UNIVERSITY OF BIRMINGHAM
School of Computer Science

Artificial Intelligence in Atari 2600 Games using Deep Q-Learning

Callum G. Tolley
Author

Alan P. Sexton
Supervisor

A thesis submitted to the University of Birmingham for the degree of Masters of Computer Science

Abstract

Games have been intertwined with the field of computer science almost from it's inception. Babbage explored building an AI for tic-tac-toe [1], and both Turing[2] and Shannon[3] developed programs to play chess.
As the power of computers has increased, more and more games have been solved by computers. Checkers by Chinkook, chess with Deep Blue, Jeopardy with Watson. More recently, deep learning has been used to accelerate this progress. The Deep Q-learning algorithm [4–6] has been used to play a large variety of Atari 2600 games with very little game-specific knowledge, only requiring a specification of the actions that can be taken, the screen pixels, and the score at each frame. This thesis includes a report of the implementation of this algorithm, as well as an analysis into the results.

Aknowledgements

I'd like to thank my mum and dad for their encouragement and support throughout.

Also to my supervisor, Alan Sexton, for advice during the final year project.

Finally to Jack and Peter, for all the laughs.

Contents

Abstract i

Aknowledgements ii

1 Introduction 1
1.1 Outline . . . 1

2 Background 2
2.1 Q-learning . . . 2
2.2 Neural networks . . . 4
2.2.1 Forward propagation . . . 4
2.2.2 Activation functions . . . 5
2.2.3 Loss . . . 5
2.2.4 Backpropagation . . . 5
2.2.5 Stochastic gradient descent (SGD) . . . 6
2.3 Convolutional neural networks (CNNs) . . . 8
2.4 Deep Q-networks (DQNs) . . . 9
2.4.1 Action repeat . . . 9
2.4.2 Frame processing . . . 10
2.4.3 History stack . . . 10
2.4.4 Reward clipping . . . 10
2.4.5 Lives . . . 11
2.4.6 No-ops . . . 11
2.4.7 ε-greedy policy . . . 11
2.4.8 Algorithm . . . 12
2.5 Glossary . . . 13

3 Implementation 14
3.1 Training . . . 14
3.1.1 Hyperparameters . . . 15
3.1.2 Neural network architecture . . . 17
3.2 Visualizer . . . 17
3.2.1 Controls . . . 19

4 Evaluation 20
4.1 AI behaviour analysis . . . 20
4.1.1 Pong . . . 20
4.1.2 Breakout . . . 22
4.1.3 Space Invaders . . . 23

4.1.4 Atlantis 24
4.1.5 River Raid 24
4.1.6 Star Gunner 26
4.1.7 Bank Heist 27
4.1.8 Fishing Derby 27
4.1.9 Montezuma's Revenge 28
4.2 Agent comparison 29
4.2.1 Google DeepMind comparison 30

5 Conclusion and future work **38**

List of Figures **39**

List of Tables **40**

List of Algorithms **41**

Bibliography **42**

A Zip Instructions **44**

Chapter 1

Introduction

This project is a primarily a replication study based on two papers by Mnih et al. [4, 5]. The first introduces the deep Q-network, and the second takes it further, runs it on more Atari 2600 games and provides more analysis into the internals of the learning process.

The aim of this project is firstly to replicate, and potentially improve, on the results attained by Mnih et al. [4, 5]. Secondly it is to provide a thorough analysis into the emergent behaviours that arise from this learning technique.

1.1 Outline

Chapter 1 introduces the topic and defines what this thesis will explore.

Chapter 2 analyses the background of the topic area. It explains what deep learning and Q-learning is, as well as the combination. The challenges and potential solutions are also outlined.

Chapter 3 describes the details the implementation – the challenges, pitfalls, and limitations encountered, and how they were overcome.

Chapter 4 contains the analysis of the implementation. Each Atari game is evaluated, and tests are presented in order to back up different hypotheses.

Chapter 5 summarizes the thesis, with suggestions on future work.

Chapter 2

Background

One of the main papers that this thesis is based on is a paper by Google DeepMind [5]. The authors combine traditional Q-learning with neural networks to create a deep Q-network that is able to play Atari 2600 games with little game-specific knowledge. They test 49 different games on the system, of which 22 can be played at an above-human level.

The key idea is the generality of the technique — it is the same algorithm that is applied to each game, with the same hyperparameters — rather than the individual scores on each game. More optimal algorithms can be found for each individual game, however this method is trying to find the optimal algorithm for *any* game.

The Arcade Learning Environment (ALE), is used to simulate the Atari 2600 [7]. Internally this uses the Stella emulator, which can run Atari games on the order of thousands of frames per second (FPS), compared to 60 FPS that the game is played at.

In this section the background of Q-learning and neural networks is described and explained, to give a better foundation of understanding for the thesis itself.

2.1 Q-learning

In reinforcement learning (RL) an agent interacts with an environment through a series of actions. For each action, the agent receives a reward which depends on the environment's state and the action taken. An overview is shown in Figure 2.1. At each step the agent takes an action and receives an observation of the world state and a reward signal. The objective of the agent is to maximize it's cumulative reward. As no domain-specific knowledge is given, the algorithm has to explore the environment and learn from this.

Q-learning is an on-policy reinforcement learning algorithm that is proven to converge to the optimal policy for an agent [8]. It is named after the function Q that returns the *value* of an action in a specified state. This value is used as a relative measure to compare different actions. It is defined in Equation (2.1).

$$Q(s_t, a_t) = r_t + \gamma \cdot \max_a Q\left(s_{t+1}, a\right) \tag{2.1}$$

The value of an action a_t in state s_t is equal to the reward r_t that the agent receives by taking this action, plus the value of the next state. The next state's value is calculated as the maximum potential value of Q, multiplied by a discount

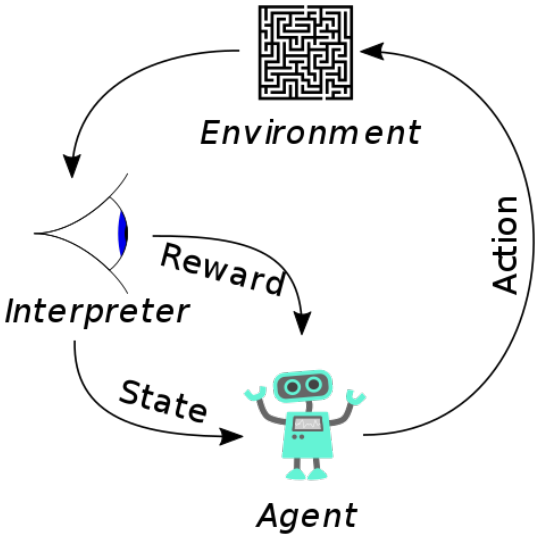


Figure 2.1: Reinforcement learning overview

factor γ. This is a hyperparameter, usually defined as 0.99, that forces Q to be finite.

This recursive definition is converted into an iterative algorithm with the update function given in Equation (2.2). Another hyperparameter, a learning rate α, is used to determine how much the agent learns at each step. It can be thought of the proportion of the old Q-value to forget.

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q\left(s_{t+1}, a\right)\right) \tag{2.2}$$

An ε-greedy strategy is used to explore the state space, using this update function. The pseudocode is given in Algorithm 1. N is the number of steps for the Q-learning algorithm to run. ε is a hyperparameter that controls the exploration-exploitation trade-off. A value of 1 means that the agent is fully random (i.e. continually exploring), and 0 means that the agent always chooses the action with the maximum Q-value. This is usually set to ≈ 0.1, or is sometimes a function of t.

Algorithm 1 Q-learning

$s_0 \leftarrow$ Initial state
for $t \leftarrow 1$ **to** N **do**
    $a_t \leftarrow \begin{cases} \text{random action}, & \text{with probability } \varepsilon \\ \text{argmax}_a\, Q(s_t, a), & \text{otherwise} \end{cases}$
    Get next state s_{t+1} and reward r_t by taking action a_t in environment
    Update Q: $Q(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q\left(s_{t+1}, a\right))$
end for

In traditional Q-learning, Q is implemented as a table, with an entry for each state-action pair. However, since video games have a large state space, Q is implemented as a neural network. This improves computational performance, since the network learns to generalize. This is further described in Section 2.4.

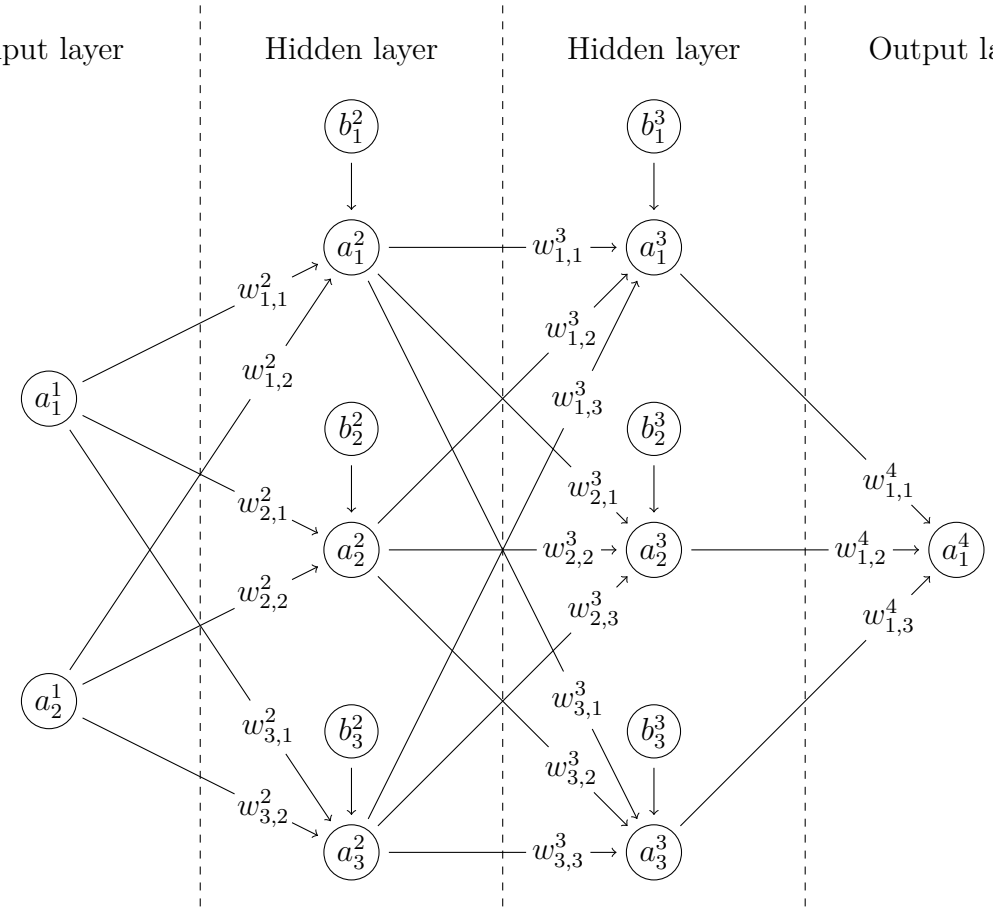


Figure 2.2: Example neural network

2.2 Neural networks

Neural networks can be viewed as a uniform way to approximate any given function. If we know a set of inputs and outputs of an unknown function, a neural network can be trained to represent that function that generated the data. This is a very useful property to have in cases where the function cannot be easily formulated, such as object recognition. They were introduced in 1958 and called *perceptrons* [9], however only recently, helped by GPUs, has it become practical to train them to solve real-world problems [10].

They have exploded in popularity recently, due to their recent successes in many different areas such as image recognition, image generation, and AI development [5, 11–14].

2.2.1 Forward propagation

Neural networks consist of a set of layers of size N. The input layer first, followed by a number of *hidden* layers, and an output layer last. Each layer i has a width l_i, which is the number of nodes in that layer, and an activation function ϕ^i. Each node j (except those in the input layer) has a set of weights $\mathbf{w}_j^i$ and a bias b_j^i that it uses to calculate the input z_j^i to the activation function, defined in Equation (2.3). An activation function ϕ^i is used to calculate the node's *activation* a_j^i, defined in Equation (2.4). The size of $\mathbf{w}_j^i$ is equal to the width of the previous layer, or l_{i-1}.

The weight from the node k in layer $i-1$ to the node j in the layer i is referenced as $w^i_{j,k}$. Figure 2.2 contains an example neural network, where $N = 4$, $m_1 = 2$, $m_2 = m_3 = 3$, and $m_4 = 1$.

$$z^i_j = \sum_{k=1}^{m_{i-1}} w^i_{j,k} \cdot a^{i-1}_k + b^i_j \tag{2.3}$$

$$a^i_j = \phi^i\left(z^i_j\right) \tag{2.4}$$

Forward propagation is the process of calculating the output from the input, i.e. normal operation of a function. Node values are computed layer-by-layer, from the first layer to the last layer, using Equations (2.3) and (2.4). GPUs can be used to increase the performance of this process, as there are many independent nodes in each layer of the network.

2.2.2 Activation functions

The activation function used in this thesis is the rectifier $\phi(x) = \max(0, x)$.

It is used as opposed to the logistic sigmoid function $\phi(x) = \frac{1}{1+e^{-x}}$, or it's more practical counterpart the hyperbolic tangent function $\phi(x) = \tanh(x)$, as it has been shown to improve training of neural networks [15].

2.2.3 Loss

In order to train the neural network to become better, a method of evaluating how effective (or ineffective) the network is must be defined. A loss function L is defined to encapsulate this. It takes two arguments: the expected (*target*) output x, and the actual output y, and returns a real number that represents how "far off" the network is from it's target.

The loss function used in this thesis is the squared error between these two values, multiplied by $\frac{1}{2}$ to simplify the gradient calculation.

$$L(x, y) = \frac{1}{2}(x - y)^2 \tag{2.5}$$

2.2.4 Backpropagation

By adjusting the network's parameters (weights and biases), L can be minimized. This is achieved by using a process called gradient descent, which requires the differential of the loss function L with respect to the parameter p, $\frac{\partial L}{\partial p}$. This is called the *gradient* of the parameter p. Backpropagation is used to calculate each gradient.

Backpropagation, like forward propagation, calculates layer-by-layer, however, unlike forward propagation, it performs this backwards, from the last layer to the first. The general idea is to use the chain rule to calculate these gradients. Equation (2.6) shows an example of the chain rule. Here, v can be any variable.

$$\frac{\partial L}{\partial p} = \frac{\partial L}{\partial v} \cdot \frac{\partial v}{\partial p} \tag{2.6}$$

The chain rule is used below, with Figure 2.3 to show that the gradient of each weight (2.7) and bias (2.8) can be calculated from $\frac{\partial L}{\partial a^i_j}$.

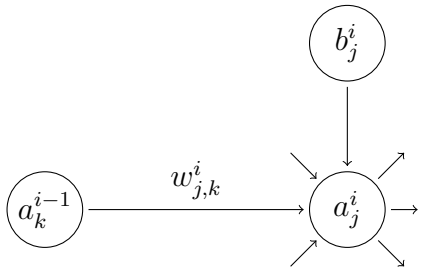


Figure 2.3: Example node

$$\frac{\partial L}{\partial w_{j,k}^i} = \frac{\partial L}{\partial a_j^i} \cdot \frac{\partial a_j^i}{\partial w_{j,k}^i} = \frac{\partial L}{\partial a_j^i} \cdot a_k^{l-1} \tag{2.7}$$

$$\frac{\partial L}{\partial b_j^i} = \frac{\partial L}{\partial a_j^i} \cdot \frac{\partial a_j^i}{\partial b_j^i} = \frac{\partial L}{\partial a_j^i} \cdot 1 \tag{2.8}$$

There are two cases for calculating $\frac{\partial L}{\partial a_j^i}$. The first is for the output layer, when $i = N$. This is the derivative of the loss function, in this case the derivative of Equation (2.5).

$$\frac{\partial L}{\partial a_j^N} = \frac{\partial}{\partial a_j^N}\left[\frac{1}{2}\left(a_j^N - y_j\right)^2\right] = a_j^N - y_j \tag{2.9}$$

The other case is when the layer is a hidden layer. Equation (2.10) shows that the gradient depends on the $(i+1)$th layer's nodes' gradients. This dependency is why backpropagation has to be performed layer by layer from the output layer backwards to the input layer.

$$\begin{aligned}\frac{\partial L}{\partial a_j^i} &= \sum_{k=1}^{m_{(i+1)}} \frac{\partial L}{\partial a_k^{i+1}} \cdot \frac{\partial a_k^{i+1}}{\partial a_j^i} \\ &= \sum_{k=1}^{m_{(i+1)}} \frac{\partial L}{\partial a_k^{i+1}} \cdot w_{k,j}^{i+1} \cdot \phi'\left(z_k^{i+1}\right)\end{aligned} \tag{2.10}$$

2.2.5 Stochastic gradient descent (SGD)

Gradient descent (GD) is the process by which a local minimum is found in a loss landscape, using the gradients calculated above. A global minimum is not guaranteed to be found, however in many practical tasks the local minimum is close to the global minimum. Additionally, optimizers (see below) can be used to mitigate this risk.

In normal GD, the whole training dataset is iterated through during an update, however in stochastic gradient descent (SGD) a batch of values, of size H_b, are sampled from the dataset to perform an update. This is necessary for large datasets, such as those evaluated in this paper, as it would take too long to iterate through the entire dataset (the entire replay buffer) each update. An explanation can be found in [16].

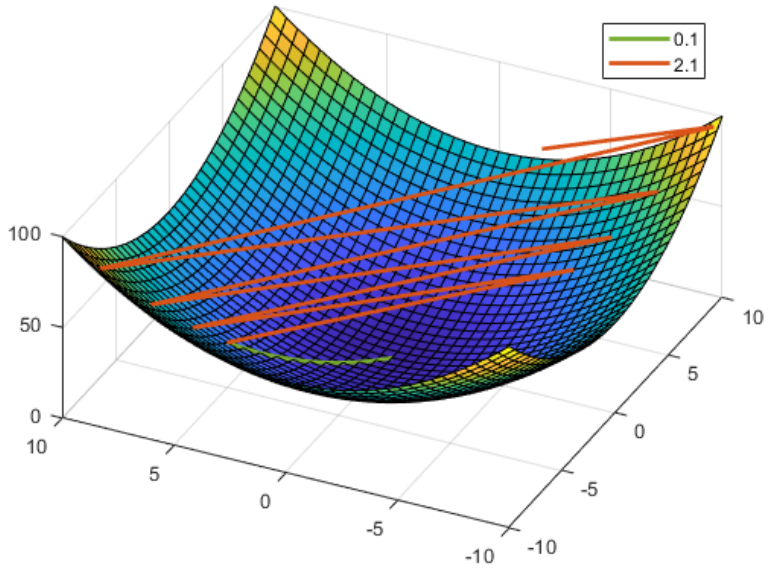


Figure 2.4: The loss function landscape of $f(x, y) = \frac{1}{2}x^2 + \frac{1}{2}y^2$, with paths of gradient descent with learning rates 0.1 and 2.1, both starting from $(-5, 5)$

At each SGD update, the parameters θ of the network are updated using Equation (2.11), where α is a learning rate, and $\frac{\partial L}{\partial \theta}$ are the gradients of the parameters.

$$\theta^{\text{new}} \leftarrow \theta - \alpha \cdot \frac{\partial L}{\partial \theta} \tag{2.11}$$

This process is repeated a predetermined number of times, or until a condition is set. The learning rate greatly affects how well the model will train. If the learning rate is too small, the model may take an extremely large amount of time to converge, conversely if it is too large it will never converge, and will spiral out of control. Figure 2.4 depicts a learning rate that is too large (2.1), and a learning rate that is just right (0.1).

Optimizers

An *optimizer* is a function that returns the new parameters of a network, given the current parameters and their gradients. Equation (2.11) just one of many different optimizers that are available. The Momentum optimizer keeps a momentum vector v and updates it each frame using Equation (2.12). A hyperparameter η, usually ≈ 0.9, discounts it at each step, dampening the momentum. This is then used in Equation (2.13) during SGD updates. This technique has been shown to increase the rigidity of the training with respect to the learning rate, allowing a wider range of learning rates to be used [16].

$$v_t \leftarrow \eta v_{t-1} + \alpha \cdot \frac{\partial L}{\partial \theta} \tag{2.12}$$

$$\theta^{\text{new}} \leftarrow \theta - v_t \tag{2.13}$$

In this thesis the Adam optimizer is used, which is similar to the Momentum optimizer. The Adam optimizer was introduced in [17]. Other than learning rate,

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

(a) Sobel kernel (horizontal)

(b) Input

(c) Output

Figure 2.5: Sobel kernel (horizontal) convolved with an image

the recommended hyperparameters defined in the paper are used, i.e. $B_1 = 0.9$, $B_2 = 0.999$, and $\epsilon = 10^{-8}$. See Section 3.1.1 for the reasoning behind the decision to use this optimizer.

2.3 Convolutional neural networks (CNNs)

CNNs are neural networks that have a convolutional layer in them. These layers are well-suited for analysing images. In the simplest case, given an image u, and a kernel k ($n \times m$ matrix of numbers), the pixel at (x, y) in the resultant image v is defined in Equation (2.14).

$$v_{x,y} = \sum_{i}^{n} \sum_{j}^{m} u_{x+i-\frac{n}{2}, y+j-\frac{m}{2}} \cdot k_{i,j} \tag{2.14}$$

An example kernel and it's result when convolved with an image is shown in Figure 2.5. The human-crafted kernel performs edge detection for horizontal edges. The kernels in a convolutional layer, however, are not human-crafted. Every value of every kernel is a parameter to the network, and is optimized as the other parameters are, using backpropagation (Section 2.2.4).

One of the reasons why convolution is successful with images is that there are less parameters to adjust, as opposed to a fully-connected hidden layer, and this helps the network to train. For example, in a 84×84 image there are 7,056 pixels. A dense layer of size 32 would mean 225,792 weights to train. A convolutional layer with 32 kernels, each of size 8×8, would only have 2,048 parameters. It also helps the network generalize, as it encourages the network's results to be invariant to translations of the input image.

Stride

The *stride* of a convolutional layer is the distance between each filter. Above, the stride is 1, however for an arbitrary stride s, Equation (2.15) is used.

$$v_{x,y} = \sum_{i}^{n} \sum_{j}^{m} u_{sx+i-\frac{n}{2}, sy+j-\frac{m}{2}} \cdot k_{i,j} \tag{2.15}$$

2.4 Deep Q-networks (DQNs)

Deep Q-networks form the basis of this thesis. They were introduced in [4]. In this paper, a network is created that learns how to play a multitude of Atari 2600 games. The network learns to play Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest and Space Invaders. The AI for each game has the same network architecture and hyperparameter settings, showing that the method generalizes to different games. This was followed up with a paper in Nature [5] that is a more thorough study of the method. In this paper they refine their method, and get an agent to achieve human-level skill on 29 different Atari games.

Traditionally in Q-learning, Q is represented as a table. Each pair of states and actions has an entry in the table, mapping to the expected reward. However, in games with high state spaces, such as Atari games, this is computationally impractical. Every state that it memorized is unlikely to be seen again. To fill up the table enough to generalize would take an excessive amount of time and memory.

In a DQN, however, a neural network with parameters θ tries to estimate the function Q. The correct Q-value for state-action pairs is calculated through play, using the method described in Section 2.1. The neural network then learns to approximate Q given this data. In [5] two problems with this method are described, and two solutions are proposed.

Firstly, the network does not perform well if the training data is fed in sequentially, as the game is played. This is because the distribution of the data is not *i.i.d*, i.e *independent* and *identically distributed*. In other words, the training data is not a faithful representation of the function that we are trying to approximate. This is a well-known issue with machine learning in general. Samples of training data generated sequentially from play are highly correlated with each other - they are not very *independent*.

In reinforcement learning, we also have the issue that both the training data and the target Q are changing constantly, as we learn more about the environment. A small change to Q can significantly change the policy, and hence the distribution that the neural network is trying to learn. This means that the learning process is constantly playing catch-up, trying to chase it's own tail.

This paper attempts to mitigate these problems with two features. First, by using a novel mechanism called experience replay. This reduces correlation in the observation sequence, increasing independence of the input data. Instead of feeding the observed data directly to the optimizer, the frames are stored in a buffer. Data is sampled from this every number of steps and used to perform gradient descent.

Second, a separate network $\hat{Q}$ with parameters $\hat{\theta} \leftarrow \theta$ is created that is used for generating the target Q-values for the original network to optimize to. This is called the *target* network, with the network being trained called the *model* network. The target parameters are set to the model parameters every H_t SGD updates.

2.4.1 Action repeat

At each step, the action chosen by the DQN algorithm is repeated for H_r frames. As the majority of the time spent training is forward and backpropagation, this roughly decreases the training time by a factor of H_r, without significantly decreasing agent quality. This thesis explores the effect of changing this parameter. It also decreases

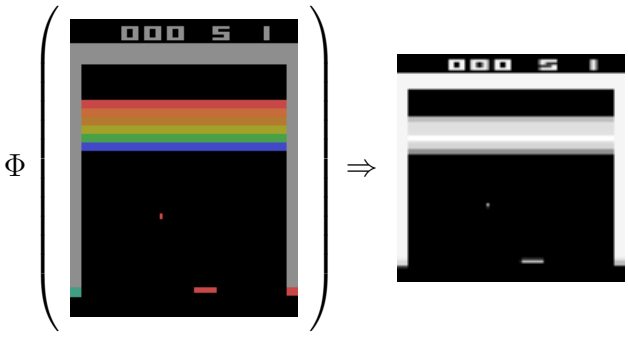


Figure 2.6: Frame processing

the time to a reward, from the perspective of the network, and so increases training speed further.

2.4.2 Frame processing

The frame of the Atari 2600 is 160x210x3 (RGB), however the frame passed to the network is downsampled to 84x84 (greyscale). This is to reduce the computation and memory requirements of the system. In addition the training time is reduced, as it reduces the number of parameters in the neural network. This processing function is referred to as Φ. Figure 2.6 demonstrates the conversion.

2.4.3 History stack

A stack of the H_h most recent frames are input to the agent, as it allows the agent to observe and estimate velocity and acceleration of objects in the scene, and gives greater context to draw from. For example, Figure 4.3a shows a frame from Breakout, but it is impossible to tell which direction the ball is going in without some other external information. A recurrent neural network doesn't need this history stack, as they can remember state internally, demonstrated in [18], however in this thesis a simple DQN is used.

2.4.4 Reward clipping

During training rewards are clipped to be in the range $[-1, 1]$. This is so that the same learning rate can be used for all games, as some games have rewards on a wildly different scale. The problem is that this makes the agent weigh all rewards equally, making the agent suboptimal in games like Pacman, where it is more optimal to get cherries and chase ghosts than eat pellets. This video demonstrates the difference: `https://youtu.be/NBczahyJLNw`. This problem was solved in [19] by estimating the mean and variance of rewards, and compensating for it, however this was not implemented due to time constraints.

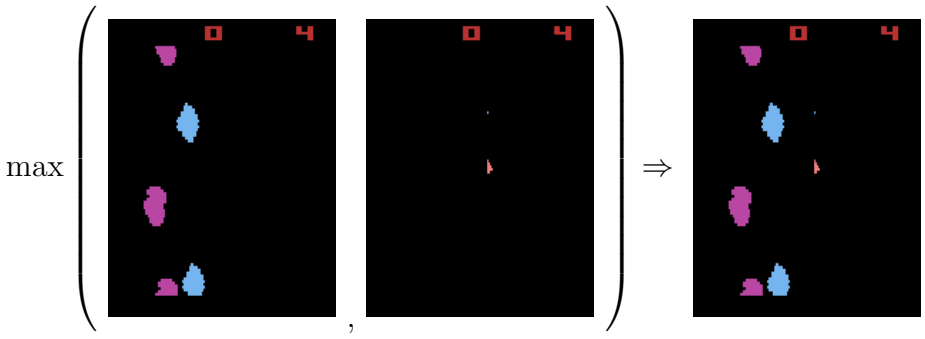


Figure 2.7: Max frame processing

2.4.5 Lives

When the player plays an Atari game, usually there are multiple lives, e.g. 3 lives for Space Invaders, 5 for Star Gunner. It has been shown, however, that training is faster if the agent only has 1 life. This is so that the agent learns that losing a life is bad faster than it otherwise would.

Also, due to action repeat, not every frame is seen. This is a problem for some games (like Asteroids), as objects are sometimes only rendered every odd or even frame due to technical limitations of the Atari 2600. To fix this issue, each frame passed to the network is actually the maximum of the previous frame and the current frame. Figure 2.7 shows this process.

2.4.6 No-ops

To increase the randomness of games, a number of no-op actions (default action that is the equivalent of the player pressing nothing) are taken whenever the game is reset. This number is sampled from a uniform distribution in the range $[0, H_n)$, where H_n is a hyperparameter (Section 3.1.1).

2.4.7 ε-greedy policy

Since this algorithm uses a ε-greedy exploration policy, ε needs to be defined. In the DQN paper, ε is linearly annealed from 1.0 to 0.1 over the first $1,000,000$ frames. In this thesis the same function is used, as defined in Equation (2.16). ε_0, ε_T and T are hyperparameters (Section 3.1.1).

$$\varepsilon_t = \begin{cases} \varepsilon_T + (\varepsilon_0 - \varepsilon_T) \cdot \frac{t}{T}, & \text{if } t < T \\ \varepsilon_T & \text{otherwise} \end{cases} \tag{2.16}$$

2.4.8 Algorithm

Algorithm 2 outlines the Deep Q-learning algorithm described in [5]. N is the total number of training steps. Hyperparameter (H) values are discussed in Section 3.1.1.

Algorithm 2 Deep Q-learning

Initialize replay buffer B with capacity H_s
Populate B with transitions by running a random agent for H_z steps
Initialize Q-value function Q with random parameters θ
Initialize target Q-value function $\hat{Q}$ with parameters $\hat{\theta} = \theta$
for $t \leftarrow 1$ **to** N **do**
    if $t = 1$ **or** episode has terminated **then**
        Initialize s_t to default state, τ_t
        Initialize recent frame stack S with capacity H_h
        $n \leftarrow$ a random number in the range $[0, H_n]$
        Take no-op action $n \cdot H_r$ times and update s_t and d ▷ Section 2.4.6
    end if
    $a_t \leftarrow \begin{cases} \text{random action}, & \text{with probability } \varepsilon \\ \text{argmax}_a \, Q(s_t, a), & \text{otherwise} \end{cases}$
    Take action a_t and repeat H_r times, calculating summed clipped reward r_t and set of frames F
    Terminal $\tau_{t+1} \leftarrow$ true if episode has ended, false otherwise
    $f \leftarrow \max\left(\Phi\left(F_{H_r-1}\right), \Phi\left(F_{H_r}\right)\right)$ ▷ Section 2.4.2
    Store f in S
    Get next state s_{t+1} from S
    Store transition tuple (s_t, a_t, r_t, s_{t+1}) in B
    if $t \mod H_u = 0$ **then** ▷ SGD update
        Sample size H_b batch of transitions (s_j, a_j, r_j, s_{j+1}) from B
        Set $y_j \leftarrow \begin{cases} r_j, & \textbf{if } \text{episode terminates at step } j+1 \\ r_j + \gamma \max_a \hat{Q}\left(s_j, a; \hat{\theta}\right), & \text{otherwise} \end{cases}$
        Perform gradient descent on the loss function L using y_j as target Q-values, i.e. minimize L using the gradient $\frac{\partial}{\partial \theta} L\left(y_j, Q\left(s_j, a_j; \theta\right)\right)$
    end if
    if $t \mod \left(H_u \cdot H_t\right) = 0$ **then** ▷ Target update
        $\hat{\theta} \leftarrow \theta$
    end if
end for

2.5 Glossary

Name	Description
Episode	Full playthrough of a game from the initial frame until the terminal frame.
Epoch	In this thesis an epoch is defined as 250,000, and has no meaning other than as a shorthand.
Score	The unclipped reward that is received from the emulator.
Reward	The clipped reward that is received from the emulator. See Section 2.4.4

Table 2.1: Glossary

Chapter 3

Implementation

One of the major challenges with this project was simply the amount of resources that were required to train the agents. Before optimization, each agent took around a day to train for 1,000,000 steps. With Google DeepMind's hyperparameters, 7-8GB of RAM is used. One of the major constraints was that the computer was frequently running out of memory, interrupting training.

The majority of the memory usage was the replay buffer, for which Google used a size of 1,000,000 frames. $84 \times 84 \times 1,000,000 = 7,056,000,000$bytes $\approx$ 7GB. Many of the machines that were utilized in this study only had 8GB of RAM, and so it didn't take much to push the machine over the edge and for it to kill the process. This is why the replay size in this thesis is 100,000, which uses around .7GB. This change allows for around 8 games to be trained in parallel, before spurious out of memory errors become more frequent.

The opaqueness of neural networks was also a challenge. Since neural networks are essentially a black-box solution, it was difficult to track down bugs in the code. The long training time further exacerbated this problem.

3.1 Training

There were four different sets of machines that I had at my disposal: my personal computer with GPU, a Google Colab machine with GPU, and 8 machines with GPUs and approximately 150 without GPUs in the computer science lab.

One of the problems with the lab machines was that occasionally training would fail. The computer was inadvertently turned off by someone in the lab, it would run out of memory, or the process would be killed.

The other problem was the limited quota, only 2200MiB. 1500MiB was taken up by Python libraries, of which 1GiB was pytorch, and 319MiB by the CUDA DNN library, leaving 381MiB for other data.

The solution to the first problem was saving the agent's training state regularly (consisting of the model Q, the replay buffer, and various other metadata). However, this, as discussed above, is a large amount of data, much larger than 381MiB. A solution to this was to save the agent's state to a remote machine. In this case, it was saved to Google Storage. Gzip compression was used to reduce the time spent saving.

After these problems were solved, training was run in parallel on the lab machines, both with and without GPUs. This was accomplished by using a *supervisor*

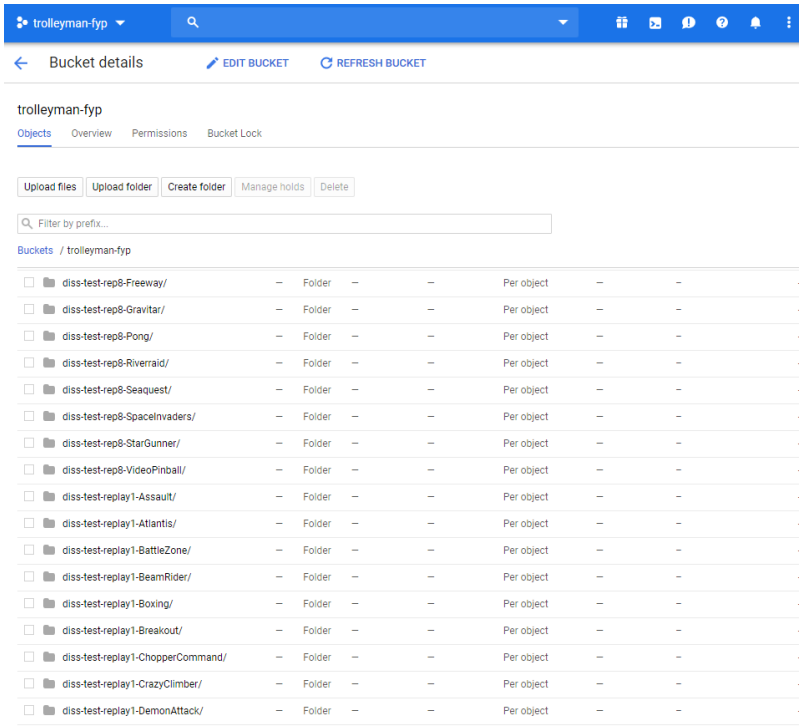


Figure 3.1: Screenshot of the Google Cloud Storage web interface

program that, when assigned a set of agent specifications, would manage training for those agents. It would keep track of which jobs were currently running, which had just died, and which computers were off limits (i.e. which ones were in use by other students). When restarting a training session, the supervisor would pass arguments to the training program which would cause it to load the last saved state from Google Storage.

Although the 150 lab machines without GPUs were much slower to train on, since there were so many a large amount of agents could be trained in parallel. Even more than 150 could be trained at once, due to the inherently single-threaded nature of the deep Q-learning algorithm. Since each computer had a multiple core CPU, many agents could be trained on the same machine without affecting performance. This applied to the GPU machines as well, since the training was usually CPU limited, and would only use $\approx 20\%$ of the GPU during training.

One of the techniques that may be effective in this situation is the IMPALA algorithm [20]. This is essentially a distributed version of deep Q-learning. It would have been great in the labs, as it doesn't even need GPUs to be fast. In fact, the paper mentions that it runs better with a large amount of CPUS, the exact environment of the computer science labs. Due to time constraints however, this wasn't implemented.

3.1.1 Hyperparameters

Table 3.1 shows the hyperparameters used in this thesis. For some tests the hyperparameters are different from the ones shown here, in which case it is noted. These hyperparameters are similar to the ones in the Google DeepMind paper [5], however there are differences.

Hyperparameter		Value	Description
H_h	history length	4	See Section 2.4.3.
H_r	action repeat	4	Each action the agent selects is repeated this many times. See Section 2.4.1.
H_b	batch size	32	Number of transitions that the network is trained over during each stochastic gradient descent (SGD) update
H_s	replay size	100,000	Maximum number of recent transitions stored, to be sampled from during SGD updates
H_t	target update frequency	2,500	Number of SGD updates between each target network update
H_u	update frequency	4	Number of steps (action selections) between each SGD update
γ	discount	0.99	Discount factor used in Q-learning update. See Section 2.1.
α	learning rate	0.00001	Learning rate used by the Adam optimizer
ε_0	initial epsilon	1	
ε_T	final epsilon	0.01	See Section 2.4.7.
T	final epsilon step	1,000,000	
H_z	replay start size	50,000	A random agent is run for this number of steps before training starts to populate the replay memory with frames.
H_n	no-op max	30	Max number of no-ops performed at the start of every episode by the agent. See Section 2.4.6.

Table 3.1: Hyperparameters

The hyperparameter differences include the replay size H_s, with the reasoning mentioned above, the optimizer, the learning rate α, and the final epsilon value ε_T.

The optimizer used in [5] is the RMSProp optimizer, however in this thesis the Adam optimizer is used instead. This is because the results of the paper could not be replicated using the RMSProp optimizer. One of the simplest games on the Atari 2600, Pong, didn't learn, even after 7,000,000 steps.

The learning rate was also changed, from 0.00025 to 0.00001, to increase the training stability of agents. This may have been necessary because of the change in optimizer. Tests of agents that were trained using both learning rates are presented in Chapter 4.

An epsilon of 0.01 was chosen instead of 0.1 to increase the training speed of the agent. The reasoning is that a 0.1 epsilon will cause the agent to miss 10% of the actions that it tries to perform, causing the training to terminate early in some long-term games where precision is needed.

For example, in the case of Breakout, an epsilon of 0.1 would mean that it would

miss 10% of it's shots. As it takes around 40 hits of the ball for it to penetrate behind the wall of bricks, there would only be a $(1-0.1)^{40} \approx 1.5\%$ chance of reaching that state. However, with an epsilon of 0.01, there would be a $(1-0.01)^{40} \approx 66.9\%$ chance of reaching that state. This is of course assuming that the agent has learnt enough to be able to reach this point in the first place. Again, this is the problem of exploration vs. exploitation.

3.1.2 Neural network architecture

The architecture of the neural network was chosen to be the same as in [5].

Shape	Layer	Parameters	Description
$H_h \times 84 \times 84$	Input	–	History stack of frames (Section 2.4.3)
$32 \times 20 \times 20$	Convolution	$2{,}048h + 32$	32 filters of size 8×8, with stride 4
— " —	ReLU	–	Rectifier nonlinearity activation function, as discussed in Section 2.2.2
$64 \times 9 \times 9$	Convolution	32,832	64 filters of size 4×4, with stride 2
— " —	ReLU	–	–
$64 \times 7 \times 7$	Convolution	36,928	64 filters of size 3×3, with stride 1
— " —	ReLU	–	–
512	Dense	1,606,144	Fully connected layer
— " —	ReLU	–	–
a	Dense	$512a + 1$	Outputs a Q-value for each action. a is the action size hyperparameter, specific to each game.

Table 3.2: Neural network architecture

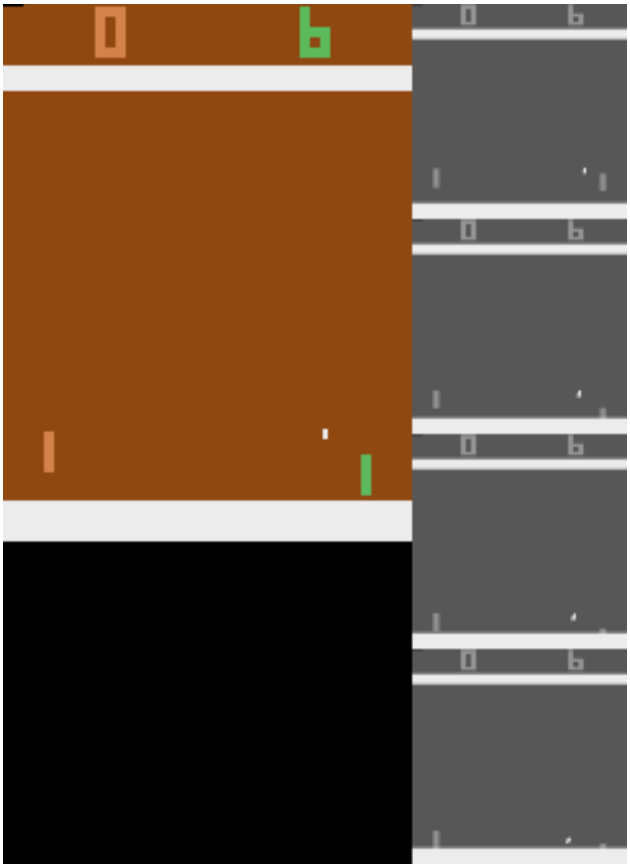
3.2 Visualizer

To demonstrate the DQN that was developed, a visualization system was implemented. This was developed to be able to demonstrate the DQN running in real-time, at 60 frames per second (FPS). This consists of two windows - the first a view of the game state, and the second a view of the agent state.

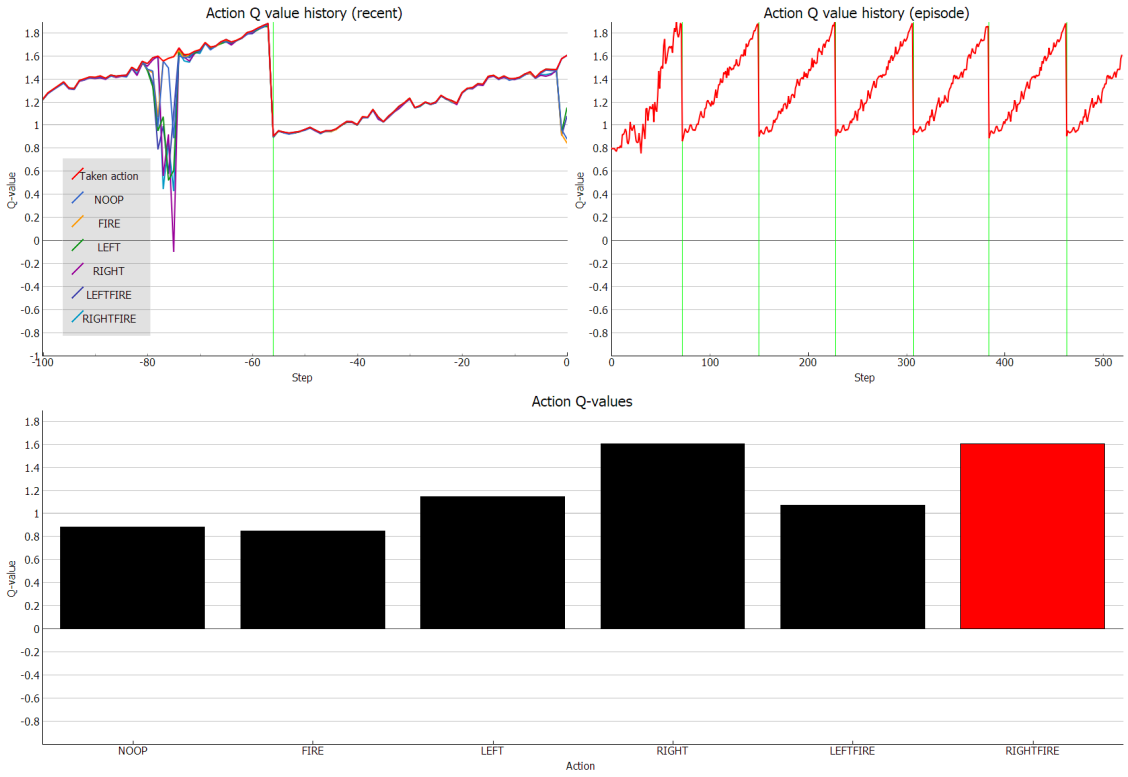
Initially the live plot of the agent state was implemented using matplotlib, a plotting library for Python. However, this became an issue due to it's slow update speed. To resolve this problem, the plot view was redeveloped using pyqtgraph, resulting in an improvement from 4 FPS to 60 FPS. Another positive of pyqtgraph was that the plots were interactive in real-time. The user can zoom in on portions of the Q-values graph to gain a better understanding of what is happening, and save the plot to a file if needs be.

Figure 3.2a shows the game state view during a game of Breakout. The unprocessed current frame is shown in the top left, and the history of processed frames that are input into the network are shown on the right, with the most recent frame being at the top, until the H_hth frame at the bottom.

Figure 3.2b shows the agent state view. There are three graphs, displaying Q-values in the short, medium and long term history respectively. The bar chart shows



(a) Game state



(b) Agent state

Figure 3.2: Visualizer

the short term – the Q-values for each action in the current state. The red bar is the taken action, usually the highest, but sometimes not, due to ε. In this case, the maximum Q-value is highlighted in green.

In the top-left is the medium term graph that displays the Q-values for every action in the past 100 steps. The long term graph is in the top-right, and this shows the Q-values for the entire episode. In this graph only the taken action's Q-values are shown, to reduce visual clutter. The green vertical lines shown are positive rewards. Negative rewards are shown as red vertical lines.

Note that *step* in these plots is not referring to the number of frames, but the number of update steps since the start of the episode. These are different because of action repeat (Section 2.4.1).

These plots are used in Chapter 4 to provide context to the AI's behaviours.

3.2.1 Controls

The visualizer state is controlled using the keyboard. A table of keys, along with their use, is given in Table 3.3. Game keys can be held down together to take multiple actions at once. A different number of game keys are supported depending on the game.

Key(s)	Description
`<escape>` or Q	Quit
F	Pause/resume
X	Step frame (when paused): Steps the environment forward one frame
C	Step agent (when paused): Steps the environment to the next DQN update. Different to stepping a frame due to action repeat (Section 2.4.1).
T	Double FPS (max 120)
G	Half FPS (min 5)
U	Toggle user-controlled mode
User Controlled Mode	
`<space>`	FIRE
`<left>`	LEFT
`<right>`	RIGHT
`<up>`	UP
`<down>`	DOWN

Table 3.3: Visualizer controls

Chapter 4

Evaluation

This chapter first analyses in depth individual game behaviours that the AI has learnt in Section 4.1, and then an overall evaluation is performed in Section 4.2, using humans, random agents, and the Google DeepMind paper [5] as comparison.

4.1 AI behaviour analysis

In this section the notable emergent behaviours of the AI in a few different games are analysed.

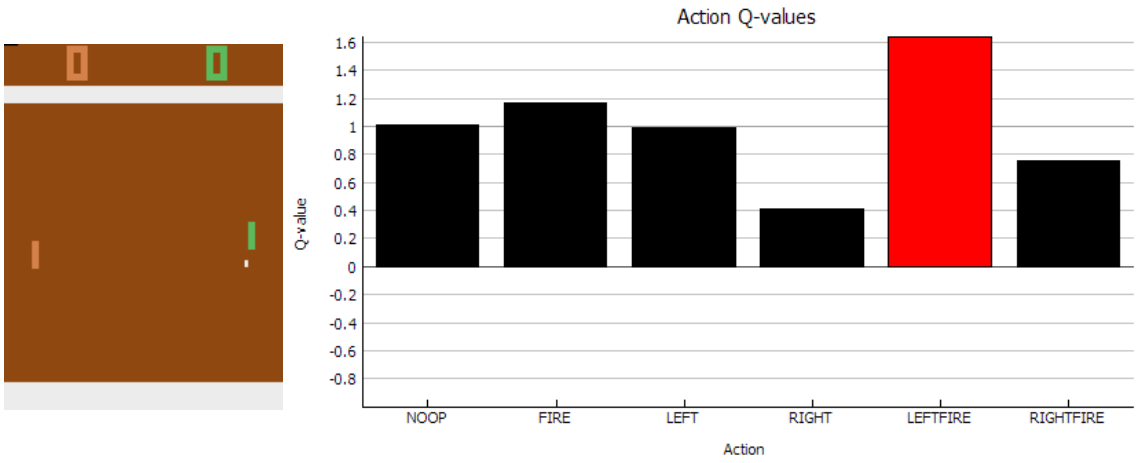
4.1.1 Pong

In Pong, the player is pitted against the game's AI. The goal is to hit the ball past the game AI's paddle. Each time this happens, the player scores a point, and vice versa for the AI. The first to 20 wins. Figure 4.1a shows an example frame from the game. The game's AI is on the left in orange, and the player is on the right in green.

The built-in game AI is a simple one: it moves up if the puck is above its paddle, and down if the puck is below its paddle. It cannot move as fast as the player though, so the challenge is how responsive and accurate it is compared to the player.

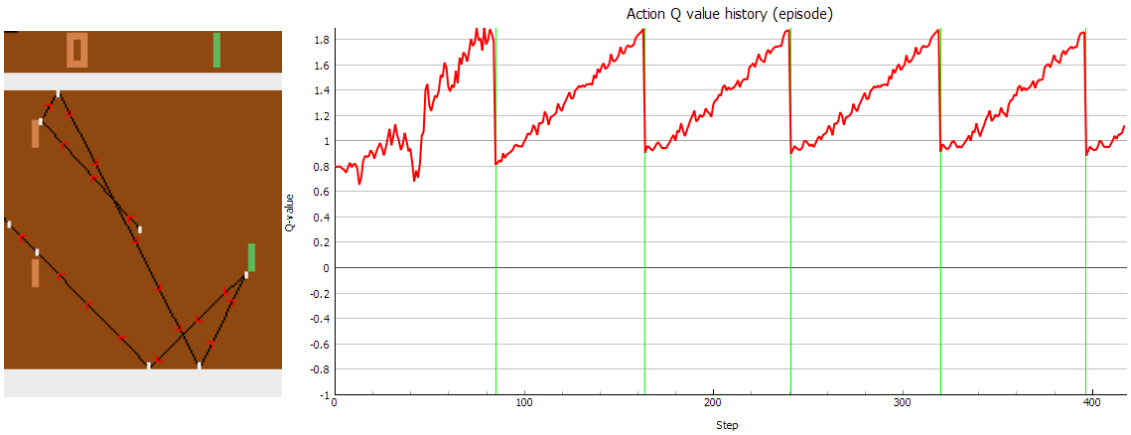
Figure 4.1b shows the Q-values for each potential action that the player can take. Note that the puck is moving down and to the right. The action that the AI has selected (the action with the largest Q-value) is the LEFTFIRE action, a combined action of LEFT and FIRE. In Pong, the LEFT input makes the player's paddle go down, and the FIRE button is used to accelerate the ball. If the player presses the FIRE button when the puck hits the paddle, the ball gets a speed boost. The AI has learned that this helps it to score points, as it is harder for the built-in AI to defend against.

In this implementation, the first point scored is the hardest for the AI to complete successfully. This is because the ball can be in a variety of different locations and velocities, and so there is a larger state space to explore and learn. Once it has scored however, the ball always appears in the same location. After this point, the AI has learned to memorize a series of moves to perform to score consistently. The trajectory of the ball is shown in Figure 4.2a, as well as the final location of the built-in game AI's paddle. This behaviour can also be seen in Figure 4.2b. The Q-values before the first point is scored are much noisier than the rest of the Q-values,



(a) *(b)*

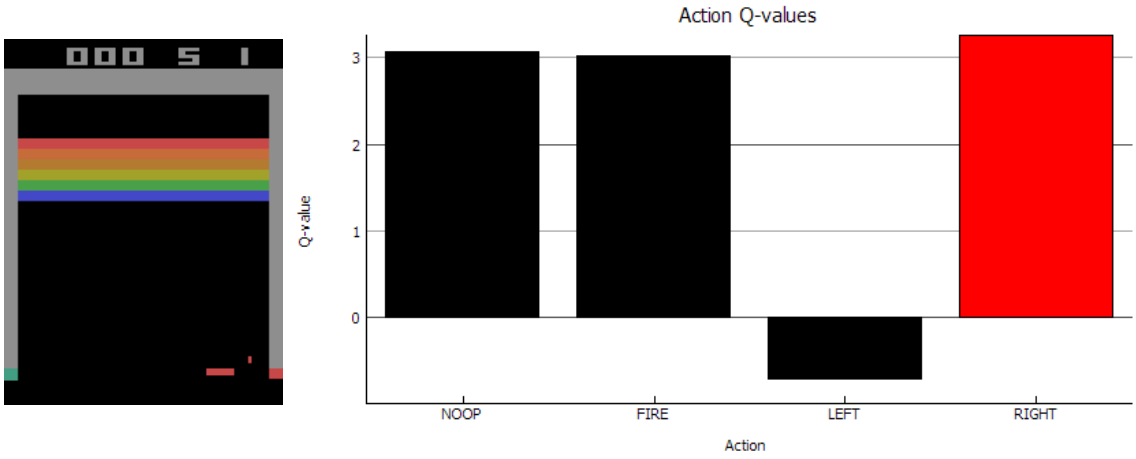
Figure 4.1: Sample Pong frame with action Q-values



(a) Pong ball trajectory

(b) Maximum Q-value history

Figure 4.2: A typical game of Pong



(a) *(b)*

Figure 4.3: Sample Breakout frame with action Q-values

which shows that the AI is much more unsure about the state that it is in.

Also, note the drop in Q-values just after a point has been scored. The AI has learned to decrease its expectations when it recognises that a point has just been scored, presumably by detecting that the ball has disappeared off of the leftmost side of the screen.

4.1.2 Breakout

In Breakout the player controls a paddle to stop a ball from reaching the bottom of the screen. If it reaches the bottom, the player loses a life. The player scores points when the ball hits any of the bricks. The brick that was hit is also destroyed.

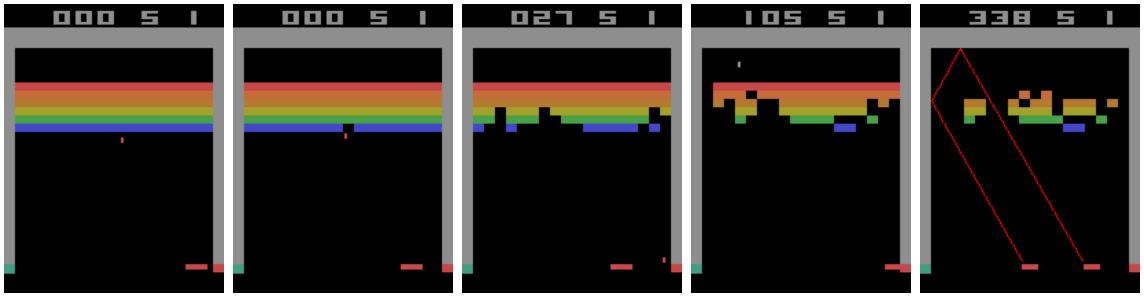
A sample frame of Breakout is given in Figure 4.3a, with each action's Q-values in Figure 4.3b. The ball is moving towards the bottom left of the screen. The Q-values indicate that the agent is confident about being able to hit the ball without moving (NOOP and FIRE), but it instead chooses to select the RIGHT action, as it has a higher Q-value.

Figure 4.4 shows a typical game of Breakout, with labels at certain interesting steps. *(a)* is the frame just before the reward is received from the ball hitting the brick. On the Q-value graph, it is just before the predicted Q-value decreases. *(b)* is the frame immediately after this. This shows that, to the AI, the expected reward decreases just after it receives a reward. This makes intuitive sense, as the Q-values up until this point have all had this reward factored into it.

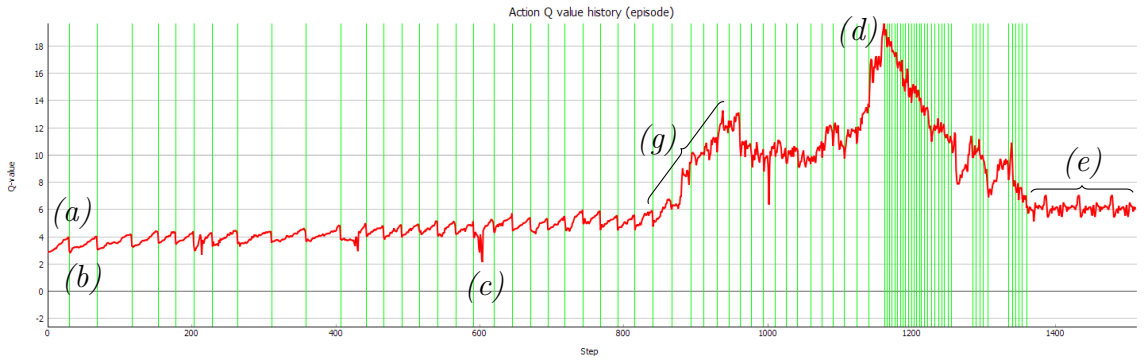
At *(c)*, step 605, the Q-values take a dive. The ball in this frame is moving towards the bottom right. It appears to be this way because the AI is uncertain about it's ability to hit the ball back again.

Another behaviour to note occurs during the time marked by *(g)*. The Q-values increase dramatically from their usual baseline before this point. 5 rewards are earned during this time frame. These 5 rewards are earned by the AI for breaking the leftmost bricks to form a tunnel on the left hand side of the frame. The tunnel can be seen in *(d)*.

To explain this behaviour, another game mechanic of Breakout must be understood. When the player manages to hit the 4^{th} level of the wall, the ball's speed



(a) Step 30 *(b)* Step 31 *(c)* Step 605 *(d)* Step 1162 *(e)* Steps 1374+



(f) Maximum Q-value history

Figure 4.4: A typical game of Breakout

increases. Since a higher ball speed leads to more frequent rewards, and the agent needn't worry about reaction times like a human, it explains why the Q-values increase. The other reason could be that the tunnel is valuable in that it increases the likelihood of a breakthrough scenario (as in *(d)*). It may be that it is a combination of both.

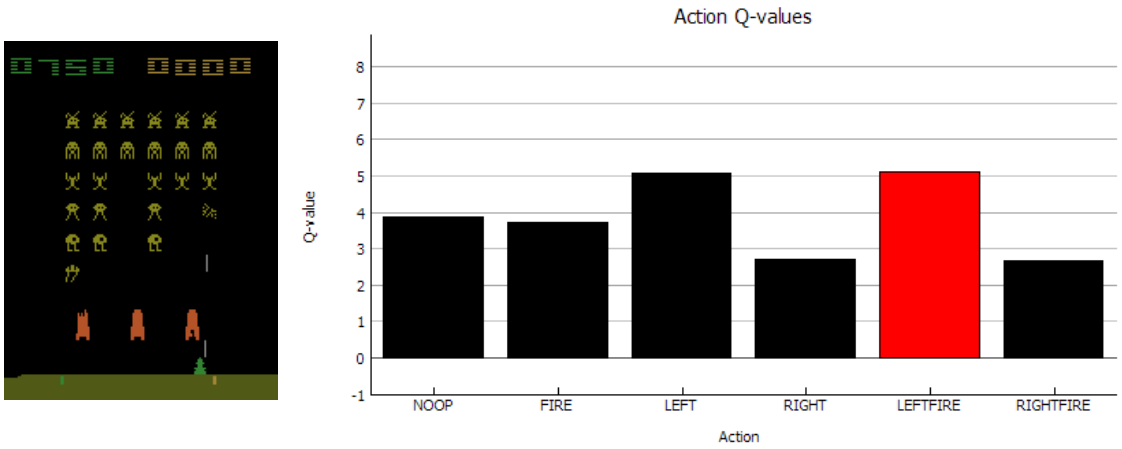
The optimal strategy in Breakout is to hit the ball repeatedly in the same spot, to create a tunnel through the wall, and then to get the ball through this behind the wall. Once there, it can bounce around and destroy bricks without the player needing to move the paddle to bounce it back again. We can see this in *(d)*. This is near the peak of the Q-value graph, and it shows that the AI has learnt that this state, with the ball behind the wall, is a valuable state.

When this happens, the ball bounces around above the wall and gains reward for the AI from each brick that is destroyed. As these bricks run out, the predicted Q-values drop as well, showing that the AI learns to incorporate this information into it's decision on what Q-value to predict.

At the end of this episode, the AI gets stuck in a loop, shown in *(e)*. The AI alternates between the left and right position, continually hitting the ball along the red marked path. A heartbeat-like pattern is shown in the Q-values graph. This occurs because the AI hasn't learnt to avoid this situation. This may be simply because the AI hasn't had enough training to avoid this yet. This behaviour might also be avoided by using RNNs [18].

4.1.3 Space Invaders

In Space Invaders the player controls a shooter at the bottom of the screen which can fire at descending aliens. When they are hit the player scores points. An example



(a) *(b)*

Figure 4.5: Sample Space Invaders frame with action Q-values

frame is shown in Figure 4.5a. It shows a frame where the shooter is just about to be hit by an incoming projectile. The player must move to the left to get out of the way to avoid losing a life. In Figure 4.5b the LEFT and LEFTFIRE have the highest Q-values, showing that the AI has learnt this behaviour.

4.1.4 Atlantis

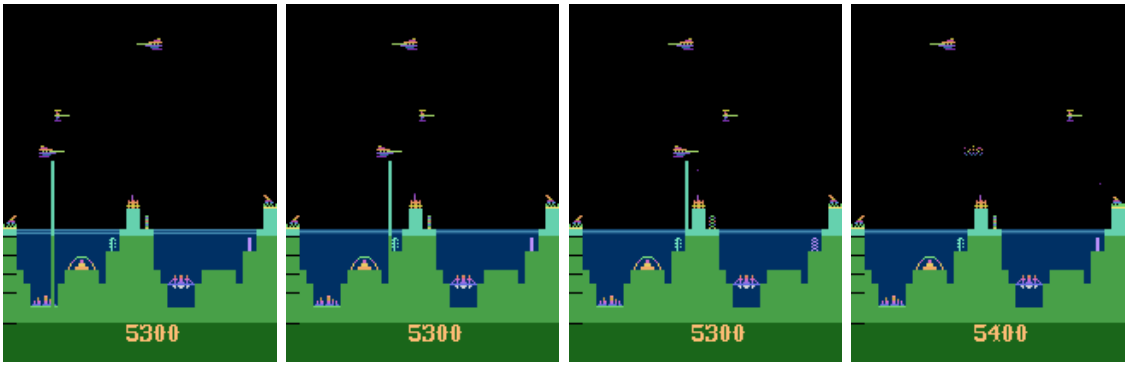
In this game the player defends against a wave of enemy spaceships that criss cross the screen and slowly descend towards the titular Atlantis. Once the ships are on the final layer, they can drop bombs that destroy the player's guns. There are three different guns at the start of the game, left, right and centre, corresponding to LEFTFIRE, RIGHTFIRE, and FIRE. Each gun has a cooldown period after firing, so it is sometimes advantageous to not fire at some points, so that the player can fire later and hit a ship, as shown in Figure 4.6. The NOOP action is taken for steps 561-563, until at step 564 a bullet is fired. The small bullet can be seen in the next step,ss 565, and more clearly in 566.

Figure 4.7a is taken just after a spaceship destroys the centre gun, at step 610. This point is shown in Figure 4.7b to be one of the lowest Q-values reached by the AI. A penalty is not given by the game – the AI doesn't lose any score – however it has learnt that this is a bad state in and of itself. It has learnt that, generally, fewer rewards are earned in states where the centre gun is lost.

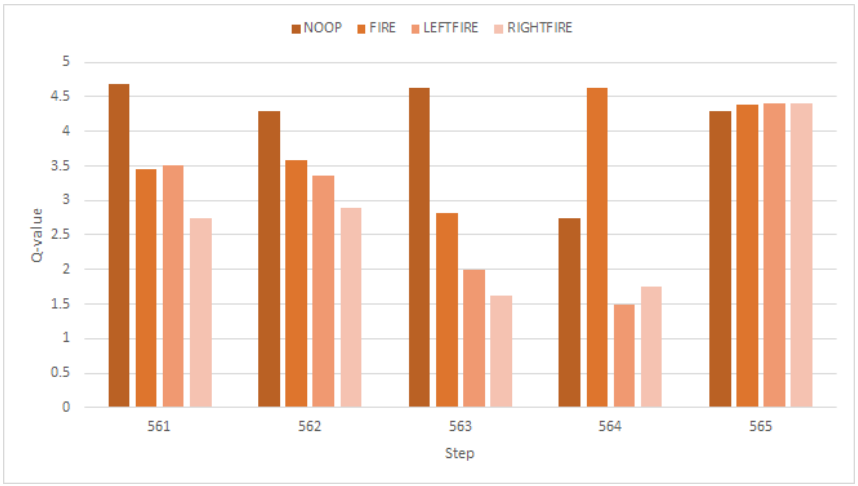
4.1.5 River Raid

In River Raid, the player controls a plane that flies along a river and blows objects up. If the plane hits the side of the bank, the player loses a life. River Raid is an example of a game with a high-dimensional action space: there are 18 discrete actions that the agent can take in any given state. This increases the time taken to train the agent.

Figure 4.8 shows a sample frame, and the Q-values of each action in this frame. The agent has learnt that actions including RIGHT (highlighted in blue) in this situation are bad actions, as they cause the ship to crash into the side of the bank,

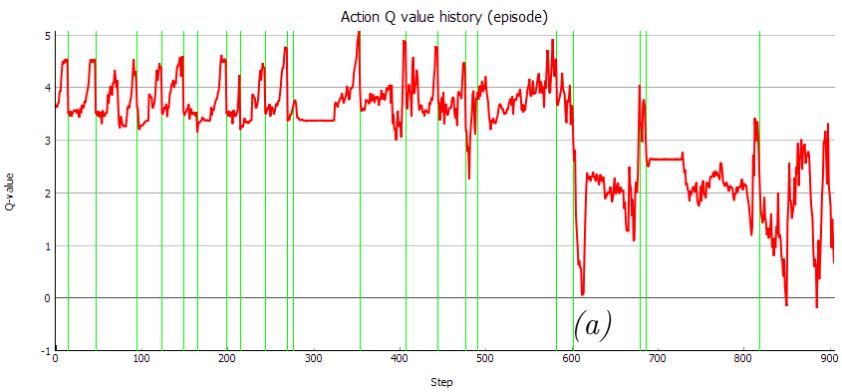


(a) Step 561 *(b)* Step 565 *(c)* Step 566 *(d)* Step 569: Ship destroyed



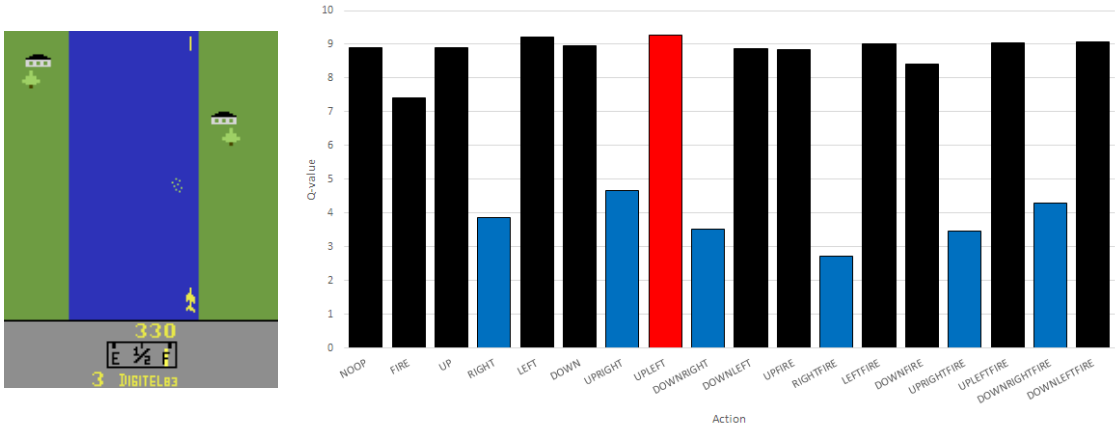
(e) Q-values for all actions for steps 561-565

Figure 4.6: Relevant sequence from Atlantis



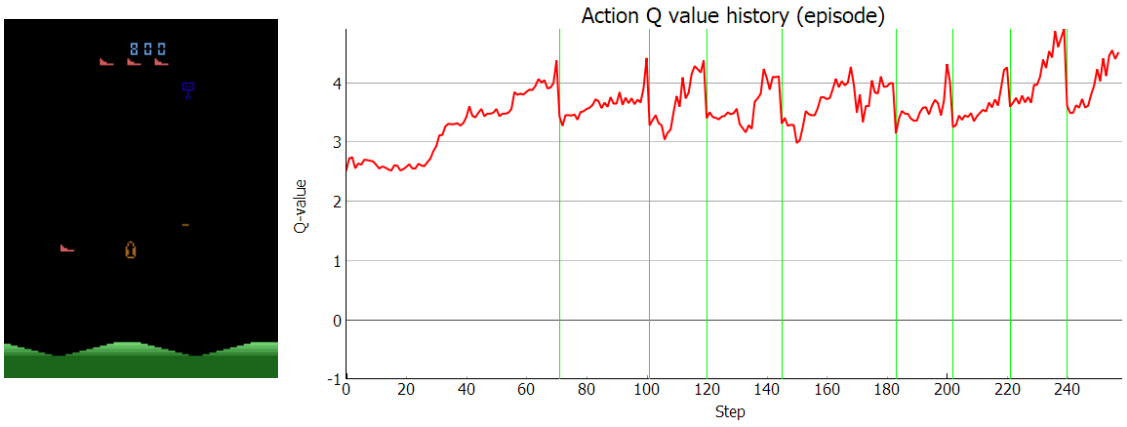
(a) Step 610 *(b)* Maximum Q-value history

Figure 4.7: An episode of Atlantis



(a) *(b)*

Figure 4.8: Sample River Raid frame with action Q-values. Actions highlighted in blue move right.



(a) *(b)*

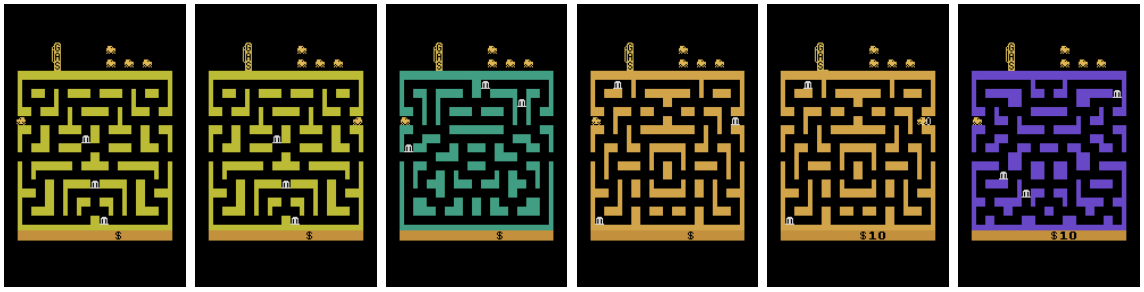
Figure 4.9: Sample Star Gunner frame with maximum Q-value history

and cause the player to lose a life. In this situation the agent chooses to take the UPLEFT action, away from the bank.

4.1.6 Star Gunner

Star Gunner, like River Raid, is a game with a high-dimensional action space, since the agent can fly all around the screen. The aim, as with many Atari games, is to shoot the enemies that appear on screen, and dodge the beams that are shot at it.

Shown in Figure 4.9 is a sample frame, and a graph of the maximum Q-value at every step in the episode up until that point. Peaks of the Q-value can be seen just before each reward, showing that the agent has learnt to recognise the increase in expected reward. When playing, however, the agent has trouble recognising which direction it is facing. It has learnt to navigate up and down to be on the same level as enemies, but not to horizontally position it's craft so that it faces them. This may be because it has not had enough training to learn that it should do this. It still eventually shoots and hits the enemies, since it is has a roughly 50-50 chance



(a) Level 1 *(b)* Level 1 *(c)* Level 2 *(d)* Level 3 *(e)* Level 3 *(f)* Level 4

Figure 4.10: Sequence of frames from Bank Heist

of being in either direction. This shows that sometimes the agent will not learn the optimal strategy, because it has already found a "good enough" strategy.

4.1.7 Bank Heist

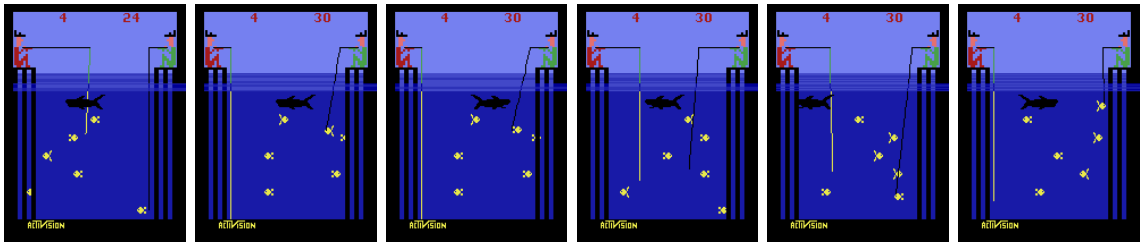
In Bank Heist the player controls a getaway car that is used to rob banks, as the name suggests. When robbed, the bank turns into a police car. Three of the four passages on the side of the screen are wrap around passages, allowing the player to get to the other side of the screen (Figures 4.10a,b). The upper right passage, however, goes to the next level (Figures 4.10b,c). There are fours different levels, and each time they are visited the banks respawn in the same place as before. The third level is interesting as one of it's banks is located near the upper right passageway (Figure 4.10d). The levels loop around, i.e. the next level after the fourth is the first, although the difficulty is increased. The time between robbing the bank and a police car spawning is reduced, and the speed of the police cars is increased. Eventually it becomes impossible to win, as the police car immediately spawns when the player robs a bank, making the player lose a life.

Hence, the optimal strategy is to gain all the reward from the current level before progressing to the next level. However, the agent doesn't learn this long-term policy. It will loop through this sequence (shown in Figure 4.10) again and again, eventually losing due to crashing into a police car that spawns immediately after a robbery. This shows the problem with the deep Q-learning algorithm - again it eschews the optimal policy for a "good enough" policy. This is the problem of exploration vs exploitation.

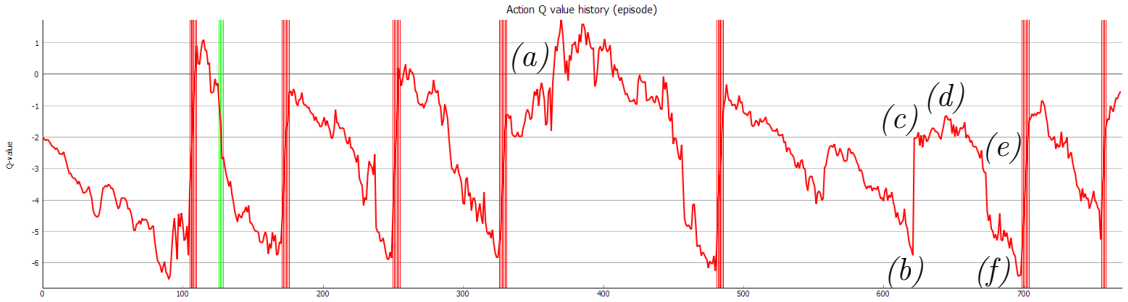
4.1.8 Fishing Derby

In Fishing Derby the player (on the left) is vs. the built-in game AI (on the right). The aim of the game is to catch as many fish as possible without the shark eating them on the way up.

Figure 4.11 shows a typical game of Fishing Derby, along with the maximum Q-values for each step in the episode. It performs poorly, losing to the built-in AI on average. However, in Google DeepMind's paper, a human doesn't do too well either, only achieving a score difference of 5.5. Although it is bad at gaining rewards, it can be shown that it has learnt to at least detect when it will lose reward, or in other words, when the built-in game AI will gain points.

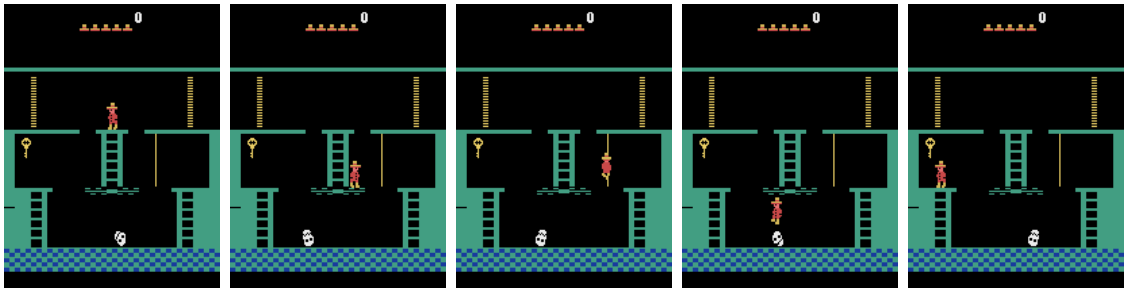


(a) Step 370 *(b)* Step 621 *(c)* Step 622 *(d)* Step 660 *(e)* Step 678 *(f)* Step 698



(g) Maximum Q-value history

Figure 4.11: A typical episode of Fishing Derby



(a) *(b)* *(c)* *(d)* *(e)*

Figure 4.12: Sequence of frames from Montezuma's Revenge

In this episode the agent gains some points early on, but then fails to achieve any more than that. Around step 370 *(a)*, the agent gets hopeful, and gets close to catching a fish, but fails to do so. The more interesting sequence is *(b-c)*. There is a large, sudden increase in Q-values from step 621 *(b)* to step 622 *(c)*. The only difference between these two frames is the shark's direction, showing that the agent manages to pick up on the fact that the opponent's fish is more likely to be eaten by the shark in this situation.

Then slowly the Q-value decreases *(d-f)* as the agent predicts the opponent's point scoring as more and more likely.

4.1.9 Montezuma's Revenge

Montezuma's revenge is a difficult game to play due to its sparse rewards. This means that the agent doesn't have an external reward signal to learn from, and so doesn't know which action to take. It continually chooses essentially random actions, since it cannot distinguish between them. This shows one of the main problems with reinforcement learning: more intelligent systems of exploration need

to be developed than a simply randomly taking actions until a reward is hit. As the number of actions needed to gain a reward increases, the probability of success decreases exponentially.

Figure 4.12 demonstrates the problem. The game starts at *(a)*. The aim is to get the key on the left. If the player just moves to the left or right though, they fall and die. They must move down the ladder instead *(b)*. Then, they must move against the left-moving conveyor belt and walk the right, jump on a rope *(c)*, jump again to the right, climb down the ladder, jump over the skull *(d)*, climb back up the ladder *(e)* and jump to pick up the key. At any point if the player doesn't complete this correctly, they lose a life. This is such a long chain of events that it is extremely unlikely to be happened upon by random exploration.

Some systems compensate for this by generating intrinsic rewards for states, to guide the agent in the "right" direction. In [21] *curiosity* is rewarded, i.e. agents are rewarded if their actions have unpredictable consequences. They test their method in a sparsely rewarded 3D maze, where the only reward is reaching the end of the maze.

One other potential method is by rewarding *empowerment*, i.e. rewarding states that have higher future potential states [22]. That is outside the scope of this thesis, however, due to time constraints.

4.2 Agent comparison

In this section the overall results of the implementation are presented and analysed.

A series of sets of deep Q-networks was developed for this thesis, each set with different hyperparameters. Table 4.1 below describes the series.

Name	**Results Table**	**Description**
D_1	Table 4.2	Uses default hyperparameters in Table 3.1.
D_2	Table 4.3	$\varepsilon_T = 0.1$
D_3	Table 4.4	$H_s = 400{,}000$, $\alpha = 0.00025$
D_4	Table 4.5	$H_s = 200{,}000$, $\alpha = 0.00025$

Table 4.1: Summary of the sets of trained DQNs.

Every 250,000 training steps, each agent was run for 150,000 testing steps with $\varepsilon = 0.01$, and the average episode score was calculated. The best testing performance was then used to give an overall score achieved for each game. This score is recorded in the results tables referenced above in Table 4.1.

For each game a random agent (agents with $\varepsilon = 1$) was run for 150,000 testing steps and the average episode score was recorded. The human results were sourced from [5]. The methodology of sourcing these human results are explained in [5].

A normalized score is calculated from each average episode score by using the random agent's score as a baseline from which to measure relative to the human's score, defined in Equation (4.1).

$$\frac{\text{DQN score} - \text{random agent score}}{\text{human score} - \text{random agent score}} \tag{4.1}$$

DeepMind defines an agent with a normalized score of $\geq 75\%$ as having a comparable skill to a human for that game, and a normalized score of $> 100\%$ means the agent has achieved superhuman performance. Conversely, a negative result implies that the agent has achieved worse performance than simply random play.

D_1 achieves superhuman performance for 18 out of the 49 games (Table 4.2), a worse result than in [5] – in that paper agents managed to achieve superhuman performance in 23 games. This could potentially be because the DQNs in this thesis have not been trained for as long as in [5]. In that paper they are all trained for 50,000,000 steps, whereas the longest a model has been trained in this thesis is 7,000,000 steps, due to time constraints.

In Figure 4.15 the average episode rewards for each game during the training of D_1 are shown, and many of them are on an upward trajectory. For example – Alien, Amidar, Assault, Asterix, Atlantis, Bank Heist, Beam Rider, Boxing, Breakout, Chopper Command, Demon Attack, Enduro, Fishing Derby (albeit slowly), Gopher, Kangaroo, Ms. Pacman, Name This Game, River Raid, Road Runner, Robotank, Space Invaders, Star Gunner, and Tutankham. 23 games, with 12 of those being of below human performance.

However, it could also be because the values of the hyperparameters are not optimal. Three more sets of deep Q-networks were developed to test this hypothesis: D_2, D_3 and D_4. Table 4.1 shows what the hyperparameters for each were, and which table the results are located in.

4.2.1 Google DeepMind comparison

Table 4.6 contains a comparison of this thesis' DQN results to Google DeepMind's results. The best performing network D_i is selected for each game and compared to Google's DQN using a similar normalization method used for comparing DQN results to human results, defined in Equation 4.2.

$$\frac{\text{Thesis DQN score} - \text{random agent score}}{\text{Google DeepMind DQN score} - \text{random agent score}} \tag{4.2}$$

The main result from the table is that this thesis' DQN's outperformed the Google DeepMind's DQN in 16 different games, with much fewer training steps. With more training steps, potentially each score in each game could be improved upon.

The fairly uniform spread of thesis DQNs suggests that there is no optimal hyperparameter set for all games. Each agent complements each other. This suggests that evolutionary algorithms with multiple agents with differing hyperparameters will be more successful than a lone agent. Due to the high computational cost however, this was impractical to explore in this thesis. In [12] the authors explore this multi-agent learning idea in order to generate agents that can play a 3D capture the flag game based on Quake 3.

As shown in Figure 4.13, this overall method produces one more superhuman agent and one more human-level agent – D_3 on Atlantis, and D_4 on Up and Down.

Game	Random	Human	DQN Score	Normalized DQN Score (% Human)	Training Steps (1,000s)
Alien	203.6	6,875	1,284.2	16.2 %	6,250
Amidar	3.2	1,676	183.4	10.8 %	6,250
Assault	248.3	1,496	2,890.2	211.7 %	7,000
Asterix	229.9	8,503	2,492.8	27.4 %	7,000
Asteroids	833.2	13,157	801.9	-0.3 %	6,250
Atlantis	16,990.4	29,028	13,726.2	-27.1 %	6,750
Bank Heist	13.7	734.4	656.4	89.2 %	6,750
Battle Zone	2,720.3	37,800	18,838.7	45.9 %	6,500
Beam Rider	377.2	5,775	3,448.3	56.9 %	6,750
Bowling	23.7	154.8	25.5	1.4 %	6,500
Boxing	0.5	4.3	86.5	2,284.8 %	6,250
Breakout	1.4	31.8	123.7	401.8 %	6,250
Centipede	2,280.7	11,963	3,501.2	12.6 %	6,500
Chopper Command	819.4	9,882	1,827.1	11.1 %	6,500
Crazy Climber	6,832.1	35,411	80,418.4	257.5 %	7,000
Demon Attack	173.4	3,401	11,188.2	341.3 %	6,750
Double Dunk	-18.5	-15.5	-17.6	29.5 %	6,250
Enduro	0	309.6	1,114.7	360 %	6,250
Fishing Derby	-93.9	5.5	-79.9	14.1 %	6,000
Freeway	0	29.6	31.8	107.3 %	6,000
Frostbite	70.8	4,335	295.5	5.3 %	4,000
Gopher	310.1	2,321	2,825.1	125.1 %	4,000
Gravitar	218	2,672	170.9	-1.9 %	4,500
H.E.R.O	441.1	25,763	3,313.6	11.3 %	5,000
Ice Hockey	-10.2	0.9	-6.8	30.7 %	4,750
James Bond	30.5	406.7	451	111.8 %	4,500
Kangaroo	51.4	3,035	5,446.6	180.8 %	5,250
Krull	1,598.3	2,395	2,409.7	101.8 %	4,750
Kung-Fu Master	444.5	22,736	27,979.6	123.5 %	5,500
Montezuma's Revenge	0	4,367	0	0 %	4,250
Ms. Pacman	253.1	15,693	1,238.2	6.4 %	4,000
Name This Game	2,250.3	4,076	8,138.6	322.5 %	4,250
Pong	-20.3	9.3	16.5	124.3 %	4,250
Private Eye	34.5	69,571	100	0.1 %	4,250
Q*bert	161.1	13,455	754.7	4.5 %	4,000
River Raid	1,484.4	13,513	2,887.7	11.7 %	4,250
Road Runner	12.2	7,845	15,127.4	193 %	3,500
Robotank	2.2	11.9	14.4	125.8 %	3,500
Seaquest	76.7	20,182	1,507.9	7.1 %	4,250
Space Invaders	156.2	1,652	362.8	13.8 %	4,000
Star Gunner	646.6	10,250	2,039.5	14.5 %	4,250
Tennis	-23.9	-8.9	-22.2	10.8 %	3,250
Time Pilot	3,820.7	5,925	4,386.7	26.9 %	5,250
Tutankham	10.4	167.6	191.5	115.2 %	3,250
Up and Down	464.2	9,082	5,600.3	59.6 %	2,500
Venture	0	1,188	52.9	4.5 %	3,750
Video Pinball	16,582.3	17,298	25,517.9	1,248.5 %	2,750
Wizard of Wor	672.5	4,757	1,411.8	18.1 %	3,250
Zaxxon	2.4	9,173	3,310	36.1 %	2,000

Table 4.2: D_1 results. Trained with hyperparameters in Table 3.1. Random agents are agents with $\varepsilon = 1$. Human scores sourced from the Google DeepMind paper [5]. The penultimate column is the normalized score of the DQN, calculated using Equation (4.1), and expressed as a percentage. Negative results mean the DQN performed worse than a random agent.

Game	Random	Human	DQN Score	Normalized DQN Score (% Human)	Training Steps (1,000s)
Alien	203.6	6,875	593.2	5.8 %	3,500
Amidar	3.2	1,676	107.2	6.2 %	3,750
Assault	248.3	1,496	2,749.9	200.5 %	3,750
Asterix	229.9	8,503	1,820.3	19.2 %	3,500
Asteroids	833.2	13,157	788.9	-0.4 %	3,500
Atlantis	16,990.4	29,028	13,626.2	-27.9 %	2,500
Bank Heist	13.7	734.4	385.2	51.5 %	4,500
Battle Zone	2,720.3	37,800	27,177.8	69.7 %	2,250
Beam Rider	377.2	5,775	1,234.5	15.9 %	2,500
Bowling	23.7	154.8	16.6	-5.3 %	3,750
Boxing	0.5	4.3	87	2,295.9 %	2,250
Breakout	1.4	31.8	29.5	92.3 %	2,500
Centipede	2,280.7	11,963	3,279	10.3 %	4,000
Chopper Command	819.4	9,882	1,191.1	4.1 %	2,500
Crazy Climber	6,832.1	35,411	80,730.6	258.6 %	1,250
Demon Attack	173.4	3,401	4,198.8	124.7 %	2,500
Double Dunk	-18.5	-15.5	-16.5	67.6 %	3,250
Enduro	0	309.6	814.5	263.1 %	4,000
Fishing Derby	-93.9	5.5	-61.9	32.2 %	3,500
Freeway	0	29.6	27.5	92.9 %	2,250
Frostbite	70.8	4,335	240	4 %	3,500
Gopher	310.1	2,321	550.9	12 %	4,500
Gravitar	218	2,672	137.3	-3.3 %	2,250
H.E.R.O	441.1	25,763	2,530.1	8.2 %	3,750
Ice Hockey	-10.2	0.9	-9.4	7.6 %	4,250
James Bond	30.5	406.7	261.2	61.3 %	4,500
Kangaroo	51.4	3,035	2,586.5	85 %	3,750
Krull	1,598.3	2,395	1,948.9	44 %	3,250
Kung-Fu Master	444.5	22,736	21,166.7	93 %	3,500
Montezuma's Revenge	0	4,367	0	0 %	4,000
Ms. Pacman	253.1	15,693	1,160.6	5.9 %	3,750
Name This Game	2,250.3	4,076	5,950	202.6 %	3,750
Pong	-20.3	9.3	-7.2	44.3 %	2,500
Private Eye	34.5	69,571	0	-0 %	3,500
Q*bert	161.1	13,455	619.8	3.5 %	3,000
River Raid	1,484.4	13,513	2,364	7.3 %	2,250
Road Runner	12.2	7,845	16,339.1	208.4 %	3,500
Robotank	2.2	11.9	13.8	119.4 %	3,250
Seaquest	76.7	20,182	1,538.6	7.3 %	2,500
Space Invaders	156.2	1,652	457.7	20.2 %	3,750
Star Gunner	646.6	10,250	1,757.6	11.6 %	4,000
Tennis	-23.9	-8.9	-22.2	11.2 %	3,500
Time Pilot	3,820.7	5,925	1,915.2	-90.6 %	3,250
Tutankham	10.4	167.6	192.2	115.6 %	4,000
Up and Down	464.2	9,082	5,990.4	64.1 %	3,250
Venture	0	1,188	0	0 %	3,250
Video Pinball	16,582.3	17,298	19,110.9	353.3 %	2,250
Wizard of Wor	672.5	4,757	1,063.6	9.6 %	3,000
Zaxxon	2.4	9,173	3,332.3	36.3 %	3,750

Table 4.3: D_2 results. Trained with the hyperparameter $\varepsilon_T = 0.1$.

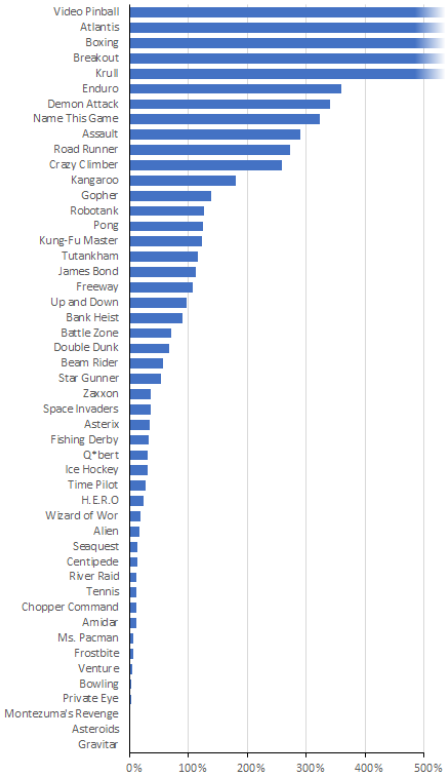


Figure 4.13: Overall normalized score comparison

Game	**Random**	**Human**	**DQN Score**	**Normalized DQN Score (% Human)**	**Training Steps (1,000s)**
Assault	248.3	1,496	3,859.4	289.4 %	4,000
Atlantis	16,990.4	29,028	3,038,100	25,097.3 %	4,750
Battle Zone	2,720.3	37,800	16,058.8	38 %	4,750
Beam Rider	377.2	5,775	3,194	52.2 %	4,750
Boxing	0.5	4.3	-22.5	-610.7 %	5,250
Breakout	1.4	31.8	310	1,013.8 %	4,750
Chopper Command	819.4	9,882	994.9	1.9 %	5,500
Crazy Climber	6,832.1	35,411	66,730.8	209.6 %	4,000
Demon Attack	173.4	3,401	947.4	24 %	5,250
Freeway	0	29.6	22.5	75.9 %	4,000
Gravitar	218	2,672	173.4	-1.8 %	3,500
Pong	-20.3	9.3	-20.9	-1.9 %	4,000
River Raid	1,484.4	13,513	1,116.1	-3.1 %	4,000
Seaquest	76.7	20,182	2,647	12.8 %	3,750
Space Invaders	156.2	1,652	694.7	36 %	2,500
Star Gunner	646.6	10,250	5,749.2	53.1 %	3,750
Video Pinball	16,582.3	17,298	228,552.9	29,617.8 %	3,500

Table 4.4: D_3 results. Trained with replay size $H_s = 400{,}000$ and learning rate $\alpha = 0.00025$.

Game	Random	Human	DQN Score	Normalized DQN Score (% Human)	Training Steps (1,000s)
Alien	203.6	6,875	100.7	-1.5 %	5,250
Amidar	3.2	1,676	14.4	0.7 %	4,750
Assault	248.3	1,496	3,789.8	283.8 %	4,500
Asterix	229.9	8,503	3,024	33.8 %	3,750
Asteroids	833.2	13,157	546.7	-2.3 %	1,750
Atlantis	16,990.4	29,028	2,022.4	-124.3 %	6,000
Bank Heist	13.7	734.4	396.3	53.1 %	3,750
Battle Zone	2,720.3	37,800	5,477.9	7.9 %	4,000
Beam Rider	377.2	5,775	1,493	20.7 %	4,500
Bowling	23.7	154.8	7.5	-12.3 %	4,750
Boxing	0.5	4.3	-22.5	-613.2 %	6,000
Breakout	1.4	31.8	3.2	6.2 %	6,000
Centipede	2,280.7	11,963	2,817.8	5.5 %	4,500
Chopper Command	819.4	9,882	606.8	-2.3 %	5,500
Crazy Climber	6,832.1	35,411	77,412.5	247 %	4,750
Demon Attack	173.4	3,401	917.2	23 %	6,000
Double Dunk	-18.5	-15.5	-21.1	-87.7 %	4,250
Enduro	0	309.6	24.7	8 %	6,500
Fishing Derby	-93.9	5.5	-84.1	9.9 %	4,500
Freeway	0	29.6	22.1	74.8 %	5,250
Frostbite	70.8	4,335	260.4	4.4 %	3,000
Gopher	310.1	2,321	3,081	137.8 %	5,500
Gravitar	218	2,672	1.1	-8.8 %	2,000
H.E.R.O	441.1	25,763	6,197.2	22.7 %	3,750
Ice Hockey	-10.2	0.9	-19.5	-83.7 %	3,500
James Bond	30.5	406.7	10	-5.5 %	5,750
Kangaroo	51.4	3,035	81	1 %	5,750
Krull	1,598.3	2,395	5,654.8	509.2 %	5,000
Kung-Fu Master	444.5	22,736	1.4	-2 %	5,500
Montezuma's Revenge	0	4,367	0	0 %	750
Ms. Pacman	253.1	15,693	1,243.7	6.4 %	4,500
Name This Game	2,250.3	4,076	3,797.3	84.7 %	5,000
Pong	-20.3	9.3	-21	-2.2 %	5,500
Private Eye	34.5	69,571	0	-0 %	1,750
Q*bert	161.1	13,455	4,239.7	30.7 %	4,250
River Raid	1,484.4	13,513	807.6	-5.6 %	4,250
Road Runner	12.2	7,845	21,433.3	273.5 %	5,000
Robotank	2.2	11.9	10.2	82.7 %	6,000
Seaquest	76.7	20,182	110.1	0.2 %	5,000
Space Invaders	156.2	1,652	653.6	33.3 %	4,750
Star Gunner	646.6	10,250	5,445.8	50 %	5,250
Tennis	-23.9	-8.9	-23.6	1.6 %	5,750
Time Pilot	3,820.7	5,925	435.6	-160.9 %	4,250
Tutankham	10.4	167.6	4.8	-3.6 %	6,000
Up and Down	464.2	9,082	8,728.9	95.9 %	4,000
Venture	0	1,188	0	0 %	4,500
Video Pinball	16,582.3	17,298	160,346.3	20,087.5 %	5,000
Wizard of Wor	672.5	4,757	133.3	-13.2 %	4,500
Zaxxon	2.4	9,173	0	-0 %	4,250

Table 4.5: D_4 results. rained with replay size $H_s = 200{,}000$, and learning rate $\alpha = 0.00025$.

Game	DQN	Google DeepMind DQN Score	Thesis DQN Score		(Diff.)	Normalized DQN Score (% Human)	Normalized DQN Score (% Google DeepMind DQN)
Alien	D_1	3,069	1,284.2	–	1,784.8	16.2 %	37.7 %
Amidar	D_1	739.5	183.4	–	556.1	10.8 %	24.5 %
Assault	D_3	3,359	3,859.4	+	500.4	289.4 %	116.1 %
Asterix	D_4	6,012	3,024	–	2,988	33.8 %	48.3 %
Asteroids	D_1	1,629	801.9	–	827.1	-0.3 %	-3.9 %
Atlantis	D_3	85,641	3,038,100	+	2,952,459	25,097.3 %	4,400.7 %
Bank Heist	D_1	429.7	656.4	+	226.7	89.2 %	154.5 %
Battle Zone	D_2	26,300	27,177.8	+	877.8	69.7 %	103.7 %
Beam Rider	D_1	6,846	3,448.3	–	3,397.7	56.9 %	47.5 %
Bowling	D_1	42.4	25.5	–	16.9	1.4 %	9.8 %
Boxing	D_2	71.8	87	+	15.2	2,295.9 %	121.3 %
Breakout	D_3	401.2	310	–	91.2	1,013.8 %	77.2 %
Centipede	D_1	8,309	3,501.2	–	4,807.8	12.6 %	20.2 %
Chopper Command	D_1	6,687	1,827.1	–	4,859.9	11.1 %	17.2 %
Crazy Climber	D_2	114,103	80,730.6	–	33,372.4	258.6 %	68.9 %
Demon Attack	D_1	9,711	11,188.2	+	1,477.2	341.3 %	115.5 %
Double Dunk	D_2	-18.1	-16.5	+	1.6	67.6 %	507.4 %
Enduro	D_1	301.8	1,114.7	+	812.9	360 %	369.4 %
Fishing Derby	D_2	-0.8	-61.9	–	61.1	32.2 %	34.3 %
Freeway	D_1	30.3	31.8	+	1.5	107.3 %	104.8 %
Frostbite	D_1	328.3	295.5	–	32.8	5.3 %	87.3 %
Gopher	D_4	8,520	3,081	–	5,439	137.8 %	33.8 %
Gravitar	D_3	306.7	173.4	–	133.3	-1.8 %	-50.3 %
H.E.R.O	D_4	19,950	6,197.2	–	13,752.8	22.7 %	29.5 %
Ice Hockey	D_1	-1.6	-6.8	–	5.2	30.7 %	39.7 %
James Bond	D_1	576.7	451	–	125.7	111.8 %	77 %
Kangaroo	D_1	6,740	5,446.6	–	1,293.4	180.8 %	80.7 %
Krull	D_4	3,805	5,654.8	+	1,849.8	509.2 %	183.8 %
Kung-Fu Master	D_1	23,270	27,979.6	+	4,709.6	123.5 %	120.6 %
Montezuma's Revenge	D_4	0	0	+	0	0 %	0 %
Ms. Pacman	D_4	2,311	1,243.7	–	1,067.3	6.4 %	48.1 %
Name This Game	D_1	7,257	8,138.6	+	881.6	322.5 %	117.6 %
Pong	D_1	18.9	16.5	–	2.4	124.3 %	93.9 %
Private Eye	D_1	1,788	100	–	1,688	0.1 %	3.7 %
Q*bert	D_4	10,596	4,239.7	–	6,356.3	30.7 %	39.1 %
River Raid	D_1	8,316	2,887.7	–	5,428.3	11.7 %	20.5 %
Road Runner	D_4	18,257	21,433.3	+	3,176.3	273.5 %	117.4 %
Robotank	D_1	51.6	14.4	–	37.2	125.8 %	24.7 %
Seaquest	D_3	5,286	2,647	–	2,639	12.8 %	49.3 %
Space Invaders	D_3	1,976	694.7	–	1,281.3	36 %	29.6 %
Star Gunner	D_3	57,997	5,749.2	–	52,247.8	53.1 %	8.9 %
Tennis	D_2	-2.5	-22.2	–	19.7	11.2 %	7.9 %
Time Pilot	D_1	5,947	4,386.7	–	1,560.3	26.9 %	26.6 %
Tutankham	D_2	186.7	192.2	+	5.5	115.6 %	103.1 %
Up and Down	D_4	8,456	8,728.9	+	272.9	95.9 %	103.4 %
Venture	D_1	380	52.9	–	327.1	4.5 %	13.9 %
Video Pinball	D_3	42,684	228,552.9	+	185,868.9	29,617.8 %	812.1 %
Wizard of Wor	D_1	3,393	1,411.8	–	1,981.2	18.1 %	27.2 %
Zaxxon	D_2	4,977	3,332.3	–	1,644.7	36.3 %	66.9 %

Table 4.6: Comparison of average episode scores achieved in this thesis with scores achieved by Google DeepMind [5]. For each game the best scoring thesis DQN (D_i) is compared against Google DeepMind's DQN. The comparison is calculated using Equation (4.2), and is expressed as a percentage. Negative results mean the DQN performed worse than a random agent.

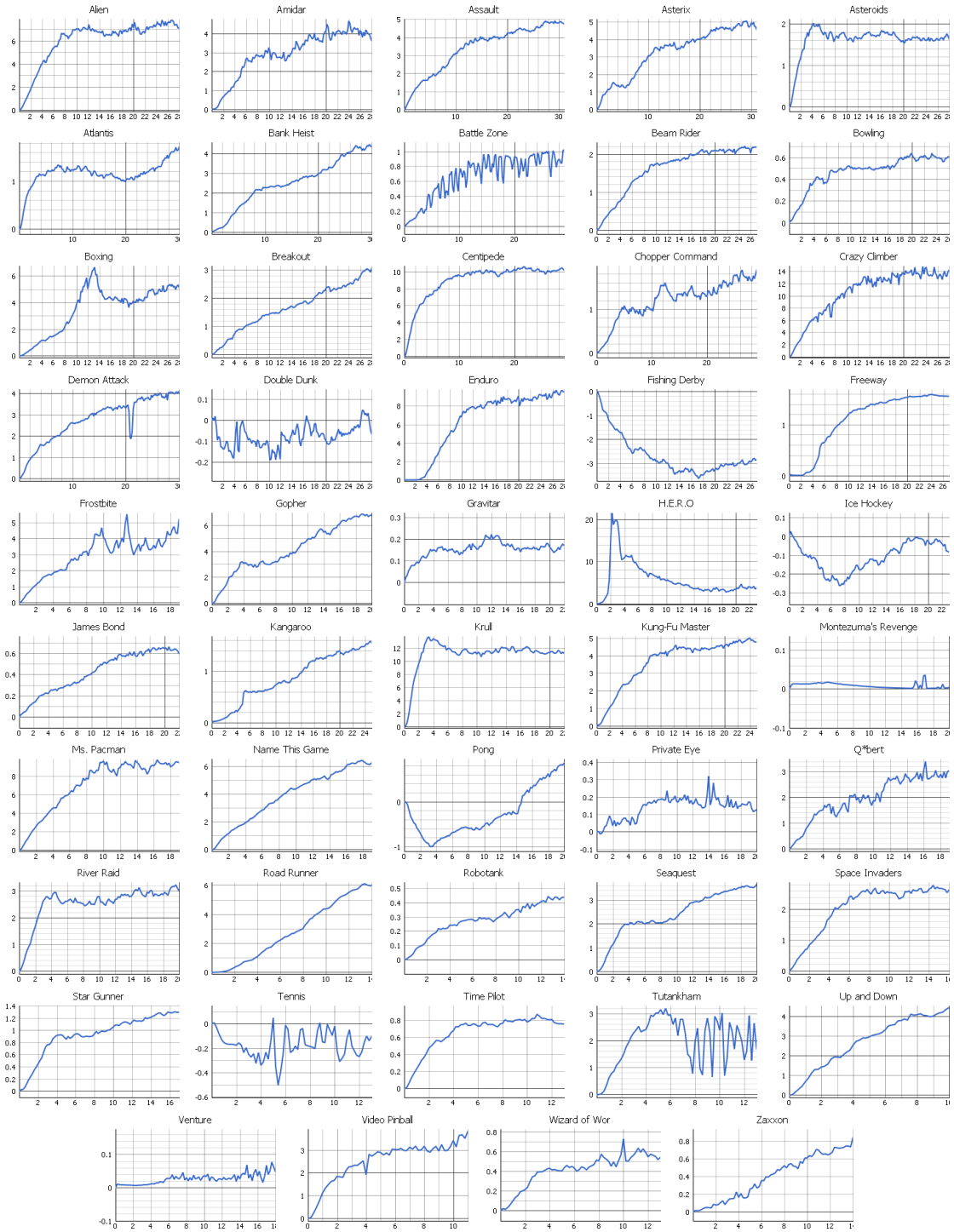


Figure 4.14: Average Q-values when training D_1 for all games. Epoch is on the X-axis, Q-values are on the Y-axis. At each point, the previous 50,000 Q-values are used as an average.

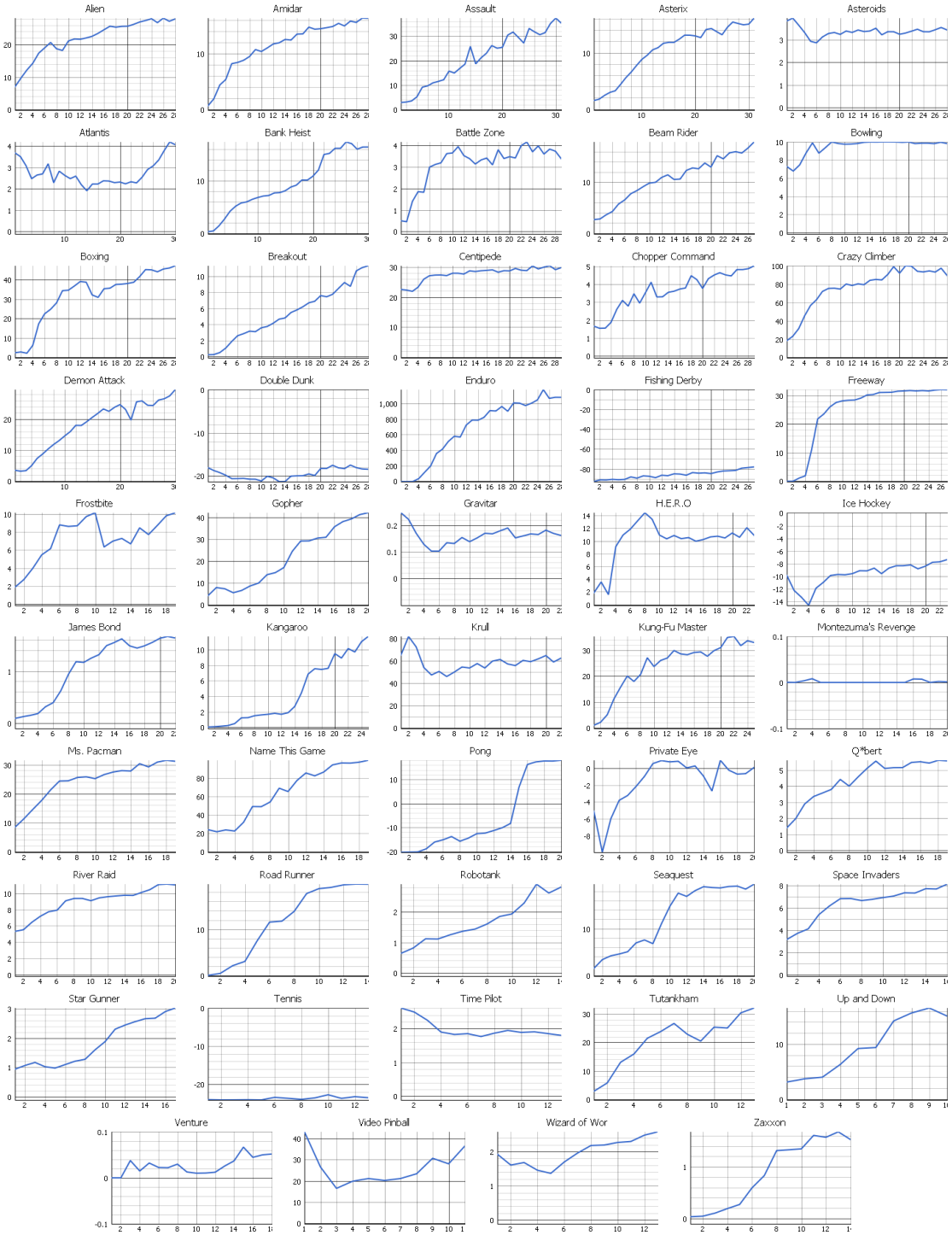


Figure 4.15: Average episode rewards when training D_1 for all games. Epoch is on the X-axis, clipped rewards are on the Y-axis (Section 2.4.4). Note that many of these graphs have a positive trend, showing that potentially higher scores could be achieved by continuing training.

Chapter 5

Conclusion and future work

This thesis has attempted to replicate two papers by Mnih et al [4, 5]. The aim was to replicate the results, but due to resource limitations a full replication of the paper could not be achieved. However, partial replication of the paper has been achieved. Even with limited resources success was achieved on a wide variety of games, even surpassing Google in some games.

There were three main differences that prevented it being a full replication of Google's paper. There was a different optimizer (Google used RMSProp, this thesis uses Adam), different replay size (100,000 vs 1,000,000), and a different number of steps. (7,000,000 max vs 50,000,000).

If there was more time to test and train, a future study would train deep Q-networks with a few different optimizers, and see the difference that makes to the scores. A grid search could also be performed on the learning rate hyperparameter to figure out the optimum value that can be achieved.

There are many potential directions from this to go in. Deep Q-learning forms a base from which there have been many different tweaks and branches off of. For example, IMPALA [20]. IMPALA is a scalable distributed algorithm for running deep Q-learning with. Many different computers run simulations of the environment (Atari 2600 or a different environment) and communicate their histories to a central computer that stores their transitions in it's internal replay buffer. This central computer occasionally performs SGD using this data, just how the regular deep Q-learning algorithm does, and sends the updated parameters of the policy network to it's clients. It has been shown to be much faster then single machine deep Q-learning, and also, strangely, more stable. This is because it has a larger pool of observations to draw from, and so has a more *i.i.d* dataset to train from (Section 2.4).

RNNs could also be investigated, as since they have a memory they could solve more interesting games that require memory [18].

List of Figures

2.1 Reinforcement learning overview . . . 3
2.2 Example neural network . . . 4
2.3 Example node . . . 6
2.4 Loss function landscape . . . 7
2.5 Sobel kernel (horizontal) convolved with an image . . . 8
2.6 Frame processing . . . 10
2.7 Max frame processing . . . 11

3.1 Screenshot of the Google Cloud Storage web interface . . . 15
3.2 Visualizer . . . 18

4.1 Sample Pong frame with action Q-values . . . 21
4.2 A typical game of Pong . . . 21
4.3 Sample Breakout frame with action Q-values . . . 22
4.4 A typical game of Breakout . . . 23
4.5 Sample Space Invaders frame with action Q-values . . . 24
4.6 Relevant sequence from Atlantis . . . 25
4.7 An episode of Atlantis . . . 25
4.8 Sample River Raid frame with action Q-values . . . 26
4.9 Sample Star Gunner frame with maximum Q-value history . . . 26
4.10 Sequence of frames from Bank Heist . . . 27
4.11 A typical episode of Fishing Derby . . . 28
4.12 Sequence of frames from Montezuma's Revenge . . . 28
4.13 Overall normalized score comparison . . . 33
4.14 Average Q-values when training D_1 for all games . . . 36
4.15 Average episode rewards when training D_1 for all games . . . 37

List of Tables

2.1 Glossary 13

3.1 Hyperparameters 16
3.2 Neural network architecture 17
3.3 Visualizer controls 19

4.1 Summary of the sets of trained DQNs. 29
4.2 D_1 DQN results 31
4.3 D_2 DQN results 32
4.4 D_3 DQN results 33
4.5 D_4 DQN results 34
4.6 Combined thesis DQN vs. Google DeepMind DQN score comparison 35

List of Algorithms

1 Q-learning 3
2 Deep Q-learning 12

Bibliography

[1] Devin Monnens. "I commenced an examination of a game called 'tit-tat-to'": Charles Babbage and the "first" computer game. In *DiGRA Conference*, 2013.

[2] Alan M Turing. Chess. In B. V. Bowden, editor, *Faster Than Thought*, chapter 25, pages 286–295. Sir Isaac Pitman & Sons, London, 1953.

[3] Claude E Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(7):256–275, 1950.

[4] 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.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. URL `https://dx.doi.org/10.1038/nature14236`.

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

[7] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013.

[8] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

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

[10] Bernard Widrow and Michael A Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9): 1415–1442, 1990.

[11] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian

Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017. URL http://dx.doi.org/10.1038/nature24270.

[12] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *arXiv preprint arXiv:1807.01281*, 2018. Blog: https://deepmind.com/blog/capture-the-flag/.

[13] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research, 2018.

[14] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, abs/1710.10196, 2017. URL http://arxiv.org/abs/1710.10196.

[15] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[18] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015. URL http://arxiv.org/abs/1507.06527.

[19] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. *arXiv preprint arXiv:1809.04474*, 2018. Blog: https://deepmind.com/blog/preserving-outputs-precisely-while-adaptively-rescaling-targets/.

[20] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. IMPALA: Scalable distributed Deep-RL with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018. Blog: https://deepmind.com/blog/impala-scalable-distributed-deeprl-dmlab-30/.

[21] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17, 2017.

[22] Shakir Mohamed and Danilo Jimenez Rezende. Variational information maximisation for intrinsically motivated reinforcement learning. In *Advances in neural information processing systems*, pages 2125–2133, 2015.

Appendix A

Zip Instructions

All code is located in the git repo below.
`https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2018/cxt510`

Inside the attached zip file there are a few Python script files, a requirements.txt file, a 'run_models.sh' script, and a folder 'model' that contains a set of preset models to use to play certain Atari 2600 games.

Before running the code, first you need to install the prerequisite Python libraries by running 'pip install -r requirements.txt'. pytorch may need to be installed separately.
Once this is complete, run the 'run_models.sh' script to start playing. Controls are below.

If on Windows, just copy and paste the commands out of the script file onto the command line to run them.

Key(s)	Description
`<escape>` or Q	Quit
F	Pause/resume
X	Step frame (when paused): Steps the environment forward one frame
C	Step agent (when paused): Steps the environment to the next DQN update. Different to stepping a frame due to action repeat (Section 2.4.1).
T	Double FPS (max 120)
G	Half FPS (min 5)
U	Toggle user-controlled mode
User Controlled Mode	
`<space>`	FIRE
`<left>`	LEFT
`<right>`	RIGHT
`<up>`	UP
`<down>`	DOWN