high-level policy selecting between several low-level policies. For these low-level policies, we proposed a combination of improvements and modifications to the Deep Q-Networks (DQN) algorithm. The resulting algorithm, referred to as Branched Rainbow from Demonstrations (BRfD), combines action branching [6] and Rainbow DQN [7] with behavioural cloning (BC) to enable both imitation learning (IL) and RL.

In order to design a effective low-level policy capable of leveraging demonstration data, we performed an extensive ablation study on our BRfD algorithm in the Treechop environment. The algorithm successfully leveraged the demonstration data to reduce its reliance on a trial and error for learning the dynamics of the environment. However, to some extent, it is sensitive to imperfect demonstrations—especially adversarial behaviour. For each ablation we also analysed the computational cost in terms of performance impact. We found our two additions to Rainbow DQN to reduce the impact of certain improvements compared to the findings of Hessel *et al.* [7]. Finally, based on the results of our BRfD algorithm, we can conclude that the agent was able to learn to act in unseen scenarios.

To address the final objective of discovering and navigating the hierarchy of sub-tasks, we implemented a HRL algorithm in the form of an options framework. Unfortunately we were unable discover options dynamically in a generalised manner. Instead, we extracted the options from the demonstration data by exploiting the increasing reward magnitudes—a common feature in fundamentally HRL problems. We learned a strategy for navigating the options by training a classifier on transitions where the demonstrators moved from one option to another. We were also able to prevent the forgetting of behaviour related to solving sub-tasks further down the hierarchy structure. Since each low-level policy only trains on data pertaining to solving its sub-task, it does not forget behaviour. Lastly, our algorithm was unable to outperform Forgetful Experience Replay (ForgER), the 2019 MineRL winning submission.

8.1 Future Work

Option discovery. Our method of discovering options is limited to cases where we can extract the options from demonstration data. For future work we would recommend investigating alternative methods proposed by various other researchers that can be applied to a much wider variety of problems [44; 45; 46; 47; 48].

Option selection. Although we were able to learn a strategy for successfully selecting options, we did not consider the rewards received by the agent. In the ablation study we found that IL methods benefit greatly when combined with RL. Additionally, forcing the agent to select an option whenever the inventory contents change may lead to the incorrect option being selected. We would recommend investigating methods that account for uncertainty in prediction, for example Bayesian NNs [49].

Planning in low-level policies. In our HRL model, the low-level policies do not consider the long-term ramifications of their actions to other policies. A short sequence of actions early in an episode may determine the success of the episode. Therefore, we suggest researching ways of incorporating this knowledge in the training of low-level policies.

Recurrent neural networks. Previous work has shown recurrent neural networks (RNNs) to be promising [50; 51; 38]. They have been used to address exploration and partial observability—both issues we face in this work. We recommend investigating RNNs as described in the cited work.

Distributional RL. While the C51 algorithm is a beneficial addition to BRfD, there exists more recent work that outperforms C51, for example: quantile regression DQN (QR-DQN) [52], implicit quantile networks (IQN) [53] and fully parameterized quantile function (FQF) [54]. Both QR-DQN and IQN have been shown to outperform C51. FQF even managed to beat the Rainbow algorithm.

Exploration. Although the demonstration data somewhat negates the need for large scale exploration, we still believe that an effective strategy would benefit performance greatly. In their paper proposing Agent57 (the first agent capable of beating humans in all 57 Atari games), Bellemare *et al.* [5] implemented several techniques of exploring effectively. If those methods can be combined with learning from demonstration data, finding a diamond would likely be easier.

Bibliography

[1] Sutton, R.S. and Barto, A.G.: *Reinforcement Learning: An Introduction*. 2nd edn. The MIT Press, 2018.

[2] Cassandra, A.R., Kaelbling, L.P. and Littman, M.L.: Acting Optimally in Partially Observable Stochastic Domains. In: *AAAI*, vol. 94, pp. 1023–1028. 1994.

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D.: Human-level control through deep reinforcement learning. *Nature*, vol. 518, pp. 529–33, 02 2015.

[4] Wang, Z., de Freitas, N. and Lanctot, M.: Dueling Network Architectures for Deep Reinforcement Learning. *CoRR*, vol. abs/1511.06581, 2015. `1511.06581`.
Available at: `http://arxiv.org/abs/1511.06581`

[5] Bellemare, M.G., Dabney, W. and Munos, R.: A Distributional Perspective on Reinforcement Learning. *CoRR*, vol. abs/1707.06887, 2017. `1707.06887`.
Available at: `http://arxiv.org/abs/1707.06887`

[6] Tavakoli, A., Pardo, F. and Kormushev, P.: Action Branching Architectures for Deep Reinforcement Learning. *CoRR*, vol. abs/1711.08946, 2017. `1711.08946`.
Available at: `http://arxiv.org/abs/1711.08946`

[7] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M.G. and Silver, D.: Rainbow: Combining Improvements in Deep Reinforcement Learning. *CoRR*, vol. abs/1710.02298, 2017. `1710.02298`.
Available at: `http://arxiv.org/abs/1710.02298`

[8] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S. and Kavukcuoglu, K.: IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. *CoRR*, vol. abs/1802.01561, 2018. `1802.01561`.
Available at: `http://arxiv.org/abs/1802.01561`

[9] Amiranashvili, A., Dorka, N., Burgard, W., Koltun, V. and Brox, T.: Scaling Imitation Learning in Minecraft. 2020. `2007.02701`.

[10] Silver, D.: UCL Course on RL. https://www.davidsilver.uk/teaching/, 2015. (Accessed on 17/03/2020).

[11] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M.A.: Playing Atari with Deep Reinforcement Learning. *CoRR*, vol. abs/1312.5602, 2013. 1312.5602.
Available at: http://arxiv.org/abs/1312.5602

[12] Bellman, R.E.: *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. ISBN 9780691079516.
Available at: https://books.google.co.za/books?id=wdtoPwAACAAJ

[13] Albawi, S., Mohammed, T.A. and Al-Zawi, S.: Understanding of a Convolutional Neural Network. In: *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6. IEEE, 2017.

[14] Guss, W.H., Codel, C., Hofmann, K., Houghton, B., Kuno, N., Milani, S., Mohanty, S.P., Liebana, D.P., Salakhutdinov, R., Topin, N., Veloso, M. and Wang, P.: The MineRL Competition on Sample Efficient Reinforcement Learning using Human Priors. *CoRR*, vol. abs/1904.10079, 2019. 1904.10079.
Available at: http://arxiv.org/abs/1904.10079

[15] Johnson, M., Hofmann, K., Hutton, T. and Bignell, D.: The Malmö Platform for Artificial Intelligence Experimentation. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, p. 4246–4247. AAAI Press, 2016. ISBN 9781577357704.

[16] Guss, W.H., Houghton, B., Topin, N., Wang, P., Codel, C., Veloso, M. and Salakhutdinov, R.: MineRL: A large-scale dataset of Minecraft demonstrations. *arXiv preprint arXiv:1907.13440*, 2019.

[17] Milani, S., Topin, N., Houghton, B., Guss, W.H., Mohanty, S.P., Nakata, K., Vinyals, O. and Kuno, N.S.: Retrospective Analysis of the 2019 MineRL Competition on Sample Efficient Reinforcement Learning. In: *NeurIPS 2019 Competition and Demonstration Track*, pp. 203–214. PMLR, 2020.

[18] Hester, T., Vecerík, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Sendonaris, A., Dulac-Arnold, G., Osband, I., Agapiou, J., Leibo, J.Z. and Gruslys, A.: Deep Q-learning from Demonstrations. *CoRR*, vol. abs/1704.03732, 2017. 1704.03732.
Available at: http://arxiv.org/abs/1704.03732

[19] Nichol, A.: Competing in the Obstacle Tower Challenge – Pickled ML. https://blog.aqnichol.com/2019/07/24/competing-in-the-obstacle-tower-challenge/, July 2019. (Accessed on 24/09/2020).

[20] Skrynnik, A., Staroverov, A., Aitygulov, E., Aksenov, K., Davydov, V. and Panov, A.I.: Hierarchical Deep Q-Network from Imperfect Demonstrations in Minecraft. 2020. 1912.08664.

[21] Nagabandi, A., Kahn, G., Fearing, R.S. and Levine, S.: Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. *CoRR*, vol. abs/1708.02596, 2017. 1708.02596.
Available at: http://arxiv.org/abs/1708.02596

[22] Watkins, C.J.C.H.: *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Cambridge, UK, May 1989.
Available at: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[23] Sutton, R.S., Precup, D. and Singh, S.: Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[24] Ma, T., Avati, A., Katanforoosh, K. and Ng, A.: CS229 Deep Learning lecture notes. Oct 2020. (Accessed on 26/10/20).
Available at: http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf

[25] Ng, A.: Deep Learning Specialization. https://www.coursera.org/specializations/deep-learning, 2019. (Accessed on 01/02/19).

[26] Krizhevsky, A., Sutskever, I. and Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in neural information processing systems*, pp. 1097–1105. 2012.

[27] Touvron, H., Vedaldi, A., Douze, M. and Jégou, H.: Fixing the train-test resolution discrepancy: FixEfficientNet. *arXiv preprint arXiv:2003.08237*, 2020.

[28] van Hasselt, H., Guez, A. and Silver, D.: Deep Reinforcement Learning with Double Q-learning. *CoRR*, vol. abs/1509.06461, 2015. 1509.06461.
Available at: http://arxiv.org/abs/1509.06461

[29] Schaul, T., Quan, J., Antonoglou, I. and Silver, D.: Prioritized Experience Replay. In: Bengio, Y. and LeCun, Y. (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016.
Available at: http://arxiv.org/abs/1511.05952

[30] Fortunato, M., Azar, M.G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C. and Legg, S.: Noisy Networks for Exploration. *CoRR*, vol. abs/1706.10295, 2017. 1706.10295.
Available at: http://arxiv.org/abs/1706.10295

[31] Sammut, C.: *Behavioral Cloning*, pp. 93–97. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8.
Available at: https://doi.org/10.1007/978-0-387-30164-8_69

[32] Bogdanovic, M., Markovikj, D., Denil, M. and De Freitas, N.: Deep Apprenticeship Learning for Playing Video Games. In: *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*. Citeseer, 2015.

[33] Chen, Z. and Yi, D.: The Game Imitation: Deep Supervised Convolutional Networks for Quick Video Game AI. *CoRR*, vol. abs/1702.05663, 2017. 1702.05663.
Available at: http://arxiv.org/abs/1702.05663

[34] Justesen, N. and Risi, S.: Learning Macromanagement in StarCraft from Replays using Deep Learning. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 162–169. IEEE, 2017.

[35] Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P. *et al.*: Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[36] Zhang, H., Dauphin, Y.N. and Ma, T.: Fixup Initialization: Residual Learning Without Normalization. *CoRR*, vol. abs/1901.09321, 2019. 1901.09321.
Available at: http://arxiv.org/abs/1901.09321

[37] Skrynnik, A., Staroverov, A., Aitygulov, E., Aksenov, K., Davydov, V. and Panov, A.I.: Forgetful Experience Replay in Hierarchical Reinforcement Learning from Demonstrations. 2020. 2006.09939.
Available at: https://arxiv.org/abs/2006.09939

[38] Paine, T.L., Gulcehre, C., Shahriari, B., Denil, M., Hoffman, M., Soyer, H., Tanburn, R., Kapturowski, S., Rabinowitz, N., Williams, D., Barth-Maron, G., Wang, Z., de Freitas, N. and Team, W.: Making Efficient Use of Demonstrations to Solve Hard Exploration Problems. 2019. 1909.01387.

[39] Kang, B., Jie, Z. and Feng, J.: Policy Optimization with Demonstrations. In: *International Conference on Machine Learning*, pp. 2469–2478. 2018.

[40] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W.: OpenAI Gym. *CoRR*, vol. abs/1606.01540, 2016. 1606.01540.
Available at: http://arxiv.org/abs/1606.01540

[41] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

[42] Guss, W.H. and Houghton, B.: The MineRL Dataset — MineRL 0.3.0 documentation. `https://minerl.io/docs/tutorials/data_sampling.html`, 2020. (Accessed on 29/07/2020).

[43] Badia, A.P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z.D. and Blundell, C.: Agent57: Outperforming the Atari Human Benchmark. In: III, H.D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, vol. 119 of *Proceedings of Machine Learning Research*, pp. 507–517. PMLR, Virtual, 13–18 Jul 2020.
Available at: `http://proceedings.mlr.press/v119/badia20a.html`

[44] Ramesh, R., Tomar, M. and Ravindran, B.: Successor Options: An Option Discovery Framework for Reinforcement Learning. In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pp. 3304–3310. AAAI Press, 2019.

[45] Stolle, M. and Precup, D.: Learning Options in Reinforcement Learning. In: *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pp. 212–223. 2002.

[46] Stolle, M.: *Automated discovery of options in reinforcement learning*. Ph.D. thesis, McGill University, 2004.

[47] Bagaria, A. and Konidaris, G.: Option Discovery using Deep Skill Chaining. In: *International Conference on Learning Representations*. 2019.

[48] McGovern, A. and Barto, A.G.: Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In: *ICML*. 2001.

[49] Neal, R.M.: *Bayesian Learning for Neural Networks*, vol. 118. Springer Science & Business Media, 2012.

[50] Hausknecht, M.J. and Stone, P.: Deep Recurrent Q-Learning for Partially Observable MDPs. In: *AAAI Fall Symposia*. 2015.

[51] Kapturowski, S., Ostrovski, G., Quan, J., Munos, R. and Dabney, W.: Recurrent Experience Replay in Distributed Reinforcement Learning. In: *International conference on learning representations*. 2018.

[52] Dabney, W., Rowland, M., Bellemare, M.G. and Munos, R.: Distributional Reinforcement Learning With Quantile Regression. In: *AAAI*. 2018.

[53] Dabney, W., Ostrovski, G., Silver, D. and Munos, R.: Implicit Quantile Networks for Distributional Reinforcement Learning. In: *International Conference on Machine Learning*, pp. 1096–1105. 2018.

[54] Yang, D., Zhao, L., Lin, Z., Qin, T., Bian, J. and Liu, T.-Y.: Fully Parameterized Quantile Function for Distributional Reinforcement Learning. In: *Advances in Neural Information Processing Systems*, pp. 6193–6202. 2019.

Appendices

A | MineRL Information

Table A.1: MineRL Obtain environments inventory observation vector with value range limitations.

Item	Description
Equipped item damage	Damage received by item as a result of usage.
Equipped item max damage	Maximum amount of damage the item can sustain before breaking.
Equipped item type	Type of item currently equipped in main hand.
Coal	Can be used as fuel in a furnace or to craft torches.
Cobblestone	Collected from mining stone with a pick-axe. Used to craft stone pick-axes or furnaces.
Crafting table	Crafted item used to craft more complicated items.
Dirt	Collected from breaking block in environment. Useful for navigating difficult terrain.
Furnace	Crafted in crafting table. Used to smelt ores into ingots.
Iron axe	When equipped, it speeds up breaking items made from wood.
Iron ingot	Used to craft iron equipment.
Iron ore	Collected with a pick-axe made from stone or better. Can be smelted into an ingot.
Iron pick-axe	Pick-axe made from iron. Needed to mine diamond ore for diamonds.
Log	Collected from breaking log blocks. Can be crafted into planks.
Planks	Used to make various wooden items.
Sticks	Crafted from planks. Used to make pick-axes and torches.
Stone	Not needed to reach our goals, but can only be collected by an enchanted pick-axe.
Stone axe	When equipped, it speeds up breaking items made from wood.
Stone pick-axe	Pick-axe made from stone. Needed to mine iron.
Torch	Crafted from sticks and coal. Lights up the nearby area.
Wooden axe	When equipped, it speeds up breaking items made from wood.

Wooden pick-axe	Pick-axe made from wood. Needed to mine stone.

B | Network Value Estimations

To demonstrate the value of modelling the reward distribution, we show the output of a BRfD model given a sample input. The model was trained for 300 episodes in the Treechop environment. Figures B.1, B.2, B.3, B.4 and B.5 show a sample input with the saliency of the CNN, the value distribution and the expected value for a selected action. The more detailed distributions give a sense of risk to selecting a specific action. For example, in Figure B.1, the expected values for the attack action show little difference between performing the action or not. Inspecting the value distribution on the other hand, reveals that not attacking is likely to yield no return.

Figure B.3 shows that performing the forward action is not beneficial. However, Figure B.2 shows that the network learned that the sprint action yields more rewards—with the risk of running past potential rewards. Figures B.4 and B.5 provide insight to how catastrophic selecting the wrong camera action can be to the expected rewards.

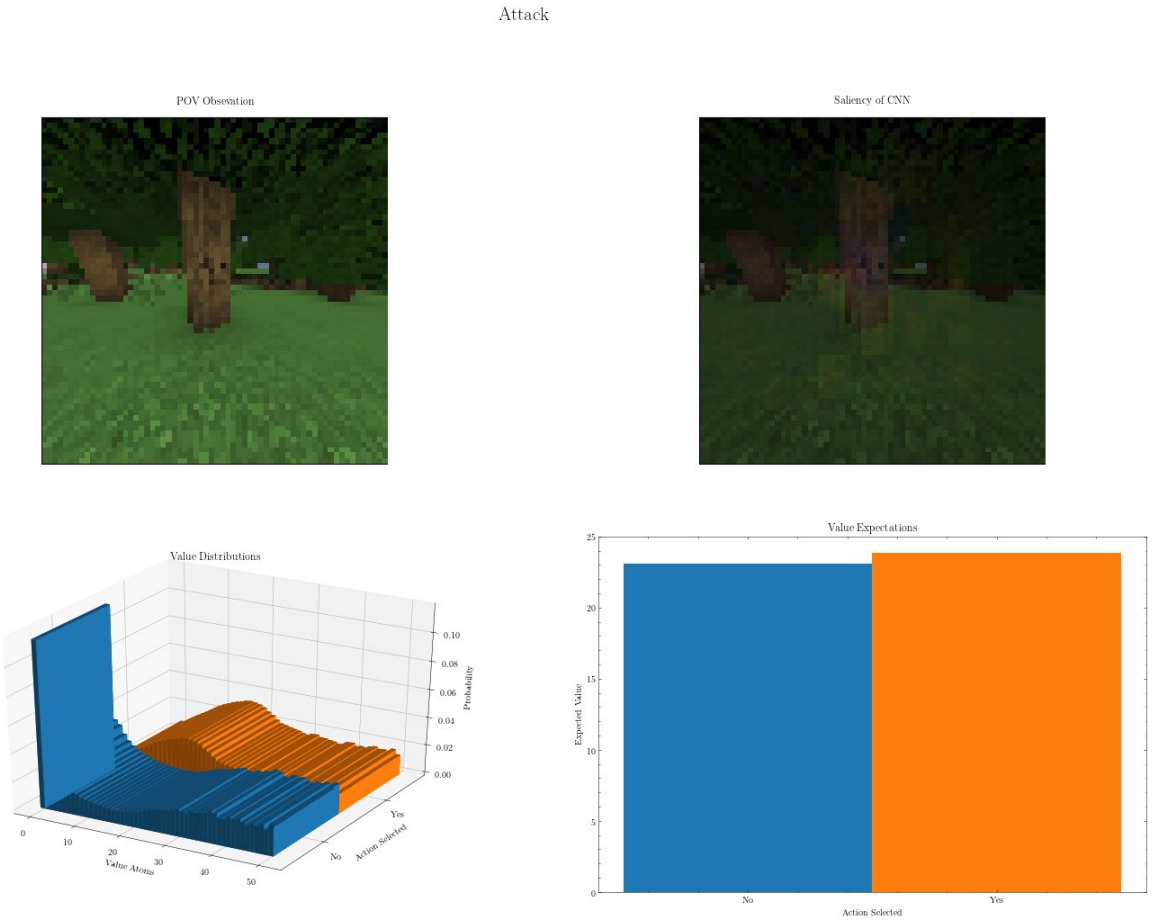

Figure B.1: Depiction of NN output for the attack action. The top left image is the POV observation seen by the agent. After a forward pass through the network, we propagate the gradients backward to obtain the saliency of the network in the top right image. The action-value distributions for each action is then shown in the bottom left image. Lastly, the bottom right image shows the expected value. Note the high likelihood of receiving zero reward if the attack action is not selected.

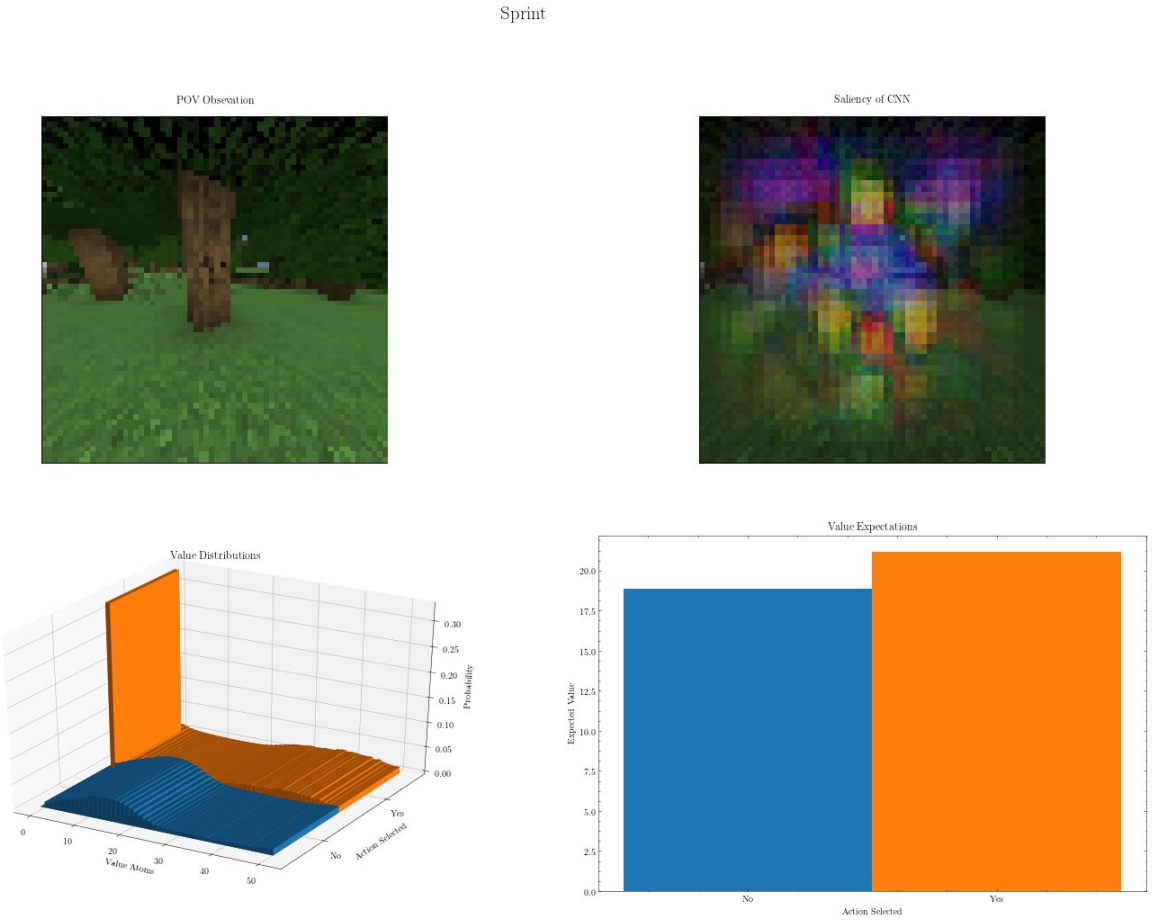

Figure B.2: Similar to Figure B.1, we show the results for the sprint action branch. Note the high probability of receiving a reward of 1 in the near future.

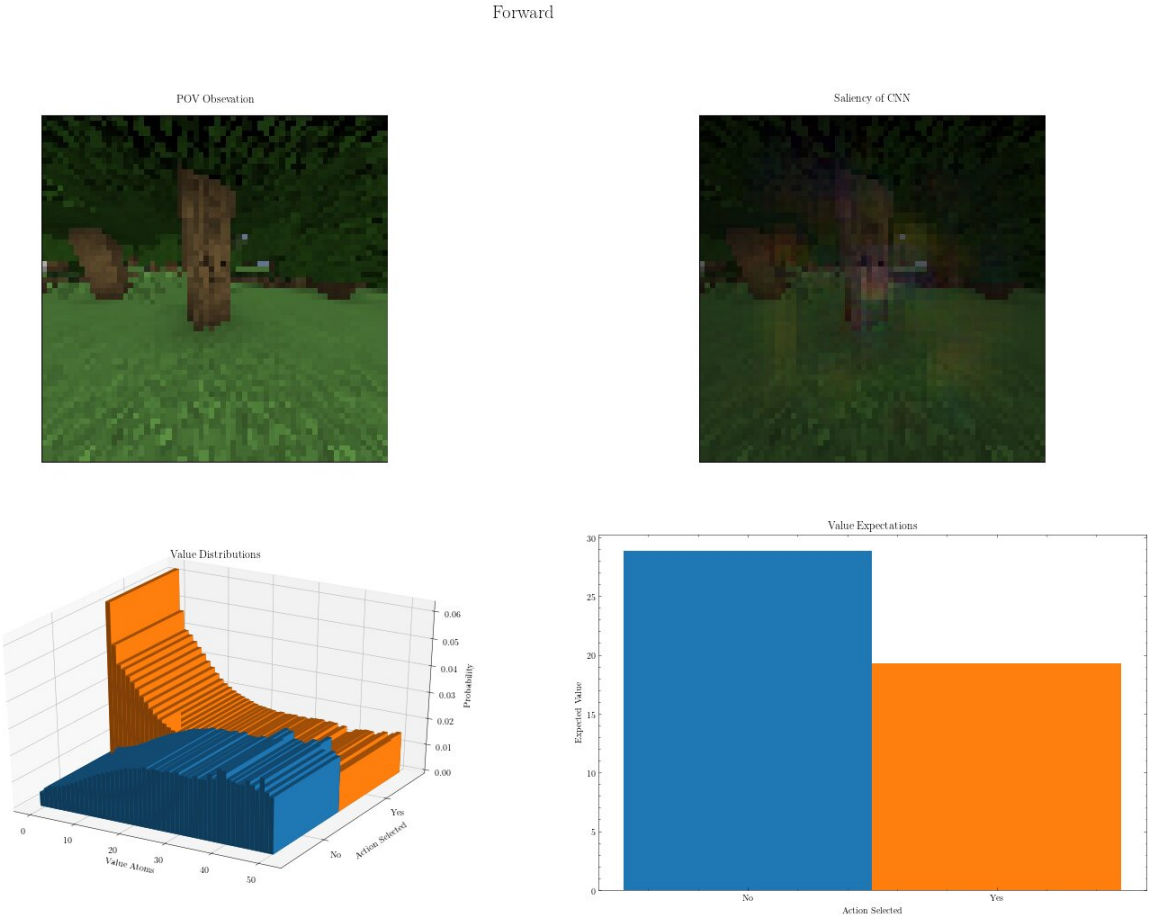

Figure B.3: Similar to Figure B.1, we show the results for the forward action branch. Note the high probability of the lower rewards when choosing the forward action as opposed to the sprint action in Figure B.2.

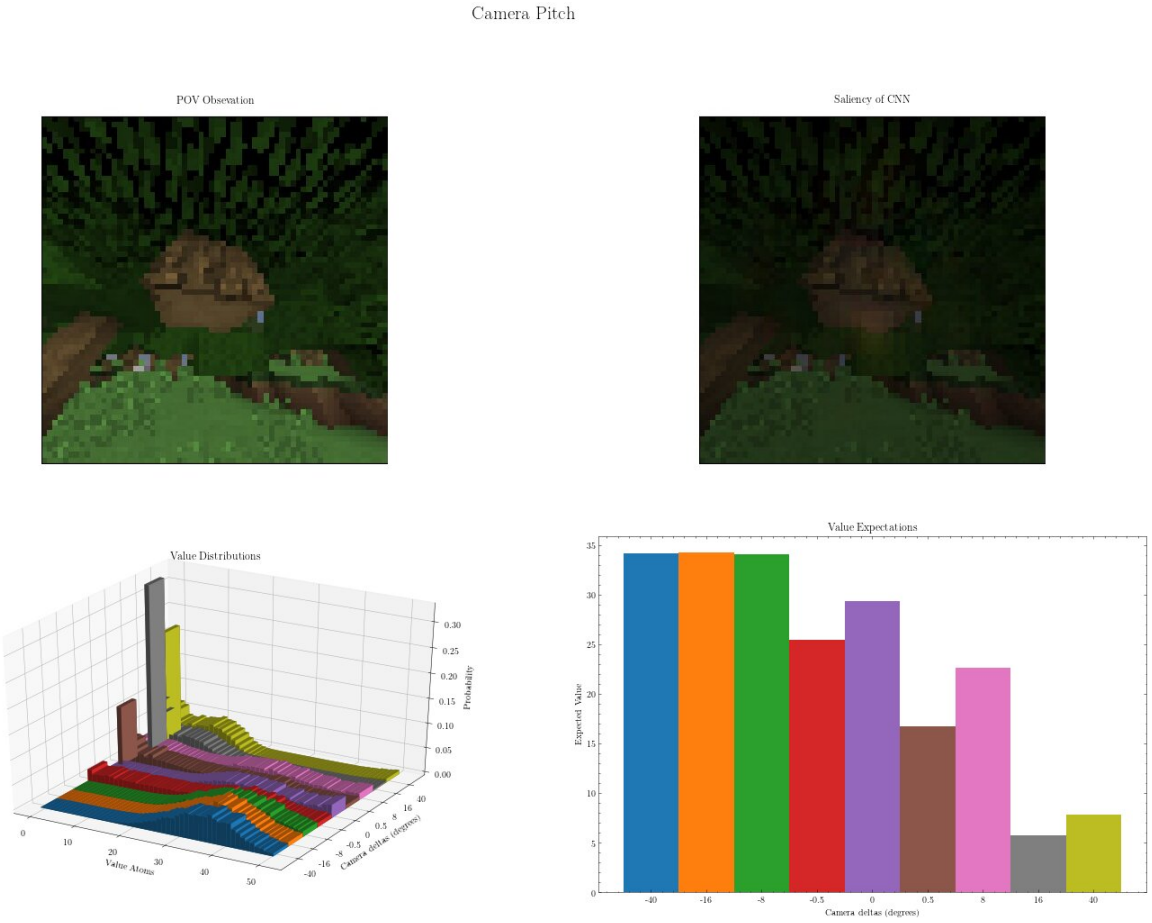

Figure B.4: Similar to Figure B.1, we show the results for the camera pitch delta action branch. In this case the actions represent the discretised camera action space defined in Section 5.3.2. Note the high probability of receiving no reward in the actions that would shift the agent's POV downwards.

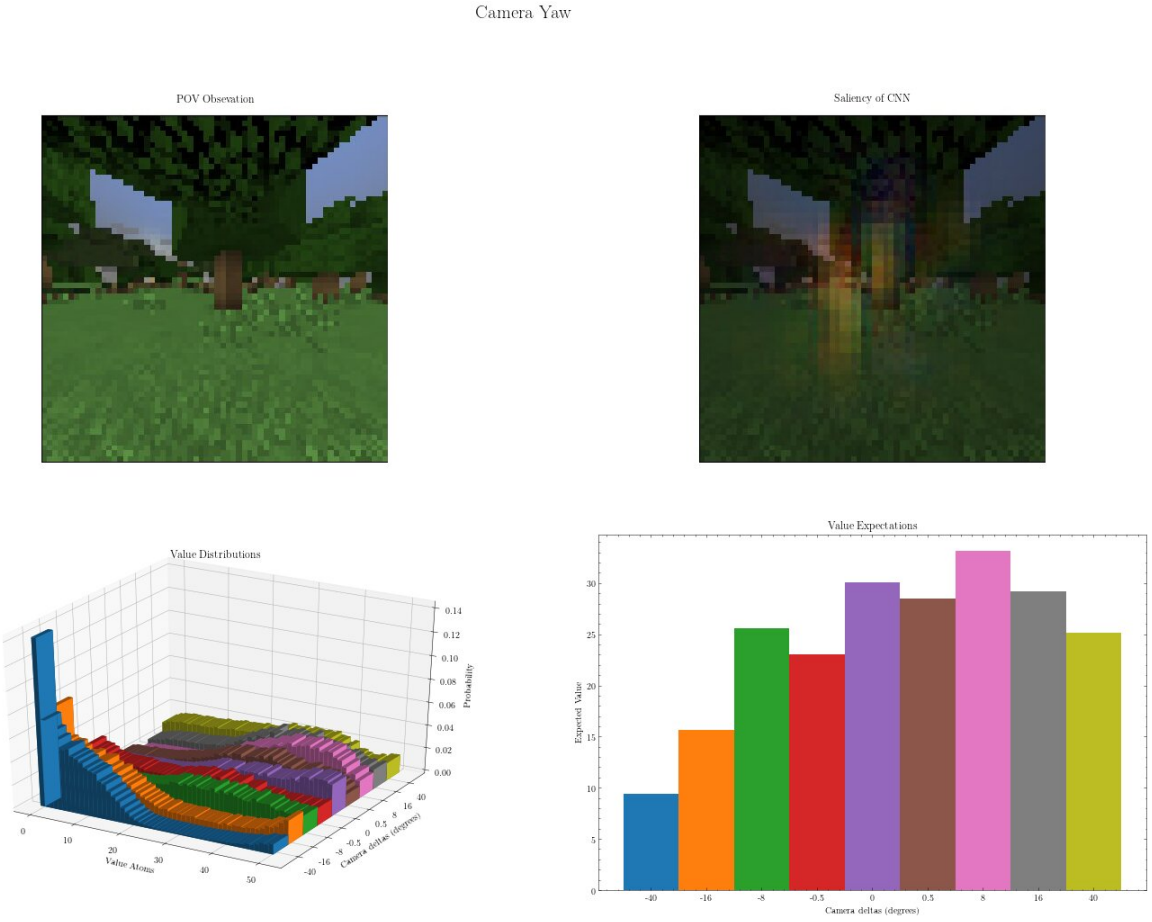

Figure B.5: Similar to Figure B.4, we show the results for the camera yaw delta action branch. Note the salient areas in the top right figure. Apart from the nearest tree, it seemed to focus slightly to the left as well.