Computer Science with AI

COC257

F018182

ARTIFICIAL INTELLIGENCE FOR PLAYING THE ATARI BOXING GAME

By

Marcos Valadares

Supervisor: Shirin Dora

Department of Computer Science

Loughborough University

May/June 2024

Abstract

This project aims to create an AI for the Atari 2600 game “Boxing”. This is a 2-player game where you fight an opponent in a boxing ring and score points for hitting them. You win by knocking your opponent out with 100 points. To train an AI opponent beyond the level of a human player, I will an agent competing against the computer-controlled opponent. It will adapt its strategy through thousands of games using reinforcement learning until the opponent is beaten even at its highest difficulty.

Keywords: AI, Reinforcement Learning, Q-Learning, Deep Q-Network, Atari 2600, Boxing.

Acknowledgements

I would like to take this opportunity to thank my supervisor Dr Shirin Dora for his advice towards my goals and approaches in this project. His introduction of OpenAI Gymnasium was what adequately started my journey of learning the implementation of reinforcement learning methods. He also provided me with valuable guidance on the writing of this report. Secondly, I want to acknowledge Dr Shaheen Fatima’s teachings in Agent-based Systems as the module served as a notable foundation for me to build my knowledge upon.

Contents

Introduction

Motivation

Training autonomous agents to navigate and master dynamic environments is a challenge that grasped by interest even before I began learning about the basics of machine learning. This project is motivated by the continuous exploration of reinforcement learning techniques applied to other video games, including classic Atari 2600 titles like “Pong” and “Space Invaders”. I chose to train "Boxing" as it offers a dynamic testing ground for an agent to learn and adapt to different strategies with precise control. The simplicity of the environment, coupled with its inherent complexities, makes it perfect for assessing the capabilities of reinforcement learning models. Beyond gaming, the practical implications of AI training methodologies extend to real-world applications, ranging from robotics to adaptive systems. I hope to gain valuable experience from this project that I can transfer to these fields in the future.

Learning Objectives

1. Exploring a variety of reinforcement learning concepts and methodologies through the means of research and practical experience.
2. Gaining a deep understanding of some reinforcement learning algorithms function, including their architecture, training procedure and key mathematical components.
3. Develop practical skills in implementing deep learning models using the PyTorch framework. This includes building neural networks, defining loss functions and optimising hyper-parameters.
4. Gain hands-on experience in training agents in complex environments provided by the OpenAI Gymnasium toolkit. Understanding the challenges of tuning a model for its optimal training procedure.
5. Learning techniques for the evaluation of agent performance, including analysing learning curves, assessing convergence and interpreting experiments.
6. Practice formal documentation and communication skills by writing about my findings and experiments effectively throughout this report.

1. Literature Review

1.1. Reinforcement Learning

The objective of Reinforcement Learning (RL) is to maximise the reward of an agent by taking a series of autonomous actions in response to some environment. Depending on the system’s design objectives, it might also aim to minimise a penalty. The state of an environment is a current representation of said environment, providing information necessary for the agent to base its actions on. The algorithm by which an agent makes decisions is the policy, which maps perceived states to their ideal actions.

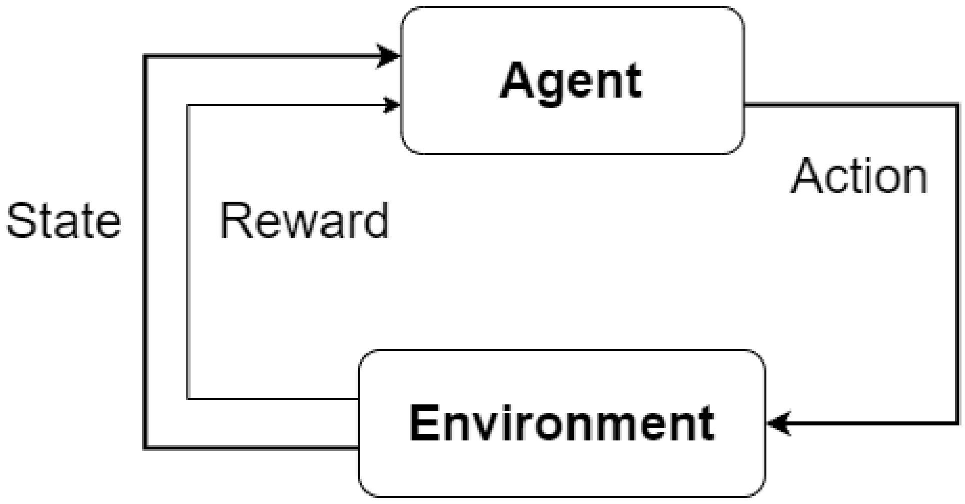


Figure 1: Reinforcement Learning components.

1.2. OpenAI Gymnasium

1.2.1 Atari 2600 Boxing

The Atari 2600 was one of the first commercially available video game consoles. Many of its classic titles have become a popular benchmark for testing and evaluating reinforcement learning algorithms. These arcade-like environments provide a challenge to agents due to their high-dimensional observation space, consisting of 210x160 pixel RGB images. The action space being a subset of 18 different actions like “LEFT” or “FIRE”, all of which are available without choice in Boxing.

Boxing is a game developed by Activision where two boxers face off in a ring until they score 100 points or the 2-minute round ends. It can be played against the computer or another player, both able to select expert or novice difficulty. You are rewarded 1 point for up-close punches and 2 points for long jabs. This opens the possibility for strategies varying from defensive to offensive playstyles. This is what the game’s environment looks like:

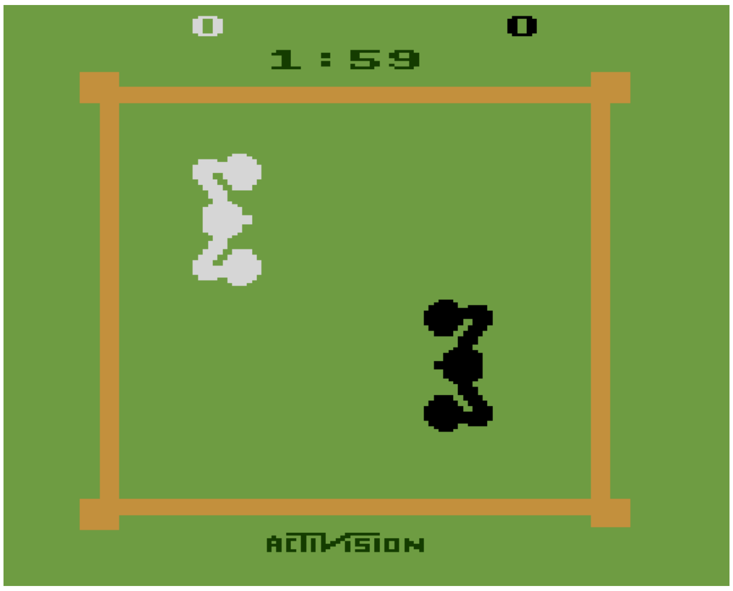


Figure 2: Atari 2600 Boxing frame.

1.2.2 Arcade Learning Environment

It was pointed out that Atari games are entirely deterministic (Bellemare, 2013). Hence, agents could achieve state of the art performance by just memorising an optimal sequence of actions and ignoring the environment. The Arcade Learning Environment (ALE) framework dealt with this through the concept of “sticky actions”. Instead of repeatedly simulating the action passed by the agent there is a small probability that the previously executed action is used once more (OpenAI, Gym Documentation, 2022). Additionally, OpenAI’s Gym python toolkit offers a frame-skipping approach (further explored in the Design and System Overview section) that alongside sticky actions results in a much more stochastic environment. Boxing was found to be one of the environments that benefits the most from a high level of stochasticity (Kaiser, 2020).

1.3. Q-Learning

1.3.1 Learning Fundamentals

One of the most key branching points in reinforcement learning algorithms is whether an agent has access to a model of the environment (model-based), or it is instead learning it through interactions with that environment (model-free). The “model” here being the function responsible for predicting state transition and reward. Q-Learning is a form of model-free RL initially proposed by Watkins in 1989, which serves as a starting point for understanding how agents can navigate intricate game dynamics and strategic decision-making. It updates an action-value function based on the Bellman equation as follows:

$$\underbrace{\text{New } Q(s,a)}_{\substack{\text{New}\\ \text{Q-Value}}} = \underbrace{Q(s,a)}_{\substack{\text{Current}\\ \text{Q-Value}}} + \underset{\substack{\text{Learning}\\ \text{rate}}}{\alpha} \Big[\underbrace{R(s,a)}_{\text{Reward}} + \underset{\substack{\text{Discount}\\ \text{rate}}}{\gamma} \overbrace{\max Q'(s',a')}^{\substack{\text{Maximum predicted reward, given}\\ \text{new state and all possible actions}}} - Q(s,a) \Big]$$

Figure 3: Bellman equation (Q-Learning)

The task is to determine an optimal policy that maximises the total discounted expected reward (Watkins, 1992). The Q-value represents the “quality” or usefulness of a given action a in state s to attain this future reward. A data structure called Q-Table guides the agent’s decision by storing the Q-values for all actions in each state, while the previous equation is used for updating these values throughout the training process. Q-learning typically employs a learning rate parameter α to control the rate at which Q-values get updated. The discount factor γ determines the importance of future rewards relative to immediate rewards. A higher discount factor encourages the agent to plan its strategy around long-term consequences.

The steps involved in this algorithm are the following:

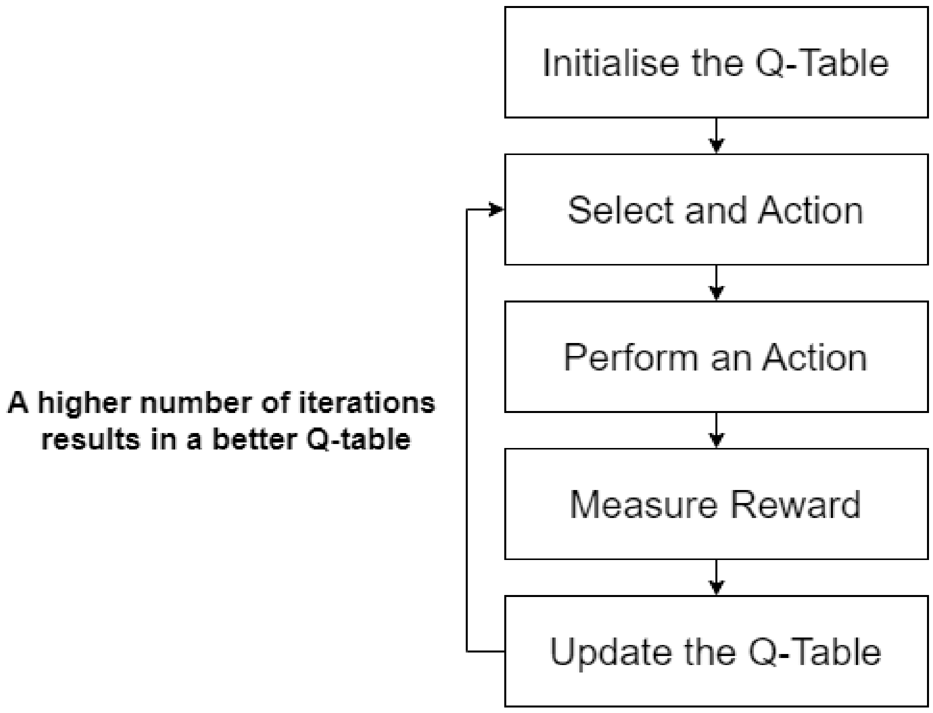


Figure 4: Q-Learning steps.

The Q-Table is often initialised with all 0 values, which raises the issue of how the agent can make a choice at this stage. This is solved with the concept of ϵ-greedy exploration and exploitation. In the beginning, the agent selects a random action with probability ϵ (epsilon parameter), which is referred as the exploration stage of training. This probability starts off as 1, slowly dropping down closely to 0 at a rate depending on the environment and goals of the model. As $1 - \epsilon$ increases the agent will more often be "exploiting" actions based on the best Q-value available in that state. This value ideally stabilises somewhere between 0.1 and 0.01, though 0.05 is considered standard for training Atari environments (Silver, et al., 2013).

$$\text{Action at time(t)} \begin{cases} \max Q_t(a) & \text{with probability } 1 - \epsilon \\ \text{any action (a)} & \text{with probability } \epsilon \end{cases}$$

Figure 5: Epsilon-greedy policy.

1.3.2 Limitations

Initially, Q-learning was primarily used for games with discrete state and action spaces, making it applicable to a range of Atari 2600 games. However, before the famous DeepMind paper (Mnih, Playing Atari with Deep Reinforcement Learning, 2013) using the Deep Q-Network (DQN) approach, Q-learning in its basic form faced challenges when applied to high-dimensional and complex environments like Atari games due to large state and action spaces.

One notable example of Q-learning applied to Boxing is the work by Bellemare in 2013 where they explored the combination of Q-learning with feature extraction techniques to reduce the dimensionality of the state space. Despite some success in learning to play simpler games, it struggled to achieve human-level performance in more complex games like Breakout. Having only beaten Boxing’s average human score of 12 points in 1 out of the 4 attempted variations (the RAM method) that was working on a different observation space than the others. It directly observed the Atari 2600’s 1024 bits of memory instead of its screen, which allowed for more efficient Q-learning.

In the following sections, I explore the evolution of RL algorithms beyond traditional Q-learning, examining how more advanced techniques, address the limitations posed by high-dimensional state spaces and dynamic game environments.

1.4. Deep Q-Networks

1.4.1 The Original DQN Framework

The DQN was first proposed in the groundbreaking paper "Playing Atari with Deep Reinforcement Learning" (Mnih, 2013) and it revolutionised the field of training game-playing agents. By combining Q-learning principles with deep neural networks, DQN tackled the challenge of efficiently navigating complex state spaces. Instead of a Q-table with poor scalability, a Q-function maps a state to the Q-values of all the actions available from that state. Learning the network’s weight parameters to achieve optimal Q-values. Similarly to Q-learning, the algorithm begins by exploring the environment and slowly exploiting it using the ϵ-greedy policy. Keeping the same notion of dual actions, where the current action has a current Q-value and the target action a target Q-value.

This architecture has two identical neural networks, the main Q-network and the target Q-network. This is commonly a convolutional neural network (CNN) designed to process raw pixel inputs from the game’s frames, allowing the agent to decide based on visual information. This is a significant advancement over traditional Q-learning as it makes the algorithm more applicable to Atari’s wide range of environments. The algorithm uses an experience replay buffer, which gathers training data by interacting with the environment. This includes the current state s_i, action a, reward r and next state s_{i+1}. This is how the predicted and target Q-values per iteration i are calculated:

$$y_i^{\text{main}} = Q(s_i, a|\theta)$$

$$y_i^{\text{target}} = r_i + \gamma \max_{a'} \hat{Q}\,(s_{i+1}, a'|\theta^-)$$

where θ and θ^- denote the hyper-parameters of the main Q-network and target Q-network respectively. When updating the neural network weights, it also uses an optimiser that minimises the mean squared error loss between Q-values and target Q-values. This is what the loss function looks like:

$$L(\theta) = E\left[\left(y_i^{\text{target}} - y_i^{\text{main}}\right)^2\right]$$

A successful optimiser in Deep Q-Learning is Adam (Kingma, 2015). This optimiser improved on the ideas of its predecessor RMSprop by combining them with adaptive learning rates based on moving averages and bias correction. These techniques result in faster convergence of the model and less sensitivity to hyper-parameter values (Saripalli, 2019).

1.4.2 Double DQN

Since the introduction of DQN, numerous extensions and improvements have been proposed to enhance its performance and address its limitations. A noble example is the Double DQN (DDQN) algorithm, introduced also by Google DeepMind (Hasselt, 2015). By separating the selection and evaluation of actions, DDQN effectively addresses the overoptimistic value estimates found in the original DQN, resulting in more precise Q-value calculations. DDQN finds the optimal action as such:

$$a^{max}(s_{\text{i+1}}|\theta) = \text{argmax}_{a'} Q(s_{\text{i+1}}, a'|\theta)$$

and now calculating the target Q-value with this optimal action:

$$y_i^{\text{target}} = r_i + \gamma \hat{Q}(s_{i+1}, a^{max}(s_{i+1}|\theta)|\theta^-)$$

Therefore, the target Q-network becomes responsible for evaluating the main Q-network’s selected actions. This method has shown significant improvement in training convergence speed and overall model reliability in comparison to nature DQN (Zhang, 2024).

1.4.3 Dueling DQN

Google DeepMind released a re-envisioned DQN architecture called Dueling DQN (Wang, 2016). Rather than estimating the Q-value for each action, this method separates the estimation of the state-value $V(s)$ and the advantage-value $A(s,a)$, where $V(s)$ is the value of a state regardless of action taken and $A(s,a)$ is the advantage of taking an action in a certain state. The agent then learns the value of actions independently of the state.

The network branches off into two streams to estimate these values and them combining them to obtain the final Q-value function, which adds advantage-value to state-value and subtracts the mean advantage-value across all actions:

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

where $|A|$ represents the action space of the environment (18 in the context of Boxing). The following figure is based on Wang’s paper and it shows how the discussed values are presented in the neural network layering of the Dueling DQN architecture:

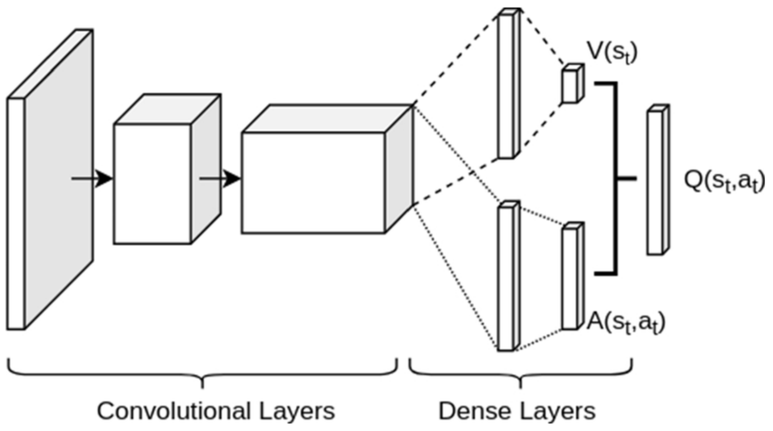


Figure 6:DQN and Dueling DQN layering (le, 2021)

1.4.4 Application to Atari 2600

In the context of training an agent for the game Atari Boxing, the DQN framework is a great starting point to developing a competent agent. By leveraging the principles of value-based reinforcement learning and the capability of deep neural networks to process visual information. DQN along with its Double and Dueling variations have the clear potential to effectively learn human-level strategies in the Arcade Learning Environment.

1.5. Policy Gradient Methods

In the following section I will explore other potential reinforcement learning approaches to have a better understanding of their potential in comparison to methodologies derived from Q-learning.

1.5.1 The Original Policy Gradient Framework

Policy gradient methods aim to directly learn the optimal policy that maps states to actions, without relying on an explicit value function. This paradigm shift allows for a more natural representation of the agent's decision-making strategy. One of the seminal papers in this domain is "Policy Gradient Methods for Reinforcement Learning with Function Approximation" (Sutton, 2000). This paper lays the groundwork for understanding the principles of policy gradients and their application in RL.

1.5.2 Actor-Critic Framework

An influential architecture within the realm of policy gradient methods is the Actor-Critic framework. Combining elements of both policy-based and value-based approaches, Actor-Critic architectures leverage a policy network (the actor) to make decisions and a value network (the critic) to estimate the expected return. "Actor-Critic Algorithms" by Konda and Tsitsiklis (2000) [7] provided me with a comprehensive overview of this framework.

Its advantage being they can reduce the variance of the policy gradient by using the critic’s value function as a baseline, meaning less samples are needed to converge. They can also incorporate temporal difference learning, allowing to bootstrap from the critic’s estimates and update the policy before the end of an episode. Its limitation is requiring two neural networks to be trained and updated simultaneously. The actor and critic may interfere with each other’s learning and increase computational cost of the algorithm.

1.6. Transfer Learning

Transfer learning (TL) offers an alternative to reduce training times and the number of necessary instances to learn a model from scratch. It reuses previously obtained knowledge from one or more source tasks to learn a new target task. In the context of deep reinforcement learning, TL can be used to transfer knowledge from one agent to another, allowing it to leverage the experience gained from previous tasks to improve learning efficiency and performance. This is done successfully in a recent paper where numerous Atari 2600 games are trained with each other's algorithms (Ramirez, 2021). The correlation matrix below demonstrates the effectiveness of the proposed source task selection methodology in improving the performance of DRL algorithms in Atari games, outperforming other traditional DRL algorithms in most cases. The x-axis represents the different games, while the y-axis represents the average score obtained by each algorithm.

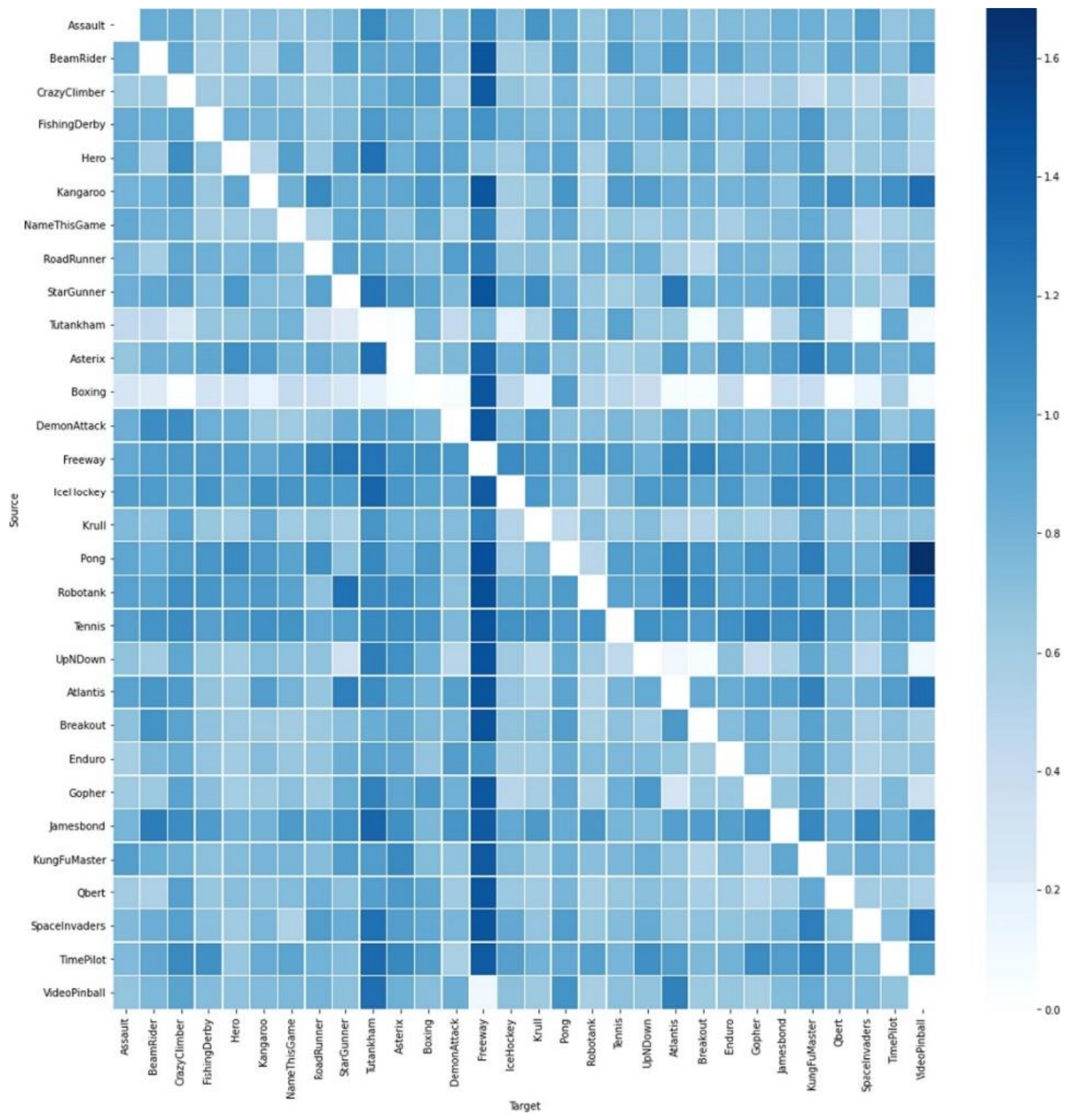


Figure 7: Transfer learning in Atari games (Ramirez, 2021).

By observing the row representative of our game Boxing, we can conclude that most games struggled to learn and achieve high scores from its pre-trained model compared to all others. It is also fair to say that for the most part, this agent would not benefit substantially from transfer learning given that its column showcases mostly negative transfer (under 1.0).

2. Design and System Overview

2.1. Methods of interest

After thorough research of different reinforcement learning methodologies, I came to the decision of focusing on the implementation and analysis of the Deep Q-network architecture and its covered variations to train an agent in the Atari Boxing environment. Namely, this will cover Nature DQN, Double DQN (DDQN) and the combination of both Double and Dueling DQN techniques (D3QN). Additionally, selecting Atari 2600 Breakout as a second environment to conduct experiments in at a faster rate and test the agent’s adaptability to other games.

2.2. Experiment Goals

The objectives of the project’s implementation are as follows:

- Achieve human-Level performance with each of the DQN architectures. For the Atari Breakout environment this would be a score of approximately 30.5, whereas for the Boxing environment a score of 12.1. (Schwarzer, 2023)
- Obtain comparable visual data between the 3 variations and the 2 environments. This will allow for a better analysis of the algorithm’s learning dynamics.
- Testing out how modifications to the agent’s hyper-parameters can impact its performance and training speed. Specifically, changes in learning rate, batch size, Gama and the ϵ-greedy’s values (minimum and decay).
- Experimenting with the effectiveness of transfer learning techniques between Atari Breakout and Boxing.

2.3. Project Design

2.3.1 DQN Architecture

The agent’s architecture as a whole and the way it interacts with its environment have been designed as follows:

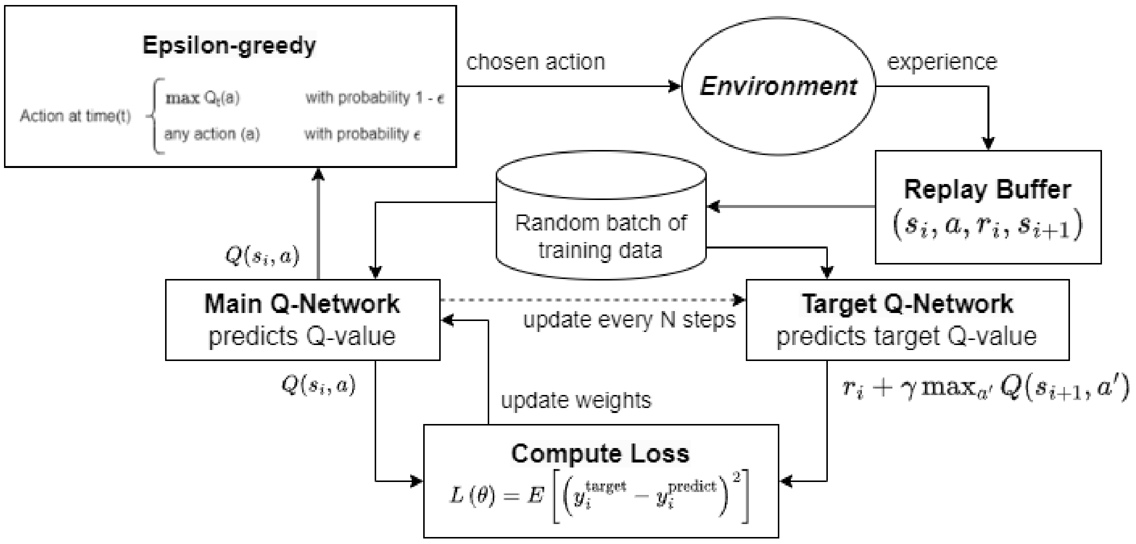

Epsilon-greedy
Action at time(t)
max $Q_t(a)$ with probability $1-\epsilon$
any action (a) with probability ϵ
chosen action
Environment
experience
Replay Buffer
(s_i, a, r_i, s_{i+1})
Random batch of training data
$Q(s_i, a)$
Main Q-Network predicts Q-value
update every N steps
Target Q-Network predicts target Q-value
$Q(s_i, a)$
update weights
$r_i + \gamma \max_{a'} Q(s_{i+1}, a')$
Compute Loss
$L(\theta) = E\left[\left(y_i^{\text{target}} - y_i^{\text{predict}}\right)^2\right]$

Figure 8: Agent's full architecture.

Depending on the current ϵ value, a step starts with either a random or calculated action chosen by the agent. The gym environment responds to this action and the experience replay buffer collects the next step’s sample of current state, action, reward and next state. The optimisation process begins as a random batch of training data is fed by the replay buffer to both the Q-networks. They make their predictions for the Q-values and then these are used to calculate the loss between them. Backpropagating the loss through the main Q-network and updating the weights using gradient descent (Silver, et al., 2013). The target Q-network’s weights remain unchanged to support the other network’s predictions by stabilising its target. It is only after N steps that the target Q-network receives a soft update to its weights by the main network, enabling it to remain accurate. This concludes each step of the iteration, so now the main Q-network is ready to send its new Q-value to the ϵ-greedy policy for a new action to be taken.

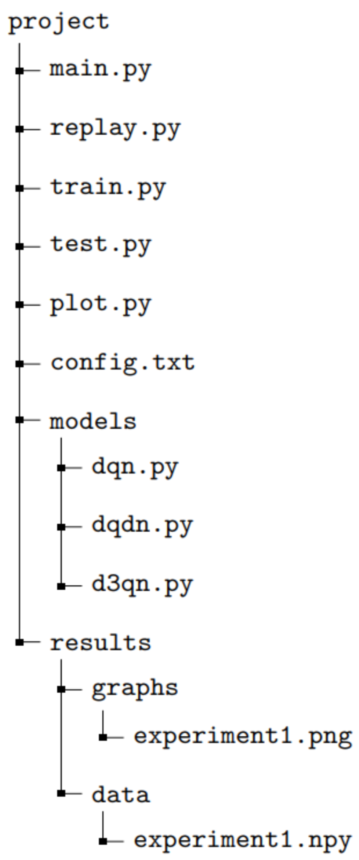
2.3.2 Libraries

At this point it is hard to predict what additional libraries might be required, however the following are sure to be requisites in this system:

- PyTorch: This is the primary deep learning framework for implementing neural network architectures used in DQN. It is extensively documented and easy to use.
- NumPy: It is an essential library for numerical computations in Python. It will be used for computing Q-values during the training process, loading large data files and other mathematical functions.
- Matplotlib: A powerful plotting tool that will help with visualising agent learning curves and comparing these between models.
- Gym[Atari]: As explored detailedly in the Literature Review section, OpenAI Gym provides a set of Atari 2600 environments necessary for AI to learn from.
- TQDM: Adds visual feedback to iterative processes such as training loops.

2.3.3 File Structure

The primary files of the agent training system will look as follows:



```
project
├─ main.py
├─ replay.py
├─ train.py
├─ test.py
├─ plot.py
├─ config.txt
├─ models
│  ├─ dqn.py
│  ├─ dqdn.py
│  └─ d3qn.py
└─ results
   ├─ graphs
   │  └─ experiment1.png
   └─ data
      └─ experiment1.npy
```

Figure 9: Project file structure.

The agent is initiated in main.py and it will either be running the training loop from train.py or the testing loop from testing.py. Either loop is utilising one of the three models, updating Q-values, computing loss and utilising the replay buffer in replay.py. Eventually, the model is stored in the data folder and plot.py will allow the generation of a graph from the stored data. The hyper-parameters involved are kept in a separate text-based format file like config.txt for easy reachability.

3. Implementation

3.1. DQN Building Blocks

3.1.1 Pre-processing

The preprocessing techniques implemented are commonly used in reinforcement learning to more efficiently extract information from the game environment frames, speeding up the training process and reducing memory consumption (Mnih, Human-level control through deep reinforcement learning, 2015). The process looks as follows:

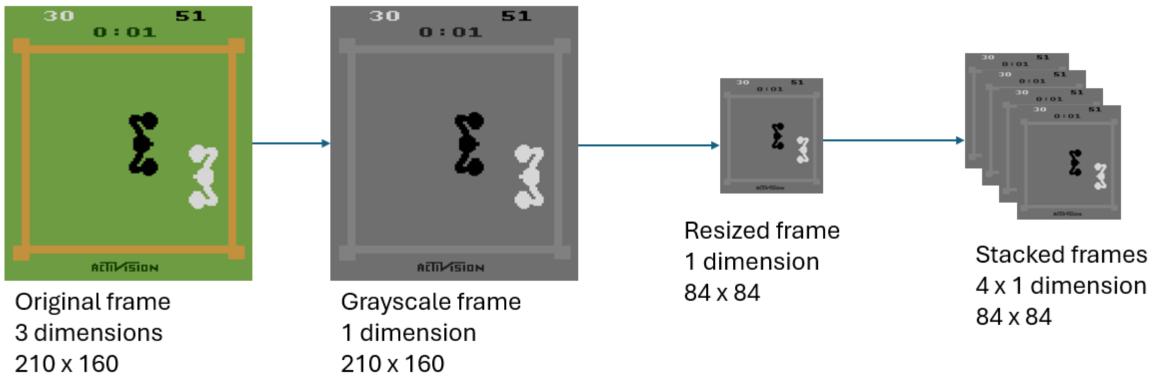


Figure 10: Pre-processing.

A frame skip technique was employed to accelerate training. The agent only takes actions and receives rewards every fourth frame, repeating the intermediate frames. In the context of temporal difference learning, frame-skipping controls bootstrapping and reduces the agent's horizon (Aranvidan, 2021). The collected game frame starts off as a colour image with Atari 2600's screen dimensions. It is then decoloured with a grayscale filter to reduce the frame's 3 input channels to 1 value. Next it is resized to a size of 84 by 84 pixels, which is recognised as a sweet spot for Atari environment frames to still be effectively analysed by the agent while not taking a substantial amount of memory. These image conversion methods were originally done with the library OpenCV, but later replaced with OpenAI Gym's Atari preprocessing wrapper for abstraction reasons. Frame stacking is then utilised to turn 4 frames into a single observation. This helps the agent to better perceive motion of objects in the environment, thus enabling it to make more informed decisions. Lastly, in each step the reward attained is clipped within a range of -1 to 1. This is done by the OpenAI Gym wrapper "ClipReward" and it prevents the difference the agent from overly preferring certain actions due to an increased reward that may not directly improve long term goals (OpenAI, Gym Documentation, 2022). For Atari Boxing in particular, this is beneficial as the agent would play too safely to avoid being hit with a close punch (-2 points) or too recklessly by always punching up close to the opponent (+2). Consistency prevented slow learning curves.

3.1.2 DQN Layering

With the necessary pre-processing done to the game environment’s frames, we are now able to feed a manageable state input to the neural network. The network structure for both nature DQN and Double DQN will look as follows:

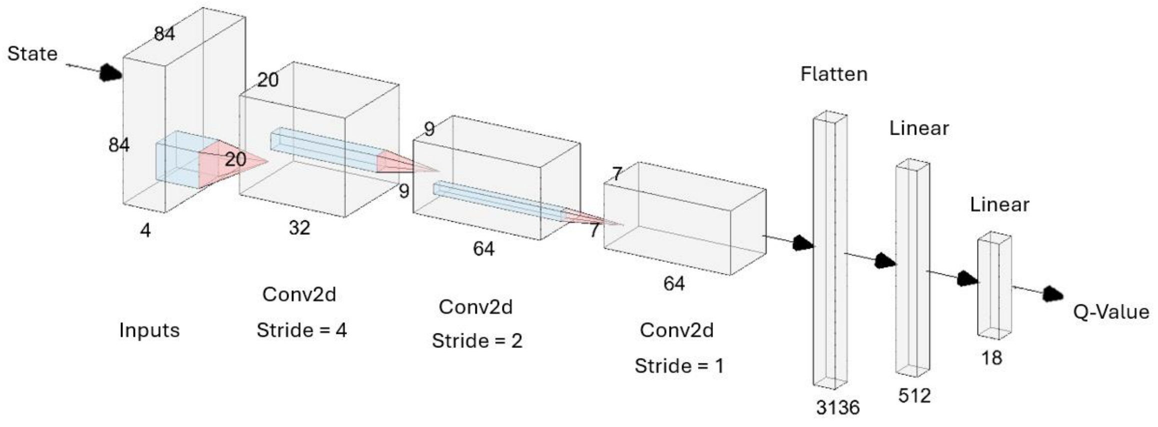


Figure 11: DQN layering structure.

The input layer takes in the 4 stacked images and passes them on as pixel values to the network. The first convolutional layer then performs feature extraction by applying a set of learnable filters (kernels) to the data. These kernels detect slight differences and patterns within the image data. Convolutional layers here are followed by a Rectified Linear Unit (ReLU) that is commonly used as an activation function in DQN algorithms. These introduce non-linearity to the network, allowing models to learn faster by overcoming the “vanishing gradient problem” (Agarap, 2018). The data continues to be passed on through the convolutional layers, increasing dimension size to 64 so that it can better capture spatial features. Their output is then flattened into a one-dimensional vector that linear layers can process. It is important to know the correct flattened size going into the first linear layer as it can affect the how the model runs. This was calculated by multiplying the dimension of the last layer with the outputted activation map’s height and width values, which according to PyTorch’s Conv2d documentation are worked out like this:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

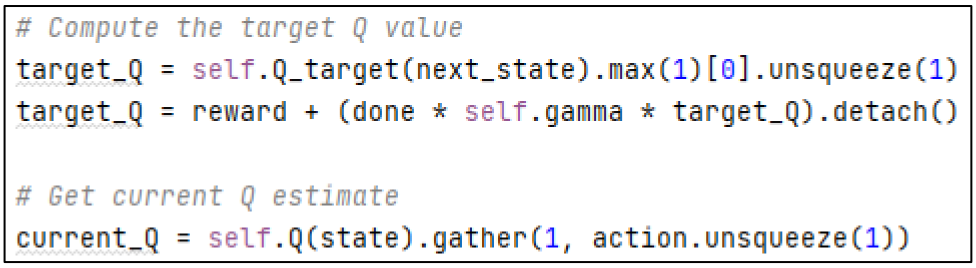
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Figure 12: Conv2d output size formula (PyTorch, 2023)

where in this specific architecture it resulted in a size of 3136 as shown in the diagram. Next, the fully connected layers (linear) process the flattened feature vectors and map them to Q-values for each available action in Boxing.

3.1.3 Double DQN Improvements

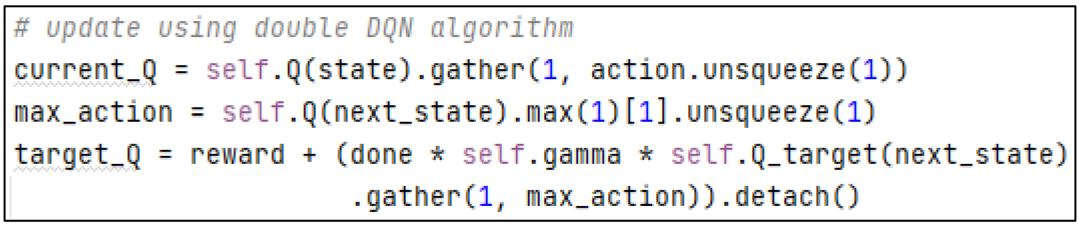
As explored in the Literature Review section, Double DQN is a significant improvement to the way nature DQN estimates its Q-values. It decouples the selection and evaluation of actions by using two separate neural networks. The code behind the original DQN’s computing of Q-values is the following:



```
# Compute the target Q value
target_Q = self.Q_target(next_state).max(1)[0].unsqueeze(1)
target_Q = reward + (done * self.gamma * target_Q).detach()

# Get current Q estimate
current_Q = self.Q(state).gather(1, action.unsqueeze(1))
```

and to create a DDQN variant of the original implementation, the math involved in acquiring the optimal action given the state and giving the target Q-function the task to evaluate actions is this:



```
# update using double DQN algorithm
current_Q = self.Q(state).gather(1, action.unsqueeze(1))
max_action = self.Q(next_state).max(1)[1].unsqueeze(1)
target_Q = reward + (done * self.gamma * self.Q_target(next_state)
                    .gather(1, max_action)).detach()
```

This change directly impacted agent performance by mitigating the overestimation bias.

3.1.4 Dueling DQN Improvements

The D3QN variant builds upon DDQN’s by combining both Double and Dueling techniques. This means the previous change remains and that algorithm’s neural network structure will be altered. The two separate streams in the network will look as follows:

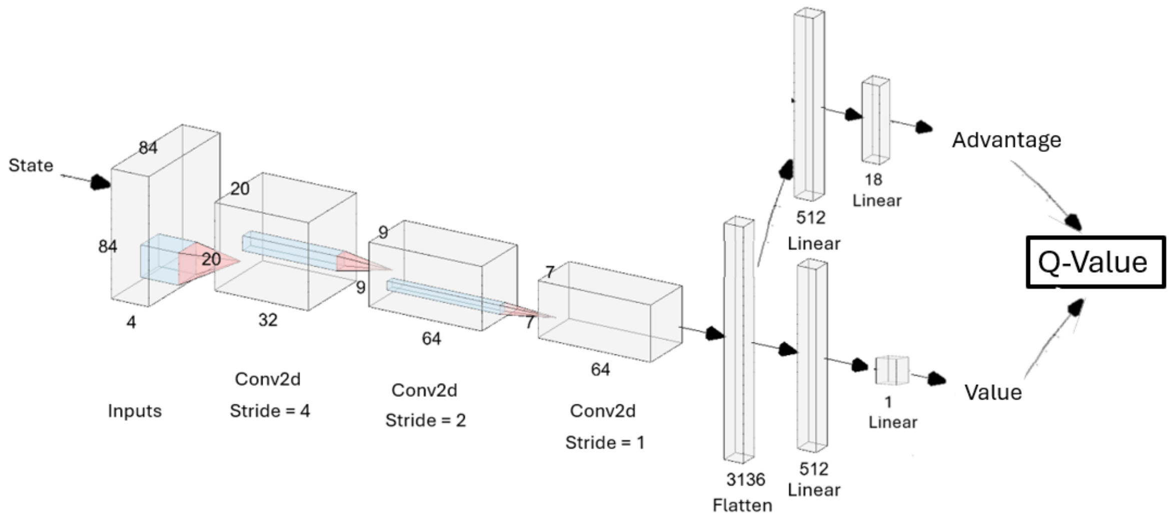
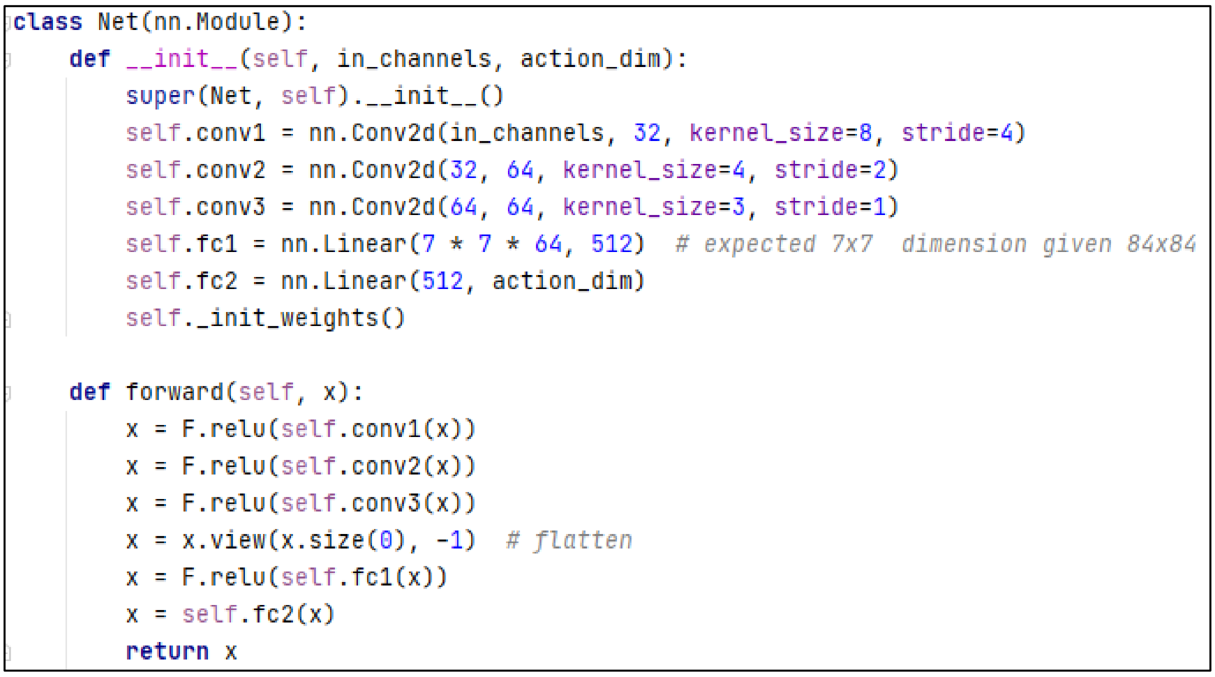


Figure 13: Dueling DQN layering structure.

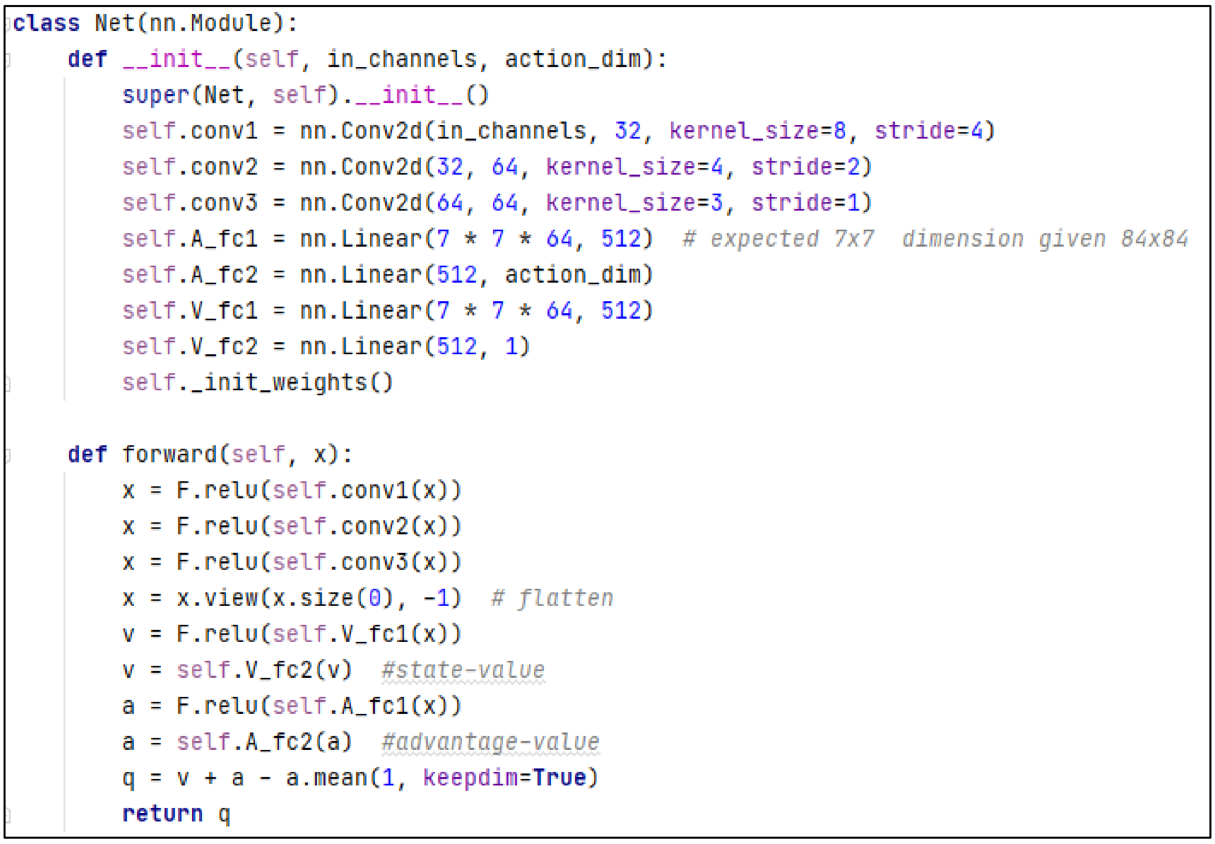
The layering structure is modified in the program by changing the original Net class:



```
class Net(nn.Module):
    def __init__(self, in_channels, action_dim):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc1 = nn.Linear(7 * 7 * 64, 512)  # expected 7x7  dimension given 84x84
        self.fc2 = nn.Linear(512, action_dim)
        self._init_weights()

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = x.view(x.size(0), -1)  # flatten
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

to have the state-value and advantage-value linear layers. Then proceeding to calculate the final action-value as explained in the Literature Review section. The implementation looks as follows:



```python
class Net(nn.Module):
    def __init__(self, in_channels, action_dim):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.A_fc1 = nn.Linear(7 * 7 * 64, 512)  # expected 7x7  dimension given 84x84
        self.A_fc2 = nn.Linear(512, action_dim)
        self.V_fc1 = nn.Linear(7 * 7 * 64, 512)
        self.V_fc2 = nn.Linear(512, 1)
        self._init_weights()

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = x.view(x.size(0), -1)  # flatten
        v = F.relu(self.V_fc1(x))
        v = self.V_fc2(v)  #state-value
        a = F.relu(self.A_fc1(x))
        a = self.A_fc2(a)  #advantage-value
        q = v + a - a.mean(1, keepdim=True)
        return q
```

3.1.5 Epsilon-greedy Policy

During the first few iterations of the algorithm, an epsilon decay technique was implemented to facilitate the initial exploration process. This value was set to decay at a rate of 0.995 (starting at 1) and stabilise at 0.01. During experiments, when the model would underperform as a result of converging too fast these values would be adjusted to allow a longer exploration period. The opposite would be attempted if convergence was not being reached before the set number of episodes.

During the testing of trained models, the ϵ value would be set very closely to 0. This is because to observe the agent's current exploration-free strategies there must be 100% chance of action exploitation.

3.2. Breakout Experiments

3.2.1 Early Iterations

The earliest prototype of the nature DQN model was built with the architecture explained in the previous sub-section. However, at this point in the project only the image conversion side of preprocessing had been implemented. There was also not much hyper-parameter tuning performed, hence it was expected that the model would underperform.

The graph below was generated using TensorFlow’s TensorBoard, which served as my metrics visualiser during early development.

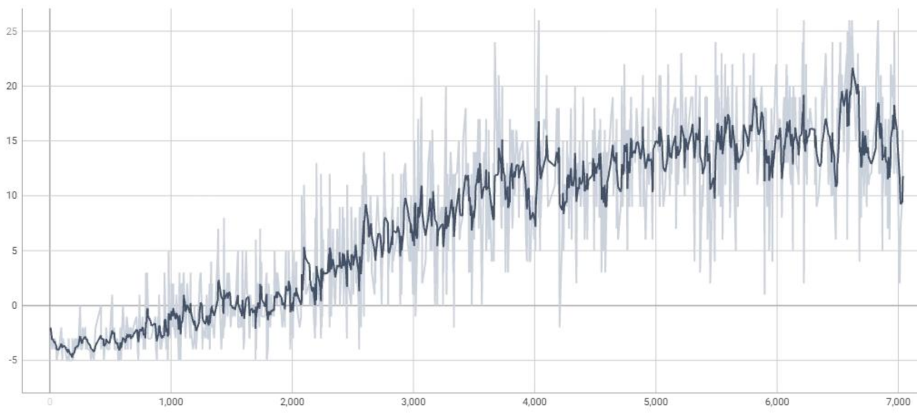


Figure 14: Early Breakout experiment.

The $BreakoutNoFrameskip - v4$ environment was trained for 7,000 episodes. The initial 2000 were dedicated to the warm-up process, where the agent begins learning at a much lower rate until it reached its normal learning rate of 0.00001. The reward discount factor γ of 0.9 and a batch size of 64 samples from the replay buffer were used. The target network was updated from the main Q-network every 100 steps. The replay memory capacity was set to 1,000,000, which ensure enough experiences were being stored.

3.2.2 Hardware Limitations

Most early experiments were conducted with an ASUS laptop consisting of an AMD Ryzen 5 processor and limited to 8GB of RAM. In the experiment covered above, the model took over 20.5 hours to train 7,000 episodes. The program utilised the CPU for training, which not only was a bottleneck to the model’s efficiency but also resulted in occasional memory overload errors. These would interrupt training for hours or force a complete restart before recurrent model backups were implemented.

These limitations were dealt with when I transferred my system to a more powerful machine with an Nvidia GTX 1070ti GPU and 16GB of RAM. This allowed me to run the training procedure in my GPU through the use of Nvidia CUDA, a parallel computing platform popular for accelerating graphics processing tasks. Going forward I was able to set my hyper-parameters to more computationally intensive values.

3.2.3 Later Iterations

The performance graphs showcased in the rest of this report have been generated by a class utilising the Matplotlib and NumPy libraries to load and display large amounts of collected data on each episode’s reward.

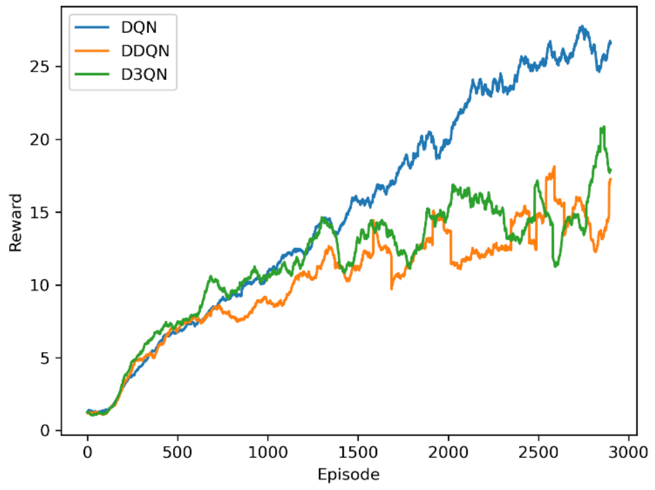


Figure 15: Post pre-processing Breakout experiment.

After implementing the explored preprocessing techniques like frame-stacking and reward clipping, the three models performed considerably better. In this specific experiment I also altered the ϵ-greedy policy to decay ϵ at a rate of 0.995 instead of 0.99 and to reach a minimum of 0.1 instead of 0.01. I did this to analyse the impact of enabling the agent to explore intensely for a longer period and continue exploring 10% of the time even when the model is converging. The policies formed until this point seemed suboptimal and so it is probable the agent was being limited by premature exploitation of the environment, causing a quick convergence. The discount factor γ was set to 0.99 at this stage. The batch size was set to 32 and the learning rate to 0.0001, which was now more computationally manageable after the system’s hardware upgrade. As demonstrated in the graph, DQN has the best learning curve between the models, achieving an average reward of 26.32 in the last 500 episodes. This is unusual for DDQN and D3QN, but this potentially a result of Double Q-learning’s additional complexity in the learning process that caused both models to be instable. Given that the models were only trained for 3,000 episodes it is possible their performance would have otherwise caught up to nature DQN’s.

3.3. Boxing Experiments

3.3.1 Early Iterations

The model created for Breakout first had to be adjusted for Boxing. This meant removing the reward deductions for losing a life since this feature was not part of the Boxing environment. That improvement to the Breakout algorithm was implemented after the previously covered experiments and its impact will be discussed in the Evaluation section. By experimenting with a faster learning rate of 0.001 and a bigger batch size of 64, the following results were captured after training a nature DQN:

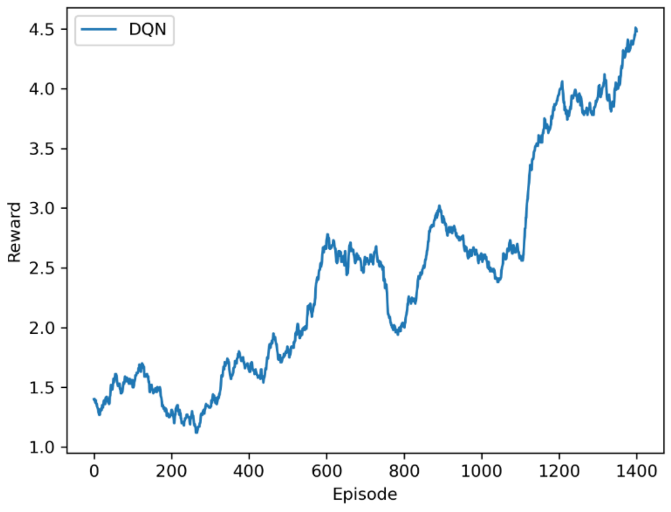


Figure 16: Early Boxing experiment

The previously stated hyper-parameter values were kept. For 1,400 steps the achieved scores were lower than expected. The faster learning rate caused unstable training dynamics, which are demonstrated by the oscillating learning curve. Although the agent shows clear signs of small improvement, it is better to tune the model until learning is more stable.

3.3.2 Low and High Gamma

The tuning of batch size, learning rate and number of episodes trained all had a positive impact on the Boxing agent’s performance. However, these results will be covered in the Evaluation section and this next experiment occurred afterwards.

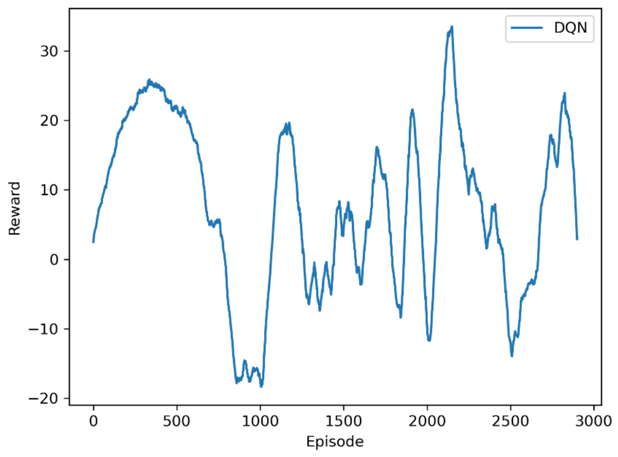


Figure 17: Low gamma experiment.

The function of the discount factor γ in any Q-learning derived methodology is to tell the agent how much it should base its decisions on the impact they have on future rewards. By setting the γ value to 0.9, the experiment is to test how the agent would perform with a more aggressive playstyle by encouraging its search for immediate reward (careless up-close punches). The batch size now set to 32 and learning rate to 0.0001. The outcome is what you could expect, an agent that behaved recklessly. Continuously reverting its strategies once facing the consequences of not fighting safely against the opponent to avoid negative reward.

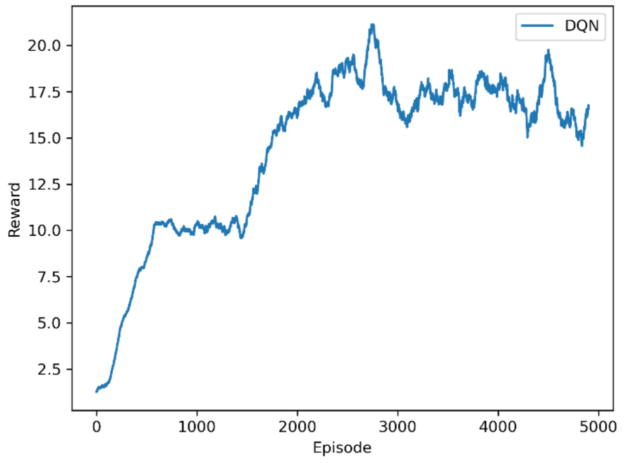


Figure 18: High gamma experiment.

With the same hyper-parameters and more training episodes, changing gamma to 0.995 had a surprising effect on the agent’s performance compared to the upcoming final results with $\gamma = 0.99$. When testing the rendered trained model, the agent demonstrated apprehension towards bring close to the opponent. This is due to the risk of losing reward in the future.

3.4. Final Results

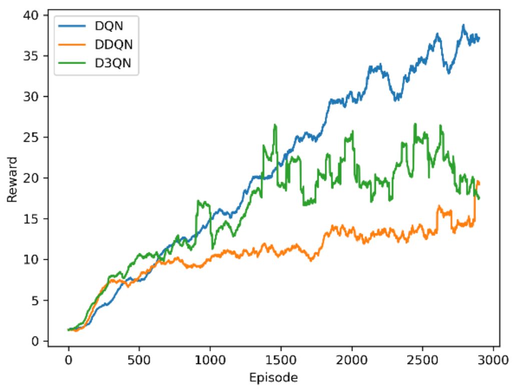


Figure 19: Final Breakout results.

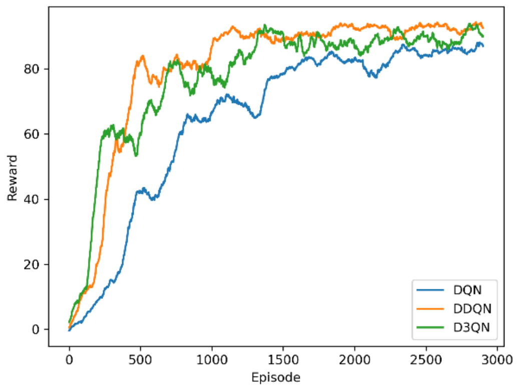


Figure 20: Final Boxing results.

Table 1: Final Breakout average scores

Episode	500	1000	1500	2000	2500	3000
Nature DQN	3.79	10.67	**17.37**	25.35	**31.47**	**36.07**
Double DQN	4.46	9.13	10.69	11.65	13.03	15.36
Dueling Double DQN	**4.93**	**11.31**	17.04	**19.65**	21.11	21.43

Table 2: Final Boxing average scores

Episode	500	1000	1500	2000	2500	3000
Nature DQN	10.61	51.62	69.93	81.78	82.75	85.61
Double DQN	28.99	**80.19**	**89.72**	**90.79**	**92.11**	**92.62**
Dueling Double DQN	**36.27**	72.43	83.42	88.13	88.34	89.79

Table 3: Final results hyper-parameters.

Parameter	LR	BS	γ	ϵ-min	ϵ-dec	episodes
Boxing	0.0001	32	0.99	0.01	0.995	3000
Breakout	0.0001	64	0.99	0.01	0.995	3000

1.1. Additional Results

The following is the performance of the final Boxing model when trained with OpenAI Gym’s expert difficulty option on:

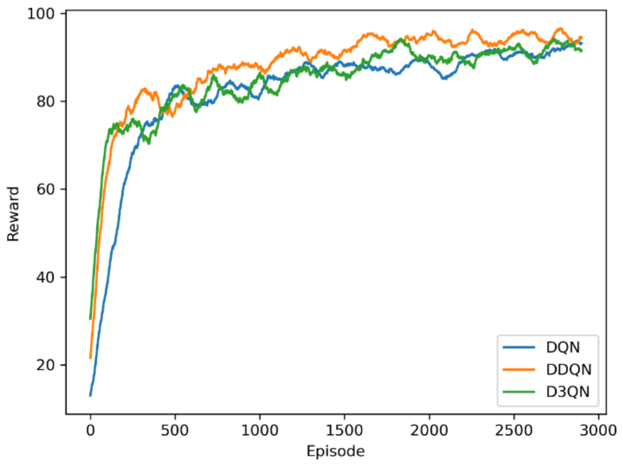


Figure 21: Boxing final model in expert difficulty.

Since the agent and opponent move at a much slower pace in this mode, all models were able to learn at far more effective rates. This was not included in the evaluation as this is not considered the default mode by other reinforcement learning papers.

The following is the performance of the final Boxing model when trained with a pre-trained final model from Breakout:

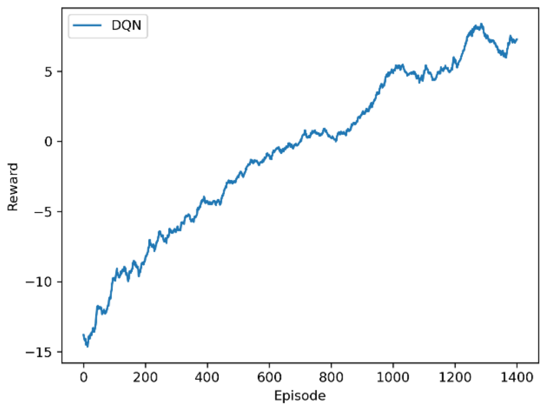


Figure 22: Transfer learning attempt.

This did not yield any impressive results as predicted in the Literature Review section.

2. Evaluation

2.1. Experiment Conclusions

From the experimental results, the findings are as follows:

- **All three DQN variations can achieve good training performance in the Atari Boxing environment.** All exceeding human-level performance of 12 points after just 1000 episodes.
- **Both Dueling DQN and Double DQN improved the performance of DQN in the Atari Boxing environment.** With the combination of both (D3QN) achieving an impressive average score of 92.62 in the last 500 episodes.
- **Dueling DQN learns at a faster pace than other variants.** In both environments it was able to achieve the best average score in the first 500 episodes.
- **Double DQN achieves contrasting convergence in Boxing and Breakout.** It converged the highest in Boxing and the lowest in Breakout.

My interpretation of these occurrences is the following:

- **Higher convergence of Double DQN in Boxing** can be attributed to the reduced overoptimism of action values during training.
- **Faster learning of Dueling Double DQN** can be attributed to its advantage in decoupling the estimation of state-value and state-advantages. Efficiently updating the network weights to reach optimal policies.
- **Difference in convergence between environments** can be attributed to the disparity of stochasticity between Atari Breakout and Boxing. As explored in the Literature Review, Boxing is highly stochastic when trained in this framework.
- **Overall good training performance** can be attributed to the Deep Q-Network architecture’s ability to handle high-dimensional and complex environments. Components such as experience replay, and multiple neural networks help the algorithm perform better than its predecessors.

2.2. Goals

Looking back to the goals I set for the implementation of this project:

- I was able to go achieve human-level performance in in Atari Boxing with all DQN approaches. I was however unable to achieve human-level performance with 2 DQN approaches in Atari Breakout.
- I was able to obtain comparable visual data between the 3 variations and the 2 environments. This allowed for a better analysis of the algorithm's learning dynamics.
- I was able to test out how modifications to the agent's hyper-parameters can impact its performance and training speed. Specifically, changes in learning rate, batch size, Gama and the ϵ-greedy's values (minimum and decay)
- I was able to experiment with the effectiveness of transfer learning techniques between Atari Breakout and Boxing.

2.3. Conclusion

Looking back to my learning objectives for this project:

1. I successfully explored a variety of reinforcement learning concepts and methodologies through the means of research and practical experience.
2. I successfully gained a deep understanding of nature DQN, Double DQN and Dueling DQN reinforcement learning algorithms. Including their architecture, training procedure and key mathematical components.
3. I successfully developed practical skills in implementing deep learning models using the PyTorch framework. This included building neural networks, defining loss functions and optimising hyper-parameters.
4. I successfully gained hands-on experience in training agents in complex environments provided by the OpenAI Gymnasium toolkit. Understanding the challenges of tuning a model for its optimal training procedure.
5. I successfully learned techniques for the evaluation of agent performance, including analysing learning curves, assessing convergence and interpreting experiments.
6. I successfully practiced formal documentation and communication skills by writing about my findings and experiments effectively throughout this report.

Overall, I found this project to be extremely valuable to my skillset and an invaluable beginning to my career in artificial intelligence.

Table of Figures

Table of Tables

References

Agarap, A. (2018). *Deep Learning using Rectified Linear Units (ReLU).*

Aranvidan, S. (2021). *An Analysis of Frame-skipping in Reinforcement Learning.*

Bellemare, M. (2013). *The Arcade Learning Environment: An Evaluation Platform for General Agents.*

Hasselt, H. v. (2015). *Deep Reinforcement Learning With Double Q-Learning.*

Kaiser, L. (2020). *Model Based Reinforcement Learning For Atari.*

Kingma, D. P. (2015). *Adam: A Method For Stochastic Optimization.*

le, N. (2021). *Deep reinforcement learning in computer vision: a comprehensive survey.*

Mnih, V. (2013). *Playing Atari with Deep Reinforcement Learning.*

Mnih, V. (2015). *Human-level control through deep reinforcement learning.* Nature.

OpenAI. (2022). *Gym Documentation.* Retrieved from https://www.gymlibrary.dev/api/wrappers/.

OpenAI. (2022). *Gym Documentation.* Retrieved from https://www.gymlibrary.dev/environments/atari/.

PyTorch. (2023). *PyTorch Conv2d documentation.* Retrieved from pytorch.org.

Ramirez, J. G. (2021). *Source tasls selection for transfer deep reinforcement learning: a case of study on Atari games.*

Saripalli, V. R. (2019). *AI Assisted Annotator Using Reinforcement Learning.*

Schwarzer, M. (2023). *Bigger, Better, Faster: Human-level Atari with human-level efficiency.*

Silver, D., Graves, A., Antonoglou, I., Riedmiller, M., Mnih, V., Wierstra, D., & Kavukcuoglu, K. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv: Learning*. Retrieved 4 26, 2024, from https://cs.toronto.edu/~vmnih/docs/dqn.pdf

Sutton, R. (2000). *Policy Gradient Methods for Reinforcement Learning with Function Approximation.*

Wang, Z. (2016). *Dueling Network Architectures for Deep Reinforcement Learning.*

Watkins, C. (1992). *Technical note Q-Learning.*

Zhang, C. (2024). *Double DQN Reinforcement Learning-Based Computational Offloading and Resource Allocation for MEC.*