Perfect Information versus Imperfect Information in Reinforcement Learning Using A Custom Developed Roguelike Environment

Third Year Project

Author:
Sultan AlRashed

Degree Program:
BSc (Hons) Artificial Intelligence

Supervisor:
Dr. Viktor Schlegel

Department of Computer Science
University of Manchester
United Kingdom
April 2022

Abstract

This report explores the effects of using perfect and imperfect information in a situation where the environment, the reinforcement learning algorithms, how the state is interpreted to the agent, and the neural network architecture are all controlled. Surprisingly, all reinforcement learning algorithms performed better in imperfect information than in perfect information when using multi-layer perceptrons as their basis; only when we use convolutional neural networks does the agent perform better in perfect information. Therefore, this report proposes treating an agent's access to information as a hyperparameter to be adjusted for maximal performance.

Contents

Chapter 1

Introduction

Whether it is a game of chess[45] or an attempt at accurately forecasting the stock market[25], artificial intelligence agents are delicate pieces of code that can outperform a human being in some tasks. The reason I use the word 'delicate' in my description of them, though, is due to my own experiences using them.

Training Artificial Intelligence agents is a process with many parameters to optimise, leading to a dramatically different result. The focus of this paper will be one of these variables; to be specific, we will see the effect of perfect information versus imperfect information on the agent's training process. Perfect information means the agent has all the information about the game at his disposal. In contrast, imperfect information means that the agent is missing some pieces of information about the game.

Changing the environment from perfect information to imperfect information (and vice versa) is done by giving the agent a full view of the map or a partial view of the map based on how much of the map it has already explored.

The hypothesis is that the agent will have a more accessible time training in an environment with perfect information. The reason is that the agent will view the entire map, making better decisions.

1.1 Motivation

In artificial intelligence, there has been a focus on developing an algorithm that can reliably succeed in most benchmarks simply through learning by experience. The pursuit of such an algorithm has led some people to search for a "holy grail" solution that applies to many applications in life, a general artificial intelligence[13][14].

In pursuit of this, many have created and assessed various algorithms. These algorithms are of varying types; this report will focus on reinforcement learning. One of the ways reinforcement learning algorithms can be assessed is by testing them in distinct environments. These environments can range from existing software, such as Atari games[29], to environments entirely built from the ground up to aid testing[39].

Although numerous parameters' effects on reinforcement learning algorithms have been investigated, some are left unexplored. Parameters such as human guidance and state-space size have been investigated previously[46]. In contrast, the effect of perfect and imperfect information on agent performance in a controlled environment has not been studied. The reason may be because the conclusion is seemingly apparent; an algorithm should perform better when more information about the game is available.

In lieu of this, this report focuses on the creation of a highly modular custom roguelike game, with the ability t

The environment is a roguelike dungeon crawler; this environment presents a neat and sensible way to test perfect information versus imperfect information. There are many reasons for this:

- The map's visibility can change to give a full view versus a limited view quickly.

- Roguelike games do not involve carrying progress over to the next game, allowing for a contained run and easy to measure metrics.
- Video game statistics are easy to measure to gauge performance (for example, amount of enemies killed).
- Roguelikes allow for randomised runs each time, preventing the agent from simply memorising the correct actions.

1.2 Aims

This project's goals are two-fold, with the first and foremost objective investigating the effects of perfect and imperfect information in a controlled environment. It bears verifying and repeating conclusions that some may deem obvious, as variations from the expected obvious result can be a tremendous boon for further research to expand on. The second goal is to create an easily extendable custom environment that allows for further exploration of the subject matter explored in this report and the ability to reproduce the results seen throughout. Roguelike games are the perfect environment due to their controlled randomness, inherent customizability, and ease of interfacing with standard reinforcement learning libraries.

1.3 Objectives

The objectives are as follows:

1. Create a roguelike game that has the following minimum attributes:
   - Endless gameplay loop.
   - Game reset on death.
   - Procedural generation of dungeons.
   - Ability to set a particular seed for fixed dungeon generation.
   - Enemies to combat.
   - Potions for healing.
   - Dungeon levels can be traversed through exits.
2. Create an environment for the agent to interact with the game.
3. Test perfect information for three different reinforcement learning algorithms.
4. Test imperfect information for the same three reinforcement learning algorithms.
5. Get empirical results of the performance difference of perfect information against imperfect information.
6. Analyse potential reasons for the performance discrepancy for the three reinforcement learning algorithms.

1.4 Report Structure

The report is structured as such:

1. Introduction: to introduce the reader to what this project explores.
2. Technical information: to give the reader a basic overview of the technologies and terminology used in the project.
3. Related works: to give a brief overview of related works that helped in the research and development of this project.
4. Game implementation: to describe to the reader how the game was implemented and the features it includes in detail.
5. Environment implementation: to give the reader a brief overview of how the game was interfaced with specific technologies to allow for the training and testing of reinforcement learning algorithms.
6. Evaluation and experiments: the results of the experiments are displayed with some discussion about how these results connect with current hypotheses and general understanding.
7. Conclusion and discussion: the report's analysis is given a conclusion and a discussion of the consequences of the project's results and how this project can be researched further.

1.5 Impact of Covid-19

Covid-19 has not affected the results of this project, but it has affected the production of its report. Due to a family member and myself falling ill from Covid-19, it was quite difficult to focus as much of my effort and dedication as I would have liked. Regardless, the code for this project had been completed before my family member, and I fell ill.

Chapter 2

Technical Information

This chapter will lay out the foundational knowledge of the technologies, terminologies, and definitions necessary to understand this report on a higher level. Firstly we will start with some definitions of specific terminology that some readers may not have heard before, which are used throughout the report regularly.

Afterwards, we will begin exploring some game concepts, such as perfect and imperfect information, which are the fundamental basis of the report. Alongside perfect information and imperfect information, we will cover the roguelike genre of games since this project involves the creation of a roguelike game from scratch to use as our testing environment.

We will follow this up by touching on neural networks, which form the basis of deep reinforcement learning methods. With our knowledge of neural networks, we can begin understanding three essential reinforcement learning methods that set up our analysis of perfect and imperfect information.

Finally, we will elaborate on some external libraries utilised to aid in creating our environment, allowing readers to reproduce the results seen within this report easily.

With all of this technical information, our analysis of perfect and imperfect information will include further depth, and the reader will have a much better background of the technologies used throughout this project.

2.1 Definitions

Clearing up some key definitions will help make this report more easily digestible to those unfamiliar with some of the concepts presented:

- Perfect Information: In a game, perfect information implies all players have complete and instantaneous knowledge of the current state and history of the game.
- Imperfect Information: In a game, imperfect information implies a player's lack of knowledge about the game's current state of history.
- Procedural Generation: A method of creating data in an algorithmic way rather than manually.
- Seed: In the context of games, a seed is a set of alphanumeric characters that fixes the randomness in world generation if it is present.
- Dungeon: In the context of games, a dungeon is a set of rooms the player can explore.
- Roguelike: A genre of games where players traverse dungeons in procedurally generated levels, the gameplay is turn-based, and the permanent death of the player is a key tenant.

- Artificial Neuron: Nodes that are rudimentary models of neurons present in biological brains. Typically aggregated into different layers[32].
- Neuron Connections: An edge between two nodes that carries a weight that typically changes as learning occurs. The weight is responsible for determining the strength of the connection[32].
- Artificial Neural Networks: Computing systems inspired by the functionality of biological neural networks present in animal brains[32].
- Multi-Layer Perceptrons: A fully connected variety of artificial neural networks where the connections between neurons do not form a cycle[37].
- Convolutional Neural Networks: A variety of multi-layer perceptrons that excel at processing visual imagery[33].
- Epoch: A complete loop over the data set. In our case, this would be one completed attempt of the game.
- Reinforcement Learning: A method for training algorithms / biological organisms that relies on punishing undesired behaviour while simultaneously rewarding desired behaviour[47].
- Deep Learning: Refers to a wide range of machine learning methods that rely on artificial neural networks, where deep implies using multiple layers in the network.
- State Space: A set of all the states that the agent can transition to.
- Action Space: A set of all actions the agent can act out in an environment.

2.2 Games

To understand this report better, a fundamental understanding of some game concepts is necessary; this section will cover perfect information, imperfect information, and the roguelike genre of games. By understanding these concepts better, our analysis, later on, will make more sense.

2.2.1 Perfect Information

Perfect information games are a class of games in which each player is completely informed of the initialisation event of the game (the cards given to each player in the beginning of a card game) and all events that have previously occurred[31].

Figure 2.1: Chess is a great example of a perfect information game.

Due to the number of potential states in the game, the optimal strategy for some games can sometimes be too challenging to compute, making it impossible to determine in a reasonable finite amount of time.

2.2.2 Imperfect Information

Imperfect information games imply a lack of information for a player in at least either the initialisation stage or the previously occurred events[48].

Figure 2.2: Gambling card games such as poker or blackjack are a great example of imperfect information.

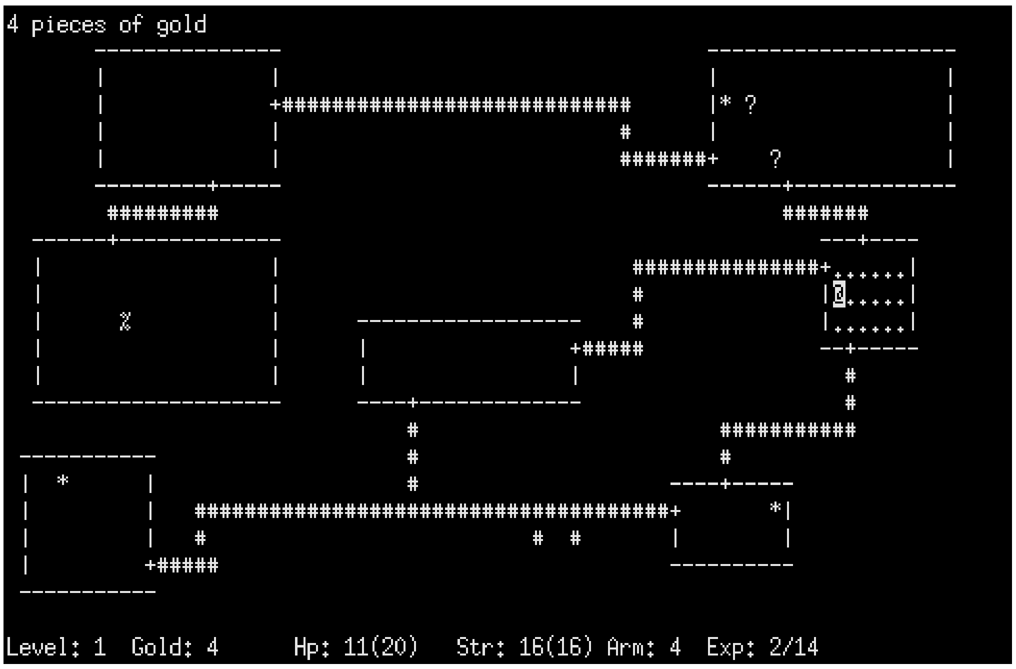
2.2.3 Roguelike Games

Roguelikes are a genre of video games where the player, upon death, loses all progress. They are also dungeon crawlers, a type of scenario in video games in which the player navigates a labyrinth environment, battling various monsters and looting any treasure they may find, progressing more profound into the dungeon[15].

Recently, roguelike games have been expanded to include some of the most critically acclaimed games released. The reason for this is the open-ended and creative nature of the genre, which allows it to explore new mechanics, locales, and play styles.

Although left open-ended, the roguelike genre still bears three key elements that define the genre as a whole:

1. The player loses all progress upon death: This mechanic prevents the player from carrying over progress from one run to the next (with a run being a play through from life to death).
2. Procedurally generated levels: The dungeon levels are randomly and continuously generated every run to prevent the player from memorising specific actions that lead to victory.
3. Dungeon crawl: The player progresses through a 'dungeon' getting deeper into it, thereby increasing score and difficulty simultaneously.



```
4 pieces of gold

Level: 1  Gold: 4      Hp: 11(20)   Str: 16(16) Arm: 4  Exp: 2/14
```

Figure 2.3: Screenshot of the game Rogue, from which all roguelike games are based[18]

2.3 Artificial Neural Networks

Artificial neural networks (ANNs) are a valuable concept to understand as they tie directly into deep reinforcement learning algorithms[16]. ANNs are computational systems that take inspiration from biological neural networks in animal brains, with it being based on a collection of neurons representing biological neurons[53]. These neurons form connections, called edges, with other neurons acting similarly to biological synapses by transmitting the information.

An artificial neuron takes real numbers, called signals, and processes them to neurons connected to it. The processing happening in each neuron is dictated by some non-linear function that sums the neuron's inputs. Neurons and edges both possess a weight that is changed as learning occurs; this weight is responsible for decreasing or increasing the strength of connections.

Neurons in ANNs are aggregated into layers, each layer performing different types of transformations on the input provided. The first layer is responsible for taking in the input, while the final layer is responsible for deciding the neural network's output. Layers between the input and output layer are typically called hidden layers, and signals travel from the input layer to the hidden layer and end at the output layer.

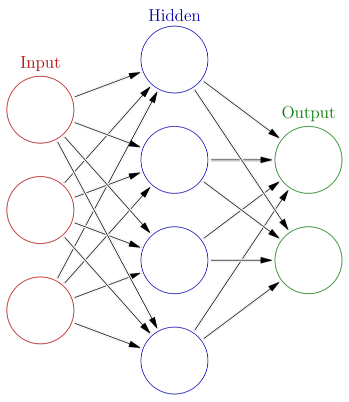


Figure 2.4: Figure showing layers of artificial neural network.

There are many different types of ANNs, but for this report, we will only focus on two key types that are utilised as the foundation for the three key reinforcement learning algorithms we will explore later.

2.3.1 Types of Networks

Multi-Layer Perceptron

A multi-layer perceptron (MLP) is a non-cyclical feed-forward artificial neural network with neurons fully connected between each layer[36]. An MLP, at minimum, consists of three separate layers: an input layer, a hidden layer, and an output layer.

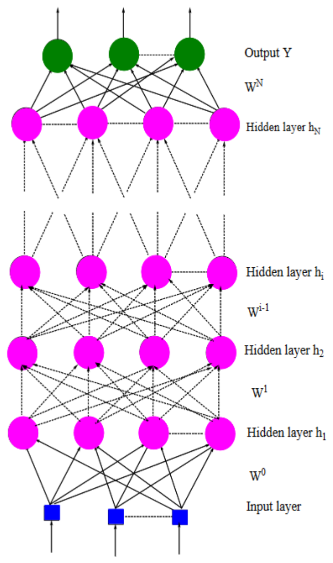


Figure 2.5: An example of a multi-layer perceptron[36].

Each neuron within the architecture utilises a non-linear activation function, which, alongside the multiple layers, helps MLPs distinguish data that is not linearly separable.

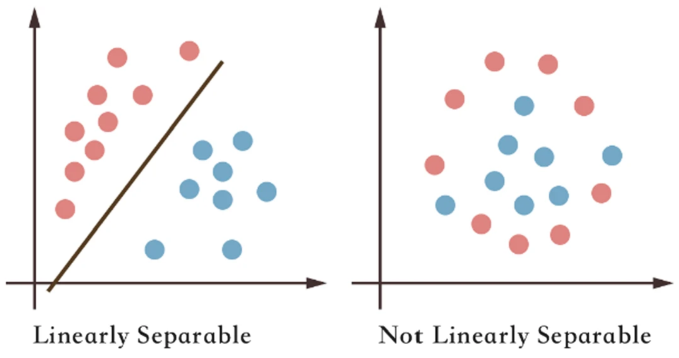


Figure 2.6: Visualization of linear separability[54].

MLPs utilise backpropagation[38], a supervised learning technique, to aid the training process. Backpropagation involves calculating the gradient of the loss function while taking into account the weights of the network efficiently. With this efficiency achieved, utilising gradient descent methods such as stochastic gradient descent[21] for training the network becomes feasible.

Convolutional Neural Network

Convolutional neural networks (CNNs) are a class of artificial neural networks that specialize in processing data that has a grid-like topology, such as an image[52].

CNNs take inspiration from a human brain's visual cortex, with each neuron working on its own receptive field. Layers in a CNN are arranged to detect simple patterns first (like lines and curves) and gradually get on to detect more complex patterns (like faces)[26].

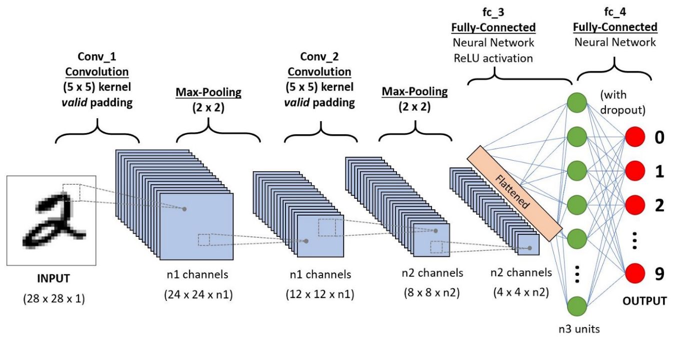


Figure 2.7: Visualization of a convolutional neural network[52]

CNNs consist of two key layers, the convolution layer and the pooling layer. The convolution layer is responsible for the bulk of the computational load in the network. The convolution layer is responsible for processing the image in waves, while the pooling layer derives a summary statistic of the image (which helps reduce the spatial size of the representation, thereby reducing computational load)[22].

2.4 Reinforcement Learning

Reinforcement learning is based on rewarding positive behaviour and punishing negative behaviour to allow an agent to perform well in a particular environment. Rewarding and punishing behaviour is accomplished by having the agent perform an action on the environment, which is then interpreted into the next state of the environment and rewards for the agent[51].

The goal of reinforcement learning is to produce a policy able to handle a particular problem optimally[51]; these problems can encompass game theory, information theory, simulation-based optimisation, control theory, and statistics. The policy is achieved by an agent training to maximise a reward function. A reward function is a function that rewards and punishes certain behaviours of the agent. In biology, rewarding and punishing behaviours can be seen, where brains interpret pain as negative reinforcement[20] and pleasure as positive reinforcement[7].

Reinforcement learning is a multi-faceted and still blooming field. However, this section will explore three key reinforcement learning algorithms used in this project to explore the effects of perfect information versus imperfect information: proximal policy optimisation, deep-q networks, and advantage actor-critic. Firstly to understand the three reinforcement learning algorithms explored, it would be wise to look at the subject of Markov decision processes, thereby making it easier to understand the three key reinforcement learning algorithms explored in this report.

2.4.1 Markov Decision Process

Markov decision processes are vital to understanding how reinforcement learning on a fundamental level. Hence, it is prudent to explore what they are so we can discuss the different reinforcement learning algorithms explored in this report. A Markov decision process is a discrete-time stochastic control process. It helps model decision making if the results of the decision are dictated by both randomness and the decision-maker[6].

Definition 1 (Markov Decision Process)**.** *A Markov decision process consists of a 4-tuple* (S, A, P_a, R_a) :

- *S: A set of states for both the environment and the agent.*

- *A: A set of possible actions that the agent can take from each state.*
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ *which signifies the probability that at time t in state s if action a the next state transitioned to at time* $t + 1$ *will be* s'.
- $R_a(s, s')$ *is the expected immediate reward after the transition to* s' *from s is performed (using action a).*

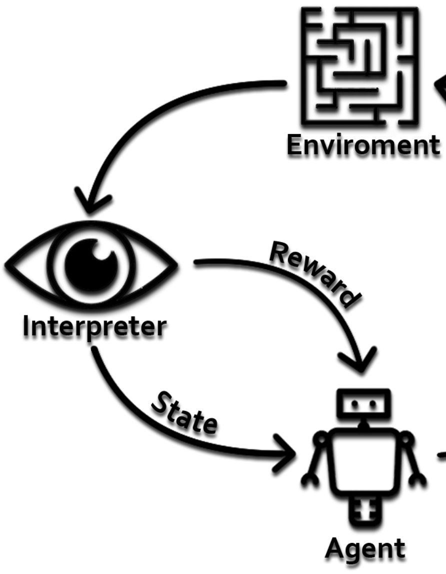


Figure 2.8: Example of the process of training an artificial intelligence agent through reinforcement learning.

Reinforcement can be modeled as a Markov decision problem. A reinforcement learning agent will interact with its environment in discrete time steps. At each time step t the agent receives the current state of the environment $s_t \in S$ (where S is the set of possible states) and the rewards achieved by its previous action $r_t \in R$, then from the list of actions available the agent chooses an action $a_t \in A(s_t)$ (where $A(s_t)$ is the set of available actions in state s_t) which is finally sent to the environment to be interpreted into the next reward $r_{(t+1)} \in R$, and state $s_{(t+1)}$. This process is repeated until the agent reaches a satisfactory policy, denoted π_t where $\pi_t(s, a)$ is the probability $a_t = a$ if $s_t = s$, that maximizes the expected reward.

We get a set of four definitions that can help us describe the function of reinforcement learning methods.

Definition 2 (Policy)**.** *A policy, $\pi(s)$, is a function that returns an action in state s in an attempt to maximize reward.*

$$\pi(s) = Pr(s_t = s, a_t = a) \tag{2.1}$$

The following definition determines the accumulated reward.

Definition 3 (Accumulated Reward)**.** *The total accumulated reward RS_t is calculated using a discount factor, γ, while summing all rewards from timestep t.*

$$RS_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1} \tag{2.2}$$

To determine a policy's reward, we need to find the expected reward of following a policy π to determine a policy's reward.

Definition 4 (Action Value Function)**.** *The action-value function represents the expected reward of following policy π in state s and taking action a.*

$$q_\pi(s,a) = E_\pi[RS_t | S_t = s, A_t = a] \tag{2.3}$$

Definition 5 (Optimal Action Value Function)**.** *An optimal action-value function q_* can be defined as the action-value function that receives the most reward overall policies.*

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.4}$$

Most reinforcement learning algorithms provide their unique spin on how to handle Markov decision processes while still maintaining the goal of finding a policy π such that $q_\pi(s,a) = q_*(s,a)$. It is best for this report to focus on three key algorithms: deep-q networks, proximal policy optimisation, and advantage actor-critic. A surface-level overview would be helpful to understand the bias present in some of these algorithms.

2.4.2 Trust Region Policy Optimization

Trust region policy optimisation (TRPO) was introduced in a paper by Schulman, Levine, Moritz, Jordan, and Abeel[41]. The paper describes an iterative method for optimising policies, the algorithm being similar to natural policy gradient methods.

Definition 6 (Policy Gradient Methods)**.** *Policy gradient methods take a stochastic gradient descent algorithm and use a computed estimator of the policy gradient in conjunction with it[41]. A common form of a gradient estimator is*

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.5}$$

Where:

- $\hat{\mathbb{E}}_t$*: the empirical average over a finite batch of samples.*
- π_θ*: stochastic policy.*
- $\hat{A}_t$*: an estimator of the advantage function at timestep t.*

Definition 7 (TRPO Objective Function)**.** *TRPO attempts to maximize the following function:*

$$L^{CPI} = \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t] \tag{2.6}$$

2.4.3 Proximal Policy Optimization

Proximal policy optimisation (PPO) was initially introduced in a paper by Schulman, Wolski, Dhariwal, Radford, and Klimov in collaboration with OpenAI[42] and builds on TRPO. With this method, alternations between optimising an objective function using stochastic gradient descent and data sampling through interaction with the environment help produce a policy model that is balanced when it comes to complexity, simplicity, and time.

Definition 8 (Optimising Objective Function)**.** *The paper proposes the following objective function, which is intended to be maximised:*

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \tag{2.7}$$

Where:

- θ*: is a hyperparameter.*
- ϵ*: is a hyperparameter*
- $\hat{\mathbb{E}}_t$*: represents the empirical average over a finite batch of samples.*
- $\hat{A}_t$*: is an estimator of the advantage function at timestep t.*
- $r_t(\theta)$*: represents the probability ratio* $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ *from TRPO's surrogate objective function.*
- $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$*:*

Proximal policy optimisation algorithms can take different forms that modify the above function to attain different results. The family of policy optimisation methods use multiple epochs of stochastic gradient descent to perform each policy update. Proximal policy optimisation presents the same stability and reliability seen in trust-region methods while being much simpler to implement[42].

2.4.4 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that primarily relies on learning to achieve objectives instead of model-based algorithms that rely on planning[50].

Q-learning works by composing a table of states and actions in that state; Q-learning then attempts to maximise the reward by learning the appropriate action to take at each state. The initial value of each table cell is initially set to zero. The cells in the table are then populated by a value, called the Q value, generated through a bellman equation which is as follows:

Definition 9 (Updating Q Value in Table)**.**

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \times (r_t + \gamma \times \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.8}$$

Where:

- $Q(s_t, a_t)$*: the old Q value.*
- α*: the learning rate.*
- r_t*: the reward.*
- γ*: the discount factor, which has to the goal of valuing rewards received earlier higher than those received later.*
- $\max_a Q(s_{t+1}, a)$*: estimate of the optimal future value.*

Given a partly random policy and infinite exploration time, Q-learning eventually converges to an optimal action-selection policy for any finite Markov decision process[27].

2.4.5 Deep-Q Networks

Deep-Q network (DQN) was an algorithm developed by the DeepMind team in 2015[30]. Its technical prowess was demonstrated after being able to solve a swath of Atari games at levels that far exceed a human's performance. This was accomplished by combining Q-learning and a deep convolutional neural network.

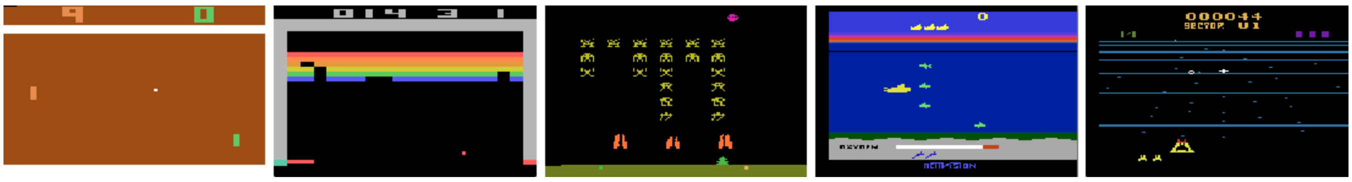


Figure 2.9: Screenshots of five different Atari games.(Left-to-right) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

The DeepMind team utilised a deep convolutional neural network, a class of artificial neural networks commonly used for visual imagery, to get their agent to perform exceptionally at Atari games.

In general, however, a neural network is used to approximate the Q value filled out into tables during Q-learning. The state is the input, while the output is the Q value of all possible actions.

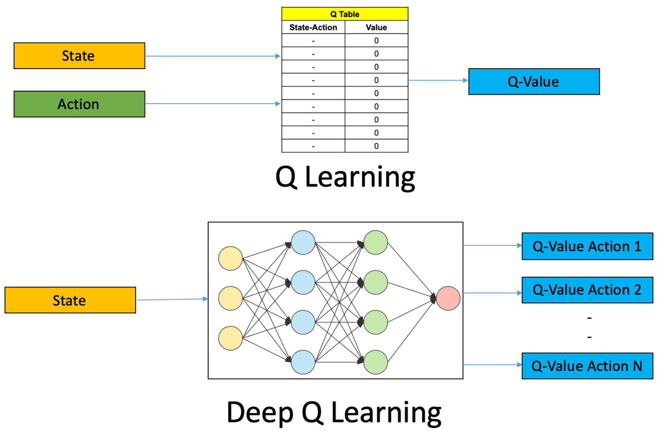


Figure 2.10: A visualised example showing how DQN differs from Q-learning

2.4.6 Advantage Actor Critic

The advantage actor-critic (A2C) algorithm combines two reinforcement learning algorithms, value-based and policy-based algorithms[28]. In value-based algorithms, the agent selects actions based on the predicted value or action. In policy-based algorithms, the agent attempts to learn a policy that maps input states to actions.

In actor-critic methods, such as advantage actor-critic, there is a critic and an actor:

- The critic estimates the value function. The value can either be the state-value or action-value (the Q value, as seen in Q-learning).
- The critic directs the actor to change the policy distribution as required.

Advantage actor-critic is a synchronous deterministic variant of an already existing algorithm, asynchronous advantage actor-critic (A3C), that waits for all actors to finish their experiences before updating by averaging over all the actors. The key benefit is the increased utilisation of graphics processing units when using larger batch sizes.

2.5 OpenAI

OpenAI is an AI research and deployment company that develops many technologies in the field while ensuring a lot of them are open source[2]. One of these open source technologies is OpenAI Baselines[1] which provides a library of high-quality implementations of standard reinforcement learning algorithms in Python.

2.5.1 Gym

Gym is a library created by OpenAI to develop and compare reinforcement learning algorithms[9]. The strength of Gym comes from the fact that unlike libraries such as the Arcade Learning Environment[5] it does not assume the environment used for training and testing. Gym also supplies the tools necessary to develop custom environments that interface with the developer's software, usage of this feature can be seen in appendix A.

Gym also provides the libraries necessary to train and test agents in an environment; in this project, however, Gym was only used for testing, with Stable Baselines 3 being used instead for training due to the number of features and documentation available. An example of using Gym to test can be seen below, with the extended version available in appendix B.

```python
from gym import Env
from stable_baselines3 import PPO

def test_model(model: PPO, env: Env) -> None:
    """
    Tests model

    Args:
        model (PPO): The model to test against.
        env (Env): The environment used for the model.
    """

    # Number of episodes to test
    episodes = 10
    for episode in range(1, episodes+1):

        # Gets environment's initial state
        obs = env.reset()
        done = False
        score = 0
        turn = 0

        # Loops until environment is done.
        while not done:

            # Returns action model predicts to be best according to algorithm used.
            action, _ = model.predict(obs)

            # Returns current state, reward, custom info, and whether episode is done.
            obs, reward, done, info = env.step(action)
            score += reward
            turn += 1
```

Listing 2.1: Testing model performance using Gym (removed logging functionality in original project for a reduction in text size).

2.5.2 Stable Baselines 3

Stable Baselines 3 is an improvement and a fork made of OpenAI Baselines; the algorithms present are accompanied by substantial documentation allowing for ease of use[35].

Many of the features provided are used in this project, including:

- Proximal Policy Optimization: Stable Baselines include their algorithm implementation.
- Deep-Q Networks: Stable Baselines includes an implementation of the algorithm.
- Advantage Actor-Critic: Stable Baselines includes an implementation of the algorithm.
- Vectorised Environments: Stable Baselines includes environments that allow for stacking multiple independent environments into a single environment (allowing for faster training time).
- Tensorboard Integration: Stable Baselines allows easy tensorboard integration during training evaluation.

```python
from gym import Env

from stable_baselines3 import PPO, A2C, DQN
from stable_baselines3.common.vec_env import SubprocVecEnv

# PPO Implementation
def create_train_model_PPO(path_to_save: str, total_timesteps: int, env: Env) -> PPO:

    # Implementation of the sub process vectorized environment allowing for multi processing during
     training.
    env = SubprocVecEnv([lambda: env])

    # Tensorboard integration is as simple as adding a log directory for tensorboard files.
    model = PPO('MlpPolicy', env, verbose=1, tensorboard_log="Environment\\Logs\\")
    model.learn(total_timesteps=total_timesteps)
    model.save(path_to_save)

    return model

# A2C Implementation
def create_train_model_A2C(path_to_save: str, total_timesteps: int, env: Env) -> A2C:

    env = SubprocVecEnv([lambda: env])

    model = A2C('MlpPolicy', env, verbose=1, tensorboard_log="Environment\\Logs\\")
    model.learn(total_timesteps=total_timesteps)
    model.save(path_to_save)

    return model

# DQN Implementation
def create_train_model_DQN(path_to_save: str, total_timesteps: int, env: Env) -> DQN:

    env = SubprocVecEnv([lambda: env])

    model = DQN('MlpPolicy', env, verbose=1, tensorboard_log="Environment\\Logs\\")
    model.learn(total_timesteps=total_timesteps)
    model.save(path_to_save)

    return model
```

Listing 2.2: Examples of using Stable Baselines 3 in the project.

Chapter 3

Related Works

This chapter will lay out the historical development of both reinforcement learning agents in games and the use of roguelikes in training reinforcement learning agents while also showing previously discovered impacts of perfect information versus imperfect information:

3.1 Reinforcement Learning in Games

3.1.1 Chess

Initially, chess was considered a game where humans reigned supreme. In 1989 at the height of chess grandmaster Garry Kasparov's career, he triumphantly announced that there would never be a chess program that could beat him. Kasparov proved this by beating Deep Thought, IBM's computer, twice that same year[34].

In 1996, Kasparov later defeated Deep Thought's successor, Deep Blue, in a six game match, with Kasparov going 4 to 2. The following year, Kasparov's announcement would finally be proven wrong, with Deep Blue going 3.5 to 2.5 (the 0.5 being a draw) against Kasparov.[CITE]

1997 was a pivotal year in which computers finally proved their mettle at beating chess grandmasters at their own game. A massive boom in reinforcement learning and artificial intelligence followed suit. Chess engines only improved as time passed, leading us to our current time where a simple program on the phone can easily outcompete the best chess players in the world without breaking a sweat. Computers have officially become better than a human being could ever be in chess.

The next game to beat for computers became Go.

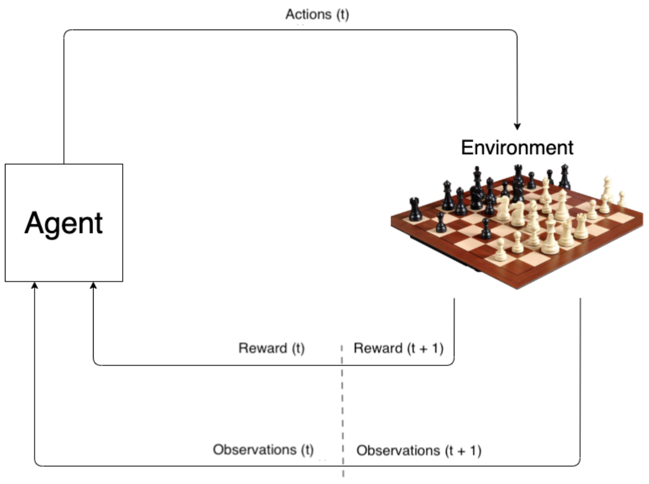


Figure 3.1: Reinforcement learning in chess.

3.1.2 Go

The game of Go is challenging for artificial intelligence to learn to master and has posed a challenge to the field as a whole when it comes to solvability or beating current champions. The reason Go is so challenging is that while chess has 10^{46} valid states, Go has 3^{361} valid states[49].

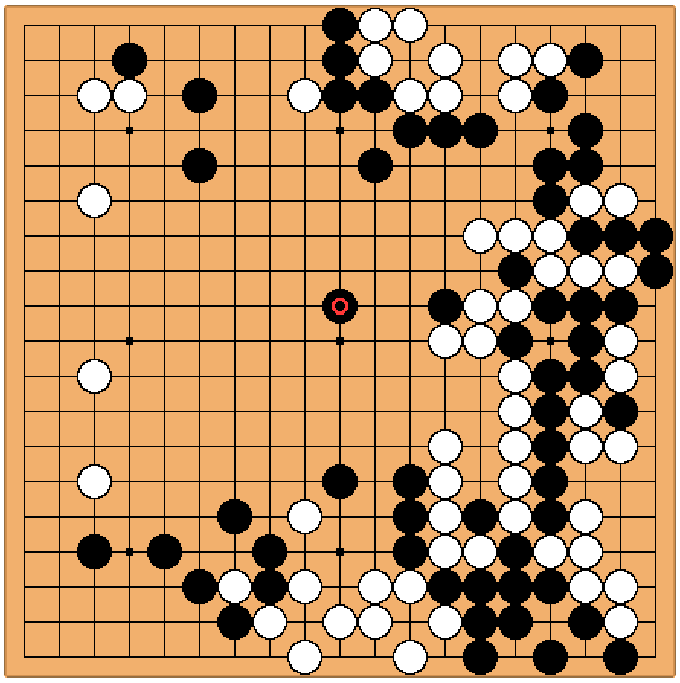


Figure 3.2: A game of Go.

Due to Go's significant branching factor, standard artificial intelligence methods that work by testing all possible moves in a game tree became prohibitively tricky as the programs struggled to compute the strength of each move and position. At best, these standard artificial intelligence programs could only operate at the level of an amateur Go player[44].

DeepMind, a subsidiary of Google, produced AlphaGo, which marked a new age for artificial intelligence in the game. In 2015, AlphaGo beat the European Go champion Fan Hui five to zero[44]. This marked the beginning of AlphaGo's eventual dominance and prominence in Go, with it being arguably the best artificial intelligence in the game.

AlphaGo's team later developed AlphaGo Zero, which was entirely self-taught. Afterwards, AlphaGo Zero was generalised into AlphaZero, which learnt to play even more games and reportedly learnt to play chess, Go, and shogi better than any human can within only 24 hours[8]. AlphaZero, later on, was succeeded by MuZero, which learns how to play without being taught the rules of the game[40].

3.2 Perfect Information

3.2.1 NeuroHex: A Deep Q-learning Hex Agent

In a paper by Young, Vasan, and Hayward, an agent was created to play the board game Hex. The researchers used deep Q-learning to train the agent, and when compared against a famous hex agent (MoHex), NeuroHex achieved a win rate of 20.4% when playing as the first player.

A key piece of information is that NeuroHex achieves this win rate without using any search compared to MoHex. It is an impressive result to witness when one considers that Hex is a perfect information game with both a massive state and action space.

3.3 Imperfect Information

3.3.1 Combining Deep Reinforcement Learning and Search for Imperfect-Information Games

The Facebook AI Research team produced a paper that proposes a framework that circumvents certain limitations imposed by the nature of imperfect games on reinforcement learning[10].

The framework proposed is nicknamed ReBeL, with it being provably able to converge to a Nash equilibrium in any two-player zero-sum game. One of the examples used is ReBeL showing superhuman performance in Texas hold 'em poker with a low amount of domain knowledge.

This exciting paper showed that it is not impossible to be exceptional in imperfect information games. Although the methodology of this paper was not used in this project, being able to reach superhuman levels in an imperfect information game inspired a bit of this project.

3.4 Perfect Information Versus Imperfect Information

3.4.1 Predicting Success in a Real-Time Strategy Game

In the paper presented by Bakkes, Spronck, Herik, and Kerbusch, an artificial intelligence agent is trained to evaluate the success of a game ahead of time in situations with perfect information and imperfect situations information[4].

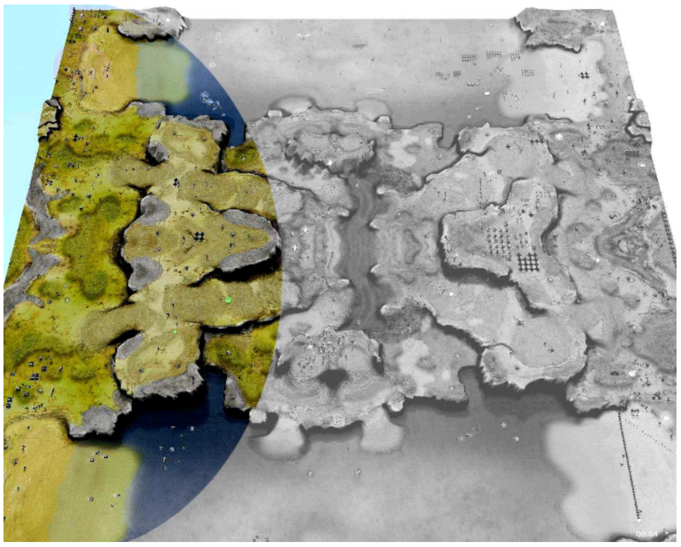


Figure 3.3: The RTS game in question, called Springer. The dark zone of the map represents player visible areas which are visible in an imperfect information environment while the light side of the map is only visible in a perfect information environment.

The researchers found that when it came to predicting success in different situations, performance in a perfect information environment was on average 1-5% better than in the imperfect information environment.

This result backs up the consensus that an agent will perform better when given perfect information.

3.4.2 Affect of Imperfect Information on Inventory Control with Product Returns

Product returns can massively disrupt supply chains and, in particular, inventory decisions[12]. The flow of product returns is uncertain and unpredictable; beating the unpredictability of product returns can make inventory decisions much more manageable. The impact of this uncertainty and imperfect information is explored in the report penned by de Brito and van der Laan[11].

Although not in the field of games, one can extrapolate inventory control with company product returns into a game of sorts where an agent tries to forecast a product return accurately. Therefore, we can attempt to extrapolate the impact of imperfect information in a situation of inventory control to a game situation and take note of the performance difference.

The most relevant result to our game situation is the (slightly obvious) conclusion that forecasting performance for product returns increases proportionally to the level of information accessible in a situation of perfect information. Alongside this result are tables showing the massive impact of imperfect information on correctly forecasting the number of product returns. The other results in the paper discuss the best strategy for a manager to use out of the methods presented in different situations, which do not apply to this report.

3.5 Roguelike Game Environments Developed for Reinforcement Learning

Roguelike game environments present an exciting way to train and test agents since they include inherent randomness. Therefore the field of creating roguelike game environments for agents is filled with beneficial knowledge and technology.

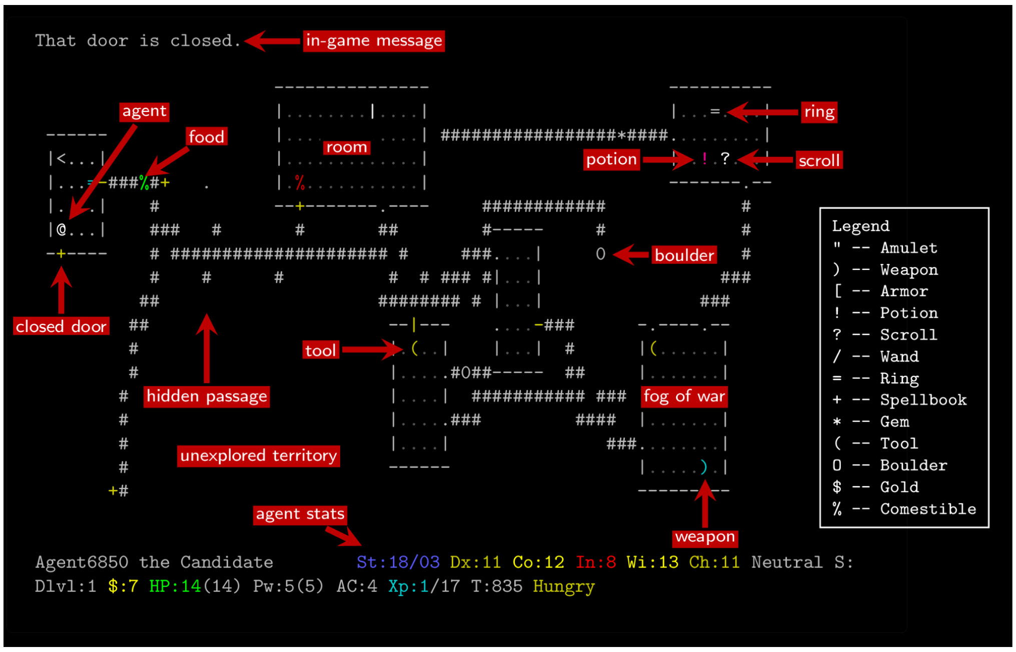
Many factors of these environments (including the randomness) can usually be adjusted through configuration files which allow for fine-tuned control of the environment to specific training cases allowing different reinforcement learning algorithms to excel. A few of these research papers are discussed below.

3.5.1 DeepCrawl

Deepcrawl presents exciting research into a different kind of avenue explored. DeepCrawl is a fully playable roguelike game for mobile platforms where the enemies are trained using deep reinforcement learning[43]. Most research on roguelike games focuses on training an agent to play as the main character, but DeepCrawl focuses on training the enemies. In DeepCrawl, the enemies act like players by moving strategically, attacking when necessary, and even using potions. DeepCrawl's main focus was finding new avenues for developing non-player characters; however, by achieving their primary goal, the researchers have managed to create an agent able to handle most aspects of a roguelike game.

3.5.2 The Nethack Learning Environment

NetHack is a popular single-player terminal based roguelike game that can be used as a scalable, stochastic, procedurally generated, challenging environment. The paper creates a new environment using NetHack as its base since the researchers argue that NetHack provides enough complexity to help explore problems such as exploration, planning, skill acquisition, and language-conditioned reinforcement learning. They also claim that the environment does not require massive computational resources to gather a large amount of experience[23].



```
That door is closed.

Legend
" -- Amulet
) -- Weapon
[ -- Armor
! -- Potion
? -- Scroll
/ -- Wand
= -- Ring
+ -- Spellbook
* -- Gem
( -- Tool
0 -- Boulder
$ -- Gold
% -- Comestible

Agent6850 the Candidate        St:18/03 Dx:11 Co:12 In:8 Wi:13 Ch:11 Neutral S:
Dlvl:1 $:7 HP:14(14) Pw:5(5) AC:4 Xp:1/17 T:835 Hungry
```

Figure 3.4: NetHack shown with labels explaining different features, items, and objects in the game.

This study shows that roguelike environments (particularly the NetHack environment) provide exciting challenges for exploration methods since there are many possible states and a wide array of possible environment dynamics to discover. An agent must be able to learn a plethora of skills, such as eating, fighting monsters, casting spells, and collecting resources, among others, to win in NetHack, which leads to further challenges for the agent.

This work is a testament to the benefits and non-performance intensiveness of roguelike environments for agent training and testing, showing that many problems can be tested in a simple and fast to run environment.

3.5.3 Rogueinabox

Similarly to the previous paper, Rogueinabox[3] provides a new learning environment built on the roguelike genre, but rather than focusing on expanding on NetHack, it builds off of the original roguelike game Rogue with highly modular elements, giving a more nuanced sense of control. (It is worth noting that the researchers for the NetHack Learning Environment promised further modularity after the release of NetHack 3.7, which adds Lua scripting to the game).

3.5.4 Rogue-Gym

By far the most valuable and vital paper for this project, in this paper by Kanagawa and Kaneko [19] a custom roguelike environment is created to evaluate reinforcement learning algorithms. The paper explicitly explores the proximal policy optimisation algorithm with and without specific enhancements to the algorithm. The results show a slight improvement in the agent's ability to generalise when specific enhancements are added to proximal policy optimisation.

This paper was beneficial because it follows much of the same initial set-up: creating a custom roguelike environment where we can test some reinforcement learning algorithms' performance. The Rogue-Gym environment also utilises many of the same libraries used in this project, allowing us to look at how a roguelike game is interfaced with the OpenAI Gym library to create a reinforcement learning environment.

Chapter 4

Game Implementation

In this chapter, we will focus on implementing the roguelike game used to train our agent. To fully understand the implementation of the game, we must touch on the following subjects:

1. Level design within the game.
2. Enemies present in the game.
3. How procedural generation is handled.
4. How randomness is handled.
5. How the game emphasises simplicity.
6. The gameplay loop that the game was built around.

Each of these points will be discussed in its respective sections.

4.1 Level Design

The dungeon levels are made up of tiles with certain properties that affect their interaction with the player and ASCII characters that represent them in-game. These tiles are as follows:

Rectangular rooms are placed randomly throughout the dungeon, and hallways are generated to connect each room to the previously generated one to ensure that all rooms are reachable from any room. Hallways are randomly determined to either start horizontally or vertically. The rooms get more extensive as the depth of the dungeon increases.

Tile	Properties	ASCII Character
Floor	Player can walk through.	.
Wall	Player can not walk through.	#
Corpse	Player can walk through.	q
Exit	Sends player to next level.	>
Potions	Heals player 10 health points.	+

4.2 Enemies

There are only two enemy types in the game, they are both described in this table with the player included for reference:

Entity	Health	Defense	Damage Output	ASCII Character
Zombie	8	1	3	z
Vampire	12	1	4	v
Player	20	2	5	@

For context, health represents the amount of damage a character can take. Defence represents the number by which attack damage is reduced. Damage output represents the amount of damage the character deals per turn. Defence exists currently for the sake of future implementations of armour.

4.3 Procedural Generation

Procedural generation is another essential cornerstone of the roguelike genre, and it is a way to create data through algorithmic and computational means rather than manually inputting it. An example of its usage in the game is the procedurally generated dungeons (levels).
The dungeons in the game are not handcrafted; instead, they are generated by following a particular set of rules:

- The rooms in the dungeon must be a minimum of 3x3, with the room size maximum increasing as the depth of the dungeon increases.
- All the rooms in the dungeon must be reachable by any other room in the dungeon.
- There must be exactly one exit tile in the dungeon.
- There must be a certain number of enemies and potions in the dungeon, with this number increasing as the depth of the dungeon increases,
- The player must spawn in the centre of one of the rooms in the dungeon.

These rules allow for a varied and balanced dungeon layout that allows the player to experience a unique run often.

4.4 Randomisation

Nevertheless, another fundamental cornerstone of the roguelike genre is randomisation. Randomisation is closely tied to procedural generation since the procedural generation follows a set of rules to generate a dungeon randomly.
Multiple factors are randomised within the game:

- Where each room is placed in the dungeon.
- Which rooms are connected by a direct corridor.
- The room the exit tile is placed.
- Which room the player spawns in.
- Size of each room.

- Whether a zombie or a vampire spawns.
- The location of zombies, vampires, and potion spawns.

Randomisation can be controlled by fixing the seed of the pseudo-random number generator to make the process of yielding number sequences deterministic when running the game or the agent. A seed is a short number that sets up the randomisation of the procedural generation of the dungeon and all random attributes; by setting it in advance, the dungeon will always be generated the same way when using the same seed.

4.5 Simplicity

The roguelike game created here focuses on simplicity; the game is reduced to the simplest form of roguelike possible. This simplicity lets the agent train as fast as possible while focusing on the critical points of this research project. The code for the game itself is extendable, allowing for the easy addition of features, enemies, tiles, room structures, and items.

Simplicity in the game is exemplified by the bump system used. A player only needs the four-movement keys up, down, left, and right to interact with the entire game.

- Enemies: if a player bumps into an enemy, they attack it.
- Potions: if a player bumps into a potion, their health increases.
- Exits: if a player bumps into an exit, they are taken to the next level with the depth of the dungeon increasing.

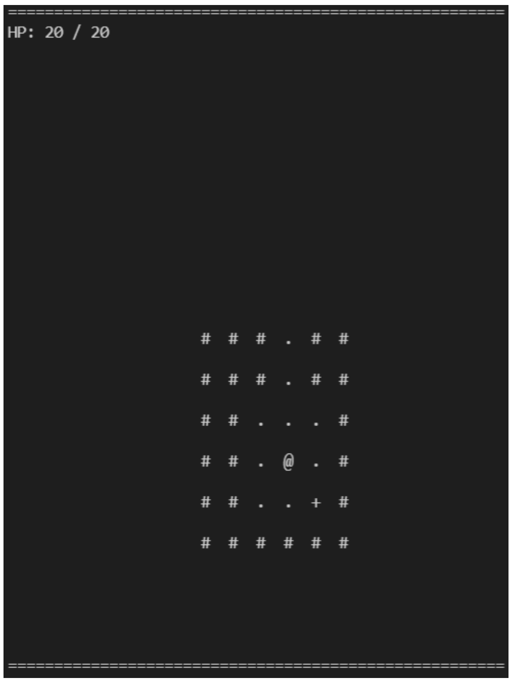
Due to the lack of weapons, armour, and currency, no inventory management is necessary, further simplifying the game.

Thanks to these reductions, the player only needs four input keys to interact with the entire game.

4.6 Gameplay

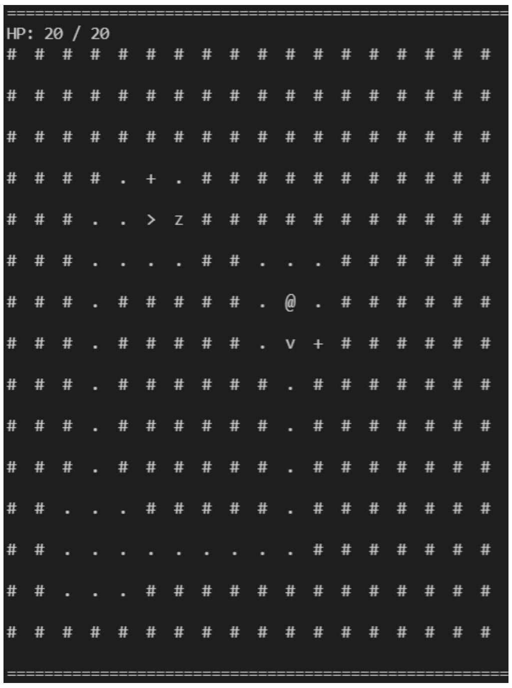
The game consists of a simple gameplay loop since emphasising simplicity was the goal of this project. We only wanted to compare the performance discrepancy between perfect and imperfect information.

The player spawns in the first level of the dungeon, unable to see due to a game concept called the fog-of-war (a concept derived from real-time strategy games such as StarCraft). Fog-of-war dissipates as the player explores more of the dungeon, keeping previously explored areas visible.



```
======================================================
HP: 20 / 20














              # # # . # #

              # # # . # #

              # # . . . #

              # # . @ . #

              # # . . + #

              # # # # # #





======================================================
```

(a) How the player initially sees a map.



```
======================================================
HP: 20 / 20
# # # # # # # # # # # # # # # # # #

# # # # # # # # # # # # # # # # # #

# # # # # # # # # # # # # # # # # #

# # # # . + . # # # # # # # # # # #

# # # . . > z # # # # # # # # # # #

# # # . . . . # # . . . # # # # # #

# # # . # # # # # . @ . # # # # # #

# # # . # # # # # . v + # # # # # #

# # # . # # # # # # . # # # # # # #

# # # . # # # # # # . # # # # # # #

# # # . # # # # # # . # # # # # # #

# # . . . # # # # # . # # # # # # #

# # . . . . . . . . . # # # # # # #

# # . . . # # # # # # # # # # # # #

# # # # # # # # # # # # # # # # # #

======================================================
```

(b) A fully visible map (through cheats).

The game's objective is to get as deep into the dungeon as possible, which is accomplished by finding and walking to the exit. On the way to the exit, the player faces enemies that attempt to stop them; the player must strategically either avoid or plan tactics to beat them. An example of one such tactic is funnelling the enemies into a corridor to handle them one at a time. Potions are available for the player to collect and restore health points (HP).

As the player gets deeper into the dungeon, the rooms become more extensive, the enemies more numerous, and the potions more plentiful to scale difficulty naturally while maintaining balance. The game does away with concepts such as inventory management, weapons, and armour as they provide unnecessary complexity to the objective of this project. The code itself is very modular, allowing for the addition of more complex functionalities as a developer sees fit.

The game loop is as such:

1. Dungeon is generated according to a given seed (or no seed for random level generation).
2. Player decides to move either up, down, left, or right.
3. Any effects from the player bumping into anything takes effect.
4. Enemies in explored regions perform their turn, attempting to close the gap between them and the player and attack them.

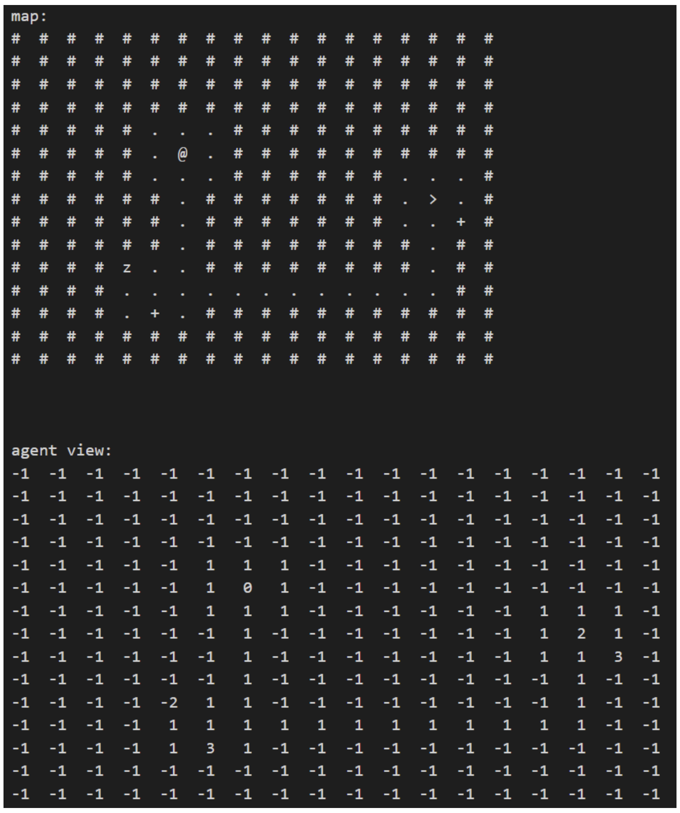
Chapter 5

Environment Implementation

This section will discuss how the environment that interacts with our roguelike game was implemented. Describing the implementation in detail will help in two regards. Firstly, it will allow readers of this report to replicate the results seen in this report. Secondly, it will allow readers to build on the experiments performed.

5.1 Agent Interaction with Game

The agent interacts with the game through the four actions: up, down, left, and right. The agent's view of the game's map is composed of numerical labels showing the map,



```
map:
#  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  .  .  .  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  .  @  .  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  .  .  .  #  #  #  #  #  #  .  .  .  #
#  #  #  #  #  #  .  #  #  #  #  #  #  #  .  >  .  #
#  #  #  #  #  #  .  #  #  #  #  #  #  #  .  .  +  #
#  #  #  #  #  #  .  #  #  #  #  #  #  #  #  .  #  #
#  #  #  #  z  .  .  #  #  #  #  #  #  #  #  .  #  #
#  #  #  #  .  .  .  .  .  .  .  .  .  .  .  .  #  #
#  #  #  #  .  +  .  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #



agent view:
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1   1   1   1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1   1   0   1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1   1   1   1  -1  -1  -1  -1  -1  -1   1   1   1  -1
-1  -1  -1  -1  -1  -1   1  -1  -1  -1  -1  -1  -1  -1   1   2   1  -1
-1  -1  -1  -1  -1  -1   1  -1  -1  -1  -1  -1  -1  -1   1   1   3  -1
-1  -1  -1  -1  -1  -1   1  -1  -1  -1  -1  -1  -1  -1  -1   1  -1  -1
-1  -1  -1  -1  -2   1   1  -1  -1  -1  -1  -1  -1  -1  -1   1  -1  -1
-1  -1  -1  -1   1   1   1   1   1   1   1   1   1   1   1   1  -1  -1
-1  -1  -1  -1   1   3   1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
```

Figure 5.1: A map of the game can be seen above as visible to a human player, below is what can be seen by the agent through the numerical labelling.

To elaborate on how the labelling works, this table shows the conversions:

What it is	Numerical Label	ASCII Character
Wall	-4	#
Vampire	-3	v
Zombie	-2	z
Corpse	-1	q
Unexplored	0	
Player	1	@
Floor	2	.
Potion	3	+
Exit	4	>

Table 5.1: Table showing level translation into labels.

By labelling the map, the observation space is strictly limited to real numbers from -4 to 4 rather than all ASCII characters. The hope was that the agent would understand the game more straightforwardly with a strict observation space reduced to the minimal amount of necessary symbols. Labelling the map also reduces the computational workload necessary since giving the agent a full RGB pixel array of the current state is a resource-intensive process that requires generating a snapshot of the current game.

How the environment handles this interaction is as follows:

1. The game engine generates the next level.
2. The environment translates the map for the agent.
3. Engine checks agent's current field of vision
4. The agent's action is translated into in-game action.
5.

5.2 Reward Function

The reward function is key to the agent's performance. Over varied testing, the agent performed best when only rewarded for two actions, discovering new areas of the map and reaching a new depth into the dungeon (stepping on an exit tile).

The reward is as follows:

Action	Reward
Exploring new tiles	(amount of new tiles explored in one turn) ÷ 2
Taking the exit	150

5.2.1 Justification

The process of verifying which reward function is best was an arduous one, involving a lot of trial and error. Picking the best result relied on using a specific statistic to benchmark agent performance. In this case, the number of exits taken was determined to be the ideal benchmark since taking as many exits as possible was the game's primary objective. The most important reward functions tested are listed below:

1. Taking exits: +150, exploring new tiles: +(amount of new tiles explored in one turn ÷ 2).
2. Taking exits: +100, exploring new tiles: +(amount of new tiles explored in one turn ÷ 2).
3. Taking exits: +150, exploring new tiles: +(amount of new tiles explored in one turn ÷ 2), bumping into a wall: -1, dying: -50, time running out: -75.
4. Taking exits: +150, exploring new tiles: +(amount of new tiles explored in one turn ÷ 2), bumping into a wall: -1, dying: -50, time running out: -75, attacking a zombie: +2, attacking a vampire: +3.
5. Taking exits: +150, exploring new tiles: +(amount of new tiles explored in one turn ÷ 2), bumping into a wall: -1, dying: -50, time running out: -75, attacking a zombie: +2, attacking a vampire: +3, killing an enemy: +20, using a potion: +(amount healed from potion), bumping into a wall: -1, dying: -50, time running out: -75.
6. Taking exits: +150, exploring new tiles: +(amount of new tiles explored in one turn).
7. Taking exits: +200, exploring new tiles: +(amount of new tiles explored in one turn ÷ 2).
8. Taking exits: +200, exploring new tiles: +(amount of new tiles explored in one turn).
9. Taking exits: +150, exploring new tiles: +(amount of new tiles explored in one turn ÷ 2), dying: -50.

Reward Function	Final Level Reached	Turns to Reach Final Level
1	4	85
2	4	95
3	2	20
4	1	0
5	2	30
6	3	30
7	3	45
8	3	60
9	2	20

The following table shows the results of the different reward functions:

The results here show that many actions in the game should not have rewards tied to them. Rewarding exploration and taking exits seems to be the most effective.

5.3 Performance Metrics and Logging

Agent performance is measured through TensorFlow, where at every 10,000 timesteps, five episodes are tested with the mean reward calculated. Using tensorboard, the reward is graphed on the y-axis while the timesteps during training are on the x-axis.

Regarding logging other metrics, though, it is split into two categories: formal logs and custom logs. In-game information and actions are recorded every five turns in the game, and these logs are made during final testing, not during training.

Formal logs contain the following information:

- Number of exits taken.
- Number of potions in the map.
- Number of enemies in the map.

Custom logs contain the following information:

- A map of the current game from a human player's view.
- A map of the current game from the agent's view.

5.4 Optimisations

One of the few optimisations implemented is using the sub-process environment, allowing the use of multiple cores simultaneously during training to speed up training time. Using an Nvidia graphics card's CUDA cores for training is another optimisation used to speed up training time.

Chapter 6

Evaluation and Experiments

In this chapter we will cover the experimental and evaluation work performed for this project. First we will start off by evaluating agent performance in perfect information, followed by imperfect information. In this evaluation stage we will look at the results of all of our three reinforcement learning algorithms (A2C, PPO, and, DQN) in both a random and fixed seed. Afterwards we will explore potential reasons for the performance discrepancies we see in this evaluation. Hopefully after this chapter is done, the reader will have seen evidence pointing to the conclusion of this report.

6.1 Testing Agent in Roguelike Environment

The agent gets tested using the PPO, A2C, and DQN algorithms to remove any bias in the result that may come from any algorithm. A random seed gets evaluated against a fixed seed, with a fixed seed enabling rote memorisation of optimal actions to maximise reward. In contrast, a random seed forces the agent to learn how the game operates and play accordingly to maximise reward.

Models get evaluated at every 10,000 timesteps by running ten evaluation runs of the model and taking the mean reward.

6.2 Evaluating Performance in Perfect Information

6.2.1 A2C

Random Seed

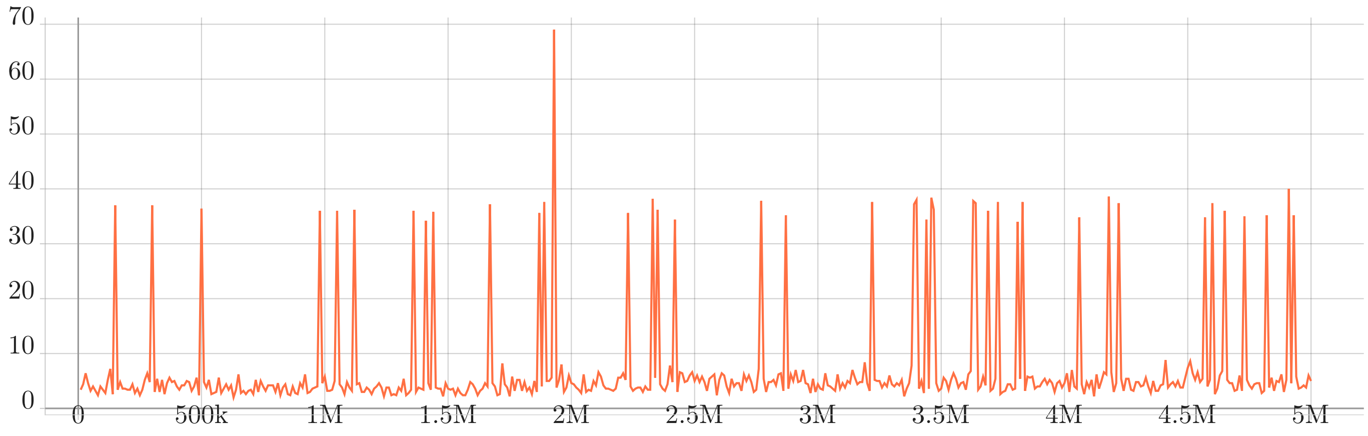


Figure 6.1: X-axis: timesteps. Y-axis: reward.

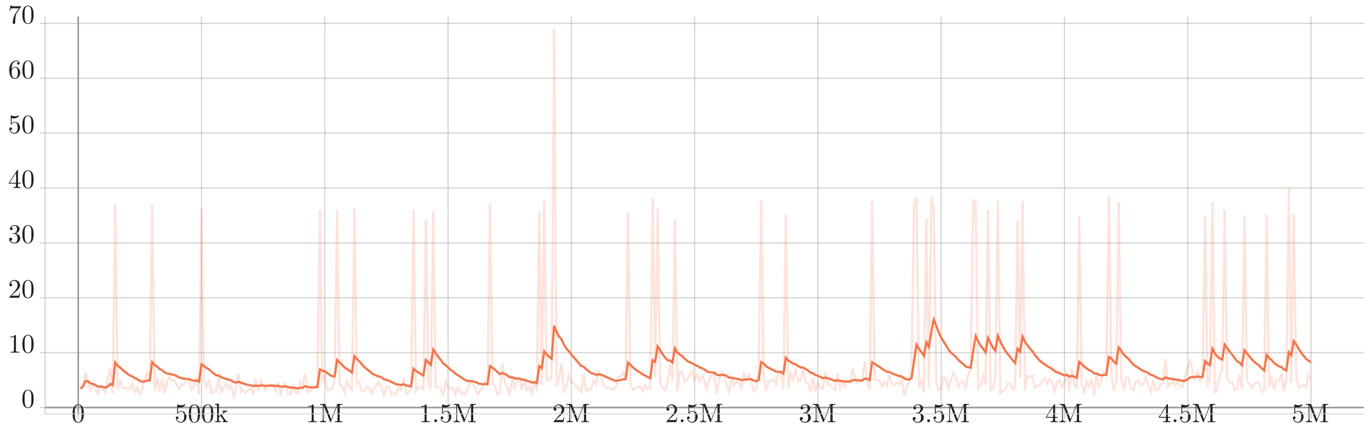


Figure 6.2: X-axis: timesteps. Y-axis: reward. (90% smoothed)

The agent performs poorly when no seed is specified, making the rooms auto-generated every time the agent tries to train, giving the agent no opportunity to memorise a map and perform the best inputs possible.

Performance peaks at a reward of 68 at the timestep $\approx$ 385,000. The lowest reward present is a reward of 3 present at multiple timesteps. The graph's distribution of highs and lows emphasises that the results were generated based on how easily the random map was. For example, if the exit was right next to the player, it could have been as simple as one input to get to the exit.

The peak performance here can be chalked up to luck rather than a reliable result that can be reproduced consistently. The results with rewards < 40 are more consistent and represent the general performance of the agent.

Specific Seed

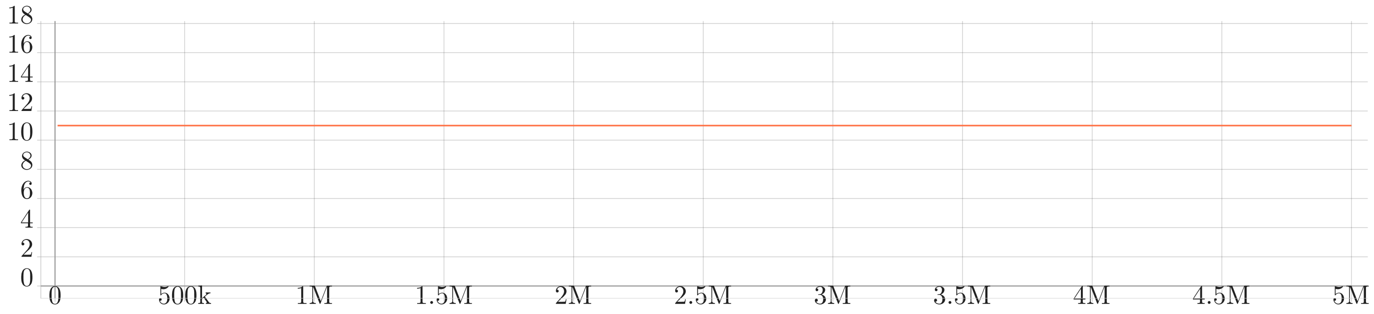


Figure 6.3: X-axis: timesteps. Y-axis: reward.

The result here is strange; although this test was repeated five times, the results did not change. A2C stagnated consistently at a reward of 11 on all timesteps. Performance is awful, and the agent shows no signs of learning or improvement throughout the training process.

6.2.2 PPO

Random Seed

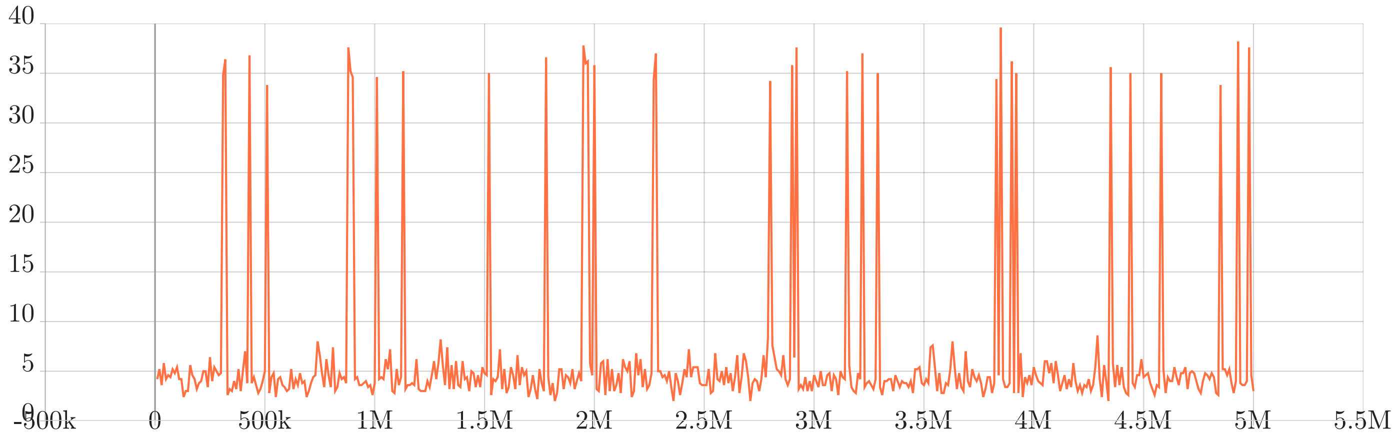


Figure 6.4: X-axis: timesteps. Y-axis: reward.

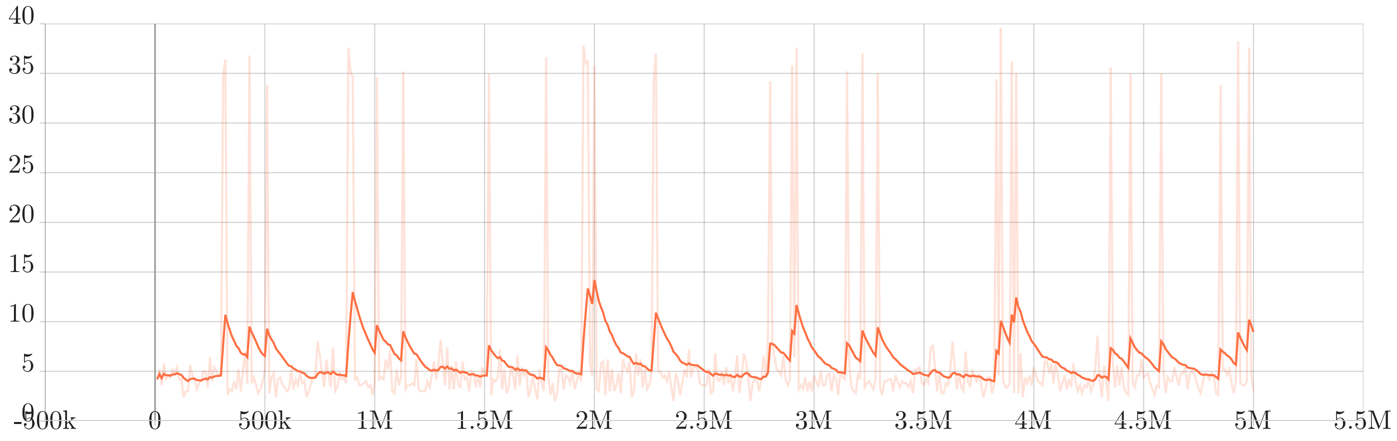


Figure 6.5: X-axis: timesteps. Y-axis: reward. (90% smoothed)

Similarly to the A2C algorithm, the agent performs poorly when there is no specified seed.

Performance peaks at a reward of 39 at the timestep ≈ 3,385,000. The lowest reward present is a reward of 3 present in multiple areas. The graph's distribution of highs and lows emphasises that the results were simply down to how easy the random map generated is. For example, if the exit was right next to the player, it could have been as simple as one input to get to the exit.

Specific Seed

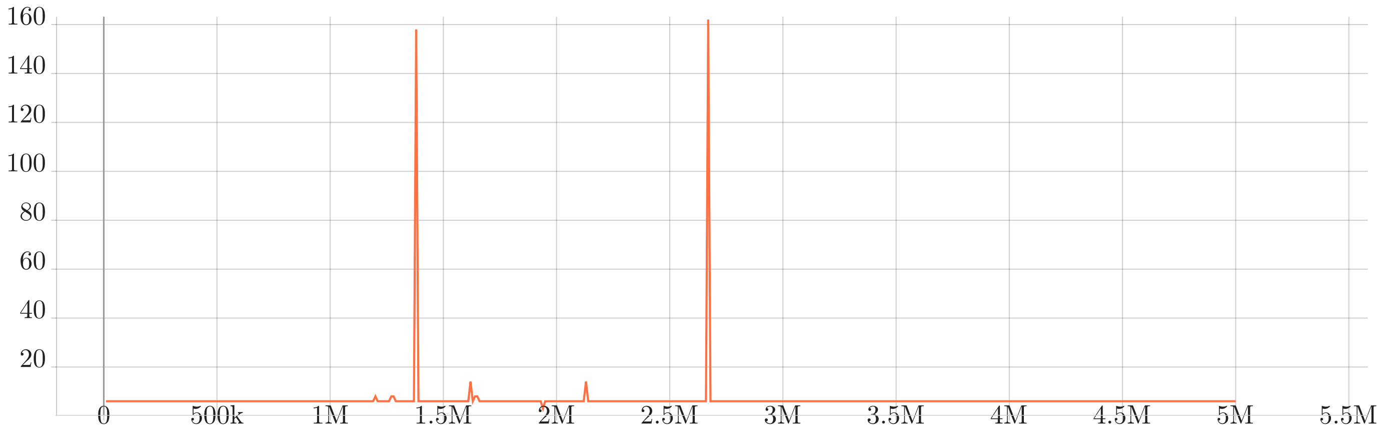


Figure 6.6: X-axis: timesteps. Y-axis: reward.

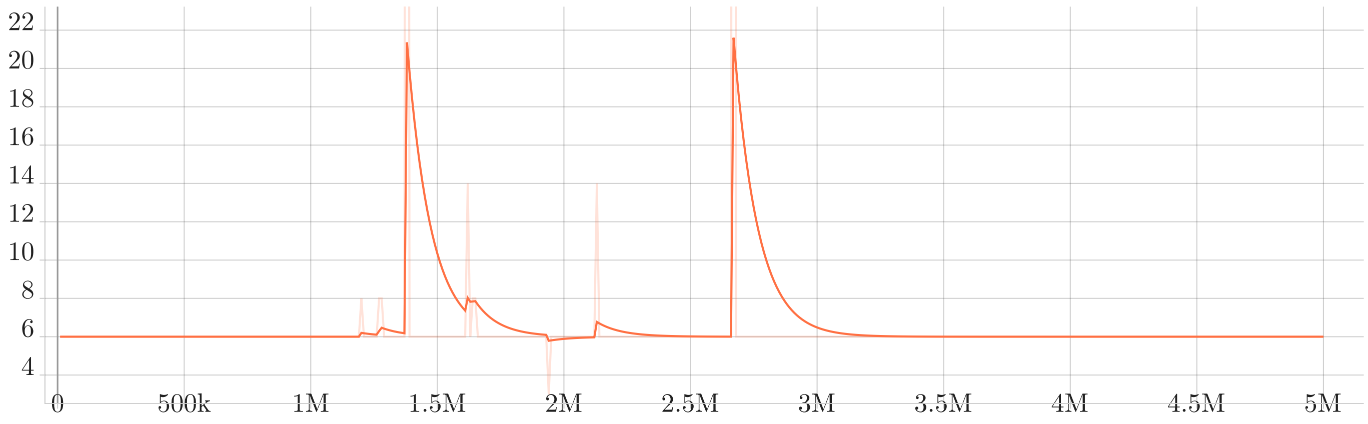


Figure 6.7: X-axis: timesteps. Y-axis: reward. (90% smoothed)

PPO underperforms at almost every timestep when given perfect information in a fixed seed, but at the two visible peaks, a reward > 150 is made. Although this could be chalked up to luck, it most likely is not since a score of > 150 means that the agent took one exit over five evaluation trials.

Still, the performance overall is abysmal compared to the random seed training session, which is interesting. The max reward is 161 at timestep $\approx$ 2.650,000. The minimum reward is 9, which can be seen at almost all timesteps. This performance is indicative of an agent unable to get past even one dungeon during the majority of training.

6.2.3 DQN

Random Seed

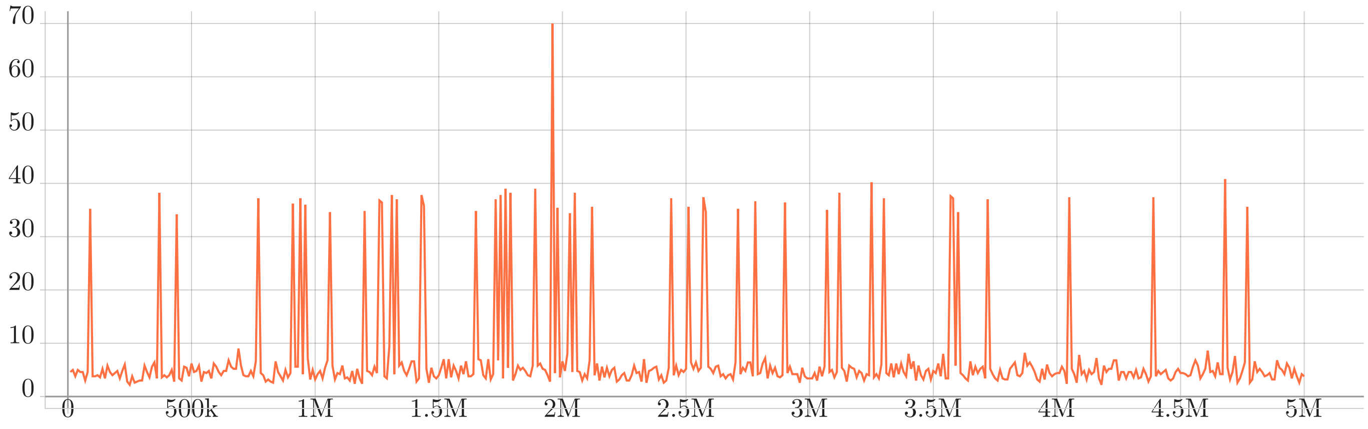


Figure 6.8: X-axis: timesteps. Y-axis: reward.

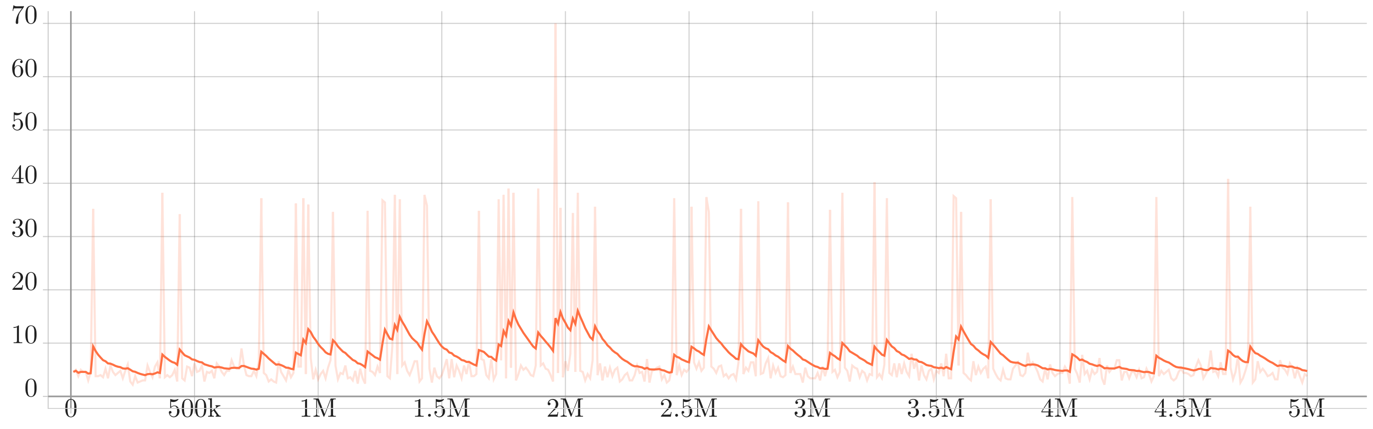


Figure 6.9: X-axis: timesteps. Y-axis: reward. (90% smoothed)

Similarly to PPO and A2C, the agent performs poorly when no seed is specified. DQN also provides similar performance to both algorithms in this situation.

Performance peaks at a reward of 69 at the timestep ≈ 1,900,000. The lowest reward present is a reward of 3 present at multiple timesteps. The graph's distribution of highs and lows emphasises that the results were simply down to how easy each randomly map generated was. Considering the number of times the reward exceeded 30, the agent can generally take at least one exit.

Specific Seed

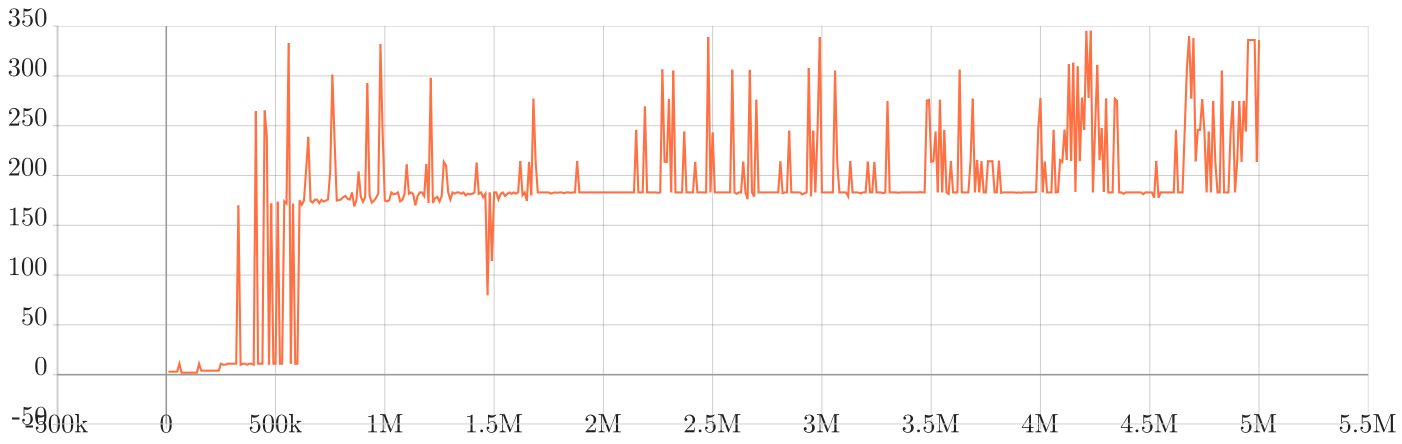


Figure 6.10: X-axis: timesteps. Y-axis: reward.

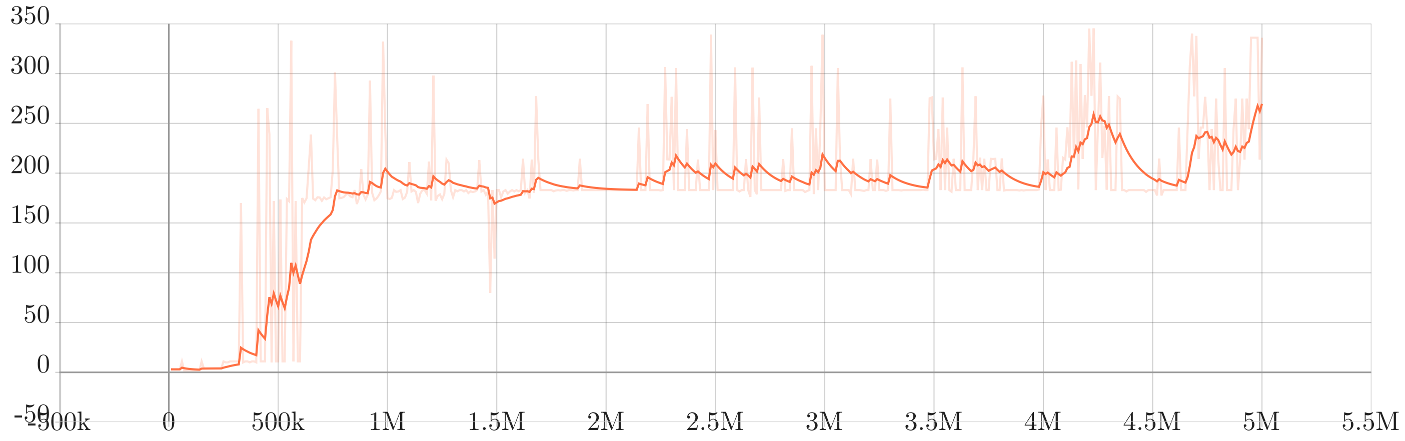


Figure 6.11: X-axis: timesteps. Y-axis: reward. (90% smoothed)

DQN provides consistent performance when given a fixed seed in a perfect information scenario. DQN seems to exceed the A2C and PPO algorithms in a situation of perfect information given a fixed seed with relative ease.

Performance reliably shows a reward > 175 at all timesteps past $\approx$ 750,000. A maximum reward of 333 can often occur in the figure, such as at timestep $\approx$ 550,000. A minimum reward of 3 is initially present from timestep 0 to $\approx$ 200,000.

With the reward never dipping down to below 175 past timestep $\approx$ 1,500,000, we can easily show that the DQN agent consistently performs well.

6.3 Evaluating Performance in Imperfect Information

6.3.1 A2C

Random Seed

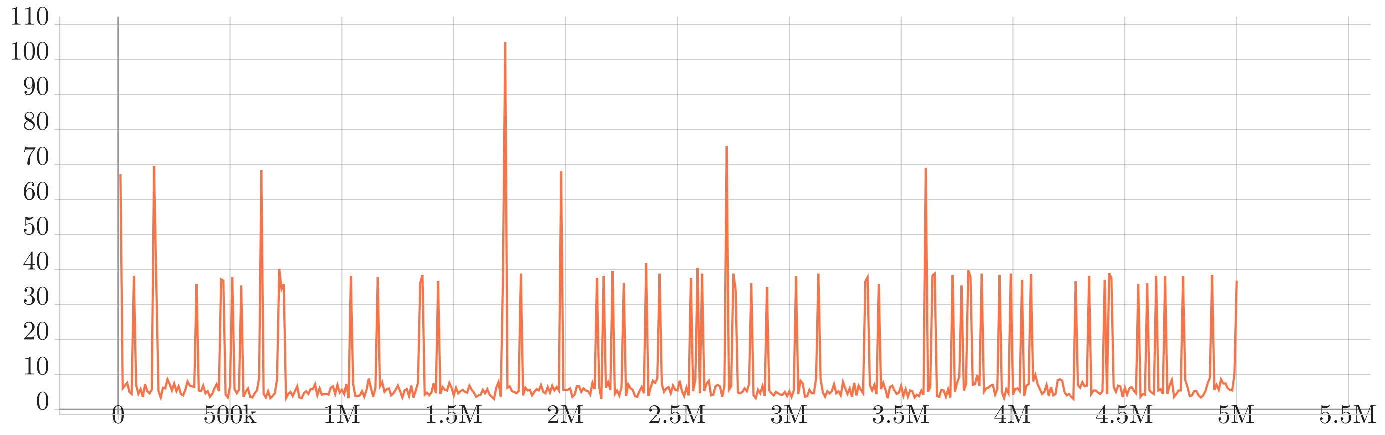


Figure 6.12: X-axis: timesteps. Y-axis: reward.

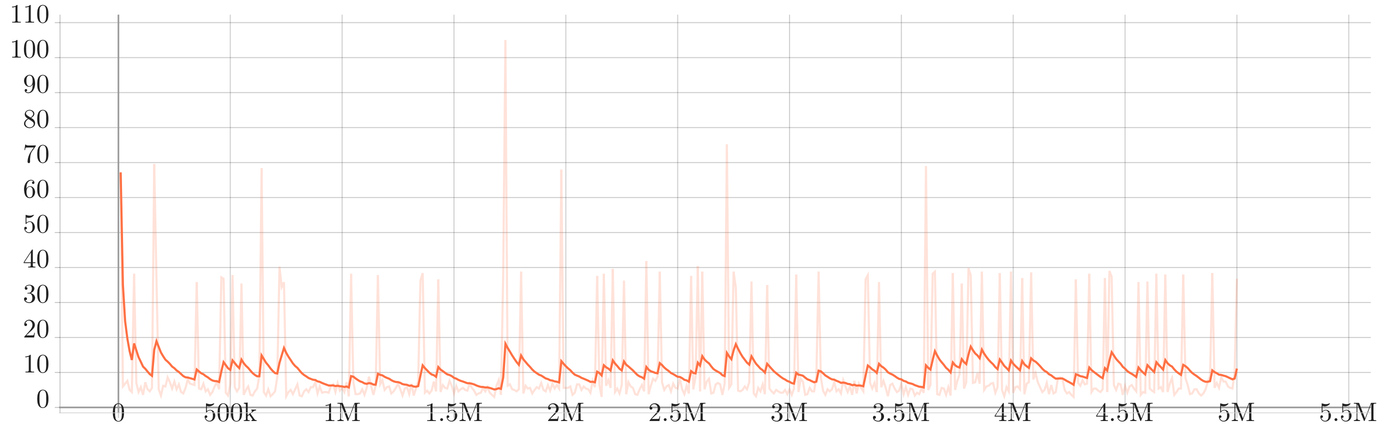


Figure 6.13: X-axis: timesteps. Y-axis: reward. (90% smoothed)

Similarly to the perfect information random seed A2C result, the agent performs poorly. Good results are inconsistent and highly spread out, with the 90% smoothed graph showing that the reward on average hovers at ≈ 10.

The maximum reward achieved is 104 at timestep ≈ 1,750,000, which can be mostly chalked up as a fluke considering the inconsistency of the results. The minimum reward achieved is three multiple times throughout training.

Specific Seed

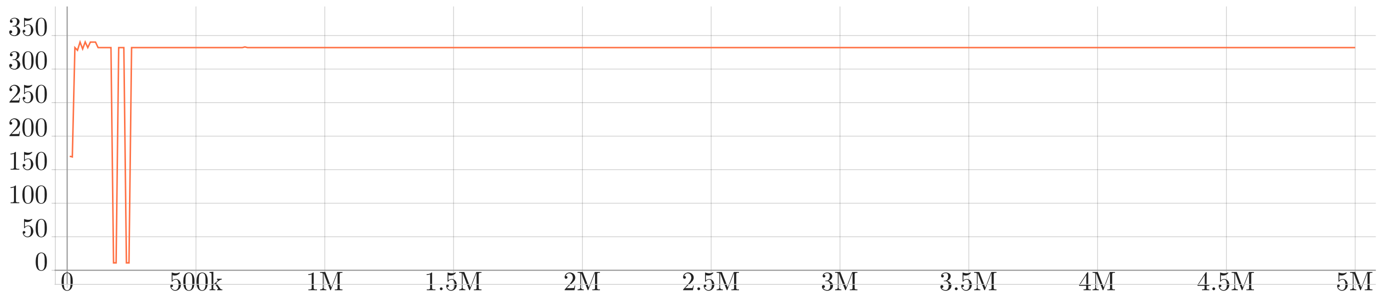


Figure 6.14: X-axis: timesteps. Y-axis: reward.

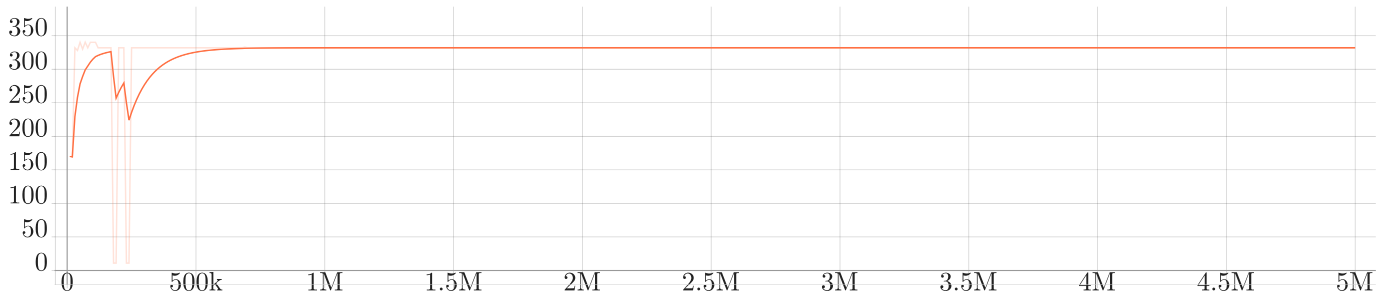


Figure 6.15: X-axis: timesteps. Y-axis: reward. (90% smoothed)

The A2C algorithm reached its peak performance quickly and never faltered, which is impressive. When given imperfect information in a fixed seed, the agent reached a maximum reward of 343 at timestep ≈ 90,000, followed by a quick dip into the minimum reward of 11 at timestep ≈ 200,000. After the two short dips into the minimum reward reached, the agent never dipped below a reward of 333.

The algorithm here showed remarkable consistency, rarely deviating and experimenting with situations that could reduce the total reward.

6.3.2 PPO

Random Seed

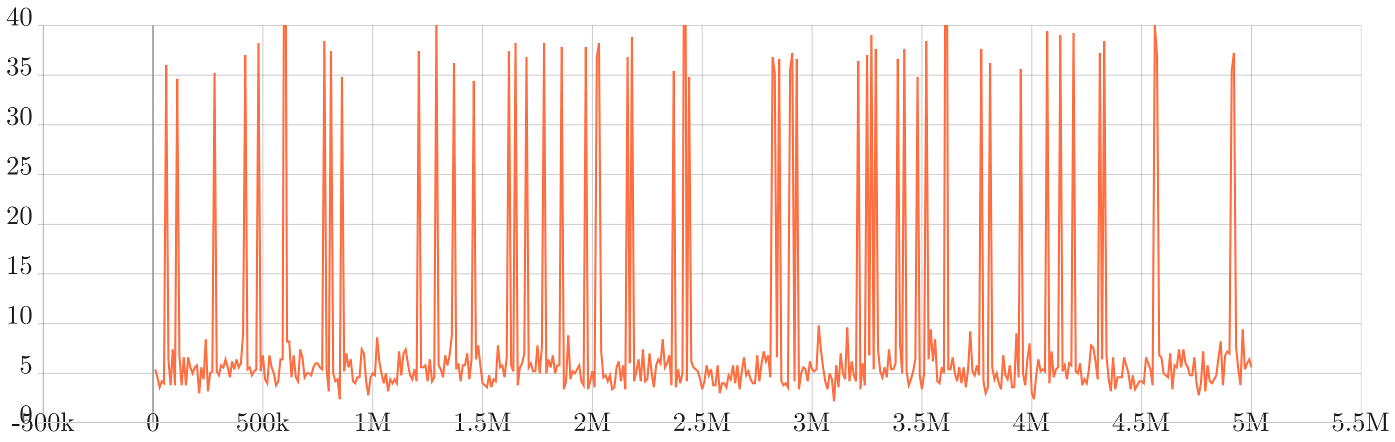


Figure 6.16: X-axis: timesteps. Y-axis: reward.

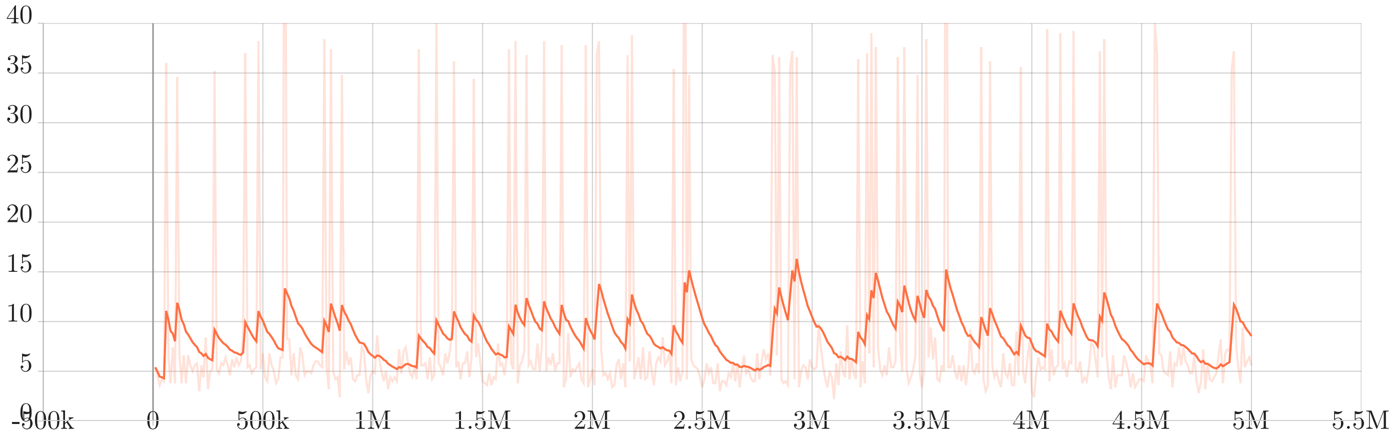


Figure 6.17: X-axis: timesteps. Y-axis: reward. (90% smoothed)

PPO performs poorly when given a random set of maps to train and test on since the agent cannot see the entire map with imperfect information, presenting even further difficulty.

The max reward present was seen at ≈ 520,000 timesteps, with a reward of 42. The minimum reward present was seen at multiple timesteps with a reward of 3.

Specific Seed

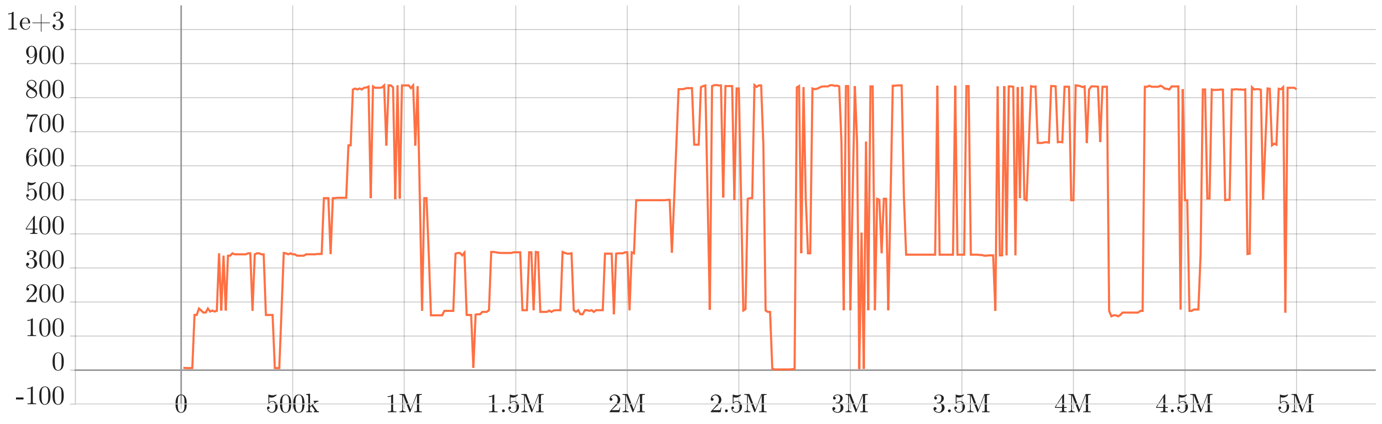


Figure 6.18: X-axis: timesteps. Y-axis: reward.

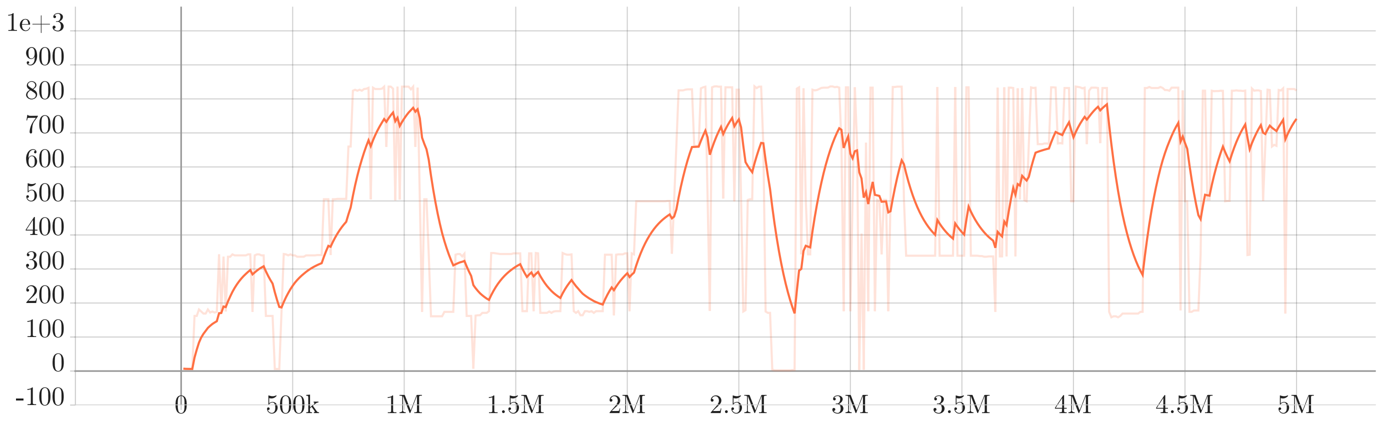


Figure 6.19: X-axis: timesteps. Y-axis: reward. (90% smoothed)

PPO performs exceptionally with a fixed seed. Even though imperfect information is present, the agent still manages to maintain a reward > 600 at timesteps t such that $750,000 < t < 1,250,000$, $2,250,000 < t < 2,600,000$, $2,800,000 < t < 3,050,000$, $3,750,000 < t < 4,200,000$, $4,400,000 < t < 4,510,000$, and $4,650,000 < t < 5,000,000$ approximately.

The performance is varied across the board, making it inconsistent. We can see that the model's reward can reach a maximum of 836 and a minimum of 6 at multiple timesteps throughout training. Therefore, one can generate a model that can maintain a reward above 700 if training is stopped at the appropriate timestep.

6.3.3 DQN

Random Seed

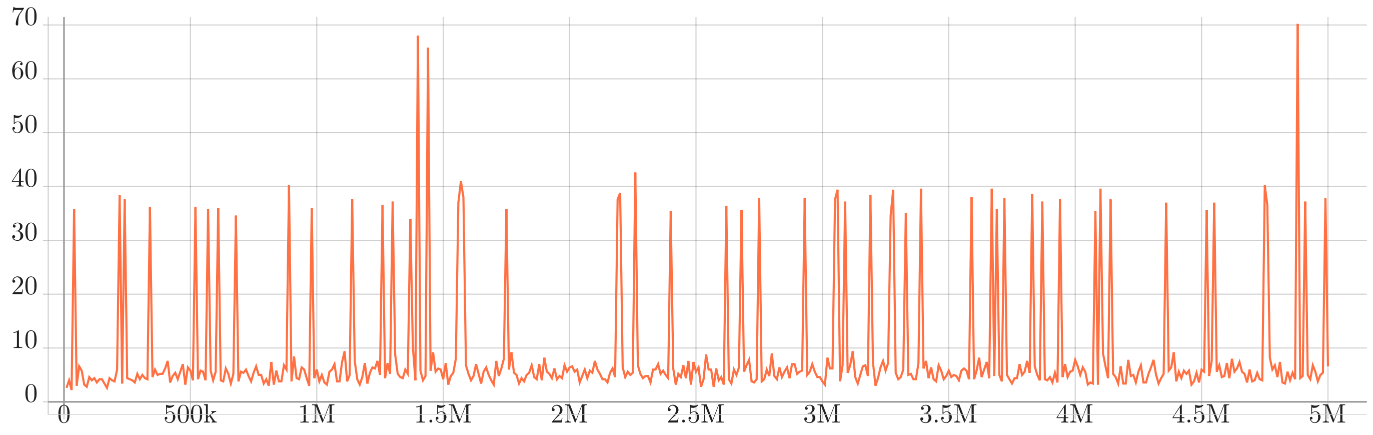


Figure 6.20: X-axis: timesteps. Y-axis: reward.

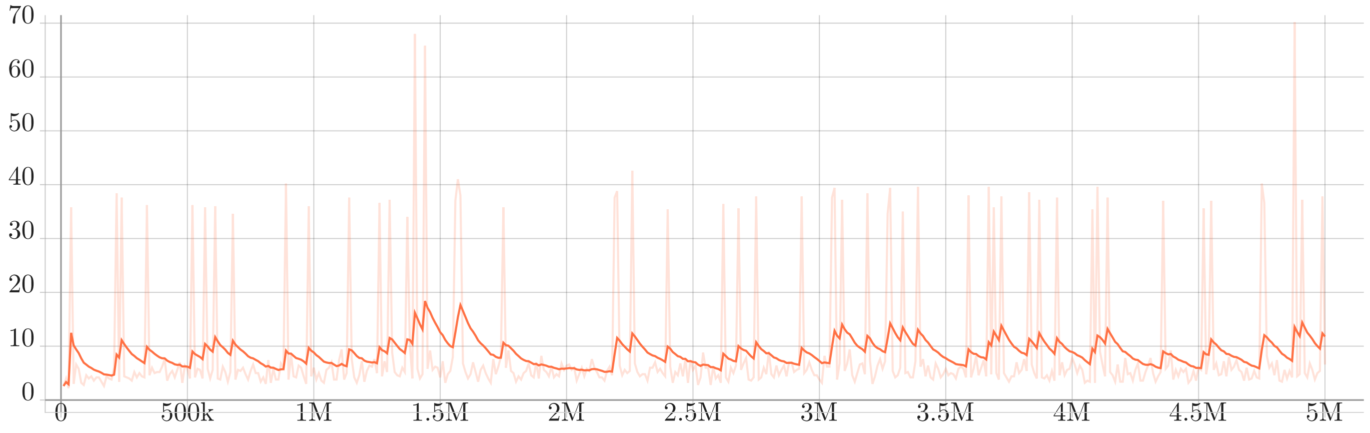


Figure 6.21: X-axis: timesteps. Y-axis: reward. (90% smoothed)

DQN performs similarly to PPO in a situation of imperfect information with a random seed. In a random seed, when given imperfect information, DQN performs poorly. Performance is inconsistent and can be seen to fluctuate dramatically, reaching a maximum reward of 70 at timestep ≈ 4,750,000 and a minimum of 6 at multiple timesteps throughout training.

Specific Seed

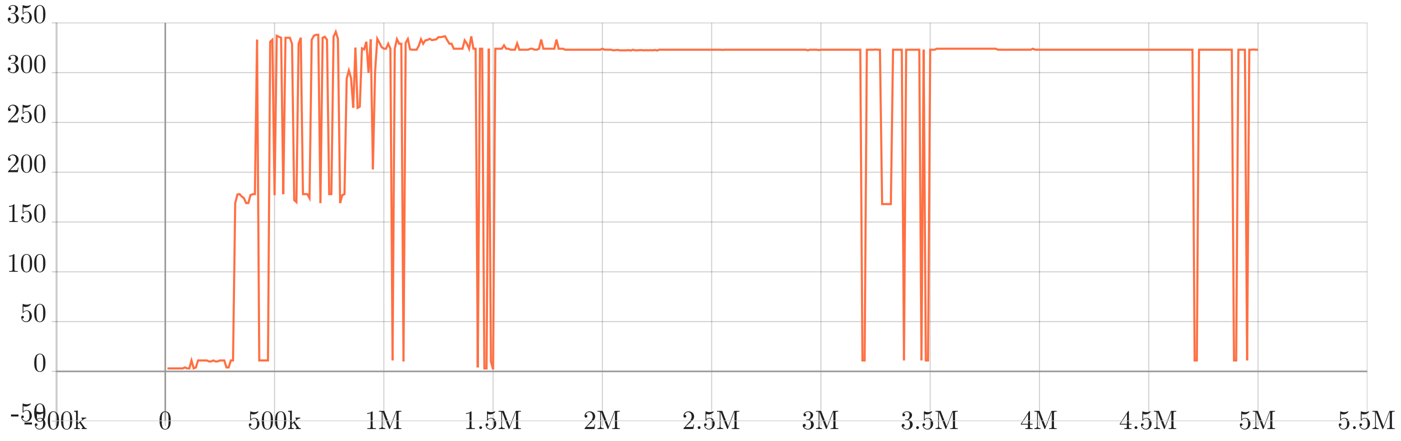


Figure 6.22: X-axis: timesteps. Y-axis: reward.

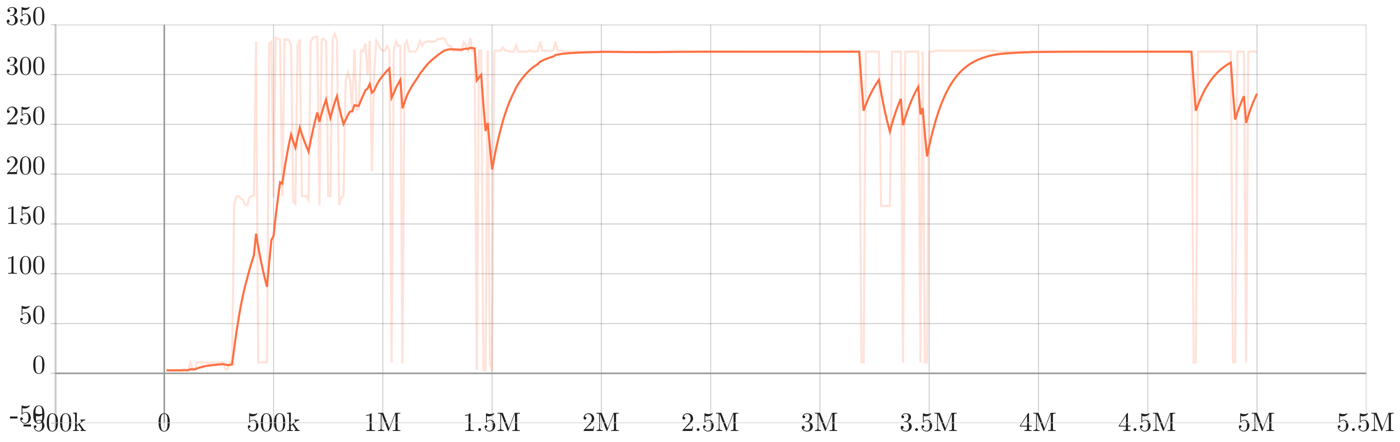


Figure 6.23: X-axis: timesteps. Y-axis: reward. (90% smoothed)

When given a fixed seed with imperfect information, DQN begins to perform similarly to the A2C algorithm when it is presented with the same situation. A much higher degree of consistency is reached when comparing DQN to PPO, but not the same level of consistency as the A2C algorithm presented.

A maximum reward of 343 is reached at timestep ≈ 750,000, and a minimum reward of 6 is reached at the beginning of training. The algorithm proceeds to experiment with actions that could raise its reward but, in the process, loses out on quite a bit of reward, which causes the dips visible in the graph. However, the DQN algorithm does maintain an average score of 333 at most timesteps in the graph.

6.4 Perfect Information Versus Imperfect Information

An astounding result presents itself from the graphs above, PPO and A2C both perform significantly better ($> 100\%$) when given imperfect information. In contrast, when given imperfect information, DQN performs $\approx 50\%$ better across all timesteps. Many factors could have led to this. It would be prudent to discuss the factors with supporting evidence before going into the theoretically reasonable factors and ending with the speculative ones.

6.4.1 Map Labelling

Since multi-layer perceptrons are used instead of convolutional neural networks, the labels chosen for characters on the map bear a noticeable effect on agent performance. In fact one can consider it a hyperparameter to be adjusted for optimal agent performance.

The lackluster performance of the perfect information agents compared to the imperfect agents can be attributed to a need for a different way to label the map for perfect information agents. When using the same map labelling used in imperfect information for perfect information the agent performs noticeably worse as can be seen here:

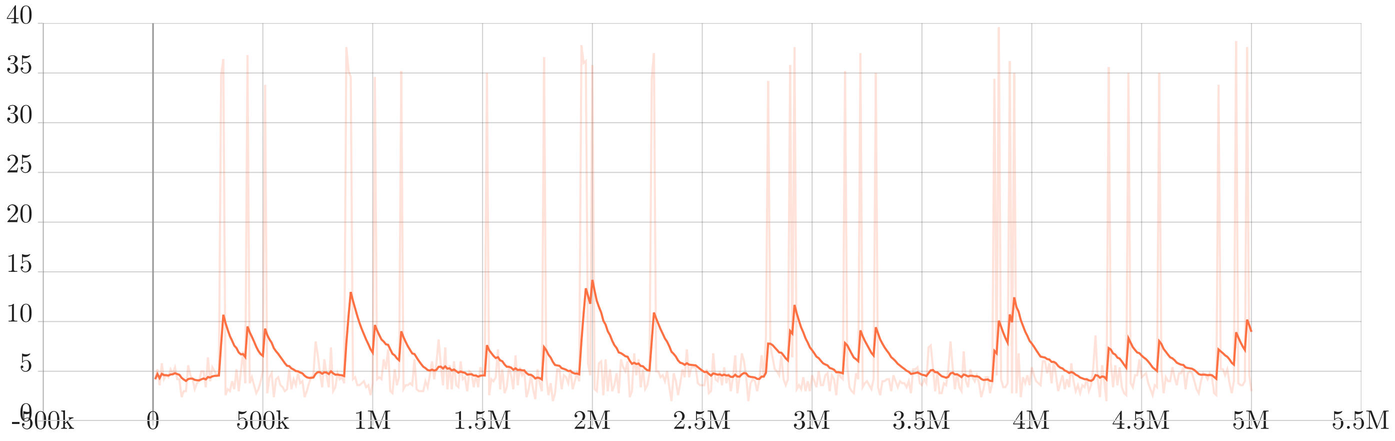


Figure 6.24: X-axis: timesteps. Y-axis: reward. (PPO algorithm)

However, a huge leap in performance is made when a different map labelling is used as can be seen here:

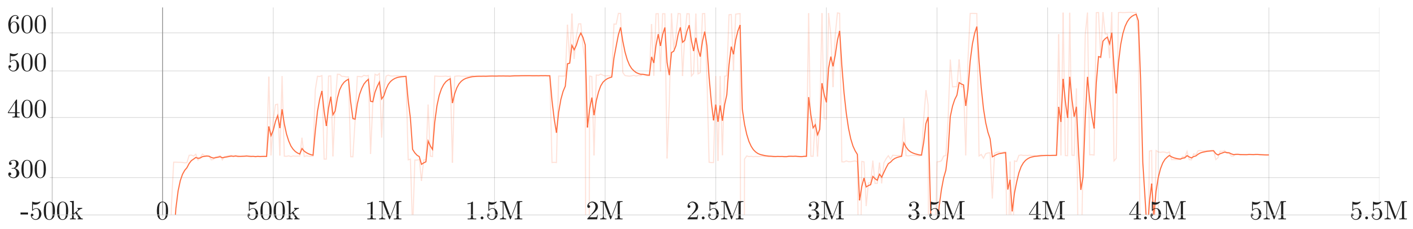


Figure 6.25: X-axis: timesteps. Y-axis: reward. (PPO algorithm)

The labelling used here was:

What it is	Numerical Label	ASCII Character
Wall	0	#
Vampire	-3	v
Zombie	-2	z
Corpse	-1	q
Player	1	@
Floor	2	.
Potion	3	+
Exit	4	>

Table 6.1: Labels used for figure 6.25

By using these labels, hidden tiles were removed, making the state space lower, which was the main contributor to the performance uplift seen in figure 6.25. However, upon further experimentation, it was discovered that these labels depended on the reinforcement learning algorithm and whether perfect or imperfect information was used.

Therefore one can conclude that the labelling used for the map may have played a significant part in skewing the results in favour of the imperfect information agent. To thoroughly verify this result, however, we can explore the effect of using the RGB values of an image of the current state of the game to circumvent the need to label.

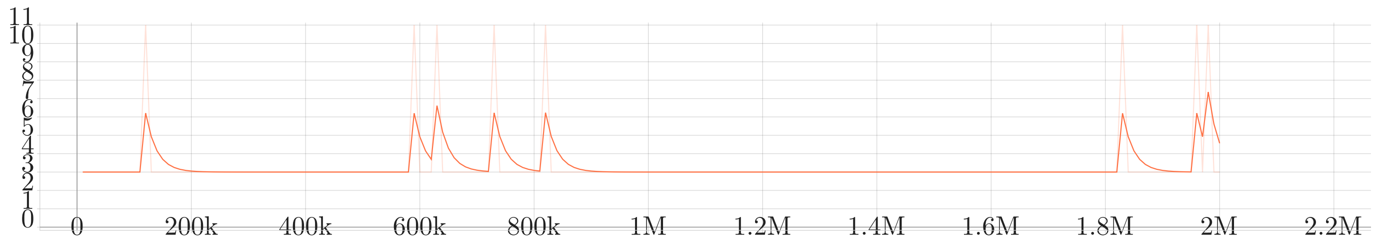


Figure 6.26: X-axis: timesteps. Y-axis: reward. PPO algorithm using our multi-layer perceptron in perfect information.

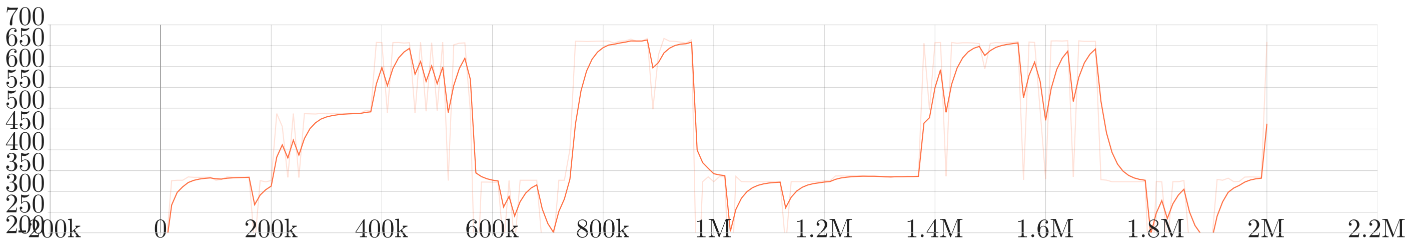


Figure 6.27: X-axis: timesteps. Y-axis: reward. PPO algorithm using our multi-layer perceptron in imperfect information.

The agent yet again performs significantly better on imperfect information. Therefore we can discern that the agent has an easier time generalising their situation in imperfect information. Even though labelling the map can help, it seems there are fundamental issues with using our multi-layer perceptron in perfect information that makes performance significantly better in imperfect information.

However, we can attempt to control the neural network architecture and see the effect of using a convolutional neural network rather than a multi-layer perceptron. Since convolutional neural networks are known to be much better suited to handle RGB values from images[24][17].

6.4.2 Effect of Neural Network Architecture

In our current use case, multi-layer perceptrons required some form of integer labelling to represent the agent's map better; however, we can forego labelling if we instead use a convolutional neural network. The reasoning is that, in our case, convolutional neural networks can take a three-dimensional array of RGB values rather than a two-dimensional array of integer labels.

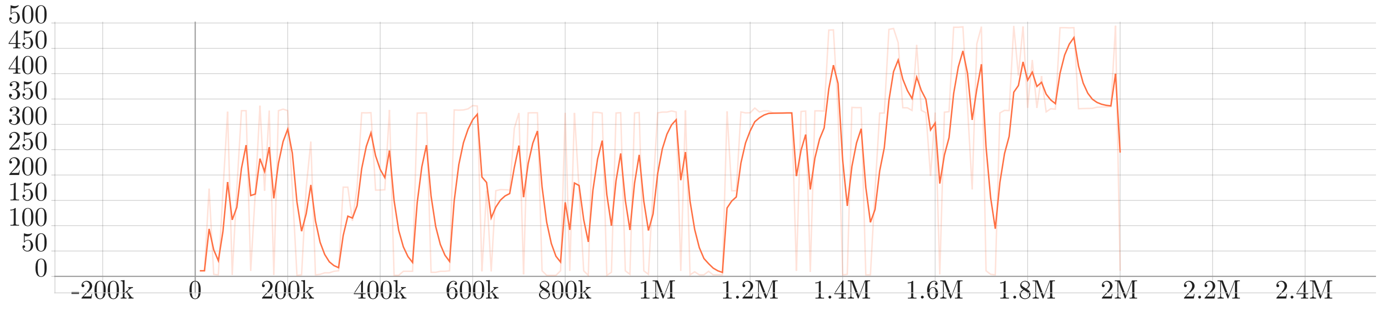


Figure 6.28: X-axis: timesteps. Y-axis: reward. PPO algorithm using a convolutional neural network in perfect information.

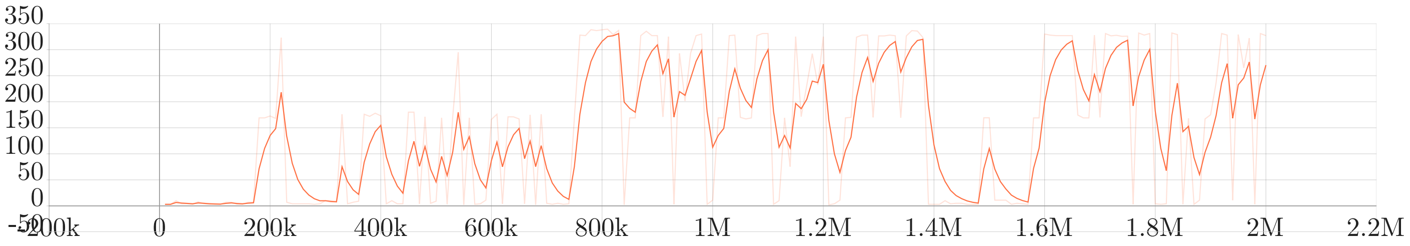


Figure 6.29: X-axis: timesteps. Y-axis: reward. PPO algorithm using a convolutional neural network in imperfect information.

Interestingly we finally see perfect information superseding imperfect information, the expected result. The agent here can generalise much better in perfect information when using a convolutional neural network than when using a multi-layer perceptron. However, the running time of extracting the image of the game's current state to get the RGB values is significantly longer when compared to our initial strategy of labelling the map (around 10x longer).

6.4.3 A2C

As discussed in the reinforcement learning subsection of the technical information section, A2C has an actor, a model, that the state as input and outputs an action for the agent. In contrast, the critic, also a model, takes the state as input and outputs a value for that state. The advantage, which represents how well your agent is doing, is the difference between the value given by the critic and some baseline expectation for the reward[].

With that in mind, it becomes easier to understand why the agent performs significantly better on a fixed seed than on a random one. The critic and the actor can adjust their value and action much better when the environment is fixed. Therefore the agent performs significantly better when put into a fixed seed.

In regards to why A2C presents much better when it comes to imperfect information, a running hypothesis is that the reduction of data points gives the agent a better ability at evaluating the potential reward of each state.

6.4.4 PPO

PPO, as described in the reinforcement learning subsection of the technical information section, ensures that the updated policy is not too different from the old policy[42] which ensures low variance. This may pose a difficulty for PPO to explore potentially new ideas and verify itself as the player.

6.4.5 DQN

DQN's focus on filling a Q-table with the best possible action for each state may have presented a large benefit in this situation, explaining the fact that it still performs reasonably well in perfect information. The reason it performs better can be explained by the same reason A2C performs better, the reduction of data points gives the agent a better ability at evaluating the potential reward of each state.

6.4.6 Speculative: The Model Does Not Recognise Itself

Although a speculative idea that should be taken with a grain of salt, potentially the model has a much more difficult time recognising who the player character its controlling is. With so many moving parts and so many different things displayed in perfect information, the model potentially cannot differentiate between itself and its surroundings. The fog of war that clears up the map as the agent explores may have had a potential side benefit of making it more obvious what the agent is controlling.

Chapter 7

Conclusion and Discussion

We can finally begin concluding our report and discussing what has been accomplished. In this chapter, we will firstly reflect on the aims and the objectives initially set during our introduction to seeing how much of our initial goals were achieved. Afterwards, we will explore some complications present throughout the project in hopes that the complications present throughout are solved when this project is expanded on or replicated.

We then explore potential future improvements that could have been done to this project, again in hopes that these improvements will be performed in this project is expanded. Finally, we summarise and discuss the findings in this report to give our reader a general sense of the results in this report.

7.1 Reflection on Aims and Objectives

Reflecting on the aims initially outlined we can see that our primary aim, investigating the effects of perfect information and imperfect information in a controlled environment, was accomplished due to the following reasons:

- The same base game was used for both perfect information and imperfect information, rather than using chess for perfect information and poker for imperfect information. Therefore we successfully control our agent's learning environment.
- Three different reinforcement learning methods were used to prevent an erroneous result that may have been due to a specific reinforcement learning algorithm's implementation. Therefore we successfully control another aspect of the testing.
- We analysed the effects of different neural network architectures, preventing an erroneous result that could have occurred due to a network's architecture. Therefore we successfully control our testing further.

With these factors in mind, it is easy to see that most large aspects of this project were controlled, making it easier to specifically analyse the effects of switching between perfect and imperfect information.

Furthermore, our second aim, creating an easily extendable custom environment that allows for further exploration of the subject matter explored in this report and the ability to reproduce the results seen throughout, was performed successfully. A full roguelike game was created that allows for the following:

- Easy addition of items.
- Easy addition of enemies.
- Ability to control difficulty scaling.

- Easy addition of different room types.
- Ability to control the player's field of vision.
- Ability to control monster and potion spawns.
- Ability to control dungeon generation in terms of map size, whether a fixed seed is used, and how rooms are tunneled between each other.

On top of all this, running the code is easy with a simple menu interface asking the user whether they want to test a reinforcement learning agent or run the game normally. Therefore the base game is relatively easy to extend with additional features and adjust to match the difficulty settings sought after by a researcher.

The objectives of this project were completed in full, in particular:

- A roguelike was created full, bearing all the attributes we initially wanted.
- An environment was successfully created wherein the agent could interact with the game seamlessly in multiple ways for the agent to act on and perceive the game's current state.
- The three reinforcement learning algorithms: PPO, DQN, and A2C, were tested separately in perfect and imperfect information to validate that results were not just due to a particular algorithm.
- Empirical results were obtained from graphs showing the agents' performance.
- Potential reasons for the discrepancy of performance in perfect information and imperfect information were analysed and discussed further.

Therefore all initially sought after objectives were completed successfully.

7.2 Complications and Future Improvements

The training was capped at five million timesteps, potentially dramatically hindering performance. If this study is reproduced, ideally, the training time should be scaled up significantly as we may not have given enough time for the agent to generalize perfect information.

Using convolutional neural networks more extensively and studying their effects would also significantly improve validating the results and testing them to the fullest. The only issue is the long running time required to train a convolutional neural network instead of a multi-layer perceptron that relies on map labelling.

7.3 Conclusion

We noticed that when using multi-layer perceptrons, the model struggles to generalize when using perfect information but has a much easier time in imperfect information. When we use convolutional neural networks instead, the agent performs as expected, performing better in perfect information than in imperfect information.

The overall reasoning for this performance discrepancy can be explained due to the multi-layer perceptron's difficulty in coping with an extensive data set for processing (which was evident when we fed RGB values into the multi-layer perceptron rather than our significantly reduced size map labelling technique). However, when we reduce the data set to its minimum components by using map labelling and imperfect information, we find that a multi-layer perceptron exceeds the performance of a convolutional neural network with the complete components provided.

Therefore we propose treating the agent's access to information as a hyperparameter to be adjusted for maximal performance. How the agent perceives the current state of the environment dramatically affects performance. Simply giving the agent more information to process than it requires is a waste of computation time and leads to an overall loss in performance. We also propose experimenting with a neural network's architecture in tandem with different interpretations of an environment's current state since different neural networks give wildly different performances depending on the state provided to them.

Many people take for granted that using perfect information will constantly improve performance. However, this study confirms that certain pieces of information we take for granted must be tested to confirm and validate the conclusions we deem apparent. This exploration has revealed that perfect information is not always universally better, as previously thought and taught.

Bibliography

[1] Openai/baselines: Openai baselines: High-quality implementations of reinforcement learning algorithms.

[2] Openai, about us. https://openai.com/, Jun 2021.

[3] Andrea Asperti, Carlo De Pieri, and Gianmaria Pedrini. Rogueinabox: an environment for roguelike learning. *International Journal of Computers*, 2, 2017.

[4] Sander Bakkes, Pieter Spronck, Jaap Van Den Herik, and Philip Kerbusch. Predicting success in an imperfect-information game. In *Proceedings of the Computer Games Workshop 2007*, pages 219–230, 2007.

[5] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[6] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[7] Michael A Bozarth. Pleasure systems in the brain. *Pleasure: The politics and the reality*, pages 5–14, 1994.

[8] Ivan Bratko. Alphazero–what's missing? *Informatica*, 42(1), 2018.

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[10] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. *Advances in Neural Information Processing Systems*, 33:17057–17069, 2020.

[11] Marisa P. de Brito and Erwin A. van der Laan. Inventory control with product returns: The impact of imperfect information. *European Journal of Operational Research*, 194(1):85–101, 2009.

[12] Moritz Fleischmann, Jacqueline M. Bloemhof-Ruwaard, Rommert Dekker, Erwin van der Laan, Jo A.E.E. van Nunen, and Luk N. Van Wassenhove. Quantitative models for reverse logistics: A review. *European Journal of Operational Research*, 103(1):1–17, 1997.

[13] Eugene C Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

[14] Thomas Gabor, Leo Sünkel, Fabian Ritz, Thomy Phan, Lenz Belzner, Christoph Roch, Sebastian Feld, and Claudia Linnhoff-Popien. The holy grail of quantum artificial intelligence: major challenges in accelerating the machine learning pipeline. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 456–461, 2020.

[15] John Harris. *Exploring Roguelike Games.* CRC Press, 2020.

[16] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

[17] Nadia Jmour, Sehla Zayen, and Afef Abdelkrim. Convolutional neural networks for image classification. In *2018 international conference on advanced systems and electric technologies (IC_ ASET)*, pages 397–402. IEEE, 2018.

[18] Mark R Johnson. The use of ascii graphics in roguelikes: Aesthetic nostalgia and semiotic difference. *Games and Culture*, 12(2):115–135, 2017.

[19] Yuji Kanagawa and Tomoyuki Kaneko. Rogue-gym: A new challenge for generalization in reinforcement learning. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[20] Priscilla Kehoe and Elliott M Blass. Central nervous system mediation of positive and negative reinforcement in neonatal albino rats. *Developmental Brain Research*, 27(1):69–75, 1986.

[21] Nikhil Ketkar. Stochastic gradient descent. In *Deep learning with Python*, pages 113–132. Springer, 2017.

[22] Phil Kim. Convolutional neural network. In *MATLAB deep learning*, pages 121–147. Springer, 2017.

[23] Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 7671–7684. Curran Associates, Inc., 2020.

[24] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[25] Jae Won Lee. Stock price prediction using reinforcement learning. In *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No. 01TH8570)*, volume 1, pages 690–695. IEEE, 2001.

[26] Grace W Lindsay. Convolutional neural networks as a model of the visual system: Past, present, and future. *Journal of cognitive neuroscience*, 33(10):2017–2031, 2021.

[27] Francisco S Melo. Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep*, pages 1–4, 2001.

[28] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[31] Jan Mycielski. Games with perfect information. *Handbook of game theory with economic applications*, 1:41–70, 1992.

[32] Costas Neocleous and Christos Schizas. Artificial neural network learning: A comparative review. In *Hellenic Conference on Artificial Intelligence*, pages 300–313. Springer, 2002.

[33] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[34] Thierry Paunin. Garry kasparov interview. *Jeux amp; Strategie*, (55), Feb 1989.

[35] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 2021.

[36] Hassan Ramchoun, Youssef Ghanou, Mohamed Ettaouil, and Mohammed Amine Janati Idrissi. Multilayer perceptron: Architecture optimization and training. 2016.

[37] Martin Riedmiller and AM Lernen. Multi layer perceptron. *Machine Learning Lab Special Lecture, University of Freiburg*, pages 7–24, 2014.

[38] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[39] Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Küttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. *arXiv preprint arXiv:2109.13202*, 2021.

[40] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[41] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[42] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[43] Alessandro Sestini, Alexander Kuhnle, and Andrew D Bagdanov. Deepcrawl: Deep reinforcement learning for turn-based strategy games. *arXiv preprint arXiv:2012.01914*, 2020.

[44] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[45] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[46] Halit Bener Suay and Sonia Chernova. Effect of human guidance and state space size on interactive reinforcement learning. In *2011 Ro-Man*, pages 1–6. IEEE, 2011.

[47] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[48] Lyn Carey Thomas. *Games, theory and applications.* Courier Corporation, 2012.

[49] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *International Conference on Computers and Games*, pages 84–99. Springer, 2006.

[50] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[51] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[52] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9(4):611–629, 2018.

[53] Bayya Yegnanarayana. *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[54] Neil Y Yen, Jia-Wei Chang, Jia-Yi Liao, and You-Ming Yong. Analysis of interpolation algorithms for the missing values in iot time series: a case of air quality in taiwan. *The Journal of Supercomputing*, 76(8):6475–6500, 2020.

Appendix A

Environment Implementation

```python
# Gym Environment

from gym.spaces import Discrete, Box
from gym import Env

from stable_baselines3.common.evaluation import evaluate_policy

from typing import List, Tuple

import random
from math import ceil
import numpy as np
import copy
import multiprocessing as mp
from multiprocessing import Pool

from src.engine import Engine
from src.utilities.actions import Take, Attack
from src.entities.entity_factory import player
from src.utilities.pathfind import get_path_to

# Translates discrete action to up, down, left, and right.
action_translator = {
    0: (1,0),
    1: (-1,0),
    2: (0,1),
    3: (0,-1)
}

# Translates level from characters to numbers for agent.
level_translator = {
    "#": -4,
    ".": 2,
    ">": 4,
    "@": 1,
    "+": 3,
    "z": -2,
    "v": -3,
    "q": -1,
    " ": 0
}
```

```python

class RLEnv(Env):

    def __init__(self, seed: int = 0, fixed_seed: bool = False, perfect_info: bool = True) -> None:
        """
        RogueLike Reinforcement Learning Environment.

        Args:
            seed (int, optional): Seed if desired. Defaults to 0.
            fixed_seed (bool, optional): Write as true if you want to run with seed. Defaults to False.
        """
        # Necessary for game functionality:
        self.player = copy.deepcopy(player)
        self.fixed_seed = fixed_seed
        self.seed_num = seed
        self.perfect_info = perfect_info
        if self.seed_num == 0:
            self.seed_num = random.randint(1, 10000)
        self.engine = Engine(self.player, self.fixed_seed, self.seed_num)

        # Necessary for environment functionality:
        self.action_space = Discrete(4)
        self.observation_space = Box(low=-4, high=4, shape=((self.engine.level.height+1),self.engine.level.width), dtype=np.int_)
        self.state, _ = self.translate_map()

        # Necessary for reward calculation:
        self.time_spent = 0
        self.path_reward = 0
        self.last_hp = copy.deepcopy(self.engine.player.hp)
        self.explored_reward = 0
        self.last_explored = 0

        # Necessary for formal logs:
        self.exits_taken = 0
        self.enemies_killed = 0
        self.potions_taken = 0

    def set_seed(self, seed: int = 0) -> None:
        """
        Sets seed and changes fixed_seed to True to allow for seed usage.

        Args:
            seed (int): Seed for random generators.

        To-Do:
            - Think about adding a system where the seed changes every 10 or 20 runs.
        """
        self.seed_num = seed
        self.fixed_seed = True


    def translate_map(self) -> Tuple[List[List[int]], dict]:
        """
        Translates map by characters into integers.

        Returns:
            Tuple[List[List[int]], dict]: Tuple of the agent's map and info collected during translation.
        """

        # Initialises info to be empty per check through
        info = {
            "enemies": 0,
            "potions": 0,
            "gold": self.engine.player.gold,
            "player health": self.engine.player.hp,
            "map": "",
            "agent view": ""
        }
        # Initialise the agent's labelled view of the map.
        self.agent_view = np.array([[0]*(self.engine.level.width)]*(self.engine.level.height+1))

        # Loops over actual map to translate it into the agent's view.
        for y in range(self.engine.level.height):
            for x in range(self.engine.level.width):

                # Handles whether the agent has perfect or imperfect information.
                if self.perfect_info:
                    self.agent_view[y,x] = level_translator[ self.engine.level.tiles[x,y].char ]
                else:
                    if self.engine.layout.tiles[x,y].explored:
                        self.agent_view[y,x] = level_translator[ self.engine.level.tiles[x,y].char ]
                    else:
                        self.agent_view[y,x] = level_translator[" "]

                # Adds enemies and potions to info for better logs
                if self.engine.level.tiles[x,y].char == "+":
                    info["potions"] += 1
                elif self.engine.level.tiles[x,y].char == "v" or self.engine.level.tiles[x,y].char == "z":
                    info["enemies"] += 1
                elif self.engine.level.tiles[x,y].char == ">":
                    self.exit_location = (x,y)

                # For rewarding agent for exploring tiles
                if self.engine.layout.tiles[x,y].explored and self.engine.level.tiles[x,y].char == ".":
                    self.explored_reward += 1

                # Build map as well for logs
                info["map"] += f"{self.engine.level.tiles[x,y].char}  "
                info["agent view"] += f"{self.agent_view[y,x]}  "
            info["map"] += "\n"
            info["agent view"] += "\n"
        info["map"] += "\n\n"
        info["agent view"] += "\n\n"

        self.agent_view[self.engine.level.height, 0] = self.engine.player.hp
        self.agent_view[self.engine.level.height, 1] = self.engine.depth
        self.agent_view[self.engine.level.height, 2] = self.engine.player.gold

        return self.agent_view, info

    def step(self, action: int) -> Tuple[List[List[int]], int, bool, dict]:
        """
        Steps through the game by doing one action for the player and one for the enemies.
        Starts by calculating fov to see if any enemies see agent.

        Args:
            action (int): The discrete action chose by the agent from the action space.

        Returns:
            Tuple[List[List[int]], int, bool, dict]: A tuple consisting of the next state, reward,
            if the episode is done, and an info dictionary
        """

        self.engine.fov()
        reward = 0
        action = action_translator[action]
        action_type, dest = self.engine.bump(self.engine.player.pos, action)

        if dest.char == "#":
            reward -= 1
        if dest.char == ">":
            self.exits_taken += 1
            print("Exit taken!")
            reward += 1000
            self.last_explored = 0

        self.engine.handle_enemy_turns()

        if self.engine.player.is_dead():
            reward -= 50
            self.done = True
            print("Player is dead.")

        if self.time_spent >= 2000:
            print("Time over.")
            self.done = True
        self.time_spent += 1
        self.last_explored = self.explored_reward
        self.explored_reward = 0
        next_state, info = self.translate_map()

        # Rewarding agent for exploring
        exploration_reward = self.explored_reward - self.last_explored
        reward += ceil(exploration_reward/2)
        print(next_state)
        return next_state, reward, self.done, info

    def render(self, mode="human") -> None:
        print(self.calculate_agent_view())

    def reset(self) -> List[List[int]]:
        # Resets all necessary values.
        self.player = copy.deepcopy(player)
        if self.seed_num == 0:
            self.seed_num = random.randint(1, 100000)
        self.engine = Engine(self.player, self.fixed_seed, self.seed_num)
        self.time_spent = 0
        self.path_reward = 0
        self.explored_reward = 0
        self.last_explored = 0
        self.exits_taken = 0
        self.done = False
        obs, info = self.translate_map()
        return obs
```

Listing A.1: Using OpenAI Gym to extend the included Environment class into a custom environment for the roguelike game made for the project.

Appendix B

Testing using Gym

```python
from gym import Env
from stable_baselines3 import PPO

def test_model(model: PPO, env: Env) -> None:
    """
    Tests model and logs detailed customised logs.

    Args:
        model (PPO): The model to test against.
        env (Env): The environment used for the model.
    """

    log_name = "CustomLog"
    custom_log_path = os.path.join('Environment', 'Custom Logs') + "\\"
    full_custom_path = next_available(log_name, custom_log_path, ".txt")

    log_name = "FormalLog"
    formal_log_path = os.path.join('Environment', 'Formal Logs') + "\\"
    full_formal_path = next_available(log_name, formal_log_path, ".txt")

    custom_log = open(full_custom_path, 'w')
    formal_log = open(full_formal_path, 'w')

    episodes = 10
    for episode in range(1, episodes+1):
        obs = env.reset()
        done = False
        score = 0
        turn = 0

        while not done:
            action, _ = model.predict(obs)
            obs, reward, done, info = env.step(action)
            score += reward
            turn += 1
            if turn % 5 == 0:
                exits_taken = info[0]["exits taken"]
                potions = info[0]["potions"]
                enemies = info[0]['enemies']
                custom_log.write(f"!!! Episode: {episode} !!!\n")
                custom_log.write(f"Score: {int(score[0])}\n")
                custom_log.write(f"Turn: {turn}\n")

                formal_log.write(f"!!! Episode: {episode} !!!\n")
                formal_log.write(f"Score: {int(score[0])}\n")
                formal_log.write(f"Turn: {turn}\n")
                formal_log.write(f"Exits taken: {exits_taken}\n")
                formal_log.write(f"Potions in map: {potions}\n")
                formal_log.write(f"Enemies in map: {enemies}\n\n\n")

                for key in info[0]:
                    if key in ["map", "agent view"]:
                        custom_log.write(f"{key}:\n {info[0][key]}\n")
                    else:
                        custom_log.write(f"{key}: {info[0][key]}\n")
    custom_log.close()
    formal_log.close()
```

Listing B.1: Testing model performance using Gym