Comparing Four DQN Variants from the Field of Deep Reinforcement Learning on Atari Breakout

Bryan Carty (19235836)
Computer Systems (LM051)

April 24, 2023

This final year project report was completed in compliance with the FYP guidelines 2022-2023, under the supervision of Mr. J.J. Collins.

Contents

List of Figures

List of Equations

List of Tables

Acknowledgements

Firstly, I would like to thank my supervisor, Mr J.J. Collins, without whom this project would not have reached its full potential. His continued guidance and feedback has been instrumental in shaping this project, and developing my knowledge and skills within the discipline. Thank you J.J. Your support and encouragement has been greatly appreciated.

To my parents, I owe an enormous debt of gratitude. A significant portion of my academic and personal success can be credited to their selflessness and willingness to go above and beyond. I am very fortunate to have such amazing parents.

1 Introduction

1.1 Overview

Reinforcement learning is a type of machine learning where an agent learns to make decisions in a dynamic environment by trial and error [41]. The term 'reinforcement learning' appears to have been first introduced in the context of Artificial Intelligence by Minsky in 1954 and 1961 [26, 25, 42]. The reinforcement learning training method is based on rewarding desired behaviours and punishing undesired ones [9]. The goal of a reinforcement learning agent is to learn a strategy (policy) that maximises its cumulative reward [41]. Q-learning is a popular reinforcement learning algorithm, first introduced by Chris Watkins in his 1989 Ph.D. thesis 'Learning from Delayed Rewards' [48].

Learning to ride a bike can be viewed as an example of reinforcement learning. Initially, the learner (agent) has no knowledge of how to ride the bike (strategy/policy). If they successfully balance on the bike and move forward, they feel a sense of satisfaction (positive feedback/reward). If they lose balance and fall off the bike, they feel frustrated (negative feedback/penalty). Over time, through trial and error, the learner (agent) adjusts their behavior (policy) to maximize the sense of satisfaction (positive feedback/reward) they feel. Reinforcement Learning is designed to solve problems that can be modelled as sequential decision processes [2]. Learning to ride a bike is a sequential decision process as every action taken by the learner (agent), transitions the environment (bike) to a new state i.e. applying pressure to the right pedal affects the bikes velocity and tilt.

Sequential decision processes are typically modelled using a mathematical framework known as Markov Decision Processes (MDPs). A Markov Decision Process captures the interactions between an agent and its environment, incorporating the agents decision-making strategies and the environments response to actions [5]. Figure 1 depicts an example Markov Decision Process, adapted from "Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow" by Aurélien Géron [9].

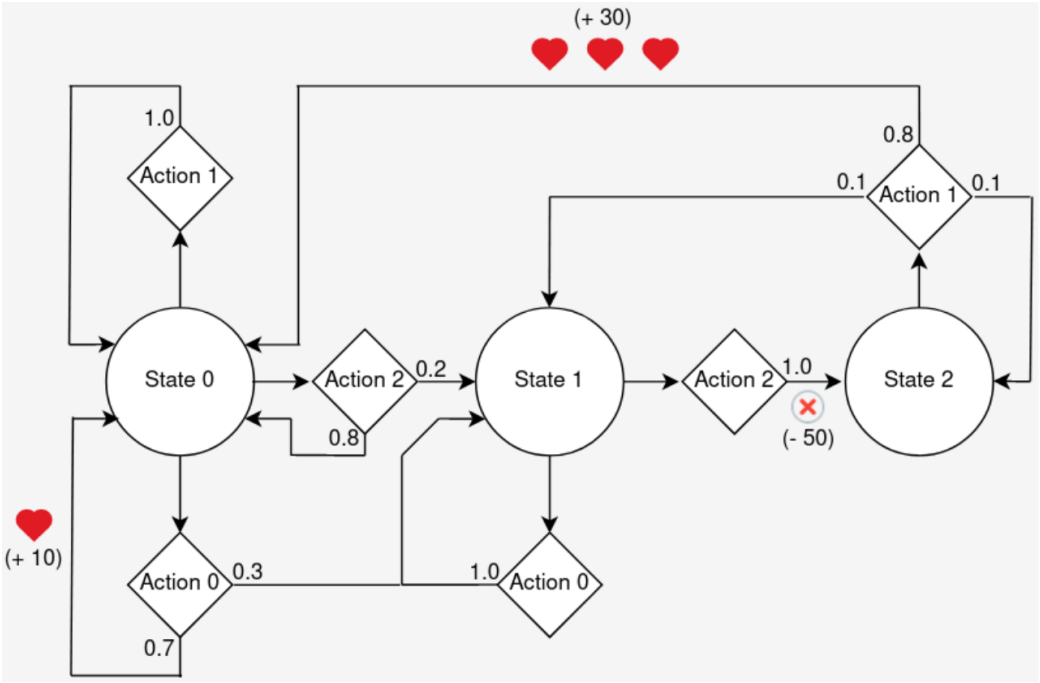


Figure 1: Markov Decision Process Example, adapted from "Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow" [9] by Aurélien Géron.

If the agent starts at *State* 0, it can choose *Action* 0, *Action* 1, or *Action* 2. If it chooses *Action* 1, it will stay at *State* 0 with 100% probability. If it chooses *Action* 0, it has a 70% probability of gaining a reward of +10 and returning to *State* 0, and a 30% probability of going to *State* 1 with no reward. If it chooses *Action* 2, it has a 80% probability of returning to *State* 0 and a 20% probability of proceeding to *State* 1. The agents goal is to find a strategy (policy) that maximises the amount of reward it receives. To do so, the agent must balance the exploration of alternative actions with the exploitation of current knowledge [9, 41] (Exploration vs Exploitation, Section 2.3.3). Additionally, as rewards can be sparse and delayed, the agent must appropriately assign credit to the action(s) responsible for a given outcome [9, 41] (The Credit Assignment Problem, Section 2.3.4).

Q-learning is a popular reinforcement learning algorithm, first introduced by Chris Watkins in 1989 [48]. Q-learning learns to estimate the optimal action for any given state in an environment i.e. it learns the optimal Q-

value function Q(s,a) for a Markov Decision Process. Q-learning is said to be model-free, meaning it does not rely on an explicit model of the environments dynamics (state-transition probabilities and reward function) to make decisions [48]. Instead it learns a Q-value function by interacting with the environment. Q-learning is also said to be an off-policy algorithm, meaning the strategy (policy) being learned is different from the strategy (policy) used to generate the data [48].

The Q-learning approach presented by Chris Watkins relied on a table to store the Q-values. However, this tabular approach is limited by the size of the problems state-action space. Consider the game of Go, which has approximately 10^{172} states, requiring a Q-table approximately 10^{172} columns wide. To put this number into perspective, there is estimated to be approximately 10^{80} atoms in the observable universe. Deep Q-learning solves the limitations of tabular Q-learning by employing a deep neural network to approximate Q-values. DeepMinds AlphaGo [39] beat world Go champion, Lee Sedol, in 4 out of 5 games in 2016 by employing deep neural networks [14].

This project is primarily a research-based project with the goal of comparing the Deep Q-network (DQN) and variant architectures on Atari Breakout. Atari Breakout has been a popular benchmark in reinforcement learning research since being used by DeepMind in their 2013 paper 'Playing Atari with Deep Reinforcement Learning' [28] and 2015 paper 'Human-level Control through Deep Reinforcement Learning' [29]. Similarly to the Deep Q-networks employed in these papers, the Deep Q-networks compared in this project will take an 84 x 84 preprocessed game frame as input, and output the Q-values for each legal action. While this project will touch on the concept of tabular Q-learning, first introduced by Chris Watkins in 1989 [48], it will place a primary emphasis on Double Deep Q-learning [29], Dueling Deep Q-learning [47], Deep Q-learning with Prioritized Experience Replay [36], and DeepMellow Q-learning [13]. This project will investigate the concepts that underlay these variants, and outline the architectures similarities, differences, pros, and cons. Four Atari Breakout agents will be trained, each employing a different Deep Q-network architecture. The agents performance will be analysed and conclusions will be drawn as to how the techniques employed either hindered or facilitated superior performance.

1.2 Research Objectives

The research question can be summarised as follows:

- **Research Question:** How does the performance of four distinct Atari Breakout agents, each utilizing a unique variant of the Deep Q-network, compare to one another?

The Deep Q-network architectures that will be compared in this final year project are the Double Deep Q-network [46], Dueling Deep Q-network [47], Dueling Deep Q-network with prioritized experience replay [36], and Deep Mellow Q-network [13].

As outlined in section 1.3.4, the Deep Q-network architectures will be compared based on several criterion, such as kernel visualizations, reward, and loss. These comparison criteria will serve as the basis for evaluating the performance and effectiveness of the different architectures. To ensure result validity, each variant will be trained twice and compared using the Mann-Whitney U-test [24].

The widespread adoption of the Atari Breakout environment as a benchmark in deep reinforcement learning research [29, 28, 13, 47, 36] played a significant role in its selection for this final year project.

In order to successfully answer the primary research question, a crucial secondary objective must first be met:

- **Secondary Objective:** Acquire a strong understanding of the foundations of artificial intelligence, machine learning, reinforcement learning, and deep reinforcement learning.

The systematic literature review approximation, outlined in Chapter 2, is responsible for meeting this secondary objective.

1.3 Methodologies

This section outlines the methodologies followed to ensure the successful execution of this project.

1.3.1 Systematic Literature Review Approximation

To ensure the prototype development and experimentation tasks were grounded in existing research, a systematic literature review approximation of artificial intelligence, machine learning, and reinforcement learning relevant texts and research papers was first conducted. Google Scholar, arXiv, and IEEE Xplore were electronic databases employed during the search process. Search terms used were "Artificial Intelligence", "Machine Learning", "Reinforcement Learning", "Deep Reinforcement Learning", "Deep Q-network", and "Deep Q-learning". The inclusion criteria were articles and research papers that discussed at least one fundamental machine learning or reinforcement learning concept, or one of the four DQN variants compared in this final year project. Educational video content and other online resources, such as framework documentation, were also queried where applicable. "Reinforcement Learning: An Introduction" [41] by Andrew Barto and Richard S. Sutton, and "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" [9] by Aurélien Géron were two texts heavily relied upon when gaining knowledge of machine learning and reinforcement learning fundamentals.

The systematic literature review approximation can be broken down into five topics, as outlined below:

1. The Foundations of Artificial Intelligence and Machine Learning: History, Semantics, PSSH, POE Model, Machine Learning Approaches.

2. The Fundamental Principles and Approaches of Reinforcement Learning: Terminology, Exploitation vs Exploration, The Credit Assignment Problem, MDPs, The Bellman Equation, Dynamic Programming, GPI, Monte Carlo Algorithm, Temporal Difference Learning, Q-learning.

3. Deep Learning: Threshold Logic Unit, Perceptron, Multilayer Perceptron, Backpropagation, Hyperparameters, Gradient Instability, Batch Normalization, Convolutional Neural Networks.

4. Deep Q-learning and Variants: Rationale, Vanilla Deep Q-learning, Double Deep Q-learning, Dueling Deep Q-learning, Prioritized Experience Replay, Network Instability, DeepMellow Q-learning.

5. Atari Breakout and Statistical Significance: OpenAI, History, Rationale, Mann-Whitney U-test.

1.3.2 Prototype

While there are already studies comparing several of the architectures examined in this project, no research was discovered that directly contrasts all four. Specifically, research examining the DeepMellow architecture was found to be insufficient. By implementing all the architectures in this project, external factors that might influence performance could be maintained at a consistent level. Moreover, implementing all architectures facilitated the validation of existing research, and the expansion on recent findings. Establishing a basis for future research, and fostering transparency were other motivating factors for prototype development.

In addition to discussing the main implementation components that comprise the variant architectures, Chapter 3 provides a comprehensive overview of the high-level architectural decisions, equipment employed, and problems encountered during experimentation. Chapter 3 can be broken down into four topics:

1. High Level Architecture Decisions: Framework, Program Structure, Implementation Medium.

2. Code Implementation: Atari Breakout Environment, DQN Architectures, Recording Training Progress.

3. Equipment Employed.

4. Problems Encountered: GPU Driver Compatibility, Memory Leak.

1.3.3 Data Collection

The following Atari Breakout reinforcement learning agent training data was recorded regularly during training:

- **Kernel visualizations** : Six kernels in the first convolutional layer were recorded every 1,000 games played. Every 1,000 games, the weights were retrieved from the first convolutional layer, plotted as a greyscale matplotlib plot, converted to a PNG image, and recorded in a Tensorboard events file.
- **Reward** : The average episode reward was recorded every 10 games played.
- **Loss** : The average episode loss was recorded every 10 games played.
- **Video** : Every 1,000 games played, the agent interacted with a test environment. The environment frames were recorded in an array, converted to a Python Image Library image, stacked, and saved as a GIF video.
- **Logs** : Logs were recorded every 100 games played. A single log consisted of the time, game number, frame number, average reward, and the replay buffer capacity.

Each Atari Breakout agent was trained on 25 million frames. Each agent was trained twice to ensure result validity. The OpenAI Atari Breakout Gym environment was used to simulate the Atari Breakout game. The max episode length was set to 18,000 frames. Running at 60 fps, 18,000 frames corresponds to 5 minutes of gameplay time.

1.3.4 Data Analysis

- **Kernel visualizations** : The kernel visualizations gave insight into the effects of gradient descent. Kernel visualizations were compared using the root mean squared error between images.
- **Reward** : Reward fluctuations were visualised using a Tensorboard graph. From which, convergence, divergence, training stability, ability to generalize, and duration could be concluded. Tensorboard also plots a (smoothed) reward, deduced by applying a technique known as Exponential Moving Average (EMA).

- **Loss** : Loss fluctuations were also visualised using a Tensorboard graph. Similarly to the reward plot, the loss plot allowed convergence, divergence, training stability, ability to generalize, and duration to be evaluated. Tensorboard also plots a (smoothed) loss, also deduced by applying the Exponential Moving Average (EMA) technique.

1.4 Contributions

This project makes two main contributions to the field of Deep Reinforcement Learning.

1. This project is the only apparent research explicitly comparing the Double Deep Q-network, Dueling Deep Q-network, Dueling Deep Q-network with prioritized experience replay, and DeepMellow Q-network.

2. As the DeepMellow Q-network is a novel network architecture, this project contributes to the very limited research investigating its practical use.

1.5 Motivation

I was drawn to Q-learning over other reinforcement learning algorithms because I find its off-policy nature fascinating. How it can learn an optimal policy by watching an agent act sub-optimally. I think Aurélien Géron put it well when he said,

"Imagine learning to play golf when your teacher is a drunk monkey" [9]

But my broader reasoning is because I want to be part of the innovation this field could bring about in the coming years. I hope to pursue a masters degree in computational biology and am particularly excited about how reinforcement learning and machine learning in general can contribute to medicine. A quote by Alan Turing comes to mind,

"We can only see a short distance ahead, but we can see plenty there that needs to be done" [44]

2 Literature Review

This chapter presents the findings of the approximated systematic literature review. At the start of this chapter, the reader is presented with a comprehensive overview of the fields of Artificial Intelligence and Machine Learning. Focus then shifts to the topic of reinforcement learning, where its key concepts, terminology and approaches are outlined. With an understanding of reinforcement learning, the rationale behind Deep Learning, Deep Q-learning and the variant DQN architectures is introduced. Finally, the Atari Breakout environment and Mann-Whitney U-test are discussed.

2.1 Artificial Intelligence

On August 31 1955, the term 'Artificial Intelligence' was coined by John McCarthy in a proposal for a two month, ten man study into Artificial Intelligence [21]. The workshop took place one year later in July and August 1956, now considered the birth date of the field.

In the years that followed there were several waves of optimism and disappointment with loss of funding. These periods came to be known as AI winters. The first AI winter spanned from 1974 to 1980. The primary cause of which was a 1973 report now known as 'The Lighthill Report'. The objective of this report was to evaluate the state of AI research in the United Kingdom. The report highlighted the failure of AI to meet its "grandiose objectives" which ultimately resulted in many institutes and organisations withdrawing investment. The second AI winter spanned from 1987 to 1992. One of the factors that led to this second AI winter was the oil crisis that once again resulted in lack of funding for AI. But each AI winter was followed by new approaches, success and renewed funding. We currently sit in the longest AI summer to date. This AI summer we're seeing AI thrive due to vast improvements in computing power, the availability of large and diverse datasets, and new techniques like deep learning.

There has been some difficulty in defining what artificial intelligence is, largely due to intelligence being such an abstract concept. We can generally agree that intelligence is a measure of some mental ability, but 'mental ability' can describe a broad range of things. In Alan Turing's 1950 paper

“Computing Machinery and Intelligence” [44], he proposed ”The Imitation Game” as a measure of intelligence, which later became known as the ”Turing Test”. In this paper he stated that a computer would deserve to be called intelligent if it could deceive a human into believing that it was human under some conditions. American philosopher John Searle argued against the Turing Test. He demonstrated this with a thought experiment now known as the 'Chinese Room Argument', first published in a 1980's article [37]. In this thought experiment Searle imagines himself in an empty room with an instruction manuel and a selection of Chinese characters. Outside the door is a Chinese speaker who passes Chinese questions wrote on paper under the door. Searle uses the instruction manual to formulate adequate responses that he then returns under the door. Despite Searle not knowing how to speak Chinese, it appears as though he does to the person outside the room. This raises the question, is a computer really thinking or is it just following instructions. Then again, it could be argued that that's all humans and other organisms do at the most fundamental level. Following Turing's paper, the term 'Artificial Intelligence' was used to describe machines that mimic and display ”human” cognitive skills i.e. skills that are associated with the human mind, such as ”learning” and ”problem-solving”. But this definition was later rejected by many AI researchers who prefer to describe artificial intelligence in terms of rationality, which does not limit how intelligence is articulated.

Initially, the physical symbol system hypothesis (PSSH) [30] was a highly regarded stance towards artificial intelligence. It was believed that:

”A physical symbol system has the necessary and sufficient means for general intelligent action.” [30]

A physical symbol system is a system that takes physical patterns or symbols as input, combines them into expressions, and manipulates them to produce new expressions. But Rodney Brooks of MIT later stated problems he had with this approach in his paper ”Elephants don't play chess” [6]. Brooks believed that explicit representations or symbols of the world would inhibit progress and that representation is the wrong unit of abstraction in building the bulkiest part of intelligent systems.

The concept of connectionism can be traced back to 1943 in a paper titled

"A logical calculus of the ideas immanent in nervous activity" by Warren McCulloch and Walter Pitts. [23]. In this paper they proposed a model of a neuron called the 'Threshold Logic Unit' and explained how neural activity in the brain can be represented using mathematical models. But it wasn't until the 1980s when texts such as "Parallel distributed processing: explorations in the microstructure of cognition" [22] by Rumelhart and McClelland established the concept of connectionism as a viable alternative to symbolic artificial intelligence systems.

Today, artificial intelligence can be thought of as a wide-ranging branch of computer science concerned with building smart machines capable of performing tasks that typically require biological intelligence. Domains such as artificial neural networks and evolutionary computation rely on the POE model of intelligence [19]. The POE model of intelligence considers three levels of organisation associated with life:

- **Phylogeny:** A hallmark example of Phylogeny is the evolution of a species. A species reproduces, producing offspring largely unchanged from the parent. The slight differences or mutations present give rise to the emergence of new organisms. If these new organisms are better suited to the environment, they will survive and prevail, becoming the more dominant iteration of the program.

- **Ontogeny:** Ontogeny refers to the development of an organism while phylogeny refers to how the organisms have evolved. In multicellular organisms, the successive division of the zygote is followed by a specialization process whereby the daughter cells specialize according to their surroundings. Similarly, as a machine learning model is trained on more data or exposed to new environment states, it becomes more adept at recognizing patterns and can make more accurate classifications or predictions.

- **Epigenesis:** Epigenesis is associated with the biological nervous, endocrine and immune systems. As an organism interacts with their environment, these systems are subject to beneficial modification. With respect to machine learning, it refers to the ongoing adaptation or modification of a model after it has been trained. Transfer learning can be regarded as a form of epigenisis. In transfer learning, a trained model is used as the starting point for the training of another model. It is believed that the

pretrained model has learned beneficial internal representations that only require fine tuning to be suitable for the new task. Online learning could also be regarded as a form of Epigenesis as in online learning the model continuously adapts to new data.

The field of artificial intelligence has been the topic of many a philosophical debate as the definition assumes biological intelligence can be so precisely defined that a machine can simulate it.

Today, 'artificial intelligence' is a household term and is utilized in many consumer products such as TikTok, YouTube, Snapchat (Recommender Systems), Siri, Alexa (Speech Recognition), Tesla (Self Driving Cars). My preferred definition for intelligence is the one that Wikipedia provides [49]:

"The ability to perceive or infer information, retain it as knowledge to be applied towards adaptive behaviours within an environment or context" [49].

The Rodney Brooks definition for artificial intelligence is:

"if a behaviour is deemed as intelligent by an external observer, then the underlying mechanism that generated that behaviour is from the field of AI"

The following section will focus on a well known subcategory of Artificial Intelligence, Machine Learning. Machine Learning can be thought of as the parent category of reinforcement learning, which can be thought of as the parent category of Q-learning, the primary focus of this project.

2.2 Machine Learning

Machine Learning is a subcategory of Artificial Intelligence, concerned with building algorithms to imitate the gradual human learning process. Machine learning often leverages large amounts of data.

Aurélien Géron referred to machine learning as:

"The science (and art) of programming computers so they can learn from data." [9]

Arthur Samuel referred to machine learning as:

"The field of study that gives computers the ability to learn without being explicitly programmed." [35]

A more engineering-oriented definition was formulated by Tom Mitchell:

"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E." [27]

There are largely 4 main categories of machine learning:

- **Supervised:** Supervised learning can be defined by its use of labelled data sets to train algorithms to classify data or predict outcomes. A supervised learning agent converges to a function that maps inputs to target outcomes. It does so by adjusting the model weights until it's fitted appropriately. A successfully trained model can then be applied to unseen data and return a prediction. Supervised learning algorithms can further be defined as classification or regression algorithms.

  Sutton and Barto define supervised learning as:

  "learning from a training set of labeled examples provided by a knowledgeable external supervisor. Each example is a description of a situation together with a specification the label of the correct action the system should take to that situation, which is often to identify a category to which the situation belongs. The object of this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set." [41]

- **Unsupervised:** Unsupervised learning uses algorithms to analyze and group unlabeled data. These algorithms can discover hidden patterns or groupings. Unsupervised learning is used for three main tasks - clustering, association and dimensionality reduction.

- **Semi-Supervised:** Semi-Supervised learning is a machine learning technique that combines supervised and unsupervised learning techniques.

It uses a small portion of labeled data and a lot of unlabeled data to train a predictive model. In Semi-Supervised learning, you train an initial model on a few labeled samples and then iteratively apply it to a greater number of unlabeled data. Neither supervised or unsupervised learning algorithms are able to make effective use of a mixture of labeled and unlabelled data.

- **Reinforcement:** Reinforcement learning is concerned with how agents take actions in an environment in order to maximize a cumulative reward. Q-learning is a model free reinforcement learning algorithm that learns the value of taking an action in a given state. Reinforcement learning will be the primary focus for the remainder of this report.

2.3 Reinforcement Learning

When a child is learning to ride a bike, they have a number of actions at their disposal. They can 'turn handlebars left', 'turn handlebars right', 'push down on right pedal', 'push down on left pedal', 'lean right', 'lean left' etc. At first, they have no knowledge of the environment (the bike), so they're likely to choose an ill-suited combination of actions that might move them a few feet but is ultimately unsatisfactory. But the child makes further attempts, sampling the action space through trial and error, receiving a greater sense of accomplishment the further they travel. Over time, the actions that received greater positive feedback were neurologically enforced and the child developed an understanding of how to traverse the action space to travel the furthest.

The computational reinforcement learning that will be covered over the next few pages models the above example. "Reinforcement learning: An Introduction" [41] by Sutton and Barto, stated:

"Reinforcement learning problems involve learning what to do - how to map situations to actions - so as to maximize a numerical reward signal" [41]

In the case of reinforcement learning, the learner is not told which actions to take, but must discover which actions yield the most reward through trial and error. Reinforcement learning differs from unsupervised machine learning in

that reinforcement learning tries to maximize a cumulative reward whereas unsupervised learning tries to find hidden structure in what appears to be an unstructured dataset.

The trade off between exploration and exploitation is one of the ways reinforcement learning differs from other types of machine learning. To receive the most reward, an agent must execute actions that have been shown to return significant rewards in the past. But if the agent limits it's actions to the ones it knows, it will never be able to discover new and possibly more favourable actions. Neither exploration or exploitation can be used exclusively to perform optimal learning. There must be a balance. The agent must try a variety of actions and progressively favour the better ones as the task progresses.

But in order to fully understand exploration vs exploitation, and the other important concepts that encompass reinforcement learning, the terminology must first be understood.

2.3.1 Terminology

- **Agent:** The learner and decision maker is called the agent. An agent is trained to perform a certain task in an environment. For example, when DeepMind released their paper 'Human level control through deep reinforcement learning' [29], the agent was the system receiving observations (game frames) and associated rewards. The agent also deciphered optimal actions based on the observations it received.

- **Environment:** The environment is the agents world, in which it lives and interacts. The agent can interact with the environment by performing some action but cannot influence the rules or dynamics of the environment by those actions. Using DeepMinds 2015 Nature article [29] as an example again, the environment was the game the system was applied to e.g. Atari Breakout, Asteroids, Pong etc. The environment changes as the agent interacts with it but the rules of the game remain the same.

- **Policy:** A policy is a mapping of environment states to actions. A policy can be regarded as the agents strategy. A policy may be a simple

function or lookup table, or it may involve extensive computation such as a search process. In the case of DeepMinds 2015 Nature article [29], the policy was defined by a deep neural network. This neural network mapped states (stacked frames) to an optimal action i.e. move left or right. A reinforcement learning algorithm is said to be 'on-policy' or 'off-policy'. On-policy learning involves training an agent using the same policy that it uses to make decisions in the environment. Off-policy learning involves training an agent using a different policy than the one it uses to make decisions in the environment.

- **Reward:** The reward signal allows the agent to deduce what the good and bad decisions are. Each time step, the environment sends a single number (reward) to the agent. The agents sole objective is to maximize the total cumulative reward it receives. The reward signal is the primary basis for altering the policy. If an action is followed by a low reward, then the policy may be changed to select some other action from that state in the future. In the case of DeepMinds 2015 Nature article [29], a reward of 1 is received every time the score is incremented. A reward function R(s, a) maps the combination of state and action to a numerical reward.

- **Value function:** A value function specifies what states are good in the long run, while rewards focus on immediate feedback. For example, a state might have a high value despite having a low reward if it is regularly followed by other states that yield high rewards. Values are a prediction of future rewards. Values are estimated and re-estimated based on environment observations. In DeepMinds 2015 Nature article [29], a type of value function known as a Q-function is employed. The Q-function is in the form of a deep neural network. It returns the value of the available actions for a given state. V(s) is used to represent a value function. The value function is not to be confused with the Q-function, Q(s, a). While the value function V(s) estimates how good a given state is, the Q-function Q(s,a) is used to estimate how good an action is in a particular state. Using the example of Atari Breakout, V(s) states how valuable the paddle position, ball position, and tile configuration are with respect to each other. Q(s,a) states how valuable a certain action is (right or left), given the current paddle position, ball position, and tile configuration.

- **Environment model:** A model of the environment is not required by all reinforcement learning algorithms. Some are referred to as 'model free' as opposed to 'model based'. Model free learners learn by trial and error. Model based learners rely on a representation that mimics the behaviour of the environment, facilitating planning. Models allow for inferences to be made about how the environment will behave. Consider an N by N grid world. In this N by N grid world, the agents goal is to navigate from a "start" tile to a "goal" tile, located on the far side of the grid world. The agent must avoid "hole" tiles, "ice" tiles and "fire" tiles along the way. The agent can move "up", "down", "left", or "right". This environment can be modelled by representing each tile as a state, denoted by a coordinate (x,y). The four actions can be specified by integers ranging from 0 to 3. On the ice tiles, the agent may slip. On these tiles the agent moves in the specified direction 80% of the time, but moves incorrectly 20% of the time. The transition probabilities can be represented as a function $P(s'|s,a)$. If the agent wished to move right, the transition probability breakdown would look like:

  $P(right\_of\_current\_state|(current\_state), move\_right) = 0.8$
  $P(above\_current\_state|(current\_state), move\_right) = 0.0667$
  $P(left\_of\_current\_state|current\_state), move\_right) = 0.0667$
  $P(below\_current\_state|(current\_state), move\_right) = 0.0667$

  A reward function that returns -1 for each standard state, -100 for each "fire" state, -100 for each "hole" state, and +100 for the goal state could be used to encourage the agent to find the optimal policy.

  The above terms are commonly used in the field of reinforcement learning. Now that we understand them, we can move on to some of reinforcement learnings more important concepts.

2.3.2 Markov Decision Processes

"Lecture 2: Markov Decision Processes" [38] by David Silver outlines how:

"Markov decision processes formally describe an environment for reinforcement learning where the environment is fully observable" [38]

A Markov Decision Process is a model for predicting outcomes i.e. how actions will affect future rewards. It attempts to predict an outcome given only the current state in an environment. Markov Decision Processes incorporate the characteristics of actions and motivations. At any given step, an agent may choose to take an action, resulting in the model moving to the next step, and the agent receiving an associated reward. Figure 2 outlines an example Markov Decision Process. The goal of the agent is to traverse the Markov Decision Process so that it maximises the total reward it receives. The black percentage values denote the probability of transitioning to the accompanying state.

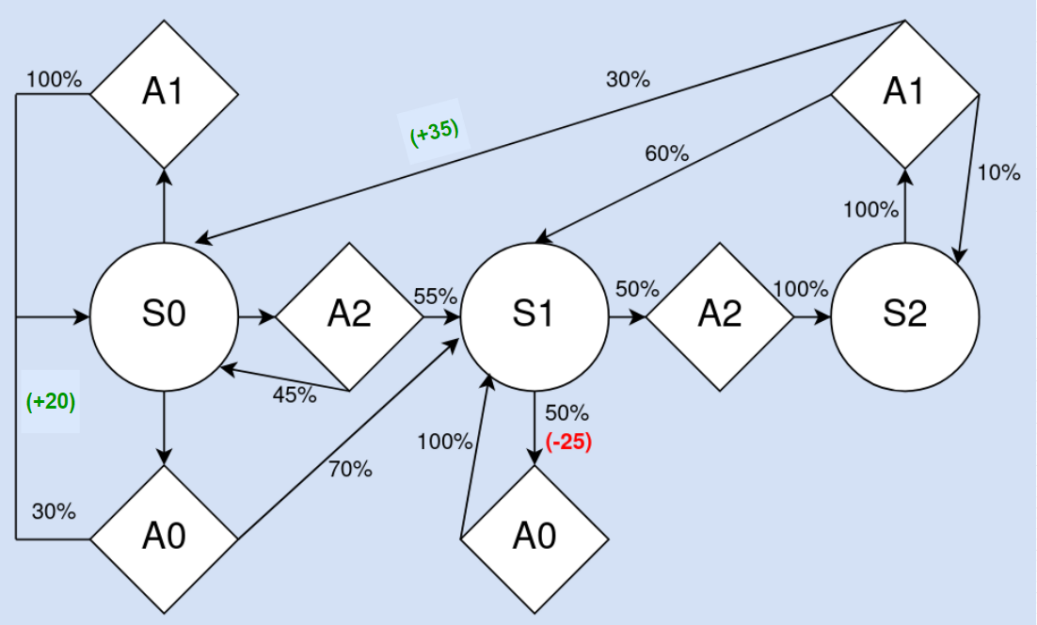


Figure 2: Markov Decision Process Example, adapted from "Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow" by Aurélien Géron [9]

A state S_t is Markov if and only if:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ..., S_t] \tag{1}$$

Equation 1 highlights a key characteristic of Markov Decision Processes, which is that the future state depends solely on the current state and not on the prior states. For example, consider a weather model that states:

$$\mathbb{P}[Cloudy|Sunny] = 0.2 \tag{2}$$

The above transition probability can be interpreted as, the probability of tomorrow being cloudy if today is sunny is 20% i.e. tomorrows state is solely dependent on todays state. A Markov Decision process is a sequence of Markov states, which means any state prior to the current state is redundant. The current state contains all the information the agent requires.

Markov Decision Processes were described by Richard Bellman in the 1950s. They are very similar to what are known as Markov chains but at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. The state transitions return some reward, and the agents goal is to find the policy that will maximize the reward over time.

2.3.3 Exploration vs Exploitation

In any reinforcement learning problem, you may notice situations where the agent chooses an action randomly, despite being aware of a better suited action. For example, an AI learning to play chess may choose an action randomly despite knowing that move x has preceded a win in the past. An analogy from "Hands on Machine Learning with Skikit-Learn, keras & TensorFlow" [9] by Aurélien Géron explains the reasoning behind such a decision.

"Suppose you go to a restaurant for the first time, and all the dishes look equally appealing, so you randomly pick one. If it turns out to be good, you can increase the probability that you'll order it next time, but you shouldn't

increase that probability up to 100%, or else you will never try out the other dishes, some of which may be even better than the one you tried." [9]

Additionally, a 2004 paper by Melanie Coggan titled "Exploration and exploitation in reinforcement learning" [7] stated that:

"Balancing exploration and exploitation is particularly important. The agent may have found a good goal on one path, but there may be an even better one on another path. Without exploration, the agent will always return to first goal, and the better goal will not be found." [7]

Essentially, the agent should not limit itself to particular actions in case there is a more suitable action that can be taken from that state. In reinforcement learning problems with static state and action spaces it may make sense to reduce the amount of exploration done over time. This results in the agent prioritizing optimal actions when it builds up sufficient knowledge of the environment and the other available actions. Epsilon-Greedy is a method employed to balance exploration and exploitation randomly. Epsilon denotes the proportion of the time that actions are chosen randomly. 1-Epsilon is the proportion of the time the optimal action is chosen. Every time the agent is to take an action, a random number between 0 and 1 is generated. If the number is greater than epsilon, the optimal action is chosen, otherwise a random action is chosen (to facilitate exploration). As mentioned prior, the core of reinforcement learning is about maximizing a cumulative reward. The only way an agent can do this is by building up a strong knowledge of the state and action space, thus the balance between exploitation and exploration and the degree to which it changes over time is crucial to the successful training of a reinforcement learning agent. An agent that interacts with a dynamic environment may maintain a higher epsilon-greedy value as it must adapt to the changing environment. Figure 3 depicts an image, adapted from Sutton and Barto [41], showing how performance can vary with differing values for epsilon.

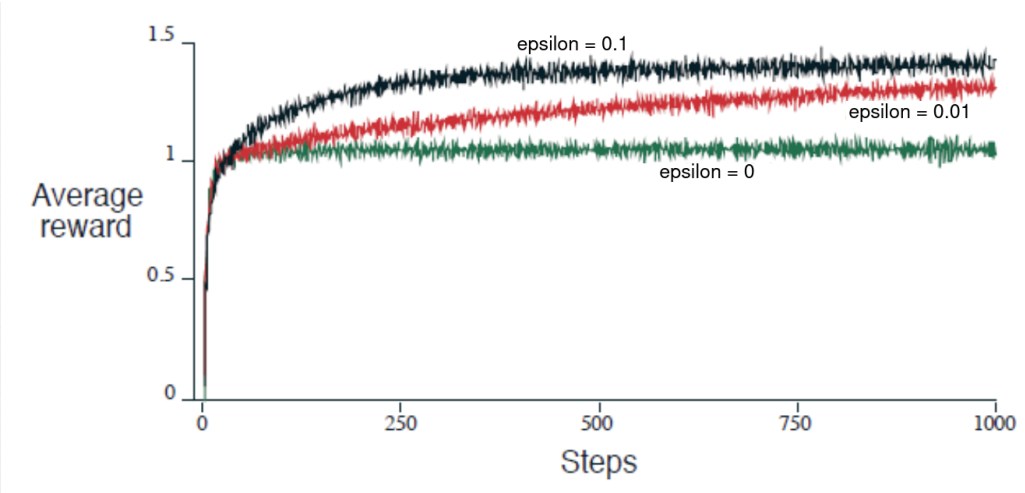


Figure 3: Performance with Varying Values for Epsilon, adapted from "Reinforcement Learning: An Introduction" [41] by Sutton and Barto.

In this figure, three agents are trained for 1000 steps each. The first agent is denoted by the black line on the graph. This agent has a value of 0.1 for epsilon, meaning it explores 10% of the time, and chooses the optimal action 90% of the time. The second agent is denoted by the red line on the graph. It has a value of 0.01 for epsilon, meaning it explores 1% of the time and exploits 99% of the time. The third and final agent, denoted by the green graph line explores 0% of the time and exploits 100% of the time.

The agent that only exploits (green) does not show improvement after the first few steps, as it is stuck in a specific path with no possibility of discovering new paths. The black agent reaches an average reward of 1.0 slower than the green and red agents but continues to show improvement and eventually convergence to the optimal path. The red agent explores at a slower rate than the black agent, but nonetheless will eventually converge to the same optimal path.

2.3.4 The Credit Assignment Problem

In supervised learning, we have a list of the outputs we want for the given inputs. These outputs are known as labels. We can reference these labels to infer beneficial policy adjustments. We don't have such an option with

reinforcement learning. If we did, we could reduce the cross entropy between the estimated probability distribution and the target probability distribution, ultimately converging to a beneficial policy. In reinforcement learning, the agent learns based on the rewards it receives. The problem with this is that the rewards are often sparse and delayed. How does the agent know which action(s) in a series of actions contributed most to a beneficial or detrimental outcome. This is known as the credit assignment problem. What action(s) should the agent give credit to. The way this problem is dealt with is by assigning the sum of all rewards following an action to that action. This essentially denotes how valuable said action was. Additionally, a discount factor is often applied at each step. As we go into the future, that action carries less and less weight so it's effect is reduced with a discount value. A high discount value, close to 1, places more emphasis on future rewards, making the agent future oriented. A discount factor close to 0 places more emphasis on immediate rewards. The agent will perform differently with different discount values so choosing a suitable value is crucial. The sum of discounted actions is called the actions return. Generally, discount factor values range from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future contribute approximately half the amount of the immediate reward. With a discount factor of 0.99, rewards 69 steps into the future contribute approximately half the amount of the immediate reward. Figure 4 is extracted from "Hands-On Machine Learning with Scikit-learn, Keras & Tensorflow" [9]. It depicts how one would compute an actions value. i.e. the sum of discounted future rewards.

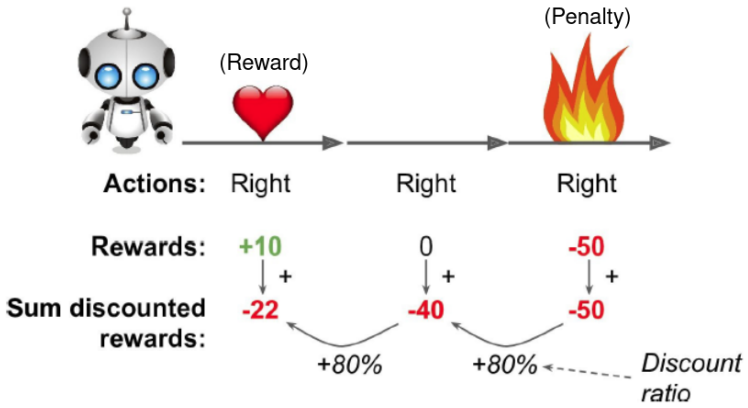


Figure 4: Credit Assignment Example, adapted from "Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow" [9] by Aurélien Géron.

Figure 4 shows three consecutive actions in three consecutive states. In the first state, the agent receives a positive reward. In the second state, it receives a reward of 0. In the third state, it receives a negative reward. To determine the value of the initial move, we calculate the:

$$reward\ +\ (estimated\ future\ value\ *\ discount factor) \tag{3}$$

Using a discount factor of 0.8 and working back from the terminal state, the calculation can be broken down as follows:

$$\begin{aligned} &-50 = Terminal\ state \\ &-40\ (Terminal\ state\ -\ 1) = 0 + (-50 * 0.8) \\ &-22\ (Terminal\ state\ -\ 2) = 10 + (-40 * 0.8) \end{aligned} \tag{4}$$

2.3.5 The Bellman Equation

Richard Bellman defined a way to accurately estimate the value of a given state, assigning to it the sum of all discounted future rewards the agent can expect, assuming it acts optimally. It is known as the Bellman Optimality Equation and is the result of pioneering work in Dynamic programming by Bellman in the 1950's [4]. The Bellman Optimality Equation states that if an agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible future states. The equation is defined as follows:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma.V^*(s')] \tag{5}$$

Where,

- T(s, a, s') is the transition probability from the state s to s', given that the agent chose action a.
- R(s, a, s') is the reward the agent receives when it moves from state s to s', given that the agent chose action a.

- γ is the discount factor.

From the Bellman Optimality Equation, we can derive the Value Iteration Algorithm, shown below:

$$V_{k+1}(s) = \max_{a} \sum_{s'} T(s, a, s')R(s, a, s') + \gamma.V_k(s')] \tag{6}$$

In this equation, $V_k(s)$ is the estimated value of the state s at the kth iteration. If all state value estimates are initialized to zero and iteratively updated using the above algorithm (eq. 6), the estimates converge to the optimal state values, corresponding to the optimal policy. To get a better understanding of the Value Iteration Algorithm and the Bellman Equation, consider the following grid world example:

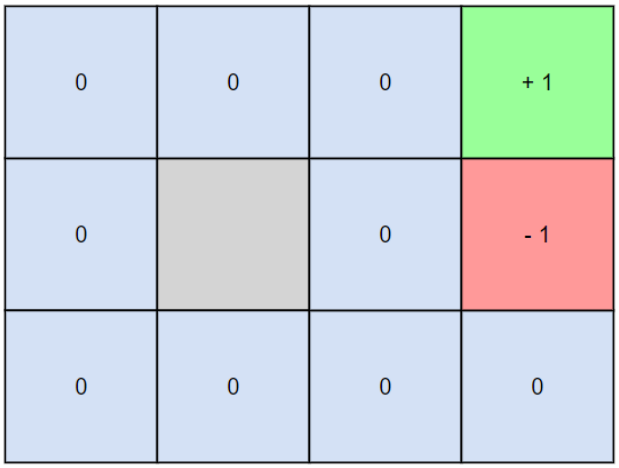


Figure 5: 4 x 3 Grid World

An agent lives in the above grid world. He currently resides in the bottom left corner but wants to travel to the top right corner. A grey wall stands between him and the top right corner. In addition, the agents actions don't always go to plan. 20% of the time the action to move in a direction results in the agent moving left or right of that move. A move left or right of that move is equally probable when it does occur.

To find the optimal path from the start state to the goal state, all states are first initialized to 0, excluding the terminal states. The agent then explores the environment, performing a value update on each state until convergence. The update is done using the value iteration algorithm outlined above (eq. 6). An example update using this value iteration algorithm is shown below (eq. 7). The reward for moving to a state is 0, excluding the terminal states which have a reward of +1 and -1. The discount value is 0.9.

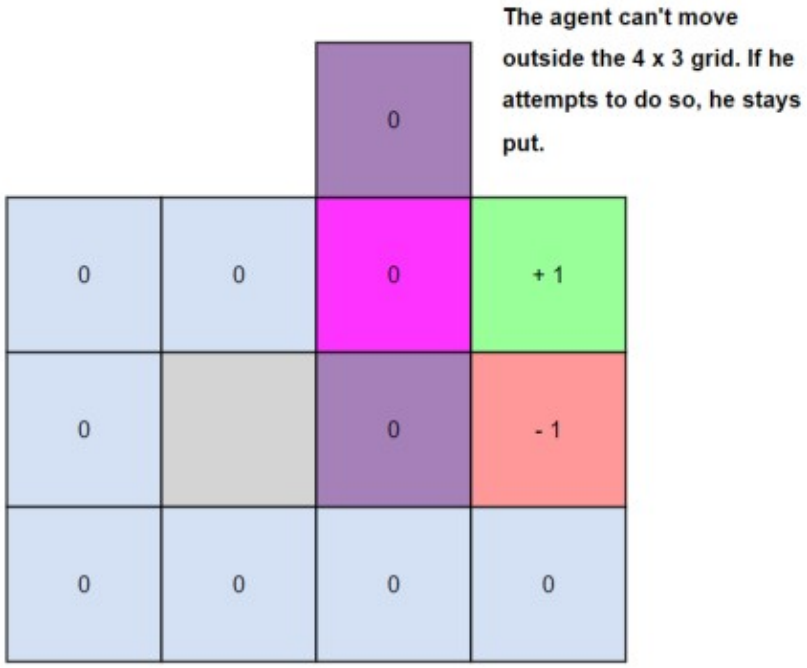


Figure 6: 4 x 3 Grid world Border Image

The below update (eq. 7) is the update for tile (3, 3) when the agent moves to the goal state.

$$V_2(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + y.V_1(s')]$$

$$V(3, 3) = \max_a \sum_{s'} T((3, 3), right, s')[R((3, 3), right, s') + (0.9).V_1(s')]$$

$$V(3,3) = 0.8[0 + 0.9(1))] + 0.1[0 + 0.9(0)] + 0.1[0 + 0.9(0)]$$
$$V(3,3) = 0.72 \tag{7}$$

Above (eq. 7) is the update for a single state, but the value iteration algorithm updates for all states, as shown in the below pseudocode:

1. Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

2. Repeat
3. $\quad \Delta \leftarrow 0$
4. $\quad$ For each $s \in \mathcal{S}$:
5. $\quad\quad v \leftarrow V(s)$
6. $\quad\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
7. $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
8. until $\Delta < \theta$ (a small positive number)

9. Output a deterministic policy, π, such that
10. $\quad \pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 7: Value Iteration Pseudocode, adapted from "Reinforcement Learning: An introduction" [41] by Sutton and Barto.

2.3.6 Dynamic Programming

According to Sutton and Barto [41]:

"The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP)." [41]

Dynamic Programming is about solving a big problem recursively. The way

this is done is by dividing the problem into smaller sub-problems and then storing the results of these sub-problems so they don't need to be recomputed later.

The generalized policy iteration algorithm is classed as a dynamic programming technique. It iteratively approximates the optimal value function, getting closer and closer on each iteration. This is a lot more tractable than the value iteration method outlined in the prior section, especially when dealing with large state spaces. The way the generalized policy iteration algorithm works is as follows:

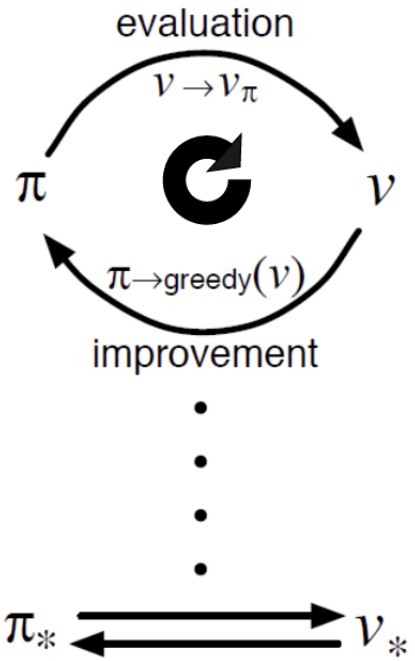


Figure 8: General Policy Iteration Diagram, adapted from "Reinforcement Learning: An Introduction" [41] by Sutton and Barto.

1. Randomly initialize a value function and start with a random policy.
2. Evaluate the values of each state with respect to this policy.
3. Update the policy, making greedy action choices with respect to the optimal value function.

4. Loop until it converges to the optimal value function and optimal policy.

Every time the policy is evaluated and the value function updated, the policy loses its greedy nature. Every time the policy is made greedy with respect to the value function, the value function estimate becomes incorrect.

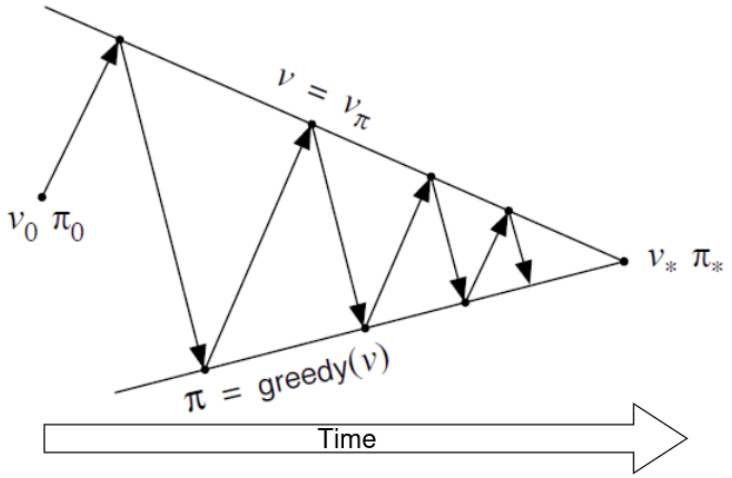


Figure 9: Generalized Policy Iteration Convergence, adapted from "Reinforcement Learning: An Introduction" [41] by Sutton and Barto.

Figure 10, on the next page, outlines pseudocode for the Generalized Policy Iteration algorithm.

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
      If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
4. If *policy-stable*, then stop and return V and π; else go to 2

Figure 10: Generalized Policy Iteration Pseudocode, adapted from "Reinforcement Learning: An Introduction" [41] by Sutton and Barto.

Figure 11 depicts the application of the Generalized Policy Iteration algorithm to a 4 by 4 gridworld with two goal states. We can see from Figure 11 that the Generalized Policy Iteration algorithm involves repeatedly jumping between updating the value function and the policy.

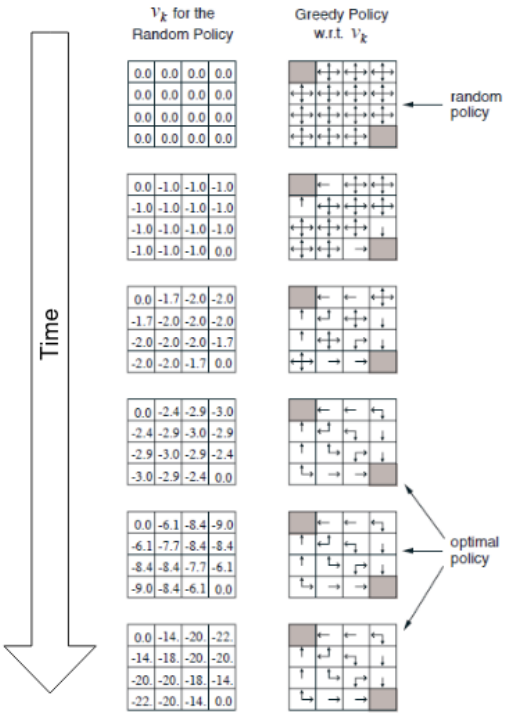


Figure 11: Policy Iteration Visualized, adapted from "Reinforcement Learning: An Introduction" [41] by Sutton and Barto.

To better understand the Generalized Policy Iteration algorithm, it was applied to a one dimensional grid world consisting of seven tiles. In this one dimensional grid world an agent can move left, denoted by '0' or right, denoted by '1'. The goal state is the center of the grid world. Both the policy and value function are initialized with zeros, as shown in Figure 12:

```
Policy: [0, 0, 0, 0, 0, 0, 0], Value Function: [0, 0, 0, 0, 0, 0, 0]
```

Figure 12: GPI Initialized Example

The GPI algorithm is then applied until convergence. The final two lines of Figure 13 outline the optimal policy and value function. The optimal action for states left of the center state is '1' (move right). The optimal action for states right of the center state is '0' (move left).

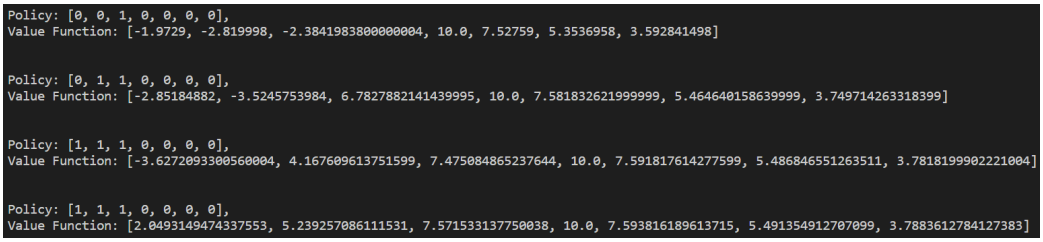


Figure 13: GPI Convergence

The following section will cover the Monte Carlo algorithm. One key difference between GPI and Monte Carlo methods is that GPI performs value iteration over the entire state space while Monte Carlo uses samples of the state space that are encountered during interactions with the environment. This means that GPI can converge to the optimal solution quicker but also requires more computation and may be less sample efficient. MC methods are more sample efficient but may require more interactions with the environment to converge to a good solution.

2.3.7 Monte Carlo Algorithm

This section transitions from model based learning to model free learning. The Monte Carlo algorithm only updates for visited states, giving it a speed advantage over the prior methods. Also, in the case of the Monte Carlo algorithm, the states don't need to be known ahead of time, they're simply discovered during interaction with the environment. Experiences are collected while interacting with the environment, and the Bellman equation is then used to estimate the returns for each state, given the current policy.

In comparison to the prior sections, the Monte Carlo algorithm relies on estimates rather than accurate values. The Monte Carlo policy is a simple lookup table mapping states to action. The algorithm starts off with a random policy but as it interacts with the environment and gains experience, the value estimates improve, ultimately improving the policy. The algorithm works as follows:

1. Initialize G (Return) to 0.
2. Initialize a record of states and returns.
3. Play a game, adding to the states and rewards record.
4. Loop backward through the list of states and rewards (s, r) performing the following:
   (a) append (s, G) to the states and returns record.
   (b) G = r + gamma * G.
5. Reverse the states and returns record to the original order.

The pseudocode for the Monte Carlo algorithm is outlined in Figure 14:

1. Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
   $Q(s,a) \leftarrow$ arbitrary
   $Returns(s,a) \leftarrow$ empty list
   $\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

2. Repeat forever:
   (a) Generate an episode using π
   (b) For each pair s, a appearing in the episode:
      $G \leftarrow$ return following the first occurrence of s, a
      Append G to $Returns(s,a)$
      $Q(s,a) \leftarrow$ average$(Returns(s,a))$
   (c) For each s in the episode:
      $a^* \leftarrow \arg\max_a Q(s,a)$
      For all $a \in \mathcal{A}(s)$:
      $$\pi(a|s) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

Figure 14: Monte Carlo Algorithm Pseudocode, adapted from "Reinforcement Learning: An Introduction" [41] by Sutton and Barto

One downside of MC methods when compared to Temporal Difference methods is that they may require more interactions with the environment in order to converge to a good solution. This is because MC methods rely on sample averages to estimate values which can be noisy and require a larger number of samples to be accurate. In contrast, Temporal Difference methods can update value estimates quicker and with fewer samples, making them more sample efficient. Temporal Difference will be covered in the next section.

2.3.8 Temporal Difference

In the Monte Carlo algorithm, the algorithm waits until the return is known before updating the value function. It uses this return as the target value in the update rule:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(s_t)] \tag{8}$$

where, α denotes the step size.

Although Monte Carlo methods and Temporal difference methods contain similarities, there is one significant difference. Monte Carlo methods must wait until the end of the episode to determine the update to $V(S_t)$. Temporal difference methods only need to wait until the next time step to update $V(S_t)$. At timestep t+1, TD methods use $R_{t+1} + \gamma V(S_{t+1})$ as the target. We call the Temporal Difference method a bootstrapping method as it bases its update on an existing estimate. The advantage of this over the Monte Carlo method is that it's implemented in an online, fully incremental fashion. This is beneficial if dealing with very long episodes or indefinitely running tasks that have no episodes at all. Additionally, as some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, Temporal Difference methods can prove more efficient. In general, Temporal Difference methods are seen to learn faster than Monte Carlo methods. The Temporal difference learning algorithm is as follows:

$$V_{k+1}(S) \leftarrow (1 - \alpha)V_k(S) + \alpha(r + \gamma.V_k(S))$$

or equivalently,

$$V_{k+1}(S) \leftarrow V_k(S) + \delta_k(s, r, s')$$

where,

$$\delta_k(s, r, s') = r + \gamma.V(s') \tag{9}$$

In the case of the above equations, it should be noted that:

α denotes the learning rate.
γ denotes the discount factor.
r denotes the reward.
s denotes the state.
v denotes the value.

Q-learning is an example of Temporal Difference Learning. In Q-learning, the agent learns to predict the expected future reward for taking a particular action in a given state based on the difference between the predicted value of the current state and the observed value of the next state. The details of Q-learning will be outlined in the next section.

2.3.9 Q-learning

Q-learning is a Temporal Difference learning algorithm. More specifically, it is a model free, off-policy, and value based reinforcement learning algorithm that learns the value of taking an action in a particular state. As mentioned in the prior sections 'model free' means it doesn't rely on a complete or partial model of the environment. Off-policy in the context of Q-learning means the updates to the value function are based on experiences that are generated by a policy that differs from the policy being learned. Off-policy algorithms are deemed particularly interesting as they facilitate the learning of an optimal policy by using a suboptimal policy. Aurélien Géron said "imagine learning to play golf when your teacher is a drunken monkey" [9] when describing them. The Q-learning algorithm is also value based, meaning it updates its value function based on an equation. In the case of Q-learning, the equation is a variant of the Bellman equation. On the other hand, a policy based

reinforcement learning algorithm estimates the value function with a greedy policy, obtained from the last policy improvement.

The Q-learning algorithm finds an optimal policy in the sense of maximizing the expected value of the total reward over all successive steps. It uses a table known as a Q-table to store state-action values or Q-values. The Q-table can be regarded as the algorithms value function.

Q-learning involves watching an agent interact with its environment and improving it's Q-value estimates to prioritize more beneficial actions (i.e. actions that yield more reward) over time. Once it has accurate or close to accurate Q-Value estimates, the optimal policy is defined by choosing the action that has the highest Q-Value of all legal actions. The Q-learning algorithm is as follows:

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma . \max_{\alpha'} Q(s',a') \tag{10}$$

The use of the max operator is a key aspect of the Q-learning algorithm as it allows the agent to learn the optimal action selection policy for a given environment. By selecting the action with the highest expected reward, the agent is able to maximize its long-term reward and learn the optimal policy for interacting with the environment.

This algorithm keeps a running average of the reward an agent gets on leaving state s, plus the sum of discounted future rewards it expects to get. The algorithm will converge to the optimal Q-values after many iterations of training. But some degree of hyper-parameter tuning is often necessary.

A completely random policy will visit every state and every transition but it will take a very long time for the algorithm to reach convergence. A beneficial alternative is to use an epsilon-greedy policy that acts randomly with a probability, epsilon. Over time, the algorithm will spend more and more time investigating the parts of the Markov decision process that are proving to be most beneficial (i.e. yield the most reward). Below is a practical implementation of the Q-Learning algorithm (eq. 11):

Consider the 3 by 2 grid world depicted in Figure 15:

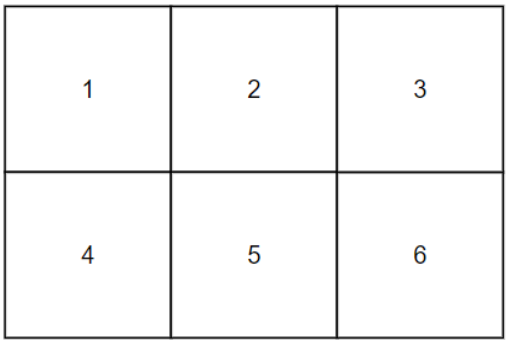


Figure 15: 3 x 2 Grid World

In this 3 by 2 grid world, the player moves from tile 2 to tile 6. The player receives 10 points for reaching tile 6 and -1 for all other tiles. The player can choose to move up, down, left or right but he/she can't move outside the given tiles. The initial state of the Q-table is outlined in Figure 16:

Actions	States				
	1	**2**	**3**	**4**	**5**
UP	0	0	0	8	5
DOWN	4	3	7	0	0
LEFT	0	2	5	0	5
RIGHT	6	4	0	4	6

Figure 16: Initial Q-table

If we assume a learning rate of 0.25 and a discount rate of 0.75, and the player moves down (Tile 5) and then right (Tile 6), we update the Q-table as follows:

$$
\begin{aligned}
Q(s,a) &= Q(s,a) + \alpha[R(s,a) + \gamma(\max Q'(s',a') - Q(s,a))] \\
Q(2,down) &= 3 + 0.25[-1 + 0.75(8-3)] = 3.6875 \\
Q(5,right) &= 6 + 0.25[10 + 0.75(0-6)] = 7.5625
\end{aligned}
\tag{11}
$$

The resultant Q-table is:

Actions	States 1	2	3	4	5
UP	0	0	0	8	5
DOWN	4	3.6875	7	0	0
LEFT	0	2	5	0	5
RIGHT	6	4	0	4	7.5625

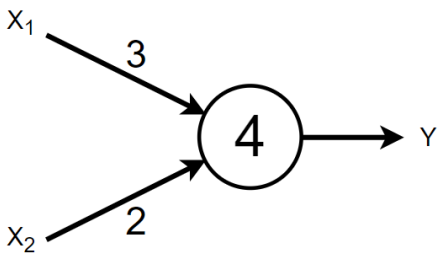
Figure 17: Updated Q-table

2.4 Deep Learning

Deep learning is a rapidly growing field of machine learning. It leverages artificial neural networks to solve a wide range of complex problems, such as computer vision, natural language processing, and speech recognition.

2.4.1 Threshold Logic Unit

The artificial neuron is the core component of deep learning. The first model of the artificial neuron was introduced by neurophysiologist Warren McCulloch and mathematician Walter Pitts in their paper "A Logical Calculus of Ideas Immanenet in Nervous Activity" [23] in 1943. Their model of the neuron was called the Threshold Logic Unit, and it was capable of performing a range of logical propositions. The Threshold Logic Unit takes a set of binary inputs. The inputs are then multiplied by assigned weights, and summed. If the summation is greater than a threshold value, the output is set to '1', otherwise it remains set to '0'. Figure 18 displays a Threshold Logic Unit capable of performing the boolean 'AND' operation.



X_1	X_2	$3X_1 + 2X_2$	Y
0	0	0	0
0	1	2	0
1	0	3	0
1	1	5	1

Figure 18: Threshold Logic Unit 'AND' Operation

2.4.2 The Perceptron and Multilayer Perceptron

Frank Rosenblatt presented an alternative artificial neuron model in his 1958 paper 'The perceptron: a probabilistic model for information storage and organization in the brain' [33]. This artificial neuron model is very similar to the Threshold Logic Unit, except that instead of applying a threshold activation function, which is better suited to boolean logic, a step activation

function is applied. Commonly used step activation functions are the heaviside and signum/sign functions. The heaviside step function returns 0 for negative values and 1 for positive values. The signum/sign function returns -1 for negative values, 0 if the value is 0, and 1 for positive values. Additionally, the perceptron generally has a bias neuron to shift the activation function along the input axis, which can be useful when separating linearly inseparable data. However, in a 1969 paper 'Perceptrons' [20] by Marvin Minsky and Seymour Papert the fact that Perceptrons were incapable of solving the Exclusive Or classification problem was raised. But this problem could be solved by stacking perceptrons on top of one another. This neural network architecture became known as the multilayer perceptron. Figure 19 outlines an example of a Multilayer Perceptron.

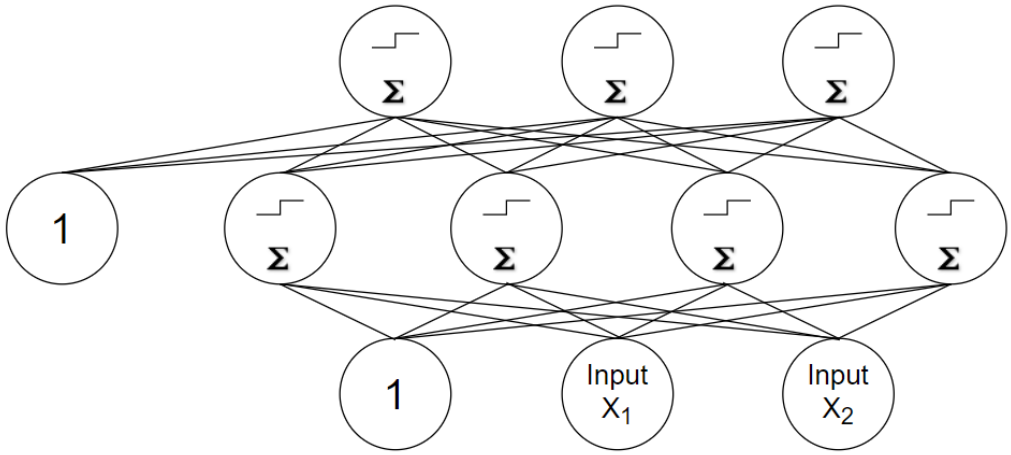


Figure 19: MultiLayer Perceptron

This Multilayer Perceptron consists of 1 input layer, 1 hidden layer, and 1 output layer. There are two input neurons in the input layer. A bias neuron is included in every layer except the output layer. Deep neural networks are networks that consist of a large number of hidden layers.

2.4.3 Backpropagation

While the first deep learning MLP was published in 1965 [12], it wasn't until 1986 that David Rumelhart, Geoffrey Hinton, and Ronald Williams published a paper titled 'Learning representations by back-propagating errors' [34] that introduced a suitable method for training these deep learning MLPs. The backpropagation algorithm can be broken down into 4 steps.

1. Pass the network inputs into the input layer. The output from each neuron is computed and passed to the next network layer. The neuron outputs are preserved as they're needed for backpropagation. This is known as the forward pass.

2. The output error is calculated i.e. target value - predicted value.

3. How much each connection contributed to the error is calculated by using the chain rule. This process moves backwards through the network, hence the name, backpropagation.

4. Gradient descent is then performed using the computed error gradients, adjusting the network weights to reduce the output error.

The step function is not an appropriate activation function for deep learning MLPs as it only contains flat segments and no gradient. To allow for effective learning, the step function can be swapped for the sigmoid function, $\sigma(z) = 1/(1 + \exp(-z)$.

2.4.4 Hyperparameters

A hyperparameter is a paremeter set prior to the training of a machine learning model, and influences the training process. Learning rate, batch size, activation function, and optimizer are some of the more important hyperparameters.

- Learning rate: The learning rate denotes the degree of adjustment applied to the weights with respect to the loss function during gradient descent. The optimal learning rate is roughly half the learning rate that causes divergence [9].

- Batch size: The batch size refers to the number of data points processed in a single training instance. Larger batch sizes allow GPUs to process the data points more efficiently, but they can cause instability, especially at the start of training.

- Activation function: An activation function is a mathematical function applied to a neurons output, to introduce nonlinearity into a neural network. By introducing nonlinearity, the network can learn to solve more complex problems. Examples of activation functions are the sigmoid function, ReLU function, and tanh function.

- Optimizer: The optimizer is the algorithm that updates the neural network weights during training. The optimizer computes the gradients of the loss function with respect to the model parameters, and updates the weights accordingly. Examples of optimizers include Stochastic Gradient Descent, Adam, and RMSprop.

2.4.5 Vanishing and Exploding Gradient Problem

As discussed in section 2.4.3, MLPs are trained by performing a process known as backpropagation. The degree that each connection contributes to an output error is calculated, and network weights are adjusted accordingly, to reduce this output error. However, in the early 2000s MLPs were losing their appeal due to unstable gradients. It was observed that the gradient becomes smaller and smaller as the algorithm works back through the network, resulting in the lower layers only receiving slight adjustment. This problem became known as the vanishing gradient problem. The exploding gradient problem refers to a similar phenomenon, where the gradient becomes larger and larger, causing algorithm divergence.

A 2010 paper 'Understanding the difficulty of training feedforward networks' [10] by Xavior Glorot and Yoshua Bengio outlined how the sigmoid activation function and certain weight initialization techniques were contributing to this problem. When employing the sigmoid activation function, very large or very small inputs saturate at 0 or 1, with only a very slight gradient. The ReLU activation function provided a suitable alternative. It was fast to compute and did not saturate for positive values. The downside of ReLU

activation was that it was prone to a problem known as 'dying ReLUs'. If the weighted sum of inputs produces a negative value, ReLU activation outputs 0. Activation functions such as leaky ReLU, ELU, and SELU can be used instead to solve this problem

Additionally, Glorot and Bengio proposed an alternate weight initialization technique, where the weights of each layer are initialized randomly, but the average number of input and output connections are constrained according to, $fan_{avg} = (fan_{in} + fan_{out})$. fan_{in} refers to the number of input connections to a layer. fan_{out} refers to the number of output connections from a layer. fan_{avg} is the average of fan_{in} and fan_{out}. This initialization technique became known as Xavier or Glorot Initialization.

2.4.6 Batch Normalization

While ReLU variants and certain initialization methods can prove effective at stabilizing gradients at the start of training, a technique known as Batch Normalization can further ensure stable gradients as training progresses. Batch Normalization was proposed by Sergey Ioffe and Christian Szegedy in a 2015 paper 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift' [11]. Batch Normalization zero-centers and normalizes each input, then scales and shifts the results, before and after each activation function in the hidden layers of the neural network. By performing Batch Normalization the model can learn the optimal scale and mean of each layers inputs and prevent a problem known as internal covariate shift. Additionally, Batch Normalization acts as a regularizer and can be employed to prevent overfitting.

2.4.7 Convolutional Neural Networks

A convolutional neural network (CNN) is a deep neural network, primarily used for image and video processing. But they can be used on alternative forms of data, such as audio. A convolutional neural network is composed of a number of convolutional layers which extract features from an image. The first convolutional layer typically extracts very low level features such as edges and corners. The second convolutional layer combines the features extracted from the first convolutional layer to form more complex features,

such as shapes. The remaining layers continue to extract features at increasing levels of abstraction. The output from the final convolutional layer is generally flattened and passed through a series of dense layers to perform classification.

Each convolutional layer is composed of a number of learnable filters or kernels. These filters/kernels are typically 2 dimensional or 3 dimensional matrices, used to produce a feature map by sliding over the input data. A feature map is typically a 2 dimensional array of features that denote the activation of a set of filters. Filter maps are passed into the next layer to extract higher level image features.

2.5 Deep Q-learning

A Q-table will not suffice for large scale problems. For example, the game of Go has 10^{172} states which would require a Q-table 10^{172} tiles wide. To put that into perspective, there is estimated to be between 10^{78} and 10^{82} atoms in the observable universe. However, DeepMinds AlphaGo did beat Lee Sedol in a five game Go match in Seoul, South Korea in March of 2016.

Instead of relying on a Q-table with 100% accurate Q-values, AlphaGo employed a deep neural network to approximate state Q-values. This form of Q-learning is called approximate Q-learning. Better known as Deep Q-learning. Deep Q-learning is said to use a Deep Q-network or DQN.

2.5.1 Vanilla Deep Q-network

Similarly to the example outlined in section 2.3.9, it's required that any given Q-value be as close as possible to the reward received after taking action a in state s plus the discounted value of playing optimally in the future.

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma . \max_{a'} Q(s',a') \tag{12}$$

The main difference is that instead of querying a Q-table to get the max Q-value of the next state, the next state is passed into a Deep Q-network

and a list of the Q-values for that state are returned. The highest Q-value is then picked and discounted to estimate the sum of discounted future rewards. The target Q-value is estimated according to the following equation:

$$Q_{target}(s, a) \leftarrow r + \gamma . \max_{a'} Q_{\theta}(s', a') \tag{13}$$

where,

s' denotes the next state and
a' denotes the next action.

In place of updating a Q-table, a gradient descent algorithm is applied to the Deep Q-network, to reduce a loss function between the estimated Q-value and the target Q-value. A commonly used loss function is mean squared error, but if one wishes to reduce the algorithms sensitivity to large outliers it may make more sense to use the Huber loss function.

The agent stores experiences (states, actions, rewards, next states) in what is known as a replay buffer. The replay buffer allows the agent to sample past experiences to train the network. Using a replay buffer can improve training stability by reducing the correlation between successive experiences, and avoiding catastrophic forgetting.

2.5.2 Deep Q-network with Fixed Q-targets

The original Deep Q-network was published by DeepMind in 2013 in the paper 'Playing Atari with Deep Reinfocement Learning' [28]. In this paper a noticeable inefficiency was resolved. If one relies on the same model to calculate the target Q-value, the function depends on itself. This is equivalent to playing hide and seek, except every time the seeker moves the hiders also move. This inefficiency gave rise to a number of Deep Q-network variants. The first being the Deep Q-network with fixed Q-targets.

The Deep Q-network with fixed Q-targets introduces an intermediary technique known as fixed Q-targets. With this technique, the feedback loop of the prior Deep Q-network is resolved by introducing another Deep Q-network.

One of the DQN's is regarded as the 'online' model, which learns at each step and decides what action to take for a given state. The other is known as the target model, and is only used to calculate target Q-values. The weights are copied from the online model to the target model at regular intervals. The target model predictions are used to calculate the target Q-values. This technique introduces greater stability as the targets stay stable for a duration of time.

2.5.3 Double Deep Q-network

While the Deep Q-network with Fixed Q-targets was a great improvement over the original DQN, it was prone to overestimation. This was largely due to the fact that if all actions from a given state are equally good, there will still remain slight variations in Q-value and the target model will still select the action with the largest value, despite it having a high probability of not being the best. In a 2015 paper 'Deep Reinforcement Learning with Double Q-learning' [46] DeepMind researchers made slight adjustments to further improve stability and prevent overestimation. Instead of passing the next state into the target network and choosing the largest Q-value (as in the Deep Q-network with Fixed Q-targets), the online model was used to select the best action for the next state and the target model was used to retrieve the Q-value of the best action.

The steps involved in Double Deep Q-learning are as follows:

1. Initialize the primary and target Networks, Q1 and Q2 with random values.
2. Observe the current state of the environment, s.
3. Use the primary Q-network, Q1, to select the action, a, that has the highest expected return.
4. Take the action and observe the resulting reward, r, and the next state, s'.
5. Use the target Q-network, Q2, to estimate the expected return for the next state, s'.

6. Calculate the loss between the target and estimated values. Perform gradient descent on the primary Q-network.

7. Every K steps, update the target Q-network, Q2, with the weights of the primary Q-network, Q1.

8. Repeat steps 2 through 7 until the agent has converged to the optimal policy.

Tabular Double Q-learning was first proposed in a 2010 paper "Double Q Learning" [45] to solve the problem of large Q-value overestimations. In vanilla Q-learning it is assumed that the best action is the one with the maximum estimated Q-value. But when the agent initially starts training it knows nothing about its environment and so the Q-values contain a lot of noise. Because of this noise it's difficult to know if the action with the largest Q-value is actually the best. In reality the best action is often the one with a smaller Q-value. The problem of overestimation refers to when the agent takes a non-optimal action because it has the largest Q-value. This creates a large positive bias in the updating procedure and increases the duration required for training. Double Q-learning addresses the overestimation problem by using two separate Q-networks to estimate the value of each action. Network 1 is used to select the action, and Network 2 is used to evaluate the value of the selected action. This helps to reduce overestimations because Network 1 and Network 2 can correct for each other's errors.

The pseudocode for the Double Q-learning algorithm is shown in Figure 20, on the next page.

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat $\rightleftarrows$
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Figure 20: Double Q-learning Pseudocode, adapted from "Double Q Learning" [45] by Hado Hasselt

2.5.4 Prioritized Experience Replay

The prior variants are said to sample experiences uniformly from the replay buffer. Each transition has equal probability of being picked. A commonly used variant of Deep Q-learning uses what's called importance sampling, facilitated by a prioritized experience replay buffer. When experiences are sampled from a prioritized experience replay buffer, the transitions that have a greater chance of contributing to faster learning are more likely to be selected. A transition with greater importance might be one that has resulted in the termination of the game, or one with a very rare state. A common approach to decipher importance mathematically is to measure the magnitude of the temporal difference error, and if sufficiently large, that transition can be regarded as surprising and more important. The temporal difference error is defined as:

$$\delta = r + \gamma.V(s') - V(s) \tag{14}$$

Initially, when a transition is recorded in the prioritized experience replay buffer its priority is set to a very large value so that it's sampled at least

once. But every other time it is sampled the priority is set anew. One important point is that since the samples will be biased towards the experiences with higher priority, the bias must be negated to a certain extent. Otherwise the model would overfit the experiences that have a higher priority. The idea of a prioritized experience replay buffer was first introduced by DeepMind in their paper 'Prioritized experience replay' [36].

The SumTree data structure is commonly used in prioritized experience replay buffer implementations. A SumTree is a binary tree data structure where each node is the sum of its child nodes. See Figure 21.

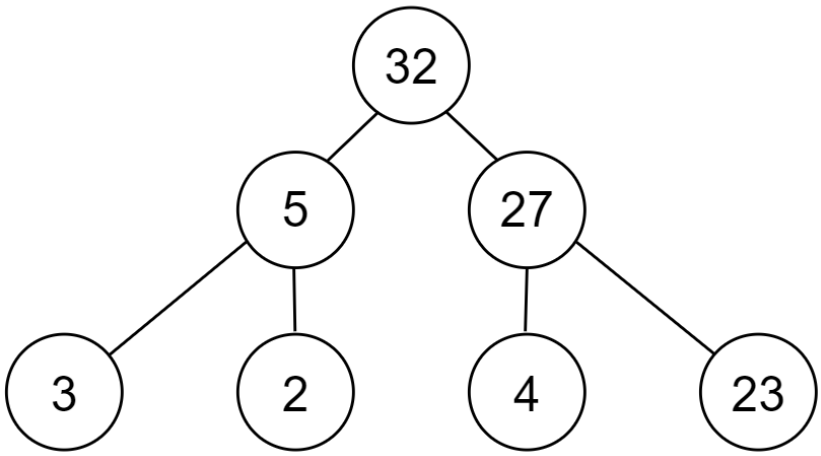


Figure 21: SumTree Example

The SumTree allows for rapid acquisition of the priorities summation by simply querying the first node in the tree. Additionally, by selecting a random value between 0 and the sum of priorities, and passing it through the tree, a buffer index, chosen according to the priorities can be retrieved. Now consider Figure 22, the SumTree alternative.

[(32, 3), (73, 2), (23, 4), (1, 23)]

Figure 22: Array PER Buffer

Figure 22 is an array based prioritized experience replay buffer. Each tuple holds the value to be sampled and the associated weight. To sample the array accordingly to the priorities, a random value between 0 and the summation of all weights is chosen. The array entry with the closest weight to this random value is then selected. However, as the priorities have to be added together every time the buffer is sampled, this method has a sampling time complexity of O(n). The SumTree implementation has a sampling time complexity of O(log n).

2.5.5 Dueling Q-network

The Dueling Deep Q-network was also introduced by DeepMind in a 2015 paper, titled 'Dueling network architectures for deep reinforcement learning' [47]. In the case of a dueling network architecture, the Q-value is expressed as a combination of the state value and the advantage of taking a particular action in that state i.e.

$$Q(s, a) = V(s) + A(s, a) \tag{15}$$

In this architecture the network estimates both the value of the state and the advantage of each possible action. The benefit of separating the network into two estimators is that the architecture can now learn which states are and which are not valuable, without having to learn the effect of each action for each state. In the Atari game, Enduro, if the car is a significant distance away from every obstacle, a move may not be necessary. This architecture is relevant as it'll only perform an action if it will affect the environment in a meaningful way.

2.5.6 Deep Q-network Instability

Deep Q-network approaches prior to DeepMinds 2015 Double Q-learning paper [46] are said to be inherently unstable. An unstable network changes its prediction dramatically when input data is modified slightly. Suppose a network is trained with 10,000 instances of training data and it performs well on our test set. If the network is a stable model, then it should still

perform relatively well on 9,000 training instances. The loss graph for training the network with 9,000 training instances should be very similar to when it's trained with 10,000 instances, minus the last 1,000 plotted graph points. But if the model is unstable, a considerable difference in the loss graphs will be noticable.

Stability can worsen due to a number of factors. If the observations/input data used to train the neural network are highly correlated then gradient descent will be performed on very similar states and the neural network will overfit them states and be insufficiently trained for alternative states. Deepminds 2013 paper [28] tackled this cause of instability by introducing a replay buffer. The observations used to train the network are stored in the replay buffer and then sampled randomly at a later time. Introducing random sampling is effective at reducing any correlation between states.

Another cause of instability is absence of a target network. The original Deep Q-network architecture relied on the same network to obtain both the target and estimated Q-values. But this has been proven to significantly impact the neural network update as whenever the model is updated for the current state it could impact the Q-values for the next state. As mentioned prior, a metaphor worth considering is a modified game of hide and seek where every time the seeker moves, the hider also moves. This is fundamentally the same problem faced when a target network is absent. The solution to this problem is to introduce a separate network, the target network. The target network is used to obtain the target Q-values, and is only updated every few episodes, maintaining fixed Q-values for the main network for some duration.

In 'An Introduction to Reinforcement Learning' [41], the concept of the Deadly Triad is introduced (Chapter 11). The Deadly triad refers to three reinforcement learning techniques that when combined significantly increase the danger of instability and divergence. The three being function approximation, bootstrapping and off policy training.

- Function approximation: Function approximation assumes that there is a function that can be defined to predict outputs/labels for a given input to a high level of accuracy. Function approximation methods are effective at generalizing a state space that is too large to be processed

or stored by a computer. In Deep Q-learning, function approximation is used in the form of a deep neural network.

- Bootstrapping: Bootstrapping involves estimating a certain value by making use of an estimate for a different value. In the case of Deep Q-learning or Q-learning in general, it is used when the estimated Q-value of the current state is calculated by relying on the estimated Q-value of the next state.

- Off-Policy training: In on-policy learning, the policy that is being improved is the same policy that is being used for action selection. In off-policy learning, the policy being improved differs from the policy that is being used for action selection i.e. the behaviour policy. SARSA is an example of an on-policy learner, whereas Q-learning is an example of an off-policy learner as it always chooses the max Q-value of the next state irrespective of the current policy.

If any two of the aforementioned techniques are present in a machine learning method then instability can be avoided. But if all three are present then it is very difficult to perform stable training. All of the techniques outlined above are present in Deep Q-learning. Deep Q-learning uses function approximation in the form of a deep neural network. It uses bootstrapping as it is a form of temporal difference learning and relies on the estimated Q-value of the next state to denote the estimated Q-value of the current state. It also uses off-policy training as its value update always uses the max Q-value of the next state irrespective of the current policy.

It proves quite difficult to successfully train a DQN agent given the default nature of Deep Q-networks, but it can be achieved if the network is trained for tens of millions of steps and a replay buffer and target network are employed. Some other techniques that can be used to improve the odds of achieving stability are gradient clipping, reward clipping and functional regularization.

- Gradient Clipping: Vanishing and exploding gradients are a well known problem during backpropagation. When weights become exceedingly

large or small, gradient descent can diverge. Clipping gradients at a certain level ensures gradient descent occurs in a more reasonable fashion even if the loss landscape of the model is more irregular.

- Reward Clipping: Reward clipping reduces the effect of extreme observations on the model update process. Reward clipping ensures the reward is limited to a specific range. The numpy sign function is employed in many Atari agents to force rewards to be in the range -1 to +1.

- Functional Regularization: While target networks have proven to be an instrumental development in the advancement of Deep Q-networks, they are an inflexible and slow method of ensuring greater stability. A 2021 paper titled 'Beyond Target Networks: Improving Deep Q-learning with Functional Regularization' [32] outlines an alternative method of ensuring greater stability via functional regularizers. In this paper it is demonstrated that the Bellman equation can be augmented with a functional regularizer to achieve both faster and more stable training.

2.5.7 DeepMellow Q-network

The target network has been fundamental to the advancement of the field of Deep Reinforcement Learning as it has improved training stability and ultimately lead to significant breakthroughs in a wide range of applications. However, a 2019 paper titled "DeepMellow: Removing the need for a target network in Deep Q-learning" [13] proposes an alternative method of performing Deep Q-learning that eliminates the need for a target network. This paper introduces a softmax operator known as MellowMax that when swapped with the DQN algorithms Max function has been shown to outperform the Deep Q-network with target network. The DeepMellow Q-network has been shown to facilitate faster and more stable training, as shown in Figure 23.

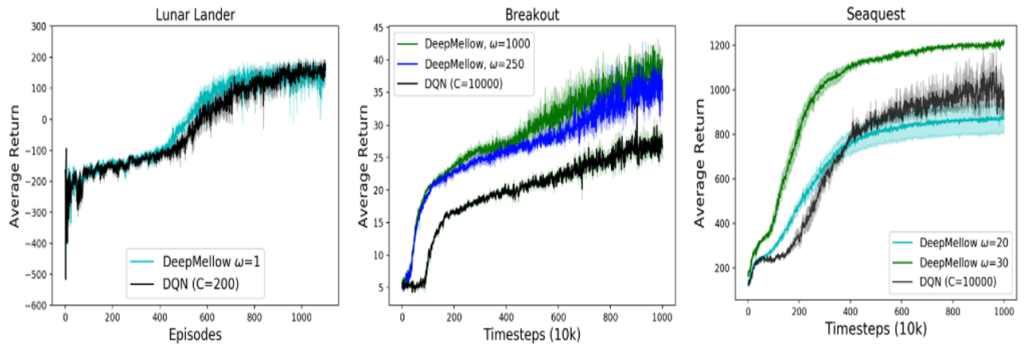


Figure 23: Deep Mellow Comparison Results, extracted from "DeepMellow: Removing the need for a target network in Deep Q-learning" [13] by Seungchan Kim et al.

The authors of "DeepMellow: Removing the need for a target network in Deep Q-learning" argue that online reinforcement learning is limited by the target network and in order to move towards online reinforcement learning target network and replay buffer alternatives must be investigated.

The MellowMax Softmax operator is defined as:

$$mm_w(x) = \frac{\log(\frac{1}{n}\sum_{i=1}^{n})\exp(wx_i)}{w} \tag{16}$$

In the context of Atari Breakout,

n is the number of actions,
x is the Q-values and,
w is a temperature value.

As the termperature value (w) goes to infinity MellowMax acts like a Max operator. When the temperature value (w) is zero, MellowMax acts as a mean average operator. If the temperature value sits somewhere between zero and infinity, it returns a soft interpolation between the mean and the max. Additionally, as the temperature value moves to negative infinity MellowMax acts like a Min operator. Figure 24 outlines how various temperature values (w) affect the functioning of MellowMax. The figure has been extracted

from a Michael Littman (of Brown University) presentation, 'An Alternative SoftMax Operator for Reinforcement Learning' [15].

$$mm_{\omega\to\infty}([1,2,3,4]) = 4.0$$
$$mm_{\omega=100}([1,2,3,4]) = 3.9861$$
$$mm_{\omega=10}([1,2,3,4]) = 3.8614$$
$$mm_{\omega=2}([1,2,3,4]) = 3.3794$$
$$mm_{\omega=1}([1,2,3,4]) = 3.0539$$
$$mm_{\omega=0}([1,2,3,4]) = 2.5$$

Figure 24: Temparature Parameter affect on MellowMax, adapted from "An Alternative Softmax Operator for Reinforcement Learning" [15] by Michael Littman

Figure 25, on the next page, shows the pseudocode for the DeepMellow algorithm.

Algorithm 1 DeepMellow

Procedure DeepMellow(ω)

Initialize experience replay memory D
Initialize action-value function Q with random θ
~~Initialize target function Q_T with initial weights $\theta^- = \theta$~~ (modification)
for episode = 1 to M **do**
  Initialize sequence s_1 and preprocess $\phi_1 = \phi(s_1)$
  for t= 1 to T **do**
    Select a random action a_t with probability ϵ
    Otherwise, select $a_t = \arg\max_a Q(\phi_t, a; \theta)$
    Execute action a_t and observe reward r_t
    Set a new state s_{t+1} and preprocess ϕ_{t+1}
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random batch $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    if episode terminates at step $j + 1$ **then**
      set $y_j = r_j$
    else
      ~~set $y_j = r_j + \gamma \max Q_T(\phi_{j+1}, a'; \theta^-)$~~ (modification)
      set $y_j = r_j + \gamma mm_{\omega} Q(\phi_{j+1}, a'; \theta)$
    end if
    Perform gradient descent on $\{y_j - Q(\phi_j, a_j; \theta)\}^2$
    ~~Every C steps, copy $Q_T = Q$~~ (modification)
  end for
end for

(Crossed-out texts denote modifications.)

Figure 25: Deep Mellow Algorithm, adapted from "Deepmellow: Removing the Need for a Target Network in Deep Q-learning" [13] by Seungchan Ki et al.

2.6 Atari Breakout

The Deep Q-network variants explored in this project will be compared on the Atari Breakout environment provided by OpenAI. Atari Breakout is an arcade game, first released on the 13th of May 1976. The player controls a paddle that can move left or right along the bottom of the screen. The game objective is to bounce a ball off the paddle to break tiles located at the top of the screen. The players score increases, the more tiles broke. But ball speed

also increases. When the player fails to intercept the ball, a life is lost. The game terminates on losing 5 lives. Atari games are a popular benchmark in AI research, particularly in the field of deep reinforcement learning. Atari games have been used to test and compare deep reinforcement learning algorithms as they offer a simple, well-defined environment that is easily accessed through libraries such as OpenAIs gym or TF-Agents.

Atari games in deep reinforcement learning research first came to light in the 2013 paper "Playing Atari with Deep Reinforcement Learning" [28] by Volodymyr Mnih, et al. They have since been used by researchers in a number of papers, the most notable being "Human level control through deep reinforcement learning" [29]. In these papers it was demonstrated how the Deep Q-learning algorithm could be applied to a variety of Atari games, and could perform at a level comparable to or better than a human player.

In DeepMinds 2015 paper 'Human level control through deep reinforcement learning' [29], the DQN agent learned to play Atari Breakout at a superhuman level. Additionally, the DQN agent found a strategy that the DeepMind researchers hadn't anticipated. The strategy being to dig a tunnel through the edge tiles and allow the ball to dig itself back out through the central tiles.

The Deep Q-network architecture presented in 'Human level control through deep reinforcement learning' [29] approximated the action Q-values, given an environment state. The network takes 4 stacked, preprocessed game frames as input. To reduce the dimensionality of the problem, frames are cropped as 84 x 84 images and converted to greyscale. Cropping only removes redundant frame features, such as the score. Frames are stacked so that the neural network can deduce ball and paddle velocity. The input layer is followed by three convolutional layers, responsible for extracting frame features. All convolutional layers are followed by the ReLU activation function. The first convolutional layer consists of 32 8 x 8 filters/kernels that move across its input with a stride of 4. The second convolutional layer contains 64 4 x 4 filters that move across its input with a stride of 2. The final convolutional layer consists of 64 kernels of size 3 x 3 that move across its input with a stride of 1. The last hidden layer is a fully connected layer that consists of 512 neurons. The output layer has the same number of neurons as legal actions. In the case of Atari Breakout, there are 4 output neurons.

The DQN architecture presented in 'Human level control through deep reinforcement learning' employed a uniform replay buffer to reduce the correlation between experiences and a target network to improve learning stability.

2.7 Wilcoxon Rank-sum Test/Mann-Whitney U-test

The Wilcoxon rank-sum test was first introduced in a 1945 paper titled "Individual Comparisons by Ranking Methods" [50] by Frank Wilcoxon. In 1947, Mann-Whitney proposed a similar test in their paper "On a test of whether one of two random variables is stochastically larger than the other" [17]. Despite Mann-Whitney proposing the test on a later date, the test is often referred to as the Mann-Whitney U-test or Wilcoxon rank-sum interchangeably. The test is a non-parametric statistical test, meaning it does not assume an underlying data distribution. This makes it applicable to reinforcement learning, as reinforcement learning often deals with data that does not have a normal distribution, or data that has an unknown distribution. For this reason the t-test [40], which assumes a normal distribution would not be suitable.

The Mann-Whitney U-test checks whether two independent samples are drawn from populations with the same distribution. The steps involved in performing the Mann-Whitney U-test are as follows:

1. Declare the null hypothesis to be that the two samples are drawn from populations with the same distribution. Declare the alternative hypothesis to be that the two samples are not drawn from populations with the same distribution.

2. Merge the two samples.

3. Sort the merged data in ascending order.

4. Assign a rank value to each data point. The smallest data point has a rank value of 1. The rank value is incremented by 1 for each subsequent data point. If two data points have equal values, the average rank value is assigned.

5. Calculate the sum of ranks for each of the original samples.

6. Calculate the U statistic for each of the original samples, according to:

$$U_1 = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1 \quad U_2 = n_1 n_2 + \frac{n_2(n_2+1)}{2} - R_2 \tag{17}$$

   where, n denotes the sample size and R the rank sum.

7. If the smaller of the two U statistics is less than or equal to the critical value, or if the p-value is less than the chosen significance level, the null hypothesis is rejected, and it can be concluded that the two samples are drawn from populations with the same distribution.

The Scipy library, an open-source library for scientific computing, offers a clear and efficient Python implementation of the Mann-Whitney U-test. This function will be used during experimentation to ensure result validity.

2.8 Conclusion

In conclusion, this chapter has provided an investigation into the relevant literature surrounding artificial intelligence, machine learning, and reinforcement learning. The chapter began with an overview of the broader fields of artificial intelligence and machine learning, before focusing on the fundamental concepts, terminology, and approaches of reinforcement learning and deep reinforcement learning.

3 Prototype

This chapter will discuss the architectural decisions, code implementation, employed equipment, and some of the more notable problems encountered with respect to the prototype development. The chapter commences with a discussion of the available machine learning libraries. It then progresses to the code implementation components that comprise the variant architectures. The code implementation spans four files, consisting of approximately 2,000 lines of code in total. The chapter finishes with an overview of the hardware that facilitated development, and some of the problems that arose, such as GPU compatibility issues, and a memory leak.

3.1 High Level Architectural Decisions

Prior to implementation, a number of design options had to be considered and selected. Determining a suitable machine learning library, program structure and implementation medium were some of the key concerns.

Factors that influenced the choice of machine learning library were the frameworks primary language, supported architecture extensions and performance capabilities. To ensure experimentation result validity, it was important to keep the framework consistent between experiments. Thus, choosing a framework that fully met all experimentation requirements was of high importance. It should be noted that all of the considered machine learning libraries provide a significant level of abstraction.

Frameworks considered:

Pytorch : Pytorch has gained popularity among researchers due to its reputation for greater flexibility and ease in implementing novel concepts and approaches [31]. A drawback to using Pytorch was limited prior experience.

$TF - Agents$: At the outset, TF-Agents was deemed the optimal choice of library for the required experimentation. TF-Agents creates multiple reinforcement learning environments and explores each of these environments in parallel [9]. By doing so, all CPU cores are exploited,

ensuring a high GPU utilization and faster training times. However, TF-Agents does not have adequate support for Prioritized Experience Replay and thus was not suitable for the required experimentation.

Tensorflow : Tensorflow was found to be the most suitable library for the required experimentation. Tensorflow is more flexible than TF-Agents, allowing for the implementation of various architecture extensions. However, there is a trade off of significantly slower training times when compared to TF-Agents. During training (with Tensorflow), GPU utilization was found to fluctuate around 15%.

Due to their technical nature, machine learning programs can be difficult to comprehend. Consequently, organizing programs in a logical and intuitive fashion is essential. In an effort to increase code readability, I closely followed the 'Single-responsibility principle' from SOLID principles [18] - 'every class has only one responsibility'.

Additionally, to organize the code into coherent sections, a Jupyter Notebook was initially employed. The Jypyter notebook structure is illustrated in Figure 26.

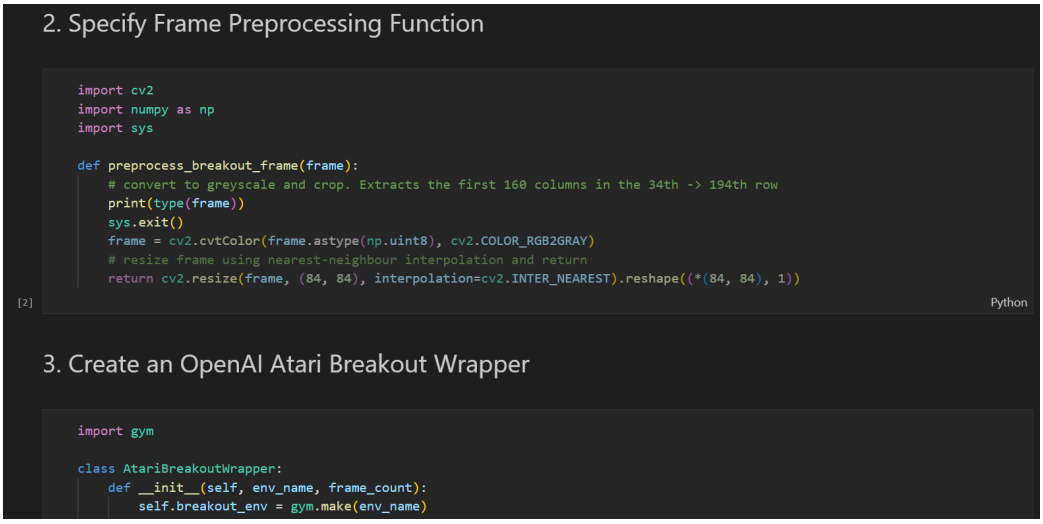


Figure 26: Jupyter Notebook Structure

Although Jupyter Notebooks offer several advantages, they posed a number of challenges in practice, specifically debugging [8]. After having to copy code over to a Python file on a number of occasions to effectively debug, I decided the pro's of Jupyter Notebooks didn't outweight their con's, and a Python file would be more suitable.

3.2 Code Implementation

The code implementation spans four files, consisting of approximately 2,000 lines of code in total.

3.2.1 Interacting with the Atari Breakout Environment

The Atari Breakout environment is created by calling Gyms make() function with a string argument, as shown in Figure 27. The 'BreakoutDeterministic-v4' string argument creates an Atari Breakout environment with deterministic gameplay, simplified graphics, and fixed frameskipping. Deterministic gameplay makes it easier to compare the performance of the DQN variants. Simplified graphics and frame-skipping decrease the computational complexity involved in training the variants.

```
self.breakout_env = gym.make('BreakoutDeterministic-v4')
```

Figure 27: Creating the Atari Breakout Environment

The above Breakout Gym environment was wrapped in a custom class that automatically performed the 'FIRE' action to start the game. Figure 28 outlines the reset_env() method of this custom class. On calling this method, the agents lives are reset to 5, the reset() method is called on the Gym environment, and the 'FIRE' action is performed to commence the game.

```
def reset_env(self):
    self.num_lives = 5
    self.frame = self.breakout_env.reset()
    self.breakout_env.step(1) # Fire to start game!
```

Figure 28: Reset Environment Method

An action is performed by calling the environments step function with an integer argument corresponding to a specific action. The custom Breakout environment wrapper also wraps the step function to 'FIRE' at the start of a game (Line 81), keep track of the agents lives (Line 78), and preprocess the returned environment frames (Line 84), as shown in Figure 29.

```
75      def step(self, action):
76          observation, reward_returned, _, info = self.breakout_env.step(action)
77          if info['lives'] < self.num_lives:
78              self.num_lives = info['lives']
79              life_lost = True
80              if self.num_lives!=0:
81                  self.breakout_env.step(1) # Fire to start game!
82          else:
83              life_lost = False
84          preprocessed_frame = preprocess_breakout_frame(observation)
```

Figure 29: Step Method

3.2.2 Joystick control settings

The BreakoutDeterministic-v4 environment offers four possible actions. The list of available actions can be obtained by calling the environments get_action_meanings() method, as shown in Figure 30 and Figure 31.

```
breakout_env = gym.make('BreakoutDeterministic-v4')
print(breakout_env.get_action_meanings())
```

Figure 30: Action Meanings Function

Figure 31: Available Actions

Each action has an index value associated with it, according to its position in the array outlined in Figure 31. The Atari Breakout Gym environment initially had a 'RIGHTFIRE' and 'LEFTFIRE' action, but they were removed as they were largely deemed unecessary. The 'NOOP' action stands for 'NO OPERATION' and can be used at the start of a game to introduce a degree of randomness. This is accomplished by the agent performing a random number of 'NOOP' actions immediately on game commencement. By applying uncertainty to the initial game states, the agent can't memorize the complete sequence of optimal actions. Introducing stochasticity in this fashion was believed to be sufficient until a 2018 paper titled 'Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents' [16] was published. This paper argued that the execution of a random number of 'NOOP' actions at the start of a game was not as effective at preventing memorization as was previously believed. As introducing randomness during training would cloud the DQN comparison results, this method was not employed.

3.2.3 Frame Preprocessing

The code outlined in Figure 32 receives an Atari Breakout frame, converts it to a greyscale image (Line 52), and crops it (Line 53-54). By converting the frames to a single channel grey image, the dimensionality is reduced and its easier for the agent to learn useful features and patterns in the game. By cropping the frame as an 84 by 84 image, irrelevant image data is removed which makes the frames easier to process and analyze.

```
51  def preprocess_breakout_frame(frame):
52      frame = cv2.cvtColor(frame.astype(np.uint8), cv2.COLOR_RGB2GRAY)
53      resized_image =  cv2.resize(frame, (84, 84), interpolation=cv2.INTER_NEAREST)
54      return resized_image.reshape((*(84, 84), 1))
```

Figure 32: Frame preprocessing

3.2.4 Vanilla Deep Q-network

The vanilla Deep Q-network was discussed in section 2.5.1. The vanilla Deep Q-network uses the same network to acquire Q-values and calculate the target Q-values. Figure 33 demonstrates how the same Deep Q-network (self.dqn_architecture) is used to acquire the predicted Q-values (Line 337) and calculate the target Q-values (Line 334).

```
334    q_values_in_next_state = self.dqn_architecture(next_states_tensor, training=False)
335    best_q_values_in_next_state = q_values_in_next_state.numpy().max(axis=1)
336    target_q_values = rewards + (gamma*best_q_values_in_next_state * (1-terminal_flags))
337    predicted_q_values = self.dqn_architecture(states)
```

Figure 33: Vanilla DQN Aquiring Q-values

Figure 34 demonstrates how gradient descent is performed. First, the Q-values corresponding to the taken actions are extracted (Line 346). Then the target Q-values are subtracted from the predicted Q-values to get the error value (Line 347). The Huber loss is applied (Line 348), and gradient descent is performed using this loss (Line 349).

```
346        Q = tf.reduce_sum(tf.multiply(predicted_q_values, one_hot_actions), axis=1)
347        error = Q - target_q_values
348        loss = tf.keras.losses.Huber()(target_q_values, Q)
349    dqn_architecture_gradients = tape.gradient(loss, self.dqn_architecture.trainable_variables)
```

Figure 34: Vanilla DQN Gradient Descent

The tf.GradientTape() API provided by Tensorflow is used for automatic differentiation, as outlined in Figure 35. Tensorflow records operations performed inside the context of the tape and then uses reverse mode differentiation to compute the gradient of the recorded operations. Automatic

differentiation is more efficient at computing gradients [3]. It saves time and effort as manually computing the gradients isn't necessary.

```
with tf.GradientTape() as tape:
    q_values_current_state_dqn = self.dqn_architecture(states)
    one_hot_actions = tf.keras.utils.to_categorical(actions, self.num_legal_actions, dtype=np.float32)
    Q = tf.reduce_sum(tf.multiply(q_values_current_state_dqn, one_hot_actions), axis=1)
    error = Q - target_q_values
    loss = tf.keras.losses.Huber()(target_q_values, Q)
dqn_architecture_gradients = tape.gradient(loss, self.dqn_architecture.trainable_variables)
```

Figure 35: Automatic Differentiation

In this DQN variant, a uniform replay buffer is used to reduce correlations between observations. A replay buffer index is acquired randomly, as shown in Figure 36. The actions, rewards, frames, and terminal_flags arrays are then sampled accordingly.

```
index = random.randint(self.num_stacked_frames, self.total_indexes_written_to - 1)
```

Figure 36: Uniform Replay Buffer Sampling

3.2.5 Double Deep Q-network

As outlined in section 2.5.3, the Double Deep Q-network mitigates the over-estimation apparent in the DQN with fixed Q-targets. It does this by using the online model when selecting the best actions for the next states, instead of the target model.

```
339    best_action_in_next_state_dqn = self.dqn_architecture(new_states_tensor, training=False).numpy().argmax(axis=1)
340    target_q_network_q_values = self.target_dqn_architecture(new_states_tensor, training=False).numpy()
341    optimal_q_value_in_next_state_target_dqn = target_q_network_q_values[range(batch_size), best_action_in_next_state_dqn]
342    target_q_values = rewards + (gamma*optimal_q_value_in_next_state_target_dqn * (1-terminal_flags))
```

Figure 37: Double DQN - Acquiring Q-values

Line 339 of Figure 37 uses the online model to get the best action in the next state. Line 340 uses the target model to return all Q-values for the next state. The optimal Q-values, according to the online model, are then retrieved from the Q-values returned by the target model (Line 341), and the targets are calculated (Line 342). Gradient descent is then performed

similarly to that outlined in Figure 34.

Figure 38 describes the network structure. The network structure of the Double DQN is the same as the Vanilla DQN.

```
96  def create_double_dqn_architecture(num_actions, input_shape, num_stacked_frames, learning_rate):
97      model_input = tf.keras.Input(shape=(input_shape[0], input_shape[1], num_stacked_frames))
98      x = tf.keras.layers.Lambda(lambda layer: layer / 255)(model_input)
99      x = tf.keras.layers.Conv2D(32, 8, strides=4, activation="relu")(x)
100     x = tf.keras.layers.Conv2D(64, 4, strides=2, activation="relu")(x)
101     x = tf.keras.layers.Conv2D(64, 3, strides=1, activation="relu")(x)
102     x = tf.keras.layers.Flatten()(x)
103     x = tf.keras.layers.Dense(512, activation="relu")(x)
104     action = tf.keras.layers.Dense(num_actions, activation="linear")(x)
```

Figure 38: Double DQN Code

The network is passed an 84 x 84 x 4 tensor where each 84 x 84 tensor is a preprocessed breakout frame. Stacking frames allows the network to deduce ball velocity. Line 97 of Figure 38 specifies the expected network input according to these dimensions. Line 98 of Figure 38 is responsible for normalizing the input values. Prior to normalization, each tensor element is a value in the range 0 to 255. By dividing each value by 255, each element is forced to the range 0 to 1. Normalizing input can help the network converge faster. Lines 99 to 101 define a number of convolutional layers. A convolutional layer applies a number of filters to the input data and produces a new set of tensors that can be passed into another convolutional layer. The output of the final convolutional layer on line 101 of Figure 38 is a large feature map, where each feature represents some characteristic of the original input. Lastly, the Keras Flatten() function is called on line 102 to convert the multi-dimensional tensor to a one-dimensional tensor. By passing the output of the Flatten() function to a fully connected layer, the network can deduce the optimal action to take based on the characteristics extracted from the original input.

Epsilon is decreased from 1 to 0.01 over 25 million frames, as shown in Figure 39, on the next page.

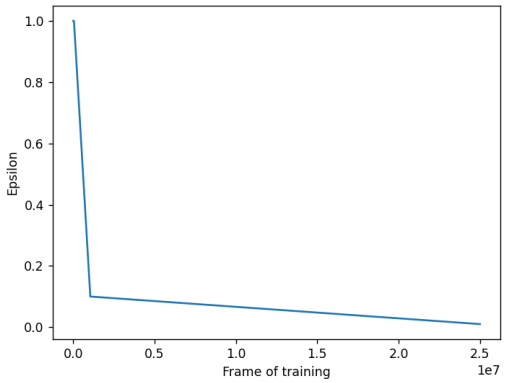


Figure 39: Epsilon Decrease Graph

Epsilon is decreased from 1 to 0.1 over the first million frames, and from 0.1 to 0.01 over the remaining frames. Setting epsilon to a high value initially is advantageous as it exposes the agent to more of the state space [41].

The epsilon value is calculated according to the frame number, as shown in Figure 40.

```
279  if frame_number < self.replay_buffer_start_size:
280      return self.initial_epsilon
281  elif (frame_number >= self.replay_buffer_start_size and frame_number <
282          self.replay_buffer_start_size + self.num_frames_to_decrease_epsilon_over):
283      return self.initial_epsilon_decrease*frame_number + self.first_intercept
284  elif frame_number >= self.replay_buffer_start_size + self.num_frames_to_decrease_epsilon_over:
285      return self.second_epsilon_decrease*frame_number+self.second_intercept
```

Figure 40: Epsilon Calculation Code

For the first 50,000 frames (as the replay buffer is filling) epsilon is kept constant at a value of 1.0, as shown on line 280 of Figure 40. From frames 50,000 to 1,050,000 a smaller value for epsilon is returned. The value for epsilon is calculated by multiplying the frame number by the initial epsilon decrease per frame, as shown on line 282 of Figure 40. This initial epsilon decrease ensures an epsilon value of 0.1 is reached at frame 1,050,000. The intercept value counteracts any decrease in epsilon that would occur while the replay buffer is filling. For frames greater than 1,050,000 epsilon decreases at a slower rate. The epsilon value is calculated by multiplying the frame number by a smaller epsilon decrease per frame, as shown on line 285 of Figure 40.

Again, an intercept value is added to counteract any decrease in epsilon that would occur prior to frame 1,050,000.

3.2.6 Dueling Deep Q-network

As outlined in section 2.5.5, the dueling network architecture diverges into two streams, one to learn the state value and the other to learn the advantage of taking a particular action over the other actions. The dueling network architecture learns to estimate the value of each action in a state more effectively and efficiently than the prior architectures [47]. Similarly to the Double DQN, the weights of the online network are copied over to the target network every 1,000 frames. The network inputs are also normalised before being passed to the convolutional layers. Line 93 of Figure 41 demonstrates how the network diverges into two separate streams.

```
93      val_stream, adv_stream = tf.keras.layers.Lambda(lambda w: tf.split(w, 2, 3))(x)
94      val_stream = tf.keras.layers.Flatten()(val_stream)
95      val = tf.keras.layers.Dense(1, kernel_initializer=tf.keras.initializers.VarianceScaling(scale=2.))(val_stream)
96      adv_stream = tf.keras.layers.Flatten()(adv_stream)
97      adv = tf.keras.layers.Dense(num_actions, kernel_initializer=tf.keras.initializers.VarianceScaling(scale=2.))(adv_stream)
98      reduce_mean = tf.keras.layers.Lambda(lambda w: tf.reduce_mean(w, axis=1, keepdims=True))
99      q_vals = tf.keras.layers.Add()([val, tf.keras.layers.Subtract()([adv, reduce_mean(adv)])])
100     model = tf.keras.Model(model_input, q_vals)
```

Figure 41: Dueling DQN Code

The tf.split() function splits a tensor into a number of sub tensors. The first argument in the function is the tensor being split, the second argument is the number of sub tensors, and the final argument is the axis along which to split the tensor. The value stream connects to a dense layer of one node (see line 95 of Figure 41), the advantage stream connects to a dense layer of four nodes (see line 97 of Figure 41), one to represent each of the actions. The Q-values are reassembled from the outputs of these streams and returned from the function (see line 99 of figure 41).

3.2.7 Prioritized Experience Replay

As outlined in section 2.5.4, the prioritized experience replay buffer records the temporal difference error (target value - predicted value) associated with each buffer entry and allows for the sampling of entries according to their importance. Buffer entries with larger temporal difference errors are weighted

more heavily during sampling. This ultimately prioritizes the frames that will contribute most to the learning process. Initially, the prioritized experience replay buffer implementation relied on arrays and had very poor performance. It would have added an additional 8 to 9 days to training time. By implementing a SumTree this issue was resolved as the SumTree has a sampling time complexity of O(log n). The array had a sampling time complexity of O(n).

The SumTree class has two main methods, get() and add(). The get() method is responsible for sampling the prioritized experience replay buffer according to the assigned priorities. This method takes a random value between 0 and the cumulative sum of all priorities, as shown on line 32 of Figure 42.

```
32      def get(self, rand_val):
33          idx = 0
34          while 2 * idx + 1 < len(self.nodes):
35              left, right = 2*idx + 1, 2*idx + 2
36              if rand_val <= self.nodes[left]:
37                  idx = left
38              else:
39                  idx = right
40                  rand_val = rand_val - self.nodes[left]
41          data_idx = idx - self.size + 1
42          frame_idx =  (data_idx) + self.size - 1
43          return self.nodes[frame_idx], self.data[data_idx]
```

Figure 42: SumTree get() Method

Between lines 33 and line 40 of Figure 42, the random value is passed through the tree. If the random value is less than the child node value, it moves to the left child node, otherwise it moves to the right child node and the left child node is subtracted from the random value. On line 41 to 43 of Figure 42, a priority and respective index is acquired and returned.

The add() method is responsible for adding an entry to the Sum tree. The add() method takes two arguments, as shown on line 26 of Figure 43. These values are the data value to add to the tree and the priority to associate with that value. Line 27 adds the value to an array storing the data values and

line 28 updates the parent nodes, altering the sum of child nodes. Lines 29 and 30 of Figure 43 alter values that specify the next tree index and the tree size.

```
26      def add(self, priority, data): # add(priority, index)
27          self.data[self.count] = data
28          self.update(self.count, priority) # update(data_index_position, priority)
29          self.count = (self.count + 1) % self.size
30          self.real_size = min(self.size, self.real_size + 1)
```

Figure 43: SumTree add() Method

3.2.8 DeepMellow Q-network

As outlined in section 2.5.7, the DeepMellow Q-network removes the need for a target network by employing the MellowMax functional regularizer. However, on implementing the MellowMax function, it was found to yield overflow errors due to the exponentiation of large values. On further research, the paper 'An Alternative SoftMax Operator for Reinforcement Learning' [1] was discovered to address this problem with a MellowMax variant, as shown in Figure 44.

$$\frac{\log(\frac{1}{n}\sum_{i=1}^{n} e^{\omega x_i})}{\omega} = c + \frac{\log(\frac{1}{n}\sum_{i=1}^{n} e^{\omega (x_i - c)})}{\omega}$$

Figure 44: MellowMax Variant

where, c is a constant equal to the max Q-value.

The code implementation of the MellowMax variant is outlined in Figure 45. Line 249 establishes the 'c' values by extracting the max Q-values. Line 250 and 251 subtract these 'c' values from the Q-values and multiply the resultant values by the temperature value. Lines 252 to 257 perform further operations to calculate the MellowMax values, before they're returned from

the function on line 258.

```
247  def mellow_max_in_practice(q_values):
248      q_values = np.float128(q_values.numpy())
249      c_vals = np.max(q_values, axis=1)
250      sub = np.subtract(q_values, c_vals[:, np.newaxis], dtype=np.float128)
251      powers = np.multiply(sub, DEEP_MELLOW_TEMPERATURE_VALUE, dtype=np.float128)
252      summation_values = np.exp(powers, dtype=np.float128)
253      summation = np.sum(summation_values, axis=1, dtype=np.float128)
254      val_for_log = np.multiply(summation, MELLOW_MAX_1_OVER_N)
255      numerator = np.log(val_for_log, dtype=np.float128)
256      mellow_vals_before_constant = np.divide(numerator, DEEP_MELLOW_TEMPERATURE_VALUE, dtype=np.float128)
257      mellow_vals_after_constant = mellow_vals_before_constant + c_vals
258      return np.float32(mellow_vals_after_constant)
```

Figure 45: MellowMax Variant Code

3.2.9 Recording Kernels

Every 1,000 games played, six kernels from the first convolutional layer were recorded. Every 1,000 games, weights were obtained from the first convolutional layer, as shown in Figure 46.

```
filters, _ = main_dqn.layers[2].get_weights()
```

Figure 46: Extracting Weights

These weights were then displayed as a greyscale matplotlib plot, as outlined in Figure 47, on the next page. The code comments describe the workings of the code.

```
for i in range(n_filters):
    # get the filter
    f = filters[:, :, :, i]
    # plot each kernel channel separately
    for j in range(4):
        # specify subplot and turn of axis
        ax = pyplot.subplot(n_filters, 4, ix)
        ax.set_xticks([])
        ax.set_yticks([])
        # plot kernel channel in grayscale
        pyplot.imshow(f[:, :, j], cmap='gray')
        ix += 1
```

Figure 47: Kernel Plot Code

Finally, the plot was converted to a png image, as described in Figure 48.

```
buf = io.BytesIO()
pyplot.savefig(buf, format='png')
buf.seek(0)
```

Figure 48: Converting Kernel Plot to PNG

An example kernel visualisation is shown in Figure 49, on the next page. Smaller values are represented as shades of white. Larger values are represented as shades of black.

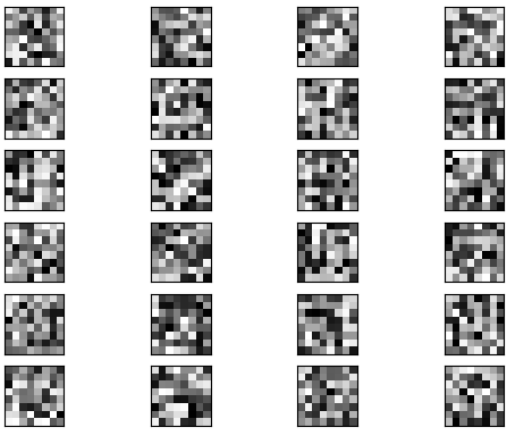


Figure 49: Kernel Visualization

3.2.10 Recording Video

Every 1,000 games played, the agent interacted with a test environment. The environment frames were recorded in an array, converted to a Python Image Library image, stacked, and saved as a GIF video. Line 337 of Figure 50 outlines the code responsible for recording the frames produced by the agents interactions with the test environment.

```
371 for i in range(MAX_EPISODE_LENGTH):
372     # Get action for breakout environment state
373     action = breakout_agent.evaluate(test_breakout_env.state)
374 
375     # Take step
376     frame, reward, terminal, life_lost = test_breakout_env.step(action)
377     observations.append(frame)
378     if terminal:
379         break
```

Figure 50: Recording Game Frames

The recorded frames are then converted to a PIL image (Line 381, Fig. 51) and saved in the specified directory (Line 382, Fig. 51), as outlined in Figure 51.

```
381    pil_video_frames = [PIL.Image.fromarray(np.squeeze(frame, axis=2)) for frame in observations[:len(observations)]]
382    pil_video_frames[0].save(video_path, format='GIF', append_images=pil_video_frames[1:], save_all=True, duration=30)
```

Figure 51: Saving Video

3.3 Equipment Employed

To facilitate experimentation, a custom PC Build was required. The main PC components are outlined below. For further information regarding the PC build, see Appendix A.1. The full specification of the custom built PC can be found at https://pcpartpicker.com/list/gFLwLs.

CPU: Intel Core i5-12600K 3.7 GHz 10-Core Processor.

GPU: MSI GeForce RTX 3060.

RAM: Corsair Vengeance 32 GB DDR5-5600 Memory.

Motherboard: MSI MAG Z790 TOMAHAWK.

CPU Cooler and Fans: Be Quiet! Dark Rock Pro 4.

3.4 Problems Encountered During Experimentation

3.4.1 GPU Driver Compatibility

Following assembly of the custom PC, installation of a linux distribution was required. The first choice of distribution was Ubuntu but subsequent to two days of debugging significant GPU driver issues, Ubuntu was rendered unsuitable. These issues led to the selection of Fedora as a viable alternative. If time permitted, additional resources would have been allocated to resolving the GPU driver issues.

Ubuntu GPU driver issues were first suspected following first boot up. Installing Ubuntu from USB via the standard method resulted in a black screen from which it was impossible to restart or shutdown. However, following brief

investigation, it was found that the issue was not an entirely uncommon problem. The Ubuntu installers startup menu is sometimes incompatible with certain graphics cards. To resolve this issue one needs to boot up with the 'nomodeset' parameter set. This parameter tells the kernel to not start video drivers until the system is up and running. This way the system wont try to use the GPU for the installers startup menu. Setting the 'nomodeset' parameter allowed for the successful installation of Ubuntu and the required GPU drivers. Following these changes, the system had to be rebooted. It was suspecting that the system would boot up correctly without having to set the 'nomodeset parameter' or boot in recovery mode. Unfortunately, the screen in Figure 52 was shown, from which the system had to be manually turned off.

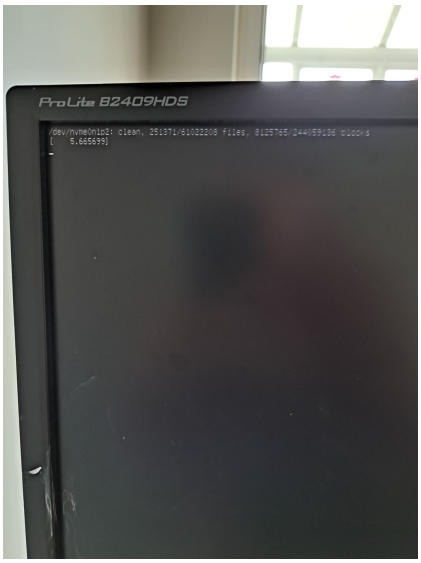


Figure 52: Ubuntu Boot Screen

In an effort to resolve the issue, the system was rebooted and the grub menu was opened. By going to Advance Options for Ubuntu → recovery mode, I could access the root shell prompt and check the status of GDM3. GDM3 or the GNOME Display Manager is responsible for providing the graphical login capabilities in a Linux distribution. On checking the status I found that GDM3 was 'inactive and dead'.

By enabling GDM3 from the prompt and continuing the boot, it was suspected that the login screen would be displayed. However, a very similar screen to the first was displayed instead. From this new screen the tty could be opened and one could successfully login but without the GUI. The GDM3 status was rechecked in the tty and it was found to be active. Various packages, display managers, and Nvidia drivers were installed to try resolve the issue. Additionally, the grub menu settings were altered, but the issue did not resolve. The Nvidia forum community suspected mismatching drivers and wrong grub/gdm settings enabled. The solution was to do a fresh install but doing so didn't resolve the issues. This issue is likely to be revisited following the completion of this project.

3.4.2 Memory Leak

The first attempt at training the Atari Breakout reinforcement learning agent was unsuccessful. Following approximately 90 minutes of training, it was noticed that the screen had started freezing, becoming unresponsive shortly thereafter. The first suspect was the GPU as the GPU fans hadn't been spinning. But subsequent to reseating the GPU and a brief investigation it was found that some GPU fans only become active above a certain temperature. To investigate further the program was reran while closely monitoring the system. RAM usage grew as the program ran but this wasn't considered to be a memory leak as it would take a few hours for the replay buffer to reach full capacity anyway. The first solution attempt involved reducing the memory requirements of the replay buffer. In doing so there was found to be a lot of unused memory. The actions were stored in a numpy array of type int32 but you could represent all actions with two bits. The rewards were stored in a numpy array of type float32 but all rewards could also be represented with just two bits. The terminal flags were stored in a numpy array of type bool. On researching numpys bool type, it was found that it takes up 8 bits despite only needing 1. To improve memory usage, the above mentioned arrays were swapped for bitarrays. The Python bitarray library allows for the efficient representation of boolean arrays. By interpreting each bool as a bit, the replay buffer memory requirements were significantly reduced. On making this change the program was reran but the RAM usage was still growing disproportionately to what was expected.

The notion of there being a memory leak was becoming more plausible following the prior attempted solution. But without halting the addition of new frames to the replay buffer, it would be difficult to know. Two lines of code were added to the start of the replay buffers add_observation() method so that the method would return immediately if the replay buffer size was greater than 50,000. The program was reran and the RAM usage continued to rise indicating that there was in fact a memory leak and it wasn't associated with the replay buffer.

The next step was to locate the memory leak. Initially there was a bias towards the issue being in the gradient descent code as there was a warning being logged (relevant to the Adam optimizer) that is often associated with memory leaks. For this reason quite a few hours were spent looking in the wrong place. It was only after looking at the rest of the code that the memory leak was found. To narrow down the defective code, the RAM usage was recorded after 600 games, prior to making any changes. Various parts of the code where the memory leak was suspected were then commented. Dummy values were returned from any methods that were critical to the functioning of the program. Once done, the program was reran and the RAM usage was recorded after 600 games. Following several iterations, the memory leak was narrowed down to the following line:

```
best_action_in_next_state_dqn =
self.dqn_architecture.predict(new_states).argmax(axis=1)
```

More specifically, the tf.keras.Model.predict() function.

The tf.keras.Model.predict() function is a tf.function and a tf.function treats any pure Python value as an opaque object, building a separate graph for each set of Python arguments it encounters. Passing python scalars or lists as an argument to a tf.function as was being done will always build a new graph. To avoid this, numeric arguments need to be passed as tensors. This would suggest that a graph was getting created every time the predict function was being called, unnecessarily using up RAM. However, despite converting the Python lists to Tensorflow tensors, the memory leak was still apparent. On further investigation, a similar bug was found filed on the Tensorflow Github repository. The suggested solution was to explicitly call the python garbage

collector in addition to the already implemented solutions. One possible explanation is that the tensor conversion is creating cyclical references that evade the Python garbage collector. Explicitly calling the garbage collector every 250 games resolved the memory leak.

3.5 Conclusion

In conclusion, this chapter started by discussing high level architectural decisions such as the machine learning library and implementation medium. The chapter then progressed to discussing the key code components that comprise the Atari Breakout environment wrapper, and variant architectures. The chapter finished with an overview of the hardware components that facilitated experimentation and some of the more notable problems that were encountered.

4 Empirical Studies

This chapter will outline the results of training the four DQN variants, with respect to the metrics outlined in section 1.3.4 and section 4.1.

4.1 Recorded Metrics

As was outlined in section 1.3.4, the following training progress data was recorded to evaluate the DQN variants:

- **Kernel visualizations** : The kernel visualizations gave insight into the effects of gradient descent. Kernel visualizations were compared using the root mean squared error between images. The Python Skikit-learn library was used to calculate the mean squared error between images and the numpy.sqrt() function was used to get the root of the mean squared error. If the root mean squared error is '0', the images are identical. As images decrease in similarity, the root mean squared error increases.

- **Reward** : Reward fluctuations were visualised using a Tensorboard graph. From which, convergence, divergence, training stability, ability to generalize, and duration could be concluded. Tensorboard also plots a (smoothed) reward, deduced by applying a technique known as Exponential Moving Average (EMA). The Exponential Moving Average calculates the average of prior rewards, weighting recent rewards more heavily. The smoothed reward plot is easier to interpret as it's less sensitive to noise/short term fluctuations. To calculate a smoothed reward, a smoothing factor α is first chosen. Typically, α is a value in the range 0 to 1. From the second data point on, the smoothed reward can be calculated according to the following equation:

$$smoothed\_reward_t = \alpha(raw\_reward_t) + (1 - \alpha)smoothed\_reward_{t-1} \tag{18}$$

- **Loss** : Loss fluctuations were also visualised using a Tensorboard graph. Similarly to the reward plot, the loss plot allowed convergence, divergence, training stability, ability to generalize, and duration to be evaluated. Smoothed Loss was also plotted using the Exponential Moving Average (EMA) technique.

4.2 Double Deep Q-network

Table 1: Double DQN Training Summary

Training Summary Data	
Start date:	2023-02-21 22:19:01
Completion Date:	2023-02-23 08:49:48
Duration:	1.437 days
Number of Frames:	25,000,000
Number of Games:	19,200
Highest Reward (Smoothed):	422.1
Highest Reward (Value):	455
Final Reward (Smoothed):	415
Final Reward (Value):	436.4

The Double DQN Atari Breakout agent implemented as part of this project achieved a reward approximately 13% higher than the Double DQN Atari Breakout agent implemented in 'Human level control through Deep Reinforcement learning' [29], while being trained on half the number of frames. The Double DQN agent implemented as part of this project was trained on 25 million Atari Breakout frames. The agent took 1.437 days to train and achieved a 'Highest Reward (value)' of 455. The Double DQN agent implemented as part of 'Human level control through Deep Reinforcement learning' [29] was trained on 50 million frames and achieved an average reward of 401.2.

4.2.1 Reward

Figure 53 is a graph of the cumulative reward received by the reinforcement learning agent per episode of Atari Breakout played. While the reward graph doesn't show complete convergence, it does show the agent's performance gradually improving over time. Over the first 8 million frames, the agents rewards were relatively small. But as training progressed, the agent was exposed to more of the state space and had further opportunities to learn

from its mistakes. Rewards grew at a faster pace following 8 million frames. The graph doesn't show a clear leveling off to indicate convergence, but there does appear to be a slowing down in the rate at which the reward per episode is increasing. Further training may not result in as significant improvements. The highest reward received by the reinforcement learning agent was 455. The reward received by the agent on the last episode of training was 415. Both of which are respectable values.

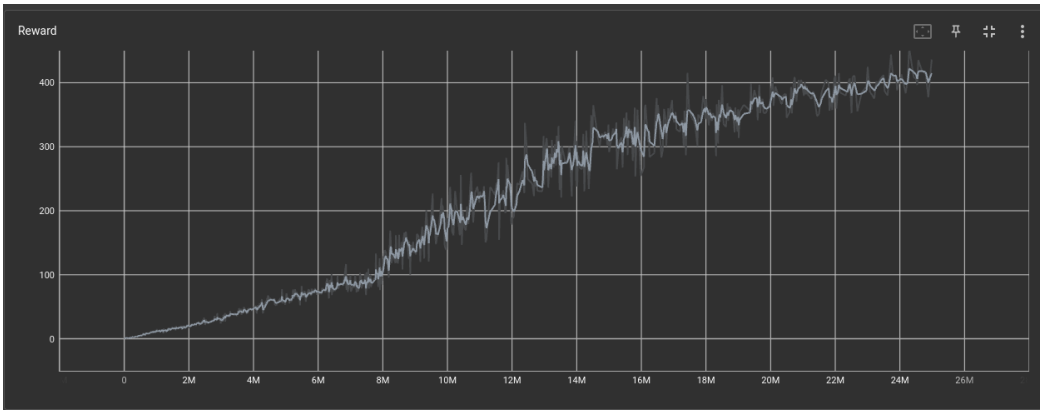


Figure 53: Double DQN Reward Plot

4.2.2 Loss

Figure 54 is a graph of the Huber loss received by the agent throughout training. A large loss at any given time generally means the network predictions are significantly different from the true values, and so a proportional adjustment needs to be applied to the network weights. An ideal graph would show an increase in loss at the start of training as the network weights are initialized to random values. As training progresses, the loss should start decreasing. This should be followed by a leveling off to indicate the true and predicted values are very similar. Figure 54 shows a sharp increase in loss following the commencement of training and a steady decrease towards the end of training. If training was to continue on, a plateau in loss would be expected.

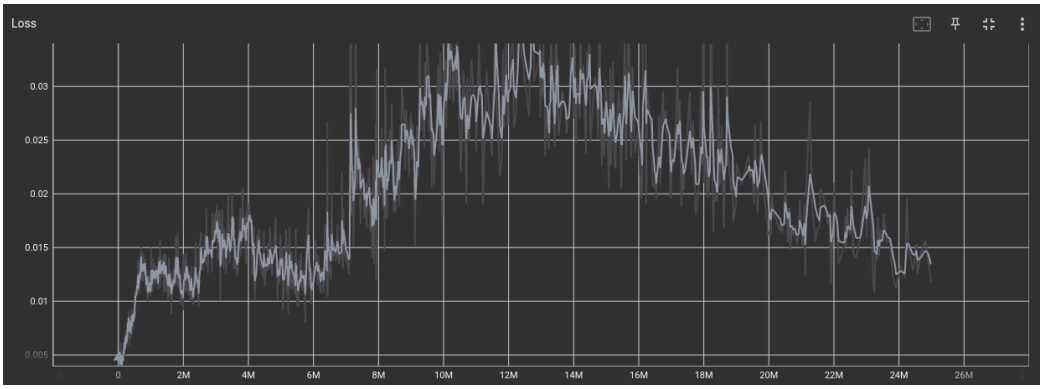


Figure 54: Double DQN Loss Plot

4.2.3 Kernel Visualization:

Every 1,000 games, 6 filters from the first convolutional layer of the neural network were recorded. A PNG image is created where each weight is represented as a greyscale color (smaller values are black, larger values are white). Figure 55 displays the kernels following the first game. A lot of the kernels contain darker squares, representing smaller valued weights.

Figure 55: Double DQN Game 1 Kernels

Figure 56 displays the same kernels following game 19,000. In this image the kernels are composed of a lot of lighter coloured squares, indicating larger valued weights. This demonstrates how the kernels were transformed during training to better extract features/characteristics from the Atari Breakout frames. Comparing both images using the root mean squared error, obtained a value of 0.11.

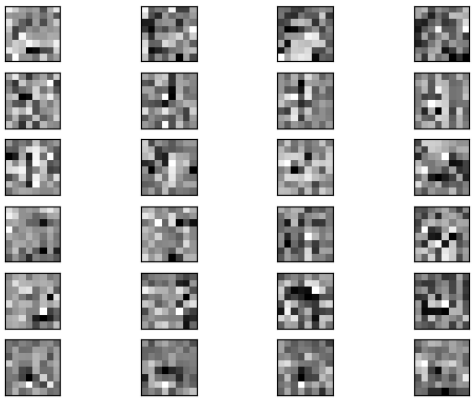


Figure 56: Double DQN Game 19,000 Kernels

4.2.4 Video

Every 1,000 games, a GIF video of the agent interacting with the Breakout environment was recorded. Below are two screenshots from such video recordings. Figure 57 is of the agent interacting with the environment following one game of training. Figure 58 is of the agent interacting with the environment following 19,224 games of training. In Figure 57, the paddle is stagnant as the ball moves from the top right of the screen to the bottom left. In Figure 58, the agent interacts with the environment trying to maximize the returned reward. For the full videos, see https://drive.google.com/drive/folders/1pXJl3pbCNuXbyF7jM6SL1b_0233Ox-qV

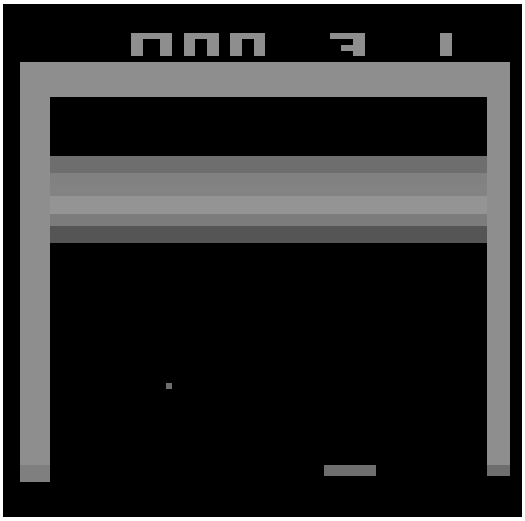


Figure 57: Double DQN Game 1 Screenshot

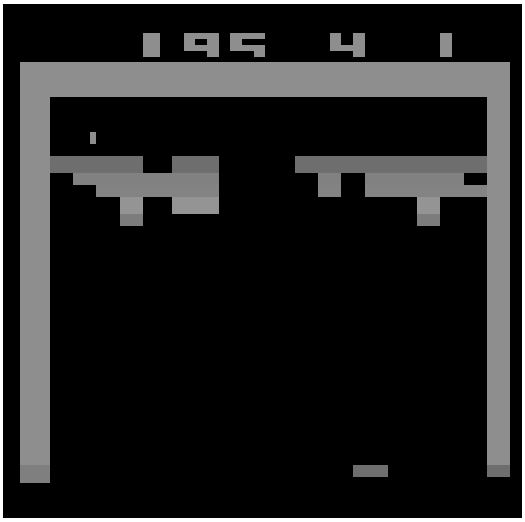


Figure 58: Double DQN Game 19,224 Screenshot

4.2.5 Statistical Significance

To ensure result validity, the Double DQN Breakout agent was trained a second time. The Mann-Whitney U-test was employed to ensure there was no significant difference between the reward plot of the first and second training instances. The Mann-Whitney U-test is also known as the Wilcoxon rank-sum test. It is a non-parametric statistical test that can be used to determine whether there is a significant difference between the distributions of two samples. Using the rewards recorded by Tensorboard, a p-value of 0.55 was obtained when comparing the two training instances. This indicates that there's no significant difference between the first training instance and second training instance.

4.3 Dueling Deep Q-network

Table 2: Dueling DQN Training Summary

Training Summary Data	
Start date:	2023-03-01 10:58:54
Completion Date:	2023-03-03 06:01:15
Duration:	1.792 days
Number of Frames:	25,000,000
Number of Games:	19,300
Highest Reward (Smoothed):	404.6
Highest Reward (Value):	442.1
Final Reward (Smoothed):	357.1
Final Reward (Value):	366.9

The Dueling DQN Atari Breakout agent was trained on 25 million frames and achieved a 'Highest Reward (Value)' of 442.1. Relevant literature does not explicitly state achieved rewards, but does make normalized comparisons to variant architectures. The Dueling DQN Atari Breakout agent implemented as part of 'Dueling Network Architectures for Deep Reinforcement Learning' [47] performed 17.56% worse than the Double DQN Atari Breakout agent implemented as part of 'Deep Reinforcement Learning with Double Q-Learning' [46]. The Dueling DQN agent implemented as part of this project performed approximately 2.85% worse than the Double DQN agent implemented as part of this project, using the 'Highest Reward (Value)'. Using the 'Highest Reward (Smoothed)', the Dueling DQN performed approximately 4.15% worse.

4.3.1 Reward

Figure 59 is a graph of the reward received by the reinforcement learning agent per episode of Atari Breakout played, using a Dueling Deep Q-network. Similarly to the Double DQN reward plot, it doesn't show complete convergence, but does show the agent's performance gradually improving over time.

The highest reward received by the reinforcement learning agent was 442.1. The reward received by the agent on the last episode of training was 366.9. In line with the literature, the Dueling Deep Q-network underperforms the Double Deep Q-network when using the Atari Breakout environment.

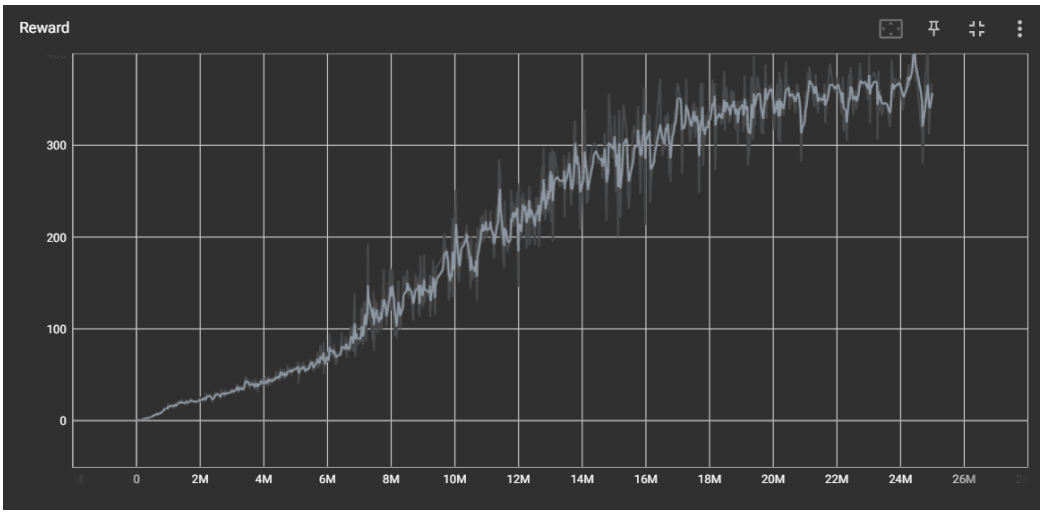


Figure 59: Dueling DQN Reward Plot

4.3.2 Loss

Figure 60, on the next page, is a graph of the Huber loss received by the agent during training. Similarly to the Double DQN loss graph, a sharp increase is observed at the start of training, followed by a steady decline towards the end.

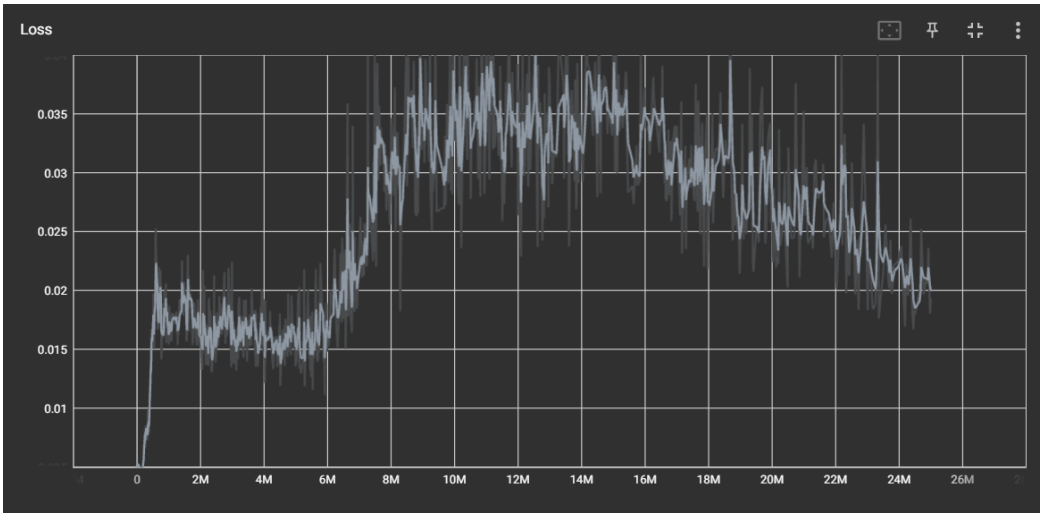


Figure 60: Dueling DQN Loss Plot

4.3.3 Kernel Visualization

Similarly to the Double DQN kernels, a noticeable difference in the Dueling DQN kernels can be seen between game 1 and game 19,000. The root mean squared error between kernel visualizations is 0.06. See Figures 61 and 62.

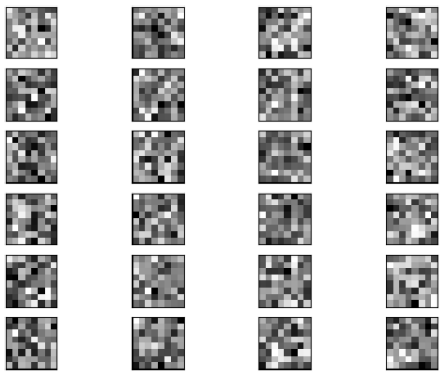


Figure 61: Dueling DQN Game 1 Kernels

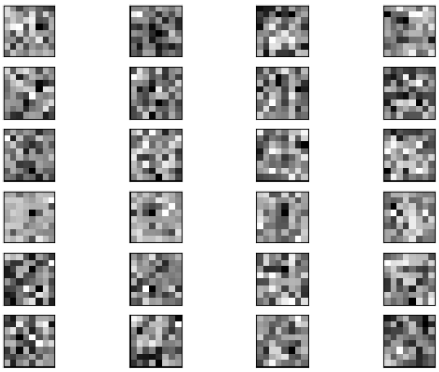


Figure 62: Dueling DQN Game 19,000 Kernels

4.3.4 Video

Figure 63 shows the agent interacting with the environment following one game of training. Figure 64 shows the agent interacting with the environment following 19,336 games of training. The agents performance is very similar to that of the Double DQN reinforcement learning agent. For the full videos, see https://drive.google.com/drive/folders/1xHLQ14mH3fVTfiO2G7Gu_EoBCXXTgLv8

Figure 63: Dueling DQN Game 1 Screenshot

Figure 64: Dueling DQN Game 19,336 Screenshot

4.3.5 Statistical Significance

To ensure result validity, the Dueling DQN Breakout agent was trained a second time and the Mann-Whitney U-test was once again employed. Using the rewards recorded by Tensorboard, a p-value of 0.5 was obtained when comparing the two training instances. This indicates that there's no significant difference between the first training instance and second training instance.

4.4 Dueling Deep Q-network with PER

Table 3: Dueling DQN with PER Training Summary

Training Summary Data	
Start date:	2023-03-15 19:03:37
Completion Date:	2023-03-20 21:14:39
Duration:	5.09 days
Number of Frames:	25,000,000
Number of Games:	19,700
Highest Reward (Smoothed):	373.1
Highest Reward (Value):	410.8
Final Reward (Smoothed):	332.2
Final Reward (Value):	352.4

The Dueling Deep Q-network with prioritized experience replay was trained on 25 million frames over 5.09 days. The Dueling Deep Q-network with PER achieved a 'Highest Reward (Value)' of 410.8. Literature evaluating the Dueling Deep Q-network with prioritized experience replay is insufficient and can not act as a basis to evaluate this projects implementation.

4.4.1 Reward

Figure 65 outlines the reward received by the Atari Breakout agent employing a Dueling Q-network with prioritized experience replay. Similarly to the prior reward plots, the reward the agent receives gradually increases as training progresses. The highest reward the agent received was 410.8. This is lower than the Dueling Q-network without prioritized experience replay, suggesting sub-optimal hyperparameters. While the Dueling Q-network with PER may not perform better than a standard Double Q-network on Atari Breakout, it would be expected to perform better than the Dueling DQN without PER, given well chosen hyperparameters. In this case, the Dueling DQN with PER was found to perform approximately 8% worse than the

standard Dueling DQN. Hyperparameters worth investigating are: ζ (importance sampling greed) and β (importance sampling bias compensation).

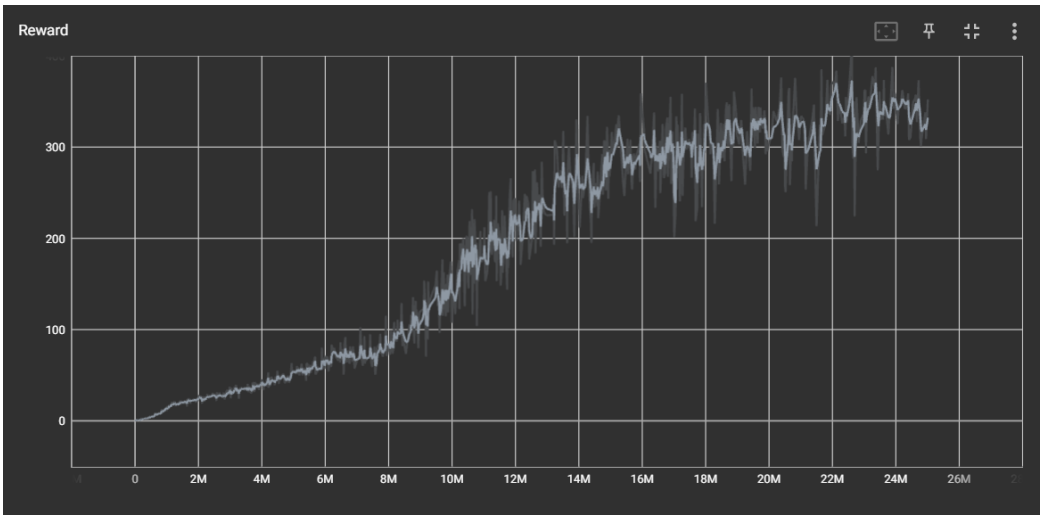


Figure 65: Dueling DQN with PER Reward Plot

4.4.2 Loss

Figure 66 outlines the Huber loss received by the agent throughout training. The graph is in line with what we would expect - a sharp increase at the start of training, followed by a steady decline.

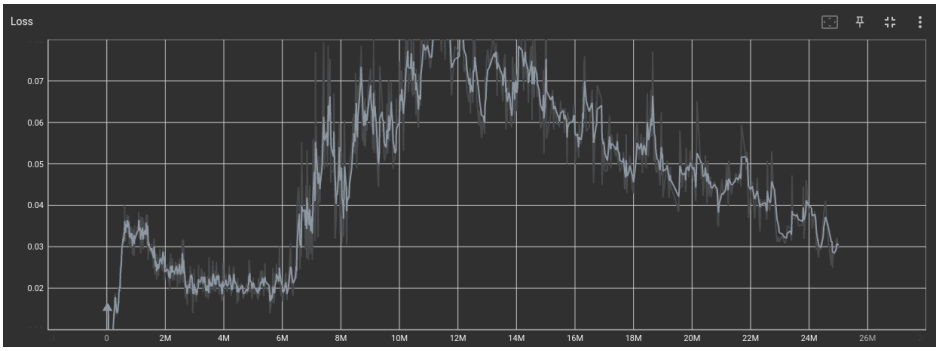


Figure 66: Dueling DQN with PER Loss plot

4.4.3 Kernel Visualization:

Figure 67 and Figure 68 are visualizations of six kernels following game 1 and game 19,000 of training using a Dueling DQN with PER. As would be expected, the kernels underwent change during training. Using the root mean squared error between images, a value of 0.06 was obtained.

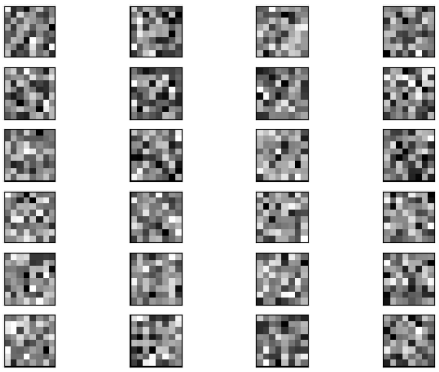


Figure 67: Dueling DQN with PER Game 1 Kernels

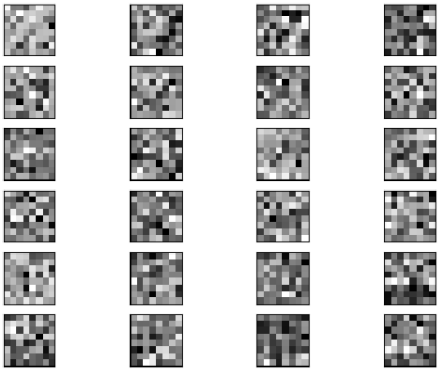


Figure 68: Dueling DQN with PER Game 19,000 Kernels

4.4.4 Video

Figure 69 and Figure 70 show the agent interacting with the environment following 1 game of training and 19,701 games of training. In figure 69, the ball moves from the top left corner to the bottom right corner, while the paddle remains stagnant. Figure 70 shows the agent interacting with the environment trying to maximize the returned reward. For the full videos, see https://drive.google.com/drive/folders/102Ft_h92xmIf6fiMUPfDkjrM1ZbwPhIU

Figure 69: Dueling DQN with PER Game 1 Screenshot

Figure 70: Dueling DQN with PER Game 19,701 Screenshot

4.4.5 Statistical Significance

As was done with the prior DQN variants, the Dueling DQN with PER was trained a second time, and the Mann-Whitney U-test was employed to ensure result validity. Using the Mann-Whitney U-test, a p-value of 0.00 was obtained, indicating a significant difference between the two training instances.

4.5 DeepMellow Deep Q-network

Table 4: DeepMellow DQN Training Summary

Training Summary Data	
Start date:	2023-03-22 14:50:33
Completion Date:	2023-03-23 21:11:10
Duration:	1.263 days
Number of Frames:	25,000,000
Number of Games:	20,400
Highest Reward (Smoothed):	395.3
Highest Reward (Value):	437.5
Final Reward (Smoothed):	283.3
Final Reward (Value):	398.9

The DeepMellow DQN Atari Breakout agent was trained on 25 million frames and achieved a 'Highest Reward (Value)' of 437.5. The DeepMellow Atari Breakout agent implemented as part of this project reached a score of 15 and 20 at timesteps 106.8×10^4 and 140.6×10^4 respectively. The DeepMellow Atari Breakout agent implemented as part of 'DeepMellow: Removing the need for a target network' [13] reached these rewards by timesteps 55×10^4 and 95×10^4 respectively. This discrepancy is likely due to hyperparamter tuning and is a potential area of further research.

4.5.1 Reward

Figure 71 is a graph of the reward received by the reinforcement learning agent per episode of Atari Breakout played, using a DeepMellow Q-network. Similarly to the prior DQN variants, the agent's performance gradually improves over time. However, there are instances when the reward drops sharply before promptly returning to its prior position. These occurrences would indicate network instability. It's worth noting that these occurrences are not present in the early stages of training. It's possible that hyperparameter

tuning may mitigate this instability. The highest reward received by the reinforcement learning agent was 437.5. The reward received by the agent on the last episode of training was 398.9. Using the (smoothed) highest reward, the Dueling DQN underperformed the Double DQN by approximately 6%.

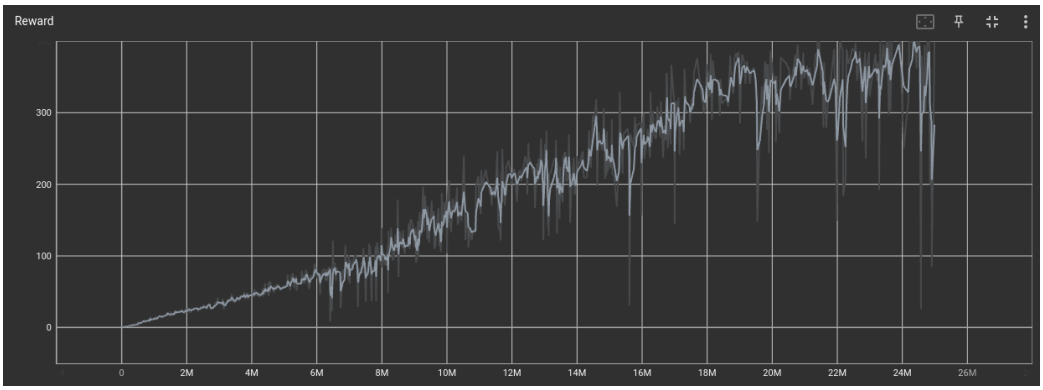


Figure 71: DeepMellow Q-network Reward Plot

4.5.2 Loss

Figure 72 shows the loss plot for the Deep Mellow Q-network. A sharp increase can be observed at the start of training, followed by a steady decline, in line with the other DQN variants.

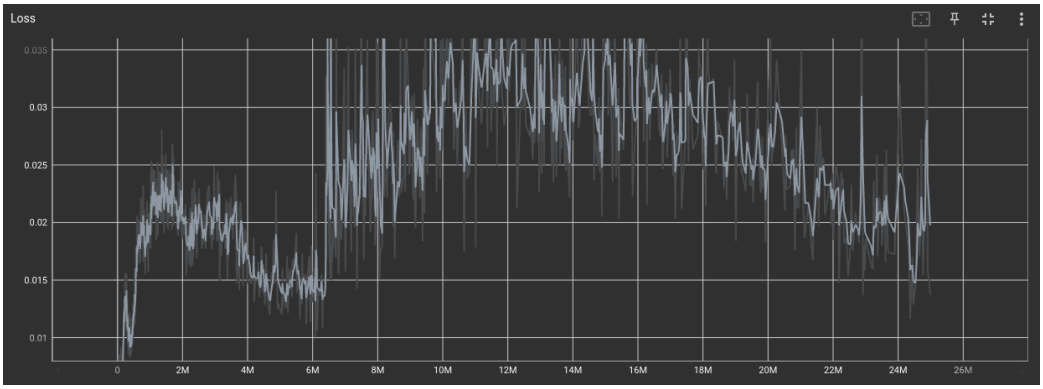


Figure 72: DeepMellow Q-network Loss Plot

4.5.3 Kernel Visualization

Figure 73 and Figure 74 are visualizations of six kernels following game 1 and game 19,000 of training using a DeepMellow Q-network. Similarly to the prior variants, the kernels underwent significant change during training. The root mean squared error between images is 0.11.

Figure 73: DeepMellow DQN Game 1 Kernels

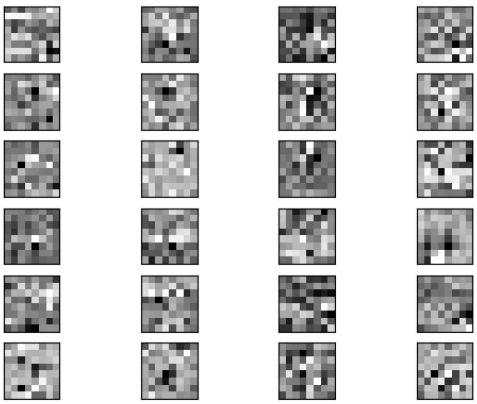


Figure 74: DeepMellow DQN Game 19,000 Kernels

4.5.4 Video

Figure 75 and Figure 76 show the agent interacting with the environment following 1 game of training and 20,475 games of training. In line with the prior variants, the agent in game 1 acts suboptimally. The agent in game 20,475 tries to maximize the amount of reward it can receive. For the full videos, see https://drive.google.com/drive/folders/1Y6Pj1GRzfIKMvp6Nv2bbZ7zyow-joEWb

Figure 75: DeepMellow DQN Game 1 Screenshot

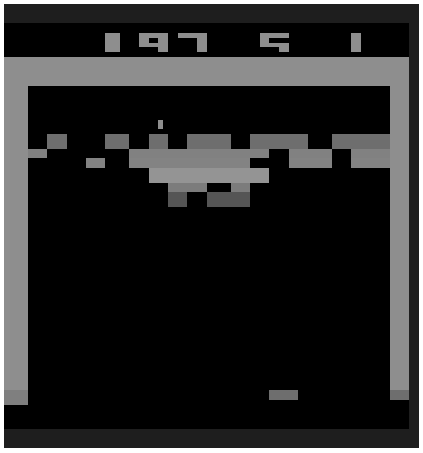


Figure 76: DeepMellow DQN Game 20,475 Screenshot

4.5.5 Statistical Significance

To ensure result validity, the DeepMellow agent was trained a second time and the Mann-Whitney U-test was once again employed. Using the rewards recorded by Tensorboard, a p-value of 0.77 was obtained, indicating no significant difference between the two training instances.

4.6 Result Analysis

4.6.1 Reward Plots

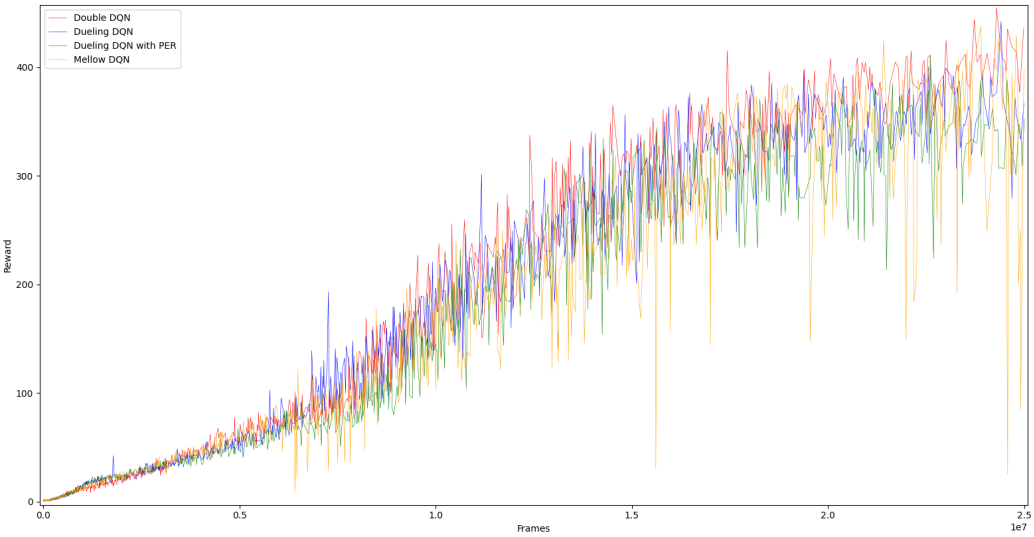


Figure 77: Reward Plots Compared

The Dueling DQN with prioritized experience replay achieved the lowest 'Highest Reward (Smoothed)' with a value of 332.2. The DeepMellow DQN was approximately 19% better with a 'Highest Reward (Smoothed) of 395.3. The Dueling DQN achieved a 'Highest Reward (smoothed)' of 404.6, approximately 2% better than the DeepMellow DQN. The Double DQN achieved the highest 'Highest Reward (smoothed)' with a value of 422.2, approximately 4% better than the Dueling DQN. All of the DQN variants reached respectable rewards. Future research could focus on hyperparameter tuning to see how these variants compare when employing the optimal hyperparameters.

It was observed that the Double DQN was the most stable DQN variant. Followed by the Dueling DQN, Dueling DQN with PER, and DeepMellow DQN, in that order. All DQN variants showed slower learning towards the end of training. The Dueling DQN and Dueling DQN with PER appeared to be the closest to achieving convergence.

4.6.2 Loss Plots

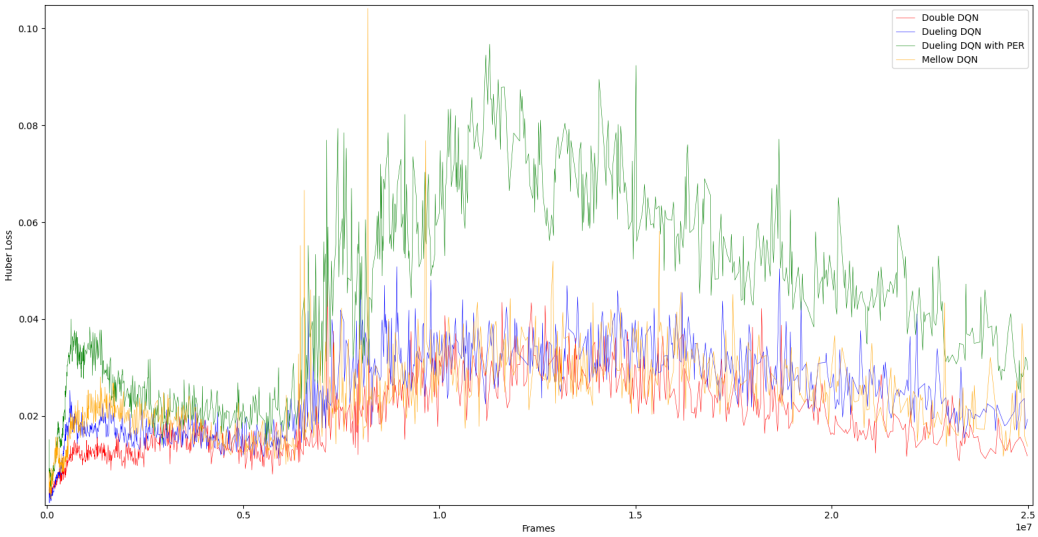


Figure 78: Loss Plots Compared

All DQN variants experienced a sharp increase in loss at the start of training, followed by a steady decrease towards the end. A sharp increase in loss was also observed after approximately 6,000,000 frames of training. This increase is likely due to the agent observing new experiences that produced higher temporal difference errors.

The Dueling DQN with prioritized experience replay showed the sharpest increases in loss. This is what is expected given that it focuses on experiences that have a higher temporal difference error.

4.6.3 Kernels

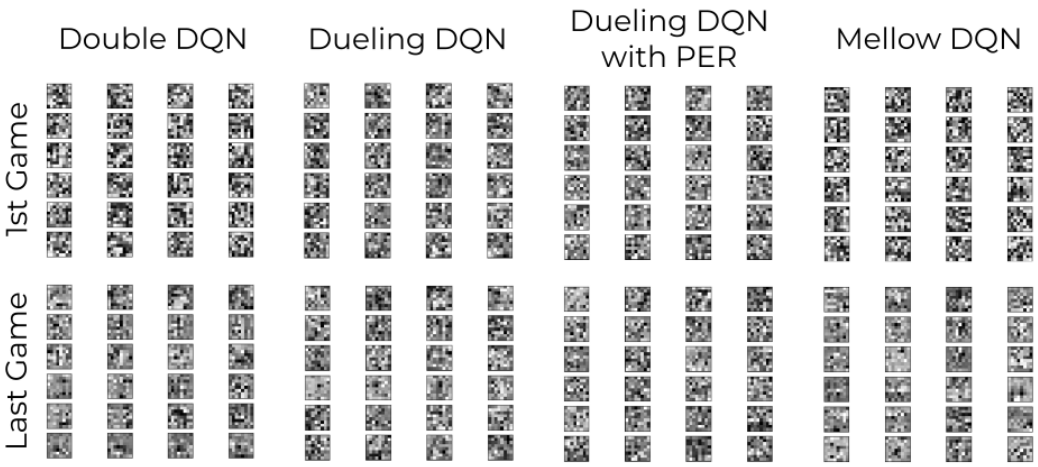


Figure 79: Kernels compared

Figure 79 shows the observed kernels. Given that the Deep Q-networks were initialized with different weights and had different architectures, the 'Last Game' kernels can't expected to be the same or similar. None of the observed kernels appear to have been subject to vanishing or exploding gradients. The root mean squared error between the first and last Double DQN kernel visualization is 0.11. The root mean squared error between the first and last Dueling DQN kernel visualization is 0.06. The root mean squared error between the first and last Dueling DQN with PER kernel visualization is 0.06. Finally, the root mean squared error between the DeepMellow DQN kernel visualizations is 0.11.

4.7 Threats to Validity

While precautions were taken to minimize threats to result validity, no study is completely exempt from potential error or bias. By addressing the potential threats to result validity, future research can build more robust conclusions and the reader can gain insight into the project limitations.

4.7.1 Selection Bias

Suitable hyperparameters were estimated on the basis of existing literature. A rigorous estimation approach, such as gridsearch, was not employed. Future research could compare the DQN variants, employing the optimal hyperparameters.

The Atari Breakout environment was chosen as the test environment due to its widespread use in deep reinforcement learning research, but alternative environments, such as Atari Seaquest, are likely to produce differing results. The Atari Breakout environment is unusual in that it yields lower rewards when the agent employs a Dueling Deep Q-network over a Double Deep Q-network.

4.7.2 Sample Size

Each Atari Breakout agent was trained twice, and the Mann-Whitney U-test was employed to ensure result validity. However, training a greater number of agents may have provided a greater estimation as to the variant performance.

Each agent was trained on 25 million frames, but complete convergence was not observed in any case. It's possible that by training the agents on a larger number of frames, a clearer distinction in variant performance could be observed.

4.7.3 Implementation Bias

The DQN variants were implemented using the Tensorflow Python library. It is unclear as to how the variants would compare if using a different library, such as Pytorch or TF-Agents.

4.8 Conclusion

In conclusion, in this chapter the metrics on which the DQNs were evaluated were restated, and the training results according to said metrics were outlined and analysed.

5 Discussions & Conclusions

In section 1.2, the research question, 'How does the performance of four distinct Atari Breakout agents, each utilizing a unique variant of the Deep Q-network, compare to one another?' was raised.

To begin answering the question, the project began with an approximated systematic literature review of relevant literature. This ensured the prototype development and experimentation tasks were grounded in existing research. The foundations of artificial intelligence, the fundamental principles and approaches of reinforcement learning, and Deep Q-learning were some of the main topics discussed.

Following acquiring in-depth expertise in Deep Q-learning, the high-level architectural decisions, code implementation, and employed equipment were discussed with respect to the prototype development.

This project made two main contributions to the field of Deep Reinforcement Learning. It is the only apparent research explicitly comparing the Double DQN, Dueling DQN, Dueling DQN with prioritized experience replay, and DeepMellow DQN. As the DeepMellow DQN is a novel network architecture, this project also contributes to the very limited research investigating its practical use.

Some of the conclusions that were made in Chapter 4 regarding the research question were that the Dueling DQN with prioritized experience replay was the worst performing variant, achieving a 'Highest Reward (Smoothed)' of 332.2. The DeepMellow DQN was approximately 19% better with a 'Highest Reward (Smoothed) of 395.3. The Dueling DQN achieved a 'Highest Reward (smoothed)' of 404.6, approximately 2% better than the DeepMellow DQN. The Double DQN achieved the highest 'Highest Reward (smoothed)' with a value of 422.2, approximately 4% better than the Dueling DQN. Other topics discussed as part of Chapter 4 were network stability and convergence.

Given that performance is highly dependent on the choice of hyperparameters, future research could focus on hyperparameter tuning, to assess how these variants compare when employing the optimal hyperparameters.

References

[1] Kavosh Asadi and Michael L Littman. An alternative softmax operator for reinforcement learning. In *International Conference on Machine Learning*, pages 243–252. PMLR, 2017.

[2] Andrew Gehret Barto, Richard S Sutton, and CJCH Watkins. *Learning and sequential decision making*. University of Massachusetts Amherst, MA, 1989.

[3] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[4] Richard Bellman. On the theory of dynamic programming. *Proceedings of the national Academy of Sciences*, 38(8):716–719, 1952.

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

[6] Rodney A Brooks. Elephants don't play chess. *Robotics and autonomous systems*, 6(1-2):3–15, 1990.

[7] Melanie Coggan. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[8] Taijara Loiola de Santana, Paulo Anselmo da Mota Silveira Neto, Eduardo Santana de Almeida, and Iftekhar Ahmed. Bug analysis in jupyter notebook projects: An empirical study. *arXiv preprint arXiv:2210.06893*, 2022.

[9] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[12] Alekseĭ Ivakhnenko. Cybernetic predicting devices. Technical report.

[13] Seungchan Kim, Kavosh Asadi, Michael Littman, and George Konidaris. Deepmellow: removing the need for a target network in deep q-learning. In *Proceedings of the Twenty Eighth International Joint Conference on Artificial Intelligence*, 2019.

[14] Chang-Shing Lee, Mei-Hui Wang, Shi-Jim Yen, Ting-Han Wei, I-Chen Wu, Ping-Chiang Chou, Chun-Hsun Chou, Ming-Wan Wang, and Tai-Hsiung Yan. Human vs. computer go: Review and prospect [discussion forum]. *IEEE Computational intelligence magazine*, 11(3):67–72, 2016.

[15] Michael Littman. An alternative softmax operator for reinforcement learning.

[16] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[17] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[18] Robert C Martin. Design principles and design patterns. *Object Mentor*, 1(34):597, 2000.

[19] Fredy Hernán Martínez Sarmiento and Alberto Delgado R. Living beings: source of inspiration for artificial design. *Tecnura*, 17(37):121–137, 2013.

[20] Minsky Marvin and A Papert Seymour. Perceptrons. *Cambridge, MA: MIT Press*, 6:318–362, 1969.

[21] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12, 2006.

[22] James L McClelland, David E Rumelhart, PDP Research Group, et al. *Parallel Distributed Processing, Volume 2: Explorations in the Microstructure of Cognition: Psychological and Biological Models*, volume 2. MIT press, 1987.

[23] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[24] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.

[25] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[26] Marvin Lee Minsky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem*. Princeton University, 1954.

[27] Tom M Mitchell and Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

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

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

[30] Allen Newell. Physical symbol systems. *Cognitive science*, 4(2):135–183, 1980.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[32] Alexandre Piché, Valentin Thomas, Joseph Marino, Gian Maria Marconi, Christopher Pal, and Mohammad Emtiyaz Khan. Beyond target networks: Improving deep q-learning with functional regularization. *arXiv preprint arXiv:2106.02613*, 2021.

[33] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[35] Arthur L Samuel. Machine learning. *The Technology Review*, 62(1):42–45, 1959.

[36] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[37] John Searle. The chinese room. 1999.

[38] David Silver. Lecture 2: Markov decision processes. *UCL. Retrieved from www0. cs. ucl. ac. uk/staff/d. silver/web/Teaching files/MDP. pdf*, 2015.

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

[40] Student. The probable error of a mean. *Biometrika*, 6(1):1–25, 1908.

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

[42] Richard Stuart Sutton. *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst, 1984.

[43] Robert Bruce Thompson and Barbara Fritchman Thompson. *PC hardware in a nutshell: a desktop quick reference*. " O'Reilly Media, Inc.", 2003.

[44] Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 2009.

[45] Hado Van Hasselt et al. Double q-learning. In *NIPS*, volume 23, pages 2613–2621. Citeseer, 2010.

[46] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[47] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[48] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[49] Wikipedia contributors. Intelligence — Wikipedia, the free encyclopedia, 2022. [Online; accessed 18-December-2022].

[50] Frank Wilcoxon. *Individual comparisons by ranking methods*. Springer, 1992.

A Appendix

A.1 PC Build

Preceding December 2022, a 3 year old HP Envy 13 laptop was employed to perform all FYP tasks. The HP Envy has a quad core, 8th generation Intel i7 processor, 8GB of RAM and a Nvidia GeForce MX250, making it sufficient for research, documentation and light experimentation. But given its limited RAM and 384 CUDA core GPU, performing any sort of DQN experimentation on the Atari Breakout testbed was infeasible. Five solutions to facilitate heavier experimentation were considered.

Purchase of an external GPU: An external GPU is a graphics processing unit that resides outside the computer. It connects to a computer via Thunderbolt or USB4 connector. The HP Envy 13 has a thunderbolt 3 port, making this a viable option. The downside to this method is that the eGPUs performance would be bottlenecked by the thunderbolt port. Thunderbolt 3.0 uses 2 to 4 PCIe lanes, whereas a dedicated GPU can use the full 16X bandwidth. Additionally, the CPU may not be capable of processing the data quick enough for the GPU, making the CPU another bottleneck. Given the downsides associated with this option and the fact there would remain only 8GB of RAM, this solution was not chosen.

Repurpose a cryptocurrency mining rig: In November 2022, access to a cryptocurrency mining rig was made available for conducting any heavier experimentation required by the project. The mining rig consisted of a quad core Intel i5-3570 CPU, 16GB of DDR3 RAM and a Nvidia GTX 1080ti with 11GB of VRAM. These components would be sufficient for any experimentation required but this option was ultimately deemed undesirable given that the project success would be dependent on the availability of external PC components.

University of Limerick GPU farm: Another option was to acquire permission to use the University of Limericks GPU Farm. This method would ensure ample compute power for performing all the required experiments. Additionally, due to the number and sophistication of the

GPUs, experimentation would be completed significantly faster than other methods. Similarly to the prior option though, the success of the project would hinge on the availability of the GPUs. Furthermore, any experimentation required outside college hours or following this project would remain limited.

Build/Purchase a suitable PC: Given that none of the above methods were suitable, it was concluded that the best option was to either purchase a prebuilt PC with suitable specification or to build one. For the reasons addressed below, it was decided that the construction of a customized personal computer was the best solution.

1. Opportunity to learn: Prior to assembling the customized PC, there was a gap in my knowledge of computer systems, specifically the hardware components that comprise a computer. This project provided the opportunity to perform my own PC Build and learn more about these components in a practical manner.

2. More Flexibility: Having full control over PC component selection, allowed for the allocation of more of the budget to components that would contribute more to the project requirements.

3. Less Expensive: Custom-built PCs often contain superior quality components [43], making them less likely to experience component failure and degradation. This can result in decreased maintenance costs and long-term financial benefit. Additionally, it is anticipated that there would be fewer instances requiring outsourced part repair or replacement, in light of an increase in understanding of computer hardware.

The main PC components are outlined below. For the full specification see, https://pcpartpicker.com/list/gFLwLs

CPU: Intel Core i5-12600K 3.7 GHz 10-Core Processor. The chosen CPU was a 10 core 12th generation Intel core i5 CPU. The CPU isn't as im-

portant as the GPU in terms of machine learning tasks, but ensuring a sufficient number of cores for programming tasks outside this project played a role in CPU choice. The decision was influenced by Intel CPUs being slightly cheaper than AMD CPUs at the time of purchase.

GPU: MSI GeForce RTX 3060. The chosen GPU was a Nvidia RTX 3060. While the RTX 3060ti and 3070 would provide greater performance, they only have 8GB of VRAM. Given that a memory related error would be more significant than slightly longer training times, the RTX 3060 was chosen.

RAM: Corsair Vengeance 32 GB DDR5-5600 Memory. Given that sufficient and fast RAM is important for machine learning tasks, it was decided that the slightly more expensive DDR5 RAM be chosen over DDR4 RAM.

Motherboard: MSI MAG Z790 TOMAHAWK. Compatibility was of significant importance when choosing a motherboard. A motherboard that would support the CPU as well as the DDR5 RAM was required. Additionally, in an effort to future proof the build, there are two free RAM slots and a number of free PCIe slots, in case a future machine learning task ever requires more RAM or an additional GPU. An 850 Watt power supply unit was purchased so such additions would not require a different power supply unit.

CPU Cooler and Fans: Be Quiet! Dark Rock Pro 4. Given that it takes a minimum of 30 hours for each DQN experiment, ensuring the CPU, GPU and other system components are kept cool is crucial. For this reason, there are three 120mm intake fans and two 120mm exhaust fans. Additionally, there is a Be Quiet Dark Rock Pro CPU Cooler.

Figure 80: Completed PC Build