LEARNABLE AGENT FOR VIDEO GAME PLAYING

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Qi Xiang

Supervised by Dr. Ke Chen

Department of Computer Science

Contents

Word Count: 11614

List of Tables

List of Figures

Abstract

In this study, I aim to develop an autonomous agent capable of playing video games with proficiency comparable to that of human players. I explore deep reinforcement learning (DRL) methods, with a focus on Deep Q-network (DQN) and Proximal Policy Optimization (PPO) techniques. The agent's performance is evaluated based on its ability to learn from dynamic game environments and effectively defeat game bosses. Success metrics include the agent's kill-to-death ratio and its ability to apply learned strategies to new, unseen gaming scenarios. The ultimate goal is to demonstrate that the agent can independently adapt and perform tasks that typically require human-level intelligence and reflexes. The final result shows that the model trained using PPO can actually perform the same as the human expert level with a kill-to-death ratio of 1.36.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor, Dr. Ke Chen, for his continued help throughout the project.

Chapter 1

Introduction

Creating an AI agent capable of achieving human-level proficiency in video games remains a frontier in artificial intelligence research. It combines the challenge of understanding complex and constantly changing environments with the practicality of algorithmic learning and development. This project challenges the limits of what AI is capable of and acts as a testing ground for refining algorithms that may have broader applications beyond gaming. This introduction summarises the motivations, objectives, methodologies, expected challenges, and deliverables of a project focused on creating an agent using deep learning algorithms.

Traditional approaches in AI, including search-based techniques and rule-based systems, have demonstrated success in various games by following predefined strategies or exploring possible moves within limited scopes. For example, as M. O. Riedl and A. Zook stated, the early use of artificial intelligence in video games serves as an intermediary between the player and the game, or, in other words, a non-player character (NPC) [18]. They will only follow predefined rules and perform the same action as programmed. However, these methods often need to be revised when considering the complexity and unpredictability of more sophisticated video games, which require strategic planning and real-time decision-making based on visual input and partial information.

In today's game development, Reinforcement learning (RL) is commonly used in creating NPCs [30]. With the help of deep learning, which can learn and make intelligent decisions based on large datasets, we can create a powerful tool for developing video game-playing agents that can start from scratch and achieve or even surpass human expert-level skills. In 2016, AlphaGo defeated Ke Jie, one of the world's best Go players, using deep reinforcement learning and Monte Carlo tree search. Also in 2019,

OpenAI FIVE, which plays the game Dota 2, became the first AI system to defeat at-that-time world champions in an esports game.

The primary objective of this project is to utilise deep learning algorithms to develop an artificial intelligence agent capable of acquiring the ability to play video games starting from a complete lack of skill. This agent can gain knowledge from its own experiences and pre-trained models from residual networks.

Enabling agents to learn from high-dimensional sets of input data is a problem that has been actively addressed in the current field of reinforcement learning [14]. Unlike other machine learning applications that rely on images, such as face recognition, this involves learning from dynamic scenes that change rapidly, such as every frame of a game or even multiple images within each frame. As OpenAI did in Dota 2, the reaction time is 7.5 frames per second [16].

Nevertheless, developing deep reinforcement learning allows us to extract useful feature data from raw image inputs [8]. In the article, the authors describe the introduction of neural networks into reinforcement learning, including convolutional neural networks. The authors employed a large convolutional neural network to handle the article's vast image data. This convolutional neural network includes five layers, and the final result is beyond the state-of-the-art image-processing neural network at that time. The integration of convolutional neural networks into reinforcement learning can be employed to create deep reinforcement learning (DRL).

I will experiment with DRL agents' flexibility and robustness in the project's dynamic, three-dimensional gaming environments. These settings represent contemporary video games and real-world situations that call for quick decisions, constant adaptation to new information, and shifting circumstances. The project's focus on 3D games also addresses the greater complexity and sensory demands these environments—as opposed to more conventional, simpler 2D environments—place on learning algorithms.

The project uses several deep reinforcement learning methods to overcome these obstacles, including Deep Q Network (DQN), Dueling DQN, and the state-of-the-art Proximal Policy Optimization (PPO) algorithm. Among the reinforcement learning algorithms, those methods work exceptionally well on tasks where the states and actions that can be taken are high-dimensional and discrete. Experience replay enhances learning efficiency and stability by enabling the network to learn from previous experiences. Convolutional Neural Networks (CNN), which are excellent at interpreting visual data and are essential for processing and comprehending complicated 3D game

environments, are used to enhance this [1].

Additionally, the exploration-exploitation trade-off is balanced by applying an epsilon-greedy strategy. This technique is essential for directing the agent's learning process because it determines whether an action should be taken randomly (exploration) or based on previously learned strategies (exploitation). This balance must be calibrated carefully to ensure the agent stays learning and adapting throughout its training and does not get stuck in local optima.

The following sections will be structured as: The background section will discuss the evolution of my methods, their relevance to modern gaming, and the theoretical frameworks underpinning those technologies. The methodology and experiments section comprehensively explains how those methods are incorporated into this project. This section will also outline the configuration of the gaming environment used for testing the DRL agent. Here, I will also describe the various experiments conducted to assess the performance of the DRL agent in the 3D gaming environment. It will detail the setup, the processes, and the criteria used to measure the agent's performance. The final part is evaluation. This section will present and analyse the agent's training and testing results.

1.0.1 Sekiro: Shadows Die Twice

Sekiro: Shadows Die Twice is an action-adventure game developed by FromSoftware and published by Activision. The player takes on the role of a shinobi (ninja) named Wolf as he sets out on a quest to rescue his kidnapped lord and seek revenge on his enemies.

1.0.1.1 Mission Explained

The game features two characters, the agent and the boss, battling in a fixed-sized arena. The agent aims to attack the boss to reduce its health points (hp) while maintaining its own survival. Both characters can attack and guard against the opponent's attacks. Successfully guarding at the correct moment prevents damage from the opponent's attacks. However, guarding at the wrong time results in a penalty by increasing the agent's posture value. The boss, a non-player character, executes pre-determined moves randomly, which means it follows a set of moves designed to defeat the agent, yet the agent cannot predict the boss's next move. The agent can perform manual moves, including moving in four directions, jumping, and, most crucially, attacking

and guarding.

To play Sekiro, the agent must address the following challenges:

High-dimension observation spaces

While the agent is fighting the boss, I use screenshots to capture the opponent's action to pass into the network, which decides the agent's next move. The dimension of the screenshot is 224 in both width and height and since the image is in RGB with a channel number of 3, this yields a total of $\tilde{1}50000$ data points for each screenshot.

Sparse Rewards

The reward signal in Sekiro can be sparse, as the rewards are given for defeating the boss or giving damage to the boss. Designing a reward function that encourages meaningful learning and exploration without leading to suboptimal behaviour traps is a significant challenge.

Figure 1.1: Sekiro game play

1.0.2 Aims and Success Criteria

The objective is to achieve and surpass the performance level of humans in these games. The success of this project will be evaluated through specific metrics: first, by assessing whether the agent can independently learn effective strategies to defeat game bosses; second, by measuring the kill-to-death ratio as a direct indicator of performance efficiency; and third, by evaluating the agent's ability to transfer learned strategies to new, previously unencountered game scenarios. These criteria will help determine the extent to which the agent has mastered the game environment and can adapt to new challenges, mirroring human-like learning and adaptability.

Chapter 2

Background and Introductions

This section aims to provide a comprehensive overview of the methods used in this project. This includes a detailed discussion of the critical components that fall under each method.

2.0.1 Autonomous Agent in Video Games

The concept of learnable autonomous agents in video games often refers to Non-character Players (NPCs) or entities that operate under their own control, guided by artificial intelligence (AI) algorithms, rather than being controlled directly by human players. These agents evolve over time from simple rule-based systems to complex entities capable of learning and evolving within dynamic virtual environments.

Rule-based AI: Early Developments

At early ages, video games were basic and relied on simple if-then-else statements and predetermined behaviours. Early games such as ”Pac-Man” (1980) used pattern-based movement for ghosts; each follows a set of rules to chase or evade the player. Although these behaviours were predictable, they paved the way for more complex AI systems in video games.

The Rise of Scripted Behaviours

With the development of games, their AI also became more sophisticated. Games like ”The Legend of Zelda” (1986) and ”Half-Life” (1998) featured enemies with predefined patterns that can react to player actions. However, these scripted AI were still predictable, highlighting the need for more adaptive and dynamic systems.

Introduction of Machine Learning Techniques

An example of incorporating machine learning into video game AI is the game called ”Black & White” (2001). This god game introduced a creature AI capable of learning from the player’s actions. It allows the creature to adjust its behaviour based on positive or negative feedback from the player.

Reinforcement Learning and Deep Learning

The most well-known application of reinforcement learning (RL) and deep learning in games is DeepMind’s ”AlphaGo” (2016). While Go is not a video game, AlphaGo’s ability to learn and master the complex game of Go inspired similar approaches in video game AI.

The Era of OpenAI and Beyond

OpenAI’s groundbreaking work of advanced AI models, such as those used in ”Dota2”, has demonstrated the capability of AI agents to compete at high levels against human players. OpenAI Five, which this project is inspired by, showcased the ability to learn complex team strategies and adapt to opponent’s tactics in real-time.

2.0.2 Deep Learning

Deep learning, a subset of machine learning, is currently leading the way in artificial intelligence research and application, bringing about essential advancements in diverse fields. Its origins can be traced back to the concept of artificial neural networks, inspired by the human brain’s architecture [2]. The term ’deep learning’ refers to the usage of neural networks. In typical cases, it is used along with multiple layers, which allows for processing complex data and enhances the ability to learn and model intricate patterns [19].

2.0.2.1 Artificial Neural Network

An ANN typically consists of various nodes or neurons connected by edges that carry weights. These neurons are combined into layers, forming the whole network. Commonly, an ANN consists of an input layer, at least one hidden layer, and an output layer. The input layer receives data that we want the neural network to learn. Then, it passes this data to the hidden layer(s), which processes the data through weighted

connections and then through the output layer, which provides the result. To make the network learn from its input data, backpropagation and optimisation techniques such as gradient descent are used, and the weights between nodes are updated according to the error.

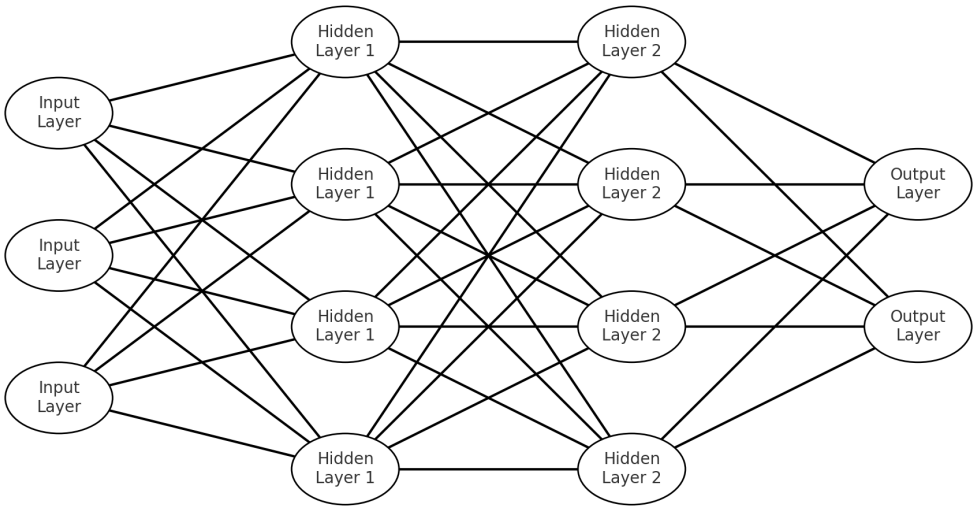

Figure 2.1: Structure of a typical ANN with two hidden layers

Various types of neural networks are used for different applications. For example, a recurrent neural network (RNN), which uses sequential or time series data, is commonly used for natural language processing and speech recognition. And there is the convolutional neural network, which is used for computer vision tasks.

2.0.2.2 Convolutional Neural Network

A convolutional neural network is a feed-forward neural network specialising in processing images. It comprises various building blocks, including the convolutional layer, pooling layer, activation functions, and fully connected layer.

Convolutional Layer

At the heart of a CNN is the convolutional layer, which requires a few components: an input image, a filter, and a feature map. The filter, commonly a 3x3 matrix, is put onto the image, and the dot product is calculated between the pixel and the filter. This output product is then put into an output array, and the number of strides shifts the filter. We repeat the process until the filter has swept across the entire image. The output from convolving the image is called the feature map. In theory, features can be learned using only fully connected layers; however, this requires many parameters, as each neuron in one layer would need to connect to every neuron in the previous layer. On the other hand, a convolutional neural network has this shared-weights architecture, which significantly reduces the number of weights needed to learn features from the image. A CNN learns from a process called backpropagation, which combines an optimisation algorithm such as stochastic gradient descent or some variants like Adam or RMSprop. In this project, Adam is used, which will be discussed later in this report. Some parameters need to be set before the learning process:

1. The number of filters determines the depth of the feature map and increases the feature detection capacity because each filter is designed to detect a specific type of feature in the input data. For example, some are designed to detect edges, and some to detect textures or more complex patterns.

2. Stride number is the number of pixels the filter shifts when it applies a convolution operation. A stride number greater than one reduces the spatial dimensions of the output feature map compared to the input.

3. Zero-padding is the technique of adding layers of zeros around the border of the input image; this usually helps preserve the information at the image's borders. There are several types of padding strategies.

    (a) Valid Padding: No padding is applied. Performing a convolution operation will reduce the input layer, resulting in reduced output dimensions.

    (b) Same Padding: This strategy ensures that the output layer has the same size as the input layer by adding enough zeros around the input to allow the filter to slide over the input border.

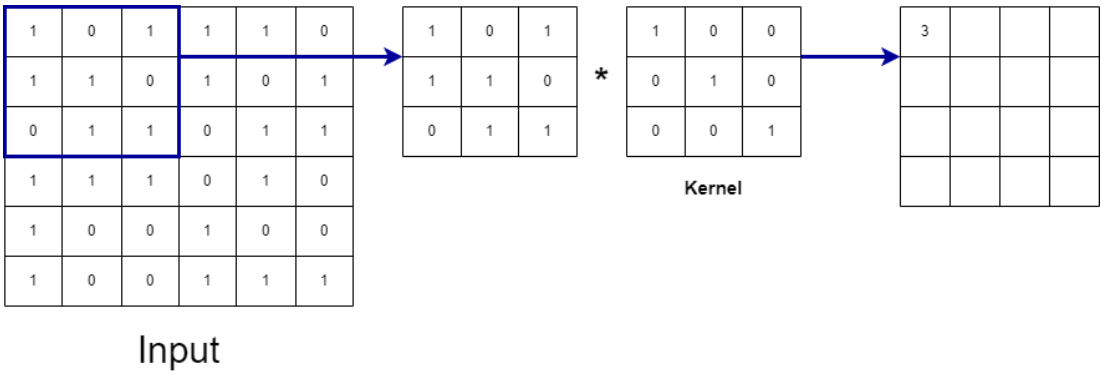

Figure 2.2: This is an example of a convolutional layer; as you can see in the graph, an element-wise multiplication is performed between the filter and the image pixels and then sums up all the multiplied values to produce a single output in the feature map [4]

Pooling Layer

A pooling layer is a form of down-sampling similar to convolution layers by sliding a kernel across the input and applying the pooling operation (maximum or average, depending on the type of pooling used). It decreases the size of the feature maps, reducing the network parameters. The only difference between the pooling layer and the convolution layer is that they don't have any weights and have no parameters to learn. There are two most common types of pooling:

1. Max Pooling: Takes the maximum value from the sub-region of the input feature map. This strategy effectively preserves the most prominent features, such as edges.

2. Average Pooling: As the name states, it takes the average value of each input sub-region. This method tends to smooth over the input features and is less commonly used in CNN than max pooling.

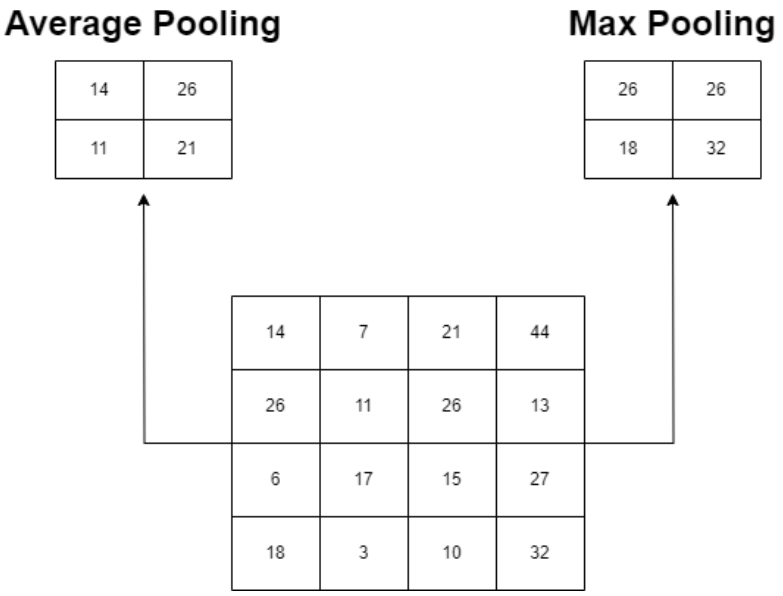

Figure 2.3: Max pooling and Average Pooling

Activation Function

An activation function is a mathematical operation applied to a neuron's input that produces an output signal for the next layer. It helps the network learn non-linear relationships between the input and output, allowing it to learn complex patterns in the data, such as those found in images, speech, and text. The most commonly used activation functions in a convolutional neural network are:

1. Rectified Linear Unit (ReLU): ReLU is currently one of the most popular activation functions, mathematically defined as $f(x) = max(0,x)$. It uses simple logic: it returns the input if it's positive and 0 if it's negative. Its advantages are straightforward: since the operations include only comparison, addition, and multiplication, the computation is very fast, and it also helps to mitigate the vanishing gradient problem. However, vanilla ReLU can suffer from the 'dying ReLU' problem, where it always outputs the same value for any input.

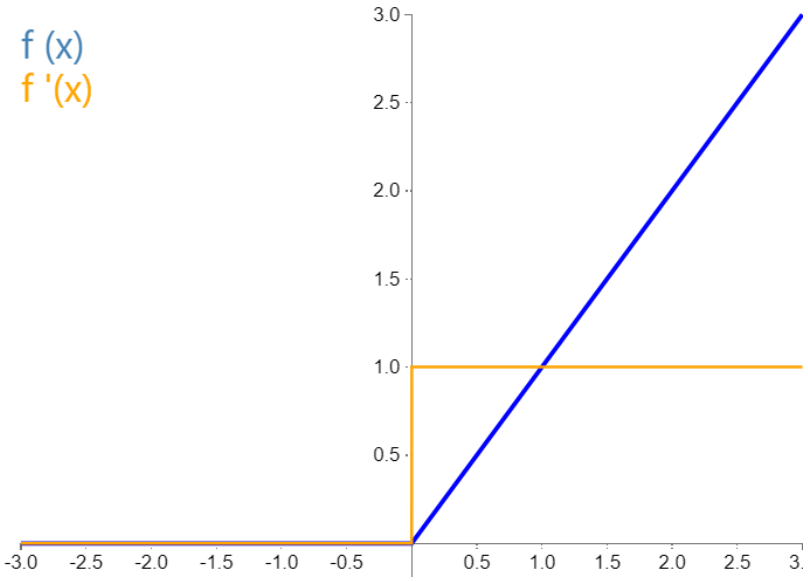

Figure 2.4: Visualisation of ReLU and its first derivative

2. Sigmoid and Tanh: while historically significant in developing neural networks, they are not as popular as the ReLU for several reasons. One of which is the vanishing gradient problem. The output range for sigmoid is between 0 and 1, and Tanh is between -1 and 1. The gradient will be minimal when the input value is far from zero. When these small gradients multiply together during backpropagation, it will lead to exponentially smaller gradients in the layers, which reduces these layers' ability to learn.

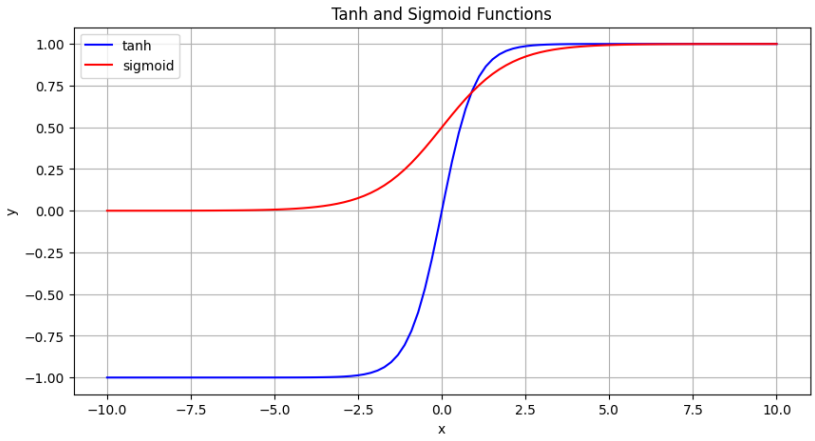

Figure 2.5: Sigmoid and Tanh

Fully Connected Layer

Unlike in the convolutional layers, where each neuron in a given layer is only connected to a small region of the input, in a fully connected layer, however, every node is connected to every neuron in the previous layer (hence the name fully connected). These layers are often put at the end of CNNs, integrating the learned features and, oftentimes, making a decision or prediction. While convolution and pooling layers tend to use ReLU as their activation function, fully connected layers usually use the softmax function to produce a probability in the range (0,1).

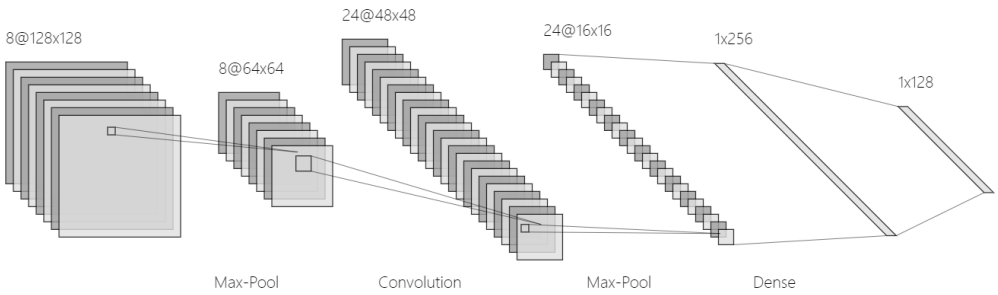

Figure 2.6: The structure of a typical CNN network

2.0.3 Reinforcement Learning and Deep Reinforcement Learning (DRL)

Reinforcement learning (RL) is a specialised approach in the larger field of machine learning that specifically deals with determining the optimal actions for agents in an environment to maximise the overall reward over time [12]. This approach is fundamentally distinct from supervised learning, wherein learning occurs through examples given by a knowledgeable external supervisor, or what we call a teacher [17]. The basic idea behind RL is trial-and-error search and delayed reward, which involves managing the trade-off between exploring new areas and exploiting existing knowledge [23]. The fundamental aspect of reinforcement learning is the interaction between the agent and its environment. In most cases, this interaction is represented as a Markov decision process (MDP), defined by an array of states, actions, and rewards [7]. At every step, the agent is in some state s, takes an action, receives a reward, and transits to the next

state s' based on the probability of the next state given the current state and action $p(s'|s,a)$.

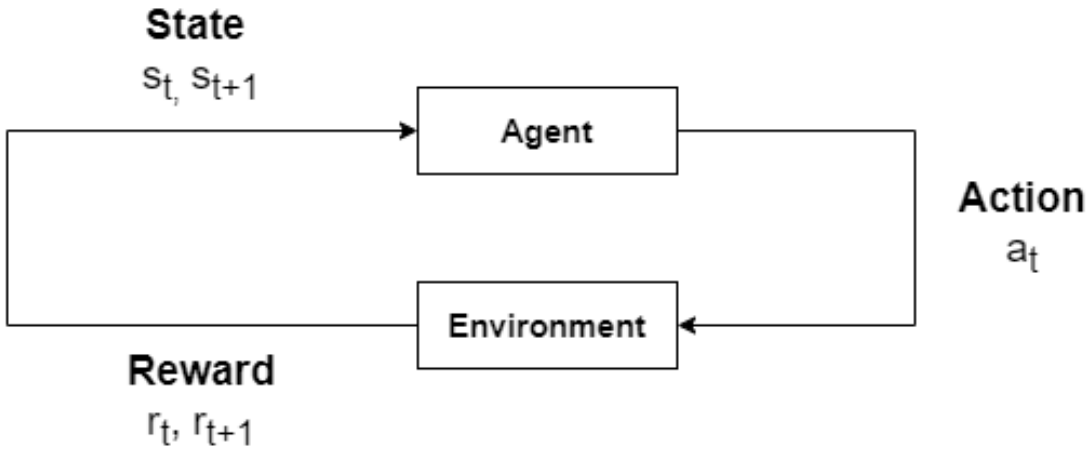

Figure 2.7: The flow of reinforcement learning

The agent aims to find a policy, a strategy for selecting actions that maximise future rewards. During this process, value functions are estimated to determine how valuable each state of a state-action pair is based on the choices the agent has made [24]. A key challenge in the field of reinforcement learning is the trade-off between exploration and exploitation [29]. An agent must explore the environment to find actions that yield rewards and exploit its current knowledge to gain rewards. Several methods were proposed to tackle this challenge, including epsilon-greedy, which involves the agent occasionally selecting a random action to explore the environment, and many others. Another significant challenge is the high dimensionality, as the state and action spaces are vast in many problems, making it impossible to explore and learn about every existing state-action pair. However, deep reinforcement learning, a combination of reinforcement learning and deep learning, has shown remarkable efficiency in solving tasks with high-dimensional state spaces, such as playing video games at the expert level [15] or mastering the game of Go [22]. Also, the recent development of policy gradients and actor-critic methods provides more ways to optimise policies directly or balance the policy's evaluation and improvement. These methods offer more reinforcement learning applications, including robotics, autonomous navigation, and financial decision-making, by providing more flexible and efficient learning algorithms, particularly in domains that require continuous control [13]. However, continuous control is outside this project's scope, as the Sekiro agent chooses its move from a discrete space of actions.

2.0.4 Q-Learning

Q-learning is a model-free, off-policy reinforcement learning algorithm that differentiates from a model-based algorithm such as SARSA. It is a form of temporal different (TD) learning, which means the agent learns from the environment iteratively and thus improves its estimation action-value function based on the reward from the environment [21]. The action-value function, represented as $Q(s,a)$, is the expected output of taking an action a in a state s following a certain policy. At the core of Q-learning is the Q-function, which the agent uses to estimate the rewards of the state-action pairs. The agent then uses rewards to choose actions that maximise the expected rewards. The core behind each update is the Bellman equation, given by $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$. In this equation, s is the current state, a is the current action, and r is the reward of taking action a at state s. s' represents the new state after taking action a. α is the learning rate, and γ is the discount factor, which is a number between 0 and 1. It acts as a balance of immediate reward and future reward. If γ is equal to 1, it means the agent values future rewards just as much as current rewards, and conversely, a gamma of 0 means the agent will only value immediate rewards. The term $Q(s',a')$ represents the estimate of the best future reward, and this Q value is updated based on the sum of the immediate reward plus the discounted future reward.

Model-free algorithm

A model-free algorithm does not necessarily incorporate the Markov decision process, which means it does not have access to the transition probabilities or the state-pair action rewards upfront [27]. In other words, in a model-based algorithm, the agent can predict the reward of some action before performing it, thereby planning what it will do by choosing the action that gives out the highest probability. Whereas in a model-free algorithm, the agent needs to carry out the actions to see what happens and learn from it [26].

Off-policy

Off-policy in reinforcement learning refers to how the agent learns and decides on actions within an environment. In a typical off-policy learning algorithm like Q-learning, the agent can learn about one policy while following another [9]. This means the algorithm can learn about the optimal policy $Q*(s,a)$ regardless of the agent's current

strategy used to learn from this state. This means the agent can explore randomly or follow a policy that might not be the best. This is also mentioned in ”Reinforcement Learning: An Introduction” by Sutton and Barto [23], in which off-policy learning is seen as a critical paradigm that enables agents to learn optimal policies through observation. This differs from on-policy methods, in which the agent learns from the actions taken.

2.0.4.1 Deep Q-Learning Network

A deep Q-learning network (DQN) is a reinforcement learning algorithm that combines Q-learning and deep learning. First introduced in 2013 in the paper ”Playing Atari with Deep Reinforcement Learning,” this paper has laid the groundwork for the possibility of using neural networks in combination with Q learning to enable models to learn policies from high-dimensional visual inputs [14]. At the core of every reinforcement learning algorithm is the concept of agents learning to make decisions by interacting with the environment. Q learning does this by keeping a tabular representation of the Q-value function of state-action pairs and estimating the total reward the agent can obtain by taking action in a given state following an optimal policy. However, because it is a two-dimensional tabular, the size becomes incredibly large when either of the dimensions is large. Imagine having 10,000 states and 100 actions in the environment (common in training environments); this will create a table of 1 million cells. DQN addresses this limitation by incorporating a deep neural network as a function approximator for the Q-value function. It takes the state of the environment as input and outputs the Q value for each possible action. This network is updated each episode in combination with exploration and exploitation strategies such as epsilon-greedy. DQN has proven effective in training agents with high-dimensional sensory inputs such as video games or robotics.

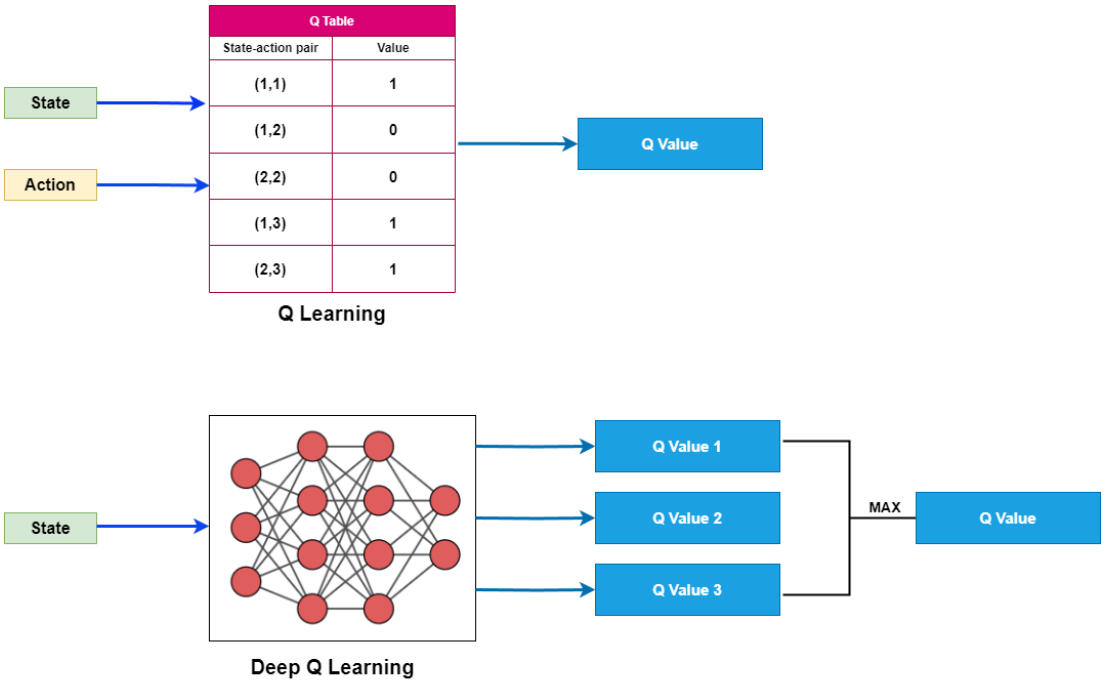

Figure 2.8: Q learning vs DQN

Experience Replay

An important component of DQN is the experience replay. This technique stores the agent's past experiences at each step $e_t = (s_t, a_t, r_t, s_t + 1)$ into a replay buffer and randomly samples them later in the learning process to train the Q-Network. This breaks the correlation between consecutive training samples, leading to more stable and effective training because highly correlated data could cause the training to diverge. Also, each experience can be reused multiple times, which is beneficial in environments where acquiring new experiences is costly.

Target Network

A main challenge in the original deep Q-learning network is the moving target problem. Since our objective is to minimise the difference (or the TD error) between the current Q-value estimate $Q(s,a)$ and the target Q-value y, as the parameters of the Q-Network change, the target y also change. This is because y is calculated using the same Q-Network that has been updated. The target network solves this issue by copying the parameters denoted as θ^- from the learning network, but they are updated less frequently. This update occurs only after hundreds or even thousands of episodes,

thereby providing a more stable target for the training network to aim for.

2.0.4.2 Dueling DQN

Dueling DQN is an advancement of vanilla DQN similar to double DQN; it provides a more sophisticated way of estimating state-action values (Q-values). The idea is that Q values can be decomposed into two distinct paths: the state value function $V(s)$ and the advantage value $A(s,a)$.

$$Q(s,a) = V(s) + A(s,a)$$

The state value function $V(s)$ represents the value of being in a certain state regardless of which action is taken. The advantage value function $A(s,a)$ compares how good an action is to other actions at a given state. The Dueling DQN network then combines those two paths into a single output of each action's Q-value. But this combination is not done using simple addition because that would lead to the issue of unidentifiability since we cannot tell apart $V(s)$ and $A(s,a)$ from $Q(s,a)$ as adding a constant to all advantages and subtracting the same constant from $V(s)$ would result in the same Q-value. Instead, the architecture uses an aggregation strategy to ensure that the advantages have a zero mean at any state. This is done by subtracting the mean of the advantage estimates from each advantage value. It ensures that the architecture can distinguish between the value of being in a state and the value of taking specific actions in that state. It ensures the model does not randomly favour arbitrary actions due to inflated advantage values (we want to learn how good or worse an action is compared to the average action in that state rather than an absolute value indicating how good the action is). Also, this normalisation prevents the network from attributing too much value to the advantages when the actual difference is from being in a specific valuable or unvaluable state. Incorporating Dueling DQN in my project is helpful since it is unnecessary to know the value of each action at any time step [31]. In Sekiro, only actions in certain states that result in a health point loss of either the boss or the player do matter (but also dodging the boss's attack, but there isn't a reward for that action).

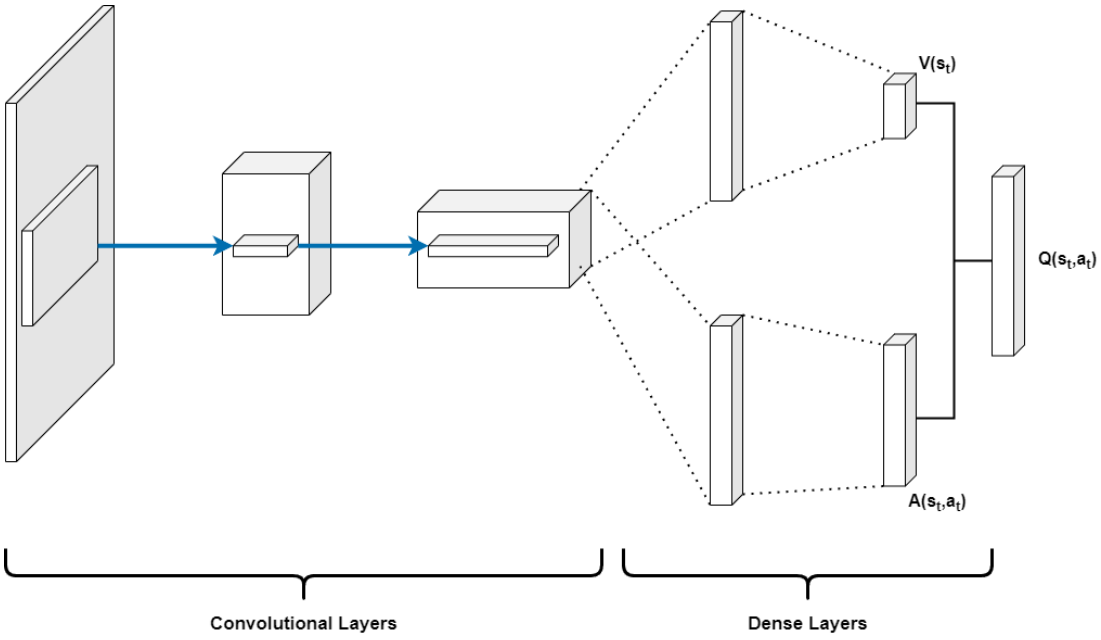

Figure 2.9: Dueling DQN network architecture

2.0.5 PPO

Proximal Policy Optimization (PPO) was introduced in 2017 by OpenAI and has now become one of the most popular algorithms for training deep neural networks to solve complex RL problems. It improves the stability and efficiency of training, which DQN suffers from due to the overestimation of Q-values. This is done by clipping the ratio of the current and the old policies in a specific range: $[1-\varepsilon, 1+\varepsilon]$. This ensures that the steps at each policy update are not too large to make the training more stable. We want to have smaller policy updates because the model is more likely to converge to optimal and avoid the gradient 'falling off a cliff'. By clipping, we make sure that the new policy doesn't go too far from the old one, thus limiting the size of each update. Before PPO, John Schulman's Trust Region Policy Optimization was suggested to fix the instability problem in DQN. It added a trust region constraint to the objective function to manage the KL difference between the old and new policies [20]. Below is the objective function for TRPO. An objective function is a mathematical function that an algorithm seeks to minimise or maximise. It represents the algorithm's goal, which is often to find the best solution and make the most accurate prediction. In deep learning, the objective function guides the training process of the deep neural network towards optimal policies. This equation can be interpreted as maximising the objective

function while satisfying the KL constraint on the step size of the policy update.

$$\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}A_{\text{old}}(s,a)\right]$$
$$\text{subject to } \mathbb{E}_{s\sim\pi_{\text{old}}}\left[D_{\text{KL}}\left(\pi_{\text{old}}(\cdot|s) \parallel \pi_\theta(\cdot|s)\right)\right] \leq \delta$$

Although TRPO was theoretically proven to improve performance, its second-order optimisation nature is highly computationally expensive and difficult to scale up in large and complex network architectures [25]. PPO seeks to keep the benefits of TRPO while reducing the computational overhead. It does this by using a clipped surrogate objective function, which is improved from the policy gradient function. It prevents each step size from being too high or too small in each update. But first, we need to look at the basic form of policy gradient objective function used in Actor-Critic or REINFORCE.

$$L^{PG}(\theta) = \mathbb{E}_t\left[\log \pi_\theta(a_t|s_t)\cdot A_t\right]$$

. The idea behind this is similar to what it does in the Dueling DQN, where we want to make our agent take actions that lead to a higher reward and avoid dangerous situations. However, this leads to two problems: if the learning rate (or the step size) used during optimisation when updating the policy parameters θ is too small, the learning process will be very slow, which could potentially need a very large number of iterations to converge to an optimal policy. If the step size is too large, it will cause the updates to overshoot the optimum, leading to unstable training. PPO tackles this by contrainting the policy change in a small range using clipping.

$$L^{CLIP}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)A_t\right)\right]$$

. The $r_t(\theta)$ is a ratio function that represents the probability of taking action a_t at state s_t in the current policy compared to the probability for the previous policy.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

. The left part of the new objective function $r_t(\theta)A_t$ replaces the original log probability. If $r_t(\theta) > 1$, then taking action a_t in state s_t is more favourable or advantageous than it was previously, and if $r_t(\theta)$ is between 0 and 1, it means the action is less

advantageous than it was before.

$$\text{clip}\,(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t$$

The right-hand side of the objective function clips $r_t(\theta)$ in the range $[1-\varepsilon, 1+\varepsilon]$, ε is a hyperparameter that needs experimentation later on; in the OpenAI paper, it's 0.2.

$p_t(\theta) > 0$	A_t	**Return Value of** min	**Objective is Clipped**	**Sign of Objective**	**Gradient**
$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	+	$p_t(\theta)A_t$	no	+	✓
$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	−	$p_t(\theta)A_t$	no	−	✓
$p_t(\theta) < 1-\epsilon$	+	$p_t(\theta)A_t$	no	+	✓
$p_t(\theta) < 1-\epsilon$	−	$(1-\epsilon)A_t$	yes	−	**0**
$p_t(\theta) > 1+\epsilon$	+	$(1+\epsilon)A_t$	yes	+	**0**
$p_t(\theta) > 1+\epsilon$	−	$p_t(\theta)A_t$	no	−	✓

Figure 2.10: Image from "Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization" by Daniel Bick, this table summarises all the possible outcomes of the objective function. The gradient column means whether the gradient will result in a backpropagation

If the probability ratio lies within the interval $[1-\varepsilon, 1+\varepsilon]$, clipping does not apply. In that case, if we have a positive advantage, we should increase the probability of our policy taking that action, thus maximising the unclipped objective. The same applies when the advantage is negative, but instead, we decrease the probability of taking that action.

Now, in the case where $p_t(\theta) > 1+\varepsilon$, we also need to consider two cases: whether the advantage is positive or negative. In the case where advantage A_t is positive, and the ratio is above the range since the probability is already larger than the previous policy, we would like to apply clipping and return the minimum value between $p_t(\theta)A_t$ and $(1-\varepsilon)A_t$. This will result in a zero gradient (as seen in the table, the line above the range is always straight), so we don't update our weights. Conversely, in the case $A_t < 0$ while $p_t(\theta) > 1+\varepsilon$, the clipped objective will evaluate to a negative value $(1-\varepsilon)A_t$. Since the probability before clipping is over the range, the unclipped part $p_t(\theta)$ will be a larger number in the negative direction. Consequently, the minimum operator will choose the unclipped objective, thus maximising the negative value $p_t(\theta)A_t$, which means we want to decrease the probability of taking that action at that state.

Finally, we consider if the probability ratio is lower than $[1-\varepsilon]$. We also need to

consider whether A_t is positive or negative. If the advantage estimate is positive, the minimum operator will return the minimal value of the objective input, which is the unclipped objective $p_t(\theta)A_t$. The gradient will point in the direction of maximising the objective function, which means we want to increase the probability of taking that action in that state. But if the advantage is negative, clipping is applied. This is because we don't want to decrease the probability of taking that action further, which is to ensure stability and prevent the policy from moving too far from its previous state.

2.0.6 Transfer Learning (ResNet)

Transfer learning in the context of machine learning is the action of reusing a pre-trained model on a new problem [5]. It serves the purpose of improving network performance and saving training time. In this project, I want to explore the possibility of leveraging ResNet with a deep reinforcement learning algorithm to build the game agent. ResNet, which stands for residual network, is a type of convolutional neural network introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in 2015. It uses residual blocks, and feature skipping connections, which allow the network to learn identity functions more effectively. As a result, it enables the training of networks that are much deeper than was previously possible.

ResNet comes in different names, with the name in the model's name indicating the layers with learnable weights. For example, ResNet34, which I used in this project, is a 34-layer convolutional neural network containing convolutional layers, max-pooling and fully connected layers. All these layers are grouped into 16 residual blocks, which are then grouped into four sets with different numbers of blocks and filters. These blocks prevent the degradation of performance when training deeper networks.

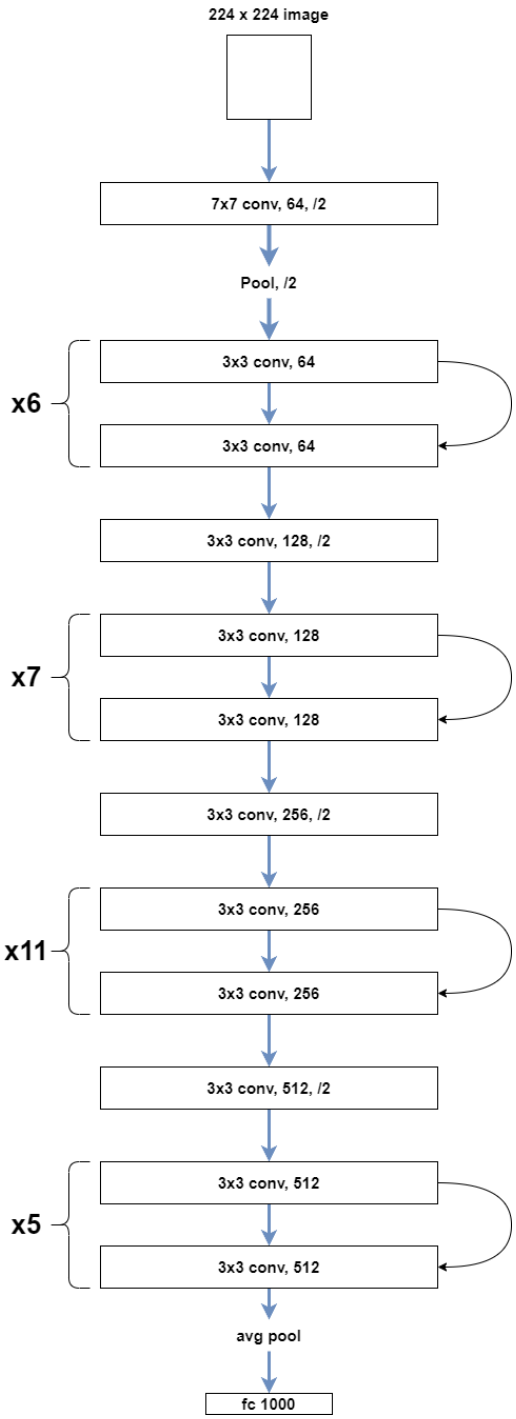

Figure 2.11: ResNet34 architecture

Chapter 3

Methodology and Experiments

3.0.1 limitations

During the experimentation process, I faced a serious challenge due to the long training time for each model. In each episode of this project, the end of each episode is determined by the agent's death. Given this setting of episodes, each one will take nearly two minutes to complete. Given this constraint, it would be impractical to train each model over several millions of episodes, as that would lead to prohibitive time requirements and cannot be done within the scope of this project's timeline. To provide a fair yet efficient comparative analysis of different models and hyperparameters, I decided to fix the number of episodes for training in all subsequent experiments. This approach is meant to carry out meaningful experiments while keeping the project timeline manageable. However, this does not mean this is the optimal solution; fewer episodes can cause underfitting and limited exploration. Nevertheless, it is left for future work.

3.0.2 Training Environment

To provide a robust testbed for evaluating various machine learning algorithms, I adapted an existing training environment by Wenhao Chen et al. [28]. I made the game run at five times the speed to speed up the training process. The program follows the same manner as performing a CheatEngine to read and manipulate game memory; it does this by injecting code into the game's memory to get the health point and endurance value for both the player and the boss. This has improved my accuracy in calculating rewards compared to my initial visual input reading approach. Nevertheless, the training data is still the visual input of the boss's posture; this is done by capturing screenshots of the screen through a small region on the screen at every step.

These images are then passed into the neural network along with the health points and endurance values for feature extraction and learning. Several challenges have been tackled to ensure the agent can run indefinitely without experiencing any errors. First, after each episode, the agent is teleported back into the centre of the arena. This is to prevent the agent from losing sight of the boss in the corner and being unable to lock vision on the boss; the lock vision is done by manipulating the memory. Secondly, since the boss has only three lives after the agent has defeated the boss, the boss is automatically revived and counts that as a kill. Finally, to prevent a significantly low sample, the boss's damage and the ratio of the agent's health point deduction are scaled as the boss does huge damage to the agent.

3.0.3 Reward function

The initial approach to designing the reward mechanism was based on a straightforward principle: provide rewards for the agent to decrease the boss's health points (HP) and impose penalties for any loss of its own HP. The objective of this approach was to create a reward structure that accurately simulates the goals of the game, which are to defeat the boss and stay alive. A large reward was given for the primary objective of reducing the boss's HP to zero, representing victory. At the same time, alternatively, a significant penalty was enforced for the agent's failure, marked by its death. However, I discovered that during the initial phases of training, the large penalty associated with the agent's death, which occurred frequently during the early stages of training, had an overly negative impact on the learning process. This imbalance resulted in a situation where instances of death overshadowed the rest of the training process, disproportionately emphasising the avoidance of death. As a result, the agent adopted an overly cautious strategy, prioritising survival over exploration, which is potentially more effective but carries a higher risk of defeating the boss. This cautiousness restricted the agent's ability to learn and execute complex or aggressive strategies requiring calculated risks. To tackle this problem, I improved the reward function. Instead of giving a constant reward every time the agent has done damage to the boss, I used a multiplier so that the reward gets larger when the boss is low on health points. The design aimed to motivate the agent to defeat the boss gradually, which could lead to a more aggressive strategy to fight the boss. However, this approach did not work as expected. As seen in the graph below, the average reward per episode was compared for all the algorithms, and none of them showed any increasing trend, which indicates the agent is not learning effectively.

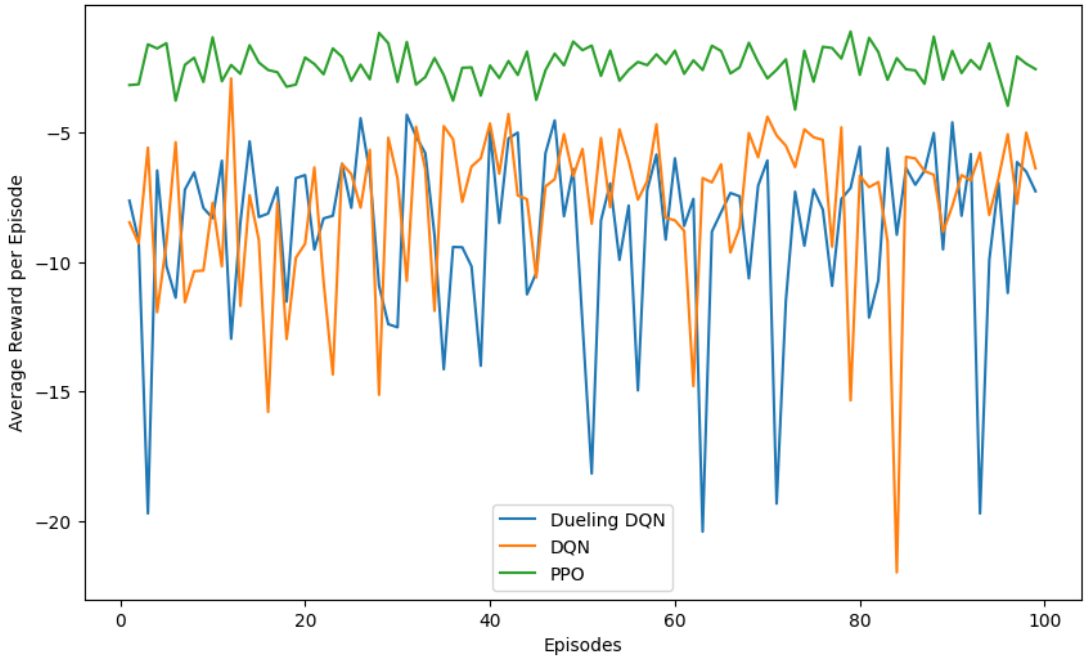

Figure 3.1: Average reward per episode using the initial reward function

I must improve the reward system to tackle this problem and establish a more equitable learning environment. One possible solution could be to lower the punishment for dying during training to decrease its overriding impact on the learning process. This would allow the agent to experiment with a broader range of strategies without the fear of incurring a substantial penalty for death. Also, I have simplified the function by using a more straightforward condition logic to assign rewards based on the clear game outcomes: the agent's defeat, the boss's defeat, and the health change for both. This way, the agent could easily understand and optimise its strategy around. Here is the result of incorporating the updated reward function:

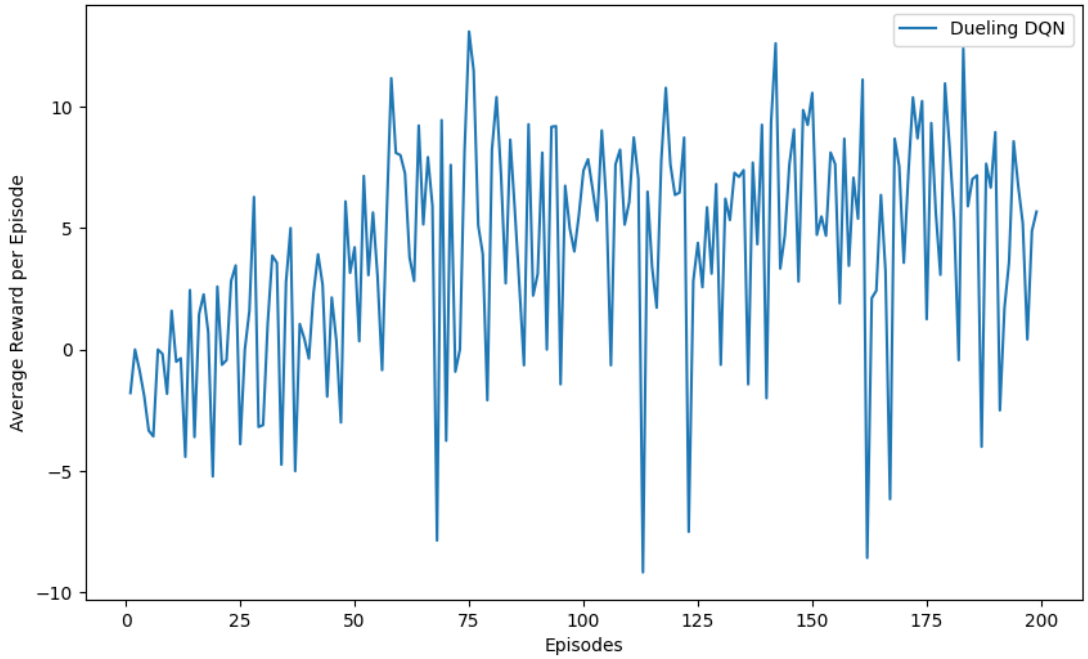

Figure 3.2: Average reward per episode using the updated reward function

I used dueling DQN to train the agent with a new reward function for 200 episodes (approx. 30000 frames). The observation of an increasing trend in the average reward per episode indicates the adjustment in the reward function is successful.

However, even though the updated reward function has shown improvement on the initial approach, there is still much room to explore in the reward function design. The ratio of positive rewards versus negative rewards is an example. The complexity of DRL learning environments indicates that even minor adjustments in the reward structure can significantly change the outcome. For this project, the ratio is four to one, which means the positive reward for defeating or causing damage to the boss is four times larger than the negative reward. Finding a ”sweet spot” for the ratio in complex environments like this project is highly challenging since reward isn’t the only factor that can affect the agent’s ability to learn.

3.0.4 loss function

The loss function is crucial to machine learning, evaluating how well our algorithm models our dataset. In deep learning, it measures the difference between the predicted output of the network and the actual data. This output is like the feedback given to the optimisation problem (Adam in this case) on how well the model performs at this

given training state. In the context of DQN, the loss function measures the difference between the predicted Q-values produced by the main neural network and the target Q-values estimated from the Bellman equation. One common loss function used in DQN is the Mean Squared Error (MSE) loss, which is the following formula:

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(s_i, a_i; \theta))^2$$

Where N is the size of the mini-batch, which in the context of experience replay is a subset generated from sampling the experience replay buffer. y_i is the Q-value produced by the target network; it is calculated as $y_i = r_i + \gamma \max_{a'} Q(s_i', a'; \theta^-)$ and $Q(s_i, a_i; \theta)$ is the predicted Q-value by the main neural network.

3.0.4.1 Huber Loss

In DQN, I implemented Huber loss as my loss function. The reason is that Huber loss combines the mean squared error function and the absolute value function, taking advantage of both functions. The Huber loss function is defined as a piecewise function:

$$L_\delta(a) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |(y - f(x))| \leq \delta, \\ \delta(|y - f(x)| - \frac{1}{2}\delta) & \text{otherwise.} \end{cases}$$

Where the top one is the MSE, and the bottom one is the MAE (also known as L1 loss) This function essentially means that for loss values smaller than a threshold δ, we use MSE, which is quadratic; otherwise, we use MAE, which is linear. MAE is designed for larger loss values to mitigate the weight we put on outliers, making the loss less sensitive to outliers than MSE. At the same time, MSE penalises minor errors quadratically, providing smoothness over L1loss near 0.

3.0.4.2 Entropy Bonus in PPO

During training, the agent consistently performs actions that give a small reward, such as defending or jumping on top of the boss (jumping on top of the boss does a small amount of damage to the boss), rather than attacking the boss directly. This means the agent has converged to a suboptimal policy [16]. To tackle this problem and promote a wider investigation of the possible actions, I implemented an entropy bonus into the loss function in the form of $entropy\_coefficient * entropy$ just as described in [16]. This strategy uses entropy, a statistical measure of randomness or unpredictability that

encourages the agent to explore a more diverse range of actions by rewarding higher entropy, or, in other words, penalising the agent less for exploring various behaviours. This bonus is added to the PPO loss as $cS(\pi)(s_t)$ just as described in the paper, where c is the hyperparameter I will experiment with. Entropy S is a measure of randomness in the agent's policy (denoted by π), and s_t denotes the state of the environment at time t. A higher entropy means the policy is more random, encouraging exploration because the agent is less likely to repeat the same action in the given state. On the other hand, low entropy means the agents tend to choose the same action repeatedly.

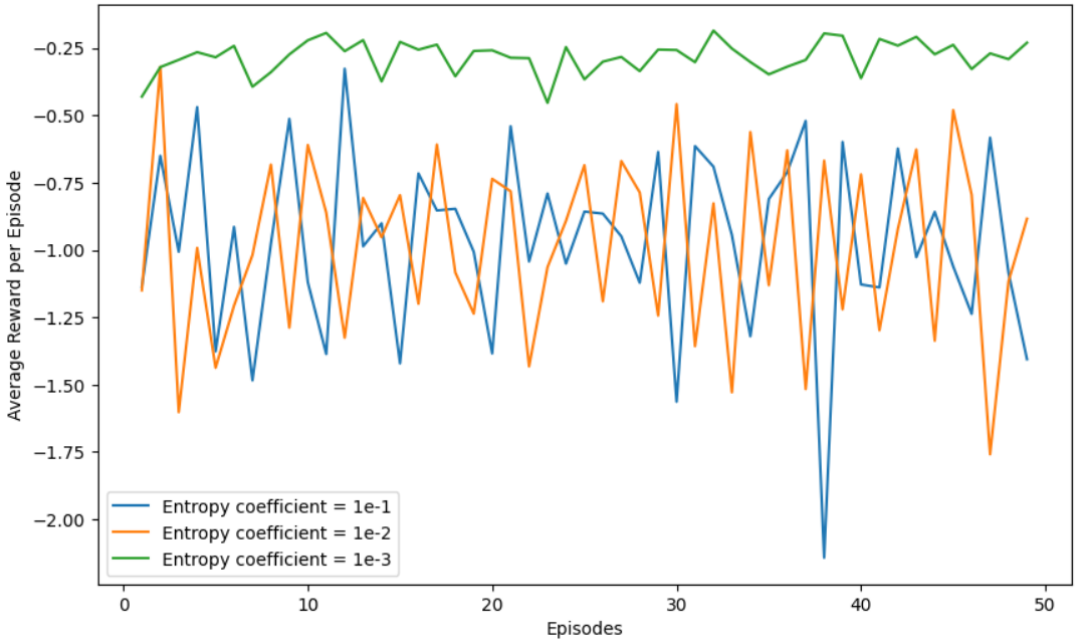

Figure 3.3: Average reward per episode with different entropy coefficient

The experiment comparing entropy coefficients of 0.1, 0.01, and 0.001 clarifies how finely-tuned exploration incentives impact the agent's learning and performance over time. The experimental results show that an entropy coefficient of 0.001 leads to the best performance in the early stages, as indicated by an increasing average reward per episode across 50 episodes. Higher coefficients of 0.1 and 0.01 result in a decrease in the average reward per episode. This pattern highlights an important observation: there is a crucial point in the value of the entropy bonus where encouraging exploration hinders learning efficiency and performance.

3.0.5 Adam

The optimisation algorithm I used in this project is Adam, the adaptive moment estimator presented in the paper [6]. It serves as the role of gradient descent for updating the parameters in the neural network, combining the advantages of the other two expansions: Adaptive Gradient Algorithm or AdaGrad and Root Mean Square Propagation or RMSProp. While traditional gradient descent uses a fixed learning rate, AdaGrad and RMSProp maintain a per-parameter learning rate. RMSProp further addresses the problem in AdaGrad by keeping a moving average of the square of gradients. Furthermore, Adam combines the benefits of both AdaGrad and RMSProp, it takes the idea of maintaining a per-parameter learning rate from AdaGrad and the idea of moving the average of square gradients to adjust the learning rate rather than accumulating all past squared gradients. It further enhances these two concepts by using the first and second moments to calculate the adaptive learning rates. The first moment is the mean of the gradients, which Adam uses to capture the direction and velocity of the gradient and helps accelerate the convergence. The second moment is the uncentered variance of the weights and is used to adjust the learning rate based on the recent gradient magnitudes. By updating the learning rate according to the second moment, it automatically scales down the step size for parameters with large gradient variance. It scales up the step size for those with small gradient variance, ensuring efficiency at handling different parameter scales.

3.0.6 Implementing PPO

3.0.6.1 Agent

In this section, I want to discuss the network architecture I implemented for PPO in this project, the hyperparameter selection, and the choice of layers.

The agent first starts by receiving a reshaped state from the environment. The reshape process sizes the original screenshot to 224x224x3 (224 width and height with three channels RGB). We want our input image to have the same size to ensure all inputs to the neural network are consistent, which is crucial for batching. We also like the input images to be squared since we use GPU for training the CNN, and GPU often performs better with squared images due to the way the memory and processing units are optimised.

3.0.6.2 Actor Network

The preprocessed state is fed into the actor network, which passes through a series of convolutional layers; the first layer convolves the image using 32 3x3 filters with stride number 1 and one layer of zero-padding. Relu is applied at the end of each convolutional layer to introduce non-linearity, enabling the network to learn more complex patterns in the image. The second layer then convolves the output of 32 feature maps from the previous layer using 64 3x3 filters, also with a stride number of 1 and one layer of zero-padding. Besides applying the ReLU activation function, I added one 2x2 max pool layer with a stride number of 2 after the activation layer. After that, the third layer takes in the 64 feature maps output from the previous layer and produces 128 feature maps with the same kernel size, number of strides, and zero-paddings. Next, the output tensor from the convolutional layers is flattened and fed into the dense layers (a sequence of fully connected layers) for further processing. The first layer takes the output feature vector of the convolutional layers and transforms it into a vector of length 512. A ReLU activation function follows this to introduce more non-linearity. Then, another fully connected layer transforms the vector with 512 dimensions into a vector of length n_actions, where n_actions represents the number of possible actions defined for the agent. Both transformations are achieved by a linear operation

$$output = input * W + b$$

where W and b are learnable parameters. Finally, a softmax function is applied to all the outputs of the fully connected layer to get a distribution over all the actions, with each value representing the estimated probability of taking that action from the network. An action is sampled from this distribution, which essentially selects the action with a higher probability of getting a better outcome.

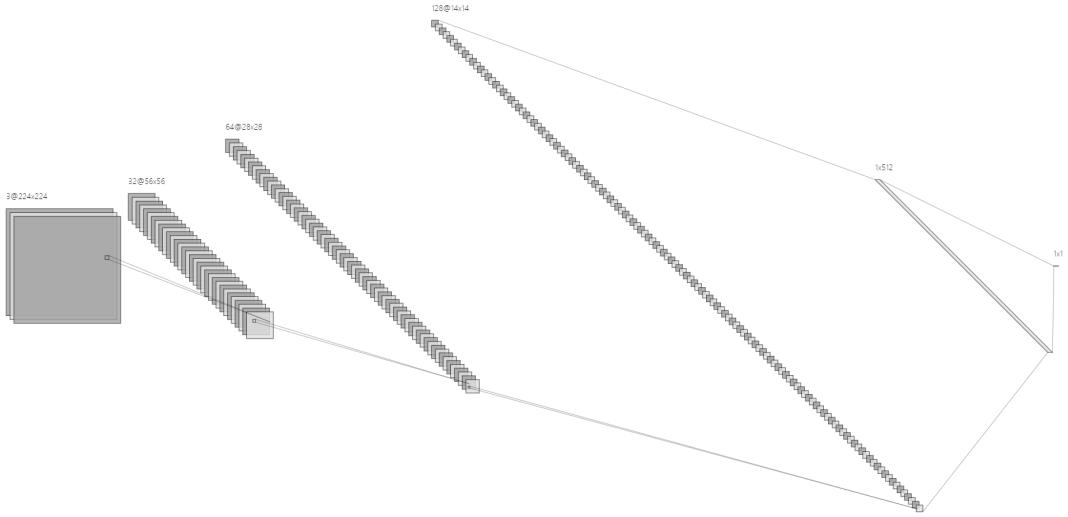

Figure 3.4: CNN architecture used in PPO

3.0.6.3 Critic Network

The critic network incorporates the same CNN structure as the actor network but differs in the dense layer architecture. While the actor network is responsible for choosing the actions, the critic network, on the other hand, evaluates the current state by predicting the expected returns (the total future rewards). So, the output of the dense layer in the critic network is a singular value, not a distribution over all actions. Since the output is a singular value, there is no need for a softmax layer to convert the distribution into probabilities.

3.0.6.4 Learning Process

The learning function incorporates a policy gradient method to update the actor and the critic network. It starts by calculating each step's advantage value in an episode. This advantage value is calculated by a technique called generalised advantage estimation (GAE). This step is crucial in policy gradient methods as it tells you how good an action is by comparing the advantage value of this action to the average advantage value under the current policy. It does this by keeping a nested loop. The outer loop iterates through every time step t in the episode except the last one. The inner loop starts from the timestep t, accumulating the discounted sum of temporal different errors

δ. The temporal different error at each timestep k is calculated by:

$$\delta_t = r_t + \gamma \cdot V(s_{t+1}) \cdot (1 - \text{done}_t) - V(s_t)$$

. Here, $\gamma \cdot V(s_{t+1}) \cdot (1 - \text{done}_t)$ calculates the discounted future rewards given that the next state is not terminal. The discount value is then updated by multiplying γ and λ; the discount factor γ prioritises immediate reward over future rewards. And the GAE parameter λ is used to control the bias-variance trade-off. γ is used to control how future reward is discounted in calculating the advantage at each timestep, while λ controls the weights for the rewards at each timestep in calculating the advantage.

For each batch of experiences, the algorithm extracts the states, old probabilities and actions into tensors and transfers them to the GPU for the actor network to use. The states are passed into the actor network, resulting in the distribution of all actions. We then calculate the log probability described in the PPO section above equation. Next, the ratio of the new probabilities of the actions taken under the current policy to the old probabilities is computed; this is used for the PPO clipping. The probability ratio is clipped in the $[0.8, 1.2]$ range to prevent excessive large updates. The clipped and unclipped ratios are multiplied by the advantage, and the minimum is taken to calculate the actor loss. The critic loss is calculated as the mean square error between the expected critic value, computed by passing the batch states into the critic network and the calculated returns, computed by adding up all the advantage values and critic values of each step in the batch.

3.0.6.5 Batch size

Using batches follows the same approach as OpenAI did in their DOTA2 project [16]. This approach shares similarities with the concept of experience replay in DQN. They both use the idea of collecting past experiences and processing them in parallel. This idea is rooted in the fundamental idea of stochastic gradient descent (SDG). In pure SGD, the model's parameters are updated based on a single point of data, which introduces much noise in the training process. This could help the model to get out of local minima, though it is inefficient and can lead to a very erratic convergence path. Mini-batch is a variation of SGD, which uses a subset of training data to compute the gradient. The use of mini-batch in SGD offers a trade-off between exploration and exploitation. Using a small batch is more like pure SGD, offers more exploration and can avoid getting into local minima. On the other hand, using a large batch is

more like the behaviour of full-batch gradient descent, where the gradient is computed on the whole dataset. The OpenAI paper describes mini-batches as the trade-off between convergence time and the number of optimisation steps [16]. This is true since larger mini-batches could lead to each epoch being processed faster due to the parallel processing nature of modern GPUs. But this comes with a drawback of worse performance than using smaller mini-batches. In this project, however, small batch sizes are used. This is because the training episodes are relatively small (see 3.1), which limits the diversity of experiences (with large mini-batches, one could end up with batches that contain the entire episode). I decided to focus on comparing how different batch sizes affect how fast the policy improves, which can show how fast the agent can start learning from the environment. The choice of batch sizes follows the same approach in the paper by doubling the smaller one.

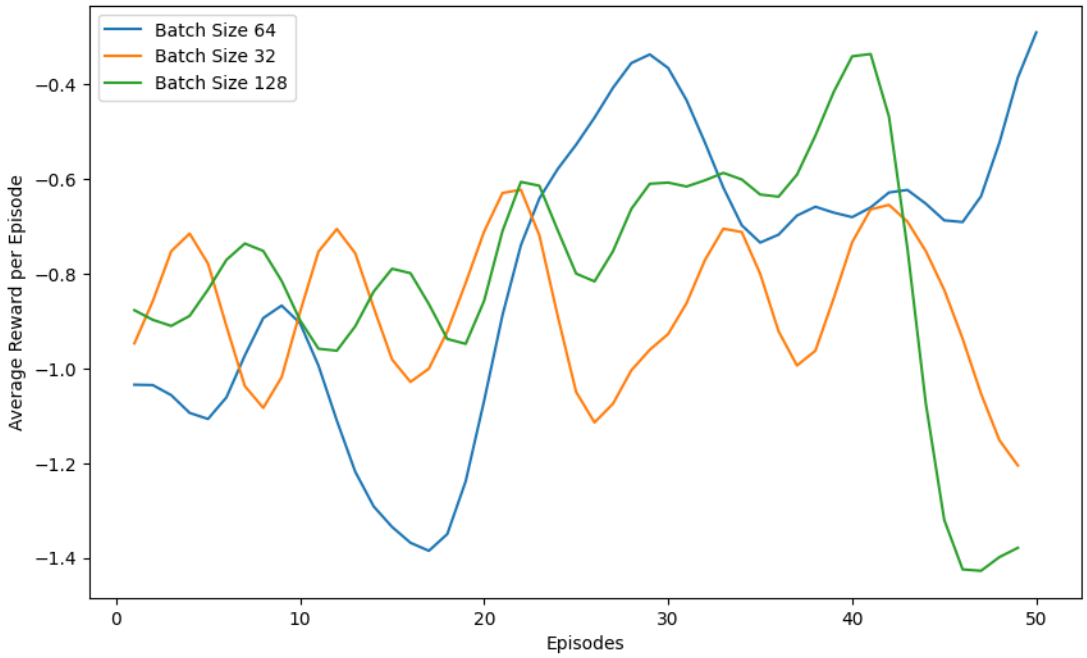

Figure 3.5: Batch Size (smoothed)

From the graph, batch sizes 64 and 128 yield an increasing trend in average reward per episode, while using batch size 32 is steady but not increasing. The hypothesis I want to make from this is: up to a certain point, the larger the batch size, the better the performance. I called this a hypothesis, but in reality, it is not. In the paper Character Behavior Automation using Deep Reinforcement Learning by H Lee et al. [11], they proved this by conducting tests on batch sizes up to 1024 in PPO. The result shows

that although smaller sizes (256 in their paper) might have a faster pace of learning in the early stages, the performance is worse than using a larger batch size.

3.0.6.6 ResNet Fine Tunning (an exploration)

To test out the ability of a pre-trained model to extract features in the environment, I decided to change the existing convolutional neural network to use a pre-train residual network model. To fit the model for this project, I have to fine-tune the pre-trained ResNet34 model. Firstly, I froze the weights of the pre-trained layers to prevent them from updating during training. By doing this, I retained the ability of the model to extract features that it learned from the large and diverse ImageNet dataset. Secondly, I unfroze the last three layers of ResNet, which allowed the model to adjust to the specific nuances of the new task, and improved its capability to identify patterns in the new environment. In the original model, the dataset used for training consists of 1000 classes. Because of this, the final fully connected layer is a vector of size 1000, representing each class's probability. However, in the case of PPO, we need a probability distribution over the feasible actions of the agent. So, I replaced the final fully connected layer of ResNet with my own fully connected layers. After training ResNet34 for 200 epochs, I discovered that these networks were less effective compared to my CNN. This could be due to the fact that my training environment is highly specific, or because the model has overfit or underfit the training data. Consequently, I decided to switch back to training my own CNN network from scratch.

3.0.7 Implementing DQN

In the DQN and Dueling DQN networks, I followed a similar approach to building the CNN network as in the original DQN paper [14] and made some changes, hoping to suit the game of this project better. The paper incorporates a CNN structure with two layers, with the first layer consisting of 16 8x8 filters and 32 4x4 for the second layer. This network structure might be suitable for Atari games but not for more detailed games like Sekiro. The paper recommends an input image size of 84 by 84, which is relatively small. This suggests that the task would benefit from capturing large spatial features from the input. However, since we need to focus on the details of the boss's movement, I increased the image input to 224x224. This increase in resolution allows for more detailed features to be present in the input. Also, I used 32 5x5 filters in the first layer and 64 5x5 filters, which increased the number of filters compared to the

original setup in the paper and decreased the filter size. Increasing the number of filters helps capture more diverse features, which is beneficial given the high-resolution input image. Also, a smaller filter size will focus on finer details in the input image without immediately abstracting too much information. Nevertheless, during training, I still observed that the agent constantly decided to choose the action 'jump'. Although this action yielded a small reward, it was not the optimal action preferred. Despite the reason discussed in the reward function section related to reward shaping, whether the neural network is deep enough can be another problem when discussing sub-optimal decision-making. This has resulted in a decreasing trend in Q value per episode and the average reward per episode.

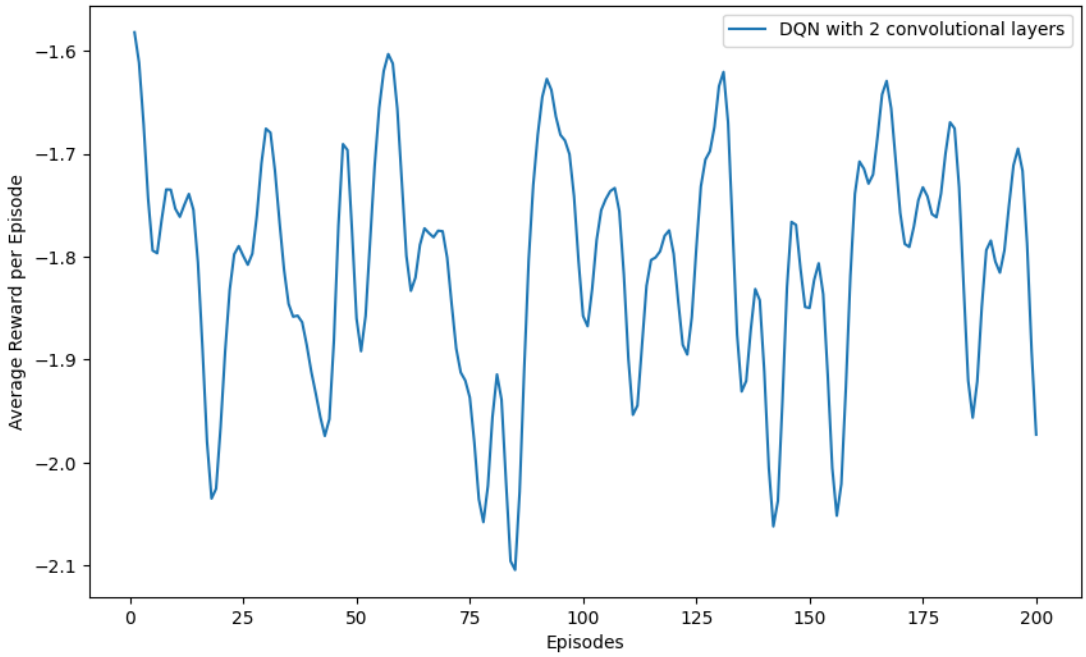

Figure 3.6: Average Reward per Episode

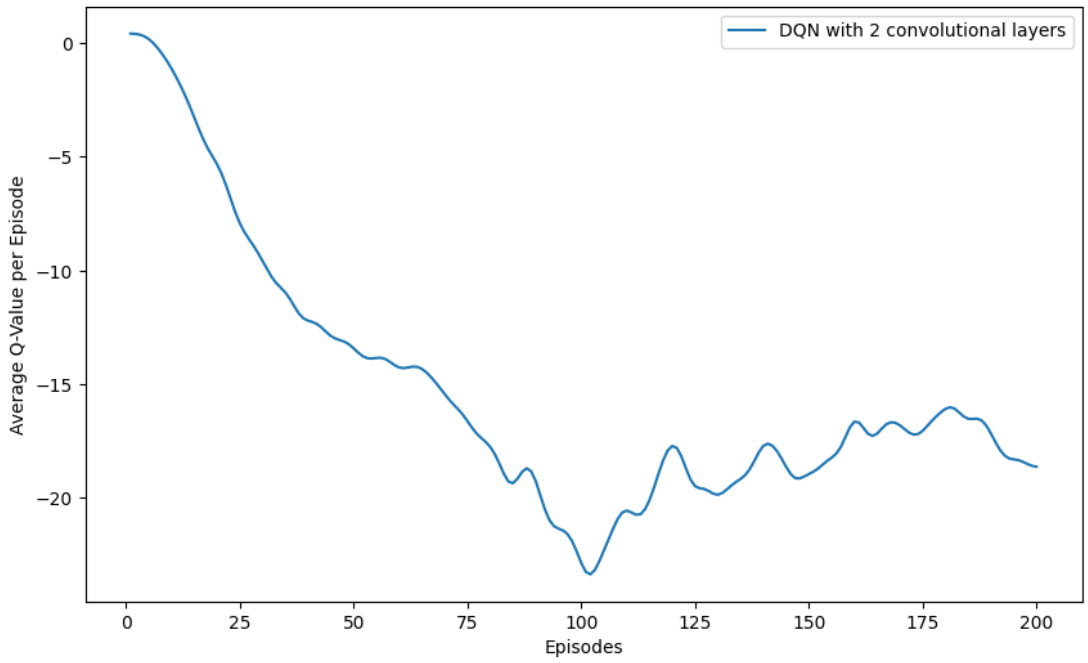

Figure 3.7: Average Q Value per Episode

One solution to this is to increase the depth of the neural network, which increases the network's capability of learning detailed features from the image input. Increasing the network's depth enables it to represent more complex functions, thus making it more possible to model the information needed from the high-dimensional image. I added two more convolutional layers to the original 2-layer architecture. Now, the third layer takes output from the previous layer, which is 64 channels, and convolves using 128 3x3 filters. Using a stride number of 1 preserves the spatial dimensions of the feature map. Finally, the fourth layer applies 256 3x3 filters with a stride of 1 on the output from the previous layer. I also used a decreasing kernel size approach, which means the features are learned hierarchically. The initial layer with a large kernel size captures the high-level features like edges and textures. The following layers with small kernels focus on more detailed and specific features. Here is the result of using a deeper CNN:

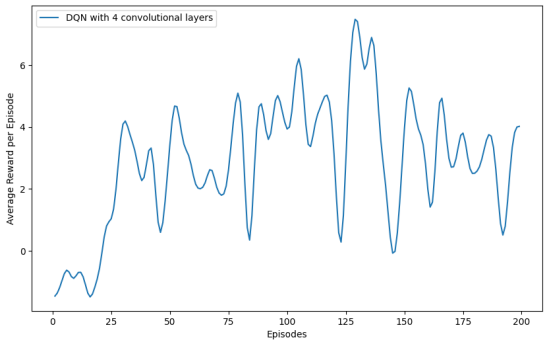

Figure 3.8: Average Reward per Episode

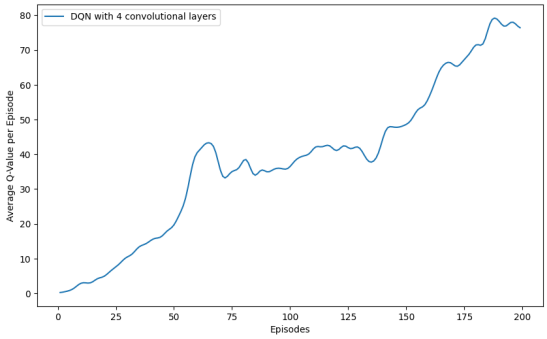

Figure 3.9: Average Q Value per Episode

3.0.7.1 Learning Process

During each interaction between the agent and the environment, several values are stored in the experience replay buffer, including the current state, the action chosen on that state, the reward of performing that action, the next state observed and the done flag, which indicates whether the state is terminal. When it is time to learn, the agent randomly samples a batch of experiences from the experience buffer. For each sampled experience, the agent calculates the Q value of taking an action in that state using the evaluation network. It also computes the maximum Q value for the next state using the target network. The target Q value is computed as the current observed reward plus the discounted maximum Q value of the next state. Next, Huber loss is used to calculate the loss between the predicted and target Q values. Then, the gradient of this loss is backpropagated through the evaluation network, and its parameters are updated using the Adam optimiser. The target network is updated once in a while. In this project, it is set to 200, which means the target network is updated every 200 steps.

3.0.8 Fram skipping in DQN

Following the same approach implemented in the original Deep Q-Network paper by Mnih et al., 2015, I implemented the technique of frame skipping to accelerate the learning process by simplifying the decision-making timeline. The core idea behind frame skipping is that the agent sees and selects an action at the first of a fixed number of frames, and this action is repeated for this fixed number of frames, during which the agent does not make any new action decisions. The rewards are accumulated for those frames. This would significantly reduce the computational demand and increase the training speed as the agent does not need to decide on each frame but rather at each k-th frame [14]; in my code, k = 4. It also reduces the number of state evaluations the neural network needs. Frame skipping is particularly useful in the game Sekiro: Shadows Die Twice and is used in many other 3D games as well, for example, Doom [10], which is a 3D First-Person-Shooting (FPS) game. In the context of Sekiro, the agent's action has effects that last for several frames, which is what these games typically refer to as "recovery."

Figure 3.10: 4 consecutive frames for an action

This is an example of the action "attack" and its following frames. The agent will need four frames to recover from the previous action and be able to perform the next one. During this recovery time, no decision is needed, so skip frames will perform the same action in those states and return the accumulated reward at the last state. This can efficiently reduce redundant decisions in frames where the agent cannot perform an action.

3.0.9 Implementing Dueling DQN

Dueling DQN differs from standard DQN by separating the estimation of state values and the advantages of each action. It uses the same CNN architecture as standard DQN, but after extracting the features, the stream is diverged. One stream estimates the value $V(s)$ of being in a state s regardless of the actions taken. The other stream

measures the advantage $A(s,a)$ of taking each possible action in the state. This advantage function measures how much better an action is compared to other actions from the same state. Dueling DQN then combines the two streams to produce the final Q value. This equation does it:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

The later part of the right-hand side compares the specific action to the average action at that state.

3.0.10 Epsilon-greedy

In DQN and Dueling DQN, I implemented the epsilon-greedy approach for action selection. It balances the exploration-exploit trade-off - what the agent already knows versus exploring a random action. Exploration can improve the agent's knowledge of each possible move, and it will get a head start on every move. On the other hand, exploitation uses a greedy approach to get the most reward by exploiting the agent's current action-value estimates (Q values in this case). Below is the strategy of action selection by epsilon-greedy:

$$\text{Action chosen} = \begin{cases} \max Q_t(a) & \text{with probability } 1-\varepsilon, \\ \text{random action} & \text{with probability } \varepsilon. \end{cases}$$

Here is the pseudocode of my implementation:

Algorithm 1 Epsilon-Greedy Action Selection for DQN Agent

```
1:  procedure CHOOSEACTION(state)
2:      if RandomNumber() > ε then
3:          Q values ← Q_eval.forward(state)
4:          Max Q value ← max(actions)
5:          action ← arg max(MaxQvalue)
6:      else
7:          action ← ChooseRandomlyFrom(action_space)
8:      end if
9:      return action
10: end procedure
```

3.0.10.1 Hyperparameters DQN & DDQN

Batch sizes, target network update frequencies, epsilon greedy decay rates and epsilon greedy minimal values are tested here. When training agents using dueling DQN, larger batches seem to have achieved higher performance than smaller ones. However, huge fluctuations have been presented in the average reward curve. Several factors can cause this; the agent might receive rewards in early training when exploration dominates. It can also be an issue with the target network, which updates frequently and can potentially cause fluctuations in the learned policy. (All these agents were trained using dueling DQN)

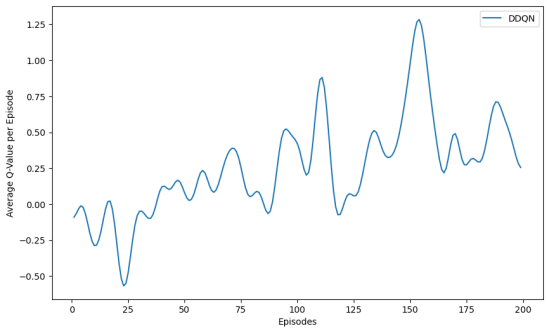

Figure 3.11: Average reward for DDQN batch size 128, trained on approx. 28000 frames (smoothed)

With batch size 128, I can safely say that the agent has learned from the environment, given the trend of average Q value and the convergence of loss value.

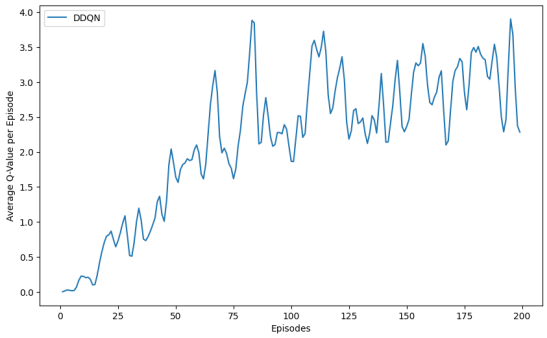

Figure 3.12: Average Q value for DDQN batch size 128, trained on approx. 28000 frames

The slight drop in average Q values in the later stage of the training process indicates that the refinement of the action values estimations is based on a deeper understanding of the environment. This means it realises the previous optimistic estimations are higher than what is achieved. Nevertheless, this observation is a positive sign indicating that the agent has been learning and adjusting its strategies based on the environment.

Here are the average rewards and Q values for batch sizes 16, 32, and 64. Results show that a batch size of 128 outperforms all other batch sizes.

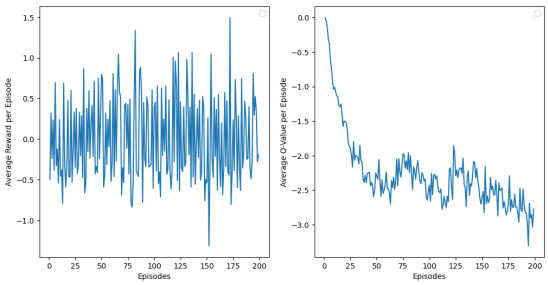

Figure 3.13: Batch size 16

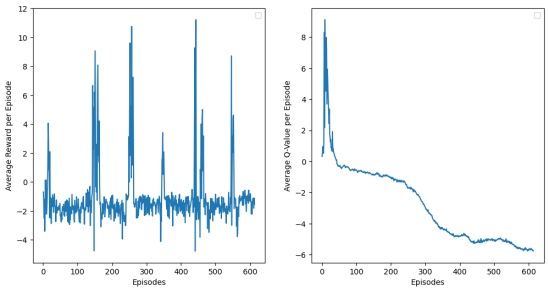

Figure 3.14: Batch size 32

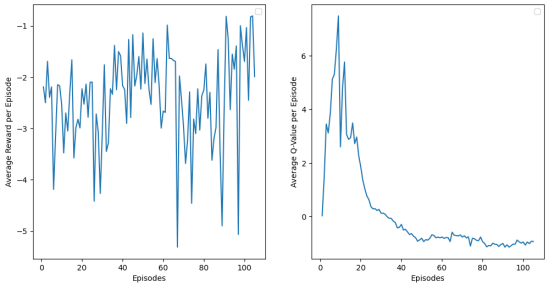

Figure 3.15: Batch size 64

The target network update frequency is crucial when discussing hyperparameters in DQN and variants. It determines how often the target network is updated.

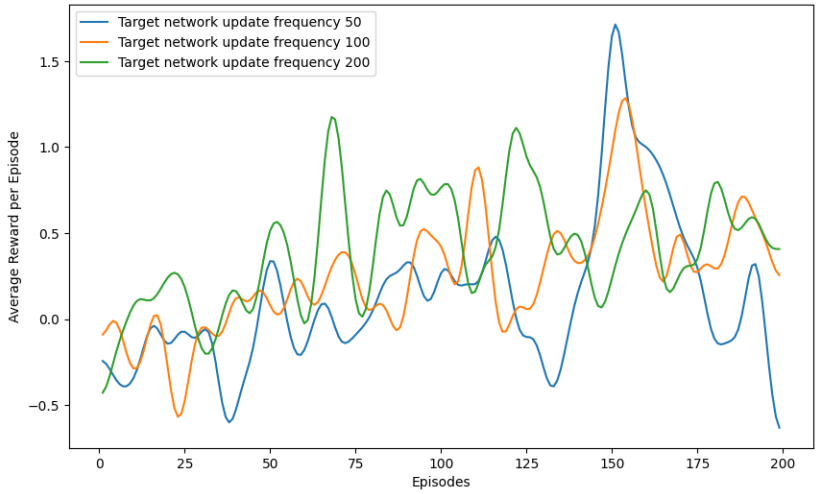

Figure 3.16: Target Network Update Frequencies

I tested three different update frequencies, starting with 50 and doubling each time for 200 episodes (approx. 30000 frames). From the results shown in the graph, it is shown that no single target network update frequency has demonstrated a consistent advantage over the others, indicating that within the tested range, this parameter may not be the primary determinant of the agent's success in learning useful information. However, the graph does show that with an update frequency of 50, there are fluctuations and some spikes in rewards, which indicates possible volatility in policy learning.

Other hyperparameters not tested individually are selected based on this paper: [3]. Despite the basic hyperparameters like GAE λ or PPO ε, the author also mentioned a few things worth investigating. The experiments were evaluated using five different robot control tasks from OpenAI gym. In the paper, the authors compared PPO loss with five other losses, and the result shows that PPO loss outperforms other losses on all five tasks. The authors also suggested using Adam as the optimizer after comparing its performance with RMSprop. Using a learning rate of 0.0003 has proven to perform well on all five tasks. Finally, the authors have compared several advantage estimators, including GAE, which is used in this project, and N-step and V-trace. The results show that GAE with $\lambda = 0.9$ performed the best. However, the author suggests not using Huber loss or PPO clipping, which is debatable when applied to this project.

Chapter 4

Evaluation

4.0.1 Training result

This chapter evaluates the performance of three algorithms in the game. It first presents their training results and then demonstrates their ability to fight the boss. The graphics card used for training these agents is a single NVIDIA 4070.

4.0.2 Dueling DQN

Firstly, the DQN network has been trained for 1000 episodes, equivalent to 98426 steps or 393704 frames.

Figure 4.1: Training result of DQN for 1000 episodes (average reward against training episodes)

This result shows that the agent has failed to find an effective strategy. So it would be necessary to replace DQN with DDQN.

The DDQN network has been trained for 1000 episodes, equivalent to 53335 steps or 213340 frames. The training was early stopped because I found out the average reward was going down over a number of episodes.

Figure 4.2: Training result of DDQN for 1000 episodes (average reward against training episodes)

The result indicates that the agent went through two distinct learning phases. In the first phase, the agent explored the environment, which led to a steady increase in the average reward. However, the second phase showed a decline in performance, suggesting that the agent had begun to rely on its learning policy.

The trend highlights the importance of exploring different hyperparameter settings and reward functions even after an effective initial training phase.

4.0.2.1 PPO

The graph below shows the training results of PPO; the model has converged as the average reward is not improving for many episodes. It has been training for 404 episodes or 245116 frames. After analyzing the data, I observed that the model's average rewards have levelled off. This suggests that the model has converged, as there is minimal improvement in the average reward over a substantial number of consecutive episodes. The average reward appears to fluctuate within a specific range, indicating

that the model's performance is stable. This stable oscillation suggests that the PPO model may have reached its learning capacity given the current environment and hyperparameter configuration. During the training process, the PPO model was more stable than Dueling DQN. This steady learning curve of PPO is due to its on-policy algorithmic nature that aligns policy updates with the expected value function.

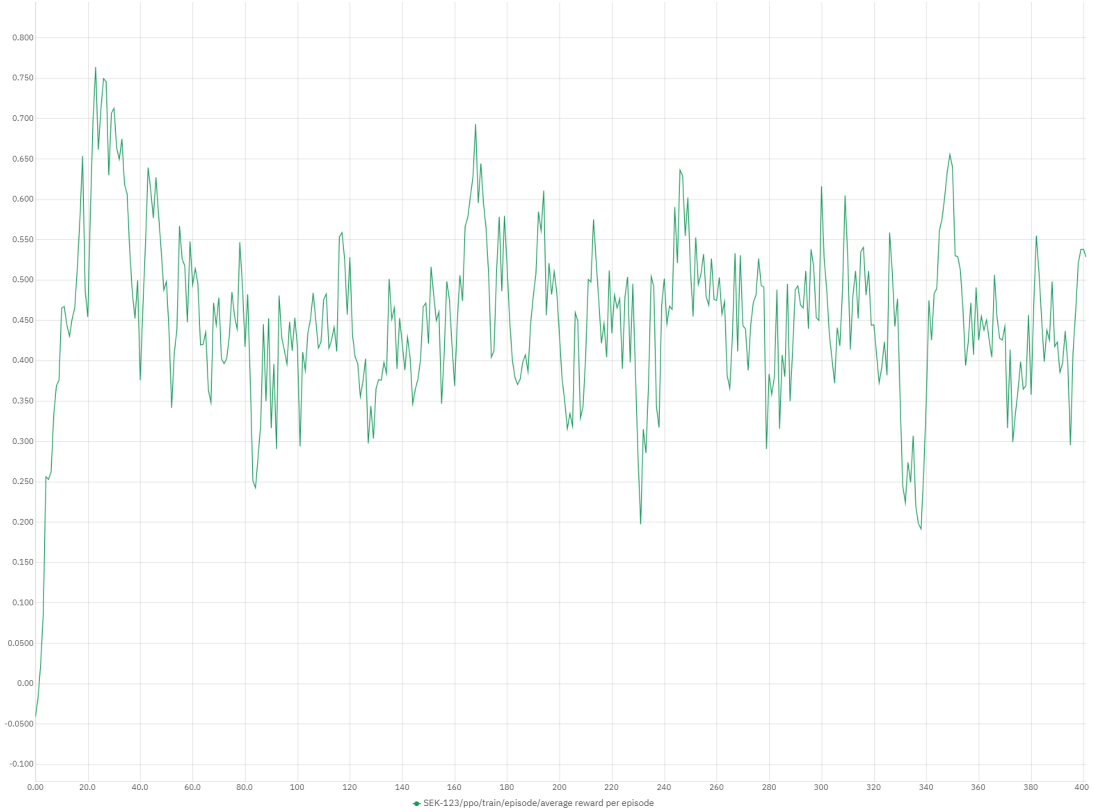

Figure 4.3: Training result of PPO for 404 episodes (average reward against training episodes)

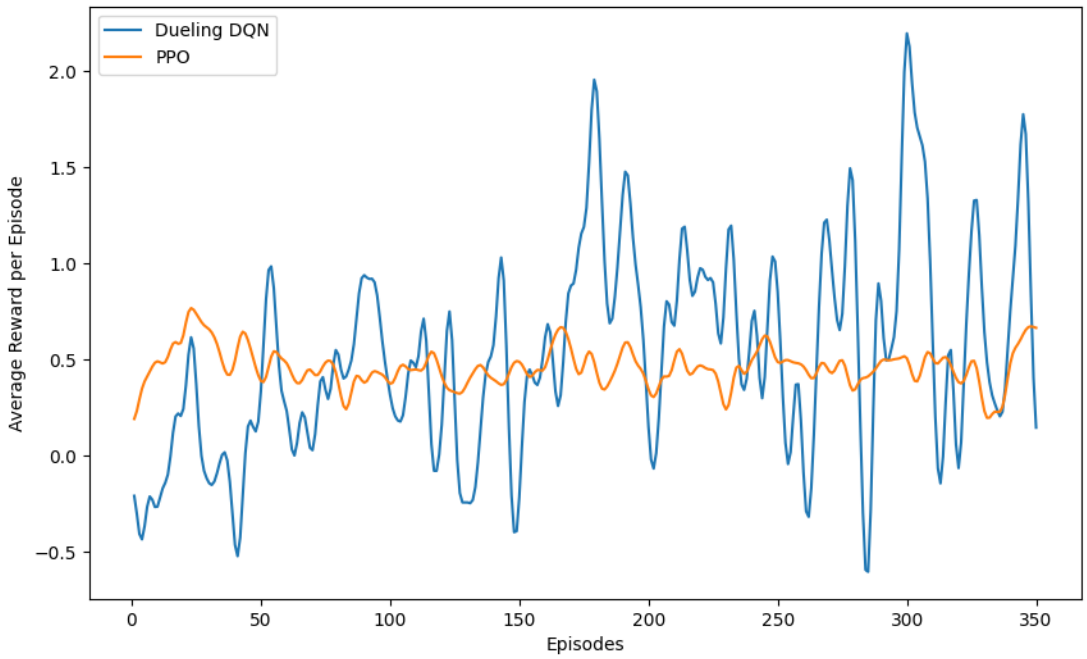

Figure 4.4: Dueling DQN vs PPO

4.0.3 Testing metrics

During testing, the agent has to fight the boss just like in the training phase, but the weight is fixed instead of updated. It is similar to training but without the learning part. The metrics showing the agent's performance include the average reward per episode and the kills-to-death ratio against the bosses. The kills-to-death ratio (KD) is calculated based on:

$$\frac{\text{Number of deaths of the boss}}{\text{Number of deaths of the agent}}$$

4.0.4 Testing result

The figure below shows the average reward of the agent trained on Dueling DQN against the boss it was trained on for 100 episodes. Convergence is found after 10 episodes, and the KD is 33%.

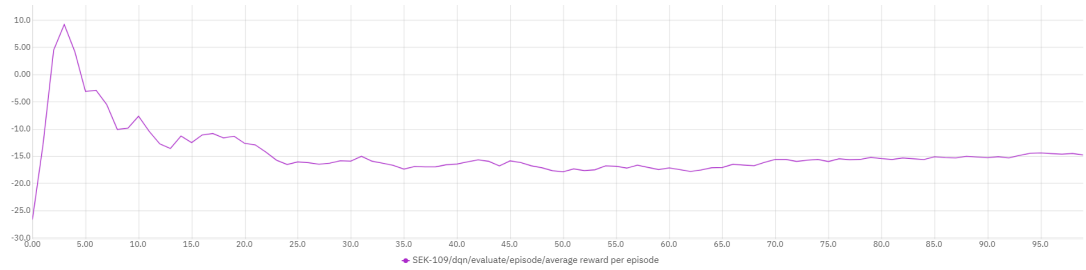

Figure 4.5: Testing result of DDQN (average reward against training episodes)

The chart depicts the average reward obtained by an agent trained on PPO for 100 episodes. The results show that the average reward achieved through PPO is noticeably higher than that of Dueling DQN. This conclusion has been established by the agent's impressive 1.36 KD against its trained boss. This means that the agent can defeat the boss 136 times with 100 deaths, which exceeds my expectations.

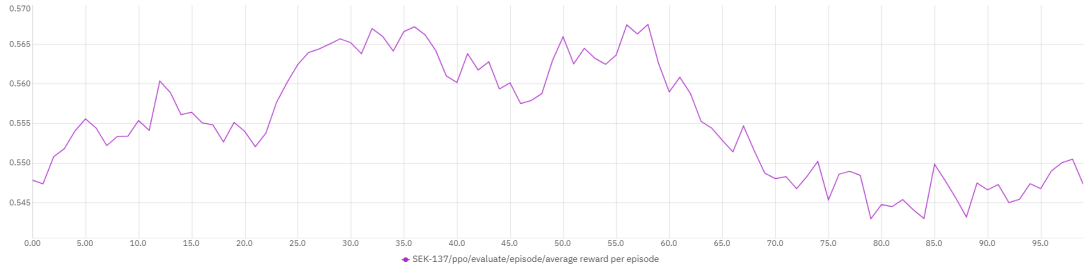

Figure 4.6: Testing result of PPO (average reward against training episodes)

The table below compares Dueling DQN and PPO on the agent's KD.

Table 4.1: Comparison of kills-to-death ratio between DQN, DDQN and PPO

Algorithm	DQN	DDQN	PPO
Boss (the agent trained on)	0.16	0.33	1.36

Chapter 5

Conclusion

In conclusion, this project successfully developed an autonomous agent that utilizes Deep Reinforcement Learning techniques, specifically Deep Q-Network (DQN) and Proximal Policy Optimization (PPO), to play video games at a level comparable to human players. I met the primary objective of enabling the agent to learn and adapt dynamically to game environments, as evidenced by its improved kill-to-death. The agent demonstrated significant learning progress, effectively employing strategies to defeat game bosses in various scenarios. The model trained using PPO achieved a KD of 1.36, showing the same level as a high-level human player.

However, there are several limitations to this project. Firstly, the model requires a significant amount of time for training, which limits the amount of iterative testing and refinement possible within the project timeline. Secondly, the agent's adaptability to completely unseen game scenarios was not as robust as hoped, indicating a need for improved generalization capabilities.

To address these limitations, further work could be done in the following areas:

Model Efficiency

Implementing more computationally efficient models or algorithms could reduce training times, allowing for more extensive experimentation and finer optimization. Also, parallelization could be used to speed up the time taken to train the agent.

Generalization

Enhancing the agent's generalization capabilities by introducing a broader range of training environments or employing techniques like meta-learning could prepare the

agent to adapt quickly to new challenges.

Alternative Algorithms

Exploring other reinforcement learning algorithms that might offer better performance and efficiency, such as Asynchronous Actor-Critic Agents (A3C) or hierarchical reinforcement learning.

5.0.1 Reflection

Throughout this project, I have appreciated the power of machine learning, particularly deep learning, in managing complex and ever-changing environments. However, I also acknowledge that the algorithms used in this game require further optimization. OpenAI Five proposed a new network structure specifically designed for the match Dota2, while the network structure I utilized was more of a general purpose. There is still much work to do to improve the agent's learning speed, which I will address in the future work section. I should mention that I did not have enough time to fully explore the potential of using transfer learning in this project. However, if there is a short training time and a limited number of episodes, it would be wise to incorporate transfer learning and fine-tuning instead of training a new Convolutional Neural Network from scratch. This would be more beneficial and efficient.

Also, as I have previously stated, the time required for training was quite lengthy. It would be advantageous to discover a more efficient approach to training. In addition to accelerating the game as I did in this project, other techniques, such as mixed precision training or utilizing GPU clusters, could aid in expediting the process.

5.0.2 Future work

The CNN network was built using existing research papers, but there is still much room to explore the network's architecture. The main focus is on the effect of adding or removing specific layers. Considering the complexity of Sekiro: Shadows Die Twice, it could be beneficial to add more layers to the CNN network as a good practice.

Incorporating Long short-term memory (LSTM) into the existing CNN networks can be an exciting point to experiment with to optimize them further. OpenAI used a 4096-unit LSTM in OpenAI FIVE, which played a massive role in deciding the next moves based on the history of moves. This can be a fascinating experiment, and while its benefits in MOBA games like Dota2 are clear, it is debatable whether it will benefit

this project. LSTM can help find a policy that maximizes long-term strategies, like managing resources and controlling territory, and short-term tactics, like the action made at each timestep. In combat games like Sekiro, long-term strategies are not often required, as the only goal is to avoid getting hit and hit the opponent. However, incorporating LSTM can help in real-game scenarios where the boss usually has more than one phase, which can require the agent to remember the tactics from the previous phase and adapt these strategies to the new phase.

Bibliography

[1] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[2] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-dujaili, Y. Duan, O. Al-Shamma, J. I. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8, 2021.

[3] M. Andrychowicz, A. Raichuk, P. Stanczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem. What matters in on-policy reinforcement learning? A large-scale empirical study. *CoRR*, abs/2006.05990, 2020.

[4] N. Elyasi and M. Moghadam. Tda in classification alongside with neural nets, 08 2020.

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[7] M. Kolobov, A. Kolobov, and M. Mausam. Planning with markov decision processes: An ai perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6, 06 2012.

[8] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[9] A. Kumar, J. Fu, G. Tucker, and S. Levine. Stabilizing off-policy q-learning via bootstrapping error reduction. *CoRR*, abs/1906.00949, 2019.

[10] G. Lample and D. S. Chaplot. Playing FPS games with deep reinforcement learning. *CoRR*, abs/1609.05521, 2016.

[11] H. Lee, M. K. Dahouda, and I. Joe. Character behavior automation using deep reinforcement learning. *IEEE Access*, 11:101435–101442, 2023.

[12] Y. Li. Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274, 2017.

[13] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. M. O. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[16] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[17] F. Osisanwo, J. Akinsola, O. Awodele, J. Hinmikaiye, O. Olakanmi, J. Akinjobi, et al. Supervised machine learning algorithms: classification and comparison. *International Journal of Computer Trends and Technology (IJCTT)*, 48(3):128–138, 2017.

[18] M. O. Riedl and A. Zook. Ai for game production. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, 2013.

[19] J. Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[20] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[21] M. Sewak. *Temporal Difference Learning, SARSA, and Q-Learning*, pages 51–63. Springer Singapore, Singapore, 2019.

[22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[23] R. Sutton and A. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998.

[24] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[25] Y. Wang, H. He, and X. Tan. Truly proximal policy optimization. *CoRR*, abs/1903.07940, 2019.

[26] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[27] C. Wei, M. Jafarnia-Jahromi, H. Luo, H. Sharma, and R. Jain. Model-free reinforcement learning in infinite-horizon average-reward markov decision processes. *CoRR*, abs/1910.07072, 2019.

[28] H. Y. Wenhao Chen, Wendi Chen. Rl-sekiro. `https://github.com/CWHer/RL-Sekiro`, 2022.

[29] R. C. Wilson, E. Bonawitz, V. D. Costa, and R. B. Ebitz. Balancing exploration and exploitation with information and randomization. *Current Opinion in Behavioral Sciences*, 38:49–56, 2021. Computational cognitive neuroscience.

[30] B. Xia, X. Ye, and A. O. Abuassba. Recent research on ai in games. In *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pages 505–510, 2020.

[31] C. Yoon. Dueling deep q networks - towards data science, Dec 2021.

Appendix A

Appendix Title

A.1 Hyperparamters used in this project

Table A.1: Hyperparameters

Choice Name	Default Value
input image width	224
input image height	224
input channels	3
action space	7
advantage_estimator	GAE
GAE λ	0.95
Value function loss	Huber
PPO-style value clipping ε	0.2
Policy loss	PPO
PPO ε	0.2
Discount factor γ	0.99
Frameskip	4
Optimizer	Adam
Adam learning rate	0.0003
Regularization type	None
Activation Function	ReLU