Solving Sparse-reward Problems in Partially Observable 3D Environments using Distributed Reinforcement Learning

by

Jacobus Martin Louw

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

Supervisor:	Prof. HA Engelbrecht
Co-supervisor:	Mr. JC Schoeman

December 2021

Stellenbosch University https://scholar.sun.ac.za

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.

Copyright © 2021 Stellenbosch University
All rights reserved

Stellenbosch University https://scholar.sun.ac.za

Abstract

In this study, we address sparse-reward problems in partially observable 3D environments. The example task is set in a simulation environment where a *reinforcement learning* (RL) agent has to deliver a first-aid kit to an immobilised miner using an image observation. We apply a deep Q-learning algorithm with several modifications to solve this problem. We first show that it helps the agent to solve problems in the partially observable environment when the agent's observation is augmented with a history of previous observations and performed actions. We then consider three main modifications made to the deep Q-learning algorithm to address this problem. The first is to dramatically increase the rate at which new data is generated by using a distributed system. Secondly, we utilise *prioritised experience replay* (PER) [39] to repeat transitions of significance more frequently to the agent. Lastly, we add the n-step return to the algorithm. The work by Hessel *et al.* [14] and Horgan *et al.* [16] shows that these modifications significantly improve the performance of the deep Q-learning algorithm on the Atari platform. The Atari platform consists mainly of simple 2D environments; however, we consider performance on a partially observable 3D environment with sparse rewards.

We confirm the results of Fedus *et al.* [10] and show that better-performing policies are trained when the replay buffer contains more recently generated data. We show that prioritising transitions and the n-step return is very important in solving the example sparse-reward problem. In addition to these modifications we also look into strategies to improve exploration. We then demonstrate that *curriculum learning* (CL) or *domain randomisation* (DR) can be used to help the agent to solve more challenging problems where it is difficult to initially receive the reward signal. Lastly, we establish that it greatly benefits the deep Q-learning agent's performance when CL is used in combination with DR to solve larger, more complex problems.

Stellenbosch University https://scholar.sun.ac.za

Uittreksel

In hierdie studie spreek ons skaars-beloningsprobleme in gedeeltelik sigbare 3D-omgewings aan. In die probleem wat ons as voorbeeld gebruik, moet 'n versterkingsleeragent 'n noodhulpkissie aan 'n gestrande mynwerker in 'n simulasie-omgewing aflewer. Die agent moet aksies, gebaseer op 'n kamerabeeld, uitvoer om die taak te verrig. Ons pas 'n diep-Q-leer algoritme met 'n paar wysigings toe, om die probleem op te los. Ons toon eerstens aan dat dit die agent help om probleme in die gedeeltlik sigbare omgewing op te los, indien sy waarneming aangevul word deur vorige waarnemings en uitgevoerde aksies. Daarna oorweeg ons drie hoofsaaklike wysigings aan die diep-Q-leer algoritme om hierdie probleem op te los. Eerstens word die spoed waarteen nuwe data gegenereer word drasties verhoog deur van 'n verspreide stelsel gebruik te maak. Tweedens gebruik ons 'n geprioritiseerde ervaringsbuffer [39] om belangrike ervarings meer gereeld aan die agent terug te speel. Laastens voeg ons n-stap opdaterings by die algoritme. Die navorsing deur Hessel *et al.* [14] en Horgan *et al.* [16] toon aan dat hierdie wysigings die werksverrigting van die diep-Q-leer algoritme op die Atari-platform aansienlik verbeter. Die Atari-speletjies bestaan hoofsaaklik uit 2D-omgewings, terwyl ons die algoritme op 'n 3D-omgewing met skaars-belonings toepas.

Ons bevestig die resultate van Fedus *et al.* [10] en toon aan dat beter gedragspatrone aangeleer word indien die ervaringsbuffer meer onlangs gegenereerde data bevat. Ons toon ook dat die prioritisering van ervaring en n-stap opdaterings baie belangrik is om die skaars-beloningsprobleem in die voorbeeld op te los. Aanvullend tot hierdie wysigings, ondersoek ons ook strategieë om die verkenning van die omgewing te verbeter. Ons toon aan dat kurrikulumleer of domein-lukraakheid die agent kan help om meer uitdagende probleme op te los, waar dit aanvanklik moeilik is om 'n beloning te ontvang. Laastens wys ons dat dit die diep-Q-leer agent verder bevoordeel indien kurrikulumleer in kombinasie met domein-lukraakheid gebruik word om groter en moeiliker probleme op te los.

Stellenbosch University https://scholar.sun.ac.za

Acknowledgements

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

- My supervisor Prof. HA Engelbrecht and co-supervisor Mr JC Schoeman for leading this study, for the weekly meetings and time allocated to my thesis, as well for their valuable input. I would also like to thank Prof. HA Engelbrecht for equipping me with the necessary tools to conduct my research.
- Francois Rossouw for technical advice and support and his willingness to assist with problems I encountered.
- My family for their unwavering support, guidance, and encouragement throughout the duration of my project. Without them, this achievement would not have been possible.

Stellenbosch University https://scholar.sun.ac.za

Dedications

Hierdie tesis word aan my Hemelse Vader opgedra.

Stellenbosch University https://scholar.sun.ac.za

Contents

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of figures

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of tables

Stellenbosch University https://scholar.sun.ac.za

Nomenclature

Notation similar to the work by Sutton and Barto [45] is used. Capital letters indicate random variables, while lower case letters are used for values of random variables and also to indicate scalar functions. Bold lower case letters are used to indicate real-valued vectors.

General

x	a scalar x
$\mathbf{x}$	a vector $\mathbf{x}$
A	matrix A
$\mathrm{A}^\top$	transpose of matrix A
$\mathrm{A} * \mathrm{B}$	convolution of A and B
$\mathrm{A} \odot \mathrm{B}$	element-wise product of A and B
$\approx$	approximately equal
$\doteq$	equal true by definition
$Pr\{X = x\}$	probability that the value x is assigned to the random variable X
$X \sim p$	random variable X sampled from distribution $p(x) \doteq Pr\{X = x\}$
$\mathbb{E}[X]$	expectation of a random variable X, i.e. $\mathbb{E}[X] = \sum_x xp(x)$
$\mathrm{argmax}_a f(a)$	a value of a which maximises $f(a)$
$\mathbb{R}$	set of real numbers
ε	probability of taking an action at random when using ε-greedy exploration strategy
α	learning rate
γ	discount rate
$\frac{dy}{dx}$	derivative of y with respect to x
$\frac{\partial y}{\partial x}$	partial derivative of y with respect to x
$\nabla_{\mathbf{x}} y$	gradient of y with respect to x

Stellenbosch University https://scholar.sun.ac.za

Markov Decision Processes

a	an action
s	state
s'	subsequent state
o	observation
o'	subsequent observation
r	a reward
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{R}$	set of all possible rewards, a finite subset of $\mathbb{R}$
$\mathcal{A}$	set of all actions
$\mathcal{O}$	set of all observations
$\in$	is an element of
$\lvert\mathcal{A}\rvert$	the cardinality of $\mathcal{A}$
t	discrete time step
T	final time step of an episode
A_t	action at time t
S_t	state at time t
R_t	reward at time t
π	policy (decision-making rule)
$\pi(s)$	a deterministic policy π that returns an action for the state s
$\pi(a\lvert s)$	probability of taking action a in state s under stochastic policy π
G_t	return following time t
$G_{t:t+n}$	n-step return from $t+1$ to $t+n$ (discounted)
$p(s', r\lvert s, a)$	probability of transitions to state s' with reward r, from state s performing action a
$p(s'\lvert s, a)$	probability of transition to state s', from state s performing action a
$r(s, a, s')$	expected immediate reward on transition from s to s' under action a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy π
$q_\pi(s, a)$	value of performing action a in state s and then following policy π

Stellenbosch University https://scholar.sun.ac.za

$q_*(s, a)$	value of taking action a in state s and then following the optimal policy π_*
V, V_t	tabular estimates of state-value function v_π or v_*
Q, Q_t	tabular estimates of action-value function q_π or q_*
δ_t	*temporal-difference* (TD) error at t

Function Approximation

$f(\mathbf{x}; \boldsymbol{\theta})$	a function of $\mathbf{x}$ given parameters $\boldsymbol{\theta}$
$\mathbf{w}, \mathbf{w}_t$	vector of weights at t
$w_i, w_{t,i}$	ith component of weight vector
d	dimension of $\mathbf{w}$
$\hat{v}(s; \mathbf{w})$	approximate value of a state s given weight vector $\mathbf{w}$
$\hat{q}(s, a; \mathbf{w})$	approximate value of state-action pair s, a given weight vector $\mathbf{w}$

Artificial Neural Networks

$\boldsymbol{\theta}$	network parameters
$\mathrm{W}^{(i)}$	weight matrix for the ith layer of the network
$\mathbf{b}^{(i)}$	bias vector for the ith layer of the network
$\mathbf{h}^{(i)}()$	hidden layer for the ith layer of the network
$\mathbf{a}^{(i)}()$	pre-activation function for the ith layer of the network
$\mathbf{g}()$	activation function
$\mathcal{L}$	loss function
J	cost function
L	number of layers in the network (excluding the input layer)

Deep Q-learning

$\mathcal{D}$	replay memory
n	n-step return
p	priority
ζ	prioritisation
η	prioritised offset
ρ	importance-sampling weight
β	importance-sampling correction

Stellenbosch University https://scholar.sun.ac.za

Acronyms

Adam	adaptive moment estimation
ANN	artificial neural network
BC	behavioural cloning
CL	curriculum learning
CNN	convolutional neural network
CPU	central processing unit
DDQN	double deep Q-network
DL	distributed learning
DNN	deep neural network
DP	dynamic programming
DQfD	deep Q-learning from demonstrations
DQN	deep Q-network
DR	domain randomisation
DRL	deep reinforcement learning
ER	experience replay
FNN	feed-forward neural network
FIFO	first in first out
Gorila	general reinforcement learning architecture
GPI	generalised policy iteration
GPU	graphics processing unit
ICM	intrinsic curiosity module
IDE	integrated development environment
IDD	independent and identically distributed
IS	importance sampling
LSTM	long short-term memory
MAE	mean absolute error
MC	Monte Carlo
MDP	Markov decision process
MSE	mean square error
PER	prioritised experience replay
POMPD	partially observable Markov decision process
PPO	proximal policy optimisation
ReLU	rectified linear unit

Stellenbosch University https://scholar.sun.ac.za

RAM	random access memory
RF	radio frequency
RGB	red green blue
RL	reinforcement learning
RNN	recurrent neural network
SGD	stochastic gradient descent
TD	temporal difference
XML	extensible markup language

Stellenbosch University https://scholar.sun.ac.za

Chapter 1

Introduction

Drones and ground robotic vehicles are being used with great success to perform tasks in areas that are difficult or unsafe for humans to access. We can use drones for aerial photography which can be valuable for farmers to make crop yield estimates. We can also use robots in situations where bombs need to be disabled and keep people out of harm's way. These tasks can usually be performed by a robot that is controlled by a human with a *radio frequency* (RF) remote control. Although these robots have been successfully utilised, there are scenarios where RF signals cannot reach the robot. An example of such a problem is where trapped miners need to be rescued in a collapsed mine. Thick layers of ground and rock separate the miners from the rescue workers. Therefore RF signals are unlikely to be able to reach a robot inside the mine. The Wolverine V2 robot addressed this issue by using a fibre cable for communications. The fibre cable made it challenging to deploy the robot in certain situations, and therefore it was generally unsuccessful [1]. Accordingly, there is a need for robots capable of artificially intelligent decision-making in situations where the robot cannot be controlled by humans.

Supervised learning can be used to develop artificially intelligent decision-making systems by replicating expert demonstrations. Similar to traditional supervised learning that maps features to labels, supervised learning can be used to map situations to actions. Techniques that directly map situations to actions are often referred to as *behavioural cloning* (BC) [3]. According to Hussein *et al.* [17], a direct mapping between situations and actions is usually not appropriate to achieve the required behaviour. There are several reasons why this solution does not work in practice. For example, the demonstration data may be unreliable, the demonstrated task may be different from the intended task, or there may be insufficient demonstrations available [17]. *Imitation learning* addresses these issues and is closely linked to BC since it also mimics the behaviour of expert demonstrations [17]. In addition, it refines behaviour by performing learned actions and then optimising learned policies [17]. Imitation learning still requires good demonstration data, which may be expensive to generate, or may simply not be unavailable.

Reinforcement learning (RL) is a paradigm that is concerned with solving sequential

Stellenbosch University https://scholar.sun.ac.za

decision-making problems. RL systems learn by trial and error from their own experience, therefore expert demonstration data is not required. This enables RL algorithms to exceed expert performance and operate in areas where there is a lack of human expertise [42]. Recently, *deep reinforcement learning* (DRL), a combination of RL and *deep neural network*s (DNNs), has been used to make progress towards the goal of exceeding human capabilities in certain areas [42]. In 2015 DeepMind used DRL, along with expert data, to create the algorithm *AlphaGo*. AlphaGo was the first algorithm to be able to beat the world champion in the *Go* board game. DeepMind continued to develop AlphaGo Zero in 2017. It solely relied on DRL and was able to win the previously published champion-defeating AlphaGo 100-0 [42]. This was a monumental achievement, as for decades, the best Go computer programs were just as good as amateur human players. The idea of RL was already established in the mid-1900s. With the achievement of AlphaGo, RL again became a subject of real interest.

Although RL research has had a great deal of success recently, the field continues to grow, and many challenges remain to be overcome. In this study, we aim to apply RL algorithms to solve problems in 3D simulation environments. This is challenging since these environments are usually observed using a first-person camera. For this reason some important aspects of the environment may be obstructed or out of the camera's view.

An objective of our research is also to specify long-term goals to RL algorithms. These goals require long sequences of correct actions to receive credit. This introduces the *sparse-reward problem*, which is central to this study. We first discuss the most important literature relevant to this study. We then define our aims and objectives.

1.1 Reinforcement learning

Here we give an overview of RL based on the work of Sutton and Barto [45]. RL tries to solve a sequential decision-making problem, which is usually modelled as a *Markov decision process* (MDP). We discuss MDPs in more detail in Chapter 3. RL is concerned with how to choose actions from situations to maximise some notion of a numerical reward signal. The reward signal is a measure of how good the performed action or sequence of actions was, and it is used to guide software agents to learn optimal behaviour in a given situation. In RL, the idea of trial and error is central, since the agent is not instructed which actions to take and it has no prior knowledge of the task at hand. The agent aims to discover the consequences of actions through experimentation. Sometimes an action may not only have an impact on the agent's current situation but may also have an impact on the subsequent situations that the agent encounters. Thus, the aspect of delayed reward signals is also an important concept, as the outcome of specific actions can be experienced at a later time.

We present the general structure of the agent-environment interaction in an MDP in

Stellenbosch University https://scholar.sun.ac.za

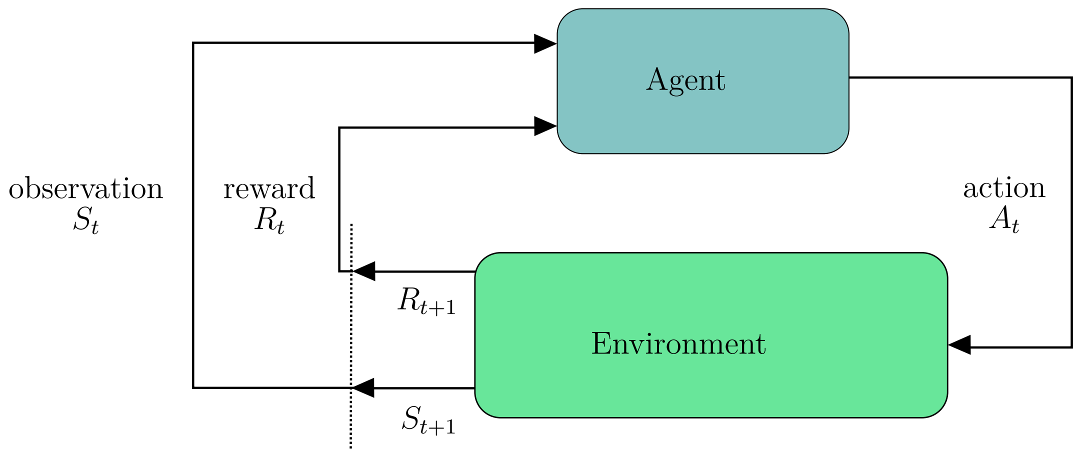

Figure 1.1: An illustration of the interaction between the agent and the environment in a *Markov decision process* (MDP), adapted from the work of Sutton and Barto [45]. At time step t, the agent receives an observation S_t and a reward R_t from the environment. The agent then performs an action A_t in the environment. The environment returns an observation S_{t+1} and reward R_{t+1} at time step $t + 1$.

Figure 1.1. The environment is the world which is assigned to the agent. It can be set in the physical world, or it can be in the form of a computer simulator. The latter is very popular to use for testing RL algorithms as simulations can be run at accelerated speeds [45]. The environment characterises the task at hand by defining the actions available, the outcome of an action taken in a given situation, and the reward signal function. The current environmental situation is known as the state of the environment.

The agent begins by receiving the initial state of the environment. The agent reacts to the state by sending an action to the environment; in return, the agent receives a new state and reward. The action of the agent is based on the observation received and chosen according to the agent's policy, where the policy determines the behaviour of the agent. The process of acting and receiving observations and rewards in return is recurrent. At each iteration of this process, the agent adjusts its policy in order to receive more rewards.

1.2 Partially observable environments

In this study, we deal with environments that are partially observable, similar to many real-world problems. This means that the environment's state is partially observed at any given time, which makes it very difficult to know what the true state of the environment is. Many observations may look similar even if they represent different states and therefore need to be treated differently. The problem addressed in this study is classified as a *partially observable Markov decision process* (POMPD). Monahan [25] states that the presence of uncertainty and the impact it has on policy optimisation is a key feature when

Stellenbosch University https://scholar.sun.ac.za

dealing with a POMPD. It is therefore necessary to know what the true underlying state of the environment is in order to be able to select the best possible action. It is known to be computationally difficult to estimate the true underlying state as this cannot be accessed directly [9]. We discuss the POMPD and possible ways to address it in Section 3.8.

1.3 Sparse-reward problems

In addition to the partially observable nature of the environment, the problem addressed in this study is classified as a sparse-reward problem or a difficult exploration problem. When an agent encounters an MDP for the first time, the environment is explored with some exploration strategy, which usually involves performing random actions. In some cases, non-zero reward signals are scarce, and the agent has to perform numerous correct actions sequentially to receive a reward. As the required number of correct actions to be rewarded increases, the probability of taking such a sequence of actions decreases exponentially. Sometimes rewards are so sparse that the number of environmental steps of random exploration needed to receive a non-zero reward becomes computationally intractable. Sparse-reward problems have generally proved difficult to solve, even for a number of state-of-the-art RL methods [31].

Furthermore, the problem of sparse and delayed rewards also leads to the credit-assignment problem – the process of distributing credit among the many decisions that may have contributed to success [45]. RL generally tries to solve the credit-assignment problem, but it is more problematic when rewards are very delayed.

1.4 Distributed systems

When dealing with major computational problems, one often needs a very powerful supercomputer to perform all the calculations within a reasonable time. However, such a computer is not always accessible. Most computers have *central processing unit*s (CPUs) with multiple cores that are capable of performing operations in parallel. Unfortunately most of the time these cores are not fully utilised.

If the task is of such a nature that the result of the previous step is not needed to perform the next step, then a distributed system can be used to perform it. Attiya and Welch [2] define a *distributed system* as a collection of separate computing devices that exchange information with each other. The goal of distributed systems is to utilise multiple processors to perform one large task. Distributed computing utilises more computational resources, but performance is accelerated.

DRL algorithms generally rely on generating data by themselves and usually require a large amount of data to learn optimal policies. For example Mnih *et al.* [24] trained

Stellenbosch University https://scholar.sun.ac.za

their DRL agent, the *deep Q-network* (DQN), for 50 million environment frames in total. This is more or less 38 days of real-time experience [24]. Furthermore, in problems where rewards are very sparse, many trajectories must be generated to reliably have transitions that contain non-zero rewards. Using a distributed system can accelerate the rate at which new data is generated.

1.5 Aims and objectives

Having discussed the most important literature relevant to this study, we formulate our aims and objectives. We aim to develop an artificially intelligent agent to solve a problem in a partially observable 3D environment where rewards are sparse. The example task we use as a test-bed is a collapsed mine where a first-aid kit needs to be delivered to an immobilised miner. We develop this example problem in a simulation environment as shown in Figure 1.2.

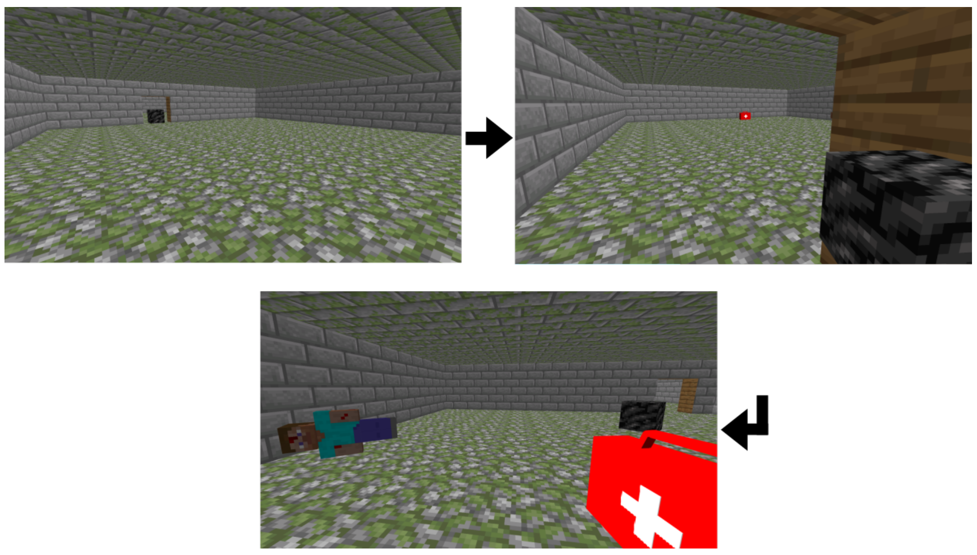

Figure 1.2: The robot's perspective of the example sparse-reward problem. The robot, miner and first-aid kit are located in random rooms in a mine. Obstacles are obstructing the entrances to the rooms. The robot therefore has to remove the obstacles to navigate the mine. The end goal is to deliver the first-aid kit to the miner.

The goal of the robot or agent is to deliver a first-aid kit to the miner. It is challenging to handcraft an agent that can deal with all the obstacles it may encounter while completing such a task. Therefore, the agent has to learn by itself to achieve this objective. Multiple sub-tasks must be completed in order to achieve the main objective. The sub-tasks entail navigating the environment, dealing with obstacles that may obstruct the agent's way, obtaining the first-aid kit, and finally delivering it to the miner. The only time the agent receives credit in the form of a positive reward signal is when the miner receives the first-aid kit. Therefore the entire policy required to complete this objective must be deduced from this single reward. The example problem is therefore a sparse-reward problem, which

Stellenbosch University https://scholar.sun.ac.za

makes it very difficult to obtain an agent with optimal behaviour, as we explained in Section 1.3.

To reach the objective of this study, we have to overcome numerous challenges. Some of these challenges include interpreting the *red green blue* (RGB) camera observation, and dealing with the partial observability of the environment. Our main objective is to address solving sparse-reward problems, as long term goals are a common occurrence in most real-world problems. We investigate several modifications to improve the performance of the deep Q-learning algorithm in solving the proposed problem. We also investigate how the exploration strategy of the agent can be improved to encounter rewards more frequently when they are very sparse. Lastly we investigate how well our approach scales to larger environments.

1.6 Methodology

We implemented a distributed version of the deep Q-learning algorithm with *prioritised experience replay* (PER) [16, 39, 23, 24]. Q-learning is an off-policy algorithm and allows the agent to learn from experience other than that generated by its latest policy. For this reason, Q-learning enabled us to use *experience replay* (ER) which breaks the correlations in the data. It also makes the algorithm more sample-efficient, since previously generated experience can be reused to improve the agent's value function. The ability to reuse old experience is important when dealing with sparse rewards. The reasoning behind this is that non-zero rewards are very rarely generated, and therefore the ability to re-sample important transitions is crucial. Since Q-learning is an off-policy algorithm it naturally accommodates the use of many alternative data sources. It is therefore a good choice for solving sparse-reward problems where it is difficult to generate good trajectories.

We added three modifications to the deep Q-learning algorithm. It has been shown by Fedus *et al.* [10] that the deep Q-learning algorithm performs better when training on more recent data. It can take a long time to generate the required amount of data if this is generated sequentially. Therefore, we distribute the data generation to significantly increase the rate at which data is generated. We also added a modification to the deep Q-learning algorithm which is referred to as *prioritised experience replay* (PER) [39]. This technique prioritises important transitions and replays these transitions more frequently to the agent in order to learn more efficiently. Prioritisation is very important since good trajectories are very scarce if rewards are sparse and the environment is explored using random actions. Lastly we added the n-step return, since Sutton and Barto [45] state that it can significantly improve TD learning. We also address exploration by using two techniques, namely *curriculum learning* (CL) and *domain randomisation* (DR). These methods are respectively discussed in Section 2.5.2 and Section 2.5.3.

Stellenbosch University https://scholar.sun.ac.za

1.7 Document outline

Chapter 2 constitutes a literature review. The deep Q-learning algorithm by Mnih *et al.* [23, 24] is considered as a breakthrough in *deep reinforcement learning* (DRL). Therefore, we review the achievements of Mnih *et al.* [23, 24] on the Atari platform. Since solving sparse-reward problems are central to this study, we investigate how similar problems were previously approached. Lastly, we review previously published distributed DRL algorithms.

In Chapter 3 we discuss *Markov decision process*es (MDPs) – a popular way of modelling sequential decision-making problems. The MDP describes the problem RL addresses and therefore it is important to review. We discuss essential concepts of the MDP, such as policies and value functions. We also review planning methods to obtain optimal behaviour in MDPs.

Having discussed MDPs, we review *reinforcement learning* (RL) in Chapter 4. We review the Q-learning algorithm – a very popular RL algorithm that is central to the deep Q-learning algorithm. We also discuss how function approximation can be used to solve problems with large state spaces and action spaces.

In Chapter 5, we review *artificial neural network*s (ANNs). We describe *deep neural network*s (DNNs), how they evaluate inputs and how the parameters of a DNN are adjusted to make more accurate predictions. The chapter ends with a discussion of the *convolutional neural network* (CNN), which is a special type of ANN that is good at extracting features from images.

In Chapter 6, we review the deep Q-learning algorithm – a Q-learning algorithm that utilises a DNN to approximate the value function. We review the main contributions made by Mnih *et al.* [24, 23] to ensure the stability of the deep Q-learning algorithm. We also discuss some improvements made to the deep Q-learning algorithm. These improvements include the *double deep Q-network* (DDQN) [12], the n-step update, dueling network architecture [50] and *prioritised experience replay* (PER) [39].

In Chapter 7, we discuss the simulation environment to perform experiments in. We consider two 3D environments, namely Minecraft and Miniworld [6]. We compare the functionality and performance of both environments in order to decide on a test-bed.

In Chapter 8 we discuss our implementation of the deep Q-learning algorithm. We discuss the functionality of the different components of the system and how they interact with each other. We then discuss how we address the problem of partial observability by using frame-stacking and action memory. We also discuss the architecture of the DNN used for the deep Q-learning algorithm. Lastly we test how well the different solutions perform to address partial observability.

Having discussed the structure of the system, we perform several experiments to test the deep Q-learning algorithm in Chapter 9. We first perform an ablation study to observe how each modification helps to solve a sparse-reward problem. Next, we further investigate the impact on performance of each modification. We then introduce a problem where it

Stellenbosch University https://scholar.sun.ac.za

is challenging to explore the environment and investigate how *curriculum learning* (CL) and *domain randomisation* (DR) help to address this issue. Lastly, we test how well the algorithm scales to problems in larger environments and combines CL and DR to solve these problems.

In Chapter 10, we summarise the objectives and goals of this study. We then review the different aspects the study addressed and how well the solutions worked. The thesis ends with potential future avenues for research.

We include links to videos of the resultant agents of our research in Appendix A. The videos illustrate the performance of the agent on the different problems we tested and show how behaviour differs when using different methods of training the agent. We additionally tested the algorithm on a snake environment and also include a video to illustrate this result. In Appendix B we show some additional results concerning the hyperparameters used for the experimental phase of our research. Finally, we include more detailed algorithms of the distributed deep Q-learning agent we implemented in Appendix C.

Stellenbosch University https://scholar.sun.ac.za

Chapter 2

Related work

In this chapter we perform a literature study to review how *deep reinforcement learning* (DRL) was previously applied to similar problems. We first review the deep Q-learning algorithm, which is considered to be a breakthrough in DRL. The main goal of our research is to solve sparse-reward problems with a DRL algorithm. We therefore focus on approaches used to address difficult exploration problems. These approaches include using expert-demonstration data to pre-train policies and utilising alternative exploration strategies to more frequently encounter rewards. We also review distributed DRL algorithms. At the end of this chapter we summarise the literature review.

2.1 Applying deep reinforcement learning to the Atari domain

This study is largely inspired by the work of Mnih *et al.* [24] where a DRL algorithm was applied to obtain a computer program that is capable of human-level control on the Atari 2600 platform. In this section we give an overview of the achievements by Mnih *et al.* [24]. In Chapter 6 we further explain how the deep Q-learning algorithm functions.

In simple sequential decision-making problems where the environment has a finite state space, a tabular RL method can easily be applied to obtain an optimal policy for the problem. Tabular methods entail utilising a table to represent the policy or value function of the agent. In most interesting problems, however, the state space of the environment is high-dimensional and large. Therefore it is a big challenge to apply RL algorithms to real-world problems, as it is difficult to find an efficient method to represent the policy or value function of the agent. For example, applying an RL algorithm to the Atari 2600 platform is very challenging since the observations are in the form of high-dimensional *red green blue* (RGB) images.

According to Sutton and Barto [45], striking results have been obtained when combining RL with backpropagation. Mnih *et al.* [24] from Google DeepMind developed a remarkable

Stellenbosch University https://scholar.sun.ac.za

algorithm that utilised a *deep neural network* (DNN) to automatically extract features from a high-dimensional observation. The agent that Mnih *et al.* [24] developed is called the *deep Q-network* (DQN). The algorithm combines Q-learning with a deep *convolutional neural network* (CNN). The CNN processes the high-dimensional 2D spatial array observations that are in the form of images. Additionally, Mnih *et al.* [23, 24] used a technique referred to as *frame-stacking* to address the partial observability of the Atari games. It entails stacking the last number of frames shown to the agent and holds information such as the direction and speed of moving objects.

The deep Q-learning algorithm was applied to the Atari platform and great results were obtained. The results show that a single RL algorithm can obtain top-level performance in various problems without using features customised to the specific domain. Mnih *et al.* [23, 24] trained the DQN agent to play 49 of the Atari 2600 video games. The agent learned optimal behaviour by only interacting with the game environment. The exact same algorithm, network architecture and hyperparameters were applied to all the different games. The DQN agent outperformed the best RL methods of the time in 43 of the 49 games without incorporating additional information about the games like previous approaches did. The DQN agent also achieved human-level or better performance in 29 of the games. The work by Mnih *et al.* [24] demonstrates that a single algorithm and architecture can learn good behaviour to solve numerous different problems with almost no information of the task at hand. The agent learns by only receiving the pixels as observation and the game score as a reward signal. Our aim was to also develop an algorithm capable of making decisions from image observations. Accordingly the problem we address is very similar to the problem Mnih *et al.* [23, 24] solved.

2.2 Deep Q-learning from demonstrations

Hester *et al.* [15] developed an algorithm referred to as *deep Q-learning from demonstrations* (DQfD) that utilises expert human demonstrations to assist in training the policy of a DQN agent. The main goal of DQfD is to accelerate learning by utilising demonstration data. Many RL algorithms perform random actions to explore the environment. This technique becomes very unreliable when rewards are very sparse. Therefore alternative smarter exploration strategies have to be used. Hester *et al.* [15] used demonstration data instead of smarter exploration. At the time the DQfD algorithm was published, this approach achieved a high score on the Atari game, *Pitfall*, which has very sparse positive rewards. No approach before DQfD has achieved any positive rewards on this game. We, therefore, considered to apply DQfD to deal with the challenging exploration problem of this study. Recall that the objective of our research is to solve sparse-reward problems in 3D environments.

DQfD first pre-trains the network of the agent by only using the demonstration data

Stellenbosch University https://scholar.sun.ac.za

with a combination of *temporal-difference* (TD) and supervised losses. The supervised loss enables the agent to imitate the demonstration data, and the TD loss allows the agent to learn a value function. After the pre-training is complete, the agent starts to interact with the environment and continues learning the value function using RL. The agent uses a mixture of demonstration data and experiences generated in the environment to update its network. According to Hester *et al.* [15], the ratio between expert demonstration data and self-generated data is essential to improve the performance of the algorithm. Hester *et al.* [15] used a technique referred to as *prioritised experience replay* (PER) [39] to control this ratio. According to Hester *et al.* [15] DQfD outperformed the *double deep Q-network* (DDQN) using the dueling architecture and PER (discussed in Chapter 6) in 41 of the 42 games tested. Hester *et al.* [15] also state that DQfD outperformed the best expert demonstration in 14 of the 42 games tested. This result shows that DQfD is able to surpass the performance of the expert demonstrations used to train it.

The work by Hester *et al.* [15] proved that using demonstration data in combination with deep Q-learning allows for better initial performance in the Atari domain. RL can then further be applied to improve performance, and in some cases to exceed the performance of expert demonstrations. Using demonstrations can also help to solve our problem where we deal with sparse rewards. The problem is that demonstration data is expensive to generate, and it is, therefore, more appealing to have an agent that is independent of demonstration data. According to Hester *et al.* [15] learning from demonstration data can be very difficult. For example, a human may solve a problem in a way that differs significantly from the policy that an agent would learn. Furthermore, humans may use certain information to solve the problem that is not available in the agent's state representation. Therefore the policy contained in the demonstration data may not be valid for the agent given the agent's state representation. For this reason, we decided against using demonstration data.

2.3 Curiosity-driven exploration by predicting sequential states

The work by Pathak *et al.* [33] addressed infrequent, sparse rewards by introducing a system referred to as the *intrinsic curiosity module* (ICM). The ICM generates intrinsic rewards to encourage the agent to perform actions when it has little knowledge of what the consequences will be. The intrinsic reward is based on how well the agent can predict the consequences of its actions, i.e. how well the agent can predict the next state given its current state and the action it has performed.

Instead of predicting in the raw observation space, Pathak *et al.* [33] extracted a feature space from the input observation that contains only information that is relevant to the agent and the action it has performed. They achieved this feature space by training an ANN to predict the agent's action given its current state and next state. The idea was that

Stellenbosch University https://scholar.sun.ac.za

the ANN would only extract information that is related to the agent or the action taken by the agent. Accordingly, the ANN would ignore other environmental developments that the agent was not responsible for, or that did not affect the agent itself. An additional model was trained to predict the feature representation of a next state given the action and feature representation of the current state. The prediction error could then be given as an intrinsic reward to the agent. In this way, the intrinsic reward was used to encourage the agent's curiosity. This is a very interesting approach for an alternative exploration strategy. However, this approach introduced an additional sub-system to the DRL algorithm, which can complicate the process of optimising the agent.

2.4 Go-Explore

One difficulty of sparse-reward problems is to obtain trajectories in the environment containing positive rewards. The previously discussed approaches utilised expert demonstration data or intrinsic motivation to assist in learning a policy to solve the problem. An algorithm called *Go-Explore*, developed by Ecoffet *et al.* [8], addressed challenging exploration problems without using expert demonstrations or intrinsic motivation.

Intrinsic motivation suffers from what Ecoffet *et al.* [8] call *detachment*. They theorise that agents with intrinsic motivation do a poor job in continuing to explore promising areas in their environment. For example, the agent may start to explore a promising area, but does not finish exploring the area entirely. The agent may by chance explore another equally promising area, but then fail to finish exploration of the first area, which it has detached from.

The Go-Explore algorithm consists of two parts that Ecoffet *et al.* [8] refer to as phases. The first phase is referred to as *phase one: explore until solved* and serves as a method to address challenging exploration. The solution by Ecoffet *et al.* [8] maintains a memory of previously visited novel states. During the first phase, the agent samples previously visited states, but is biased towards newer states that have hardly been visited. The environment is set to the chosen state, and the agent continues to explore from that state. The idea is that the agent will sample a state near unexplored states, and will eventually stumble on novel states. The sequence of actions to obtain a given state from the agent's initial state is stored alongside the state in memory. If a better trajectory to a stored state is obtained, then the state is updated with the new trajectory. The likelihood of the state being chosen is also reset. The agent repeats this process until it obtains a successful trajectory, i.e. a trajectory that completes the task.

Once one or more high-performing trajectories are obtained, the algorithm starts with phase two. This is referred to as *phase two: robustify.* As the first phase utilised determinism in the simulator, the trajectories will not be robust to stochastic versions of the problem. In this phase the algorithm runs imitation learning on the trajectories

Stellenbosch University https://scholar.sun.ac.za

obtained in the first phase. The reason for this is to obtain an agent with a more robust policy that reliably solves the problem. Phase two of Go-Explore was inspired by the work of Salimans and Chen [38]. Their algorithm entails starting the agent near the last state in the trajectory and then training the policy of the agent by using an RL algorithm. In this case Go-Explore used *proximal policy optimisation* (PPO) to train the agent. Once the agent has learned to obtain a score similar to or higher than that of the example trajectory, the agent's starting state is shifted back to an earlier state. The process repeats until the agent can solve the problem from the initial state.

The first phase of Go-Explore may be problematic to implement in other domains than the 2D Atari games it was originally tested on. The reason for this is, to maintain a memory of previously visited states, the algorithm represents a state as a downsampled 8×11 grayscale image. The technique of downsampling states may not be able to differentiate novel states in all other domains. The first phase also requires a deterministic environment in order to precisely set the environment to states previously visited. This should not be a problem, as most RL algorithms are trained in simulation environments. It should be noted that work in the Atari domain prior to Go-Explore rarely exploited simulators in such a manner.

2.5 Shaping

According to Sutton and Barto [45], the difficulty of solving sparse-reward problems is not only due to the scarcity of the reward signal, but also because the policy of the agent is inadequate to frequently visit rewarding states. The technique of *shaping* was introduced by the psychologist Skinner [43], and has been borrowed for the RL paradigm. Sutton and Barto [45] state that this involves giving the RL agent a sequence of fairly simple problems that will ultimately lead to a difficult issue of interest. The reward signal changes during the process of training the agent. Therefore, the agent first receives a non-sparse reward signal and the reward signal is gradually modified to suit the target problem. The RL agent encounters problems with increasing difficulty. The knowledge of previous problems it encountered enables the agent to solve the more difficult problems. Shaping usually requires domain knowledge of the target problem and therefore the RL algorithm does not learn optimal behaviour entirely by itself. We now discuss some of the primary methods of shaping used in RL.

2.5.1 Reward function shaping

A simple solution to the sparse-reward problem is to reward the agent more frequently. The method of augmenting the native reward function of the MDP with additional rewards is called *reward shaping*. It can be very effective to accelerate learning [29]. The idea of reward shaping is to reward the agent for making approximations to the desired behaviour.

Stellenbosch University https://scholar.sun.ac.za

For example, to encourage the agent to navigate to a goal, a positive reward can be returned when the agent moves closer to the goal. Therefore the agent is guided to the solution of the problem. The benefit is that the agent does not have to solve a problem with significantly delayed rewards. However, the shaped reward function must be designed very carefully to train an optimal policy. This method can restrict the behaviour of the agent to act on the basis of the shaped reward function, and the agent is not truly free to find innovative solutions on its own.

The work by Randløv and Alstrøm [35] also showed that agents may enter cycles to optimise the shaped reward function that are not aligned with the optimal policy for the target problem. According to Sutton and Barto [45], it is not advisable to reward RL agents for achieving sub-goals. The agent may learn a strategy to obtain a lot of rewards without solving the intended task, i.e. this may lead to sub-optimal behaviour.

Designing an optimal shaped reward function can be time-consuming, as it requires fully understanding the dynamics of the environment. Furthermore, a new reward function must be handcrafted for every new problem assigned to the agent. Therefore reward shaping is not viable for learning optimal behaviour on a variety of different tasks.

2.5.2 Curriculum learning

Another method of shaping according to Randløv and Alstrøm [35] is to develop a multi-stage problem that is trained part by part. This idea is more in line with the definition of *shaping* by Sutton and Barto [45]. This approach is referred to as *curriculum learning* (CL). A sequence of increasingly complex tasks are specified. The agent is then trained on a task until it is able to reliably solve it before moving on to the next with the current policy. The process continues until the agent is able to solve the intended task. The goal is to improve the learning speed or final performance of the agent. The work by Selfridge *et al.* [40] showed that transferring knowledge from a simple version of a problem can help in learning a more difficult version.

The work by Salimans and Chen [38] used a similar technique to train an RL agent to play the Atari game, *Montezuma's Revenge*. This game is known to be very difficult for RL agents to solve due to the sparse nature of its rewards. Salimans and Chen [38] used a single successful trajectory in Montezuma's Revenge to train the agent. The agent was trained by presenting the trajectory to the agent in reverse order. The agent starts in a state close to its final goal and once it reliably solves the problem it is moved back along the trajectory. The difficulty of the starting state is gradually increased until the agent encounters the final intended problem.

Stellenbosch University https://scholar.sun.ac.za

2.5.3 Domain randomisation

Domain randomisation (DR) refers to randomising different aspects of a simulation environment during training. DR is often used to transfer models from simulation to the real world. The work by Tobin *et al.* [46] used DR to transfer an ANN used for object localisation from simulation to the real world. Various aspects of the simulation environment were randomised to enable the model to generalise to the real world. Tobin *et al.* [46] found that by randomising the simulation environment enough during training, the real world may be made to look to the trained model like just another version of the problem.

In this study, we investigate whether randomising certain aspects of the environment can aid the agent in solving sparse-reward problems. The goal is to expose the agent to a greater variety of problems. Similar to CL, the idea is that the agent first learns to solve easier versions of the problem and then use this knowledge to solve the intended problem.

2.6 Distributed deep reinforcement learning

Next, we investigate distributed DRL methods. The first is a method referred to as *general reinforcement learning architecture* (Gorila) and it distributes several components of a DRL algorithm. We then investigate an algorithm referred to as *Ape-X* that distributes data generation and prioritises important transitions.

2.6.1 Gorila

Gorila is an architecture that distributes various components of a DRL algorithm. The goal of Gorila is to have a scalable architecture that enables one to utilise plenty of computational resources. Nair *et al.* [27] used Gorila to implement the deep Q-learning algorithm of Mnih *et al.* [23, 24]. According to Nair *et al.* [27] the performance of the Gorila DQN surpassed the non-distributed DQN in 41 of the 49 Atari games. Nair *et al.* [27] also found that in most games, Gorila decreased the wall-time to obtain the results by an order of magnitude. Here we give a quick overview of how Gorila distributes components of the deep Q-learning algorithm.

Gorila employs parallel actors to increase the rate at which new transitions are generated. Gorila has several parallel actor processes. Each actor process has access to an instantiation of the same environment. The actors generate trajectories in their respective environments in parallel. The generated experience is either stored locally in a replay buffer on the actors' machines, or in a central distributed database.

Additionally, the architecture has a central parameter server that holds a distributed main ANN to represent the value function or policy. Each actor contains a replica of the main ANN in order to perform actions in its respective environment. The parameters

Stellenbosch University https://scholar.sun.ac.za

of the actors' networks are periodically synchronised with the parameters of the central parameter server.

The architecture also accommodates parallel learners that are trained from the stored experience (transitions). The learners also have access to copies of the main ANN. The learners are responsible for computing the desired changes to the parameters of the main ANN. Accordingly, the learners sample batches of transitions from the replay buffer to compute the gradients of the network. The gradients indicate how the parameters of the ANN should be adjusted. The gradients are communicated to the parameter server and the parameters of the respective learners' ANNs are periodically updated. The parameters of the main ANN are split across several machines. Each machine is responsible for adjusting the values of a subset of parameters. An asynchronous stochastic gradient descent algorithm is used to modify the main ANN's parameters using the gradients received from the learners.

Gorila is a very flexible architecture and allows for an arbitrary number of actors, learners, and parameter servers. Therefore any component of the architecture can be scaled in case there is a bottleneck.

2.6.2 Ape-X

In this section, we review a framework by Horgan *et al.* [16] that is referred to as *Ape-X*. Horgan *et al.* [16] applied it to the deep Q-learning algorithm by Mnih *et al.* [23, 24]. The Ape-X framework decouples acting from learning to generate orders of magnitude more data. Parallel actors generate experience in their own instances of the environment and perform actions according to a central ANN. Therefore the main focus of this approach is to generate data at a faster rate. Furthermore, Horgan *et al.* [16] utilised *prioritised experience replay* (PER) to focus on the most important transitions. In contrast to Gorila by Nair *et al.* [27], Ape-X utilises a centralised replay buffer, and instead of sampling uniformly, Ape-X uses prioritised sampling. Since transitions are centralised, transitions with high priority can benefit the whole system.

Prioritised experience replay (PER) by Schaul *et al.* [39] assigns maximum priority to newly acquired transitions. This is done to ensure that newly generated transitions are sampled. Priorities of transitions are only updated once the learner samples them. Horgan *et al.* [16] state that this approach does not scale well with Ape-X. Due to the large volume of data generated with the Ape-X framework, the learner will not be able to update priorities of transitions fast enough. Accordingly, this would lead to a focus on the most recent transitions generated. Therefore, Horgan *et al.* [16] used the actors to compute priorities for the transitions they have generated. At the time the Ape-X paper was published, Horgan *et al.* [16] state that this approach achieved state-of-the-art performance in the Atari 2600 games, using a fraction of the wall-time compared to previous approaches.

Stellenbosch University https://scholar.sun.ac.za

2.7 Summary

In this chapter, we reviewed work that addressed problems similar to the problem we address in our research. We first reviewed the achievements of the deep Q-learning algorithm by Mnih *et al.* [23, 24], since it is regarded as a breakthrough in DRL. The deep Q-learning algorithm is also central to this study. Mnih *et al.* [23, 24] applied deep Q-learning to 2D Atari games, whereas we apply the algorithm to a problem in a partially observable 3D environment with sparse rewards.

We then reviewed several approaches that deal with challenging exploration problems. We first reviewd DQfD, which utilises expert demonstrations to pre-train the RL agent's policy. We will not utilise this approach in this study since, among other reasons, it is expensive to generate expert demonstrations. We also reviewed a curiosity-driven approach by Pathak *et al.* [33]. Although this technique seems very interesting, it adds additional complexities to the DRL algorithm. We then reviewed the algorithm Go-Explore by Ecoffet *et al.* [8]. This approach exploits the characteristics of the simulator to find successful trajectories. Once a successful trajectory is obtained, Go-Explore uses an approach very similar to *curriculum learning* (CL) to solve the sparse-reward problem. We also reviewed shaping, including reward function shaping, CL and DR. Reward function shaping is not recommended by Sutton and Barto [45] since it may lead to sub-optimal policies as discussed in Section 2.5.1. Therefore we do not use this approach in this study. Since CL and DR are very straightforward to implement, we investigate how these approaches aid in solving difficult exploration problems.

Finally we reviewed two approaches that distributed DRL algorithms namely, Gorila and Ape-X. Since the work by Horgan *et al.* [16] show very good results, we investigate how this approach can aid in solving the sparse-reward problem we address.

Stellenbosch University https://scholar.sun.ac.za

Chapter 3

Markov decision processes

In this chapter we review the *Markov decision process* (MDP) which is a popular way to model sequential decision-making problems. The MDP is a formulation of a decision making problem, where actions do not only have an influence on the immediate situation and reward, but also affect future situations and future rewards. MDPs contain the concept of rewards that are delayed and therefore the idea of choosing between immediate rewards and future rewards is important. In this chapter we discuss key MDP-related concepts such as rewards, returns, value functions, policies, and Bellman equations. This chapter is largely based on the work by Sutton and Barto [45] and Russell and Norvig [37].

3.1 Markov chains

We start by introducing the Markov chain as a stepping stone to the more general MDP. A Markov chain consists of number of states

$$\mathcal{S} = s_0, s_1, \ldots, s_n \tag{3.1}$$

and has a transition model that models transitions from one state to another. The transition model is a probability distribution that specifies the probability of a state given the history of all previous states, i.e.

$$Pr\{S_{t+1}|S_0, S_1, \ldots, S_t\} = Pr\{S_{t+1}|S_{0:t}\}, \tag{3.2}$$

where $S_{0:t}$ represents the states $S_0, S_1, \ldots, S_t$. The problem is that, as t increases, the history of states $S_{0:t}$ can become very large and computationally intractable. This can be solved by the Markov assumption which states that the entire history of states $S_{0:t}$ is contained in a fixed finite number of previous states. The first order Markov assumption simplifies this even further and assumes that complete state history is contained only in the current state S_t. The first order Markov assumption can now be used to simplify the transition model to

$$Pr\{S_{t+1}|S_{0:t}\} = Pr\{S_{t+1}|S_t\}. \tag{3.3}$$

Stellenbosch University https://scholar.sun.ac.za

Equation 3.3 states that the current state provides sufficient information to make a prediction about the next state that is independent of all past states [37].

The Markov chain can therefore be used to model many stochastic processes such as the growth of populations. Unfortunately, these models can only be used to predict future states based on current states. In addition, there is no way to interact with the model in order to influence the state of the model. The Markov chain also does not allow for specifying desirable states in the model.

3.2 Agent and environment interaction

The MDP is a generalisation of the Markov chain and is defined by the following:

- a state space $\mathcal{S} = s_0, s_1, \dots, s_n$.
- an action space $\mathcal{A} = a_0, a_1, \dots, a_n$.
- an initial state distribution p_0 that models the state in which the MDP starts.
- a state transition model $p(s'|s,a) \doteq Pr\{S_{t+1} = s'|S_t = s, A_t = a\}$. It describes the probability of reaching the state s' at time step $t+1$, after taking action a in state s at time step t.
- a reward function $r(s,a,s') \doteq \mathbb{E}[R_{t+1} = r|S_t = s, A_t = a, S_{t+1} = s']$. It defines the reward obtained arriving at state s' at time step $t+1$, after taking the action a in state s at time step t.
- a discount rate $\gamma \in [0,1]$, defining the importance of immediate rewards versus future rewards.

The MDP describes a problem where one can learn to achieve a goal through interaction. This is the problem *reinforcement learning* (RL) tries to solve. It usually consists of an agent and an environment, as shown in Figure 1.1. The agent is also known as the decision maker or learner and repeatedly interacts with the environment by performing actions. The environment reacts to the actions of the agent by providing it with new situations and a reward signal. In an MDP it is presumed that the environment is fully observable, therefore the agent accurately observes the precise state of the environment. The agent performs actions that are based on the states the environment presents.

The interaction between the agent and the environment takes place at discrete time steps, $t = 0, 1, 2, 3, \dots$. In most problems this interaction continues for a finite T time steps. In continuous tasks T is infinite, but this study is only concerned with episodic tasks where T is finite. A trajectory of agent-environment interaction is usually in the form of

$$S_0, A_0, S_1, R_1, A_1, S_2, R_2, A_2, \dots, S_T, R_T, \tag{3.4}$$

Stellenbosch University https://scholar.sun.ac.za

and at time step T the trajectory is completed. At each time step of the trajectory, the transition model of the MDP is used to obtain the following state and reward. Therefore, when the agent is in the state s, and takes the action a, the transition model that describes the dynamics of the MDP returns a probability of reaching the state s' and receiving the reward r. Sutton and Barto [45] define the transition model as

$$p(s', r|, s, a) \doteq Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}. \tag{3.5}$$

The transition model uses the first order Markov assumption, therefore the probability of reaching state s' and receiving the reward r depends only on the current state s and action a.

3.3 Rewards and returns

The goal of an RL agent is described in terms of the reward signal that it receives at every time step. The reward signal is a real number, $R_t \in \mathbb{R}$ and the objective of the agent is to maximise the sum of rewards received over time, i.e. the cumulative sum of rewards received in the course of its trajectory and not just the immediate reward. The sum of all the rewards received during a trajectory in an MDP is called the *return*, denoted with G_t. In the simplest case, the return G_t of a trajectory is the sum of rewards received after the time step t,

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{3.6}$$

where T is the final time step of the trajectory. We now also discuss the idea of discounting rewards. By using this approach, the agent's objective is to choose actions in order to maximise the expected sum of discounted rewards. Sutton and Barto [45] define the discounted return G_t as

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \end{aligned} \tag{3.7}$$

Gamma $\gamma \in [0, 1]$ is the *discount rate* and determines the value of future rewards by scaling rewards based on the time step when it is acquired. A value close to 0 favours rewards that are acquired in the near future, which leads to a *myopic* evaluation, whereas a value close to 1 assigns equal importance to all future rewards and leads to a more *far-sighted* evaluation. Discounting rewards allows us to favour rewards based on the time they are received. It also allows the return G_t to be finite in tasks where $T = \infty$. Note that there is a recursive relationship between returns at successive time steps in Equation 3.7

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{3.8}$$

Stellenbosch University https://scholar.sun.ac.za

This recursive relationship is fundamental to RL algorithms and is also present in the Bellman equation covered in Section 3.5.

3.4 Policies and value functions

In an MDP the agent has to sequentially choose actions to accumulate rewards with the goal to receive the highest possible return. Unfortunately a fixed sequence of actions will not solve an MDP due to the uncertainty that is present in the transition model. The agent therefore needs to find a solution where each state in the MDP is mapped to a probability distribution that gives the probability of choosing each possible action. Such a solution is described by Sutton and Barto [45] as a stochastic rule which selects actions as a function of states, and is defined as policy π. If the agent is following a policy π at time step t, then $\pi(a|s)$ is the probability of taking the action a given the state s. The state-value function $v_\pi(s)$ is a way to evaluate the quality of a policy. The state-value function $v_\pi(s)$ is the expected return G_t if the agent is in state s and then follows a policy π and is defined by Sutton and Barto [45] as

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t | S_t = s\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \text{ for all } s \in \mathcal{S}. \end{aligned} \tag{3.9}$$

The state-value function $v_\pi(s)$ is thus an estimate of how good it is to be in a given state. The action-value function $q_\pi(s,a)$ is another way to evaluate a policy π. The action-value function $q_\pi(s,a)$ is the expected return G_t if the agent is in state s, takes action a and then follows policy π. Sutton and Barto [45] define it as

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]. \end{aligned} \tag{3.10}$$

The action-value function $q_\pi(s,a)$ is therefore an estimate of how beneficial it is to perform a specified action in a given state and then following the policy π afterwards.

3.5 Bellman equation

The previous section described the policy of an agent, which is used by the agent to perform actions in an MDP. For a given MDP, the value function is used to describe the quality of a policy π. In this section we review the Bellman equation. It can be used to break down the value function to describe the recursive relationship between a state's value and the values of the states that follow it [45].

The recursive relationship in Equation 3.8 can be used to decompose the expected return G_t in Equation 3.9 as the expected immediate reward R_{t+1} plus the expected

Stellenbosch University https://scholar.sun.ac.za

discounted return at the next time step γG_{t+1}. Sutton and Barto [45] show that Equation 3.9 becomes

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(S_{t+1})\Big], \text{ for all } s \in \mathcal{S} \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s\Big]. \end{aligned} \tag{3.11}$$

The expected discounted return G_{t+1} is replaced with the discounted state-value function of the next state $\gamma v_\pi(S_{t+1})$. The outer expectation becomes a summation over the variables a, s', and r. For each combination of the variables a, s', and r, the probability $\pi(a|s)p(s', r|s, a)$ is computed. The value between the brackets is weighted by each probability, then a sum over all possibilities is computed to get an expected value. Equation 3.11 is known as the Bellman equation for v_π, and is named after Richard Bellman, who developed it in 1957. It states that the value of a given state is equal to the discounted value of the expected next state plus the value of the expected received reward. The action-value function can be similarly decomposed and gives

$$q_\pi(s, a) = \mathbb{E}_\pi\Big[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a\Big]. \tag{3.12}$$

The Bellman equation can be used to check for optimality and for recursive assignments during policy evaluation, which is further discussed in Section 3.7

3.6 Policies and value functions that are optimal

Essentially, in order to solve the MDP, a policy is needed that will maximise the received reward over time. If the expected return of a policy π is greater than or equal to the expected return of a policy π' for all states, then the policy π is said to be better than the policy π'. Therefore Sutton and Barto [45] state that

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s), \text{ for all } s \in \mathcal{S}. \tag{3.13}$$

They show that every MDP has a deterministic *optimal policy* π_* that is better than or equal to all other policies,

$$\pi_* \geq \pi, \text{ for all } \pi. \tag{3.14}$$

Sutton and Barto [45] state that all optimal policies achieve the same state-value function,

$$v_{\pi_*}(s) = v_*(s), \tag{3.15}$$

Stellenbosch University https://scholar.sun.ac.za

called the *optimal state-value function.* Similarly, Sutton and Barto [45] state that all optimal policies achieve the same action-value function,

$$q_{\pi_*}(s,a) = q_*(s,a), \tag{3.16}$$

called the *optimal action-value function.* The MDP is considered to be solved when the optimal value function is found, as it stipulates the best possible performance in the MDP. The optimal state-value function $v_*(s)$ yields the maximum expected return for all possible policies in the MDP and is given by Sutton and Barto [45] as

$$v_*(s) \doteq \max_{\pi} v_\pi(s), \text{ for all } s \in \mathcal{S}. \tag{3.17}$$

Similarly, the optimal action-value function $q_*(s)$ yields the maximum expected return for all policies and is given by Sutton and Barto [45] as

$$q_*(s,a) \doteq \max_{\pi} q_\pi(s,a), \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s). \tag{3.18}$$

An optimal policy π_* can be easily determined once v_* is obtained. Sutton and Barto [45] describe the *Bellman optimality equation* as

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}\Big[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\Big] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_*(s')\Big]. \end{aligned} \tag{3.19}$$

It is said that a state value under optimal policy must be equal to the expected return for the best action from that state [45]. Similarly the Bellman optimality equation for q_* according to Sutton and Barto [45] is

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1},a') | S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \max_{a'} q_*(s',a')\Big]. \end{aligned} \tag{3.20}$$

Sutton and Barto [45] state that by acting *greedy* with respect to an optimal value function v_*, an optimal policy π_* is being followed. This means that with v_* you have to do a one-step search to select the best action(s) that gives the maximum expected return. It is even easier with q_*, as no one-step search is needed and you only have to choose the action that gives the maximum expected return:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname*{argmax}_{a \in \mathcal{A}(s)} q_*(s,a) \\ 0 & \text{otherwise} \end{cases}. \tag{3.21}$$

Stellenbosch University https://scholar.sun.ac.za

The value functions v_* and q_* are optimal for the expected long-term return, and therefore by acting greedily with respect to them, one is also choosing the best action at every time step to receive the most reward in the long term.

Solving the Bellman optimality equation is a way to find an optimal policy for the MDP, and therefore a way to solve the MDP. Sutton and Barto [45] state that it is rarely feasible to solve it in practice and one should rather use other methods to approximate solutions.

3.7 Dynamic programming

Dynamic programming (DP) algorithms can be used to obtain optimal policies given a perfect model that describes the dynamics of the environment. The model refers to the transition probability distribution and reward function of the environment. The idea of DP is to break a problem into a series of sub-problems, the solutions to which then can be combined to solve the original problem [4]. The algorithms discussed in this section rely on the recursive relationship described with the Bellman equation (Equation 3.11). They use this recursive relationship to introduce the idea of *bootstrapping*. Bootstrapping entails updating a prediction based on another prediction. The main DP algorithms reviewed to obtain optimal value functions and policies are policy iteration and value iteration. These methods are referred to as *model-based* methods as they require a dynamics model of the environment to obtain optimal behaviour. *Generalised policy iteration* (GPI) is also discussed here, which form the basis of the DP methods discussed in the chapter. GPI is also the basis of the *model-free* (where no dynamics model is needed) methods introduced in Chapter 4.

3.7.1 Policy evaluation

Iterative policy evaluation is a method that evaluates a given policy π to obtain its value function. We refer to the problem of evaluating a policy as the *prediction problem*. The Bellman equation, Equation 3.11, is applied to do iterative expectation backups in order to estimate v_π. At each iteration k of the process, the value of each state, $s \in \mathcal{S}$, is updated using the value of its successor states. A single iteration of iterative policy evaluation to update the value of a state s is defined by Sutton and Barto [45] as

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi\Big[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s\Big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big]. \end{aligned} \tag{3.22}$$

We define a *sweep* as updating all the states of the state space once. It is said that repeating this process for $k \to \infty$ will converge to v_π [45]. It should be noted from Equation 3.22 that the dynamics model is required, and therefore this is a model-based method.

Stellenbosch University https://scholar.sun.ac.za

3.7.2 Policy iteration

Policies can be evaluated with policy evaluation, but ideally we would also like to be able to improve a policy. *Policy iteration*, a model-based method to improve a policy π, consists of two main steps. The policy π is first evaluated to obtain v_π using policy evaluation discussed in the previous section. After the policy π is evaluated, it is improved by defining a new policy that selects the greedy actions with respect to the obtained state-value function v_π. The new greedy policy π' is given by Sutton and Barto [45] as

$$\begin{aligned}\pi'(s) &\doteq \underset{a}{\operatorname{argmax}}\ q_\pi(s,a)\\ &= \underset{a}{\operatorname{argmax}}\ \mathbb{E}\Big[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\Big]\\ &= \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big].\end{aligned} \tag{3.23}$$

After the policy has been improved, the process repeats itself and the newly improved policy is evaluated again and so forth. If improvement stops, the optimal policy has been found, i.e.

$$q_\pi\Big(s, \pi'(s)\Big) = \max_{a\in\mathcal{A}} q_\pi(s,a) = q_\pi\Big(s,\pi(s)\Big) = v_\pi(s). \tag{3.24}$$

Accordingly the Bellman optimality equation has been satisfied,

$$v_\pi(s) = v_*(s), \text{ for all } s \in \mathcal{S}, \tag{3.25}$$

which makes π an optimal policy. Sutton and Barto [45] suggest that the optimal policy π_* will always be obtained when repeating the policy iteration process.

3.7.3 Value iteration

In Section 3.6 we established that the optimal value function of an MDP stipulates the optimal performance in the MDP. Acting greedily with respect to the optimal state-value function v_* therefore yields the optimal policy π_*. Value iteration is a method where no explicit policy is needed to achieve the optimal value function of the MDP. If we know the solution of $v_*(s')$, then we can find the solution of $v_*(s)$ (where s' is the successor state to s). Iterative Bellman backups can be applied to obtain v_*. At each iteration $k+1$ of this process, the value of each state $v_{k+1}(s)$ is updated from the value of its successor state $v_k(s')$ and is expressed by Sutton and Barto [45] as

$$\begin{aligned}v_{k+1}(s) &\doteq \max_a \mathbb{E}\Big[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a\Big]\\ &= \max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big].\end{aligned} \tag{3.26}$$

Equation 3.26 is essentially policy iteration, but at each iteration, the policy evaluation is stopped after one sweep.

Stellenbosch University https://scholar.sun.ac.za

3.7.4 Generalised policy iteration

Policy iteration, discussed in Section 3.7.2, consists of two mechanisms that interact with each other. The first is a process that entails learning the value function of the current policy, i.e. policy evaluation. The second mechanism is a process that updates the policy to be greedy with respect to the estimated value function, i.e. policy improvement. In policy iteration the current process finishes before the other starts. However, each process does not have to completely finish before the next starts. For example value iteration, discussed in Section 3.7.3, performs only a single sweep of policy evaluation between each step of improving the policy. The term *generalised policy iteration* (GPI) is used to refer to the concept of policy evaluation and policy improvement procedures interacting in order to find an optimal policy that is consistent with its greedy counterpart value function. GPI forms the basis of most RL methods and is used in the model-free control methods reviewed in Chapter 4.

3.8 Partially observable Markov decision processes

We previously assumed that the environment of the MDP is fully observable, therefore the agent always knows the exact environmental state. This assumption combined with the Markov assumption means that an optimal action at any time step only depends on the agent's current state. Unfortunately in most real-world problems, the environment is not fully observable. Furthermore the value function of a state s usually does not only depend on the state s, but also on *how much the agent knows* when it is in s. The *partially observable Markov decision process* (POMPD) formulates the above variation of the MDP. The POMPD is defined by the following:

- a finite state space $\mathcal{S} = s_0, s_1, \dots, s_n$.
- a finite action space $\mathcal{A} = a_0, a_1, \dots, a_n$.
- a finite set of observations $\mathcal{O} = o_0, o_1, \dots, o_n$.
- a state transition model $p(s'|s,a) \doteq Pr\{S_{t+1} = s'|S_t = s, A_t = a\}$.
- a reward function $r(s,a,s') \doteq \mathbb{E}[R_{t+1} = r|S_t = s, A_t = a, S_{t+1} = s']$.
- an observation function $p(o|s',a) = Pr\{O_{t+1}|S_{t+1} = s', A_t = a\}$.
- γ is a discount factor $\gamma \in [0,1]$.

In a POMPD the agent can no longer observe its true state S_t, only an observation $O_t \in \mathcal{O}$ that provides partial information of the state S_t. For convenience we assume that the reward R_t is included in the observation O_t.

Stellenbosch University https://scholar.sun.ac.za

Sutton and Barto [45] define the *history* H_t as the complete sequence of actions, and observations of a trajectory up to a time step t:

$$H_t = A_0, O_1, \dots, A_{t-1}, O_t. \tag{3.27}$$

The idea of a state that satisfies the Markov assumption is that the state summarises the history and can be used instead of the actual history to make predictions about the future. If the agent's observations no longer satisfy the Markov assumption, the actual history can be used to recover the idea of a state (that satisfies the Markov assumption). A *belief state* $b(h)$ is a probability distribution over the latent or hidden states, given the history h,

$$b(h) = \Big(Pr\{S_t = s^1 | H_t = h\}, \dots, Pr\{S_t = s^n | H_t = h\}\Big). \tag{3.28}$$

The *belief state* $b(H_t = h)$ can be used to recover a belief of the state S_t which satisfies the Markov assumption. As the history becomes longer, this approach does not scale well computationally and is not recommended by Sutton and Barto [45] to be used in artificial intelligence applications.

Sutton and Barto [45] recommend the *kth-order history* approach which is a simple alternative to approximate an unknown current state S_t. It entails using only the k last observations and actions of the agent to represent the state S_t,

$$S_t \doteq O_t, A_{t-1}, O_{t-1}, \dots, A_{t-k}, \text{ where } k \geq 1. \tag{3.29}$$

Although it is a very simple solution, the agent can perform significantly better than when only using the latest observation O_t as the current state S_t [45].

3.9 Summary

In this chapter, we introduced the MDP which is a method to describe the problem RL is trying to solve. In summary an MDP models a sequential decision-making problem in an environment that is fully observable and stochastic with a transition model that satisfies the Markov assumption [37]. Furthermore, an MDP also includes a cumulative reward function. We also discussed that an optimal policy or value function is needed to solve an MDP.

Dynamic programming, encompassing methods that require a perfect environment model, has been reviewed. These methods include policy iteration and value iteration and are used to obtain optimal policies and value functions in the MDP. Even if an accurate model of the environment is available, these methods may still be unfeasible due to large computational requirements. In the next chapter, we investigate methods to find policies in an MDP that do not require a model that perfectly describes the dynamics of the environment.

Stellenbosch University https://scholar.sun.ac.za

We also introduced the POMPD – a variation of the MDP where the environment returns an observation O_t that does not satisfy the Markov assumption. Most real-world MDPs are in the form of a POMPD and it is therefore important to consider.

Stellenbosch University https://scholar.sun.ac.za

Chapter 4

Reinforcement learning

Chapter 3 provided an overview of *Markov decision process*es (MDPs) as a way of modelling a sequential decision-making problem. Model-based algorithms to solve MDPs, such as policy iteration and value iteration, were discussed in Section 3.7. Unfortunately, these methods need access to a state-transition model and a reward function to compute optimal policies. In most real-world problems, a perfect model describing the dynamics of the environment is rarely available and is often too expensive to use or to store. Therefore methods such as policy iteration and value iteration can very seldom be used. In order to solve real-world problems, it is essential to be able to achieve an optimal policy and value function without knowing the underlying model of the environment.

This chapter explores tabular methods that can achieve optimal value functions and policies by only interacting with the environment. These methods *learn* optimal behaviour from experience generated in the environment and do not require knowledge of the dynamics of the environment. Therefore these methods are termed *model-free*. In this study, we are concerned with value-based methods, but first, we differentiate between value-based and policy-based methods. *Monte Carlo* (MC) methods, where samples of complete episodes are needed, are then discussed and used to introduce value-based methods. We then discuss *temporal-difference* (TD) methods that utilise bootstrapping in the subsequent section.

For each category, we first consider how a policy π can be evaluated to obtain the value function v_π. Recall that this is referred to as the *prediction problem* or as the process of performing *policy evaluation*. We then discuss how these methods can be applied to obtain the optimal value-function v_* and an optimal policy π_*. We refer to this as the process of performing *control*. Both MC learning and TD learning are based on *generalised policy iteration* (GPI) – the iterative process of evaluating and improving a policy as discussed in Section 3.7.4.

In Section 4.4, we discuss why tabular control methods are not feasible to use in problems with large state spaces and action spaces. As tabular methods do not scale very well, Sutton and Barto [45] recommend approximating the value functions using a parameter vector of lower dimensionality. We, therefore, review how function approximation can be

Stellenbosch University https://scholar.sun.ac.za

used in combination with the discussed TD control methods. This chapter is primarily based on the work of Sutton and Barto [45].

4.1 Policy-based and value-based methods

We first differentiate between the main model-free control categories, namely *value-based* and *policy-based* methods. The goal of value-based methods is to estimate the optimal action-value function q_* of an MDP, which allows us to also obtain an optimal policy π_*. Value-based methods allow for the learning of optimal behaviour using *off-policy* experience. Off-policy methods are described by Sutton and Barto [45] as methods that evaluate or improve a policy different from that used to generate experience. Therefore the *target policy* (the policy the agent learns about) can be different from the *behavioural policy* (the policy used to generate experience). The advantage of off-policy methods is that any experience generated by interacting with the environment, regardless of the method used to explore the environment, can be used to update the value function. There are also value-based methods where optimisation is performed using *on-policy* experience, for example SARSA, which we discuss in Section 4.3.2. According to Sutton and Barto [45], on-policy methods evaluate or improve a policy using experience that is generated by the same policy. Therefore the target policy and behavioural policy are the same when performing on-policy optimisation.

Policy-based methods, such as REINFORCE, aim to directly optimise a parameterised policy without explicitly representing the value function. According to Nachum *et al.* [26], the advantage of this approach is that it directly optimises the objective of interest, and these methods are usually also stable under function approximation. Furthermore, according to Silver [41], policy-based methods are suitable to use in continuous or high-dimensional action spaces, can learn policies that are stochastic and have good convergence properties. Policy-based methods almost always optimise the agent's policy using on-policy experience. According to Nachum *et al.* [26], the biggest drawback of these methods is that they are usually not very sample-efficient. Sample efficiency is important since in some environments it can be very expensive to generate environmental transitions. Value-based and policy-based methods can be combined, referred to as *actor-critic* methods, to address the shortcomings of the different approaches.

In this study, we focus on value-based methods as they naturally accommodate the use of off-policy experience. Off-policy learning methods can be more sample-efficient as they allow the use of other sources than on-policy experience for learning. For example, Sutton and Barto [45] state they can learn from experience generated by an expert human, by a conventional non-learning controller or reuse experience that the agent has previously generated. We reuse previously generated data, as mentioned in the discussion of our approach in Chapter 6.

Stellenbosch University https://scholar.sun.ac.za

4.2 Monte Carlo methods

In this section we review *Monte Carlo* (MC) methods to introduce value-based model-free control. MC methods can obtain optimal policies and value functions by learning from experience which is in the form of entire episodes. These methods, therefore, do not use bootstrapping and work well for tasks with well-defined returns. Recall that bootstrapping entails updating an estimate based on another estimate (see Section 3.7). Tasks that can be separated into episodes are therefore suitable for these methods. We introduce certain key concepts in this section that are also relevant to other model-free control methods.

4.2.1 Monte Carlo policy evaluation

We first discuss how MC methods can be used for prediction or policy evaluation. It has been shown by Sutton and Barto [45] that if the agent follows a policy π and encounters a state s, the average return that follows the state s will converge to the state's value $v_\pi(s)$, as the number of times the state is visited approaches infinity. If an average is tracked for each action taken in the state s, then the state-action value $q_\pi(s, a)$ can similarly be determined. These methods of estimating the value functions are called *MC methods* because they entail averaging over many random sample episodes using the actual returns received [45]. As MC methods use samples of entire episodes for policy evaluation, the estimate of the value function is unbiased. The drawback is that MC methods have to wait until the return G_t is known. Therefore values of states can only be updated at the end of an episode.

Each time a state s is encountered we refer to it as a *visit* to s. The first time the state s is encountered, we refer to as the *first visit* to s. There are two main MC policy evaluation methods to estimate v_π, namely the *first-visit* MC method and the *every-visit* MC method. It has been shown by Sutton and Barto [45] that the first-visit MC method estimates $v_\pi(s)$ as the mean return following the first visits to s. On the other hand, the every-visit MC method estimates $v_\pi(s)$ as the mean return following all visits to s. The first-visit policy evaluation (tabular) procedure adapted from the work by Sutton and Barto [45] is shown in Algorithm 1. According to Sutton and Barto [45], first-visit MC and every-visit MC converge to $v_\pi(s)$ as the number of times s is encountered, approaches infinity. The result is an unbiased estimate of the expected value.

According to Silver [41], $V(s)$ can also be updated incrementally

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}\Big[G_t - V(S_t)\Big], \tag{4.1}$$

where G_t is the return after time step t in an episode $S_1, A_1, R_2, \ldots, S_T$. The number of times the state S_t has been visited is represented by $N(S_t)$. Methods where an average is used over all returns achieved following a state S_t, are appropriate for problems where the reward function is stationary, i.e. where the reward function does not change over time. In

Stellenbosch University https://scholar.sun.ac.za

Algorithm 1: First-visit MC prediction, adapted from Sutton and Barto [45]. It is used for estimating $V \approx v_\pi$. An infinite number of episodes are completed. In every episode when a first visit is made to a state S_t, the return following state S_t is appended to returns(S_t). This allows us to estimate $V(S_t)$ by averaging all the returns obtained after visiting S_t for the first time. This algorithm can be transformed to every-visit MC evaluation by removing the if statement in line 9.

```
1  input: a policy $\pi$ to be evaluated
2  initialise: value function $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
3  initialise: returns($s$) $\leftarrow$ an empty list, for all $s \in \mathcal{S}$
4  for each episode do
5      generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
6      $G \leftarrow 0$
7      for each step of episode, $t = T-1, T-2, \ldots, 0$ do
8          $G \leftarrow G + R_{t+1}$
9          if $S_t$ not in $S_0, S_1, \ldots, S_{t-1}$ then
10             Append $G$ to returns($S_t$)
11             $V(S_t) \leftarrow \text{average}\Big(\text{returns}(S_t)\Big)$
12         end
13     end
14 end
```

non-stationary problems where the reward function may change over time, it makes sense to give weight to more recent rewards. It is argued by Silver [41] that it can be useful to track a running mean and therefore forget old episodes in non-stationary problems. Equation 4.1 becomes

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big], \tag{4.2}$$

where α is a constant step-size parameter.

4.2.2 Action-value function for model-free policy improvement

In Section 3.7 a model of the environment was required to perform a greedy policy improvement over a state-value function. Recall that a greedy policy improvement over $v_\pi(s)$ is given by

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big]. \tag{4.3}$$

In Equation 4.3 it is clear that the probabilities $p(s', r|s, a)$, which characterise the dynamics of the environment, are required to perform the policy improvement. It is therefore impossible to use Equation 4.3 in model-free settings, where no model is available. In these applications, the action-value function is handy as it can be used to improve a policy without having access to the environment's model. A greedy policy improvement over $q_\pi(s, a)$ is given by

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\ q_\pi(s, a). \tag{4.4}$$

We therefore use the action-value function in MC control in order to do policy improvements.

Stellenbosch University https://scholar.sun.ac.za

4.2.3 ε-greedy exploration

Acting *greedily* with regard to the current estimated action-value function allows us to exploit our current knowledge of the system and allows for the maximisation of the expected reward on the current step. By selecting nongreedy actions, alternative actions are explored and allow for improving the estimates of values of the nongreedy actions. Alternative actions may be better than what current knowledge estimates to be best. After discovering better actions, these can be exploited to produce greater reward in the long run. There is, therefore, a conflict between exploration and exploitation as it is not possible to perform both with a single action.

MC methods use samples of experienced episodes to update value functions and to improve policies. Sufficiently exploring all possible state-action pairs with MC methods can be problematic. By selecting only actions that are estimated to be the best, i.e. always performing greedy actions, alternative actions, which may be better, are never selected. We must therefore ensure that the algorithm continues to explore alternative state-action pairs. According to Sutton and Barto [45] we can solve this problem by starting episodes with randomly sampled state-action pairs (with all state-action pairs having a none-zero probability of being sampled). This guarantees that all state-actions pairs will be encountered an infinite number of times for an infinite number of episodes. Sutton and Barto [45] define the above assumption of randomly initialising state-action pairs at the start of an episode as *exploring starts.* In simulated environments, exploring starts can often be implemented, but this method is usually not feasible to use with real-world problems.

An alternative way to address the problem of maintaining exploration is to have a stochastic policy that has a non-zero probability of selecting all possible actions in each state. In applications where exploring starts cannot be used, the ε-*greedy* exploration strategy is convenient to use. It is a very simple, effective exploration strategy that ensures continual exploration [23]. Silver [41] defines the ε-greedy strategy as

$$\pi(a|s) = \begin{cases} \varepsilon/|\mathcal{A}(S_t)| + 1 - \varepsilon & \text{if } a^* = \underset{a}{\operatorname{argmax}}\ Q(s,a) \\ \varepsilon/|\mathcal{A}(S_t)| & \text{otherwise} \end{cases}, \tag{4.5}$$

where $|\mathcal{A}(S_t)|$ is the cardinality of the action space. Every time the agent performs an action, it has a probability of $1-\varepsilon$ of selecting the greedy action, else (with a probability of ε) the action is randomly sampled.

4.2.4 Monte Carlo control

We proceed to discuss MC control where an optimal value function v_* and optimal policy π_* are obtained by using entire sample episodes for optimisation. On an episode-by-episode basis, MC control methods combine the policy evaluation and policy improvement

Stellenbosch University https://scholar.sun.ac.za

steps of *generalised policy iteration* (GPI)(see Section 3.7.4). An iterative process can be implemented that estimates the action-value function of a policy and then improves the policy with regard to the estimated action-value function. As MC control works on an episode-by-episode basis, the action-value function and policy are only updated every time an episode is completed.

Algorithm 2 shows the pseudo code of an ε-greedy on-policy MC control algorithm. It

Algorithm 2: Monte Carlo Control: On-policy first-visit MC control using ε-greedy exploration, adapted from the work by Sutton and Barto [45]. The algorithm estimates the optimal policy π_*. A policy π is initialised and the action-value function q_π is estimated (rather than v_π). The ε-greedy policy improvement step allows improving the current policy while the new policy still explores nongreedy actions from time to time. The ε-greedy exploration strategy is important to include, as this algorithm does not utilise exploring starts.

```
1  initialise: small $\varepsilon > 0$
2  initialise: $\pi \leftarrow$ an $\varepsilon$-greedy policy
3  initialise: $Q(s,a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
4  initialise: returns$(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
5  for each episode do
6      generate an episode following $\pi : S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
7      $G \leftarrow 0$
8      for each step of episode, $t = T-1, T-2, \ldots, 0$ do
9          $G \leftarrow G + R_{t+1}$
10         if $S_t, A_t$ not in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$ then
11             append $G$ to returns$(S_t, A_t)$
12             $Q(S_t, A_t) \leftarrow$ average$\Big($returns$(S_t, A_t)\Big)$
13             $A^* \leftarrow \underset{a}{\text{argmax}} Q(S_t, a)$
14             for all $a \in \mathcal{A}(S_t)$ do
15                 $\pi(a|s) = \begin{cases} \varepsilon/|\mathcal{A}(S_t)| + 1 - \varepsilon & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
16             end
17         end
18     end
19 end
```

is an on-policy algorithm as the *target policy* (which is ε-greedy) is also used for behaviour, i.e. generating experience. In the policy improvement step (line 15), we do not improve the policy by making a greedy improvement. However, we instead do an ε-greedy improvement with regard to the current estimated action-value function. The ε-greedy improvement to the policy allows continuing to explore nongreedy actions that may lead to better policies.

According to Sutton and Barto [45], we need a policy that still explores, and therefore the on-policy method in Algorithm 2 compromises by not learning about the optimal policy, but about a near-optimal policy that still explores. Sutton and Barto [45] state

Stellenbosch University https://scholar.sun.ac.za

that GPI does not require improving all the way to a greedy policy, but only toward a greedy policy. Therefore, the ε-greedy improvement with regards to q_π is guaranteed to be better than or equal to π [45].

The main benefit of MC control over DP is that it is a model-free control method. MC control can learn value functions and optimal policies from sample episodes that are completed in the environment. A drawback of this method is that it is unable to learn continuous tasks. Furthermore, it cannot update the value function before an episode has ended. This is not problematic for tasks with short episodes, but it can significantly increase the time taken to learn the optimal policy. According to Sutton and Barto [45], MC methods (which do not bootstrap) may be less susceptible to the violations of the Markov assumption, as they do not update estimated state values based on estimated state values of subsequent states.

4.3 Temporal-difference methods

Temporal-difference (TD) methods are central to RL and, like MC methods, can learn optimal policies and value functions from environmental interaction without the need for a model that describes the dynamics of the environment. Unlike MC methods, these methods do not have to wait until the end of an episode before updating the value function and improving the policy. They achieve this ability by using bootstrapping, which we introduced with dynamic programming (DP) in Section 3.7. Bootstrapping entails updating estimated values based on other estimated values. This allows them to be more effective in solving continuous tasks or tasks with long episodes, which are problematic for MC methods.

We again first discuss how TD methods can be applied to solve the *prediction problem* or to evaluate a policy. Recall that policy evaluation entails estimating the value function v_π for a policy π. We then discuss two TD *control* methods. Recall that control methods entail finding optimal value functions and optimal policies. Like DP and MC methods, TD methods are also based on GPI. The main difference between the methods is in how they approach policy evaluation.

4.3.1 Temporal-difference policy evaluation

In Section 4.2.1, we explained how MC methods use sample returns from experience following the policy π to approximate a value function $V \approx v_\pi$. Recall that an every-visit MC policy evaluation method for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]. \tag{4.6}$$

From Equation 4.6 it can be seen that the entire sample return G_t is required to update the estimated value $V(S_t)$. Recall the relationship described by the Bellman equation,

Stellenbosch University https://scholar.sun.ac.za

Equation 3.11,

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\Big[G_t|S_t = s\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma G_{t+1}|S_t = s\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s\Big]. \end{aligned} \tag{4.7}$$

TD methods use this relationship to estimate the return G_t. The estimated return is also called the *TD target*. The simplest TD learning algorithm, *TD*(0), also known as *one-step TD*, updates the value $V(S_t)$ toward the one-step TD target. The one-step TD target is the immediate reward received plus the estimated value of the subsequent state, $R_{t+1} + \gamma V(S_{t+1})$. We refer to this method as *one-step* TD, because it is a special case of the n-step TD methods discussed in Section 4.3.4. We therefore replace the return G_t in Equation 4.6 with the one-step TD target,

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]. \tag{4.8}$$

TD methods update the estimated value of the state by using the estimated value of the subsequent state and therefore use bootstrapping.

The value

$$\delta \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{4.9}$$

is known as the *TD error*. It defines the error between the current state value $V(S_t)$ and the TD target. The goal is to minimise the TD error. As the TD error becomes smaller, the value function becomes stable, therefore the estimated value function V of the policy π converges to the true value function v_π. TD methods are capable of *online learning*; that is, they can update estimated state values at every time step and do not have to wait until the end of an episode. TD methods combine the sampling from MC methods and the bootstrapping of DP. They differ from DP methods in that they use sample updates and do not use expected updates based on a complete distribution of all possible subsequent states and rewards. The estimates used in TD methods have significantly lower variance than MC methods, as they do not use complete returns for estimates. The drawback is that TD methods are biased and initial estimates may be very wrong. A tabular one-step TD algorithm for policy evaluation is shown in Algorithm 3.

4.3.2 SARSA for model-free control

In the previous section, we explained how TD methods can be used to estimate the value function v_π from experience that follows a policy π. We now review a model-free control TD method, in order to obtain optimal value functions and optimal policies. According to Sutton and Barto [45], the idea of TD learning methods is that the approximate policy and approximate value function can interact with each other in such a way that they both move towards their optimal values. The first part of this process is to estimate the value

Stellenbosch University https://scholar.sun.ac.za

Algorithm 3: Tabular one-step TD for evaluating a policy π ($V \approx v_\pi$), adapted from Sutton and Barto [45].

```
1  input: a policy $\pi$ to be evaluated
2  initialise: value function $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}^+$, $V(\text{terminal}) = 0$
3  initialise: step size $\alpha \in (0, 1]$
4  for each episode do
5      initialise $S_0$
6      repeat for each step $t$ of episode, $t = 0$
7          choose $A_t$ from $S_t$ using policy $\pi$
8          perform $A_t$, observer $R_{t+1}$, $S_{t+1}$
9          $V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$
10     until $S_{t+1}$ is terminal
11 end
```

function to predict the returns of the current policy. The second part is to improve the policy with regard to the estimated value function.

In this section, we review an on-policy TD control method, *SARSA*. Again the action-value function q_π is used instead of the state-value function v_π, as the former does not require a transition model of the environment to improve the policy (refer to Section 4.2.2). The goal is to obtain an optimal action-value function q_* and therefore also an optimal policy π_*. We first discuss how a policy π is evaluated, and then how the policy is improved.

Similarly to the previous section, we learn a value function from samples of experience. Here state-action pair to state-action pair is considered, and the estimated values of state-action pairs are updated. The estimated action-value $Q(S_t, A_t)$ is updated:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]. \tag{4.10}$$

The update is done for non-terminal states, and if state S_{t+1} is terminal then $Q(S_{t+1}, A_{t+1})$ is defined as zero.

The estimated action-value function Q is then used to improve the policy. An ε-greedy strategy can be used to choose the action A_{t+1} from the state S_{t+1} using the estimated action-value function Q. The ε-greedy strategy acts mostly greedy with regard to the estimated action-value function but also explores nongreedy actions from time to time, as discussed in Section 4.2.3. In every update $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ are present and these give rise to the name SARSA. SARSA for on-policy control is shown in Algorithm 4. This method continually estimates q_π while doing an ε-greedy improvement. According to Sutton and Barto [45], SARSA with ε-greedy exploration converges to an optimal action-value function q_* and therefore an optimal policy π_*. For convergence, all state-action pairs must be visited an infinite number of times, and the policy must converge to the greedy policy in the limit (by setting $\varepsilon = 1/t$). In practice, it is not possible to visit all states an infinite

Stellenbosch University https://scholar.sun.ac.za

Algorithm 4: SARSA: An on-policy TD control algorithm for estimating the optimal action-value function $Q \approx q_*$, adapted from Sutton and Barto [45].

```
1  initialise $\alpha \in (0, 1]$
2  initialise small $\varepsilon > 0$
3  initialise $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily, $Q(\text{terminal}, \cdot) = 0$
4  for each episode do
5      initialise $S_0$
6      choose $A_0$ from $S_0$ using policy derived from $Q$ (e.g. $\varepsilon$-greedy)
7      repeat for each step $t$ of episode, $t = 0$
8          take action $A_t$, observe $R_{t+1}$, $S_{t+1}$
9          choose $A_{t+1}$ from $S_{t+1}$ using policy derived from $Q$ (e.g. $\varepsilon$-greedy)
10         $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$.
11     until $S_{t+1}$ is terminal
12 end
```

number of times, but it is also not necessary to converge all the way towards q_* in order to obtain an optimal policy π_*.

4.3.3 Q-learning for model-free control

Q-learning is an off-policy TD method which Sutton and Barto [45] describe as a breakthrough in RL. It is classified as an off-policy method as the behaviour policy is independent of the target policy. The target policy (the policy the agent learns about) is greedy with respect to $Q(s, a)$:

$$\pi(S_{t+1}) = \underset{a}{\operatorname{argmax}} Q(S_{t+1}, a). \tag{4.11}$$

The Q-learning target can be simplified to

$$\begin{aligned} G_t &\approx R_{t+1} + \gamma Q\Big(S_{t+1}, \underset{a}{\operatorname{argmax}} Q(S_{t+1}, a)\Big) \\ &= R_{t+1} + \gamma \max_a Q(S_{t+1}, a). \end{aligned} \tag{4.12}$$

Sutton and Barto [45] define the Q-learning update step as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]. \tag{4.13}$$

The behaviour policy can be anything, but it is usually ε-greedy with respect to $Q(s, a)$. The behaviour policy can also take the form of experience generated by older policies or by expert demonstrators. The off-policy nature of the algorithm allows it to be more sample-efficient as it can reuse previously generated experience to learn. This also makes it possible to learn from several different policies at the same time. Algorithm 5 shows pseudo code for a tabular Q-learning model-free control algorithm. Sutton and Barto [45] state that with Q-learning, the estimate action-value function Q converges to the optimal action-value function q_* when all state-action pairs are infinitely updated. Again, in practice it is not necessary to converge all the way to q_* to obtain an optimal policy π_*.

Stellenbosch University https://scholar.sun.ac.za

Algorithm 5: Q-learning: An off-policy TD control algorithm for estimating $\pi \approx \pi_*$, adapted from Sutton and Barto [45].

```
1  initialise $\alpha \in (0, 1]$
2  initialise small $\varepsilon > 0$
3  initialise $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, randomly except that $Q(\text{terminal}, \cdot) = 0$
4  for each episode do
5      initialise $S_0$
6      repeat for each step $t$ of episode, $t = 0$
7          choose $A_t$ from $S_t$ using policy derived from $Q$ (e.g. $\varepsilon$-greedy)
8          take action $A_t$, observe $R_{t+1}$, $S_{t+1}$
9          $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big]$.
10     until $S_{t+1}$ is terminal
11 end
```

Sutton and Barto [45] state that optimal performance exists when combining TD methods and MC methods. In the next section we explore how the n-step update can be applied to unify TD methods and MC methods.

4.3.4 n-step temporal-difference methods

Having discussed MC methods and TD methods, we now discuss n-step TD methods. n-step methods cover a spectrum with MC methods on one end and one-step TD methods on the other. Sutton and Barto [45] state that the best method is often between the two extremes.

A problem with one-step TD methods is that bootstrapping is done after one time step. One time step is often too short for the state to significantly change. According to Sutton and Barto [45], bootstrapping works best if a recognisable state change has occurred. n-step methods enable bootstrapping after multiple time steps, hence allowing the state to change more significantly.

n-step methods are still TD methods since one estimate is updated using a later estimate. The later estimate is now n time steps, instead of one time step, later, as explained before. We call these methods *n-step TD methods.*

Recall that MC methods update the estimate $v_\pi(S_t)$ in the direction of the complete return of an episode, hence

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T, \tag{4.14}$$

where T is the last time step of the episode. The return G_t is the target used for MC methods. The target used in one-step TD methods is the immediate reward plus the discounted estimated value of the next state, hence

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}). \tag{4.15}$$

Stellenbosch University https://scholar.sun.ac.za

We refer to Equation 4.15 as the *one-step return*. We use the same subscripts on $G_{t:t+1}$ as Sutton and Barto [45]. This indicates that the discounted sum of rewards until time step $t+1$ is computed, and the rest of the terms of the full return in Equation 4.14 are replaced by the estimate $\gamma V_t(S_{t+1})$. The n-step return is somewhere between the extremes of Equation 4.14 and Equation 4.15. It is defined by Sutton and Barto [45] as

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \tag{4.16}$$

for all n, t such that $n \geq 1$ and $0 \leq t < T - n$. If the n-step return extends beyond the terminal state at time step T $(t + n \geq T)$, then the n-step return becomes the full return and the missing terms become zero, i.e. $G_{t:t+n} \doteq G_t$ if $t + n \geq T$. Note that the n-step return involves future rewards and states that are not available at the time of generating the state for which the estimated value is being updated. One must therefore wait until R_{t+n} and S_{t+n} are available and V_{t+n-1} is computed. These values are available at time step $t + n$.

The n-step for SARSA is very similar to Equation 4.16, but is now defined in terms of estimated action values. Sutton and Barto [45] define the n-step return for SARSA as

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \tag{4.17}$$

where $n \geq 1$ and $0 \leq t < T - n$. Again the n-step return is equal to the return $G_{t:t+n} \doteq G_t$ if $t + n \geq T$. An update to an estimate of a state-action value is then

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha\Big[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\Big], \tag{4.18}$$

where $0 \geq t < T$.

The n-step return can also be extended to off-policy algorithms. Off-policy learning entails learning from a policy π while following a different policy b. The policy π is often the greedy policy, while b is an exploratory policy. Sutton and Barto [45] state that to use the data from b to learn from π, we must take the difference of these policies into account. They suggest using *importance-sampling* (IS) weights to weight n-step updates. The weights are computed using the relative probabilities of π and b performing the actions that were previously taken. We do not cover the theory of off-policy n-step learning with IS here. The reason for this is that Hernandez-Garcia and Sutton [13] have already shown that IS is not always necessary when learning off-policy.

A logical next step would be to introduce eligibility traces. Sutton and Barto [45] use n-step methods as a stepping stone to introduce eligibility traces. According to Sutton and Barto [45], eligibility traces have significant computational advantages over n-step methods. However, using eligibility traces with the algorithm of this study would be very cumbersome. We further elaborate on the reasons for the difficulty of using eligibility traces in Chapter 6. Therefore we do not cover the theory of eligibility traces here.

Stellenbosch University https://scholar.sun.ac.za

4.4 Value function approximation

Up to this point, we discussed tabular learning, where a table was used to represent the value function. Each state or state-action pair has an associated value in the table. This data structure allows for updating the estimated values independently from each other. The lookup table made it easier to understand the concepts that were explained earlier and can be applied with great success to simple problems with relatively small state spaces and action spaces. Unfortunately, many real-world problems have vast state spaces and action spaces. For example, one of the objectives of this study, as discussed in Section 1.5, requires an agent to make decisions based on an RGB image in order to achieve the goal of delivering a first-aid kit to an immobilised miner. The problem is that a camera can produce more possible images than there are atoms in the universe [45]. A problem with such a large state space is that it is practically impossible to have enough memory to store a value for each possible state or state-action pair. Moreover, as there are so many states, most of the time, the agent will encounter new states that it has never seen before. The agent, therefore, rarely revisits states and thus, the estimated values of previously visited states are rarely updated. It is argued by Sutton and Barto [45] that even with supercomputational resources, we cannot expect to find solutions to problems such as the problem proposed in this thesis using tabular methods.

The solution lies in interpolation and approximation – being able to use a small subset of states to generalise and make decisions over a much larger subset [45]. In this section, we discuss function approximation, with the intention to combine the previously discussed model-free control methods with function approximation. The value function is now rather approximated by a function parameterised with a vector of weights $v_\pi(s) \approx \hat{v}(s; \mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^d$ and d is the number of weights. For example, if $\hat{v}$ is a linear function in features of the state, then $\mathbf{w}$ is the vector of feature weights [45]. If the function $\hat{v}$ is a more general *artificial neural network* (ANN), then $\mathbf{w}$ is the vector of connection weights in all the layers. Using function approximation allows for better generalisation, i.e. when a single state is updated, the change made to the weight vector also affects the value of other surrounding states. This means more effective learning, but also makes it more challenging to understand and manage the system. In the rest of this section, we show how the previously discussed tabular techniques can be combined with function approximation. We also discuss the difficulties of combining off-policy methods with function approximation.

4.4.1 Stochastic gradient descent

According to Sutton and Barto [45] *stochastic gradient descent* (SGD) is one of the most popular function approximation methods and works well with online RL. Gradient-descent methods aim to approximate $\hat{v}(s; \mathbf{w}) \approx v_\pi(s)$, where $\mathbf{w}$ is a column vector with a fixed number of real valued components, i.e $\mathbf{w} \doteq [w_1, w_2, \dots, w_d]^\top$. The function $\hat{v}(s; \mathbf{w})$ is a

Stellenbosch University https://scholar.sun.ac.za

differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$. We adjust the weight vector $\mathbf{w}$ at discrete time steps and we indicate the weight vector at the time step t as $\mathbf{w}_t$. Usually we do not have direct access to $v_\pi(s)$. For now assume at each time step we observe an example state S_t and the true value $v_\pi(S_t)$ under the policy π.

The objective function we minimise is the mean of the squared differences, defined as

$$J(\mathbf{w}) = \mathbb{E}\Big[\Big(v_\pi(s) - \hat{v}(s;\mathbf{w})\Big)^2\Big]. \tag{4.19}$$

We refer to Equation 4.19 as the *mean square error* (MSE). Sutton and Barto [45] state that this is not necessarily the best objective function to minimise to find the best value function. However, since it is not clear what would be a better objective function, we continue with the MSE.

The gradient $\nabla f(\mathbf{w})$ for any function $f(\mathbf{w})$ denotes the partial derivatives with respect to the components of the weight vector, i.e.

$$\nabla f(\mathbf{w}) \doteq \left[\frac{\partial f(\mathbf{w})}{w_1}, \frac{\partial f(\mathbf{w})}{w_2}, \dots, \frac{\partial f(\mathbf{w})}{w_d}\right]^\top. \tag{4.20}$$

Gradient-descent methods entail adjusting the weight vector in the direction of the negative gradient of the objective function. We use the examples to update the weight vector, rather than calculating the full expectation over all states. We assume that the states appear in examples with the same distribution as the expectation in Equation 4.19. We now adjust the weight vector in the direction to minimise the error on the example, hence

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}_t}\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]^2 \\ &= \mathbf{w}_t + \alpha\Big(v_\pi(S_t) - \hat{v}(S_t;\mathbf{w}_t)\Big)\nabla_{\mathbf{w}_t}\hat{v}(S_t;\mathbf{w}_t),\end{aligned} \tag{4.21}$$

where α is a positive step-size parameter referred to as the *learning rate*. Although in some cases we can move all the way in this direction and eliminate the error on the example, it is not advised by Sutton and Barto [45]. The reason for this is that if we completely correct the error for one example, we may increase the error on another example. We rather only take a small step in the direction of the negative gradient to find an approximation that balances the errors for all states.

SGD methods are called *stochastic*, as samples are stochastically selected and used to update the weight vector $\mathbf{w}$. By using the process in Equation 4.21 on many examples, the MSE in Equation 4.19 is minimised.

4.4.2 Temporal-difference methods with function approximation

In this section, we combine the previously discussed *temporal-difference* (TD) control methods with function approximation. We again rather use the action-value function than the state-value function as no model is required to make greedy policy improvements with

Stellenbosch University https://scholar.sun.ac.za

regard to the value function. The action-value function is therefore now approximated with a parametric function as

$$\hat{q}(s,a;\mathbf{w}) \approx q_*(s,a), \text{ where } \mathbf{w} \in \mathbb{R}^d. \tag{4.22}$$

According to Sutton and Barto [45], extending function approximation to off-policy methods is significantly harder than to on-policy methods. Therefore we first explore how to combine function approximation with the on-policy method, SARSA.

We use Equation 4.21 to update the weight vector $\mathbf{w}$ at time step t. As we are using the action-value function instead of the state-value function, Equation 4.21 becomes

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\Big(q_\pi(S_t,A_t) - \hat{q}(S_t,A_t;\mathbf{w}_t)\Big)\nabla_{\mathbf{w}_t}\hat{q}(S_t,A_t;\mathbf{w}_t). \tag{4.23}$$

We do not have access to an oracle or to the true value function q_π in Equation 4.23, and therefore we substitute q_π with the TD target, $R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1};\mathbf{w}_t)$. According to Sutton and Barto [45], a one-step SARSA update to the weight vector is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1};\mathbf{w}_t) - \hat{q}(S_t,A_t;\mathbf{w}_t)\Big]\nabla_{\mathbf{w}_t}\hat{q}(S_t,A_t;\mathbf{w}_t). \tag{4.24}$$

We now adapt the SARSA algorithm in Algorithm 4 to incorporate function approximation instead of tabular learning. SARSA with function approximation is shown in Algorithm 6.

Algorithm 6: SARSA with function approximation for estimating $\hat{q} \approx q_*$, adapted from Sutton and Barto [45].

```
input: a differentiable action-value function parameterisation $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$
initialise: $\alpha > 0$
initialise: $\varepsilon > 0$
initialise: action-value function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily
for each episode do
    $S_0, A_0 \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
    repeat for each step $t$ of episode, $t = 0$
        take action $A_t$ based on $S_t$, observe $R_{t+1}, S_{t+1}$
        if $S_{t+1}$ is terminal then
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big[R - \hat{q}(S_t,A_t;\mathbf{w})\Big]\nabla_{\mathbf{w}}\hat{q}(S_t,A_t;\mathbf{w})$
            go to next episode
        end
        choose $A_{t+1}$ as a function of $\hat{q}(S_t,\cdot;\mathbf{w})$ (e.g., $\varepsilon$-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big[R + \gamma\hat{q}(S_{t+1},A_{t+1};\mathbf{w}) - \hat{q}(S_t,A_t;\mathbf{w})\Big]\nabla_{\mathbf{w}}\hat{q}(S_t,A_t;\mathbf{w})$
    until $S_{t+1}$ is terminal
end
```

Off-policy learning with function approximation is more challenging than on-policy learning due to higher variance. Variance is higher when using off-policy learning, as the behavioural policy is different from the target policy. In fact, Sutton and Barto [45] go so far as to refer to bootstrapping, off-policy learning and function approximation together as the *deadly triad*, since when these aspects are combined, instability is very likely.

Stellenbosch University https://scholar.sun.ac.za

4.5 Summary

In this chapter, we first gave an overview of tabular model-free control methods. The two main methods that we discussed are TD methods and MC methods. As TD methods allow online learning, they are preferred to MC methods which depend on samples of complete episodes.

Unfortunately, tabular control methods do not scale well to solve complex problems with large state spaces. Therefore we have discussed how function approximation could be used to approximate value functions. We mentioned that it is challenging to combine off-policy methods, bootstrapping and value function approximation. Accordingly we discussed how function approximation could be incorporated only in the on-policy method, SARSA. In Chapter 6 we discuss deep Q-learning as introduced by Mnih *et al.* [24] – a Q-learning algorithm that successfully utilises function approximation by means of an *artificial neural network* (ANN). Before we discuss the deep Q-learning algorithm, we discuss the *artificial neural network* (ANN) – a very powerful tool for approximating non-linear functions.

Stellenbosch University https://scholar.sun.ac.za

Chapter 5

Artificial neural networks

At the end of the previous chapter, we discussed that approximating the value function for problems with large state spaces allows for better generalisation and thus for more effective learning. Recall that for the decision-making problem we consider in this study, the observations are in the form of high-dimensional *red green blue* (RGB) camera images. Therefore, the value function must be approximated. Since the observations are high-dimensional, it is essential to extract appropriate features in order to approximate the required value function. The problem is that manual feature extraction from complex structures such as images can be challenging, and consequently, traditional computer vision techniques will be difficult to use [30]. For this reason, we investigate the *artificial neural network* (ANN), a powerful non-linear function approximator that is inspired by the biological neural network. Ng and Katanforoosh [28] describe the ANN as an end-to-end learning technique, as it only requires the input features $\mathbf{x}$ and the output y to learn intermediate features by itself. To learn these features, we must provide the ANN with enough training examples [28]. According to Ng and Katanforoosh [28], the ANN can sometimes discover complex features which are difficult for humans to understand.

In this chapter, we discuss ANNs in the context of solving regression problems using supervised learning. Regression entails predicting continuous values such as temperatures, house prices, and in our case action values. Supervised learning entails training an ANN to map inputs to desired outputs. A training dataset that contains example inputs with associated labelled outputs is used to train the ANN. The goal is to obtain a general function that is also capable of making accurate predictions on unseen inputs.

We start with a discussion of the *feed-forward neural network* (FNN) to introduce concepts such as pre-activation and activation functions, forward propagation, loss functions, backpropagation and batch methods. Unfortunately, FNNs perform poorly at approximating functions from data in the form of 2D arrays such as images. We therefore also discuss the *convolutional neural network* (CNN). The latter is more suitable for automatically extracting features from these types of data structures. This chapter is primarily based on the work of Goodfellow *et al.* [11] and Ng and Katanforoosh [28].

Stellenbosch University https://scholar.sun.ac.za

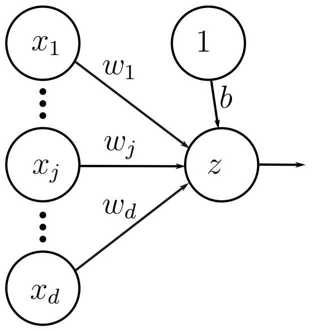

Figure 5.1: An illustration of a single neuron. The neuron receives an input vector $\mathbf{x} = [x_1, \ldots, x_d]$, where d is the dimension of the input. Each entry of the input vector $(x_1, \ldots, x_d)$ is connected to the output of the neuron, indicated by the arrows. Each arrow has a value associated with it; we refer to these values as the *weights*. The weights of the neuron are represented by the vector $\mathbf{w} = [w_1, \ldots, w_d]$. Additionally, the neuron also has a parameter that we refer to as the bias b. The function $h(\mathbf{x})$ determines the output of the neuron and consists of two operations, namely the pre-activation function and the activation function. The pre-activation step entails calculating a weighted linear combination of the input and adding the bias to obtain a scalar result (shown in Equation 5.2). A non-linear transformation, known as the activation function, is then performed on the result to obtain the output of the neuron (shown in Equation 5.3).

5.1 Feed-forward neural networks

Goodfellow *et al.* [11] state that the goal of an FNN, as with other ANNs, is to approximate some function $f(\mathbf{x})$. These models are called *feed-forward* because there are no loops in the network and therefore outputs cannot influence inputs. FNNs form the basis of many other variants of the ANN.

If the goal is to approximate $f(\mathbf{x})$ in order to do regression where an input $\mathbf{x}$ is mapped to a value y, then the FNN defines a mapping $y = \hat{f}(\mathbf{x}; \boldsymbol{\theta})$ and learns the parameters $\boldsymbol{\theta}$ that result in the best function approximation $\hat{f}(\mathbf{x}; \boldsymbol{\theta}) \approx f(\mathbf{x})$ [11]. The problem is that we do not have direct access to the function $f(\mathbf{x})$, but usually we have a training dataset

$$\mathcal{D} = \left\{(\mathbf{x}^{[1]}, y^{[1]}), (\mathbf{x}^{[2]}, y^{[2]}), \ldots, (\mathbf{x}^{[n]}, y^{[n]})\right\} \tag{5.1}$$

that evaluates the function $f(\mathbf{x})$ at inputs $\mathbf{x}$ to produce associated outputs y. We therefore know what outputs the network should yield for the sample inputs in the training dataset. Before we discuss how the training dataset is used to adjust the parameters of the FNN approximate $f(x)$, we first discuss how to evaluate an input provided to the FNN. We start by looking at the single neuron to introduce FNNs and then use this as a stepping stone to discuss more complex deep neural networks.

5.1.1 Single neuron

The single neuron, shown in Figure 5.1, is the most basic form of an FNN. The neuron consists of a scalar bias b, a weight vector $\mathbf{w} \in \mathbb{R}^d$, and receives an input $\mathbf{x} \in \mathbb{R}^d$, where d is the dimension of the input. For now, we assume that the values of $\mathbf{w}$ and b are given.

Stellenbosch University https://scholar.sun.ac.za

We now discuss the *forward pass*, the process of evaluating an input. We first discuss evaluation of an input using a single neuron. In the next section, we discuss evaluating an input using a DNN, which consists of multiple neurons.

The neuron performs two functions on the input, namely the pre-activation and the activation step, and produces a scalar output. The *pre-activation* function $a(\mathbf{x})$ is defined by Ng and Katanforoosh [28] as a weighted linear combination of the input $\mathbf{x}$ plus some offset b, i.e.

$$a(\mathbf{x}) = b + \mathbf{w}^\top \mathbf{x}. \tag{5.2}$$

The *activation* function then performs a non-linear transformation on the result of the pre-activation function, namely

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \mathbf{w}^\top \mathbf{x}). \tag{5.3}$$

Ng and Katanforoosh [28] state that without performing the non-linear transformation in Equation 5.3, the ANN performs linear regression. The ANN will therefore not be able to approximate non-linear functions. Three popular examples of non-linear activation functions are sigmoid

$$g(a) = \operatorname{sigm}(a) = \frac{1}{1 + e^{-a}}, \tag{5.4}$$

tanh

$$g(a) = \tanh(a) = \frac{e^{2a} - 1}{e^{2a} + 1}, \tag{5.5}$$

and *rectified linear unit* (ReLU)

$$g(a) = \operatorname{ReLU}(a) = \max(0, a). \tag{5.6}$$

The ReLU activation function is the default recommendation by Goodfellow *et al.* [11] and is therefore the activation function of choice for this study. Unfortunately, only relatively simple functions can be represented by a single neuron. We therefore shift our attention to *deep neural network*s (DNNs) to represent more complex functions.

5.1.2 Deep neural networks

Ng and Katanforoosh [28] state that multiple neurons can be connected to form a *network*, so that one neuron's output forms the input to the next neuron. This results in a more complex network, representing a more complex function [28]. Goodfellow *et al.* [11] describe such a network as combining different functions. A directed acyclic graph is usually used to describe the way the functions of the network are combined. For example, three functions $f^{(1)}, f^{(2)}, f^{(3)}$ can be chained together to form a network with three layers $\hat{f}(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. The input $\mathbf{x}$ is located at layer zero, as we use zero indexing. The function $f^{(1)}$ is known as the first layer, $f^{(2)}$ is known as the second layer, and so forth. The output layer is the last layer of the network and in this example is $f^{(3)}$. Notation

Stellenbosch University https://scholar.sun.ac.za

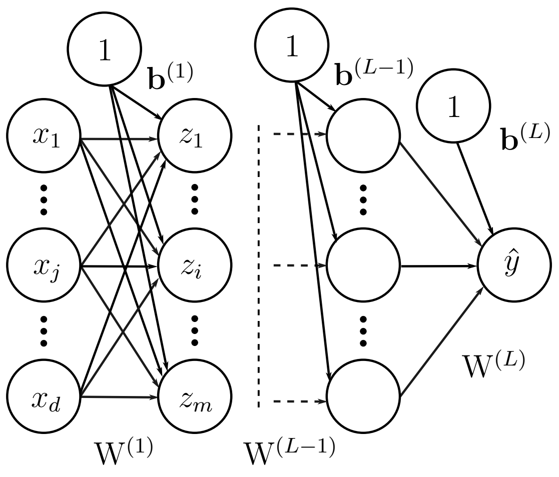

Figure 5.2: Illustration of a deep FNN with L layers. The FNN receives an input $\mathbf{x} = [x_1, \dots, x_d]$. The output of the first layer is represented by the vector $\mathbf{z} = [z_1, \dots, z_m]$. The connections between the neurons of each layer i are indicated with arrows and are associated with the weights $W^{(i)}$. Similarly each layer i contains a bias vector $\mathbf{b}^{(i)}$ and connections are indicated with arrows. The network produces an output $\hat{y}$.

similar to Ng and Katanforoosh [28] is used, where the superscript (i) indicates the ith layer of the network. The number of layers in the network determines the *depth* of the network.

A network with multiple layers can be observed in Figure 5.2. The network receives an input vector

$$\mathbf{x} = [x_1, \dots, x_d]^\top. \tag{5.7}$$

The input layer is followed by multiple layers referred to as *hidden* layers that are indicated by $\mathbf{h}$. In this example, all neurons of the input layer are connected to all neurons of the first hidden layer. Such a layer is known as a fully connected layer. We refer to the last layer of the network as the output layer and it is indicated by o.

Similar to the single neuron, the connections (arrows) between the neurons of different layers have associated values, referred to as weights. For example, the weights of layer one of the network in Figure 5.2 are defined as

$$\underbrace{W^{(1)}}_{m\times d} = \begin{bmatrix} w_{11} & \dots & w_{1d} \\ \vdots & \ddots & \\ w_{m1} & & w_{md} \end{bmatrix}, \tag{5.8}$$

where m is the dimension of the output and d is the dimension of the input of the first layer. Each layer also contains a bias vector, for example the bias vector of layer one is defined as

$$\mathbf{b}^{(1)} = [b_1, \dots, b_m]^\top. \tag{5.9}$$

The weights and biases of all the layers in the network make up the parameters $\boldsymbol{\theta}$ of the network.

Stellenbosch University https://scholar.sun.ac.za

When training such a network, the parameters $\boldsymbol{\theta}$ of the network $\hat{f}(\mathbf{x};\boldsymbol{\theta})$ are adjusted to approximate $f(\mathbf{x})$. A training algorithm is responsible for deciding the values of the parameters of the different layers of the network in order to achieve the desired outputs at the end of the network. The intermediate layers are known as the *latent* or *hidden* layers, as we do not directly specify what their outputs should be. Before we continue the discussion on how the network parameters $\boldsymbol{\theta}$ are trained, we first discuss forward propagation in a deep FNN.

5.1.3 Forward propagation

We now discuss the process of evaluating an input provided to the deep FNN, i.e. forward propagation. The process is shown in Algorithm 7. We first calculate the output of the

Algorithm 7: Evaluating an input $\mathbf{x}$ to an FNN with forward propagation, adapted from the work by Goodfellow *et al.* [11]. For simplicity, this algorithm computes the predicted output for a single input example $\mathbf{x}$. Practical applications usually rather use a mini-batch.

```
1 input: an input $\mathbf{x}$
2 input: ANN parameters $\{\mathrm{W}, \mathbf{b}\} \in \boldsymbol{\theta}$
3 input: network depth, $L$
4 $\mathbf{h}^{(0)} \leftarrow \mathbf{x}$
5 for $k = 1, \dots, L$ do
6     $\mathbf{a}^{(k)} \leftarrow \mathbf{b}^{(k)} + \mathrm{W}^{(k)}\mathbf{h}^{(k-1)}$
7     $\mathbf{h}^{(k)} \leftarrow \mathbf{g}(\mathbf{a}^{(k)})$
8 end
9 $\hat{y} \leftarrow \mathbf{h}^{(L)}$
```

first layer of the network, that is the vector $\mathbf{z}$ in Figure 5.2. To calculate the output of the first layer, we first perform pre-activation step

$$\underbrace{\mathbf{a}^{(1)}(\mathbf{x})}_{m\times 1} = \underbrace{\mathbf{b}^{(1)}}_{m\times 1} + \underbrace{\mathrm{W}^{(1)}}_{m\times d}\underbrace{\mathbf{x}}_{d\times 1}. \tag{5.10}$$

We then proceed with the activation step to calculate the output of the first hidden layer. The result of Equation 5.10 is used as the input to the activation function of the first hidden layer

$$\mathbf{h}^{(1)}(\mathbf{x}) = \underbrace{\mathbf{z}}_{m\times 1} = \underbrace{\mathbf{g}\Big(\mathbf{a}^{(1)}(\mathbf{x})\Big)}_{m\times 1}. \tag{5.11}$$

It is clear that the output of any layer in the network is dependent on the output of the previous layer. We therefore define more general equations to calculate the output at any layer of the network. The output of any layer k in the network can be calculated by first calculating the pre-activation, i.e.

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathrm{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x}), \tag{5.12}$$

Stellenbosch University https://scholar.sun.ac.za

which depends on the output of the previous layer $\mathbf{h}^{(k-1)}(\mathbf{x})$. The activation function of layer k is then applied to the result of Equation 5.12 to obtain the output of layer k. Therefore

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}\Big(\mathbf{a}^{(k)}(\mathbf{x})\Big). \tag{5.13}$$

These operations are repeated for each layer to obtain the predicted output $\hat{y}$. The output $\hat{y}$ is the result at the output layer o and is calculated similar to Equation 5.12 and Equation 5.13,

$$\hat{y} = o^{(L)}(\mathbf{x}) = \mathbf{g}\Big(b^{(L)} + \mathrm{W}^{(L)}\mathbf{h}^{(L-1)}(\mathbf{x})\Big). \tag{5.14}$$

We have now discussed how to evaluate an input to the network if the parameter values $\boldsymbol{\theta}$ of the network are known. We still have to discuss how to obtain optimal parameters $\boldsymbol{\theta}$ for the network.

5.1.4 Loss and cost functions

Before we discuss backpropagation, we give a quick overview of loss and cost functions. The aim is to minimise a loss function $\mathcal{L}(\hat{y}, y)$ over all entries in the dataset. The loss function surrogates the difference between the predicted output $\hat{y}$ and the target y with a differentiable function. The loss function usually takes a single data sample as an argument. We define the cost function as the mean of the loss function over all data samples in the dataset, hence

$$J(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(\hat{y}^{[i]}, y^{[i]}), \tag{5.15}$$

where n is the number of samples in the dataset. Since the loss function and thus the cost function are differentiable, the partial derivative of the loss or cost function with respect to the network parameters can be calculated – this is essential when using gradient-based optimisation techniques such as SGD.

The process of minimising the cost function is referred to as *full gradient descent*. Full gradient descent uses the entire dataset to compute the network gradients and according to Ng and Katanforoosh [28] it leads to more accurate gradients. On the other hand, the process of minimising the loss function using single examples is referred to as *stochastic gradient descent* (SGD). It uses single samples in an attempt to approximate gradients from full gradient descent, but this results in noisy gradients. According to Ng and Katanforoosh [28], it can be challenging to do a full gradient descent update; they suggest an alternative method called mini-batch gradient descent. Mini-batch gradient descent is a compromise between SGD and gradient descent. It entails, as the name suggests, the use of small batches of samples to estimate gradients in order to update the parameters of the network. Ng and Katanforoosh [28] define the mini-batch cost function J_{mb} as

$$J_{mb}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{B}\sum_{i=1}^{B}\mathcal{L}^{(i)}(\hat{y}^{[i]}, y^{[i]}), \tag{5.16}$$

Stellenbosch University https://scholar.sun.ac.za

where $\mathcal{L}^{(i)}$ is the loss for a single sample and B is the number of samples in the batch. In practice, mini-batch gradient descent is found to work more effectively than SGDt [28]. We next discuss the most popular loss and cost functions used for regression.

Mean square error The *square loss* is a frequently used regression loss function. It measures the squared difference between a predicted value and a target value. Wang *et al.* [49] define the square loss as

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2. \tag{5.17}$$

The square loss result is always positive, and a perfect prediction will have a result of 0. The gradient of the square loss is variable. The gradient is larger for samples with large errors and decreases if the error is close to 0. According to Wang *et al.* [49], this property of the square loss allows for fast convergence to optimal parameters and is also beneficial to the accuracy of the model. However, the square loss is more sensitive to outliers as gradients may be very large. The MSE cost function is derived from the square loss. The MSE calculates the mean of the squared differences between the predicted values and the target values of a batch of samples. The MSE is defined as

$$J_{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{B}\sum_{i=0}^{B}(\hat{y}^{[i]} - y^{[i]})^2, \tag{5.18}$$

where B is the number of samples in the batch.

Mean absolute error The *absolute loss* is another popular regression loss function. It measures the absolute error between a predicted value and a target value. Wang *et al.* [49] define the absolute loss as

$$\mathcal{L}(\hat{y}, y) = |\hat{y} - y|. \tag{5.19}$$

The absolute loss is less sensitive to outliers as the gradient is not variable like the square loss. The problem with the absolute loss is that the gradient is not smooth when the error is zero, and therefore it is not as popular as the square loss. The *mean absolute error* (MAE) cost function calculates the mean of the absolute differences between the predicted values and the target values of a batch of samples. The MAE is defined as

$$J_{MAE}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{B}\sum_{i=0}^{B}|\hat{y}^{[i]} - y^{[i]}|, \tag{5.20}$$

where B is the number of samples in the batch.

Huber loss We finally discuss the *Huber loss* – the loss function of choice for this study. The Huber loss combines the advantages of square loss and absolute loss. It has the benefit of fast convergence like the square loss, but is also resilient to outliers like the absolute

Stellenbosch University https://scholar.sun.ac.za

loss. Wang *et al.* [49] define the Huber loss as

$$\mathcal{L}(\hat{y}, y) = \begin{cases} \frac{1}{2}(\hat{y} - y)^2 & \text{if } |\hat{y} - y| < \delta \\ \delta|\hat{y} - y| - \frac{1}{2}\delta^2 & \text{otherwise,} \end{cases} \tag{5.21}$$

where δ is a parameter that acts as a boundary to decide whether a sample is an outlier. We again use Equation 5.16 to obtain the Huber cost function.

5.1.5 Backpropagation

Ng and Katanforoosh [28] state that for any given layer k, we have to update the weights

$$\mathrm{W}^{(k)} \leftarrow \mathrm{W}^{(k)} - \alpha \frac{\partial J}{\partial \mathrm{W}^{(k)}}, \tag{5.22}$$

and the biases

$$\mathbf{b}^{(k)} \leftarrow \mathbf{b}^{(k)} - \alpha \frac{\partial J}{\partial \mathbf{b}^{(k)}}, \tag{5.23}$$

where α is a small step size we refer to as the learning rate. We therefore have to obtain the partial derivative of the cost function with respect to the weights and biases of each layer of the ANN.

Obtaining an analytical expression for the gradients of an ANN is not too difficult, but can become computationally expensive to evaluate. We now come to the backpropagation algorithm – an efficient way to calculate the gradients of a DNN. Backpropagation allows information about the cost function to flow backwards through the network to efficiently calculate the network gradients. Once we have computed the gradients of the network, a gradient-based optimisation technique such as SGD can be used to adjust the parameters $\boldsymbol{\theta}$ of the network.

We now explain the backpropagation procedure; refer to Figure 5.2. We first calculate the gradient on the output of the last layer, i.e. the output layer. We then use the chain rule to transfer the gradient to the activation of the output layer, hence

$$\frac{\partial J}{\partial a^{(L)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a^{(L)}}. \tag{5.24}$$

By also using the chain rule, we now are able to obtain the gradients on the weights and biases of the output layer. The gradients on the weights of the output layer are obtained with

$$\frac{\partial J}{\partial \mathrm{W}^{(L)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a^{(L)}} \times \frac{\partial a^{(L)}}{\partial \mathrm{W}^{(L)}}. \tag{5.25}$$

The gradients on the biases of the output layer can be obtained in a similar way:

$$\frac{\partial J}{\partial \mathbf{b}^{(L)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a^{(L)}} \times \frac{\partial a^{(L)}}{\partial \mathbf{b}^{(L)}}. \tag{5.26}$$

We apply the chain rule again to transfer the gradients to the output of the previous layer. The process then repeats, i.e. the gradients on the activation of the layer are computed,

Stellenbosch University https://scholar.sun.ac.za

followed by the gradients on the weights and biases of the layer. We here define more general equations to obtain the gradients on the weights and biases of any layer k of the network. The gradients on the weights of the layer k are obtained with

$$\frac{\partial J}{\partial \mathrm{W}^{(k)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a^{(L)}} \times \frac{\partial a^{(L)}}{\partial \mathbf{h}^{(L-1)}} \times \cdots \times \frac{\partial \mathbf{h}^{(k)}}{\partial \mathrm{a}^{(k)}} \times \frac{\partial \mathbf{a}^{(k)}}{\partial \mathrm{W}^{(k)}}. \tag{5.27}$$

Similarly, the gradients on the biases of the layer k are obtained with

$$\frac{\partial J}{\partial \mathbf{b}^{(k)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a^{(L)}} \times \frac{\partial a^{(L)}}{\partial \mathbf{h}^{(L-1)}} \times \cdots \times \frac{\partial \mathbf{h}^{(k)}}{\partial \mathrm{a}^{(k)}} \times \frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{b}^{(k)}}. \tag{5.28}$$

We summarise the backpropagation process with Algorithm 8. It can compute the gradients

Algorithm 8: Computing the gradients of the activations $\mathbf{a}^{(k)}$ of each layer k. The gradients of the output layer are first calculated, then work backwards to the first hidden layer. From these gradients we can obtain the gradients on the parameters of each layer. This algorithm is adapted from the work by Goodfellow *et al.* [11].

```
1  input: ANN parameters $\{\mathrm{W}, \mathbf{b}\} \in \boldsymbol{\theta}$
2  input: input, $\mathbf{x}$
3  input: target $y$
4  $\hat{y} \leftarrow$forward_pass($\mathbf{x}$) // obtain prediction
5  $\mathbf{r} \leftarrow \nabla_{\hat{y}} J(\hat{y}, y)$ // compute the gradient on the output layer
6  for $k = L, L-1, \ldots, 1$ do
7      $\mathbf{r} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{r} \odot \mathbf{g}'(\mathbf{a}^{(k)})$ // convert the gradient on the layer's
         output to the pre-activation (element-wise multiplication if g
         is element-wise)
8      $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{r}$ // compute the gradient on the layer's biases
9      $\nabla_{\mathrm{W}^{(k)}} J = \mathbf{r}\mathbf{h}^{(k-1)\top}$ // compute the gradient on the layer's weights
10     $\mathbf{r} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathrm{W}^{(k)\top}\mathbf{r}$ // propagate gradients to the following lower
         layer's activations
11 end
```

of the network by using a single example input $\mathbf{x}$, but usually a batch of samples is used. A forward pass is first completed using Algorithm 7. The gradient on the output layer is then calculated. The algorithm then iterates in a reverse order through the layers of the network, starting at the output layer. At each iteration, the gradient on the current layer's output is converted to the pre-activation of the layer (line 7). This operation requires the following partial derivative to be calculated:

$$\frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{a}^{(k)}} = \frac{\partial}{\partial \mathbf{a}^{(k)}} \mathbf{g}\left(\mathbf{a}^{(k)}\right) = \mathbf{g}'\left(\mathbf{a}^{(k)}\right). \tag{5.29}$$

If the activation function $\mathbf{g}$ is chosen to be ReLU, then the derivative is

$$\mathbf{g}'\left(\mathbf{a}^{(k)}\right) = \begin{cases} 1 & \text{if } \mathbf{a}^{(k)} > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{5.30}$$

Stellenbosch University https://scholar.sun.ac.za

Next the gradient on the pre-activation of the layer is used to compute the gradient on the weights and biases of the layer. To obtain the gradient on the weights (line 8), the following derivative has to be computed:

$$\frac{\partial \mathbf{a}^{(k)}}{\partial \mathrm{W}^{(k)}} = \frac{\partial}{\partial \mathrm{W}^{(k)}}\Big(\mathbf{b}^{(k)} + \mathrm{W}^{(k)}\mathbf{h}^{(k-1)}\Big) = \mathbf{h}^{(k-1)}. \tag{5.31}$$

To obtain the gradient on the biases (line 9), the following derivative has to be computed:

$$\frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{b}^{(k)}} = \frac{\partial}{\partial \mathbf{b}^{(k)}}\Big(\mathbf{b}^{(k)} + \mathrm{W}^{(k)}\mathbf{h}^{(k-1)}\Big) = 1. \tag{5.32}$$

Finally the gradient from an activation of the layer is transferred to the output of the previous layer (line 10). This requires the following derivative to be computed:

$$\frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{h}^{(k-1)}} = \frac{\partial}{\partial \mathbf{h}^{(k-1)}}\Big(\mathbf{b}^{(k)} + \mathrm{W}^{(k)}\mathbf{h}^{(k-1)}\Big) = \mathrm{W}^{(k)}. \tag{5.33}$$

This process repeats until the first layer of the network is reached, and all the network gradients are computed. After all the gradients are obtained, the network parameters are adjusted using an optimiser, i.e. a gradient-descent method. We give an overview of some of the most popular optimisers in the next section. The entire process is then repeated for a new sample or batch of samples from the dataset. Once all the entries of the dataset have been sampled, we say an *epoch* is completed. Multiple epochs can be completed to improve the accuracy of the network.

5.1.6 Optimisers

The optimiser is responsible for adjusting the network parameters using the gradients computed with the backpropagation algorithm. In this section, we give an overview of the optimisers relevant to this study. We briefly review *batch gradient descent*, *stochastic gradient descent* (SGD) and *mini-batch gradient descent*. We then give an overview of two new gradient-descent methods, namely *momentum* and *adaptive moment estimation* (Adam). This section is based on the work of Ruder [36].

Batch gradient descent *Batch gradient descent*, also referred to as *full gradient descent*, entails computing the gradients of the cost function with regard to the network parameters for the entire dataset. Ruder [36] defines it as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t - \nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t). \tag{5.34}$$

Ruder [36] states that batch gradient descent can be computationally expensive or sometimes intractable if the datasets do not fit in memory. It also does not allow us to update the parameters online, i.e. with newly obtained examples on-the-fly.

Stellenbosch University https://scholar.sun.ac.za

Stochastic gradient descent We introduced SGD in Section 4.4.1. Recall that SGD entails approximating full gradient descent by utilising single random examples. An SGD update is defined by Ruder [36] as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\hat{y}^{[i]}, y^{[i]}; \boldsymbol{\theta}_t). \tag{5.35}$$

SGD is much faster than batch gradient descent and can learn online. Since SGD uses single examples, the updates have high variance, and this can lead to an unstable objective function.

Mini-batch gradient descent We have already referred to mini-batch gradient descent in Section 5.1.4. Mini-batch gradient descent compromises between SGD and batch gradient descent. It performs an update for a mini-batch of B examples, and is defined by Ruder [36] as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} J(\hat{\mathbf{y}}^{[i:i+B]}, \mathbf{y}^{[i:i+B]}; \boldsymbol{\theta}_t). \tag{5.36}$$

According to Ruder [36] mini-batch gradient descent reduces the variance of parameter updates and leads to more stable convergence. Mini-batch gradient descent is usually used when training ANNs [36]. In this study we utilise mini-batch gradient descent, leaving out the parameters $\hat{\mathbf{y}}^{[i:i+B]}, \mathbf{y}^{[i:i+B]}$ for simplicity.

Momentum Sutton [44] states that SGD has trouble navigating in areas which curve more sharply in some directions than in others, referring to these areas as *ravines*. According to Ruder [36] ravines are common around a local optima. SGD tends to oscillate across the slopes of the ravine and, thereby making slow progress towards the bottom of the local optima.

Momentum [34] is a technique that helps to address this problem by accelerating SGD in the relevant direction and by decreasing oscillations. A fraction of the update vector of the previous time step is added to the current update vector, hence

$$\mathbf{m}_t \doteq \rho \mathbf{m}_{t-1} - \alpha \nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t), \tag{5.37}$$

where ρ is the momentum term and is usually set to 0.9. The network parameters are then adjusted, creating

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t - \mathbf{m}_t. \tag{5.38}$$

Adam *Adaptive moment estimation* (Adam) was introduced by Kingma and Ba [20] and is based on adaptive estimates of lower-order moments. These estimates are used to compute adaptive learning rates for different parameters of the ANN. An exponentially decaying average of past gradients $\mathbf{m}_t$ is tracked, in the same way as *momentum*. This is define by Kingma and Ba [20] as

$$\mathbf{m}_t \doteq \rho_1 \mathbf{m}_{t-1} + (1 - \rho_1) \nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t). \tag{5.39}$$

Stellenbosch University https://scholar.sun.ac.za

Adam additionally also tracks an exponentially decaying mean of past squared gradients $\mathbf{v}_t$, which is defined by Kingma and Ba [20] as

$$\mathbf{v}_t \doteq \rho_2 \mathbf{v}_{t-1} + (1 - \rho_2)\nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t)^2. \tag{5.40}$$

The vectors $\mathbf{m}_t$ and $\mathbf{v}_t$ are initialised as zero and Kingma and Ba [20] observe that these vectors are biased towards zero. This bias occurs especially at initial time steps and when small decay rates are used, i.e. when ρ_1 and ρ_2 are close to one. Kingma and Ba [20] calculate bias-corrected first and second moment estimates to counter these biases. The bias-corrected first moment estimate is defined by Kingma and Ba [20] as

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \rho_1^t}. \tag{5.41}$$

The bias-corrected second moment estimate is defined by Kingma and Ba [20] as

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \rho_1^t}. \tag{5.42}$$

The parameters are then updated, defined by Kingma and Ba [20] as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t. \tag{5.43}$$

5.2 Convolutional neural networks

We now discuss CNNs: ANNs that specialise in analysing and processing images and other high-dimensional data arranged in 2D arrays [45]. The CNN incorporates a convolutional layer, which is based on the mathematical convolution operation. A convolutional layer reuses network parameters to do operations at different parts of the input to produce several 2D arrays, called *feature maps*. A feature map is a 2D array that contains features identified by a filter applied to a previous layer. For example, a convolutional layer may be responsible for detecting a particular curve in an image. The convolutional layer will then produce a feature map with activated entries wherever the curve was identified in the image.

Goodfellow *et al.* [11] state the idea of sharing parameters in convolutional layers arises from the fact that there are many statistical properties to natural images that are invariant to translation. The convolutional layer is an example of incorporating domain knowledge into the architecture of the network in order to achieve a more efficient network with fewer weights [11].

5.2.1 Operations of the convolutional layer

We now discuss the operations of the convolutional layer and how the feature maps are produced. The discussion is based on the work of LeCun *et al.* [21]. A convolutional layer

Stellenbosch University https://scholar.sun.ac.za

receives a 3D array X as input, which is a set of n_1 2D array feature maps each of size $n_2 \times n_3$. Each input feature map is denoted X_i. The output Y of a convolutional layer is also a 3D array which is a set of m_1 2D array feature maps, each with a size $m_2 \times m_3$. The output feature maps are obtained by convolving several filters, each with a trainable weight matrix K_{ij}, over the input.

Karpathy *et al.* [19] state that zero-padding can be applied to the input. Zero-padding entails adding a border of zeros around the edge of the input feature maps. The zero-padding parameter p specifies the border's thickness, i.e. the number of zeros to be added to the border. It allows for controlling the spatial size of the output feature maps.

LeCun *et al.* [21] state that the filter K_{ij} connects the input feature map X_i to the output feature map Y_j. The filters have a specified receptive field $l_1 \times l_2$, which is the scope (width and height) of the respective input feature maps that can be observed at once. LeCun *et al.* [21] state that the output feature map Y_j can be calculated by

$$\mathrm{Y}_j = \mathrm{B}_j + \sum_i \mathrm{K}_{ij} * \mathrm{X}_i, \tag{5.44}$$

where $*$ is the 2D discrete convolution operator and B_j is a trainable bias. The convolution operation in Equation 5.44 entails moving a filter across the width and the height of the input, while performing the dot product between the weights of the filter and the entries of the respective input feature map that fall within in the scope of the receptive field [19].

The individual units of feature maps are calculated identically except that the receptive field of the filter is shifted to a different location on the array of incoming data when doing the calculation [45]. The amount the filter is moved between operations is called the stride, denoted s. With the input size, filter size, stride, and amount of zero-padding applied, we can calculate the width m_2 and height m_3 of the output feature maps, which are specified by Karpathy *et al.* [19] as

$$m_2 = \frac{n_2 - l_1 + 2p}{s} + 1, \tag{5.45}$$

and similarly

$$m_3 = \frac{n_3 - l_2 + 2p}{s} + 1. \tag{5.46}$$

In Figure 5.3 we utilise Equation 5.44 to transform a number of input feature maps to an output feature map. In this example one can use Equation 5.45 and Equation 5.46 to obtain the size of the output feature map. An element-wise activation function such as ReLU is usually applied to the output volume of a convolutional layer [19]. Convolutional layers are fully differentiable, therefore, as discussed in the previous section, backpropagation can be used to optimise the parameters of the filters in these layers.

5.2.2 Architecture of a convolutional neural network

Sutton and Barto [45] state that subsampling layers can be added between convolutional layers to reduce the spatial resolution of the feature maps. The benefit of adding subsam-

Stellenbosch University https://scholar.sun.ac.za

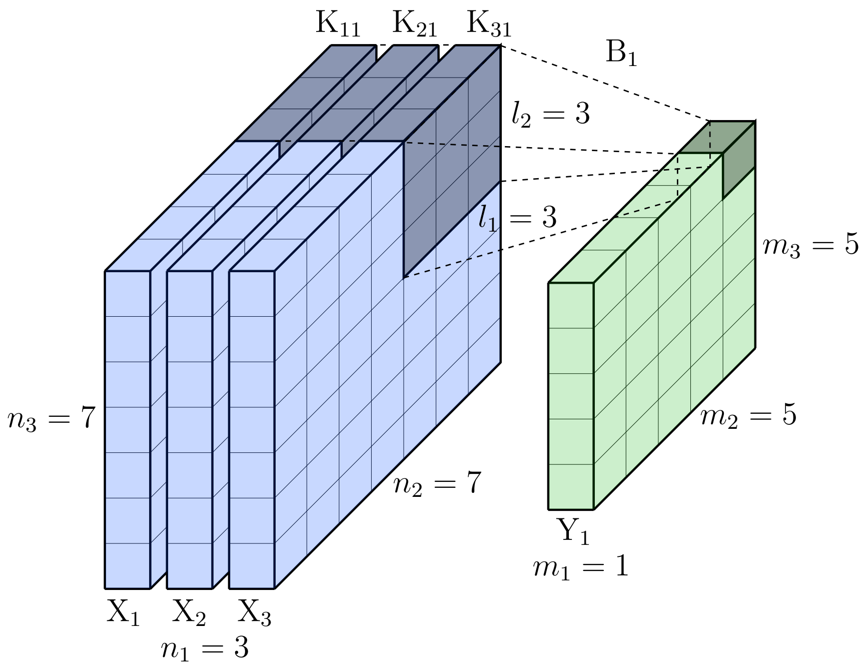

Figure 5.3: Three 2D array filters K_{11}, K_{21}, K_{31} (dark blue) with a receptive field 3×3 and a stride $s = 1$ are convolved over three input feature maps X_1, X_2, X_3 (light blue) of size 7×7. No zero-padding is applied, therefore $p = 0$. A single output feature map Y_1 (green) of size 5×5 is obtained. Additional filters, for example K_{12}, K_{22}, K_{32}, can be applied to obtain additional output feature maps.

pling layers is that the CNN is then less sensitive to the spatial locations of the features detected. Spatial invariance is helpful when classifying images, as we want the CNN to be able to classify an object irrelevant to its location in the image. However, in RL applications, spatial information is often crucial. For example, in the problem addressed by this study, it is essential to know whether the first-aid kit is to the left or to the right of the agent in order to perform the best possible action. In order to have spatial information, we omit the subsampling layers, and the CNN only consists of convolutional and fully-connected layers. The drawback of omitting the subsampling layers is that this dramatically increases the number of weights in the CNN. Therefore it increases the number of examples needed to train the CNN.

We end off this section with an example of a CNN architecture shown in Figure 5.4. The network receives an input. The first part of the network consists of three convolutional layers. The input is propagated through these layers. Each layer produces a number of feature maps which is then passed to the next layer. The output of the third convolutional layer is flattened to a vector. The flattened output of the last convolutional layer serves as the input to a fully connected feed-forward layer. The feed-forward layer produces the output of the network. We omit the subsampling layers in the architecture.

Stellenbosch University https://scholar.sun.ac.za

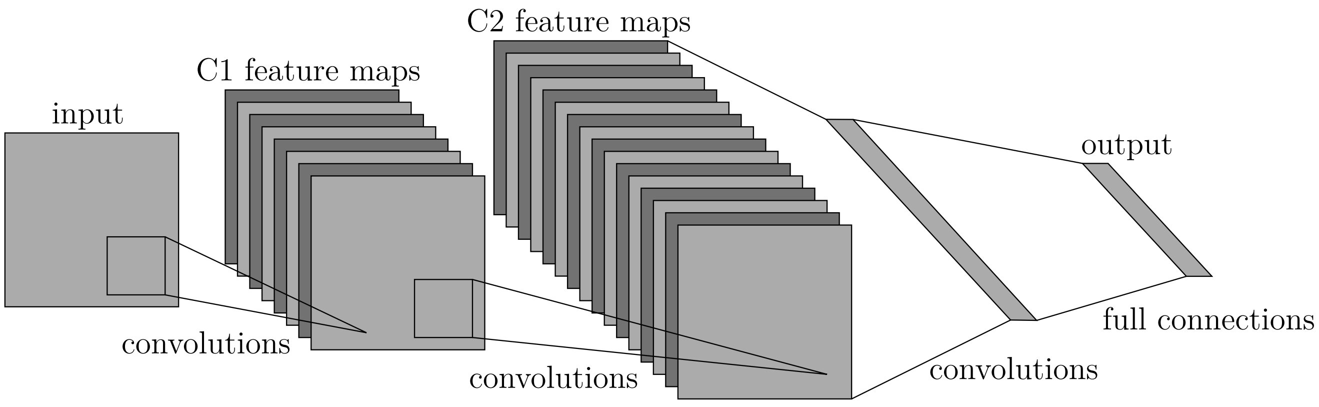

Figure 5.4: A typical CNN architecture adapted from the work by LeCun *et al.* [21]. The subsampling layers are omitted. The output volume of the last convolutional layer is flattened to a vector and ends off with a fully connected feed-forward layer.

5.3 Summary

In this chapter, we discussed how ANNs could be applied to a supervised learning problem where there is a dataset available. We discussed how an input to the ANN is evaluated using forward propagation. We reviewed the most popular loss functions and looked at how to effectively compute the gradients of an ANN using backpropagation. We also discussed the CNN and the operations of the convolutional layer.

The goal is to use a CNN to approximate the action-value function of a Q-learning agent. In the next chapter, we discuss how the ANN can be combined with the Q-learning algorithm. However, the assumption of an available dataset usually does not apply in RL problems. Other problems also arise when combining the off-policy Q-learning algorithm with an ANN. The next chapter identifies these problems and reviews the deep Q-learning algorithm, which addresses these issues.

Stellenbosch University https://scholar.sun.ac.za

Chapter 6

Deep reinforcement learning

In Chapter 4 we discussed the Q-learning control algorithm. Recall that Q-learning is an off-policy algorithm as the agent learns about an optimal policy while following a different policy. The off-policy nature of the algorithm allows the agent to be more sample-efficient. The reason for this is that the agent can use any experience generated in the environment to improve its value function and policy.

We also discussed that it is better to approximate the value function with a parameterised function than to use tabular methods when dealing with problems that have vast state spaces and action spaces. In the previous chapter we introduced the *artificial neural network* (ANN) as it is a flexible function approximator that allows for end-to-end learning. Unfortunately, with the combination of off-policy learning, function approximation, and bootstrapping – referred to as the deadly triad – the danger of instability and divergence arises, as stated by Sutton and Barto [45].

In this chapter, we review *deep Q-learning* – a Q-learning algorithm that utilises an ANN to approximate the action-value function. This algorithm was developed by Mnih *et al.* [23, 24] and it trains an ANN called the *deep Q-network* (DQN). Deep Q-learning overcomes the instabilities of the deadly triad by utilising two concepts: experience replay *(ER)* and *frozen targets*. Since the DQN was introduced, many improvements have been made to the original algorithm. We therefore also discuss improvements such as the *double deep Q-network* (DDQN), the n-step update and *prioritised experience replay* (PER). The following chapter is largely based on the work of Mnih *et al.* [23, 24].

Stellenbosch University https://scholar.sun.ac.za

6.1 Deep Q-learning

Deep Q-learning, by Mnih *et al.* [23, 24], uses a *convolutional neural network* (CNN) together with Q-learning to learn successful policies from high-dimensional inputs. The CNN automatically extracts the relevant features from the high-dimensional observations in order to learn control policies with Q-learning. In this section we discuss the deep Q-learning algorithm and the main contributions made by Mnih *et al.* [23, 24] to ensure its stability.

Algorithm 9 shows the deep Q-learning algorithm that is discussed throughout this section. It is very similar to Q-learning shown in Algorithm 5. However, an ANN is now

Algorithm 9: Deep Q-learning with *experience replay* (ER), adapted from the work by Mnih *et al.* [23, 24]. The algorithm or agent runs in the environment for a defined number of episodes. For each step of an episode the agent interacts with the environment using an ε-greedy strategy (line 8 and 9). The resultant transition from the interaction is stored in the replay buffer $\mathcal{D}$ (line 10). Once there are enough transitions in the replay buffer ($> B$), B sized mini-batches $(S_j, A_j, R_{j+1}, S_{j+1}) \backsim U(\mathcal{D})$ of transitions are randomly sampled at each iteration (line 12). For each transition j in the batch, the target Y_j is computed (line 13). The TD error δ_j for each transition in the batch is then computed (line 14). Each TD error δ_j is used to compute the gradients on the policy network parameters – the gradients are also accumulated (line 15). The policy network parameters are adjusted in the direction of the resultant gradients (line 17). Every C steps the target network parameters are set to the policy network parameters (line 18).

```
1  input: $\varepsilon > 0$, learning rate $\alpha$, mini-batch size $B$, capacity $N$
2  initialise replay memory $\mathcal{D}$ to size $N$, $\delta = 0$
3  initialise action-value function $Q$ with random weights $\boldsymbol{\theta}$
4  initialise target action-value function $\hat{Q}$ with weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
5  for each episode do
6      initialise $S_0$
7      repeat for each step $t$ of episode, $t = 0$
8          observe $S_t$ and choose $A_t \backsim \pi_{\boldsymbol{\theta}}(S_t)$ ($\varepsilon$-greedy)
9          execute action $A_t$ in environment and observe reward $R_{t+1}$ and state $S_{t+1}$
10         store transition $(S_t, A_t, R_{t+1}, S_{t+1})$ in $\mathcal{D}$
11         for $j = 1$ to $B$ do
12             sample random transition $(S_j, A_j, R_{j+1}, S_{j+1})$ from $\mathcal{D}$
13             $Y_j = \begin{cases} R_{j+1} & S_{j+1} \text{ terminal} \\ R_{j+1} + \gamma \max_{a'} \hat{Q}(S_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
14             compute TD error $\delta_j = Y_j - Q(S_j, A_j; \boldsymbol{\theta})$
15             accumulate weight change $\Delta \leftarrow \Delta + \delta_j \cdot \nabla_{\boldsymbol{\theta}} Q(S_j, A_j; \boldsymbol{\theta})$
16         end
17         update weights $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \Delta$, reset $\Delta = 0$
18         every $C$ steps $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
19     until $S_{t+1}$ is terminal
20 end
```

Stellenbosch University https://scholar.sun.ac.za

used to approximate the action-value function (also known as the Q function) instead of the tabular method that we previously used.

Sutton and Barto [45], Mnih *et al.* [23, 24], and Tsitsiklis and Van Roy [47] state that instability can occur when off-policy TD learning is combined with non-linear function approximation. Mnih *et al.* [23, 24] explain that there are several reasons for this instability. RL algorithms such as Q-learning usually use sequential transitions to update the action-value function. The observations contained in the transitions are highly correlated due to the sequential nature of the transitions. Using correlated data samples to update the parameters of the ANN can lead to divergence. For this reason, traditional supervised learning algorithms sample at random to break correlations in data samples.

Furthermore, the parameterised action-value function that is being estimated is also used to compute target values $Y_j = R_{j+1} + \gamma \max_{a'} Q(S_{j+1}, a')$. Therefore action values and target values are also correlated. In addition, the target is a function of the same parameters that are being updated. Therefore updates made to the parameters of the action-value function will change the target values. The implication is that target values are non-stationary and according to Sutton and Barto [45] this complicates the process compared to supervised learning, where targets are stationary. Mnih *et al.* [23, 24] address these problems by utilising two ideas, *experience replay* (ER) and *fixed Q-targets* to transform the RL problem to be closer to a supervised learning problem. We next discuss the functionality of these components.

6.1.1 Experience replay

Supervised learning assumes *independent and identically distributed* (IID) data and works better in such cases. In contrast, RL normally uses highly correlated sequential transitions to update the value function [23, 24]. Processing sequential states that are highly correlated can likely cause the RL algorithm to become unstable. Mnih *et al.* [23, 24] address this problem with ER, which was first studied by Lin [22], and is essential to the deep Q-learning algorithm.

Lin [22] describes ER as the ability of the agent to remember past experiences and to then repeatedly present these experiences to itself. Experiences at each time step $e_t = (S_t, A_t, R_{t+1}, S_{t+1})$ are stored in a replay buffer $\mathcal{D} = \{e_1, \ldots, e_t\}$ with a fixed size, N. The buffer only holds the most recently generated transitions. A benefit of using ER is that the algorithm can reuse previously generated experience, which results in better sample efficiency. ER can only be used with an off-policy RL algorithm, and therefore Q-learning was a suitable choice. Furthermore, ER breaks correlations in the data, as randomly sampled mini-batches of transitions are used to update the network parameters. Mnih *et al.* [23, 24] state that ER reduces the variance of the updates and increases the stability of the algorithm. ER therefore transforms the RL problem to be closer to a supervised learning problem.

Stellenbosch University https://scholar.sun.ac.za

The work by Fedus *et al.* [10] shows that the performance of a deep Q-learning agent is very dependent on the age of the transitions in the replay buffer. The *replay ratio* is a measure used by Fedus *et al.* [10] to indicate the number of network updates made to DQN per transition generated. Fedus *et al.* [10] define the replay ratio as

$$\text{replay\_ratio} = \frac{\#\text{network\_updates}}{\#\text{transitions\_generated}}. \tag{6.1}$$

The replay ratio allows for quantifying the rate at which the network is updated relative to the rate at which transitions are generated. Fedus *et al.* [10] describe the replay ratio as a measure of the frequency at which the agent is learning on existing experience or learning on newly acquired experience.

The *age* of a transition is described by Fedus *et al.* [10] as the number of network updates made since the transition was generated. The age of the oldest transition in the replay buffer is a measure used by Fedus *et al.* [10] to describe the age of the transitions in the replay buffer. Usually a *first-in-first-out* (FIFO) replay buffer is used, i.e. when the replay buffer is full, the oldest transitions in the replay buffer are replaced with new transitions. If we calculate the number of network updates performed in the time to replace each transition in the replay buffer, we also obtain the number of network updates completed since the oldest transition in the replay buffer was generated. Therefore multiplying the replay ratio with the replay capacity is equal to the age of the oldest transition in the replay buffer:

$$\text{oldest\_transition\_age} = (\text{replay\_ratio})(\text{replay\_capacity}). \tag{6.2}$$

Equation 6.2 shows that the age of the replay buffer is directly proportional to the replay ratio. Smaller replay ratios thus decrease the age of the transitions in the replay buffer.

6.1.2 Fixed Q-targets

Recall that the targets in RL are nonstationary, unlike the stationary targets that are usually found in traditional supervised learning problems. Mnih *et al.* [23, 24] used a technique referred to as *fixed Q-targets*, which transforms Q-learning closer to the traditional supervised learning problem, while still being able to bootstrap.

One simple solution by Mnih *et al.* [23, 24] is to have two versions of the action-value network. We refer to these networks as the *policy network* $Q(\boldsymbol{\theta})$ and a *target network* $\hat{Q}(\boldsymbol{\theta}^-)$. The parameters $\boldsymbol{\theta}$ of the policy network Q are updated at each iteration of the algorithm and are used by the agent to interact with the environment. Every C steps of updating the policy network, its parameters are loaded to the target network $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$. The parameters of the target network are then kept frozen for C steps, which after it is updated to the latest parameters of the policy network. The target network is used to compute the TD targets $Y_j = R_{j+1} + \gamma \max_{a'} \hat{Q}(S_{j+1}, a'; \boldsymbol{\theta}^-)$. By keeping the target network's parameters

Stellenbosch University https://scholar.sun.ac.za

frozen for several iterations of updating the network, it allows for more stationary targets. Accordingly the RL problem becomes more similar to a traditional supervised learning problem.

6.1.3 Optimising the deep Q-network

We next discuss the process of optimising the deep Q-learning agent's action-value function, i.e. the agent's network. The algorithm samples random mini-batches of transitions uniformly from the replay buffer $(S_j, A_j, R_{j+1}, S_{j+1}) \backsim U(\mathcal{D})$. The mini-batches are used to improve the action-value function. As this is a regression problem, Mnih *et al.* [23, 24] minimise the *mean square error* (MSE) cost function at each iteration of the algorithm as follows:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \sum_j \left[\Big(\underbrace{R_{j+1} + \gamma \max_{a'} \hat{Q}(S_{j+1}, a'; \boldsymbol{\theta}^-)}_{\text{target}} - \underbrace{Q(S_j, A_j; \boldsymbol{\theta})}_{\text{action-value}} \Big)^2 \right] \\ &= \mathbb{E}_{(S_j, A_j, R_{j+1}, S_{j+1}) \backsim U(\mathcal{D})} \left[\Big(R_{j+1} + \gamma \max_{a'} \hat{Q}(S_{j+1}, a'; \boldsymbol{\theta}^-) - Q(S_j, A_j; \boldsymbol{\theta}) \Big)^2 \right]. \end{aligned} \tag{6.3}$$

Next, we differentiate the cost function with respect to the network parameters to obtain the gradients of the network. Recall that the gradients indicate the direction in which the network parameters should be adjusted to minimise the cost function. We differentiate the cost function with respect to the network's parameters, hence

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}\left[\Big(R_{j+1} + \gamma \max_{a'} \hat{Q}(S_{j+1}, a'; \boldsymbol{\theta}^-) - Q(S_j, A_j; \boldsymbol{\theta}) \Big) \nabla_{\boldsymbol{\theta}} Q(S_j, A_j; \boldsymbol{\theta}) \right]. \tag{6.4}$$

Recall that the gradients of the network are computed with the backpropagation algorithm that we discussed in Section 5.1.5. Mnih *et al.* [24] also clipped the TD error δ to between -1 and 1. The result is that the cost function corresponds to the *mean absolute error* (MAE) cost function (discussed in Section 5.1.4) outside the interval of $(-1, 1)$. Accordingly, the cost function is a variant of the Huber cost function we discussed in Section 5.1.4 and which, according to Mnih *et al.* [24], improves the stability of the algorithm.

6.1.4 Double Q-learning with the n-step return

Hasselt *et al.* [12] state that Q-learning can sometimes be overoptimistic when estimating action values. They showed that these overestimates are common in practice and can be harmful to the performance of the deep Q-learning agent. For this reason Hasselt *et al.* [12] introduced the *double deep Q-network* (DDQN) (or double Q-learning) to counter the overestimation of action values. Double Q-learning modifies the target for non-terminal states in line 13 of Algorithm 9. Hasselt *et al.* [12] define the DDQN target as

$$Y_j^{\text{DDQN}} \doteq R_{j+1} + \gamma \hat{Q}\Big(S_{j+1}, \underset{a'}{\operatorname{argmax}} Q(S_{j+1}, a'; \boldsymbol{\theta}); \boldsymbol{\theta}^- \Big). \tag{6.5}$$

Stellenbosch University https://scholar.sun.ac.za

Equation 6.5 utilises both the target network and the policy network to compute the target value. Specifically, DDQN utilises the policy network to choose the optimal action and the target network to estimate the corresponding action value, where previously only the target network was used. Hasselt *et al.* [12] show that double Q-learning reduces overestimation of action values and leads to better performance on several Atari games.

We considered implementing *eligibility traces*, as mentioned in Section 4.3.4. According to Sutton and Barto [45], eligibility traces are the first line of defence against long-delayed rewards. For this reason, we suspect that eligibility traces may greatly help to address this study's problem where rewards are generally very delayed. The obstacle is that eligibility traces require the processing of the transitions of an episode in the order that the agent acquired them. As the agent samples at random from the replay buffer with ER, it will be challenging to use eligibility traces.

For this reason, we rather use the n-step return that we also discussed in Section 4.3.4. Recall that the n-step return utilises several transitions of future time steps to better estimate the target value of the current action value. The transitions needed to compute the n-step return can be grouped in the replay buffer, and therefore it is possible to combine the n-step return with ER. We modify the estimated return in Equation 6.5 to include the n-step return, hence

$$Y_j^{\text{DDQN\_}n\text{-step}} \doteq \sum_{k=0}^{n-1} \gamma^k R_{j+k+1} + \gamma^n \hat{Q}\Big(S_{j+n}, \underset{a'}{\operatorname{argmax}} Q(S_{j+n}, a'; \boldsymbol{\theta}); \boldsymbol{\theta}^-\Big). \tag{6.6}$$

Equation 6.6 is very similar to Equation 4.16, except that the DDQN target is used for the estimated value of nth state. We use a similar implementation as the work by Hessel *et al.* [14], thus we do not use *importance sampling* (IS).

6.1.5 Dueling architecture

The DNN that Mnih *et al.* [23, 24] used for the original DQN agent consists of three convolutional layers and two fully connected feed-forward layers. However, the dueling architecture requires that the input to the *feed-forward neural network* (FNN) be split into two streams, replacing the single stream that is used in existing algorithms such as that of Mnih *et al.* [23, 24]. The dueling architecture is compared to the popular single stream architecture in Figure 6.1. The streams feed into two separate estimators: an estimator for the state value function $V(s)$ and an estimator for the state-dependent action advantage $A(s, a)$ function. This architecture therefore separates the estimation of state values and that of action advantages. The dueling architecture then combines the outputs of the estimators to obtain the state-action value $Q(s, a)$, which is defined by Wang *et al.* [50] as

$$Q(s,a) = V(s) + \Big(A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')\Big). \tag{6.7}$$

Intuitively, this architecture learns valuable states without having to learn the effect of each action for each state. According to Wang *et al.* [50], the dueling architecture has the

Stellenbosch University https://scholar.sun.ac.za

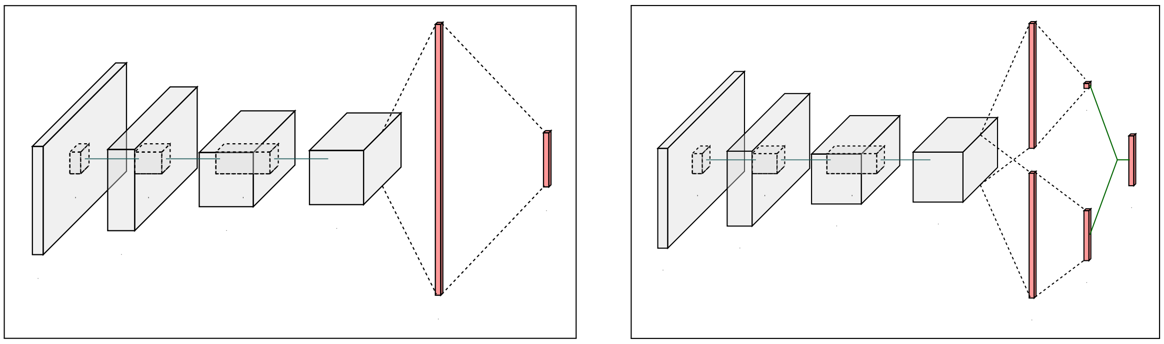

Figure 6.1: The single Q-network (left) as compared to the dueling Q-network (right), adapted from the work by Wang *et al.* [50].

benefit of generalising across actions without having to change anything to the underlying RL algorithm. The results by Wang *et al.* [50] show that the dueling architecture leads to better policy evaluation in the presence of actions with similar values.

6.2 Deep Q-learning with prioritised experience replay

The deep Q-learning algorithm we discussed in Section 6.1 samples transitions uniformly at random from the replay buffer. Mnih *et al.* [23, 24] suggest that some transitions may be more important than others. Accordingly, there may be better sampling strategies to learn the most from the important data in the replay buffer. For this reason we investigate deep Q-learning with *prioritised experience replay* (PER) – introduced by Schaul *et al.* [39]. PER is very similar to the previously discussed ER, with the exception that PER uses an alternative sampling strategy. PER assigns priorities to the transitions contained in the replay buffer. Transitions are then sampled according to the priorities they have been assigned. The work by Schaul *et al.* [39] shows that inclusion of PER significantly improves the performance of the deep Q-learning algorithm.

Algorithm 10, adapted from the work by Schaul *et al.* [39], shows the double deep Q-learning algorithm with PER. An n-step update of $n = 1$ is used in Algorithm 10, but other values for n can also be used. We discuss Algorithm 10 throughout this section. We first review how Schaul *et al.* [39] calculate the priorities of transitions. Since PER leads to a bias toward transitions with higher priorities, we also discuss how this bias can be corrected with *importance-sampling* (IS) weights.

6.2.1 Transition priorities

PER prioritises transitions based on the temporal-difference (TD) error made in the prediction step of the Bellman equation. Schaul *et al.* [39] define the probability of

Stellenbosch University https://scholar.sun.ac.za

Algorithm 10: Double deep Q-learning with proportional prioritisation adapted from the work of Schaul *et al.* [39].

```
input: $\varepsilon > 0$, learning rate $\alpha$, mini-batch size $B$, capacity $N$, exponents $\zeta$ and $\beta$
initialise replay memory $\mathcal{D}$ to size $N$, $\Delta = 0$, $p_1 = 1$
initialise action-value function $Q$ with random weights $\boldsymbol{\theta}$
initialise target action-value function $\hat{Q}$ with weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for each episode do
    initialise $S_0$
    for each step $t$ of episode do
        observe $S_t$ and choose $A_t \backsim \pi_{\boldsymbol{\theta}}(S_t)$ ($\varepsilon$-greedy)
        execute action $A_t$ in environment and observe reward $R_{t+1}$ and state $S_{t+1}$
        store transition $(S_t, A_t, R_{t+1}, S_{t+1})$ in $\mathcal{D}$ with priority $p_t = \max_{i<t} p_i$
        for $j = 1$ to $B$ do
            sample transition $j \backsim Pr\{j\} = (p_j)^\zeta / \sum_i (p_i)^\zeta$
            compute importance-sampling weight $w_j = \big(N \cdot Pr\{j\}\big)^{-\beta} / \max_i(w_i)$
            $Y_j = \begin{cases} R_{j+1} & S_{j+1} \text{ terminal} \\ R_{j+1} + \gamma \hat{Q}\big(S_{j+1}, \underset{a'}{\text{argmax}} Q(S_{j+1}, a'; \boldsymbol{\theta}); \boldsymbol{\theta}^-\big) & \text{otherwise} \end{cases}$
            compute TD error $\delta_j = Y_j - Q(S_j, A_j; \boldsymbol{\theta})$
            update transition priority $p_j \leftarrow |\delta_j|$
            accumulate weight change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_{\boldsymbol{\theta}} Q(S_j, A_j; \boldsymbol{\theta})$
        end
        update weights $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \Delta$, reset $\Delta = 0$
        every $C$ steps $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
    end
end
```

sampling a transition i as

$$Pr\{i\} \doteq \frac{p_i^\zeta}{\sum_k p_k^\zeta}, \tag{6.8}$$

where $p_i > 0$ is the priority of transition i. The exponent ζ determines the amount of prioritisation used. If $\zeta = 0$, then priorities of transitions are uniform. Schaul *et al.* [39] describe two methods, namely rank-based and proportional-based, to calculate the priority p_i. In the results by Schaul *et al.* [39] it is found that the performance of the proportional-based method is better than that of the rank-based method, therefore we only consider the former. The priority of the proportional-based method is calculated by Schaul *et al.* [39] as

$$p_i = |\delta_i| + \eta, \tag{6.9}$$

where δ_i is the TD error associated with transition i. A small value η is added to ensure that no transition has a zero probability of being sampled.

Newly generated transitions have unknown priorities and are stored with maximum priority to ensure that all transitions are sampled once. Only the priorities of transitions

Stellenbosch University https://scholar.sun.ac.za

that are sampled are updated, as it is computationally very expensive to sweep over all the priorities in the replay buffer with each update made to the DQN.

We would like to sample according to priorities in the replay buffer. A simple solution is to perform a cumulative sum on all the priorities of the replay buffer. One can sample a random number from a uniform distribution, from zero to the maximum cumulative priority. The element in the replay buffer that has a corresponding cumulative interval to the number sampled, is then selected. Although this is a viable solution for prioritised sampling, it has a time complexity of $O(N)$, where N is the size of the replay buffer. Therefore, sampling is expensive for very large replay buffers.

As suggested by Schaul *et al.* [39], the sum-tree is a binary tree that allows for more efficient prioritised sampling. It has a time complexity of $O(log_2(N))$ and was therefore this study's method of choice for prioritised sampling.

6.2.2 Importance-sampling weights

Schaul *et al.* [39] state that the estimation of the expected value with stochastic updates relies on the fact that the updates correspond to the same distribution as the expectation. One problem with PER is that the distribution used to sample transitions is no longer uniform, i.e. some transitions are sampled more frequently than others. The change in this distribution introduces a bias which entails that estimates converge to a solution that differs from the expectation. This problem can be solved by using *importance-sampling* (IS) weights to counter the bias that is introduced. Schaul *et al.* [39] calculate the IS weight associated with transition i as follows:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{Pr\{i\}}\right)^{\beta}, \tag{6.10}$$

where N is the size of the replay buffer and the exponent β specifies the amount of IS correction. The IS weights are incorporated into the Q-learning update by using $w_i\delta_i$ instead of δ_i. Schaul *et al.* [39] normalise weights by $1/\max_i(w_i)$ which scales weights downward for stability reasons. If the exponent $\beta = 1$ then the algorithm fully compensates for the non-uniform probabilities. As the process is already non-stationary due to changing policies, state distributions and bootstrap targets, it is said by Schaul *et al.* [39] that it is most important to have unbiased updates at the end of training when the action-value function is close to convergence. Schaul *et al.* [39] therefore hypothesise that the small bias that occurs at the beginning of training can be ignored and they suggest using a scheduled β that starts from $\beta = \beta_0$ and reaches $\beta = 1$ at the end of training.

6.3 Summary

In this chapter we discussed the deep Q-learning algorithm which combines Q-learning with ANNs. We discussed the main contributions made by Mnih *et al.* [23, 24] to ensure its

Stellenbosch University https://scholar.sun.ac.za

stability. The important components discussed were ER and frozen Q-targets. We discussed the functionality of each component with regard to the deep Q-learning algorithm.

We also discussed some additional improvements made to the original deep Q-learning algorithm. These improvements include the DDQN and the n-step update. Both these improvements are small adjustments and can be easily be added to the original deep Q-learning algorithm.

Hessel *et al.* [14] showed that PER drastically improves the deep Q-learning algorithm's performance on the Atari benchmark. We therefore also discussed the deep Q-learning algorithm with PER. We looked at how Schaul *et al.* [39] computed proportional-based priorities for transitions in the replay buffer. Prioritised sampling introduces a bias towards certain transitions. We also reviewed how this bias can be corrected using IS weights.

Before we discuss our implementation of a distributed deep Q-learning algorithm with PER, we discuss the simulation environment of choice for the experimental phase of this study. The simulation environment would in many ways influence the architecture of our implementation and therefore it is important to discuss this first.

Stellenbosch University https://scholar.sun.ac.za

Chapter 7

Simulation environment

The simulation environment is an important component of a *reinforcement learning* (RL) framework. It determines the outcome of actions taken by the agent and also how the agent's state is observed. It specifies the objective with a reward function and thus dictates the desired behaviour of the agent. Although RL can be done in a real world environment, computer simulators and video games are commonly used and have proven to serve as effective test-beds. They provide safe environments that are easily accessible with rewards that are quantifiable. From a performance point of view, computer simulators have the advantage of running at accelerated speeds and multiple environments can be simultaneously processed. This chapter considers the possible simulation environments than can be used as a test-bed for experiments like ours and examines them according to their features and performance.

7.1 OpenAI Gym

Gym is an RL toolkit developed by OpenAI. It includes a range of simulation environments that can be used for developing and testing RL algorithms [5]. The Gym toolkit also specifies a standard abstract class (`gym.Env`) that can be inherited by third party environments. All environments that inherit the `gym.Env` abstract class implement its attributes and methods, and thus operate similarly. The main attributes are *action_space* and *observation_space*; they respectively specify the agent's available actions and the format of the observations received from the environment. The main functions are `reset`, `step` and `render`. The `reset` function resets the environment to an initial state (start of the episode) and returns the observation of this state. The `step` function allows the agent to perform an action in the environment and returns the following: the next observation, a reward, a signal that indicates whether the episode is done, and extra information that can be used for diagnostic purposes. The `render` function renders and displays the environment. The `gym.Env` class with main attributes and methods can be viewed in Algorithm 11. This class is widely adopted by most RL environments and can be seen as the standard for

Stellenbosch University https://scholar.sun.ac.za

Algorithm 11: gym.Env

```
1 attributes:
2     action_space // specifies the available actions
3     observation_space // specifies the format of the observations
       returned from the environment
4 def reset(): // resets the environment
5     return observation // returns the initial observation
6 def step(action): // accepts and performs action
7     return observation, reward, done, info
      // observation - observation of the current state
      // reward - reward returned
      // done - whether the episode has ended
      // info - contains auxiliary diagnostic information
8 def render(): // renders and displays the environment to the screen
```

agent-environment interaction. The simulation environments investigated in this chapter all inherit the `gym.Env` class. This allowed us to develop a more general agent that could easily be applied to many other Gym environments.

7.2 Description of simulation environments

One of the objectives in Section 1.5 was to obtain an agent that is capable of performing tasks in a 3D environment with partial observability. This was therefore an important consideration in selecting an appropriate environment. The task required by our study is that an agent must deliver a first-aid kit to an injured miner. Numerous sub-tasks must be completed to achieve the main specified objective. The task requires the agent to navigate the environment, and to also interact with elements inside the environment. With this in mind, two possible 3D environments were investigated in order to choose a suitable environment to use as a test-bed for the proposed problem.

Minecraft Minecraft is a 3D, open-ended, open-world sandbox game with an extensive range of possible goals to achieve by the player, in this case the RL agent. Game worlds are procedurally generated and consist of discrete blocks with which the player can interact, i.e. by breaking blocks or placing new blocks. The environment is viewed with a first-person camera, which makes observations representative of that of an actual robot that is fitted with an RGB camera. Minecraft also features a basic physics system, a vast range of items and entities, and an extensive crafting system.

Project Malmo is a framework developed by Microsoft in 2016 that allows Python code to interface with the Minecraft environment [18]. Project Malmo also gives developers the functionality to design various custom tasks for their agents inside Minecraft by means of *extensible markup language* (XML) scripts. MineRL is a framework built on Project Malmo

Stellenbosch University https://scholar.sun.ac.za

and was first released in 2019. MineRL has all the functionality of Project Malmo, but inherits the `gym.Env` abstract class and streamlines the process of developing RL agents for Minecraft. The performance of MineRL is also dramatically better than that of Project Malmo, which made it a promising platform for us to use as an RL test-bed.

We wanted to use the built-in mechanics of Minecraft to create an environment representative of the problem statement. The problem was that the tools available to create customised environments are not well documented and we found them difficult to use. In many cases the tools were found to be very limited and the desired problem was difficult to design. Furthermore, Minecraft is a closed-source project that does not allow adding of additional functionality.

MiniWorld MiniWorld [6] is an alternative environment that also inherits `gym.Env`. It is developed in Python with Pyglet and OpenGL and it is open-source. Similar to Minecraft, it is a 3D environment that is observed by a first-person camera. MiniWorld allows the agent to navigate the environment and to interact with some elements found in the environment. The interaction entails picking up, carrying and dropping items, but the environment does not nearly have the amount of features that Minecraft has. With limited features, the range of possible problems that can be created in MiniWorld is more limited. However, the source code of MiniWorld can be modified, therefore it would be possible to add the mechanics needed to construct the desired problem for our study. These mechanics had to be considered beforehand, as they could be time-consuming to add.

A summary of how the features of how these two environments compare can be observed in Table 7.1. After using both environments, we concluded that MiniWorld would be

Environment Features		
Feature	MineRL	MiniWorld
3D environment	true	true
first-person RGB observation	true	true
open-source	false	true
modifiable	difficult	easy
performance	slow	fast
out-of-the-box features	many	few
ease of use	medium	easy

Table 7.1: Feature comparison of MineRL and MiniWorld

easier to use and that the ability to modify it would allow us to create an environment that more closely represents the problem statement of the study. The performance of the environments are compared in the next section.

Stellenbosch University https://scholar.sun.ac.za

7.3 Performance

This section considers the performance and the resource requirements of MineRL and MiniWorld. In previously published deep Q-learning results, millions of transitions were used to train the respective deep Q-learning agents [14, 23]. It was expected that it would be necessary for us, too, to generate large amounts of data, therefore the speed at which new transitions could be generated by the respective environments was considered. A lightweight environment with low computational resource requirements was also needed in order to implement a distributed version of a deep Q-learning algorithm where multiple environments are instantiated.

We first compare the time to initialise each environment, which is referred to as the *startup time*. We also compare the time to reset an episode in each environment, which is referred to as the *reset time*. Lastly, we measure the time to perform an action in each environment, which is referred to as the *step time*. The various timing measurements can be observed in Figure 7.1. The startup time for MiniWorld is shown to be almost negligible

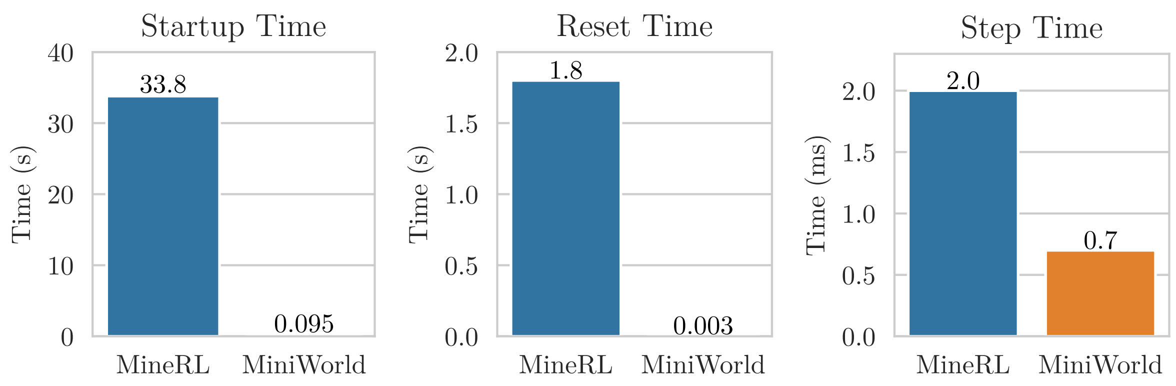

Figure 7.1: The mean startup time, mean reset time and mean step time of the simulation environments MineRL and MiniWorld. The average over 100 trials was taken to obtain this result.

compared to that of MineRL. MineRL's long startup time would not be a big concern when training an agent, as the startup only takes place at the start of the training process. However, the faster startup time would indeed help to shorten the debugging process, as the time to test a portion of the program that depends on the environment would be much shorter. It was also observed that both the time to reset and the time to step of MiniWorld are much faster than those of MineRL. Resetting the MineRL environment would require regenerating the entire Minecraft world, which is computationally intensive. Slow reset times could significantly slow down the speed at which new data is generated, especially in scenarios where the environment is often required to be reset.

In this study, data generation was distributed where memory usage can be expensive, as it was required to instantiate multiple environments. As the number of environments being

Stellenbosch University https://scholar.sun.ac.za

used increases, memory usage increases by the same factor. The memory requirements for MineRL and MiniWorld were therefore also compared. It was found that a MineRL environment would require 4GB of system memory while a MiniWorld environment would require only 20MB. In this study, where it would be required to instantiate multiple environments, MiniWorld's lower memory requirement made it the more viable option.

7.4 Environment modifications

Based on the findings of the previous two sections, it was decided to continue this study in the MiniWorld environment. This section examines the changes made to the environment in order to comply with the problem statement. From the problem statement it was deduced that the agent had to perform the defined task in an environment that is representative of a mine. A mine that consists of a specifiable number of rooms thus had to be created in MiniWorld. The layout of the mine had to be stochastic, therefore the rooms, doorways and entities had to be randomly located each time the environment is initialised. This section discusses the main changes made to the environment in order to comply with these requirements.

7.4.1 Randomised map layouts

To train a DQN that is able to solve various versions of the same problem, it would be required that the layout of the problem be randomised between episodes. If the layout of all the episodes was precisely the same, then the agent would only have to learn a single sequence of actions to solve the problem. In addition to randomising the entity locations between episodes, we required the room layout of the mine to be random as well.

MiniWorld has the functionality to easily add rooms to the environment and to connect them with doorways. To generate an environment that consists of multiple rooms that are connected to each other, the boundaries of the rooms and the locations of the doorways had to be specified. We wanted to have the functionality to generate random layouts by only specifying the number of rooms and the size of each room.

Functionality was added to allow layouts that consist of multiple rooms to be automatically generated. During the process of generating a layout, it had to be verified that rooms do not overlap. It was therefore necessary to check whether a new room would intersect with any existing rooms. The rooms were then connected with doorways such that there was a pathway that would allow all rooms to be accessed. This added functionality allowed us to generate a vast amount of different layouts and saved us having to design these layouts ourselves. The agent could therefore be trained in an environment that would be randomly initialised each time. This would make for an agent that would be resilient to the stochastic nature of the environment.

Stellenbosch University https://scholar.sun.ac.za

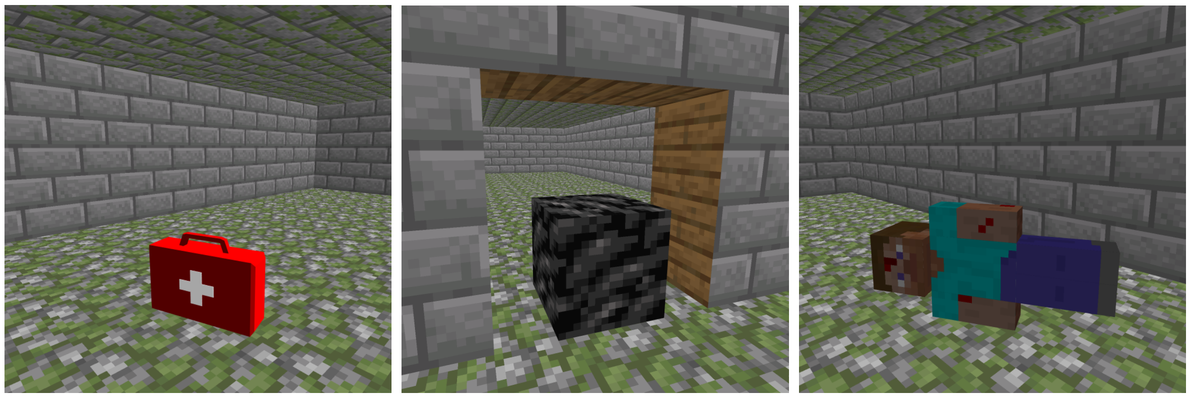

Figure 7.2: The following entities were added to the MiniWorld environment: first-aid kit (left), obstacle or boulder (middle), and an injured miner (right).

7.4.2 Environment functionality

In the process of delivering the first-aid kit to the injured miner, the agent would have to deal with some obstacles. The desired assets for this were not included in MiniWorld, but models could easily be imported and some functionality added to achieve the required behaviour. The following entities were added to MiniWorld: first-aid kit, boulder, injured miner – shown in Figure 7.2.

MiniWorld already had the functionality to pick up, carry and drop items. The ability for an agent to exchange the item it is carrying with an item on the floor would greatly aid it in completing the proposed tasks. The modification was made to allow the agent to easily exchange the item it was carrying with another item. Boulders were added which could obstruct the agent's path, but the agent is able to pick up boulders and to move them to other locations. The above mentioned new entities could now be used to create interesting problems in the MiniWorld environment.

7.5 Summary

In this chapter, MiniWorld and MineRL were compared to investigate each as a possible test-bed for this study. Although the feature-rich MineRL environment was a very attractive option, it was found that its performance is slow, that it has large memory requirements, and that it is difficult to add additional functionality to the environment. MiniWorld had limited features, but its fast performance and the ability to easily modify it, made it the environment of choice. Although various features had to be added to MiniWorld, the open-source project made the process of changing the environment straightforward. With the newly added entities and functionality, we were able to create various versions of the proposed problem in the MiniWorld environment.

This concludes the investigation regarding the simulation environment. We now proceed to discuss the implementation of a distributed deep Q-learning algorithm to solve problems in the modified version of MiniWorld.

Stellenbosch University https://scholar.sun.ac.za

Chapter 8

Implementation of the deep Q-learning algorithm

Having discussed the simulation environment, we proceed to discuss the implementation of a distributed deep Q-learning algorithm with *prioritised experience replay* (PER). Distributed computing entails that the algorithm is divided into smaller parts that can be executed in parallel, to improve performance. It can be utilised in many ways to accelerate *deep reinforcement learning* (DRL) algorithms. For example, the approach by Dean *et al.* [7] can be used to parallelise the computation of gradients, in order to optimise the *artificial neural network* (ANN) at a faster rate.

In our case, we followed an approach similar to that of Horgan *et al.* [16], where the generation of environmental transitions is distributed. This is done by decoupling the process of updating the *deep Q-network* (DQN) from generating transitions. This allows generating transitions at much faster rates, which results in a large amount of data available to update the DQN. Since an extensive amount of data is available, PER is used to focus on the most important transitions. It is expected that giving the algorithm access to more data can significantly improve performance, especially when dealing with sparse rewards. Our implementation focused on distributed data-generation but this can be combined with approaches such as distributed gradient computation.

The hardware and software available for this study are now outlined here. The design of the system as a whole is then reviewed, followed by a discussion of each individual component of the system. Descriptions of the components are aided by diagrams that contain pseudo code to describe the basic functionality of the different classes. More detailed versions of these algorithms are included in Appendix C. Next, we discuss the methods used to augment observations to address partial observability. We then review the architecture of the DNN used for the deep Q-learning algorithm. Finally, we test how well the proposed approaches work to address the partial observability of the environment.

Stellenbosch University https://scholar.sun.ac.za

8.1 Hardware and software

A Linux system with specifications listed in Table 8.1 was used for the software development and to perform experiments in this study. A 6 core, 12 thread Ryzen *central processing unit*

Computer Specifications	
CPU	AMD Ryzen 3600 (6 cores, 12 threads)
RAM	64GB
GPU	Nvidia RTX2070 Super
Storage	512GB SSD

Table 8.1: Specifications of the computer used to conduct the study.

(CPU) was available and would allow having up to 12 simultaneous processes which are utilised to distribute data generation. Multiple instances of the environment would be loaded in memory, and additionally, the replay buffer would require a large amount of memory. The system had 64GB of *random access memory* (RAM), to meet the memory requirements. *Graphics processing unit*s (GPUs) are very effective in doing parallel computations. As the backpropagation algorithm entails matrix multiplication, where operations can be performed in parallel, a GPU could be utilised to accelerate this process significantly. An Nvidia RTX2070 super GPU was therefore employed, which dramatically accelerated the process of updating the DNN.

Python was used for development in our research. The essential packages used for development included PyTorch and Ray. PyTorch is framework used for developing deep learning models. PyTorch provides tensor computation with GPU acceleration and allows for the construction of DNNs on a tape-based automatic differentiation system. The core logic of PyTorch is written in C++, which means a lower overhead compared to other frameworks [32]. In this study, it provided a high-performance environment and was used to maximise the available GPU to accelerate learning. Ray is a Python package that is used to perform distributed computation. Ray is very intuitive to use, and Python classes could easily be converted to Ray actors that could be executed on different threads of the CPU simultaneously.

8.2 System overview

An overview of the architecture of the distributed prioritised deep Q-learning algorithm is shown in Figure 8.1. The architecture consists of multiple components that are all contained in the main simulation class. The simulation class is responsible for instantiating, initialising and controlling the different components of the system. Each component in Figure 8.1 is a Ray actor and therefore the methods of the different objects can be executed simultaneously.

Stellenbosch University https://scholar.sun.ac.za

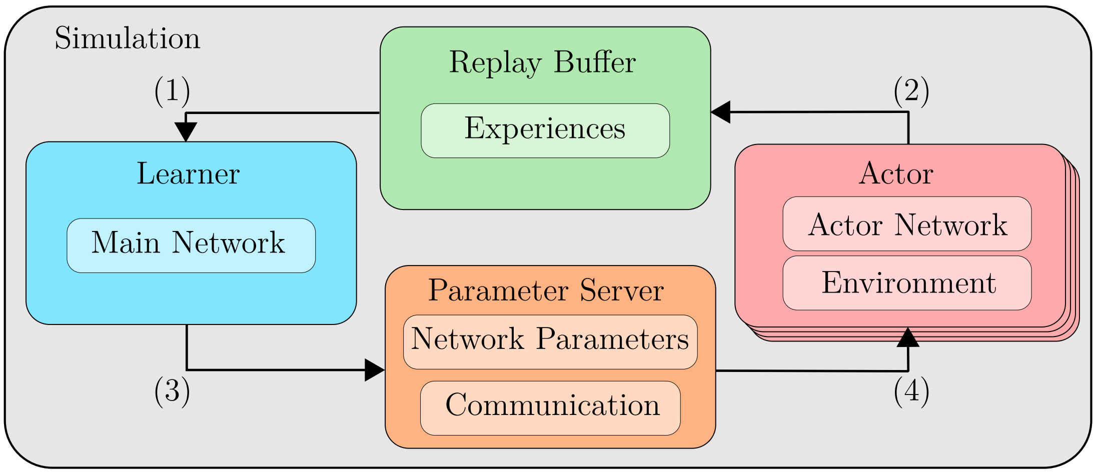

Figure 8.1: The diagram illustrates the interaction between the main components of the simulation class. Interaction between different components is indicated with arrows. Actors (red) are responsible for generating new transitions in their respective environments and storing them in a centralised replay buffer (2). The learner (blue) samples transitions from the replay buffer (1) and uses it to update the main network. The learner uploads the latest parameter values of the main network to the parameter server (orange) from time to time (3). The actors query the parameter server for new network parameter values between completing their respective episodes. If available, they update their respective networks with the latest parameter values (4).

The process of generating transitions in the environment is referred to as *acting*, performed by the *actor(s)*. Furthermore, we refer to the process of updating the DQN as *learning*, performed by what we refer to as the *learner*. The algorithm decouples acting from learning and allows multiple actors to generate transitions in parallel. As seen in Figure 8.1, the actors and learner have access to a centralised replay buffer. The data generated by the actors is stored (2) in the replay buffer. The learner again samples from the replay buffer to update the main DQN to which we refer as the *main network*. The actors and learner also have access to a shared parameter server which allows these components to communicate (3,4). Communication entails, amongst other things, to give the actors access from time to time to the learner's latest network parameters. The parameter server can also be used to notify the actors to stop acting, or to change the version of their environment if *curriculum learning* (CL) is used.

We use the approach similar to Horgan *et al.* [16], where actors asynchronously generate and add data to the replay buffer as fast as possible. The learner again asynchronously samples data from the replay buffer and updates the main network as fast as possible. The rate at which the system generates data and adds it to the replay buffer can be altered by simply changing the number of actors used to generate data.

8.3 Replay buffer for storing experience

In this study, we implemented a prioritised replay buffer [39], as discussed in Section 6.2. This was used as centralised storage for environment-generated transitions. The prioritised

Stellenbosch University https://scholar.sun.ac.za

replay buffer is shown in Algorithm 12. New transitions, along with priorities, are added

Algorithm 12: Prioritised replay buffer based on the work of Schaul *et al.* [39]. The algorithm has the functionality to store transitions along with priorities, sample transitions according to priorities, and finally to update the priorities of existing transitions.

```
def init(capacity, $\alpha$, $\eta$):
    initialise capacity, $\alpha$, and $\eta$
def add_batch(batch, $\boldsymbol{\delta}$):
    calculate priorities from $\boldsymbol{\delta}$
    add batch to storage
    add priorities to sum-tree
def sample_batch(batch_size, $\beta$):
    sample indices from sum-tree of batch_size
    calculate IS weights using the sampled indices' priorities and $\beta$
    use indices to collect transitions from storage
    return transitions, indices, is_weights
def update_priorities(indices, $\boldsymbol{\delta}$):
    calculate new priorities from $\boldsymbol{\delta}$
    update priorities at specified indices of sum-tree
```

to the replay buffer by calling the `add_batch` function. The `add_batch` function accepts a batch of transitions along with the corresponding TD errors. The TD errors are used to calculate the transitions' priorities. The transitions are then added to the storage, and their priorities are added to a sum-tree data structure. Recall from Section 6.2.1 that the sum-tree data structure allows for more efficient prioritised sampling. Transitions are added to the replay buffer in a *first-in-first-out* (FIFO) manner, i.e. if the replay buffer reaches its maximum capacity, new transitions replace the oldest transitions in the replay buffer.

The learner samples the transitions from the replay buffer, randomly or by priority, to update the main network. The `sample_batch` method is used to sample a batch of transitions, where the batch size and beta β must be specified as arguments. The function uses the sum-tree to sample indices of transitions according to the priorities that were assigned earlier. Priorities are then used to calculate the *importance-sampling* (IS) weights using Equation 6.10. The function returns the sampled transitions, the indices where they are stored, and their IS weights. The indices are later used by the learner to update the priorities of the sampled transitions. Priorities of existing transitions in the replay buffer can be updated with the `update_priorities` function. The indices of the transitions that must be updated, and corresponding TD errors, must be specified. New priorities are then calculated and the relevant entries of the sum-tree are updated.

Stellenbosch University https://scholar.sun.ac.za

8.4 Parameter server for communication

Decoupling the learner and actors is computationally very advantageous, but makes it problematic for them to communicate with each other. The problem is that the learner and actors cannot be accessed externally while cycling through their respective instructions, and therefore these objects cannot directly interact with each other. Communication is necessary for the actors to fetch the most recent network parameters from the learner. Communication is also necessary to notify actors when their respective environments should change if CL is used.

We decided to address this problem by using a parameter server, similar to Yoon [51]. The learner and actors have access to a central parameter server, shown in Algorithm 13. It acts as an intermediate communication medium between them. The parameter server's attributes can be changed by using the `update_attributes` function. These attributes include a copy of the main network's parameters and the CL phase. The value of these attributes can again be accessed with the `fetch_attributes` function. The learner and the actors have independent cycles where they access the parameter server to request the most recent values of attributes or to update attributes.

Algorithm 13: Parameter server for communication between actors and learner. Attributes are updated with the `update_attributes` function. Attribute values are fetched with the `fetch_attributes` function.

```
def init():
    initialise network_parameters
    initialise phase
def update_attributes(network_parameters, phase):
    update the values of the various attributes
def fetch_attributes():
    return the various attributes
    return network_parameters, phase
```

8.5 Learning from experience in the replay buffer

The learner is responsible for sampling batches of transitions from the replay buffer and updating the parameters of the main network. We separate this process into two functions, namely `load_minibuffer` and `run_learner`. The pseudo code for the learner is shown in Algorithm 14.

The `load_minibuffer` function samples batches of transitions from the replay buffer in the background and prepares and stores the data in the desired format in a mini buffer. While the `load_minibuffer` function is being executed in the background, the `run_learner` method optimises the main network. The learner optimises the main network

Stellenbosch University https://scholar.sun.ac.za

Algorithm 14: Learner for updating the main network. This algorithm is inspired by the work of Horgan *et al.* [16].

```
def init(replay_buffer, parameter_server):
    initialise main_network and target_network
    assign reference to replay_buffer and parameter_server
def load_minibuffer():
    while true do
        sample batch of transitions from replay buffer
        prepare batch for optimisation
        store batch in mini_buffer
    end
def run_learner(T):
    for $t \leftarrow 1$ to T do
        fetch batch from mini_buffer
        compute loss
        update parameters of main_network
        update priorities in replay buffer using TD errors
        periodically copy main_network to target_network
        periodically copy main_network to parameter_server
        if CL then
            update CL phase according to specified criteria
        end
    end
```

for a specified number T of iterations. For each iteration, a batch of transitions is collected from the mini buffer. The experience is already in the correct format, i.e. PyTorch tensors, and can be utilised directly by the learner to pass it through the main network, and to calculate the TD targets by using the n-step return.

We used the backpropagation algorithm to compute the gradients on the network parameters. The next step was to adjust the parameter values of the main network using an optimiser of choice. Kingma and Ba [20] state that *adaptive moment estimation* (Adam) is an appropriate optimiser for non-stationary objectives and problems with noisy and sparse gradients. We therefore hypothesised that Adam would be well suited for the problem we are addressing, as the *reinforcement learning* (RL) problem is highly non-stationary, and sparse rewards lead to noisy and sparse gradients. According to Vitay [48], Adam works out-of-the-box with minimal hyperparameter tuning (learning rate, momentum), at the expense of minimum that may be slightly worse. Furthermore, Adam is preferred by Vitay [48] for DRL, as the goal is to find a policy that is able to solve the problem, and not to perfectly minimise the loss function.

After the main network's parameters are updated, the TD errors are used to update the priorities of the transitions which were sampled from the replay buffer. The latest values of the main network's parameters are stored on the parameter server from time to time to allow actors to access the most recent version of the main network.

Stellenbosch University https://scholar.sun.ac.za

8.6 Generating new experience

Actors are used for generating transitions and the pseudo code for the Actor class is shown in Algorithm 15. Multiple instances of this class can be initialised to generate transitions

Algorithm 15: Actor for generating transitions. This algorithm is inspired by the work of Horgan *et al.* [16].

```
def init(replay_buffer, parameter_server):
    initialise reference to replay_buffer and parameter_server
    initialise actor_network
    initialise wrapped environment
    initialise local_buffer
def run_actor($T$):
    remote call to parameter_server to update actor_network
    reset environment and retrieve initial state
    for $t \leftarrow 1$ to $T$ do
        observe state and perform action according to $\varepsilon$-greedy policy
        add $n$-step transition to local_buffer
        if size of local_buffer $\geq B$ then
            compute TD errors of transitions in local_buffer
            move transitions of local_buffer to centralised replay_buffer
            clear local_buffer
        end
        if episode is done then
            remote call to parameter_server to update actor_network
            if CL then
                remote call to parameter_server to update phase update
            end
            reset environment and retrieve initial state
        end
    end
```

in parallel.

Each actor has access to an instance of the simulation environment. Additionally, the actors use the latest available knowledge from the learner in order to perform actions (ε-greedy) in their respective environments. Therefore, each actor has access to the parameter server to update their respective networks with the latest parameter values. The learner uploads the main network's latest parameter values to the parameter server every 500 network updates. Actors are asynchronously completing episodes in their respective environments and are verifying the parameter server between episodes for updated values of the main network's parameters. If available, they duplicate the new parameters to their respective networks. Therefore, the actors' networks age with 500 network update steps every time they update their respective networks' parameters. We decided to update the

Stellenbosch University https://scholar.sun.ac.za

actors' network parameters every 500 completed network update steps to also ensure the actors use very recent network parameters to generate transitions.

The actor class has a method `run_actor`. When this method is called, the actor interacts, for a specified number of steps or indefinitely, with its simulation environment generating transitions. The transitions that each actor generates are stored in a local buffer and are periodically moved (every B transitions) to the centralised replay buffer.

As we used PER, the priorities of newly generated transition had to be calculated at some point. Schaul *et al.* [39] assigns maximal priority to newly generated transitions to ensure that all new transitions are at least sampled once. Horgan *et al.* [16] instead suggests calculating priorities of newly generated transitions immediately. The reason for this is that the distributed prioritised experience replay algorithm is capable of generating large amounts of data, and by immediately assigning the correct priorities to transitions, the learner only focuses on the most important transitions. We used the latter approach. Between episodes, actors request the latest parameter values of the main network and also check for environment phase updates (if CL is enabled).

8.7 Simulation

We now review the simulation class in which the previously discussed components were all hosted. The simulation class is shown in Algorithm 16. With the initialisation of the

Algorithm 16: Simulation

```
1  initialise replay_buffer
2  initialise parameter_server
3  initialise learner
4
5  run actors to pre-load replay_buffer
6  wait for pre-loading phase to complete
7  run learner for specified number of network updates
8  run actors indefinitely
9  wait for learner to finish
10 wait for actors to terminate
```

simulation object, the replay buffer, parameter server, learner and actors are all initialised. The simulation class provides the actors and the learner with the centralised replay buffer and parameter server.

After all the relevant objects are initialised, the simulation class runs each actor for $50,000$ steps to pre-load the replay buffer. We pre-load the replay buffer to provide the learner with data to immediately start improving the main network. After the replay buffer is pre-loaded, the `run_learner` function of the learner is called with the specified number of network update steps as an argument. The simulation class reruns each actor. The

Stellenbosch University https://scholar.sun.ac.za

actors will now indefinitely generate transitions in their respective environments until the learner has completed the specified number of network updates. Recall that the learner and actors are executed simultaneously. The program and therefore all actors are terminated after the `run_learner` function has been completed.

8.8 Deep neural network design

In this section, we discuss the architecture of the DNN used for the deep Q-learning agent. We also discuss an important component that we refer to as the *environment wrapper*. The environment wrapper is responsible for the format of the input applied to the network and is therefore crucial in the discussion of the network's design. Once we have discussed the environment wrapper, we review the layout of the DNN.

8.8.1 Environment wrapper

The environment wrapper is an intermediate communication medium between the actors and their environments. It allows us to process observations from the environment before returning it to the actors. It also allows us to process actions from the actors before they perform these in their respective environments. We specifically used the environment wrapper in this study to process observations. The goal was to manipulate observations to address the problem of partial observability.

Pre-processing of frames Applying convolutional layers to high-dimensional data can be computationally very expensive. This problem was addressed by Mnih *et al.* [23, 24] by reducing the dimensionality of observation frames. A reduction in input dimensionality was achieved by extracting the luminance from observation frames and rescaling frames to achieve lower resolution grayscale images. We did not have to downscale observation frames as we configured the environment to render relatively low-resolution frames with a dimensionality of $\mathbb{R}^{3\times 44\times 76}$. The convention we used to define the dimensionality of image data is $\mathbb{R}^{C\times H\times W}$, where C is the number of channels (e.g. RGB channels), H is the height of the image, and W is the width of the image. Furthermore, we decided instead to retain the colour channels and not to grayscale the observation frames. This decision was made based on the problem addressed, where a red first-aid kit must be located, fetched and delivered to an injured miner. As colour can be important for human beings to classify objects, we suspected that colour information would also be a valuable feature to our agent in completing the specified task.

Frame-stacking To address the problem of partial observability, we investigated an approach, called *frame-stacking*, used by Mnih *et al.* [23, 24]. Frame-stacking entails stacking and keeping the last number of frames observed in memory. We stacked the

Stellenbosch University https://scholar.sun.ac.za

frames along the dimension of the colour channels, as the RGB colour channels are present in each frame. The image observation that the MiniWorld environment returns at each time step is an RGB frame with a dimensionality of $\mathbb{R}^{3\times 44\times 76}$. After the environment wrapper stacked the last f frames, the dimensionality of the resultant observation is $\mathbb{R}^{3f\times 44\times 76}$.

Action history In addition to frame-stacking, we also investigated an approach we refer to as *action history*. It entails keeping the last h number of actions performed by the actor in memory. We define the action history vector as

$$\mathbf{A}_{\text{history}} = [A_{t-1}, A_{t-2}, \ldots, A_{t-h}], \tag{8.1}$$

where h is the length of the sequence of actions kept in memory and t is the current time step. We decided to one-hot encode each action in the $\mathbf{A}_{\text{history}}$. We then flattened the result to a vector. The one-hot feature representation yielded better performance than than assigning a numerical value to each action. As there are $|\mathcal{A}|$ possible actions to perform in the simulation environment, the one-hot encoded action history vector has a dimensionality of $\mathbb{R}^{|\mathcal{A}|h}$. For our problem the cardinality of the action space is five $|\mathcal{A}| = 6$, therefore the action history vector has a dimensionality of $\mathbb{R}^{6h}$. The one-hot encoded action history vector could be used in combination with the stacked frame observation as a representation of the state of the agent.

8.8.2 Deep Q-network architecture

Having discussed the format of observations returned by the environment wrapper, we now review the architecture of the DNN used for the deep Q-learning agent. We augmented the environment's observation with frame-stacking and action history to address the partial observability of the environment, We therefore had to design a network that would accommodate these augmentations. In Figure 8.2 we present the architecture of the network. It comprises of a CNN that is followed by an FNN.

Stellenbosch University https://scholar.sun.ac.za

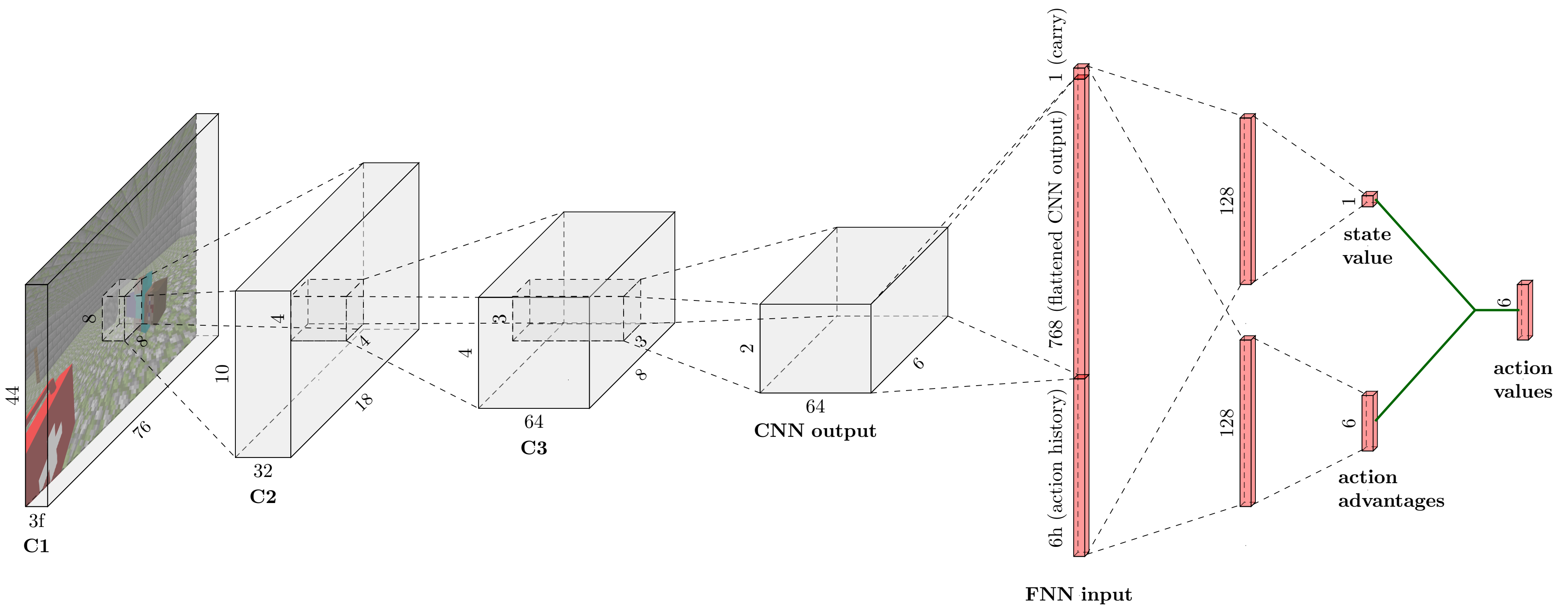

Figure 8.2: The DNN architecture we used for our research – inspired by Mnih *et al.* [23, 24], and Wang *et al.* [50]. The network starts with three convolutional layers (C1, C2, C3), each followed by a ReLU activation function. The network receives an input volume of $\mathbb{R}^{3f\times 44\times 76}$, where f is the number of frames stacked. The filters of the first convolutional layer, C1, have a stride of four, a receptive field of $\mathbb{R}^{8\times 8}$ and produce a volume of $\mathbb{R}^{32\times 10\times 18}$. The filters of C2 have a stride of two, a receptive field of $\mathbb{R}^{4\times 4}$ and produce a volume of $\mathbb{R}^{64\times 4\times 8}$. The filters of the last convolutional layer, C3, have a stride of one, a receptive field of $\mathbb{R}^{3\times 3}$ and produce a volume of $\mathbb{R}^{64\times 2\times 6}$. The output volume of C3 is flattened and results in a vector with a dimensionality of $\mathbb{R}^{768}$. Recall that the one-hot encoded action history has a dimensionality of $\mathbb{R}^{6h}$, where h is the length of the history. The action history and the one-hot carrying indicator are then concatenated to the flattened output of C3 to obtain the input to the FNN. The FNN input therefore has a dimensionality of $\mathbb{R}^{768+6h+1}$. The FNN consists of two sub-networks. We refer to the first sub-network as the *state-value network*. This sub-network consists of two layers. The first layer consists of 128 neurons, followed by a ReLU activation function. The second layer produces a scalar value that is referred to as the *estimated state-value*. We refer to the second sub-network as the *action-advantage network*. The first layer of this sub-network also consists of 128 neurons and is also followed by a ReLU activation function. The second layer produces a vector that contains an *action advantage* for each available action. The architecture then combines state value and action advantages to obtain the action values.

Stellenbosch University https://scholar.sun.ac.za

Convolutional neural network The architecture of the CNN is based on the work by Mnih *et al.* [23, 24]. The CNN consists of three convolutional layers. Refer to Figure 8.2 for the specifications of the filters of each layer and the dimensionality of the output volume at each layer. We use Equation 5.45 and Equation 5.46 to obtain the dimensions of the output at each layer. The CNN produces an output volume of $\mathbb{R}^{64\times2\times6}$ at C3. This volume is then flattened to a vector of size $\mathbb{R}^{768}$.

Feed-forward neural network We still have to accommodate the action history as input to the FNN. This problem was addressed by merely concatenating the one-hot encoded action history $\mathbb{R}^{6h}$ to the flattened output of $C3$. Additional we also concatenated a binary value that indicates if the agent is carrying an item. This resulted in the input to the FNN having a dimensionality of $\mathbb{R}^{768+6h+1}$. The design of the linear layers was based on the dueling architecture, which we discussed in Section 6.1.5. Therefore the input to the FNN splits into two streams. The first estimates the state value, and the second estimates the action advantages. The state value and action advantages are then combined to obtain the action values. Refer to Figure 8.2 for the specifications of the FNN's layers.

8.8.3 Frame-stacking and action history experiments

With all the components of the system explained, we can now report on how we tested the agent. Before going on to the experimental phase of the study where the focus would be on solving sparse-reward problems, we first tested whether the inclusion of frame-stacking and action history assist in addressing the problem of partial observability.

Here, we introduce the most simplest version of our problem statement where the agent, first-aid kit and injured miner are all placed at random locations in a single room. The agent has to locate the first-aid kit, collect it, and transport it to the miner. We expected that task in the partially observable environment to prove difficult without any memory of previous actions performed or observations seen.

We therefore performed the following four experiments: frame-stacking and action history disabled, only frame-stacking enabled, only action history enabled, and frame-stacking and action history enabled. When frame-stacking was enabled, we decided to stack the last four frames observed by the agent, similarly to Mnih *et al.* [23, 24]. We arbitrarily decided on an action history that would hold the last 20 actions performed by the agent. The configuration of the agent is shown in Table 8.2. The values of the hyperparameters are based on respective papers by Mnih *et al.* [24], Schaul *et al.* [39], Horgan *et al.* [16] and Hessel *et al.* [14]. The action space of the agent is shown in Table 8.3.

We used the mean return per episode achieved by the actors as a function of the completed network updates to measure performance. The result of this experiment is shown in Figure 8.3. This type of result is referred to as a *learning curve*. Recall from Section 8.5 that the learner is responsible for adjusting the parameters of the main network

Stellenbosch University https://scholar.sun.ac.za

Experimental configuration	
number of actors	7
ε (epsilon)	0.01 (all actors)
α (learning rate)	3×10^{-5}
batch size	32
optimiser	Adam
γ (discount rate)	0.99
n (n-step)	6
replay capacity	$1,048,576$
PER	enabled
ζ (prioritisation)	0.6
β_0 (IS)	0.4
η (prioritisation constant)	1×10^{-5}

Table 8.2: Default agent configuration. Refer to Appendix B for additional info on the chosen values.

Action space		
#	action	unit
1	move forward	0.5 units
2	move backward	0.5 units
3	turn right	9 degrees
4	turn left	9 degrees
5	pickup/drop/exchange item	-
6	no operation	-

Table 8.3: Action space of the environment.

that represents the agent's policy. Each time the main network parameters are adjusted is defined as a *network update step*. We measured performance as a function of the completed network update steps. Recall from Section 8.6 that the actors' network parameters age with 500 completed network update steps. Therefore the main network is evaluated every 500 network update steps. A *run* is defined as training the main network from start to finish for a specified number of network update steps. The results of this section present an average result of completing 30 runs. By averaging results over multiple runs a more reliable result is obtained. Since performing a single run was time-consuming enough, we did not perform more runs.

The result shows that when we included either frame-stacking or action history, the agent achieved higher mean episode returns earlier in training. Surprisingly, the experiment in which we did not enable one of these approaches had similar performance to the other experiments at the last network update steps. It is also interesting that there was no benefit to combine frame-stacking and action history. The environment is mainly static with no dynamic entities. We therefore speculated that there would be no need for frame-stacking,

Stellenbosch University https://scholar.sun.ac.za

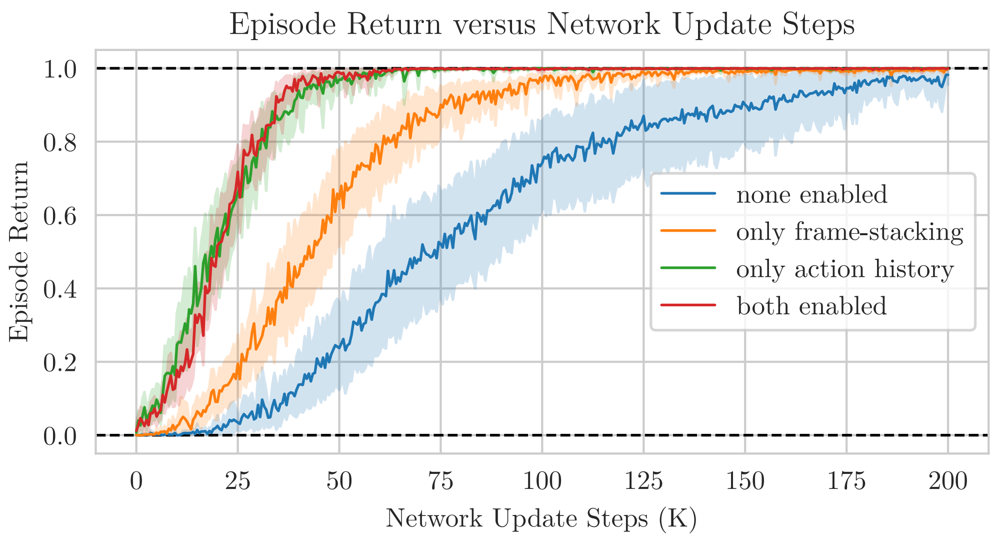

Figure 8.3: The mean episode return as a function of completed network update steps when enabling frame-stacking and action history is compared. The blue curve shows the performance when both frame-stacking and action history are disabled. The orange and green curves respectively show performance when only action history and only frame-stacking are enabled. The red curve shows the performance of the agent when frame-stacking and action history are combined. The maximum and minimum achievable returns are indicated by the dashed lines respectively at 1.0 and 0.0.

which allows for the capture of information on the movement of external entities. For this task, an action history with a capacity of 20 seemed to work well, but this value may be task-dependent. As this value proved to work well, we would continue to use it for the experiments reported in Chapter 9. Using frame-stacking is more expensive than using action history. As there was no clear benefit for it to be used in our environment, we would omit frame-stacking in future experiments to save on computational costs. However, frame-stacking would most likely be very important in environments with more dynamic elements.

8.9 Summary

In this chapter we considered the critical choices made during the development of the distributed deep Q-learning agent with prioritised experience replay. We discussed the main components of the system and how the they interact with each other. The components we discussed include the replay buffer, parameter server, learner and actor. The environment wrapper was also reviewed, as was the format of the input to the DNN. We examined the architecture of the DNN used for the deep Q-learning agent. Finally we tested the performance of frame-stacking compared to action history to address partial observability. We can now continue to the experimental phase, to investigate how different components of the system help to address the problem of sparse rewards.

Stellenbosch University https://scholar.sun.ac.za

Chapter 9

Experiments and results

In this chapter we report on several experiments that deal with sparse-reward problems. In all the experiments the goal of the agent is to deliver a first-aid kit to a miner, as discussed in Section 1.5. The mine consists of multiple rooms with obstacles, which make it challenging to navigate. To achieve this goal multiple sub-tasks must be completed, and a positive reward signal is only returned when the entire task is completed. Sub-tasks include navigating the mine, removing obstacles, fetching the first-aid kit, and finding the miner.

In the first section of this chapter, we investigate three main improvements made to the deep Q-learning algorithm. We report on an ablation study where we investigated the impact each modification had on the system as whole. We then further analyse the modifications to better understand the influence each has and to find hyperparameters that achieve the best performance. We consider the influence that distributing data generation has on the execution time of experiments. We also consider the effect which the rate at which data is generated has on the agent's performance. We then examine the influence of *prioritised experience replay* (PER). We investigate whether certain transitions were sampled more frequently with PER and how this affected performance. Lastly we show how we tested and found the best performing value for the n-step return.

In the second section of this chapter, we report on the capability of the agent to learn to perform a task where rewards are even more sparse. Exploration in this problem was more challenging, since the agent had to perform a longer sequence of correct actions to obtain the reward signal. We also look at whether *curriculum learning* (CL) and *domain randomisation* (DR) could assist the agent to solve the more challenging exploration problem.

The chapter ends off with a section where the deep Q-learning agent is tested on problems with a larger mine with more rooms. The goal was to observe at what stage the agent is would no longer be capable of learning optimal policies. We also show how we trained the agent using a combination of CL and DR on the larger version of the problem to observe if this approach improved the agent's performance.

Stellenbosch University https://scholar.sun.ac.za

9.1 Experiment 1: Testing modifications on a sparse-reward problem

We now introduce the first version of the sparse-reward problem discussed in Section 1.5. We investigated the influence the three modifications has on the performance of the deep Q-learning algorithm to solve this problem. The default configuration shown in Table 8.2 was used to perform the experiments on this chapter, except if stated otherwise. Optimal values for hyperparameters are depended on the specific problem. Our default configuration was based on the relevant papers by Mnih *et al.* [23, 24], Schaul *et al.* [39], Hessel *et al.* [14] and the supplementary experiments in Appendix B. We used this as a starting configuration. In the experiments, we tested for an optimal value for the number of actors, prioritisation (ζ) and the n-step return.

9.1.1 Description of the task

The first task consisted of two rooms connected by a doorway, which are representative of a small part of a mine. In Figure 9.1 we illustrate the task as a top-down view. The

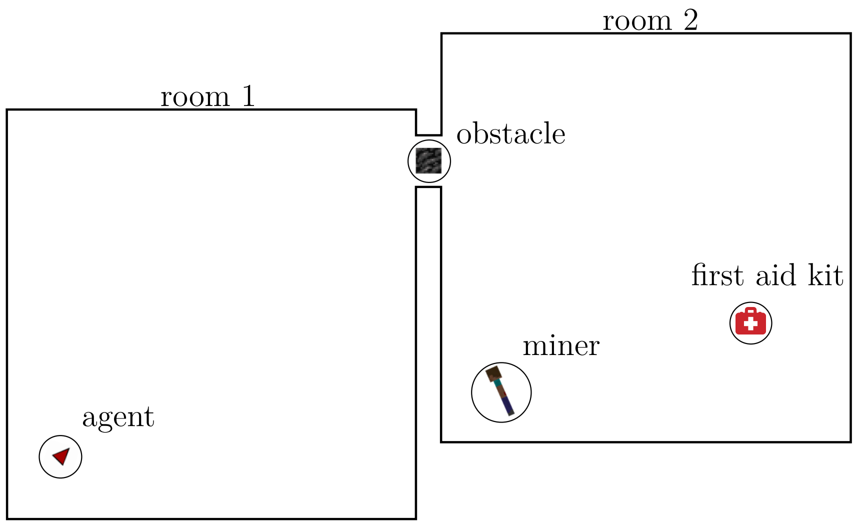

Figure 9.1: Problem one: A top-down illustration of the sparse-reward problem. The environment consists of two rooms. The agent is randomly placed in room 1. The first-aid kit and an injured miner are randomly placed in room 2. An obstacle obstructs the way between room 1 and room 2.

locations of the rooms and the doorway are randomised between episodes. The doorway is obstructed by an obstacle which represents debris that can occur in a collapsed mine. The agent is placed in the first room at a random location, facing a random direction, and an injured miner and a first-aid kit are placed at random locations in the second room. The agent's goal is to find the first-aid kit and deliver it to the immobilised miner. The size of each room is 16×16 units. The agent can interact with the environment by using the

Stellenbosch University https://scholar.sun.ac.za

action space defined in Table 8.3. The action space was the same for all experiments in this chapter.

The agent has to navigate from the first room to the doorway. An obstacle obstructing the doorway must be picked up and moved out of the way to proceed to the second room. The agent then has to locate the first-aid kit in the second room, pick it up and take it to the miner. The reward function returns a reward of 1.0 when the miner receives the first-aid kit and returns a reward of 0.0 for all other time steps. In each episode of this task, the agent has 1000 time steps to complete the task, after which the episode ends. We limited the number of time steps in the problem to enable us to evaluate policies over multiple episodes. The required number of steps to complete the task would be far fewer than the limit we set, but we wanted to give the agent additional time steps to explore the environment.

9.1.2 Ablation study

We performed an ablation study to observe the influence of each modification on the performance of the deep Q-learning algorithm. The modifications tested were distributed data generation, *prioritised experience replay* (PER), and n-step returns. We tested the deep Q-learning algorithm with all modifications enabled, i.e. the default configuration shown in Table 8.2. We refer to this configuration as the *all included* algorithm. We then tested configurations where in each test a single modification is disabled. We tested the following configurations:

1. *no n-step*, where the n-step update is removed, i.e. one-step TD is used;
2. *no distributed*, where only a single actor is used for data generation, i.e. no distributed data generation is used;
3. *no PER*, where we do not use PER, and standard ER is used; and
4. *none included*, where we disable all modifications.

Recall from Section 6.1.1 that the replay ratio indicates the rate at which the main network is updated relative to the rate at which new transitions are generated. For all configurations, except for the *no distributed* and *none included* configurations, seven actors were used to generate data. The resultant replay ratio was 0.023. Only one actor was used to generate data for the *no distributed* and *none included* configuration and the resultant replay ratio was 0.111. In Section 9.1.3 we further investigate how the number of actors affected the replay ratio.

We measured each configuration's performance while training the policy of the agent, i.e. the main network. In Figure 9.2 we present the mean episode return as a function of the completed network update steps. This shows the performance penalty when disabling each

Stellenbosch University https://scholar.sun.ac.za

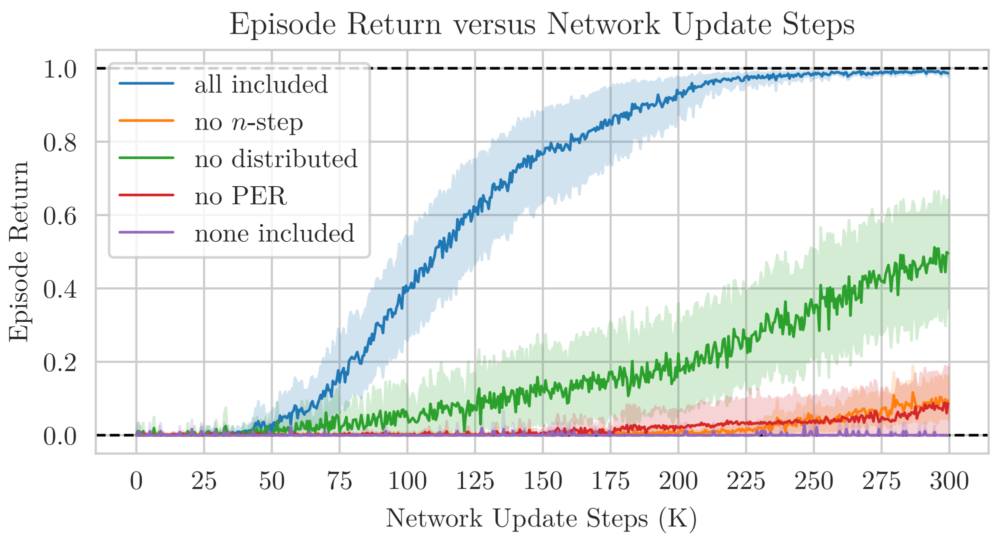

Figure 9.2: The mean episode return versus network update steps when different modifications are excluded. The maximum and minimum achievable returns are indicated by the dashed lines respectively at 1.0 and 0.0.

modification. We measured performance using the same method as described in Section 8.8.3. The results of this section present an average result after completing 30 runs.

In Figure 9.3 we also present the mean episode duration as a function of the completed network update steps. A shorter episode duration suggests better performance, as it is

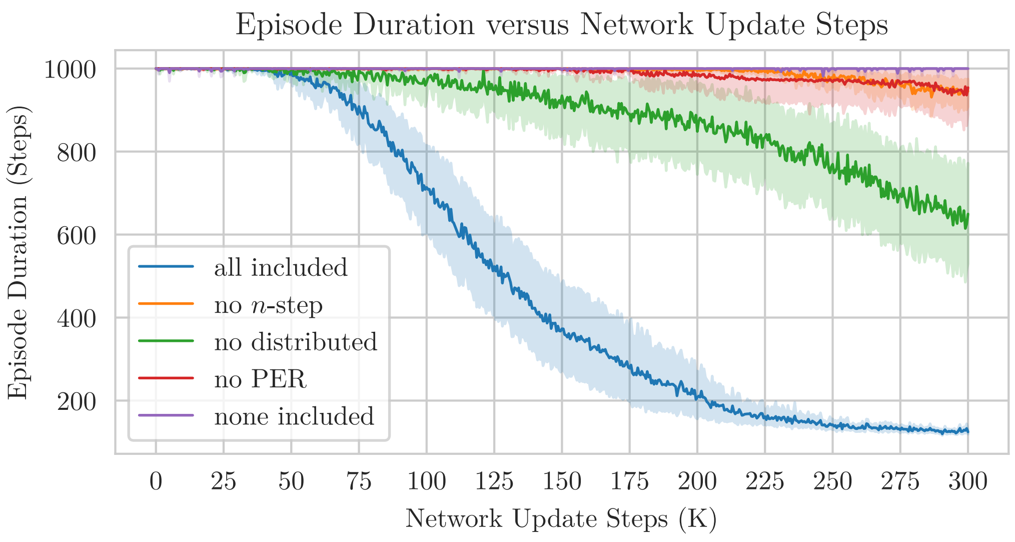

Figure 9.3: The mean episode duration versus network update steps when different modifications are excluded. A lower duration indicates better performance.

an indication that the task is performed in a smaller number of time steps. There is no way of ending the episode in fewer than the maximum allocated time steps, other than to complete the task successfully. The results shown in Figure 9.2 and Figure 9.3 are highly

Stellenbosch University https://scholar.sun.ac.za

correlated. We therefore only use the mean episode return to evaluate the rest of the algorithms of this section.

Some configurations are more demanding on the system, since different modifications are enabled. Accordingly execution times may vary. In Figure 9.4 we present the number of network updates steps completed per second for each configuration. It shows that some

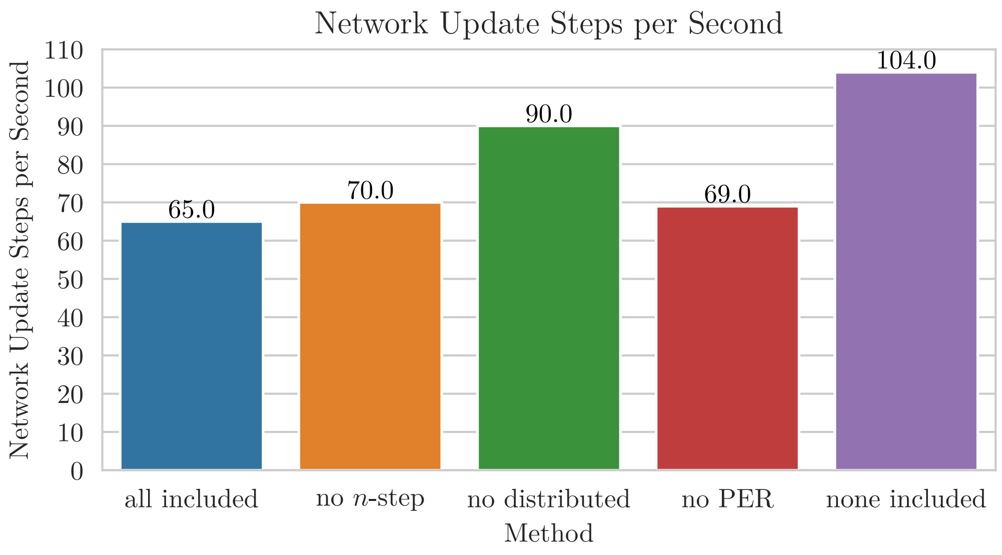

Figure 9.4: The number of network update steps completed per second for each configuration.

configurations perform network updates faster than others. For example, the *non included* configuration performs network updates much faster than the *all included* configuration.

For this reason, we also evaluated performance over wall-time. We present the mean episode return as a function of time in Figure 9.5. It shows the actual time required to achieve a certain level of performance. This result was very dependent on the implementation of the algorithm and the system used to perform the experiment. However, it was still an important result to consider as a goal was to train the best policy in the shortest time.

The results show that the final deep Q-learning algorithm with all modifications performed very well. By the end of the experiment, the algorithm obtained a near perfect mean episode return of 1.0. Therefore it was successful at completing the task almost all the time. Figure 9.5 shows that the all included agent was trained to a near perfect policy in 70 minutes. This result was achieved on the system described in Table 8.1, but training could be significantly accelerated by using more powerful hardware. The algorithm's performance decreased significantly when any of the modifications was disabled. Therefore distributed data generation, PER, and the n-step update are all very important in order to solve the sparse-reward problem described in Section 9.1.1. Furthermore, when all of the modifications were disabled, performance did not increase at all during the entire experiment.

Stellenbosch University https://scholar.sun.ac.za

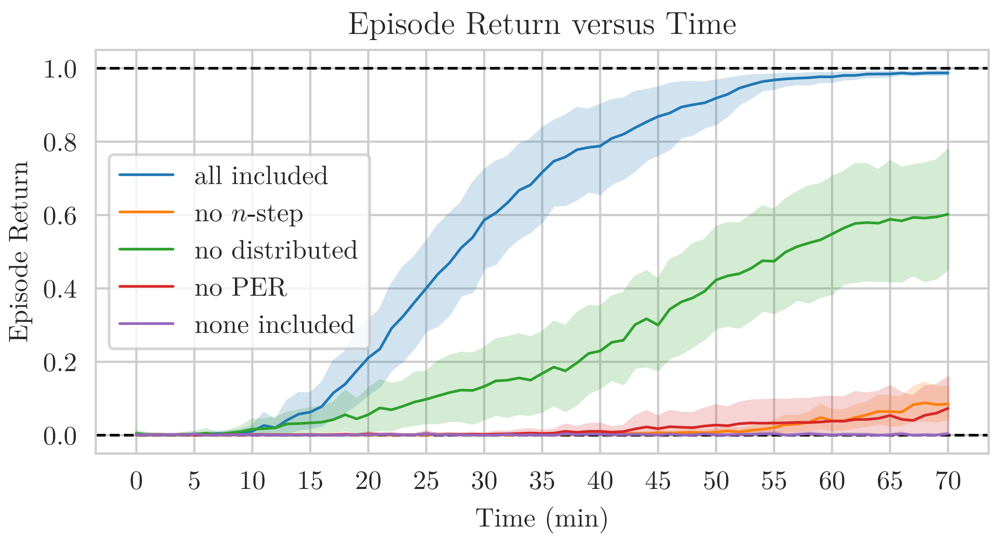

Figure 9.5: The mean episode return versus time when different modifications are excluded. The maximum and minimum achievable returns are indicated by the dashed lines respectively at 1.0 and 0.0.

9.1.3 Generating experience

Next, we describe how the rate at which new transitions are generated influenced performance. We first show that the distributed system allowed us to utilise additional resources to increase the data generation rate, thereby also lowering the replay ratio. The goal was to lower the replay ratio while keeping the duration of experiments as short as possible. The default experimental configuration in Table 8.2 was used for all experiments reported in this section, and therefore PER and the n-step update were enabled. We altered only the number of actors used.

Our system had nine threads available for actors, as the other threads were utilised for other operations such as updating the main network. Accordingly, the maximum number of actors we could run in parallel was nine. We performed an experiment using nine actors to complete the task specified in Section 9.1.1. This configuration resulted in a replay ratio of 0.017. We then performed an experiment that only utilised one actor, but we manually set the replay ratio to be similar to the first configuration, i.e. 0.017. We did not test other configurations, since these tests are very time consuming. In Figure 9.6 we present the mean episode return as a function of the completed network update steps. It shows that the different configurations' performance is virtually identical over completed network update steps. This result shows that performance is identical when the replay ratios are the same. Performance is therefore not dependent on the number of actors, but rather on the replay ratio.

In Figure 9.7 we present the mean episode return as a function of time. It shows that it took around five times longer to complete the experiment that utilised only one actor compared to the experiment that used nine actors. We increased the number of actors

Stellenbosch University https://scholar.sun.ac.za

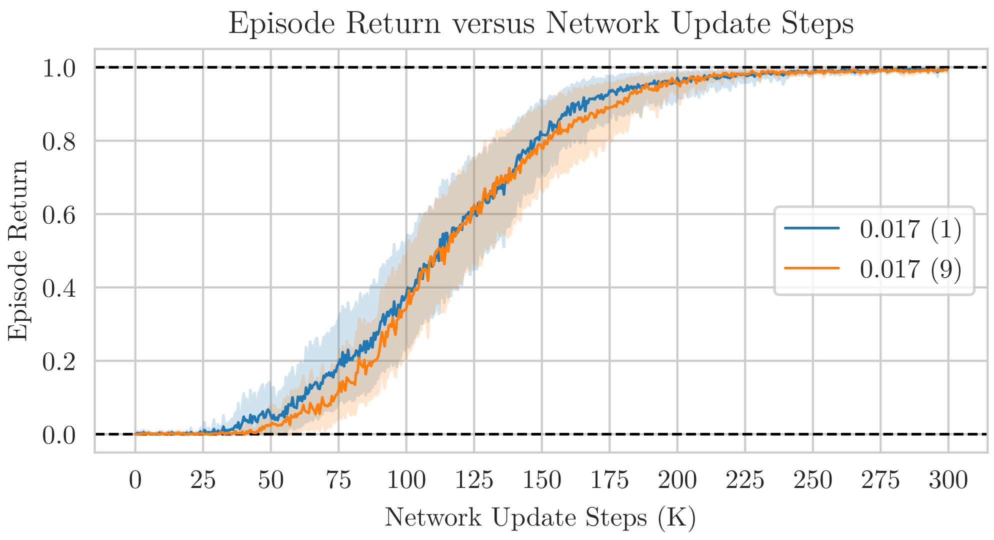

Figure 9.6: The mean episode return as a function of the completed network update steps. Utilising one actor is compared to utilising nine actors. Both configurations achieve a replay ratio of 0.017.

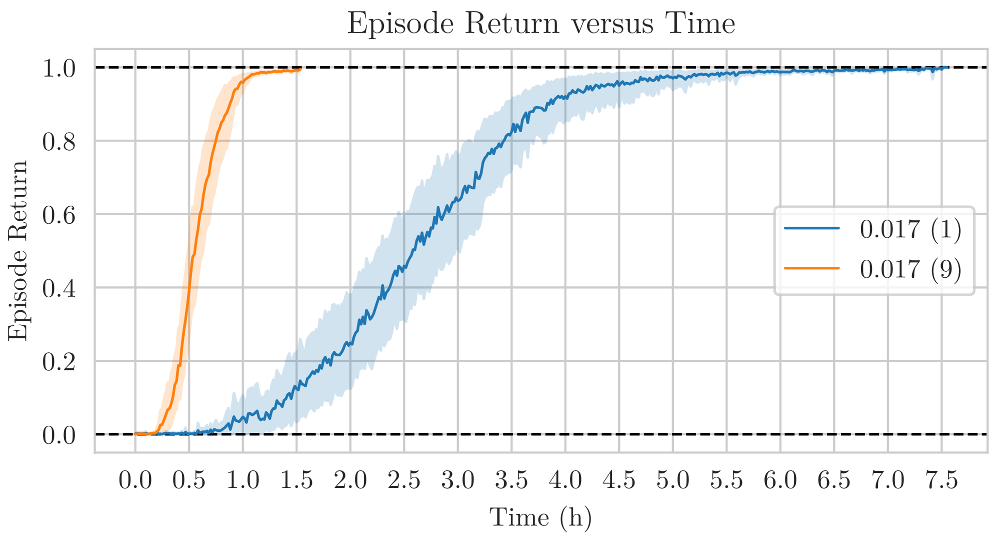

Figure 9.7: The mean episode return as a function of time utilising one actor compared to utilising nine actors with a replay ratio of 0.017.

from one to nine, but the experiment duration only decreased five times. The reason for this is not clear, but there may be some inefficiencies in the system. For example, all actors and the learner access the same replay buffer and this may lead to a queue at the buffer. Using more actors increases the strain on system resources such as the GPU. Therefore the speed of all processes may slow down as we increase the number of actors. Although the duration of the experiment did not decrease by the same ratio as we increased the actors, this result shows that using multiple actors lowers the replay ratio while keeping the duration of experiments short.

Next, we describe the effect of the number of actors used on the replay ratio. Recall

Stellenbosch University https://scholar.sun.ac.za

that all the components of the system ran asynchronously. Accordingly by using more actors in parallel, we increased the data generation rate, thus lowering the replay ratio. We decided to test the system using the maximum and the minimum number of actors, i.e. nine actors and one actor. We also tested configurations with three, five and seven actors.

In Figure 9.8 we present the transition rate relative to the completed network update steps when using different numbers of actors. It shows that on increasing the number of

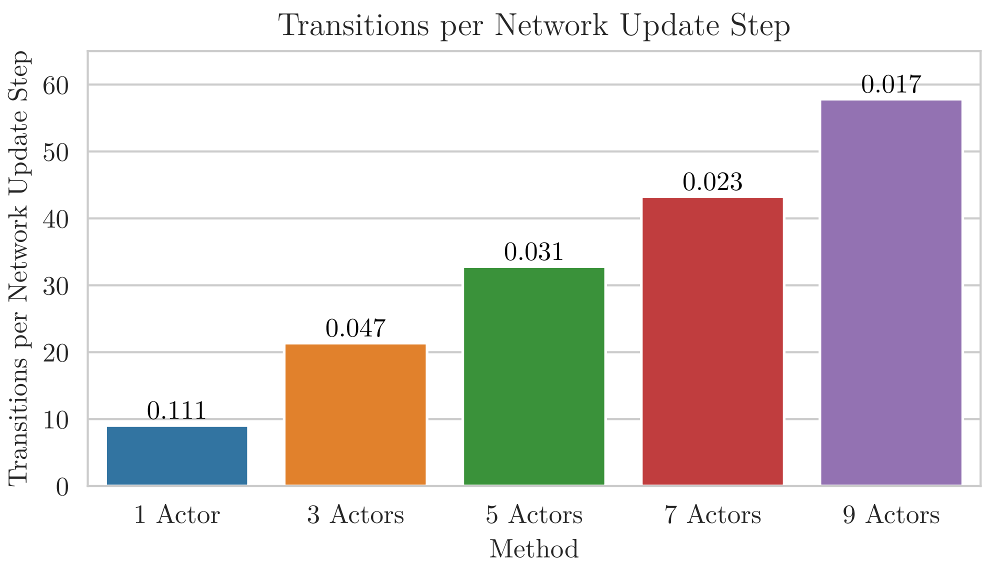

Figure 9.8: The number of transitions generated per network update step while using different numbers of actors. At the top of each bar, the replay ratio of each configuration is shown.

utilised actors, the data generation rate relative to network updates increased linearly. Accordingly, the replay ratio in Equation 6.1 decreased.

Next, we look at whether it was beneficial to lower the replay ratio. Below, we can observe how well the previously achieved replay ratios performed. In Figure 9.9 we present the mean episode return as a function of the completed network update steps using different replay ratios. It shows that the learner trains a better performing policy with fewer completed network update steps when the replay ratio is lower.

We also present the rate at which the learner completed the network update steps when altering the number of actors (Figure 9.10). It shows that on increasing the number of actors, the rate at which the learner updates the network decreased linearly.

Therefore, we also considered the performance of the different configurations as a function of time, as shown in Figure 9.11. Decreasing the replay ratio is computationally expensive because more actors are active on the system. A slower network update rate may result in inferior performance when measured over time. The result suggests that the optimal number of actors for this specific task on this particular hardware is seven. Increasing the number of actors to nine slowed down the network update rate, and the overall performance versus time is then worse than when using seven actors. We concluded

Stellenbosch University https://scholar.sun.ac.za

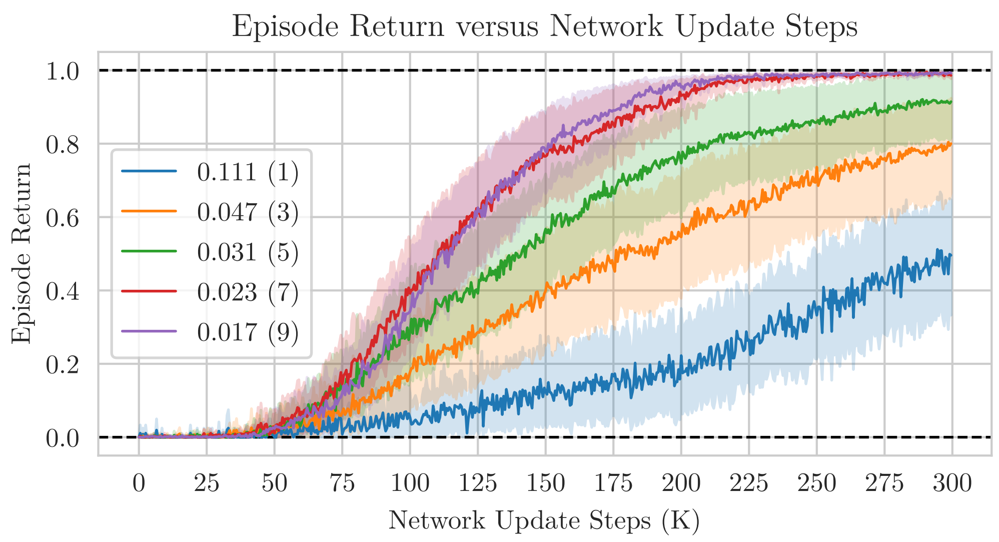

Figure 9.9: The mean episode return versus network update steps for different replay ratios. The replay ratio with the number of actors used is indicated for each curve – replay ratio (#actors). The maximum and minimum achievable returns are indicated by the dashed lines respectively at 1.0 and 0.0.

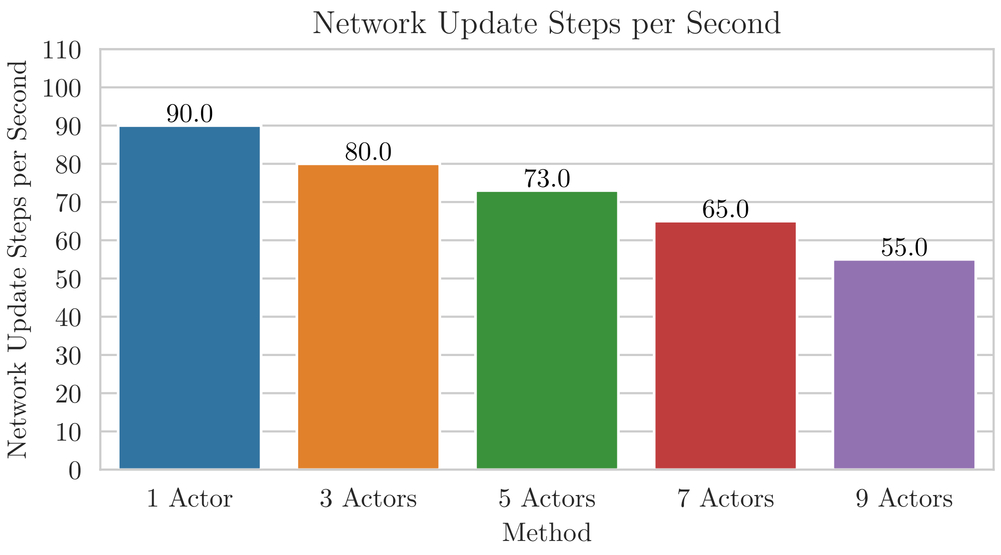

Figure 9.10: The number of network update steps completed per second for different numbers of actors.

that lower replay ratios are beneficial to the agent's performance. Having multiple actors generating data in parallel was found to be an effective way to lower the replay ratio while keeping the duration of experiments short.

9.1.4 Prioritising experience

In this section, we show the impact PER has on performance when solving the sparse-reward problem described in Section 9.1.1. In this experiment, we tested several configurations of

Stellenbosch University https://scholar.sun.ac.za

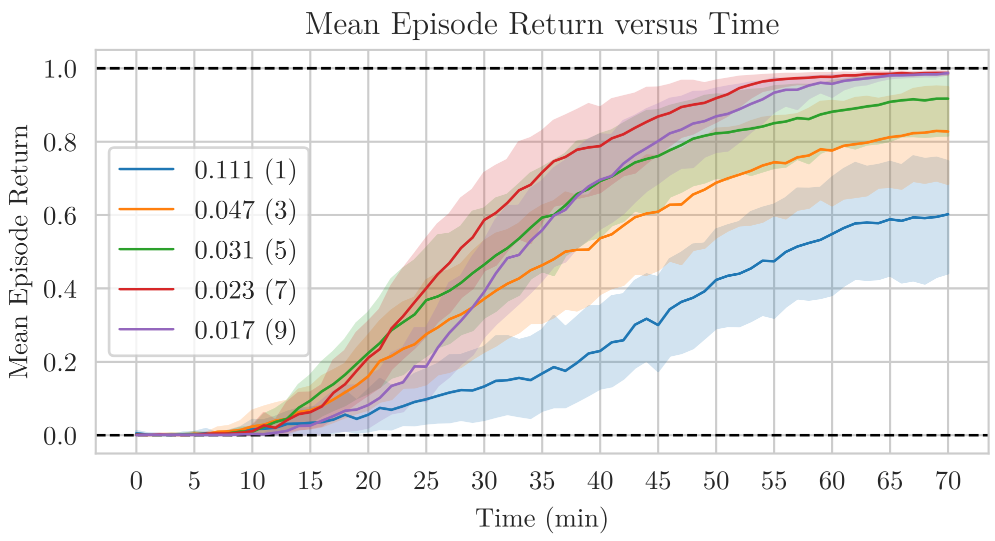

Figure 9.11: The mean episode return versus time for different numbers of actors. The replay ratio with the number of actors used is indicated for each curve – replay ratio (#actors). The maximum and minimum achievable returns are indicated by the dashed lines respectively and 1.0 and 0.0.

the agent. We started with standard ER ($\zeta = 0$) where the learner sampled uniformly from the replay buffer. Next, we tested PER where we increased ζ with 0.2 in each following test. We tested the following values for ζ: $\zeta = 0.2, 0.4, 0.6, 0.8, 1.0$. When PER is enabled, the learner samples according to the priorities assigned to the transitions in the replay buffer, as discussed in Section 6.2. By increasing ζ, there is a stronger focus on the more important transitions. The default configuration in Table 8.2 was used again for this experiment, i.e. seven actors and an n-step return of six were used. The replay ratio was the same for all the configurations of the agent.

We first investigated the influence PER had on the ability to sample transitions containing non-zero rewards. In Figure 9.12 we present the mean reward sampled for the different agents' early training stages. We assume that the agents' replay buffers' contents were comparable at the start of training. The reason for this assumption is that we initialised all agents' networks similarly, and the agents followed the same ε-greedy exploration strategy to generate transitions. Figure 9.12 shows that as we increased ζ, a higher mean reward was sampled. On the other hand, standard experience replay rarely sampled the non-zero reward. Although transitions containing non-zero rewards were not the only transitions that were important to sample, one can easily argue that it is very important to sample these transitions since the entire policy is deduced from the reward.

Next, we compare the performance of an agent using standard ER to agents using PER with the different prioritisation values (ζ). In Figure 9.13 we present the mean episode return as a function of the completed network update steps. It shows that by using higher values for ζ better performing policies were trained. At the end of the experiment, the agents using PER with higher values for ζ obtained better performing policies. The agent

Stellenbosch University https://scholar.sun.ac.za

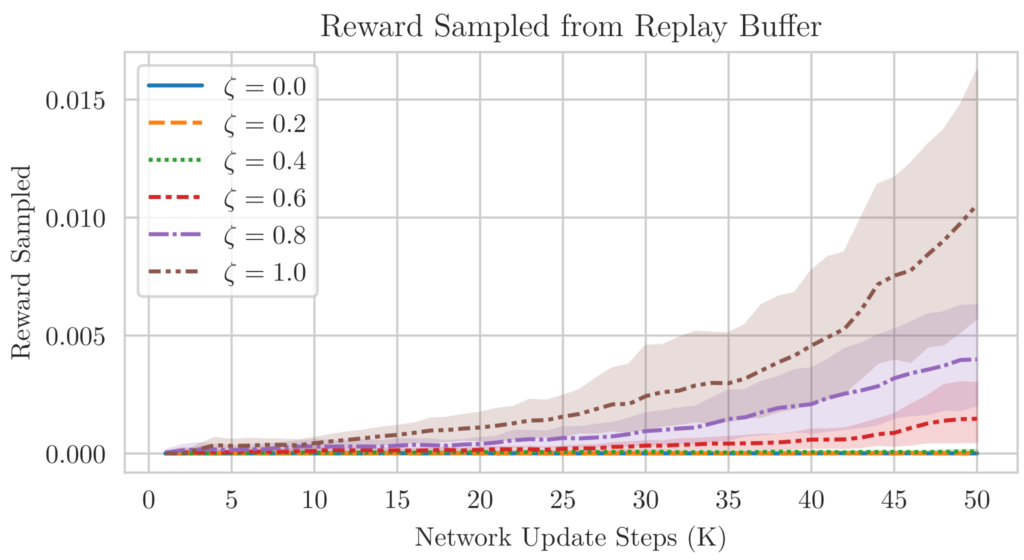

Figure 9.12: The mean reward sampled from the replay buffer for different prioritisation (ζ) values.

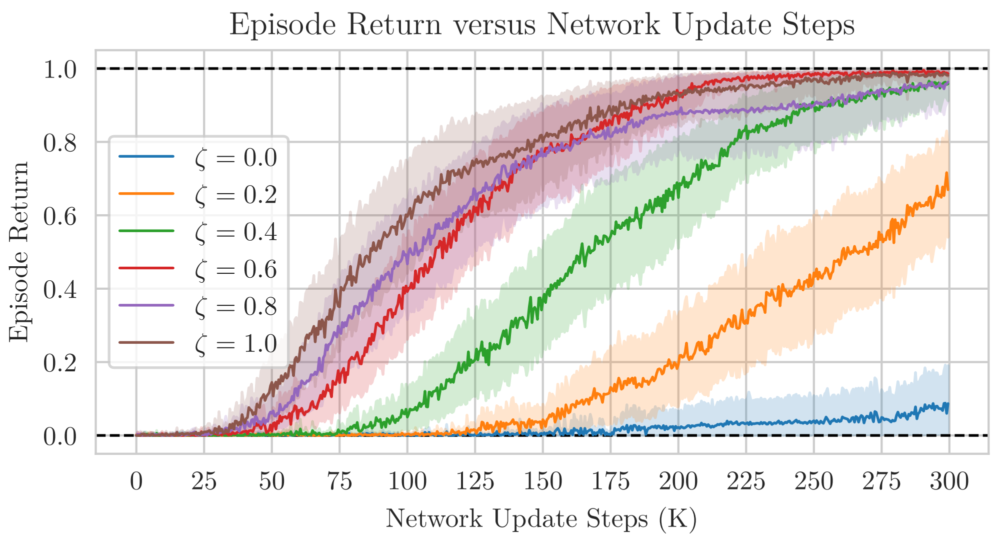

Figure 9.13: The mean episode return as a function of the completed network update steps for different prioritisation values (ζ) is shown.

with $\zeta = 0.6$ received a near-perfect mean episode return of 1.0, whereas the agent using standard ER ($\zeta = 0$) received a mean episode return of around 0.1. The execution times of these algorithms were very similar, as shown in Figure 9.4. The agents using PER utilised critical transitions more effectively than the agent using standard ER, leading to more sample-efficient agents. This result shows that when dealing with sparse rewards, PER is very important to include in the deep Q-learning algorithm.

Stellenbosch University https://scholar.sun.ac.za

9.1.5 n-step returns

We applied n-step return and tested the agent's performance for different values for n. We started with $n = 1$ and incremented the value of n until the agent's performance no longer improved. The goal was to find an optimal value for n. We tested the following values of n: $n = 1, 2, 3, 4, 5, 6, 7$. We used the default configuration in Table 8.2 to perform the experiment. Therefore seven actors were used and PER was enabled. We used a constant replay ratio for all experiments to ensure that the results were comparable to the completed network update steps.

In Figure 9.14 we present the mean episode return for different values of n as a function of the completed network updates. We also present the mean episode return as a function

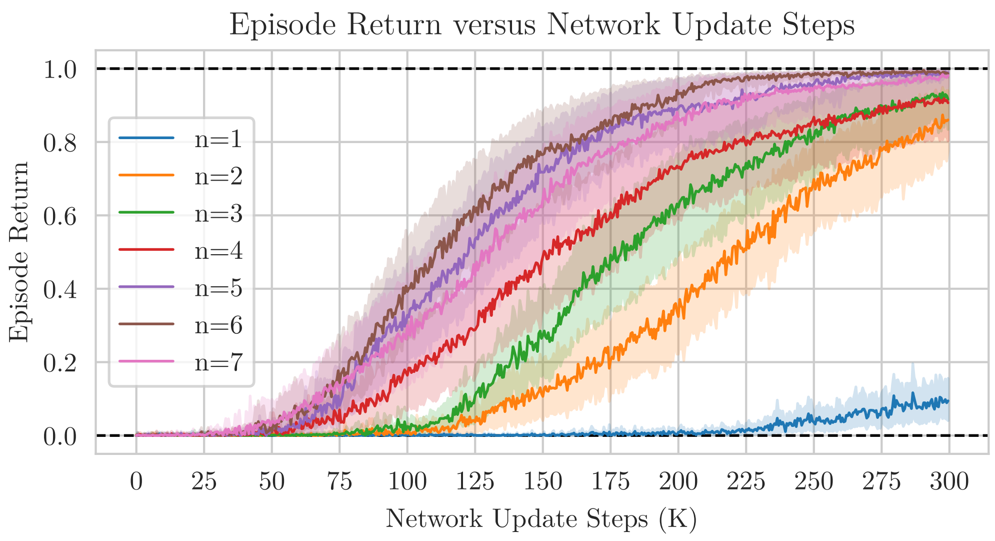

Figure 9.14: The mean episode return versus completed network update steps for different values of n. The maximum and minimum achievable returns are indicated by the dashed lines respectively at 1.0 and 0.0.

of time in Figure 9.15.

The results show that the one-step return ($n = 1$) performed very poorly and only started to improve after 225k network update steps. It achieved a mean return of around 0.1 at the end of the experiment. On increasing n to two, the agent's performance drastically improved. This trend continued as we further increased the value of n. Sutton and Barto [45] state there is an optimal point between one-step TD methods and MC methods. We found the optimal value for n to be six, since for this value the mean return was the highest for all network update steps. Figure 9.14 shows that if we increase n from six to seven, the agent's performance starts to decrease. The difference in performance between $n = 6$ and $n = 7$ is slightly larger in Figure 9.15 than in Figure 9.14. The reason for this is that it is more expensive to use higher values of n. Recall that the n-step return consists

Stellenbosch University https://scholar.sun.ac.za

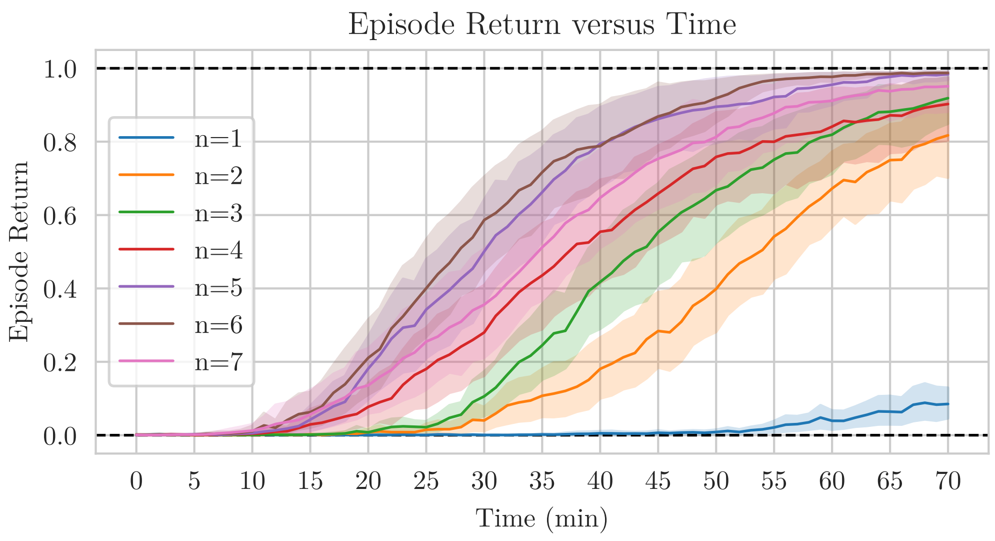

Figure 9.15: The mean episode return versus time for different values of n. The maximum and minimum achievable returns are indicated by the dashed lines respectively at 1.0 and 0.0.

of a sum of n terms. The sum of returns must be computed for each transition. Therefore lower values of n are, more desirable from a computational cost perspective.

Stellenbosch University https://scholar.sun.ac.za

9.2 Experiment 2: Addressing exploration

In this section, we increased the previous problem's difficulty so that the non-zero reward would be even more scarce. Therefore exploration was very challenging in this task. Here, we investigate *curriculum learning* (CL) and *domain randomisation* (DR) to address the more difficult exploration problem. These methods also use the ε-greedy exploration strategy, but we initialised the environment differently. We first describe the new task used as a test-bed for the experiments of this section. We then describe the different methods used to train the agent. Lastly, we present and discuss the result of this experiment.

9.2.1 Description of the task

We modified the task described in Section 9.1.1 and the new task is illustrated in Figure 9.16. We added an extra room and obstacle to the original problem. The mine now

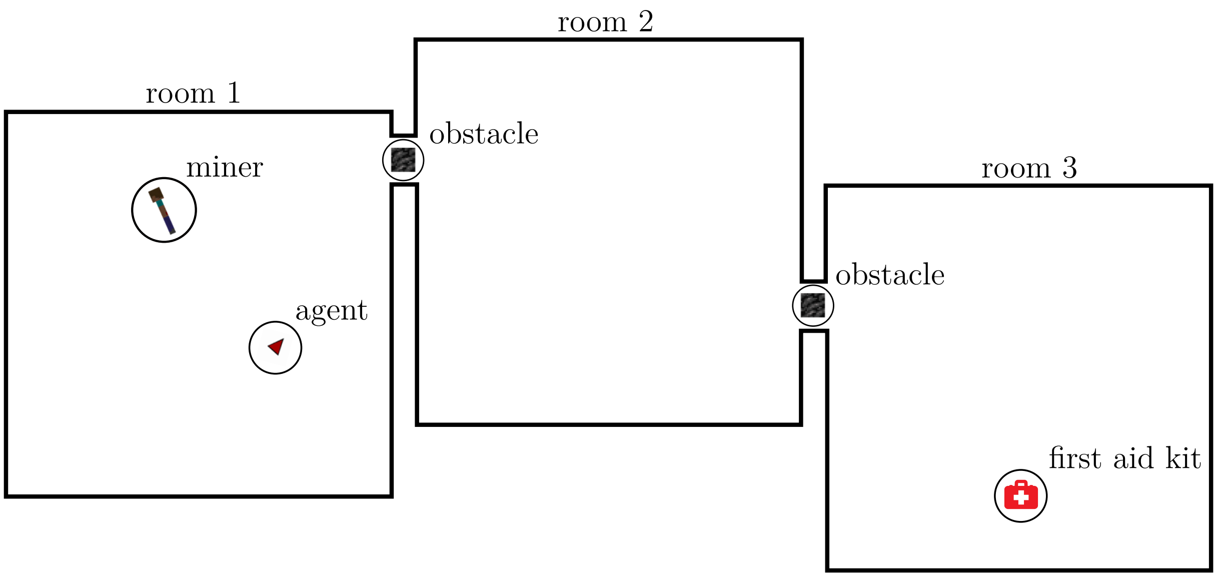

Figure 9.16: Problem two: A top-down illustration of the more difficult sparse-reward problem. The agent and miner are randomly placed in room 1. Obstacles obstruct the way between rooms. The first-aid kit is placed in room 3, the farthest room from which the agent is placed.

consisted of three rooms. Doorways connected the different rooms to allow navigation between rooms. Each doorway had an obstacle obstructing the way between the rooms. In this experiment, the agent and injured miner were placed at random locations inside the first room, and the first-aid kit was placed at a random location inside the third room. To complete this task, the agent had to make its way from the first room to the third room and clear the path by picking up the obstacles and moving them out of the way. Once the agent was in the third room, it had to locate and pick up the first-aid kit. The agent then had to deliver the first-aid kit to the miner in the first room.

The additional room, obstacle and the fact that the agent now had to return to the first room made the reward more challenging to obtain than before. The agent had access

Stellenbosch University https://scholar.sun.ac.za

to the action space shown in Table 8.3. Again the environment would return a reward of 1.0 when the miner receives the first-aid kit, and for all other time steps, a reward of 0.0 would be returned. In each episode, the agent could take a maximum of 2000 steps to complete the task, after which the episode would end.

9.2.2 Curriculum learning implementation

Curriculum learning (CL) is an approach that can be used to learn to solve more difficult tasks, as discussed in Section 2.5.2. CL entails defining a related but simpler problem that would be easy for the RL agent to solve. The difficulty of the problem was then incrementally increased until the agent was capable of solving the complete problem.

We addressed this problem following a CL approach, with the goal of guiding the agent to an optimal policy by presenting it with a series of simpler problems. We had to consider what the definition of a simpler version of the problem would be and at what point the agent would switch to a more difficult problem. This made CL difficult to apply in RL, and usually, a process of trial and error is followed. We decided that it would be sensible to divide the main task into six problems we call *phases* and to present the phases to the agent in reverse order. The different phases are defined as follows (refer to Figure 9.16):

- Phase 1: The complete problem as defined in Section 9.2.1.
- Phase 2: The agent is placed in room two, and the obstacle between room one and room two is moved out of the way.
- Phase 3: The agent is placed in room three, and both obstacles are moved out of the way.
- Phase 4: The agent is placed in room three, carrying the first-aid kit. Both obstacles are moved out of the way.
- Phase 5: The agent is placed in room two, carrying the first-aid kit. Both obstacles are moved out of the way.
- Phase 6: The agent is again placed in room one (same room as the miner), but the agent is now carrying the first-aid kit. Both obstacles are moved out of the way.

This method ensured that we would decrease the simulation environment's phase every time the rolling mean episode return of the last 50 episodes completed by the actors was greater than 0.6. After the phase had been decreased, the rolling mean was set to zero. This process continued until the first phase was reached.

Stellenbosch University https://scholar.sun.ac.za

9.2.3 Domain randomisation implementation

We also investigated a second method referred to as *domain randomisation* (DR), as discussed in Section 2.5.3. Our implementation of DR entailed using an alternative method to initialise the environment to help the agent to improve exploration of the environment. Instead of the starting state described in Section 9.2.1, we sampled the environment's starting state from a larger set of starting states. With this method, we hoped to improve exploration as well as the agent's ability to acquire non-zero rewards.

Instead of training the agent on the problem task described in Section 9.2.1, we trained the agent on a similar task but altered the locations where entities were placed in the environment. The different entities of the environment (agent, first-aid kit, and injured miner) were now randomly placed in any of the rooms. Therefore, placement was not constrained to specified rooms like in the description in Section 9.2.1. Although this changed the task's description, the new alternative definition of the task still contained the original task. With this technique, the agent first learns to solve simpler versions of the problem and then later learns to solve the complete problem.

9.2.4 Experimental results

We will now discuss three methods used to train our agent to solve the problem described in Section 9.2.1. The first we refer to as the *baseline*, where the actors rely on ε-greedy exploration to explore the environment of the problem described in Section 9.2.1. The second we utilised was the CL approach described in Section 9.2.2. With the CL approach, the agent is first trained on a more straightforward problem and then incrementally introduced to the complete task. The third method is DR – where we alter the initialisation of the environment to improve exploration.

Since we trained each agent with a different method, the learning curves are not directly comparable. We wanted to observe how well each configuration learned to solve the problem where the reward is sparse. We made use of an evaluation actor. The purpose of the evaluation actor was to evaluate an agent on the problem of interest while using a different problem description to train the agent, in our case CL or DR. This allows for comparing the different approaches solving the problem of interest where the reward is sparse. This agent was initialised alongside the other actors and evaluated the main network for ten episodes every $10,000$ steps on the complete problem during training. Therefore, we could use the evaluation actors' results to compare the performance of the different methods we investigated. Each agent was trained for 10^6 network update steps. Since the duration of testing the configurations were very long, each test was repeated 20 times.

In Figure 9.17 we show how the different approaches compare in terms of mean episode return as a function of the completed network update steps. Figure 9.18 shows the mean

Stellenbosch University https://scholar.sun.ac.za

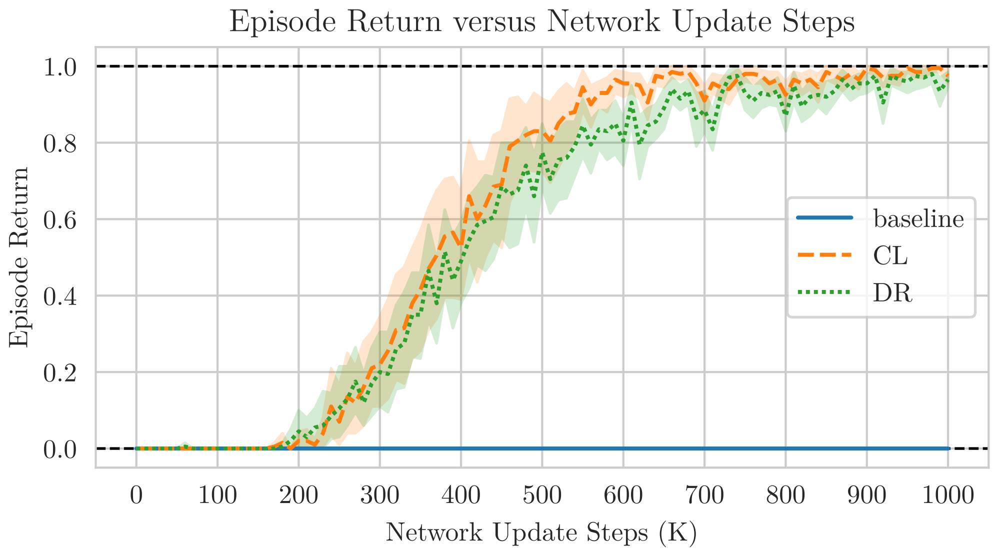

Figure 9.17: The *baseline* plot refers to the agent trained on the complete problem from the start. It shows the mean episode return as a function of the completed network update steps for the baseline configuration, CL and DR.

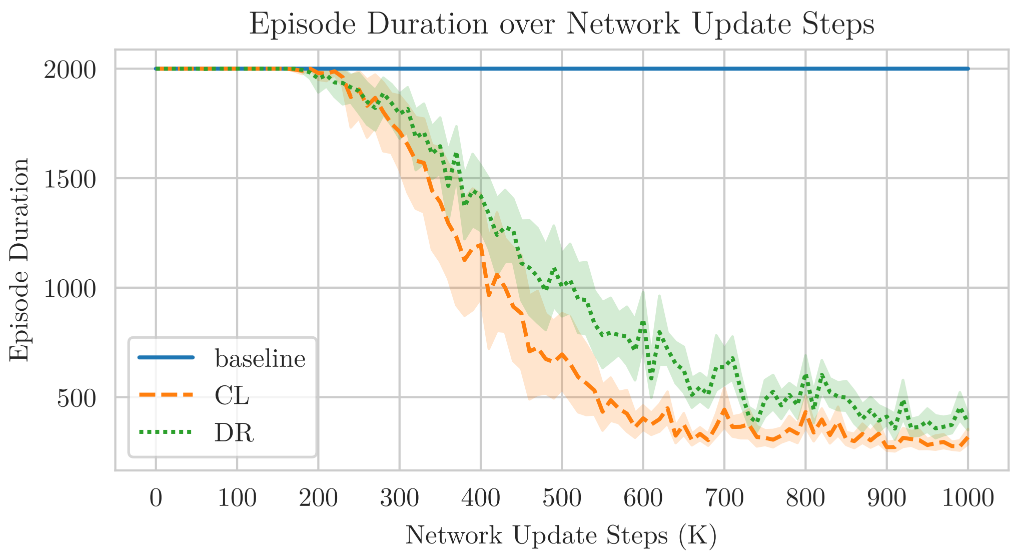

Figure 9.18: The *baseline* plot refers to the agent trained on the complete problem from the start. It shows the mean episode duration as a function of the completed network update steps for the baseline configuration, CL and DR.

episode duration versus network update steps of the different approaches. The baseline result shows that when the agent faces the complete problem from the start, an ε-greedy exploration strategy is no longer sufficient to obtain non-zero rewards. Therefore the agent was not able to learn an optimal policy to complete the task. We, therefore, concluded from the baseline that obtaining non-zero rewards in this task is very challenging.

The two alternative approaches, CL and DR worked similarly well, and these agents were able to learn policies capable of reliably solving the complete task. One should note that when using DR (entities are placed in any of the rooms), the agent learns to solve

Stellenbosch University https://scholar.sun.ac.za

more variations of the problem compared to CL. Figure 9.18 shows that CL was able to complete the task with a shorter duration than DR. DR describes a more general and challenging version of the problem, and therefore performance was a bit poorer on the target task.

We have shown that difficult problems where rewards are very sparse can be challenging for our agent to learn to solve from scratch. The results show that incrementally training the agent by exposing it to both simple versions and complex versions of the problem allow the agent to learn to solve more complicated tasks. We conclude that initialisation of the environment is important when dealing with very sparse rewards and that CL and DR are effective methods to enable the agent to learn to solve the more difficult sparse-reward problem.

9.3 Experiment 3: Ability to scale to larger Environments

In the previous section, we found that CL and DR are effective methods to address challenging exploration problems where rewards are very sparse. DR also describes the more general task where entities of the environment can occur in any of the rooms. In this section, we continue to use DR and illustrate how well this algorithm scales to larger environments by increasing the number of rooms.

9.3.1 Description of task

The description of the problem used to perform this section's experiments is primarily the same as the previous section's task. The main difference is that entities were now always placed in random rooms. The doorways between rooms were all obstructed by obstacles. We aimed to observe how well the algorithm scales to larger environments; therefore, the number of rooms in the environment could be specified. In each episode, a maximum of 2000 steps was available to complete the task, after which the episode would end.

9.3.2 Scaling domain randomisation

We performed three experiments in this section. The environment consisted of a different number of rooms in each experiment. We tested the agent's performance on an environment with three rooms, four rooms and five rooms, to observe whether the agent could learn to solve the task in larger environments. For each experiment, we trained the agent for 10×10^6 network update steps.

In Figure 9.19 we present the mean episode return as a function of the completed network updates. It shows that the agent performed relatively well in the three-room and

Stellenbosch University https://scholar.sun.ac.za

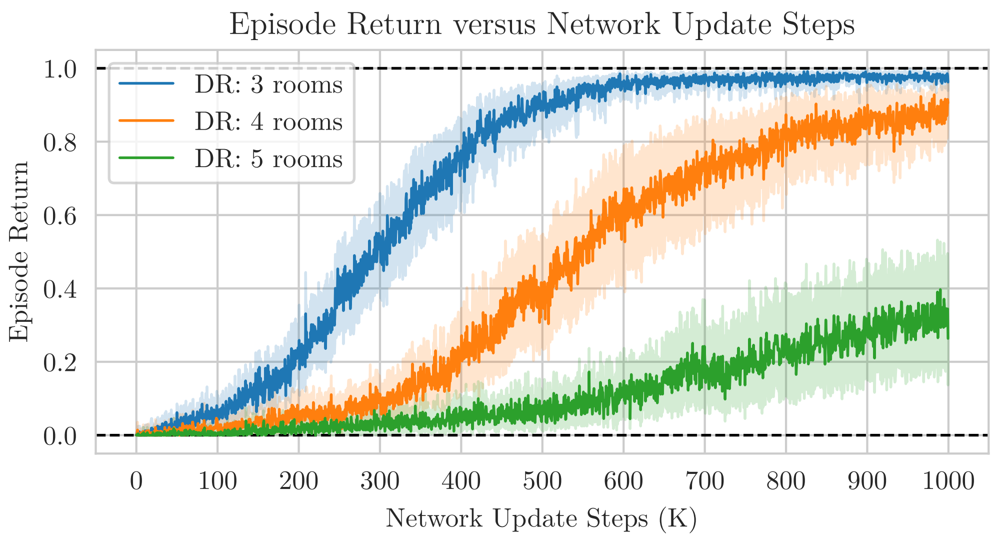

Figure 9.19: The mean episode return versus the completed network update steps for environments consisting of a different number of rooms. In the first experiment we train the agent on an environment with three rooms. With each following experiment, the room count is increased by one, until performance starts to decrease drastically. The maximum and minimum achievable returns are indicated by the dashed lines respectively at 1.0 and 0.0.

four-room problems, but that performance drastically decreased when we tested the agent on a problem consisting of five rooms.

9.3.3 Combining domain randomisation and curriculum learning

For these experiments we used 10×10^6 network update steps to test each configuration, which was already very expensive to perform. It becomes infeasible to train the agent for even more network updates to observe how well it performs when it converges to an optimal policy. We decided to focus on improving the five-room agent in the given number of network updates. The three-room and the four-room problems were not that different from the five-room problem, and we therefore decided to use a CL approach to try to improve the performance of the five-room agent. We divided the allocated network updates into three equal parts. We first trained the agent on the three-room problem. After the agent completed a third of the network update steps, we switched to the four-room problem. Two thirds into training, we switched the agent to the five-room problem. Figure 9.20 shows how this procedure compares to training from the start in the five-room problem. The curves in Figure 9.20 are comparable from 667 thousand network update steps onward, as all the agents are then facing the same problem. This result again shows that when dealing with challenging tasks where rewards are very sparse, it is beneficial to train an agent using a CL approach. We also considered performance as a function of time, as shown in Figure 9.21. This method allowed us to obtain agents with good performance in less time.

Stellenbosch University https://scholar.sun.ac.za

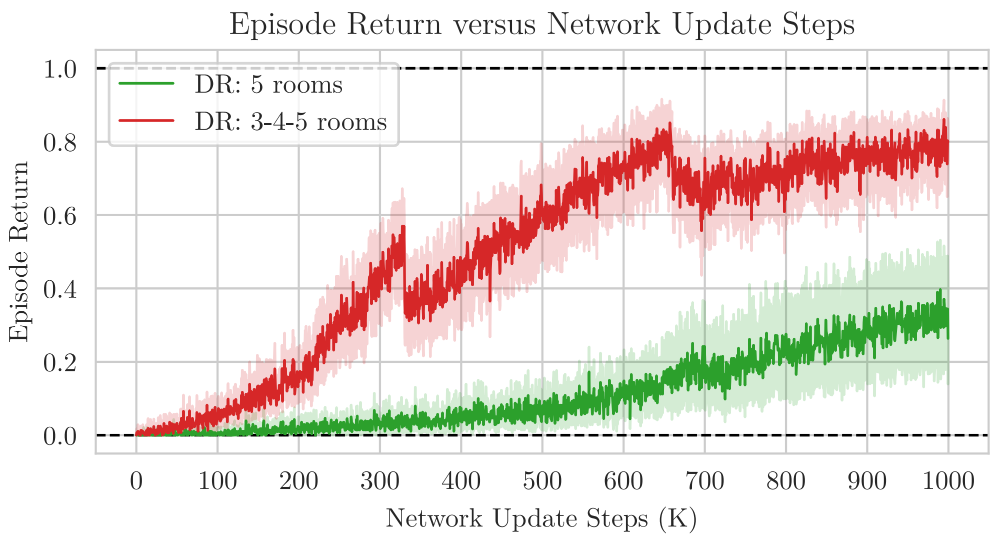

Figure 9.20: A CL (red) approach is used to train the agent. The agent is first trained on the three-room problem. The agent is then switched to the four-room problem a third into training. The last third of training the agent is trained on the five-room problem.

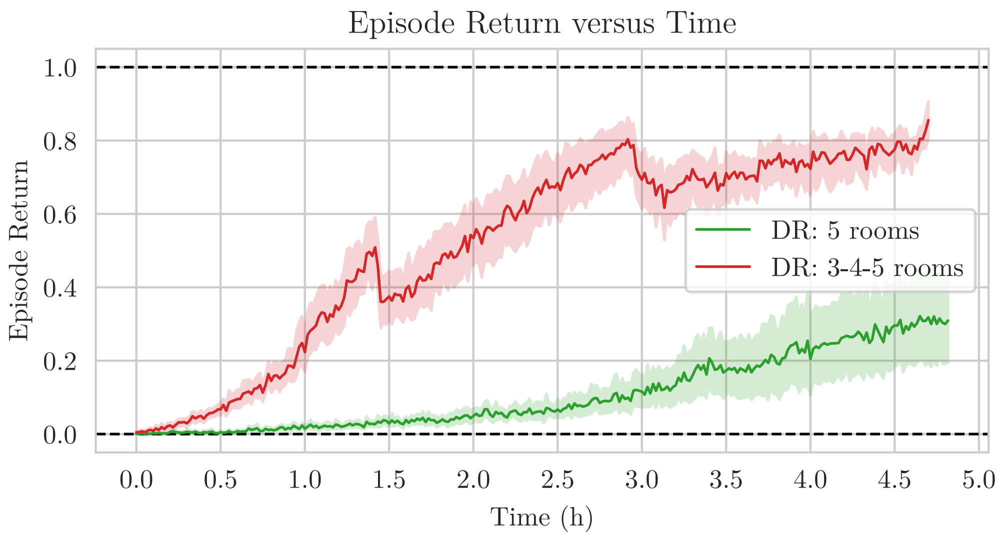

Figure 9.21: The mean episode return as a function of time. A CL (red) approach is used to train the agent. The agent is first trained on the three-room problem. The agent is then switched to the four-room problem a third of the way into training. In the last third of the training, the agent is trained on the five-room problem.

9.4 Summary

In this chapter, we reported on three experiments to test the performance of a deep Q-learning agent on sparse-reward problems. In the first experiment, we tested three modifications made to the original deep Q-learning algorithm to enable it to solve a sparse-reward problem. We showed the importance of each in an ablation study and concluded that these modifications are all crucial to solving the sparse-reward problem.

Stellenbosch University https://scholar.sun.ac.za

In Section 9.1.3, we investigated the effect that the rate at which data is generated has on performance. From the results obtained from this section, we conclude that it is advantageous to the learner if new transitions are generated at a high rate. Therefore, better policies are trained when the learner samples from a replay buffer that contains more on-policy data.

Next, we investigated the importance of prioritised sampling. We compared PER with the standard ER. We showed that non-zero rewards are sampled more frequently when PER is used than when ER is used. The results of the experiments carried out show that PER performed significantly better than the agent using standard ER. For this reason, we conclude that PER is very important when dealing with sparse rewards. We also think that PER works well when transitions are generated at a high rate. Actors generate large amounts of data and, with PER, the learner samples only the most important transitions.

In Section 9.1.5 we tested n-step TD methods. We showed that the n-step update significantly improved the performance of the deep Q-learning agent. We started with the one-step update and incremented n to find an optimal value for n. The agent's performance was optimal for $n = 6$ on the problem we tested.

Next, we addressed exploration in a more difficult sparse-reward problem. We addressed exploration by altering how we initialised the environment. We investigated whether CL and DR help to solve the more difficult problem. We showed that both methods are effective in solving the more difficult task.

Lastly, we tested how well the DR agent scales to larger environments, in Section 9.3. The results of this section showed that the performance of the agent quickly decreases as the size of the environment is increased. We then combined DR with CL and showed that the agent's performance improves when learning incrementally.

Stellenbosch University https://scholar.sun.ac.za

Chapter 10

Conclusion

This study aimed to examine how to train artificially intelligent agents to perform tasks in a 3D environment with sparse rewards. The example task we have addressed is a scenario where a mine has collapsed, and a wounded, immobile miner needs access to a first-aid kit. A problem central to this study was that of sparse rewards – problems where rewards are scarce and difficult to obtain. Enabling RL algorithms to solve sparse-reward problems allows us to specify more long-term and abstract objectives to artificially intelligent agents.

We trained a *double deep Q-network* (DDQN) algorithm without any modifications on the first problem we introduced, but it was unable to acquire a policy that was able to solve the problem. We showed that there are three important modifications to include to solve this problem. The modifications are distributed data generation, *prioritised experience replay* (PER) and the n-step update. We used *curriculum learning* (CL) and *domain randomisation* (DR) to solve a more difficult sparse-reward problem where exploration is challenging. We also showed that CL and DR can be used in combination to solve larger and more complex problems.

10.1 Summary and contributions

In this study we addressed a problem in a 3D environment with observations from a first-person perspective. The environment is only partially observable because some features of the environment's state may be behind the agent or around a corner. Furthermore, by observing a single frame, the agent has no information about the direction in which it was previously moving. We used function approximation in the form of a DNN to interpret the image observations. Sutton and Barto [45] state that function approximation allows us to extend RL to partially observable environments. According to Sutton and Barto [45], if certain aspects of the state are not observable, then the parameters of the function simply do not depend on it. However, function approximation cannot augment the state with additional information such as past observations. For this reason, we augmented observations by using frame-stacking and action memory to help address the

Stellenbosch University https://scholar.sun.ac.za

problem of partial observability. The results of the experiments in Section 8.8.3 show that frame-stacking and action memory both improve the agent's performance.

Successful trajectories were initially very scarce using an ε-greedy strategy to explore the environment, since rewards are sparse. For this reason, we used a distributed system to generate data at a higher rate, hence also generating more successful trajectories. Then by using PER from Schaul *et al.* [39], the learner prioritises the important transitions to train the network. We showed that transitions with non-zero rewards are more frequently sampled when using PER compared to standard ER. We also showed in Section 9.1.4 that PER is very important to achieve a well-performing agent. Furthermore, PER and distributed data generation work well together – large amounts of data are generated, and PER enables the agent to only focus on the important transitions.

An additional effect of generating data at a high frequency relative to updating the agent's network is that the learner completes updates using more recent knowledge, i.e. on-policy data (refer to Equation 6.2). We confirmed the results by Fedus *et al.* [10], thus performance is generally better when a lower replay ratio is used. The same results can be obtained by using a non-distributed system, but we showed in the first experiment of Section 9.1.3 that by generating data in parallel, the wall-time of the experiments can be significantly decreased.

Distributing credit to actions that lead to success is a problem RL tries to solve and is referred to as the credit-assignment problem. Credit assignment is especially challenging when dealing with sparse and delayed rewards. In the definition of the problem we addressed, the agent only receives credit after completing a long sequence of correct actions. For this reason rewards are delayed. The n-step return entails bootstrapping after multiple time steps. It allows for a more significant state change to occur and usually performs better than one-step TD, as discussed in Section 4.3.4. We showed in Section 9.1.5 that the n-step update significantly improves the performance of our agent solving the sparse-reward problem.

Finally, we addressed a more difficult sparse-reward problem. Exploration of the environment was very challenging in this problem. Therefore, the probability of encountering non-zero rewards using random exploration or ε-greedy exploration was very low. We showed that the agent was not able to encounter the reward using ε-greedy exploration. For this reason, no optimal policy was trained. We successfully solved this problem using CL and DR. Both these methods required altering the way the environment is initialised. Therefore an easy-to-modify simulation environment is necessary to implement these methods. Furthermore, domain-specific knowledge is required.

Stellenbosch University https://scholar.sun.ac.za

10.2 Future work

Although we consider this to be a very complicated task to be solved by a software agent, it still does not, in many ways, compare to a real-world problem. An avenue for future research will entail converting the agent from simulation to operating in the real world. For example, the work by Tobin *et al.* [46] can be applied for simulation to real-world conversions. Furthermore, training the agent in a simulation environment with more realistic physics will also allow for more straightforward simulation to real-world conversions.

We suspect that partial observability may still be problematic in solving certain problems where long-term memory is a requirement. A *recurrent neural network* (RNN), for example, a *long short-term memory* (LSTM) module, can be applied to provide an agent with memory. Paine *et al.* [31] used an LSTM to obtain an agent with memory. This is an avenue for future research, since their agent performed poorly on memory-intensive tasks.

Additional possibilities for research will be to investigate other exploration strategies that are not dependent on the above-mentioned requirements, in other words, strategies that are able to obtain non-zero rewards by only training on the sparse-reward problem. Agents with curiosity or intrinsic motivation, for example, the work by Pathak *et al.* [33], is an interesting field to further explore.

We showed that a DRL agent is capable of achieving a relative long-term goal without receiving demonstrations and with minimal feedback in the form of credit in a simplistic environment. Although we are far from general agents that are capable of solving very complex, abstract and long-term tasks, this study highlights some important areas to address and some of the shortcomings of current systems.

Stellenbosch University https://scholar.sun.ac.za

List of References

[1] Alla, H., Kalyan, B. and Murthy, C. (2015 12). Mine rescue robot system - a review. *Procedia Earth and Planetary Science*, vol. 11, pp. 457–462.

[2] Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley. ISBN 9780471453246.
Available at: https://books.google.co.za/books?id=3xfhhRjLUJEC

[3] Bain, M. and Sammut, C. (1995). A framework for behavioural cloning. In: *Machine Intelligence 15*, pp. 103–129.

[4] Bellman, R. (1966). Dynamic programming. *Science*, vol. 153, no. 3731, pp. 34–37.

[5] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. (2016). Openai gym. arXiv:1606.01540.

[6] Chevalier-Boisvert, M. (2018). Gym miniworld environment for openai gym. https://github.com/maximecb/gym-miniworld.

[7] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K. *et al.* (2012). Large scale distributed deep networks. In: *Advances in neural information processing systems*, pp. 1223–1231.

[8] Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K.O. and Clune, J. (2021). First return, then explore. *Nature*, vol. 590, no. 7847, pp. 580–586.

[9] Endo, Y. (2008). Anticipatory robot control for a partially observable environment using episodic memories. In: *2008 IEEE International Conference on Robotics and Automation*, pp. 2852–2859. IEEE.

[10] Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M. and Dabney, W. (2020). Revisiting fundamentals of experience replay. *arXiv preprint arXiv:2007.06700*.

[11] Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep learning*. MIT press.

[12] Hasselt, H.v., Guez, A. and Silver, D. (2016). Deep reinforcement learning with double q-learning. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pp. 2094–2100. AAAI Press.

Stellenbosch University https://scholar.sun.ac.za

[13] Hernandez-Garcia, J.F. and Sutton, R.S. (2019). Understanding multi-step deep reinforcement learning: a systematic study of the dqn target. *arXiv preprint arXiv:1901.07510*.

[14] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In: *Thirty-Second AAAI Conference on Artificial Intelligence*.

[15] Hester, T., Vecerík, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., Dulac-Arnold, G., Agapiou, J., Leibo, J.Z. and Gruslys, A. (2018). Deep q-learning from demonstrations. In: *AAAI*.

[16] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H. and Silver, D. (2018). Distributed prioritized experience replay. In: *International Conference on Learning Representations*.
Available at: https://openreview.net/forum?id=H1Dy---0Z

[17] Hussein, A., Gaber, M.M., Elyan, E. and Jayne, C. (2017). Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35.

[18] Johnson, Hofmann, H. and Bignell (2016). The malmo platform for artificial intelligence experimentation. https://github.com/Microsoft/malmo.

[19] Karpathy, A. *et al.* (2016). Cs231n convolutional neural networks for visual recognition. *Neural networks*, vol. 1, no. 1.

[20] Kingma, D.P. and Ba, J. (2015). Adam: A method for stochastic optimization. In: Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.
Available at: http://arxiv.org/abs/1412.6980

[21] LeCun, Y., Kavukcuoglu, K. and Farabet, C. (2010). Convolutional networks and applications in vision. In: *Proceedings of 2010 IEEE international symposium on circuits and systems*, pp. 253–256. IEEE.

[22] Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, vol. 8, no. 3-4, pp. 293–321.

[23] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[24] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. *et al.* (2015). Human-level control through deep reinforcement learning. *nature*, vol. 518, no. 7540, pp. 529–533.

[25] Monahan, G.E. (1982). State of the art a survey of partially observable markov decision processes: theory, models, and algorithms. *Management science*, vol. 28, no. 1, pp. 1–16.

Stellenbosch University https://scholar.sun.ac.za

[26] Nachum, O., Norouzi, M., Xu, K. and Schuurmans, D. (2017). Bridging the gap between value and policy based reinforcement learning. In: *Advances in Neural Information Processing Systems*, pp. 2775–2785.

[27] Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S. *et al.* (2015). Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*.

[28] Ng, A. and Katanforoosh, K. (2018). Cs229 lecture notes deep learning.

[29] Ng, A.Y., Harada, D. and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In: *ICML*, vol. 99, pp. 278–287.

[30] O'Mahony, N., Campbell, S., Carvalho, A., Harapanahalli, S., Hernandez, G.V., Krpalkova, L., Riordan, D. and Walsh, J. (2019). Deep learning vs. traditional computer vision. In: *Science and Information Conference*, pp. 128–144. Springer.

[31] Paine, T.L., Gulcehre, C., Shahriari, B., Denil, M., Hoffman, M., Soyer, H., Tanburn, R., Kapturowski, S., Rabinowitz, N., Williams, D. *et al.* (2019). Making efficient use of demonstrations to solve hard exploration problems. *arXiv preprint arXiv:1909.01387*.

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

[33] Pathak, D., Agrawal, P., Efros, A.A. and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 16–17.

[34] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, vol. 12, no. 1, pp. 145–151.

[35] Randløv, J. and Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. In: *ICML*, vol. 98, pp. 463–471.

[36] Ruder, S. (2017). An overview of gradient descent optimization algorithms. 1609.04747.

[37] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. 3rd edn. Prentice Hall Press, USA. ISBN 0136042597.

[38] Salimans, T. and Chen, R. (2018). Learning montezuma's revenge from a single demonstration. *arXiv preprint arXiv:1812.03381*.

[39] Schaul, T., Quan, J., Antonoglou, I. and Silver, D. (2016). Prioritized experience replay. *CoRR*, vol. abs/1511.05952.

[40] Selfridge, O.G., Sutton, R.S. and Barto, A.G. (1985). Training and tracking in robotics. In: *IJCAI*, pp. 670–672.

Stellenbosch University https://scholar.sun.ac.za

[41] Silver, D. (2015). Ucl course on rl. https://www.davidsilver.uk/teaching/. (Accessed on 03/17/2020).

[42] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. *et al.* (2017). Mastering the game of go without human knowledge. *nature*, vol. 550, no. 7676, pp. 354–359.

[43] Skinner, B.F. (1938). *The behavior of organisms: An experimental analysis.* Prentice Hall, Englewood Cliffs, New Jersey.

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

[45] Sutton, R.S. and Barto, A.G. (2018). *Reinforcement learning: An introduction.* MIT press.

[46] Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W. and Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30. IEEE.

[47] Tsitsiklis, J.N. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, vol. 42, no. 5, pp. 674–690.

[48] Vitay, J. (). Deep reinforcement learning.

[49] Wang, Q., Ma, Y., Zhao, K. and Tian, Y. (2020 04). A comprehensive survey of loss functions in machine learning. *Annals of Data Science.*

[50] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In: *International conference on machine learning*, pp. 1995–2003.

[51] Yoon, C. (2019 Nov). Understanding and implementing distributed prioritized experience replay (horgan et al., 2018).
Available at: https://towardsdatascience.com/understanding-and-implementing-distributed-prioritized-experience-replay-horgan-et-al-2018-d2c1640e0520

Stellenbosch University https://scholar.sun.ac.za

Appendices

Stellenbosch University https://scholar.sun.ac.za

Appendix A

Links to example videos of resultant agents

Links to videos that demonstrate the behaviour of the agents we have trained during the experimental phase of our research:

- Experiment 1: The agent with all modifications enabled solving problem one.
- Experiment 2: The agent using *domain randomisation* (DR) to solve problem two.
- Experiment 2: The agent using *curriculum learning* (CL) to solve problem two.
- Experiment 3: The agent solving the five room problem by using a combination of DR and CL.
- The same algorithm applied to the game *Snake*.

Stellenbosch University https://scholar.sun.ac.za

Appendix B

Additional results

Here we show some additional results with regard to the chosen configuration shown in Table 8.2. The default configuration for all the hyperparameters except the one tested in each experiment was used to obtain the results below. These experiments were performed on the problem described in Section 8.8.3.

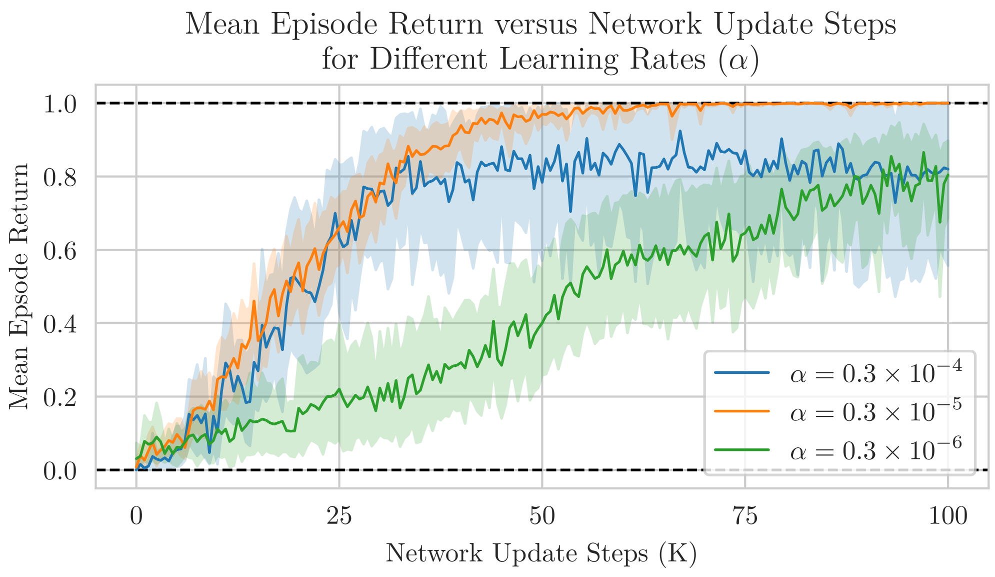

Figure B.1

Stellenbosch University https://scholar.sun.ac.za

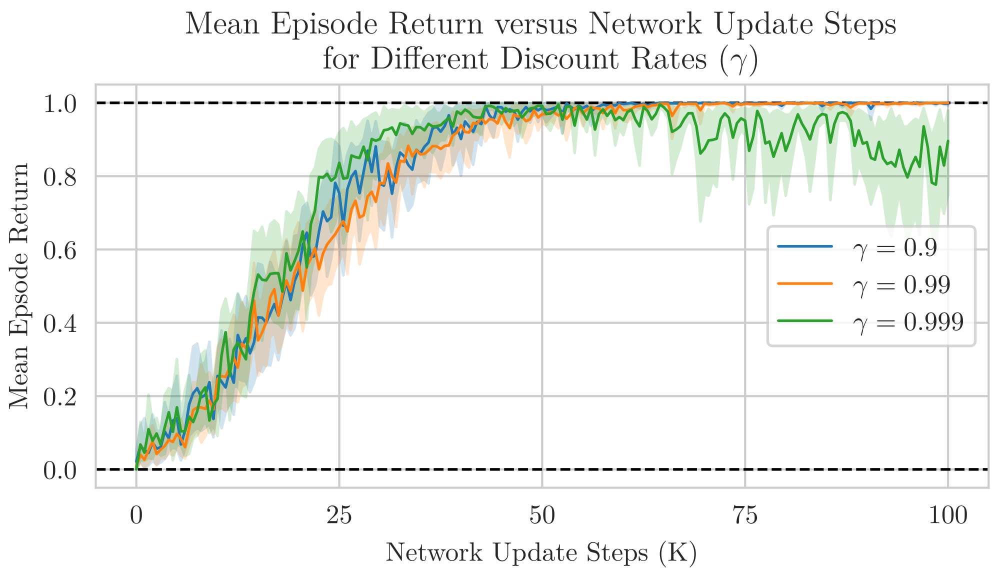

Figure B.2

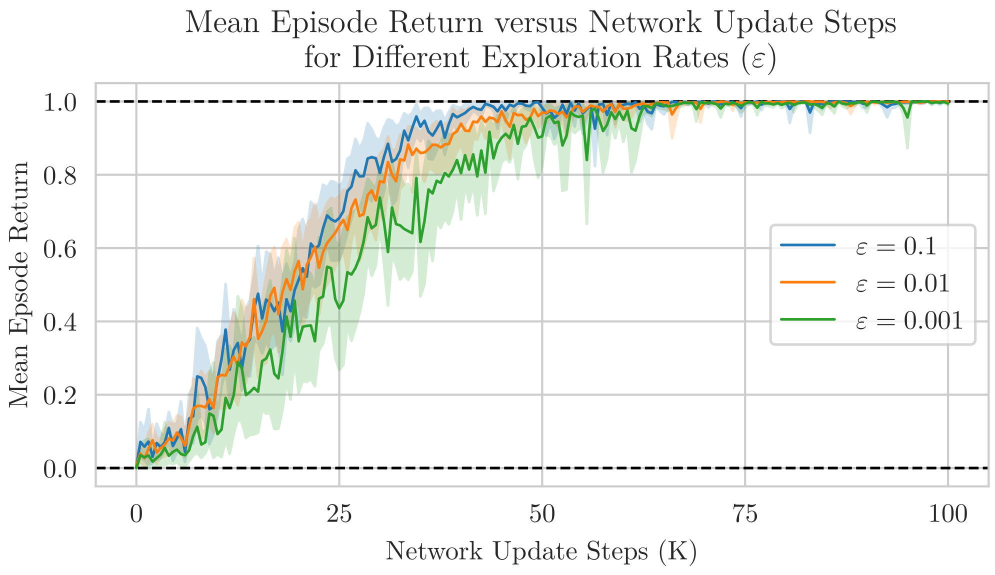

Figure B.3

Stellenbosch University https://scholar.sun.ac.za

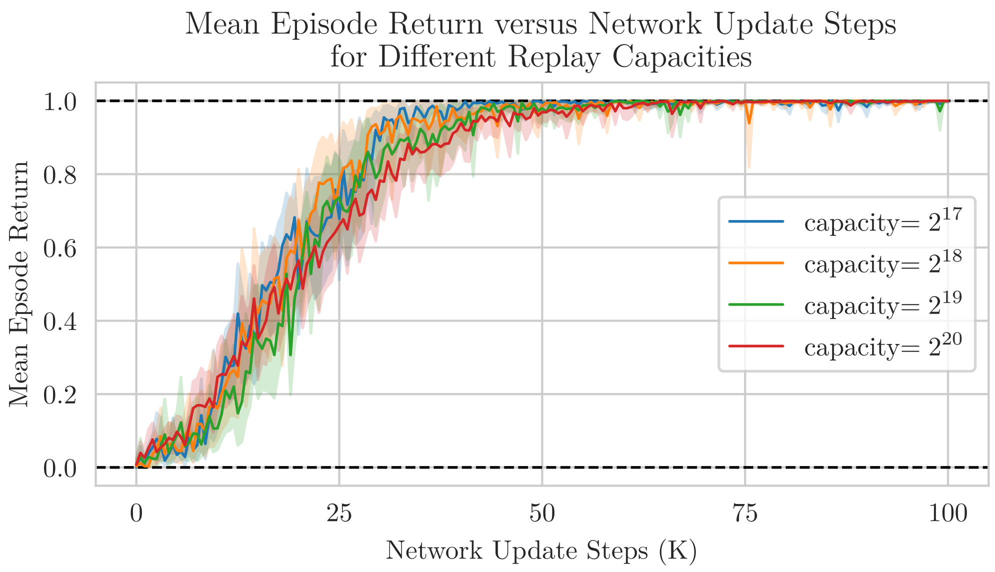

Figure B.4

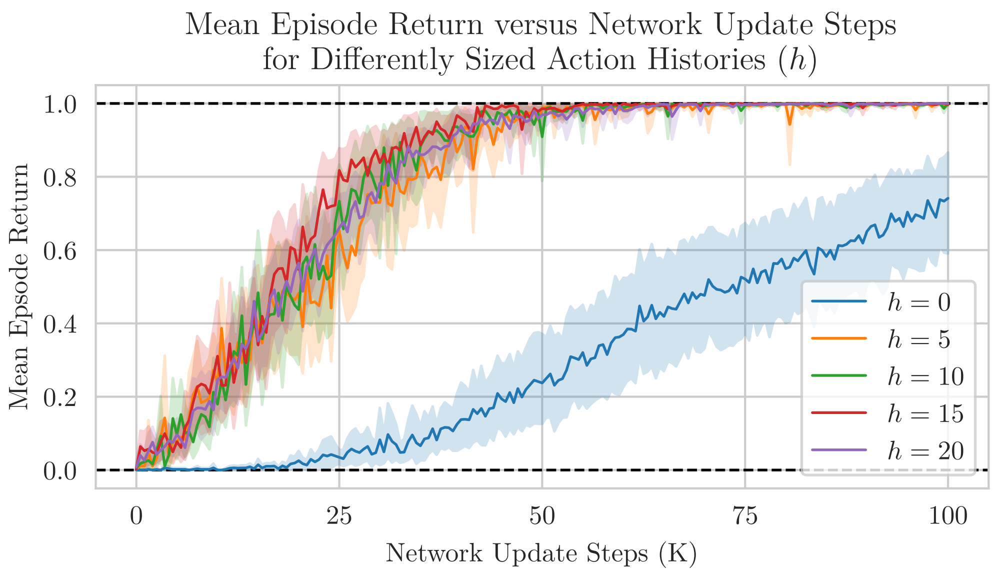

Figure B.5

Stellenbosch University https://scholar.sun.ac.za

Appendix C

Detailed algorithms for the implementation

Below are the detailed algorithms of our distributed deep Q-learning agent.

Algorithm 17: Prioritised replay buffer for storing and sampling transitions according to priorities.

```
def init(capacity, $\zeta$, $\eta$):
    capacity: int ← capacity // capacity of replay buffer
    storage: list // storage
    next_idx: int // index to store next transition
    num_transistions_added: int // counter, tracks the total number
     transitions stored
    $\zeta$: float ← $\zeta$ // prioritisation
    $\eta$: float ← $\eta$ // prioritisation offset
    sum_tree: SumSegmentTree // sum-tree to sample according
     priorities
    min_tree: MinSegmentTree // min-tree to obtain smallest priority
def add_batch(batch, $\boldsymbol{\delta}$):
    $p$ ← get_priorities($\boldsymbol{\delta}$) // calculate priorities from TD errors
    add_batch_to_storage(batch) // add the batch transitions to
     storage
    sum_tree.set_batch($p$) // add priorities to sum-tree
    min_tree.set_batch($p$) // add priorities to min-tree
def sample_batch(batch_size, $\beta$):
    indices ← sample_proportional(batch_size) // sample indices using
    sum-tree
    $\boldsymbol{w}$ ← calculate_weights(indices, $\beta$) // calculate IS weights
    samples ← storage[indices]
    return samples, indices, $\boldsymbol{w}$
def update_priorities(indices, $\boldsymbol{\delta}$):
    $p$ ← get_priorities($\boldsymbol{\delta}$) // calculate priorities from TD errors
    sum_tree.set_batch(indices, $p$) // add priorities to sum-tree
    min_tree.set_batch(indices, $p$) // add priorities to min-tree
```

Stellenbosch University https://scholar.sun.ac.za

Algorithm 18: Parameter server for communication between learner and actors.

```
1  def init():
2      $\boldsymbol{\theta}$: dict // dictionary where network parameters are stored
3      network_update_step: int // network update step associated with
        network parameters
4      phase: int // specifies the learning phase for curriculum learning
5      run_complete: boolean ← False // specifies whether the run is
        complete
6  def update_parameters($\boldsymbol{\theta}'$, network_update_step, phase, run_complete):
    // update values of attributes
7      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}'$
8      network_update_step ← network_update_step
9      phase ← phase
10     run_complete ← run_complete
11 def fetch_parameters():
       // return attributes
12     return $\boldsymbol{\theta}$, network_update_step, phase, run_complete
```

Stellenbosch University https://scholar.sun.ac.za

Algorithm 19: Learner for updating the parameters of the main network.

```
def init(replay_buffer, parameter_server):
    $Q(;\boldsymbol{\theta})$: DQN // main deep q-network
    $\hat{Q}(;\boldsymbol{\theta}^{-})$: DQN // target deep q-network
    replay_buffer ← replay_buffer // reference to replay buffer
    parameter_server ← parameter_server // reference to parameter
     server
def load_minibuffer: // in a background thread
    while true do
        transitions, indices, is_weights ← replay_buffer.sample(batch, $\beta$)
        // sample transitions along with their indices (location in
        replay buffer) and importance sampled weights
        transitions_tensor ← convert_to_tensors(transitions) // convert
        transitions to tensors
        mini_buffer.append((transitions_tensor, indices, is_weights)) // append
         tuple to the mini buffer
    end
def run_learner(num_network_updates):
    for $t \leftarrow 1$ to num_network_updates do
        experience_batch, indices, is_weights ← pop_minibuffer()
        states, actions, discounted_returns, nth_states, dones ← experience_batch
        current_qvalues ← get_current_q(states, actions)
        nth_qvalues ← get_nth_q(nth_states, dones)
        target_values ← nth_qvalues $\times\gamma+$ discounted_returns
        $\boldsymbol{\delta}$ ← target_values − current_qvalues
        replay_buffer.update_priorities(indices, td_errors)
        loss ← smooth_l1_loss(current_qvalues, target_values)
        loss ← (loss × is_weights).mean()
        update_network(loss)
        if t % update_interval == 0 then
            update_parameter_server()
        end
        if $t \% C == 0$ then
            $\boldsymbol{\theta}^{-} \leftarrow \boldsymbol{\theta}$
        end
    end
```

Stellenbosch University https://scholar.sun.ac.za

Algorithm 20: Actor for generating environment transitions.

```
def init(replay_buffer, parameter_server):
    replay_buffer ← replay_buffer // reference to replay buffer
    parameter_server ← parameter_server // reference to parameter server
    $Q(;\boldsymbol{\theta})$: DQN // initialise actor network
    environment: MiniWorld_Wrapped // initialise wrapped environment
    local_buffer: List // initialise local buffer
def run_actor($T$):
    $\boldsymbol{\theta}$ ← parameter_server.parameters() // remote call to obtain network parameters
    $s$ = environment.reset() // reset environment and get state
    for $t \leftarrow 1$ to $T$ do
        $a$ ← epsilon_greedy($s$) // with probability $\varepsilon$ select a random action otherwise select action with maximum q-value
        $s$, $r$, $d$ ← environment.step($a$) // perform action in environment and retrieve: state, reward, done
        add_to_local_buffer($s$, $r$, $d$) // add data to local buffer
        if local_buffer.size() $\geq B$ then // size of local buffer larger than $B$
            replay_buffer.add(local_buffer, $\boldsymbol{\delta}$) // add local buffer along with TD-errors (to compute priorities) to replay buffer
            local_buffer.clear() // clear local buffer
        end
        if $d$ then // if episode is done
            $\boldsymbol{\theta}$ ← parameter_server.parameters() // fetch new network parameters if available
            s ← environment.reset() // reset environment and get state
        end
    end
```

Stellenbosch University https://scholar.sun.ac.za

Algorithm 21: The simulation class where all components of the system are instantiated.

```
attributes:
    replay_buffer: Prioritised_Replay_Buffer
    parameter_server: Parameter_Server
    actors: list[Actor ]
    learner: Learner
def init(acting_steps: int, network_update_steps: int):
    for actor ∈ actors do // pre-load the replay buffer
        actor.run_actor(steps)
    end
    learner.run_learner(network_update_steps) // start learner process
     for an allocated number of steps
    for actor ∈ actors do // let actors run indefinitely
        actor.run_actor($\infty$)
    end
    wait for learner to finish
    send message to terminate actors
    wait for actors to terminate
```