School of Computing
FACULTY OF ENGINEERING AND PHYSICAL SCIENCE

UNIVERSITY OF LEEDS

Final Report

Playing Atari Games using Deep Reinforcment Learning Techniques

Lewis Anthony Gordon Stuart

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2020/2021

40 credits

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Written report	PDF	Minerva (10/05/20)
Source code	GitHub repository: *https://github.com/Woebegonemite/atarai*	Public (10/05/20)

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(signature of student)

Summary

Deep Reinforcement Learning is a branch of machine learning that focuses on learning behaviours and has recently become a powerful framework for solving high-dimensional problem spaces. The Arcade Learning Environment (1995) was one of the first breakthroughs of Deep Reinforcement Learning that used a Deep Q-Network to learn how to play a series of different Atari games effectively. This project builds upon this foundation by proposing a model that utilises more modern reinforcement learning algorithms, such as Double Deep Q-Learning and REINFORCE, while also allowing user customisation of parameters.

Acknowledgements

Many thanks to my supervisor Dr. Abdulrahman Altahhan for guiding and supporting the development of my machine learning Agent. Also, thank you to my assessor Dr. David Head for providing insightful feedback on the project during our contacted hours. While we never spoke personally, I want to thank Dr. Matteo Leonetti for running the excellent Machine Learning module; a lot of the techniques that were utilised in this project were learned from that module. Finally, I must praise my colleague Josh Green and his brother Adam Green for providing the hardware that was used for optimising the training of my Agent.

Contents

Chapter 1

Introduction

1.1 Project Overview

Before the 20th century Deep Learning revolution, researchers had to spend large amounts of time developing custom artificial intelligence (AI) solutions to individual complex problems. Due to the influx in computing power, Deep Learning has provided advanced algorithms that can function in high-dimensional problem spaces to achieve an efficient solution, given the correct training and parameters [21, p.15]. Henceforth, solving seemingly impossible tasks has become almost trivial through utilising Deep Learning. At its core, this project involves initialising modern-day deep learning techniques to solve problems that could not have been solved as effectively in the past. As AI has become an increasingly important sector in computer science, it is important to utilise these techniques in old problem spaces to view how far AI has developed.

The Atari 2600 is a vintage gaming console released in 1978 consisting of a large variety of simplistic games that engaged and entertained the user [20]. At the time, the logic for the AI that were utilised for these games, such as enemy attack patterns, was hard coded using the rules of the game; no machine learning or more complex models were implemented. This project involves developing various models and solutions that utilise modern deep learning techniques for creating an AI that can effectively interact with Atari games, and ultimately achieve a 'winning state' in a series of different games. The AI accepts the same visual frame data as the user, and based on previous experiences with the environment, can produce a logical move from a learned optimal policy in response [4, p.75].

There are different techniques and paradigms for implementing a solution, with the most prominent being reinforcement learning. This project will explore the basic reinforcement learning algorithms and explore how each impacted the performance of the agent.

1.2 Project Aims and Objectives

1.2.1 Aims

The aim of the project is to develop a software package that can train an agent using deep reinforcement learning algorithms to successfully play various vintage Atari games, given the correct parameters for the model [3]. Various machine learning techniques will be researched and implemented, and then executed on different games on the Atari 2600 system [16].

Developing an efficient model is a complex process, hence the main goal is to not create bespoke solutions for each game, but rather employ machine learning practices that have the potential to

learn from any given game environment. Factors such as user interaction with the model should also be considered to allow customisation of parameters and settings.

Ideally, by the end of the project, a series of implementations of the model should be proposed with specified parameters that are shown to work successfully for a specific Atari game environment.

1.2.2 Objectives

- To develop a reinforcement learning model that can effectively interact and learn with a given game environment.
- Use various reinforcement learning algorithms and highlight the differences in training performance.
- Suggest several unique implementations for different games that achieve competent scores.
- Provide a basic interface to allow users to develop customised models.

1.3 Deliverables

Upon project completion, the following deliverables will be provided:

- Link to the source code repository, which contains a README file explaining all of the information for correct execution of the project code, as well as links to other project resources.
- Project report, given in the form of an electronic PDF.

Chapter 2

Background Research and Justification of Technologies

2.1 Atari Games

The Atari 2600 was the first popular video game console amassing a dominant 75% of home videogame system sales by 1981 [25, p. 17]. At the time the Atari console ran on limited hardware: consisting of 128 bytes of RAM and a CPU single-core clock speed of 1.19 MHz. Therefore, only 128 possible colours could be displayed on the console and a maximum resolution of 160 x 192 pixels could be achieved [26]. The importance of this comes down to how Atari games have become great candidates for evaluating modern-day machine learning techniques.

Restrictions in computational power meant games developed on the Atari had a simplistic environment, with most games having all information about the game state displayed to the user each frame; this is what makes Atari games Markovian [4]. Consequently, this means that each frame of the visual output can be taken as the input to a machine learning agent, and by using the limited number of pixels, the agent can learn to perform optimal moves for a given game state.

Overall, Atari game environments provide a useful resource for training agents to learn to master each game, with some implementations performing better than professional players [27]. Currently, 181 different machine learning datasets have been established for the Atari 2600 game range, all using select algorithms for optimising performance for each game [28]. Hence, Atari games will be the environments used for developing the project.

2.2 Machine Learning

Machine learning in essence is the technique and practice of imitating the two fundamental elements of intelligence: learning and adapting [9, p.4]. A model needs to be developed that can adapt to make an optimal action given a visual game state, and iteratively learn by improving on unsuccessful actions. The main challenge is that before starting each game, the model has no previous knowledge of the internal logic of the game; machine learning is utilised to adapt to the given environment and to maximise the score, hence learning from experience. The main paradigm that will be used for the development of the project is deep reinforcement learning: which combines the regression and classification of neural networks with the techniques from reinforcement learning.

2.2.1 Game state classification and action-value regression

One of the main challenges is associating respective rewards and actions with each game state. The AI needs to be given a state from the Atari environment and be able to approximate an optimal action to take which maximises the respective game score. There are two methods of solving this problem:

1. Regression- given a game state, a series of predicted values are returned for each action. The highest action-value pair represents the optimal move to take.
2. Classification- for each game state, a series of action probabilities are returned which predict the likelihood of each action being successful.

Machine learning has provided several different algorithms for solving this problem through both classification and regression, with the most superior method being neural networks.

2.2.1.1 Neural Networks

The neural network model is inspired by the architecture of the human brain, which consists of roughly 10^{11} neurons and 10^{15} synapses connecting each neuron. Excluding biological complexities, the mathematical equivalent was developed, defined as the McCulloch-Pits neuron [29].

Each neuron has a series of weighted inputs (features), with each weight defining the strength of that connection, and an adder function defining the total value of the neuron. This adder function is typically a dot product between the input value and its respective weight. Finally, a step-activation function concludes whether a neuron should 'fire' based on if the value of the neuron surpasses an activation threshold [3, lecture 4]- the most common is the Rectified Linear Activation function.

A set of neurons is divided into layers, with a single layer consisting of neurons that are not connected to each other. This defines a single-layered perceptron. However, multiple connected layers can be utilised which characterises a multi-layered perceptron (neural network).

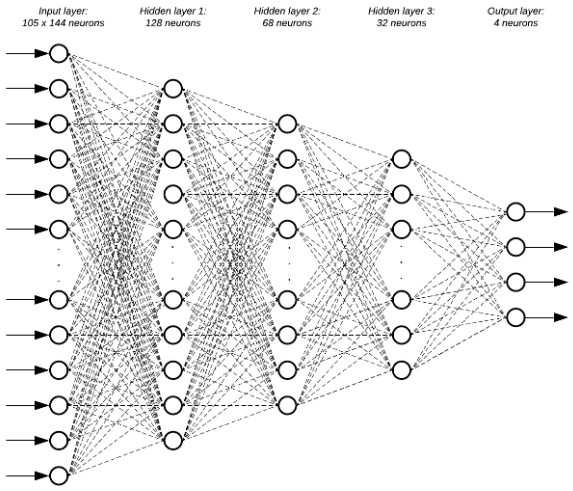


Figure 1: A Multi-layered perceptron- this is the first neural network that was used in the project. The inputs are the frame pixels, and the outputs are the available actions for the game Breakout.

The main benefit of the deep neural network architecture is that, depending on the weights, any value can be output depending on the respective input; neural networks are often described as function approximators [30]. Therefore, neural networks can be utilised to approximate optimal actions, given frame data of the game state. The number of layers and neurons is an important factor for developing successful neural network architectures. More than one layer is required for problems that are not linearly separable, but too many layers can lead to overfitting and cause

inefficient computational times. Hence, through using intuition [31], 3-4 layers will be suitable for a problem space such as Atari games.

2.2.1.2 Optimisation

When neural networks are first initialised, a random weighting is set for each neuron as no information about the current environment is known, hence a neural network must be optimised to correctly approximate the optimal moves. Optimisation can occur through training of the neural network. Every input must include a predicted outcome to effectively train the neural network. Improving the neural network is achieved through adjusting the weights between neurons in each layer to minimise the error; this process is achieved through backpropagation [41, p.197]. The difference between the expected and actual output defines an error model, which can be used to iteratively improve the neural network. Each error is modelled through loss functions, which defines the error value that will be used for improving the neural network. The goal to reach a point where the number of errors is zero.

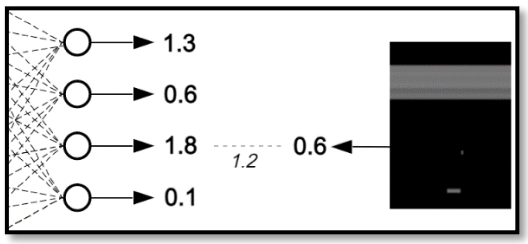


Figure 2: The predicted Q-Values from the neural network and the actual Q-Values from the breakout environment. The difference between the optimal action-value and the predicted value is the error.

The most common loss function is the mean-squared error. While the mean-squared has been shown to suffer from slower learning and saturation [41, p.226], it is still considered optimal for most regression problems [42, pg.155] and should work efficiently with the project scope. Another loss function which will be utilised is the cross-entropy (logarithmic loss), which estimates the probabilities in classification problems. Cross-entropy has been proven to be an efficient algorithm for improving performance with neural networks with sigmoid and softmax outputs [41, p.226], making it a suitable choice for the project. Both methods will be utilised separately for value-based and policy-gradient reinforcement learning algorithms, which will be discussed in section 2.2.

There still needs to be an optimiser which will update the weights of the neural network, and for that the Adam algorithm will be used. To briefly summarise, the Adam optimisation algorithm was published in 2015 and produces excellent results when used on non-convex optimisation problems. Since then, Adam has become the most popular algorithm in the field of deep learning due to achieving good results in fewer iterations than traditional stochastic gradient descent optimisation methods. Overall, Adam has since been defined as the ‘best overall choice’ [43, p.11] when considered with other optimisation algorithms such as Insofar, RMSprop and Adadelta; hence Adam will be the main optimiser used for the project.

2.2.1.3 Convoluted Neural Networks

For image classification problems, simply having the neural network input as the pixel values of the Atari game is inefficient for large problem sizes; many neurons will be required and ultimately finding patterns in corresponding inputs will be difficult. Convoluted Neural Networks (CNN) attempt to solve this issue through implementing layers that perform operations for compressing and extracting key information from the input image.

CNNs were inspired by the structure of the visual cortex [44] and are a branch of neural network which typically consist of three types of layers: convolution, pooling, and fully connected layers. Convolutional layers perform feature extraction through a kernel which traverses the image and condenses the pixel values into a feature value which produce a feature map. The kernel size, stride and padding all impact the filter's journey over the input image.

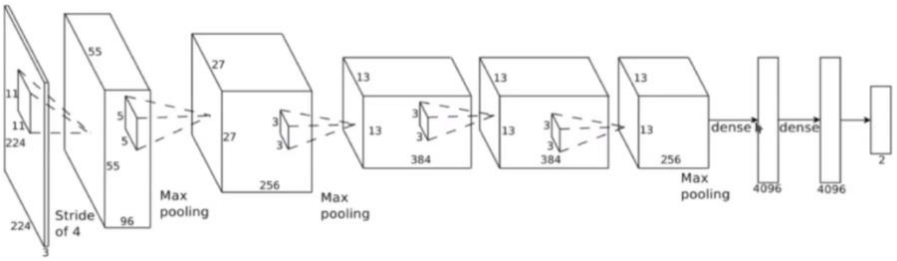


Figure 3: CNN Architecture for AlexNet [45]; uses a series of convoluted layers with two fully-connected layers. The kernel size can be shown for each convoluted layer, with a stride of 4 being utilised for the first layer.

The architecture of a CNN ultimately depends on the problem space; famous successful CNNs, such as AlexNet, can have over 500,00 neurons and many convolutional layers [45]. As Atari games have simplistic state representations, it is safe to consider that roughly three convolutional layers and one fully connected layer should be appropriate for all Atari environments.

Another potential solution involves Capsule Neural Networks, which add structures known as capsules to neural networks. The main benefit of Capsule networks is viewpoint invariance, which is the current main fault of the current CNN designs [46]. However, current support for Capsule networks is limited with software libraries and CNN should be powerful enough for the Atari environments being utilised.

2.2.1.4 Other Techniques

Currently, Neural Networks are the industry standard for classification and regression problems, however there are other machine learning methods that provide similar learning capabilities.

Forests of Decision Trees consist of a combination of different decision trees to classify new data points (states) based on a series of rules. The benefits of decision trees are that they are very fast to generate and have the unique ability to classify metric data [9, p. 263]. However, they are not suitable for working on Atari environments due to being too general and liable to overfitting.

K-Nearest Neighbours is a supervised algorithm where each data point is classified by viewing the k-closest classified points and assigns that class based on the labels assigned to k-closest data points. While K-nearest neighbours benefits from being simplistic and easy to use, it is unsuitable for Atari environments due to having slow predictions and sensitivity to irrelevant features, which neural networks do not suffer from [47].

Linear regression algorithms are another important machine learning technique that produce a hyperplane that classifies points on the dataset. The benefit of linear regression is that they are mathematically proven to work for a specific problem space, while neural networks do not have this guarantee as they iteratively improve overtime with no specified guarantee of success. However, linear regression often require many parameters and are too specific to learn a series of different environments, making them unsuitable for learning a range of Atari games.

2.2.2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm focused on behaviours. A 'trail-and-error' approach is taken where an agent, which is the entity interacting with the environment, tries many different combinations of actions to assess the validity through a returned reward [9, p 231]. Hence, the best actions are learned overtime. The main benefits of using RL for the project are:

1. Actions are time sensitive, meaning that decisions at a point in the environment episode (a playthrough of the game) will impact the choices made later in that episode. This means that long-term rewards and 'winning' states are realised by the agent given enough episodes to finalise an optimal policy.
2. The RL framework is generalised, meaning that a near perfect model can be developed to solve a particular problem given enough training and correct learning parameters, making it perfect for creating an agent that should perform well in a series of different Atari games.
3. Training is very similar to how humans and animals learn to solve problems, and since Atari games are designed to be played and mastered through similar human interaction, RL is a powerful framework for this problem space.

One of the main issues with using RL is that it is considered data-hungry, as a lot of data and computation is required. However, since Moore's law has continued to hold true, performing numerous episodes is not a significant issue. Furthermore, another issue is that RL assumes that the environment is Markovian, meaning that each game state can be viewed independently from one another; the agent only needs one state to extract all the information needed to make decision. For Atari environments this is not an issue as each game can be correctly articulated by the Markov Decision Process (MDP).

2.2.2.1 Markov Decision Process

MDP is a discrete-time stochastic control process that has been used as early as the 1950s and defines the principles for how an agent envisions a particular environment. The proposed model characterises an environment as a tuple of the following elements:

- **S**- Set of states which characterise the current environment.
- **A**- Current set of actions that an agent can take in response to a returned state.
- **r**- A reward for the relative action that the agent has taken.
- **T**- transition function that maps a state-action pair at time t to a series of states at time t_{+1}.

At every step that the agent takes in the environment, a state of the game is returned s_t, which for Atari games will be the current pixel values. Alongside this, a reward r_t will be returned for the previous action that the agent took, normally this will be the increase/decrease in the game score. The reward symbolises whether the action that the agent took was successfully. Using the current state, the agent will then take the next action a_t based on the current policy π. The available actions that the agent can take are dependent on the environment that is being learned from. Hence, a continuous perception-action loop is defined which characterises how the agent interacts with the environment.

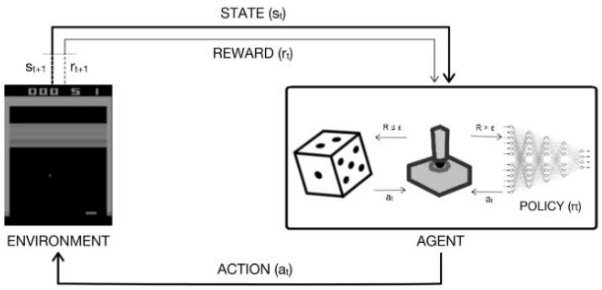


Figure 4: Perception-action loop taken by the agent. At time t, the agent accepts the current state s_t and reward r_t from the environment. A number $R \in \mathbb{R}$, $R \in [0,1]$ determines, based on an epsilon-greedy strategy, which action to take. Once this action is executed, a step in the environment is taken, and the next state s_{t+1} and reward r_{t+1} are returned. This diagram is based on the figure from reference [48, figure 2].

The purpose is to ultimately derive an optimal policy π^* that returns the maximum expected return for a given state; this is achieved through learning based on the return from a given action.

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}[R|\pi].$$

The return G_t is the accumulative discounted reward from the current state until the episode terminates. Each reward from the previous state is affected by a discount value γ, which determines how important future rewards are compared to the current state over a trajectory T. Discount values are important in determining whether the agent values short-term or long-term reward investments, as well as suspending an infinite cumulative reward for non-episodic MDPs (as non-episodic MDPs have no set end-point, the risk of an infinite sum of rewards is a prominent issue).

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$
$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

2.2.2.2 Value-Based Algorithms

Value-based methods use state-action-value functions (quality functions $\boldsymbol{Q^{\pi}}$) to calculate the expected return of a given state. The optimal policy is then defined, which takes the action with the maximum return value produced by the quality function, per respective state:

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) = \mathbb{E}[R|\mathbf{s}, \mathbf{a}, \pi].$$

Learning is performed through dynamic programming, which involves defining the function as a Bellman equation which has the following iterative form:

$$Q^{\pi}(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}_{\mathbf{s}_{t+1}}[r_{t+1} + \gamma Q^{\pi}(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1}))]$$

Simply put, the policy can be iteratively refined by taking the current estimated optimal action values, which estimate the return for taking each action, and comparing these to the return that was retrieved through interacting with the environment; this difference in actual return is compared to the predicted return form a temporal difference (TD) error which is used to adjust each estimation to improve the accuracy of the policy: this is why value-based approaches are considered regression problems.

An important factor is the learning rate $\boldsymbol{\alpha}$, which defines how much to adjust the policy based on the TD error. A learning rate which is too high will result in the policy never converging to an optimal solution, but a learning rate that is too low will never converge to an optimal solution [56]. Hence, using a defined learning rate, an update rule is defined as follows:

$$Q^{\pi}(s, a) \leftarrow Q^{\pi}(s, a) + \alpha(r + \gamma Q^{\pi}(s', a') - Q^{\pi}(s, a))$$

The quality function is adjusted by subtracting the error from the expected return, multiplied by the learning rate. Hence, through many experiences with the environment, each prediction is tuned to correctly anticipate the expected return of taking an action in a specific state.

Finally, there are two types of learning algorithm that use the update rule in contrasting ways: off-policy and on-policy approaches [55]. On-policy approaches use the current policy to improve the estimated return values through the experiences generated by the policy, with the most common algorithm being SARSA. While off-policy approaches improve through using a different policy for improving the estimated return values. Q-Learning is the most popular off-policy algorithm that uses the maximum estimated action value to improve the policy:

$$Q^*(s, a) \leftarrow Q^*(s, a) + \alpha(r + \gamma \max_{a'} Q^*(s', a') - Q^*(s, a))$$

Q-Learning will be the main value-based RL algorithm used for the project. Both methods have respective benefits, however SARSA and other on-policy algorithms cannot make use of replay memory used in Deep Q-Learning, therefore Q-Learning will be the algorithm of choice. Henceforth,

action values using Q-Learning will be referred to as Q-Values. For context, action-values for each state are contained within a data structure known as a Q-Table.

2.2.2.3 Policy-Gradient Algorithms

Policy-gradient methods do not maintain a value function model, but rather update the policy directly based on the return. A parameterised policy $\boldsymbol{\pi_\vartheta}$ is arbitrarily chosen and the parameters are optimised overtime to maximise the expected return. Optimisation can occur through gradient-free or gradient-based approaches, but gradient-based methods are usually the preferred method of choice due to being more sample efficient.

To compute an expected return, the average of the plausible trajectories induced by the policy needs to be calculated. For gradient-based learning, a Monte Carlo estimate of the expected return is determined. However, since gradients cannot pass through samples of a stochastic function, an estimator of the gradient is used, which is defined as the REINFORCE rule.

The REINFORCE rule can be used to compute the gradient of an expectation of a function $\boldsymbol{f}$, over a random variable $\boldsymbol{X}$ with respect to parameters $\boldsymbol{\vartheta}$:

$$\nabla_\theta \mathbb{E}_X[f(X;\theta)] = \mathbb{E}_X[f(X;\theta)\nabla_\theta \log p(X)]$$

This rule is the backbone for the REINFORCE policy gradient algorithm. After each episode, for each sampled action and state, the expected return $\boldsymbol{G_t}$ of each state over trajectory $\boldsymbol{T}$ is calculated, and the policy parameters are updated as follows:

$$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)$$

Many other policy-gradient algorithms exist [58], each with respective improvements to the basic REINFORCE algorithm, this will be the focus of the project due to its simplicity and effectiveness [59, p. 7].

2.2.2.4 Other Algorithms

Actor-critic methods combine technologies from policy-gradient and value-based algorithms. An ‘actor’ (policy) learns using the results from a ‘critic’ (value function). Actor-critic methods are very powerful because there is a guarantee of convergence, which methods like Q-Learning cannot provide. However, these will not be used as only basic algorithms will be utilised in the project.

Model-based RL is a separate method that involves defining a predictive model of an environment that an agent learns from. While model-based RL has been shown to provide better learning efficiency than model-free RL [53], due to the loss of generality between environments this approach will not be used.

2.2.2.5 Exploration and Exploitation

An important factor in learning is the concept of exploration and exploitation. An agent that explores too much will never converge to an optimal policy and will waste computation time exploring states

that return unsatisfactory rewards. However, over exploiting the policy could lead to situations where states that lead to better rewards are never explored, and thus the policy converges to an inefficient minimum.

As the agent has no information about the environment, the best action strategy to employ is to take non-optimal, random actions to return information about the environment. However, as the policy is improved, the agent should start to execute optimal actions more often. A simple ε-greedy strategy achieves this by choosing a random probability $R \in [0,1]$, and if $R > \varepsilon$ then a random action is taken, otherwise the optimal action is executed from the current policy. ε is slowly decreased each episode before the agent relies solely on the optimal policy. ε-greedy strategies have been proven to work effectively with almost any environment given the correct ε-decay rate.

Reward-based ε-greedy strategies have also been proposed [40] which base the exploitation rate on the agent's current highest return. The efficiency of this strategy ultimately depends on the environment and the types of rewards that are returned, but can be useful if utilised correctly. Other strategies exist, however ε-greedy is simplistic and efficient enough for this problem space.

2.2.2.6 Deep Reinforcement Learning

While RL methods offer powerful machine learning techniques for solving a specific problem, there is a significant deprecation in feasibility with high-dimensional problem spaces, such as Atari game environments. Given the game Breakout, which has an image size of 3x210x160 with 4 possible actions, and considering that each state of the game is contained within 128 bytes of Atari 2600 RAM, a Q-Table would consist of a table size of $4 \times 256^{3 \times 210 \times 160}$; this is both unfeasible and inefficient. However, with the rise of deep NNs, dynamic and achievable function approximation was made possible for high-dimensional problem spaces. Hence, deep reinforcement learning (DRL) uses deep NNs to approximate an optimal policy, which is trained through RL techniques. An NN in the context of Atari games will obtain the current pixel values, either through convolution or as standard representation, and output the most optimal action.

Value functions use a Deep Q-Network (DQN), which outputs the predicted Q-Values based on the current state. Using the current experiences gained through continuous playthrough, the agent will compare the current predicted Q-Values against the instanced Q-Values, which provides a loss that can be used to optimise the NN, and hence the policy. First implementations of DQN found that simply evaluating each episode undivided lead to unstable early Q-Values due to high correlation between actions and states; experience replay and target networks were introduced to resolve this issue [50].

Target networks are a replication of the current neural network parameters. However, the weightings are periodically synchronised with that of the main DQN and this used for returning the predicted Q-Values; the disconnection between the two networks provides improved training

stability. Replay memory stores current experiences and samples a batch of experiences arbitrarily when optimisation is required.

Double Deep Q-Networks (DDQN) expand upon the target network and use both the main policy network and the target network for returning the predicted Q-Values.

$$R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

It has been proven that utilising the target network in this fashion leads to remarkably better training results due to lower estimation errors [34].

Policy Gradient methods utilise neural networks through predicting probabilities of optimal actions. The architecture is very similar to that of a DQN. However the output values are converted into probabilities using a softmax function, which produces the likelihood of an action being optimal. Hence, an agent can choose the action with the largest probability to return the predicted optimal action for a state. Furthermore, unlike DQN or DDQN, policy-gradient methods do not require replay memory or a target network to provide stabilisation, as optimisation improves the policy directly rather than predicting values for each action. The optimisation of the policy is performed episodically, with each policy update being performed at the end of each episode.

Overall, DQN methods have taken dominance for most early DRL problems, with the first breakthrough being the Arcade Learning Environment (1995) which first successfully utilised Deep Q-Learning architectures to train an agent to successfully play a series of Atari games [51]; this is the foundation of the project. Policy-gradient methods are more modern and generally considered more concise than value-based approaches, mainly due to solving issues related to the exploration of the environment, as well as Q-Value instabilities [52]. However, fewer founding models have been developed for policy-gradient approaches, hence there will be more complexities when developing a framework for the project.

2.3 Software libraries and frameworks

2.3.1 PyTorch

PyTorch is one of the most popular open-source machine learning libraries currently available, providing a large variety of tools for solving machine learning tasks. PyTorch serves as a solid introduction to machine learning through its clear syntax, streamlined API, and easy debugging framework; this is especially useful as this project acts as an introduction to DRL [37]. Ultimately, the main benefit of using Pytorch is its *tensor*; a multidimensional array that provides complex mathematical properties optimised for machine learning practices.

Another open-source library that was considered is TensorFlow, a very similar machine learning library that provides almost identical functionalities to PyTorch. However, while TensorFlow is generally considered more computationally efficient [18] and has a larger community, PyTorch is much easier to learn while still providing the same functionalities [24]. Another method is creating

an agent only using NumPy arrays. While this would provide a deep insight into how machine learning libraries function, the complexities of creating a custom agent would take too much time and ultimately be inefficient.

2.3.2 Gym and Atari_py

For the environment interface, Gym will be used for training the agent. Gym is the most popular toolkit for developing and comparing reinforcement learning algorithms by providing a large variety of environments to run. The benefit of using Gym environments is that after each action that the agent takes, the current state (stored in an array of pixel values) and the reward for that action are returned; these are both key components in providing context for the agent. Furthermore, extra rewards such as the current agent lives are also returned which can be used to customise the reward passed to the agent. The Atari_py library must also be installed to provide the Atari ROMs for Gym to execute.

2.3.3 Other libraries

Other modules that are less important but will be required are:

1. MatPlotLib- creates graphs of the agent's progress, such as the reward for each episode.
2. Numpy- array manipulation and data handling.
3. Pandas- storing data frames of the agent's progress.

Each of these modules are obvious choices for the function they are satisfying, due to being simplistic in syntax as well as efficient.

Chapter 3

Project Planning

Due to the unique paradigm of machine learning, planning for a project such as this introduces unique challenges; a proposed model cannot be pre-determined whether it will be successful or not, only after training can evaluation on a given model be performed. However, standards defined by reinforcement learning will be followed that will lead to the greatest chance of success with a given model.

3.1 Project Methodology

The project will be split into two stages: initial working design and iterative improvement.

When the first iteration of the project is developed, a waterfall approach will be taken because of the importance and scale of the initial design phase. Before different custom models can be established for the different Atari games, an initial solution is needed to be developed that can perform competently in a simple environment; this requires in-depth planning to conform to the reinforcement learning schematic. Change-intolerant processes, such as Waterfall, ensure that a robust initial solution could be developed on time, with capabilities to be expanded to match the complexity of more complicated environments, such as Atari games.

Once an initial working solution is established, the modifications that are required will be for adding minor optimisation features to the model and adjusting the model parameters for each Atari game environment to converge to a working solution. Hence, a more Agile approach was taken, as iterative approaches match with the training and optimisation cycle that is employed with developing efficient machine learning solutions for each Atari game.

3.2 Initial Plan

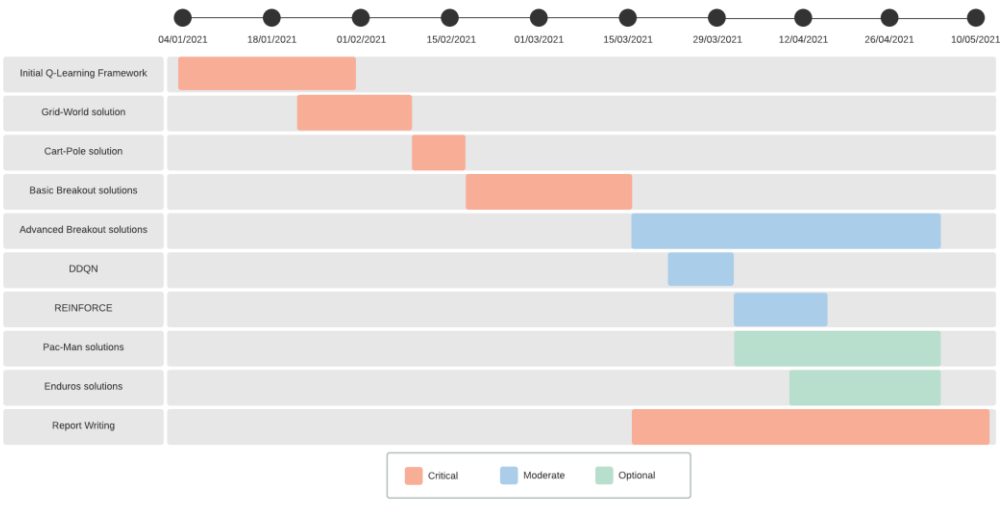


Figure 5: A Gantt chart of the project development, with respective time frames and severity scale

Background research and planning will occur before the start of the second university semester. However, research will still be performed throughout the life cycle of the project to support progress. Development of the project can be categorised into the following divisions:

1. **Initial framework** (*waterfall*)- An initial reinforcement learning framework will be established following the methods defined in the reinforcement learning schematics. The proposed model will be simplistic in design and focus on developing a working solution for a Grid-World environment, before proceeding to more complex solutions for a Cart-Pole environment. Q-Learning will be utilised as this is the most simplistic learning method and will help establish theorems needed for more complex architectures.
2. **Breakout solution** (*agile*)- By iteratively improving the proposed model, with more complex experience handling and utilising deep neural networks, a solution will be established for the Atari game Breakout [23]. Breakout will be the main environment of choice when developing potential models due to its simplistic nature. After a working solution has been established, parameters will be adjusted to improve model performance.
3. **DDQN & REINFORCE** (*agile*)- More complex policy approaches will be employed to further improve the previously established Breakout solution, and thus increase the complexity of the developed model.
4. **Pac-man & Enduros** (*agile*)- Once a complex solution has been developed for Breakout, other games will be considered to provide comparisons for the virtues and shortcomings of the current model. Both games differ significantly from Breakout in terms of gameplay, hence the flexibility of the model can be recognised and improved before completion.

It is important to note, developing a model for the game Breakout is the focus of the project; once a model has been established for one game, other solutions can be developed by altering the parameters and including more complex learning mechanisms. Developing a robust reinforcement learning model is the main objective of the project, not fully optimising for every type of Atari game.

3.3 Risk Mitigation Strategy

Due to the unpredictability of machine learning, potential risks that negatively impact the learning of the agent must be addressed early to avoid stunts in the project progress. Hence, the largest risk is unforeseen implementation bugs in the architecture of the model; these types of bugs are notorious for being difficult to locate and handle, as many different factors can lead to a model being unsuccessful. To avoid these types of issues, each change in the implementation will be followed by running the agent on default parameters to find if the model is still successful. If a failure occurs, then the latest modifications caused a hidden bug, which can then be located and patched.

Another large risk is time constraints. Training the agent to completion takes a considerable amount of time, hence there is the risk that a complex solution will not be developed before project submission due to training time constraints. While there is no solution for this issue, the amount of

time can be reduced by only training the agent partially to view if the current model shows any progress, as well as performing training on a GPU due to the parallelism speed up for neural network optimisation [19, p.15-16].

Finally, loss of project data can be avoided by performing rigorous and efficient version control, which will be achieved by utilising a Git repository.

3.4 Version Control

An important aspect of any large-scale computing project is implementing effective version control. As previously stated, version control mitigates any risk of data loss through valid commits and pushes to an online repository. Furthermore, during project development, commits will be made after every modification to the implementation of the project; if a modification causes the model to fail, reverting to a previous stable state will be a viable option. GitHub is the repository hosting platform of choice, due to the advantages of its Markdown language and collaborative features [22]. Once the project has been developed to a stable state, the repository will be made public allowing for cloning and inspection from different developers in the community.

Chapter 4

Software architecture

4.1 Overview

The purpose of this chapter is to provide insight into the available processes offered by the software package, and how each software component interface to provide an effective and customisable RL model. How each of the available processes were utilised for each Atari game will be discussed in section 5.

The reinforcement learning agent was developed in Python; many of the required modules were only developed for Python and the high-level features provided by the language were critical in the development of the agent. Other languages, such as C++ and Java, did not provide the required features for the project.

4.2 Structure

Each major element of the model has been assigned a class which is instanced with custom parameters defined by the user. Each of these classes provide methods that are called by the training loop when the agent begins learning and contains properties that ultimately impact how the learning process operates.

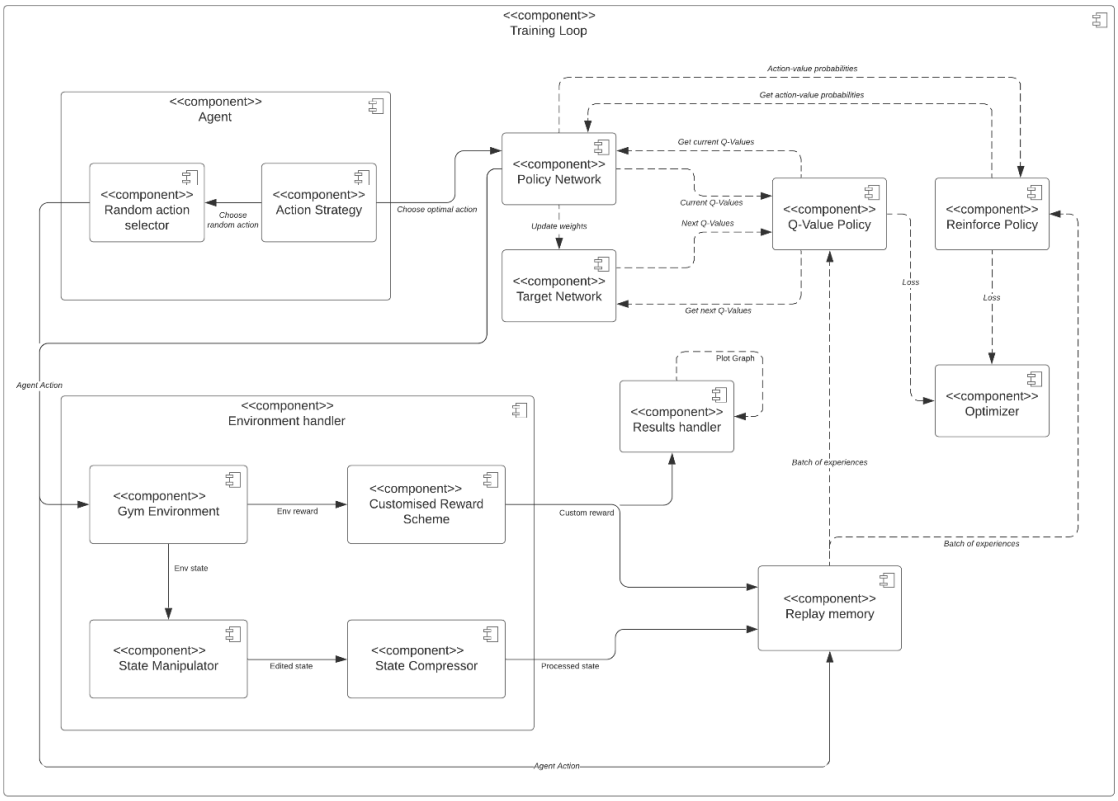


Figure 6: A Software Component diagram showing the basic relationships between each unit of the training loop. Solid lines represent data transfer after each step the agent takes, while dashed occur depending on the user defined parameters.

It is also significant to note that the main structure and logic of the source code is based on the Reinforcement Learning introduction provided by PyTorch [1] and inspired by the refined version shown on Deeplizard.com [2]; the original source code was designed for the Cart-Pole environment, but modifications have been made to make it suitable for learning Atari environments.

4.3 Environment Handler

Every time an action is made by the agent, the Gym environment returns the state and current reward from taking that action. The purpose of the environment handler is to act as the main interface between the Gym environment and the agent, manipulating returned states and reward to optimise the performance of the agent. How the states and rewards are returned are both dependent on the specific environment as well as the user defined parameters. For the Pac-Man environment, due to the complex nature of each game, extra state and reward manipulation was required (section 5).

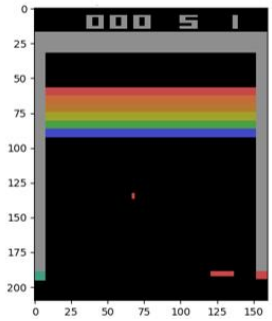
4.3.1 States

It was clear that returning the default pixel values from the environment to the agent is not efficient for optimised learning. Most of the state frame was not necessary to the agent and in some cases, not enough information is present in the default state. Hence a state queue was implemented that stores the previous states to combine with the current state to give the agent more context of the current environment. There are four different employed options for how the state is manipulated, each with advantages for different environments:

1. Standard- simply returns the current state of the environment; useful where all intrinsic information is visible in the returned state.
2. Difference- extracts the difference in pixel value between the current state and last state in the queue; practical for environments such as CartPole where small differences (movement) need to be the key focus that the agent learns to optimise.
3. Append- each state in the queue is returned as an individual tensor; utilised for environments such as Breakout, where previous states are required to infer the current direction of the ball.
4. Morph- each state in the state queue is combined into one tensor, with the previous state pixel values being discounted to give focus to the most recent state.

Before states are stored in the state queue, pre-processing ensures that only values required for learning are present in each state. Each state is cropped, and the colour scheme is set to either RGB or grayscale. For most environments, colour is not required; in Breakout the colours of the bricks are not important for learning to aim the ball to maximise the rewards. However, in some games such as Pacman, the colours are important in learning, as the colour of the enemy ghosts is a necessary input to avoid losing a life. Finally, each state is resized to a smaller tensor, this is to improve the

performance of the neural network by reducing input density. However, resizing the state excessively will result in intrinsic value being lost.



7.1: Normal Environment state.

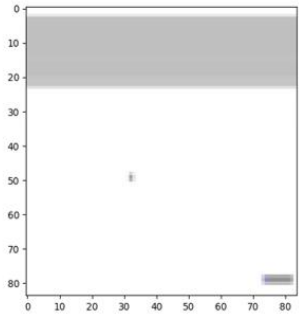


Figure 7.2: Standard processed state.

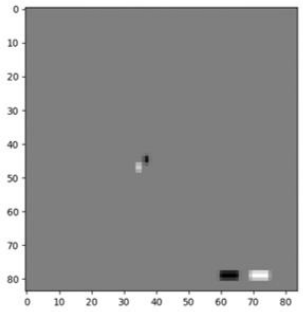


Figure 7.3: Difference state using 2 previous states.

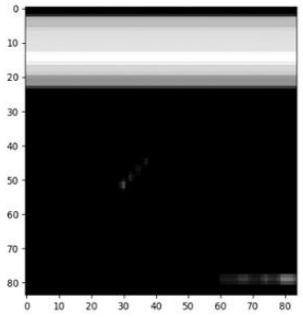


Figure 7.4: Morphed state using 4 previous states.

4.3.2 Reward

For context, the default reward returned from the environment directly relates to the current score of the environment. It is often required that extra reward shaping be utilised to optimise learning. The following options are available for reward customisation:

1. One life game- if the agent loses a life, then the environment handler returns that the current episode has finished.
2. Use given rewards- the rewards returned from the environment may not actually be beneficial for agent learning, hence these can be disabled.
3. Live change reward- depending on whether the agent gains or loses a life, a specific reward can be returned; this can be utilised for prioritising not losing a life over getting a high in-game score.
4. Normalise rewards- always returns a score of one for each reward returned by the Gym environment, meaning high-value rewards are valued the same as low-value rewards.

These available methods provide high customisation to optimise the agent's performance by highlighting actions that the agent should take. Fundamentally, which options to exercise depends on the given environment, this is covered more in each implementation (section 5).

4.4 Agent

A very simple design was employed for the agent. The purpose of the agent is to choose a given action based on the current state and action strategy that is being employed. The action strategy determines which action the agent should take, and for value-based algorithms an epsilon-greedy strategy is employed.

If a random action is chosen by this action strategy, then an arbitrary action is chosen from the available actions which are determined by the current environment. However, if the action strategy

dictates that an optimal action be performed, then the neural network is queried and the action presenting the highest Q-Value are returned.

However, if Policy-Gradient methods are utilised, the actions are returned based on the current probabilities of each action returned by the neural network. A simple action strategy is employed where the likelihood of an action being taken is based on its returned probability. Therefore, actions with a high probability of success are more likely to be chosen than other actions. Hence, as more training is employed, actions leading to high rewards will be given preference by the agent.

4.5 Epsilon Greedy Strategies

4.5.1 Linear Epsilon Decay

Before learning begins, the agent has no knowledge of the current environment, hence the first action that the agent should take should be randomised. As the agent interacts more with the environment, the policy is optimised to predict better actions. Therefore, a basic epsilon strategy is employed where the percentage chance that the agent takes an exploratory action decreases linearly with a fixed decay value per episode, to an endpoint when the exploratory rate is fixed.

$$\epsilon = \begin{cases} start - (decay.episode), & \text{if } decay.episode > end \\ end, & \text{if } decay.episode <= end \end{cases}$$

After some testing, it was clear that choosing a fixed end value was inefficient; once the epsilon rate was fixed, it was either too high to reach new states in the environment or too low where new states were not correctly explored. A simple solution was to employ a simple mid-point solution where afterward a mid-point is reached, the epsilon rate decreases at a lower linear rate.

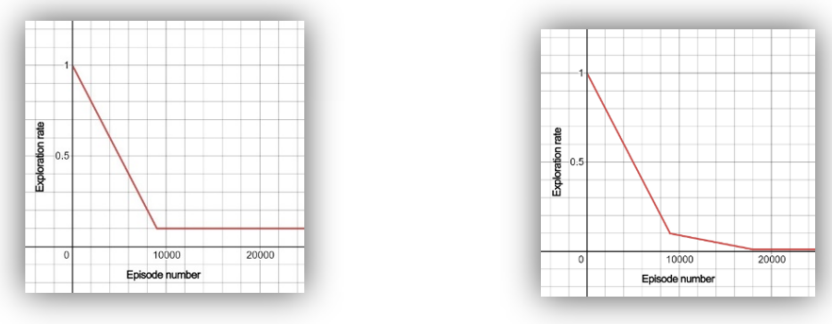


Figure 8: Two proposed Epsilon strategies. First shows Epsilon Linear Decay with parameters- {start: 1, decay: 0.001, end: 0.1}. Second shows Epsilon Linear Decay with two decay rates with parameters: {start: 1, middle: 0.1, end: 0.01, start decay: 0.001, end decay: 0.0001}

4.5.2 Reward-Based Epsilon Decay

Focusing on the reward is a powerful technique for deciding the moves that the agent should take, however it was clear that solely focusing on changes in the reward would not be a suitable option for some of the more complex environments offered by the Atari package. An example is in the game breakout; once the ball reaches the top of the board there is a massive increase in reward as the ball bounces between the top of the board and the bricks, lowering the exploration rate afterwards

would limit future learning as it is not confirmed that the agent performed this action through an optimal policy but rather could have 'gotten lucky' through exploration. However, that is not to say that reward influencing is not a viable action strategy, hence a proposed epsilon model uses both a linear decay over episodes as well as decay through high achieving rewards. This provides the stability of linear decay as well as employing less exploration given better agent performance.

4.6 Replay Memory

Every time the agent interacts with the environment it obtains experiences that can then be used for optimisation. Each experience contains all the information about the specific interaction and is modelled as follows:

$$experience_t =< state_t, action_t, state_{t+1}, reward_{t+1} >$$

The purpose of the replay memory is to store a sequential array of experiences that are extended with each transaction of the environment. Value-based methods require that experiences be returned in arbitrary batches; the reason for this is to avoid correlations between consecutive samples that result in unstable starting Q-Values and inefficient learning. However, Policy-Gradient algorithms do not suffer from this issue, and the purpose of the replay memory for these methods is to simply store and return all experiences serially when required.

An important part of the replay memory is the volume of experiences that are stored at one time, with new experiences overwriting existing indices in the array. Around 25,000 experience storage was considered optimal for learning for most environments, as this allowed enough information to be stored while also avoiding excessive memory usage. Furthermore, the batch size determines how many experiences are returned and is an important factor in agent learning; a balance must be struck between learning excessively from the same experiences in memory while also learning enough before new experiences are stored. Depending on the environment, between 20-50 experiences were considered appropriate for the batch size.

4.7 Neural Networks

There are several proposed NNs which can be exercised and each one allows for customisation of the available network properties, such as number of neurons. Each established NN has a different output depending on the policy method that was being utilised. DDQN and DDQ consisted of each output being related to the Q-Value for each respective action for the environment, while REINFORCE had probabilities output by a softmax function.

The first utilised NN consisted of a basic architecture with three linear layers. Ultimately, due to the lack of convoluted layers, using this policy for complex environments was inefficient. However, environments such as CartPole benefited from a direct translation from pixel values to action-values; this network was only used for establishing the original framework.

A later developed advanced CNN was established that consisted of three convoluted layers and one fully connected layer. While the number of neurons, kernel size, and strides can be altered for each layer, it was soon discovered that altering these for each environment did not impact the performance of the agent as the neural network was robust enough to adapt to all test Atari environments. The default parameters will be discussed more in section 5.

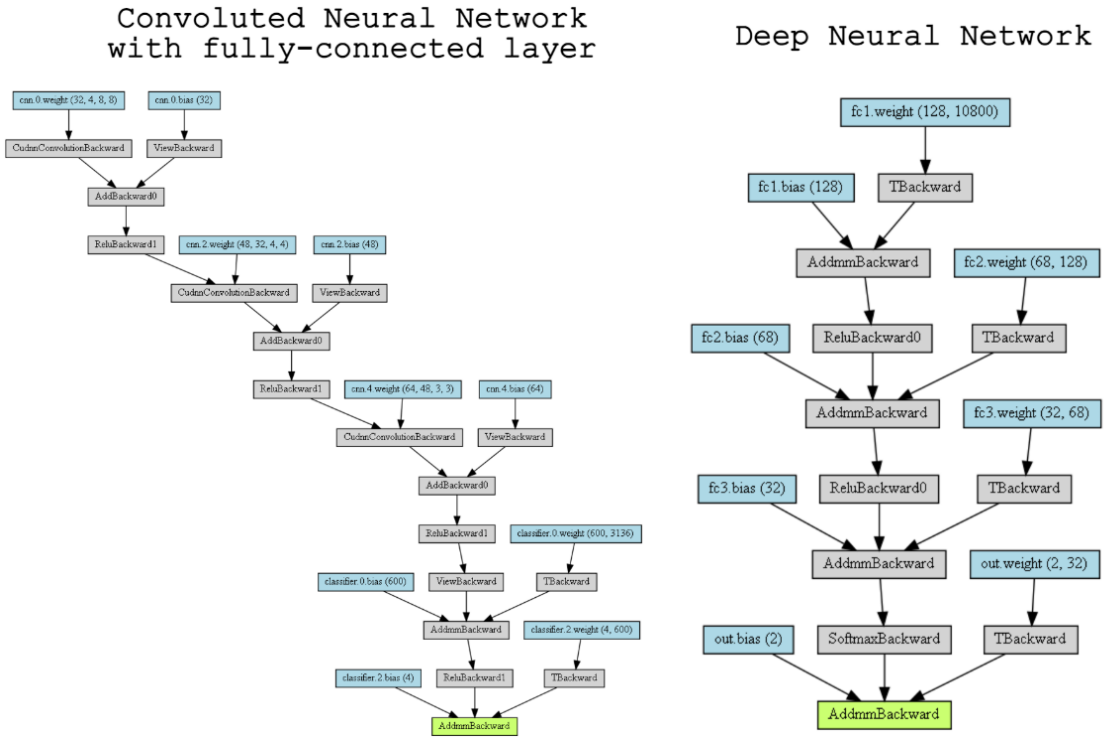


Figure 9: The Convoluted Neural Network and Deep Neural Network used by the agent, both with default parameters for DQL implementations (Section 5.1).

4.8 RL Algorithms

4.8.1 Deep Q-Learning

For DQL and DDQL, a policy network is established which is used for returning the current actions and Q-Values for a given state. Alongside this, a target network is also instantiated that is updated periodically, depending on the target update factor, which is used for returning the target Q-Values of a given state. To optimise the network, a loss function needs to be defined for performing backpropagation, value-based methods use the basic Mean-Squared loss function which accepts the current Q-Values and target Q-Values.

The current Q-Values are simply the action values for a given state in the policy network. However, the target Q-Values consist of retrieving all the states proceeding each state and selecting those that are not the ending state of the current episode; final states are the actions that lead to the end of the episode and thus have a reward of 0, hence these are not included for updating the policy network as these will have a Q-Value of 0. The maximum Q-Values of each of the next-states are then returned and these are used to estimate the target Q-Values:

```
# Uses formula E[reward + gamma * maxarg(next state)] to update Q values
target_q_values = (next_q_values * discount) + rewards
```

Figure 10: Calculation of the current target values by finding the expected values from the target network.

Hence, the difference between the actual Q-Values and target Q-Values provides a loss value that is used for optimising the policy network. The selected optimiser for all the policies is the Adam optimiser with the default PyTorch settings.

4.8.2 Double Deep Q-Learning

DDQL uses both the policy and target network when providing the target Q-Values. The updated target values are returned by obtaining the index of the highest Q-Value from the policy network and use that index to obtain the Q-Value from the target network. Hence, both networks are used for inferencing which has been calculated to lead to more stable Q-Values:

```
with torch.no_grad():
    # Find the max actions from the policy net
    max_policy_values = policy_net(non_final_states).detach().max(dim=1)[1]
    # Return values for max actions in target net and the policy net
    values[non_final_state_locations] = target_net(non_final_states).detach().gather(dim=1,
                                                                                     index=max_policy_values.
                                                                                     unsqueeze(-1)).squeeze(-1)
```

Figure 11: Selection of Q-Values using both the target and policy network. This code snippet was inspired by reference [6].

4.8.3 REINFORCE

REINFORCE is the only Policy-Gradient algorithm currently defined for the project. These methods do not require a target network or replay memory structure; hence these are not instantiated for REINFORCE agents. The main algorithm works by calculating the expected reward and using the mean logarithmic probabilities for each taken action as a loss to update the policy.

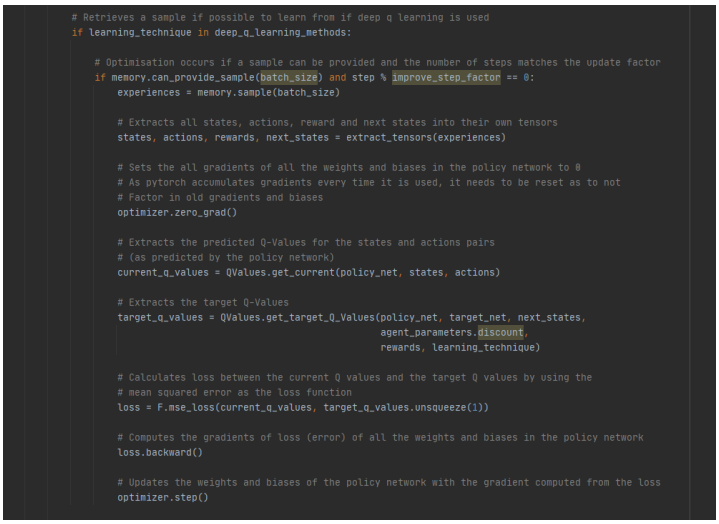
```
# Calculate logarithmic probabilities
logprob = torch.log(policy_net(states))

# Returned estimated probabilities with reward
calculated_logprobs = rewards * torch.gather(logprob, 1, actions.unsqueeze(1)).squeeze()

# Final mean loss
loss = -calculated_logprobs.mean()
```

Figure 12: Calculated loss of the Policy Gradient class. This code snippet was inspired by reference [11].

4.9 Training

Training is executed in the *main.py* file and executes a simple loop that iterates through the number of episodes defined by the user parameters. Each training loop iterates indefinitely until the episode in the environment has finished, with each iteration defining a step that the agent takes. With each step, the agent produces an action based on the action strategy that is being employed, and this action is taken in the environment. After the action is taken, the environment handler returns a reshaped reward and the next processed state. Finally, the current state, action, reward and next state are stored in replay memory to be sampled for optimisation.

An update factor is defined by the user and defines how often to optimising the agent. If the agent is using DQL or DDQL, and a batch of experiences can be provided, then the agent retrieves a batch of experiences from the replay memory to optimise. The current Q-values and target Q-values are returned from the target and policy networks, and the policy network is optimised:



```
# Retrieves a sample if possible to learn from if deep q learning is used
if learning_technique in deep_q_learning_methods:

    # Optimisation occurs if a sample can be provided and the number of steps matches the update factor
    if memory.can_provide_sample(batch_size) and step % improve_step_factor == 0:
        experiences = memory.sample(batch_size)

        # Extracts all states, actions, reward and next states into their own tensors
        states, actions, rewards, next_states = extract_tensors(experiences)

        # Sets the all gradients of all the weights and biases in the policy network to 0
        # As pytorch accumulates gradients every time it is used, it needs to be reset as to not
        # Factor in old gradients and biases
        optimizer.zero_grad()

        # Extracts the predicted Q-Values for the states and actions pairs
        # (as predicted by the policy network)
        current_q_values = QValues.get_current(policy_net, states, actions)

        # Extracts the target Q-Values
        target_q_values = QValues.get_target_Q_Values(policy_net, target_net, next_states,
                                                      agent_parameters.discount,
                                                      rewards, learning_technique)

        # Calculates loss between the current Q values and the target Q values by using the
        # mean squared error as the loss function
        loss = F.mse_loss(current_q_values, target_q_values.unsqueeze(1))

        # Computes the gradients of loss (error) of all the weights and biases in the policy network
        loss.backward()

        # Updates the weights and biases of the policy network with the gradient computed from the loss
        optimizer.step()
```

Figure 13: Optimisation used for DQL and DDQL, if a batch of experiences can be returned and an update is scheduled, then a batch of experiences are returned; the current and target Q-Values are returned to form an MSE loss which is used to optimise the current policy network.

After an episode has finished, all the information about the episode is stored for later reference and the performance (total reward, steps taken, moving average, current epsilon, time taken) is displayed to the user. Furthermore, after a defined period of episodes, the current progress of the agent is presented on reward graph that displays the agent performance. If the agent is using REINFORCE, then optimisation of the policy is performed at the end of a defined number of episodes.

Once the number of episodes has been exceeded and training has concluded, the stored results of each episode are output to an Excel file and the weights of the policy network are saved, allowing for later execution of the currently training agent.

4.10 User Customisation

The program allows for the user to define custom models that can be executed on a given environment. Each proposed model for each environment is stored in a JSON file named *game_parameters.json* and is loaded each time the program is executed. Each model has a required set of parameters that must be defined for training to execute; with each parameter being validated to ensure the greatest chance of successful learning. A full explanation of each of the available parameters can be found in Appendix C.

A *settings.json* file is also included, which defines program parameters disconnected to the training process, which define user interaction factors as well as what custom model to employ.

Chapter 5

Implementations

The focus of this chapter is to present the proposed models that were developed for each of the selected Atari environments. It is important to note that for each of the following implementations, there were many that failed to learn. It was through using these failed attempts that the final models could be developed.

5.1 Generalised Model Parameters

After many iterations of each executing each environment, it was clear that some agent parameters needed to be constant for efficient learning to commence. Having consistent parameters is not only beneficial for minimising potential discrepancy errors, but also provides a basis for comparing each implementation. Other variable parameters are altered for each implementation for maximum learning efficiency.

Firstly, it was discovered that a learning rate of *0.0005* was optimal for each environment. When the learning rate was set higher, it was found that in some environments the agent failed to converge on an optimal policy, and general performance in later stages of the game suffered consequently. However, lower learning rates generally took look longer to achieve higher scores, hence being less computationally efficient.

For Atari environments, actions that are taken early heavily impact the reward that the agent achieved in later stages of the game; in Breakout, hitting the ball towards a brick will result in a reward roughly 15 steps after the initial contact with the paddle. Hence, a discount factor of between *0.99-0.995* was found to be optimal for prioritising future rewards.

As more environments were tested, it was discovered that very few substitutions were required for the CNN, as it was flexible enough to adapt to all proposed environments. Hence, the following layers and properties were homogenised for all implementations:

Layer Number	Layer Type	Number of Neurons/Inputs	Kernel Size	Stride
1	*Convoluted*	32	8	4
2	*Convoluted*	58	4	2
3	*Convoluted*	64	3	1
4	*Fully-Connected*	600	-	-

Table 1: List of chosen CNN parameters for all implementations

The reason why the kernel size and stride decrease with each convoluted layer is because the first

layer has the main objective of extracting the main intrinsic values from the image and removing unnecessary information; a large kernel and stride ensure a large coverage of the image and only important values are kept when passed to the next layer. With each succeeding layer, the kernel size and stride are reduced to fine-tune the remaining values and avoid losing valuable image information. Neuron sizes were chosen as such to ensure that enough neurons are available to ensure the neural network functions correctly, while also avoiding becoming too bulky and reducing computation times.

5.2 Breakout

The Breakout game consists of a simple environment where the goal of the agent is to maximise the number of bricks hit before running out of lives. The agent controls a paddle which bounces the ball upwards towards the bricks. Available actions are: left, right, hold. The environment rewards are the game score counter, which increments every time the agent hits a brick, with more points being given for bricks that are harder to hit.

The main benefit of Breakout is the simplistic nature of the environment; the only components are the ball, paddle, and bricks- therefore Breakout is the default environment for developing Atari RL implementations. However, the default state for Breakout is not fully Markovian; the trajectory and velocity of the ball cannot be inferred without previous states. Hence, previous states from the environment state queue are incorporated as individual tensors when returning the current state. Furthermore, as the colour of the bricks is not important, the state is converted to a grayscale image. Two approaches were taken when developing implementations for Breakout.

The first approach was to consider only focusing on maximising reward from hitting the bricks, with the reward being the returned score from the Breakout environment. Overall, this approach resulted in achieving optimal scores, with the agent even learning to target the ball to hit bricks at the side of the screen resulting in the ball bouncing on top of the bricks, achieving large reward increases. However, the issue with this method was that a lot of training was required to achieve an optimal policy. Furthermore, once enough bricks were destroyed, the agent struggled to keep the ball in session as late-game states had not been optimised yet.

For this approach, a reward-based epsilon decay was taken for choosing the action strategy, because there is a direct correlation between the environment score and the robustness of the current policy; if the agent is achieving high scores, it should exploit the policy more. However, once the agent achieves a score greater than 100, the reward no longer impacts the exploration rate due to the influx in reward the agent achieves when the ball bounces off the bricks on the top layer, as this would result in the agent no longer taking as many explorative actions when the policy is not fully optimised.

However, another method was developed which prioritised not losing a life and disregarded the native scoring system of the environment. Rather than focusing on the score, the agent now receives

a -10 reward whenever a life is lost. Furthermore, as the bricks are not the priority, these have been completely removed from the return state, with only the current position of the ball and paddle being emphasised. Ultimately, this method was computationally more efficient, as the state size is reduced, and this converged on an optimal policy in fewer episodes as the only focus is to not lose a life. However, ignoring the bricks meant that towards the end of each episode, when fewer bricks were present on screen, the agent would not target the remaining bricks and result in an infinite loop of keeping the ball in session. Overall, this was the preferred method mainly due to the speed of training and high scoring.

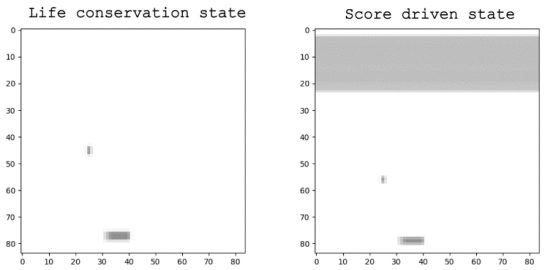
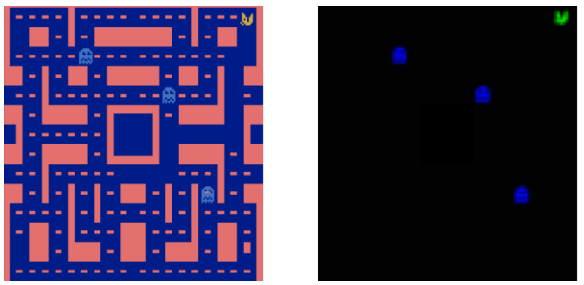


Figure 14: The processed states for the two different Breakout implementations.

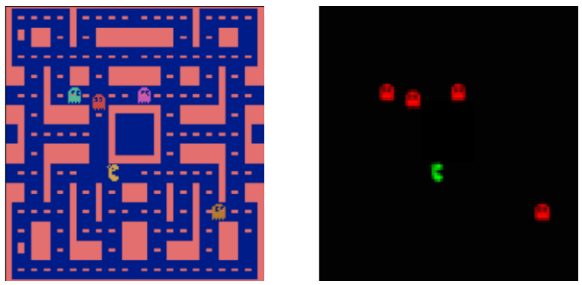
5.3 Pacman

Pacman, or Ms. Pacman for Atari 2600, has the agent controlling a Pacman sprite which navigates through a maze-like environment. The goal is to avoid the incoming ghosts, eating as many pills as possible, and eating vulnerable ghosts after a power pill is consumed. Available actions are: up, down, left, right, hold. The environment rewards are the score from eating a pill or eating a vulnerable ghost.

It was clear that simply resizing and cropping the returned environment state was not suitable for such a complex environment. The main issue is the ghost spirits, as each ghost has a unique RGB value that the agent must learn to avoid. Hence, each ghost must be normalised so that each pixel containing a ghost has the same value. A proposed solution was developed where each dimension of the RGB array only contains key important information that defines the properties of the environment, these are the: Pacman sprite position, hostile ghosts, and vulnerable ghosts. For context, a ghost is hostile when in contact with the Pacman sprite and will result in a loss of life, while vulnerable ghosts can be eaten by Pacman for a large point bonus.



15.1: Normal and processed vulnerable ghost state



15.2: Normal and processed hostile ghost

Two issues were soon made apparent after running the agent. Returned environment rewards generally do not convey the true objective of the game: staying alive. Currently, the reward corresponds directly with the points achieved in the score counter. Each eaten pill is worth a score of 10, however eating a ghost gives a score of 200, with each successive eaten ghost providing a multiplier of 2. Hence, rather than conserving lives, the agent will prioritise traversing to the nearest power pill and moving rapidly to try and maximise the number of ghosts eaten. This methodology has no regard for conserving lives and ultimately leads to unsatisfactory results.

Hence a new custom reward scheme was derived; the new reward is the current distance between the Pacman sprite and the nearest hostile ghost. Therefore, the agent will prioritise maintaining a broad distance from the hostile ghosts, which conserves lives. However, the reverse is taken for when each ghost is vulnerable, with the reward given to how close Pacman is to the nearest vulnerable ghost. Now, the agent will focus on maintaining a large distance from hostile ghosts, while also chasing vulnerable ghosts.

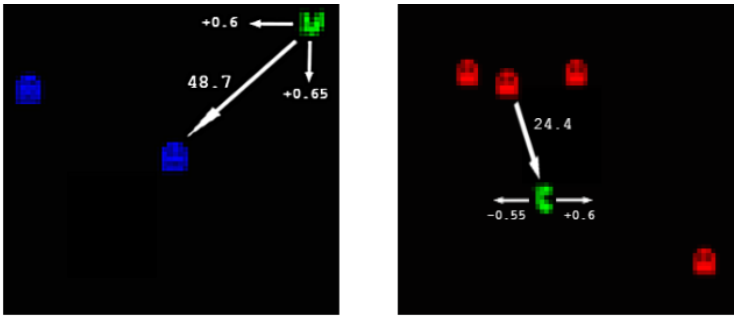


Figure 16: Hostile and vulnerable processed ghost states with the distance between Pacman the nearest ghost. The reward for taking certain actions can be seen based on the increase/decrease in relative distances.

Currently, the environment is not fully Markovian, as the agent has no context of how long each ghost is vulnerable for. Fortunately, each ghost alternates between white and blue before becoming hostile again. When this occurs, the ghost is represented as being fully hostile to the agent, which results in the reward converting to maximising the respective distance. It is important to note that the Pacman environment contains several known bugs where in some state frames ghosts do not always appear in their respective positions; this has been accounted for by storing the previously known positions of each ghost and substituting this into the state if a ghost is not currently present.

Finally, the batch size was lowered to about 15 for DDQL and DQL implementations. Because each choice that the agent makes drastically impacts the trajectory of success, evaluation of the policy should be focused on short batches of experience. It was found that a large batch size generally resulted in worse performance as there was a higher correlation between starting states and future states, thus resulting in the Pacman sprite oscillating between left and right at the origin position. Furthermore, a one-life approach was taken, as it was found that the long death and starting animations generally lead to less cohesive learning.

5.4 Enduro

Enduro views the agent controlling a car driving on a road. Other cars are present, traveling down the road at a constant speed. The goal of the game is to overtake as many cars as possible before the time runs out. Available actions are: accelerate left, accelerate right, accelerate forward, brake, decelerate, coast, left, right, hold. The environment reward is +1 for when the agent overtakes a car, and -1 when the agent gets overtaken.

The interesting aspect of the Enduro environment is that it is one of the few Atari environments that have a concept of 3D space and perspective; cars further along the road appear smaller and increase in size the closer they are to the agent car sprite. However, this perspective shift does not pose larger complexities for the agent compared to other 2D environments, as the size of each car provides all the information needed for the agent to transition the car into a position that will avoid hitting the current car in front. Enduro has all properties needed to be Markovian and hence the only state manipulation required was to crop and greyscale each state. Furthermore, the reward for the environment correctly correlates with the main goal of the agent, with a point being given every time the agent overtakes a car, hence no extra reward manipulation was required.

For DQL implementations the epsilon strategy was altered. The interesting part of the Enduro environment is that the agent must hold the accelerator for a long period to propel the car forward and overtake other cars, which produces a reward. Hence, exploring heavily results in the agent not receiving any rewards early, as the agent will never take a series of subsequent actions that result in enough speed being gained. Therefore, the agent exploits the policy earlier, when the policy has optimised enough that holding the accelerator is considered the most optimal action. However, after a certain epsilon value, the decay is lower; the agent will now take enough optimal actions to gain speed, but enough random actions that the agent learns that avoiding oncoming cars results in a positive reward for overtaking.

Finally, it is important to consider that the Enduro environment has four different weather/day cycles during each episode: sunny, inverted, night and foggy. Each cycle has a different representation of the environment. It was decided not to normalise each state like the Pacman implementation, as the neural network should be robust enough to still identify each car and adjust the position accordingly.

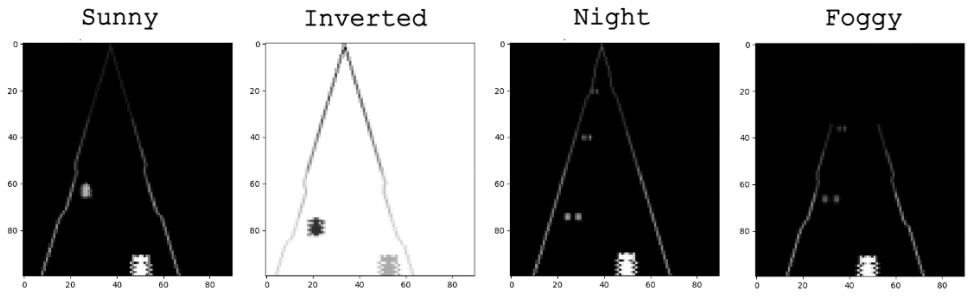


Figure 17: Different processed states for the different Enduro environment cycles.

Chapter 6

Evaluation

6.1 Experiment Results

Breakout

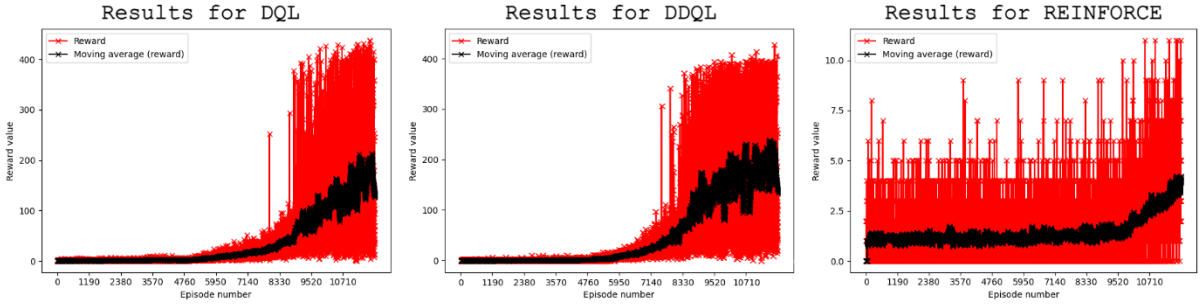


Figure 18: Results for the Breakout implementations for each different RL algorithm. The returned score/reward is used for determining the effectiveness of each implementation. For authentication, each graph was created by the project software.

Pacman

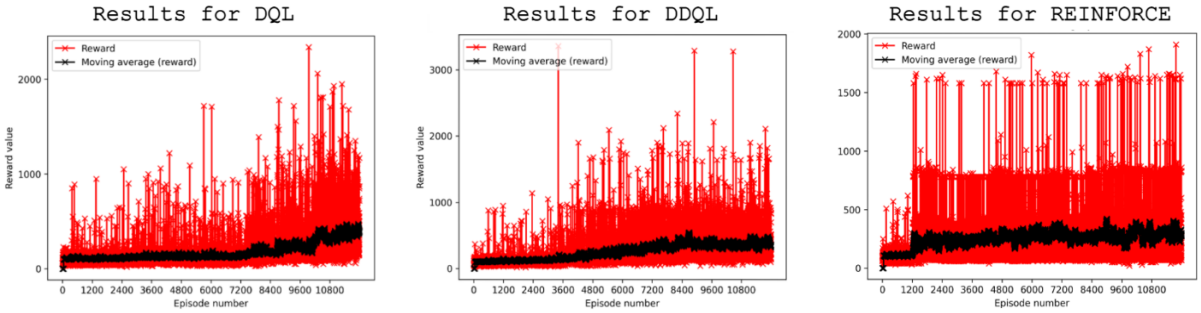


Figure 19: Results for the Pacman implementations for each different RL algorithm.

Enduro

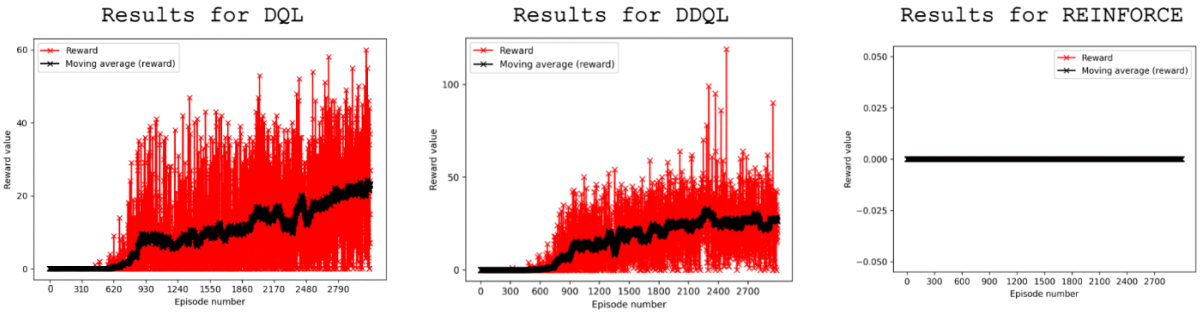


Figure 20: Results for the Enduro implementations for each different RL algorithm.

Compiled Results

Environment	*DQL*	*DDQL*	*REINFORCE*	*Random*	*Human*
Breakout	165.16 (±120.52)	186.63(±134.14)	3.2 (±2.08)	1.70	27.90
Pacman (one-life)	385.65 (±249.12)	377.03(±261.47)	300.51(±254.66)	98.90*	7687.50*
Enduros	18.2(±12.35)	25.95 (±13.80)	0.0 (±0.0)	0.00	740.20

Table 2: Summary of the average scores achieved through executing each implementation. When training the agent, 3000 episodes were conducted for the Enduros agent, while 12000 episodes were conducted for Breakout and Pacman, this was due to Enduros having a fixed game length of ≈3000 steps. Each implementation ran with parameters defined in Appendix D and each implementation was executed 3 times, each with arbitrary starting seeds. The score for each implementation takes the average score for the final 1000 episodes; scores before this do not accurately display the performance of the implementation. The human and random scores were extracted from the implementation used by Nair et al [34, Data Table 2]. As the Pacman implementation was executed with only one-life, the random and human values were divided by 2 to account for the 3-life difference. The scores were divided by 2 as generally, more points are gained on the first life, as eaten pills do not regenerate. While this does not always scale linearly, this will work effectively for the sake of comparison.

6.2 Implementation Analysis

It is important to consider that a fixed number of training episodes were set for each implementation; given more training episodes, the average score may have improved, but due to computation constraints a set number of episodes were utilised rather than terminating when the average score no longer improved.

The Breakout implementation was extremely effective; with the DQL method scoring over 600% better than the average human player. This is mainly because the environment is extremely simplistic, with the only moving entities being the paddle and ball, both critical components for scoring highly. The only way that improvements can be made to the implementation would be to invoke higher-level RL algorithms to improve learning rates or fine-tune parameters.

While the Pacman implementation failed to achieve high scores, the policy was roughly 380% more efficient than the invoked random policy for DQL. The drawback of the implementation is most likely due to focusing completely on life conservation, as it was clear the agent would often traverse straight towards the nearest power pill to turn the hostile ghosts vulnerable but would not focus on maximising scores through eating the rest of the pills currently in session. A better solution would be to use the returned environment rewards but reshape each reward so that a good policy can be developed where the agent focuses on maintaining lives while also maximising the in-game score. Overall, considering that Pacman has so many different components to consider, achieving even the low scores shows that the implementation has the potential to improve given more development.

The Enduro implementation generally took around 500 episodes to learn that accelerating was the best policy for most states since this is when the agent started to gain rewards through overtaking cars. It was observed that the agent quickly learned that the best strategy was to stick close to the side of each road, as this maximised the chances of avoiding oncoming cars. However, the agent

generally performed poorly when anticipating oncoming cars that should be avoided, with the agent normally accelerating and sticking to either side of the road. Reshaping rewards could help solve this issue; the negative reward for letting a car overtake could be increased so that the agent learns rather than taking the greedy action, which is to simply accelerate regardless of any oncoming cars, to try to transition to smoothly overtake oncoming cars. Furthermore, the way that the states were processed meant that there were different representations for the different time cycles; a simpler representation could also improve scores. Overall, the Enduro was relatively successful, but generally could be improved to achieve much higher scores.

6.3 Evaluation and Comparison of Implemented Algorithms

Unfortunately, the implemented REINFORCE algorithm did not perform as well as initially envisioned. It can be seen for the Breakout implementation that after around 9000 episodes the agent started to learn coherent actions; the algorithm implementation does work, however there is a clear fault with some of the parameters. Furthermore, the REINFORCE algorithm performed unexpectedly for one of the Pacman implementations. After roughly 1200-1500 episodes, the score suddenly increased from roughly 100 to 300 and did not improve after this point, this is visible in figure 19. The algorithm should slowly converge to an optimal policy, so a sudden increase in reward and no subsequent learning is an unexpected result, implying that there are underlying issues with the algorithm implementation.

One issue could simply be that the learning rate is too low, as it took the agent many episodes before any adequate actions were taken for Breakout, potentially increasing the learning rate may fix this. Furthermore, the action strategy could contribute to the low learning rate. Usually, for Softmax action strategies a temperature parameter **T** is utilised which denotes how often an agent takes a random action [60]. This however wasn't utilised, and each action probability is simply taken as the probability that action will be chosen; this could have resulted in the environment not being explored enough, leading to inadequate learning through only slow exploitation.

However, the most likely reason is that not enough episodes were performed for the agent to start learning. It is important to consider that the value-based algorithms perform optimisation on large batches of experiences, with the Breakout implementation perfoming an update every 4 steps with a batch size of 20. Hence, ignoring that samples are returned arbitrarily, this means that each experience is utilised for optimisation five times. While policy-gradient methods only perform an update at the end of every episode, with each experience used for optimisation once. Therefore the ratio of learning is 1:5. The reason why this was not factored in when compiling the final results was simply that 60000 episodes could not have been calculated in time for the project due date; this is addressed in section 8.2.

Nevertheless, it was clear when executing these algorithms that Policy-Gradient approaches are much more concise than DQL or DQLL. The lack of replay memory and target network reduces the

complexity model significantly. Furthermore, the exploration issue was reduced further by the ability to use the probabilities generated by the model as the action strategy, rather than having to tinker with epsilon values to ensure that the agent explores the environment effectively.

DDQL was more effective for converging on an optimal policy than its counterpart DQL. This is supported by the results for Breakout, with the model performing an average of 10% better at 12000 episodes. The use of the target network when returning the target Q-Values generally lead to more effective optimisation and less overestimation of errors. However, DDQL does not solve all the issues related to value-based approaches, as unstable Q-Values and overestimation are still a hinderance for optimised learning. Algorithms such as Deep Quality-Value Networks attempt to solve this through training multiple separate networks [61], and generally more similar complex solutions should be used to the now deprecated DQL and DDQL algorithms.

Chapter 7

Ethical issues

7.1 Legal Considerations

While most of the source code was developed individually, the main structure was based on the source code provided by PyTorch. However, this does not breach copyright law: PyTorch allows redistribution of source code, provided that the copyright notice is contained within the source code and names of the copyright holders are used to endorse or promote products which utilise Pytorch [39].

Libraries that were included for this project are all open-source and thus licensed under the respective BSD and MIT licenses. No alterations were made to the source code directly and credit has been given in the references and appendix of the report. Other resources that have been utilised have been correctly referenced and are protected through the fair dealing exception under the copyright, designs, and patents act of 1988 [62].

7.2 Ethical Considerations

Given the current scale of the project, the machine learning agent does not provide any ethical issues at the current version. However, this does mean that if more complex machine learning techniques are utilised, the ethical considerations will need to be reassessed. Furthermore, if more complex environments, which involve human behaviours and personal data, were to be used to train the agent, then the ethical issues will need to be respected. Hence, any potential ethical considerations will realistically need to be framed in the context of the environment that is being learned from, as the agent has specifically been designed for Atari games this is not a concrete issue.

7.3 Social Considerations

As stated above, the agent currently learns from only Atari game environments, and thus there is little human interaction with the agent. Furthermore, no external users were utilised for testing the performance of the agent; only comparisons between other methods and agents were utilised. Therefore, unless modifications are made, no social issues are associated with the project.

7.4 Professional Considerations

Currently, the project is considered small scale and unfit to be distributed for a professional environment, which reduces the risk of potential professional issues. However, all development for the project has been in accordance with the code of conduct defined by the British Computer Society and the ACM Code of Ethics and Professional Conduct [38].

Chapter 8

Conclusion

8.1 Objective Evaluation

"To develop a reinforcement learning model that can interact and learn with a given game environment."

Overall, this objective has been met. All the environments that were provided to the agent were correctly handled, and given correct parameters, the agent was able to use the gained experience to optimise each move to improve the overall scores. Modern and established machine learning techniques were utilised efficiently to develop a successful DRL model.

"Use various reinforcement learning algorithms and highlight the differences in training performance."

While several learning algorithms were utilised, only the DQL and DDQL worked effectively with the Atari environments. The REINFORCE algorithm seemed to show promise, and did work for the Breakout and Pacman implementation, but unfortunately an efficient model could not be developed in the respective time frame. However, comparisons between the methods were still performed and potential issues with the implementation of REINFORCE were addressed. Hence, this objective has only been partially met.

"Suggest several unique implementations for different games that achieve competent scores."

For each implementation, a series of state and reward manipulation techniques were developed that correctly simplified each complex environment to improve the agent performance. By far the most successful implementation was Breakout, which showed performance that surpassed standard human interaction. While the Pacman implementation did not achieve high scores, learning did occur and given the complex nature of the environment, the agent was still able to make logical actions for a given state even if they were not always optimal for future situations. Given the fact that each implementation did lead to successful training and agent learning, this object has been met.

"Provide a basic interface to allow users to develop customised models."

This criterion has been achieved. The user can add individual custom models and execute these using the defined JSON files, with each parameter being validated to reduce chances of user error. A wide variety of features can be easily employed to improve agent performance. Furthermore, abstraction of the environment handler class further allows users to further customise models, given an understanding of the fundamental processes of the architecture.

8.2 Personal Reflection

It is clear before starting the project, I did not fully appreciate the complexities that occur when developing DRL models. Firstly, the main issue is that debugging DRL models is an incredibly

ambiguous process. If a model fails to learn an environment, there is often no clear reason why, it will simply fail to converge to an effective solution. When I first started to develop the main Atari model, I was often left clueless as to why the current model was failing to learn, as the issue could range from something as simple as having the wrong parameters to an inflexible Neural Network. The solution to this issue is to research more of other similar implementations and evaluate why those worked when compared to my model. If this were done earlier, about 3 weeks of troubleshooting could have been avoided, which would have meant more DRL algorithms could have been developed.

Furthermore, what I failed to consider was that training an agent takes a lot of computation time. On my hardware, the average time to complete a normal episode of Breakout would take around 10 seconds; hence a full training cycle consisting of 10000 episodes would take 28 hours. The original plan was to upgrade my GPU, but due to the GPU shortage (2021) [63], I failed to purchase one before project termination. Rather than training on my hardware, I should have contacted the University to negotiate using the large range of specialised GPUs that are offered. If this had occurred, so much time would have been saved waiting for the agent to finish training so the results could be analysed. Furthermore, the evaluation was impacted by this, as I struggled to retrieve all the results in time for the project due date; more episodes could have been executed, resulting in more coherent comparisons between the results.

This leads me to the largest lesson that was learnt from this project: time management. Rather than working a few hours a day on the project consistently, I would often work in infrequent large batches. Not only is this bad practice, but this was also hindered further when developing DRL models, as each time I had to wait for training to finish before evaluation of the proposed solution could commence. With future projects, I will space my time out better to avoid these types of issues.

Moreover, undertaking this project has taught me the value of patience; there were many times where a model failed to learn, and it was tempting to terminate the project prematurely as it seemed like I was making no progress. However, after every model that failed, I learnt a little more about DRL and the potential reasons the model did not learn, until finally a working solution was discovered.

Even though the project was not as optimal as originally intended, I am pleased with how the project concluded. Reinforcement learning is such a new and complicated sector of computer science, and I had never undertaken a project of this scale. Furthermore, while the solutions that were proposed are considered simplistic compared to current standards, the important fact is that I understood all the techniques that were being employed, and this has given me experiences which will be invaluable for the next project I undertake, especially if it utilises machine learning.

8.3 Further Work

While a successful solution has been developed, there are certainly improvements and expansions that need to be employed:

1. Improve parameters for the implemented REINFORCE algorithm.
2. Currently very basic DRL algorithms have been employed, and while these are successful, more complex approaches that have been shown to achieve more successful results. The next step will be to employ better value-based algorithms, starting a Duelling DQN and later expand to using a Deep Quality Value Network [12]. The next algorithm that will be employed is A3C, an Actor-Critic algorithm. Finally, the pinnacle of the project would be to design a new, bespoke DRL algorithm that works efficiently for Atari games.
3. The Pacman environment provides many unique challenges for developing DRL solutions, due to the high complexity. A better implementation which achieves high scoring results would be an interesting and unique challenge which would provide further insight into to the shortcomings of the current model.

8.4 Final Assessment

Overall, I am reasonably content with how the project concluded. Given the experience that I have gained through managing a project of this scale, I am confident that the next reinforcement learning project that I undertake will be handled a lot more effectively. I am excited to see what new machine learning techniques are developed in future years now that I understand the fundamentals.

References

[1] Paszke A. (n.d.). *Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.8.0 documentation*. [online] Available at: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html. [Accessed 02 Mar. 2021].

[2] Dr. Matteo Leonetti. (2020/2021). *Leeds University: COMP3611 Machine Learning (34333)-lectures/recordings*

[3] Arulkumaran, K., Deisenroth, M.P., Brundage, M. and Bharath, A.A. (2017). *Deep Reinforcement Learning: A Brief Survey.*

[4] Sutton, H. (2014). *Reinforcement Learning: An Introduction*

[5] Deeplizard.com. (n.d.). *Deep Q-Network Code Project Intro - Reinforcement Learning*. [online] Available at: https://deeplizard.com/learn/video/FU-sNVew9ZA [Accessed 02 Mar. 2021].

[6] Alvin Wan. DigitalOcean. (n.d.). *Bias-Variance for Deep Reinforcement Learning: How To Build a Bot for Atari with OpenAI Gym*. [online] Available at: https://www.digitalocean.com/community/tutorials/how-to-build-atari-bot-with-openai-gym [Accessed 08 Feb. 2021].

[7] Aakashns (n.d.). *01-pytorch-basics - Jovian.* [online] jovian.ai. Available at: *https://jovian.ai/aakashns/01-pytorch-basics*. [Accessed 01 Dec 2020].

[8] Pytorch.org. *Deep-Learning-With-Pytorch*. [online] Available at: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html [Accessed 01 Dec 2020].

[9] Marsland, S. (2015). *MACHINE LEARNING: an algorithmic perspective, second edition.* S.L.: Crc Press.

[10] JasonBian97 (2020). *jasonbian97/Deep-Q-Learning-Atari-Pytorch*. [online] GitHub. Available at: https://github.com/jasonbian97/Deep-Q-Learning-Atari-Pytorch [Accessed 2 Mar. 2021].

[11] Finspire13 (2020). *Finspire13/pytorch-policy-gradient-example*. [online] GitHub. Available at: https://github.com/Finspire13/pytorch-policy-gradient-example [Accessed 2 Apr. 2021].

[12] Shao, K., Tang, Z., Zhu, Y., Li, N. and Zhao, D. (2019). A Survey of Deep Reinforcement Learning in Video Games. *arXiv:1912.10944 [cs]*. [online] Available at: https://arxiv.org/abs/1912.10944 [Accessed 14 Apr. 2021].

[13] Mnih, V., Kavukcuoglu, K., Silver, D. *et al.* Human-level control through deep reinforcement learning. *Nature* 518**,** 529–533 (2015). https://doi.org/10.1038/nature14236

[14] Madhu Sanjeevi (2018). *Ch:13: Deep Reinforcement learning — Deep Q-learning and Policy Gradients (towards AGI).* [online] Medium. Available at: https://medium.com/deep-math-machine-

learning-ai/ch-13-deep-reinforcement-learning-deep-q-learning-and-policy-gradients-towards-agi-a2a0b611617e [Accessed 2 Apr. 2021].

[15] Tabor, P. (n.d.). *A Conceptual Introduction to Policy Gradient Methods*. [online] www.neuralnet.ai. Available at: https://www.neuralnet.ai/a-conceptual-introduction-to-policy-gradient-methods/ [Accessed 12 Apr. 2021].

[16] web.archive.org. (2016). *Atari: 1972 - 1984*. [online] Available at: https://web.archive.org/web/20160314004954/https://www.atari.com/history/1972-1984 [Accessed 4 Apr. 2021].

[17] tutorialspoint.com (2019). *SDLC Agile Model*. [online] www.tutorialspoint.com. Available at: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm.

[18] www.neuraldesigner.com. (n.d.). *Performance comparison of dense networks in GPU: TensorFlow vs PyTorch vs Neural Designer*. [online] Available at: https://www.neuraldesigner.com/blog/training-speed-comparison-gpu-approximation [Accessed 4 Apr. 2021].

[19] Lind, E., Pantigoso, Ä., Skolan, K., Elektroteknik, F. and Datavetenskap, O. (n.d.). *A performance comparison between CPU and GPU in TensorFlow*. [online] . Available at: https://www.diva-portal.org/smash/get/diva2:1354858/FULLTEXT01.pdf [Accessed 4 Apr. 2021].

[20] Gamasutra.com. (2019). *Gamasutra - A History of Gaming Platforms: Atari 2600 Video Computer System/VCS*. [online] Available at: https://www.gamasutra.com/view/feature/3551/a_history_of_gaming_platforms_.php [Accessed 16 Apr. 2021].

[21] Sejnowski, T.J. (2018). *The deep learning revolution*. Cambridge, Massachusetts: The Mit Press.

[22] Clancy, J. (2018). *The Pros and Cons of Using GitHub for Repository Management*. [online] CodeClouds. Available at: https://www.codeclouds.com/blog/advantages-disadvantages-using-github/.

[23] atariage.com. (n.d.). *AtariAge - Atari 2600 Manuals (HTML) - Breakout (Atari)*. [online] Available at: https://atariage.com/manual_html_page.php?SoftwareID=889 [Accessed 16 Apr. 2021].

[24] Hackr.io. (n.d.). *PyTorch vs TensorFlow: Difference you need to know*. [online] Available at: https://hackr.io/blog/pytorch-vs-tensorflow.

[25] Bogost, N. (2020). *RACING THE BEAM : the atari video computer system.* S.L.: Mit Press.

[26] problemkaputt.de. (n.d.). *Atari 2600 Specifications*. [online] Available at: https://problemkaputt.de/2k6specs.htm [Accessed 6 Apr. 2021].

[27] Deepmind. (2016). *AlphaGo: The story so far*. [online] Available at: https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[28] paperswithcode.com. (n.d.). *Papers with Code - Atari Games*. [online] Available at: https://paperswithcode.com/task/atari-games [Accessed 6 Apr. 2021].

[29] Chandra, A.L. (2018). *McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron*. [online] Medium. Available at: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1.

[30] Goodfellow, I., Yoshua Bengio and Courville, A. (2017). *Deep learning*. Cambridge, Massachusetts: The Mit Press.

[31] Brownlee, J. (2018). *How to Configure the Number of Layers and Nodes in a Neural Network*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/.

[32] pythonprogramming.net. (n.d.). *Python Programming Tutorials*. [online] Available at: https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/.

[33] colab.research.google.com. (n.d.). *RL Cheatsheet*. [online] Available at: https://colab.research.google.com/drive/1eN33dPVtdPViiS1njTW_-r-IYCDTFU7N [Accessed 18 Apr. 2021].

[34] Van Hasselt, H., Guez, A. and Silver, D. (2015). *Deep Reinforcement Learning with Double Q-learning*. [online]. Available at: https://arxiv.org/pdf/1509.06461.pdf.

[35] Neven Piculjan(n.d.). *Schooling Flappy Bird: A Reinforcement Learning Tutorial*. [online] Available at: https://www.toptal.com/deep-learning/pytorch-reinforcement-learning-tutorial [Accessed 18 Apr. 2021].

[36] Hubbs, C. (2019). *Learning Reinforcement Learning: REINFORCE with PyTorch* [online] Medium. Available at: https://towardsdatascience.com/learning-reinforcement-learning-reinforce-with-pytorch-5e8ad7fc7da0 [Accessed 11 Apr. 2021].

[37] Stevens, E., Luca Antiga, Viehmann, T. and Soumith Chintala (2020). *Deep learning with PyTorch*. Shelter Island, Ny: Manning.

[38] Association for Computing Machinery (2018). *ACM Code of Ethics and Professional Conduct*. [online] Acm.org. Available at: https://www.acm.org/code-of-ethics.

[39] PyTorch (2017). *PyTorch BSD 3-Clause* [online]. https://github.com/pytorch/examples/blob/master/LICENSE

[40] Aakash Maroti (2019). *Reward Based Epsilon Decay* [online]. https://arxiv.org/abs/1910.13701

[41] Nielsen, M.A. (2019). *Neural Networks and Deep Learning*. [online] Neuralnetworksanddeeplearning.com. Available at: http://neuralnetworksanddeeplearning.com/chap2.html.

[42] Reed, R.D. and J, R. (1999). *Neural smithing: supervised learning in feedforward artificial neural networks*. Cambridge, Mass.: The Mit Press.

[43] Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. [online] arXiv.org. Available at: https://arxiv.org/abs/1609.04747.

[44] Hubel, D.H. and Wiesel, T.N. (1968). *Receptive fields and functional architecture of monkey striate cortex*. London: Cambridge University Press.

[45] Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, [online] 25. Available at: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[46] SOOD, K. and Fiaidhi, J. (2020). *Capsule Networks: An Alternative Approach to Image Classification Using Convolutional Neural Networks*. [online] figshare. Available at: https://www.techrxiv.org/articles/preprint/Capsule_Networks_An_Alternative_Approach_to_Image_Classification_Using_Convolutional_Neural_Networks/12100713/1 [Accessed 19 Apr. 2021].

[47] Chatterjee, M. (2020). *The Introduction of KNN Algorithm | What is KNN Algorithm?* [online] GreatLearning. Available at: https://www.mygreatlearning.com/blog/knn-algorithm-introduction/.

[48] Arulkumaran, K., Deisenroth, M.P., Brundage, M. and Bharath, A.A. (2017). Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine*, 34(6), pp.26–38.

[49] Bellemare, M.G., Naddaf, Y., Veness, J. and Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, [online] 47, pp.253–279. Available at: https://arxiv.org/pdf/1207.4708.pdf.

[50] TORRES.AI, J. (2020). *Deep Q-Network (DQN)-II*. [online] Medium. Available at: https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c [Accessed 21 Apr. 2021].

[51] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In Proceedings of the 12th International Conference on Machine Learning (ICML 1995)

[52] Matthieu Geist, Bilal Piot, Olivier Pietquin. Is the Bellman residual a bad proxy?. NIPS 2017 - Advances in Neural Information Processing Systems, Dec 2017, Long Beach, United States.

[53] Janner, M., Fu, J., Zhang, M. and Levine, S. (2019). When to Trust Your Model: Model-Based Policy Optimization. *arXiv:1906.08253 [cs, stat]*. [online] Available at: https://arxiv.org/abs/1906.08253 [Accessed 23 April 2021].

[54] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D. (2015). Human-level control through deep

reinforcement learning. *Nature*, [online] 518(7540) Available at: https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf.

[55] GeeksforGeeks. (2019). *SARSA Reinforcement Learning*. [online] Available at: https://www.geeksforgeeks.org/sarsa-reinforcement-learning/ [Accessed 26 Apr. 2021].

[56] Jason Brownlee (2019). *Understand the Impact of Learning Rate on Neural Network Performance*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

[57] tcnguyen.github.io. (n.d.). *SARSA vs Q - learning*. [online] Available at: https://tcnguyen.github.io/reinforcement_learning/sarsa_vs_q_learning.html [Accessed 1 May 2021].

[58] Lilian Weng (2018). *Policy Gradient Algorithms*. [online] Lil'Log. Available at: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html.

[59] Sutton, R., Singh, S. and Mcallester, D. (n.d.). *Comparing Policy-Gradient Algorithms*. [online]. Available at: http://incompleteideas.net/papers/SSM-unpublished.pdf [Accessed 3 May 2021].

[60] Tijsma, A.D., Drugan, M.M. and Wiering, M.A. (2017). *Comparing exploration strategies for Q-learning in random stochastic mazes*. Institute Of Electrical And Electronics Engineers Inc. -02-09.

[61] Sabatelli, M., Louppe, G., Geurts, P. and Wiering, M. (n.d.). *Deep Quality-Value (DQV) Learning*. [online]. Available at: https://arxiv.org/pdf/1810.00368.pdf. [Accessed 6 May 2021]

[62] Gov.uk (2019). *Copyright, Designs and Patents Act 1988*. [online] Legislation.gov.uk. Available at: https://www.legislation.gov.uk/ukpga/1988/48/contents.

[63] PCWorld. (2021). *Nvidia expects crippling GPU shortages to continue throughout 2021*. [online] Available at: https://www.pcworld.com/article/3614866/nvidia-expects-crippling-gpu-shortages-to-continue-throughout-2021.html.

Appendix A

External Materials

External web resources that which were used or inspired some of the components of the project:

PyTorch RL Tutorial [8]	*https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html*
Deeplizard RL Tutorial [5]	*https://deeplizard.com/learn/video/FU-sNVew9ZA*
Deep Q-Learning Agent [10]	*https://github.com/jasonbian97/Deep-Q-Learning-Atari-Pytorch*
PyTorch Basics tutorial [7]	*https://jovian.ai/aakashns/01-pytorch-basics*
REINFORCE Agent [11]	*https://github.com/Finspire13/pytorch-policy-gradient-example*

Required External Software:

Python (3.8 or higher)	*https://www.python.org/downloads/release/python-381/*
PyTorch	*https://pytorch.org/get-started/locally/*
Torchvision	*https://pytorch.org/vision/stable/index.html*
Gym	*https://gym.openai.com/docs/*
Numpy	*https://numpy.org/install/*
Atari_py	*https://github.com/openai/atari-py*
Matplotlib	*https://matplotlib.org/stable/users/installing.html*

Recommended External Software:

Torchvis	*https://pypi.org/project/torchviz/*
GraphVis	*https://graphviz.org/download/*
Pandas	*https://pandas.pydata.org/getting_started.html*
Xlsxwriter	*https://pypi.org/project/XlsxWriter/*
OpenPyxl	*https://openpyxl.readthedocs.io/en/stable/*

Appendix B

Project Documentation

GitHub Repository:

Contains all project source code for correct execution of all topics discussed in the report. Also includes a README file explaining how to run the project, as well as links to extra project resources:

https://github.com/Woebegonemite/atarai/

Hardware:

The hardware specifications that were used for development of the project. Hence, these are the hardware specifications that are recommended for training the agent; better specifications are encouraged:

Operating System	*Windows 10 Pro*
CPU	*AMD Ryzen 7 3700X 8-Core Processor (3.6 Ghz)*
GPU	*Geforce Gtx 970*
Memory	*32 GB (DDR4)*
Minimum Available Storage	*10GB*

Integrated Development Environment:

For the duration of the project, development was performed on the PyCharm IDE. This is the recommended IDE for running the project, as this has been tested to work. While other platforms for running the Python source code should work, it is guaranteed to run effectively on PyCharm:

https://www.jetbrains.com/pycharm/

Recommended Project Settings File:

These are the current settings stored on the repository inside the *settings.json* file, these have been configured to work if recommended modules are not installed:

```
{
  "running_atari_game": "BreakoutDeterministic-v4",
  "parameter_index": 0,
  "num_training_episodes": 15000,
  "show_progress": true,
  "plot_update_episode_factor": 100,
  "save_policy_network_factor": 200,
  "render_agent_factor": 1,
  "show_processed_screens": false,
  "test_agent_file": null,
  "show_neural_net": false,
  "save_results_to_excel": false,
  "set_seed": 0
}
```

Appendix C

Agent Parameters

Parameter	***Required for Value-based methods?***	***Required for Policy-gradient methods?***	***Description***
Learning technique	True	True	The RL algorithm to be used for learning; currently the supported algorithms are: DQL, DDQL and REINFORCE.
Learning rate	True	True	The learning rate used by the Adam optimiser.
Epsilon strategy	True	False	States which epsilon strategy to use.
Epsilon values	True	False	The values used by the strategy (e.g. Epsilon Decay)
Batch size	True	False	The number of experiences over which each gradient descent update is performed
Memory size	True	False	Number of experiences stored for sampling
Memory start size	True	False	States when to start returning batches and optimising value-based policy methods.
Target network update frequency	True	False	The frequency with which the target network weights are updated with the policy network
Step update frequency	True	False	How many steps before performing an optimisation using gradient descent.
Episode update frequency	False	True	How many episodes before performing an optimisation using gradient descent.
Discount	True	True	The discount factor (gamma) for each reward.
Resize	True	True	The size of each processed state in pixel width and height.
Interpolation mode	True	True	The interpolation mode for resizing the image.
Crop values	True	True	The percentage width and height to crop off the original returned state from the environment.
Screen process type	True	True	How the environment handler returns the final state. The available options are: append, morph, difference or standard.

Colour type	True	True	Whether to return the full RGB values (1 dimension) or to return a grayscale state (3 dimension).
State queue size	True	True	Number of previous states stored in the environment handler, used for process types such as append or morph.
Policy	True	True	The neural network to use as the policy.
Policy Parameters	True	True	The parameters used for the neural network architecture.
Reward Scheme	True	True	How the environment handles the returned rewards.

Appendix D

Implementation Parameters

Breakout:

DQL/DDQL		REINFORCE	
Learning Parameter	**Value**	**Learning Parameter**	**Value**
Learning rate	*0.0005*	*Learning rate*	*0.0005*
Epsilon Start	*1*	*Discount*	*0.99*
Epsilon Midpoint	*0.1*	*Episode update frequency*	*1*
Epsilon End	*0.01*	-	-
Epsilon Start Decay	*0.0001*	-	-
Epsilon End Decay	*0.00001*	-	-
Discount	*0.99*	-	-
Batch Size	*20*	-	-
Memory Size	*25000*	-	-
Memory Start size	*5000*	-	-
Target network update Frequency	*6*	-	-
Step Update Frequency	*4*	-	-

Environment Parameter	**Value**
Colour type	*Gray*
Screen resize	*84, 84*
Screen process type	*Append*
State Queue Size	*4*
Width Crop Percentages	*0.05-0.95*
Height Crop Percentages	*0.45-0.95*
Use Environment Reward	*False*
Lives Change Reward	*-10*
End when life lost	*False*
Normalise Environment Rewards	*False*

CNN Layer Number	**Layer Type**	**Number of Neurons/Inputs**	**Kernel Size**	**Stride**
1	*Convoluted*	*32*	*8*	*4*
2	*Convoluted*	*58*	*4*	*2*
3	*Convoluted*	*64*	*3*	*1*
4	*Fully-Connected*	*600*	-	-

Pacman:

DQL/DDQL		REINFORCE	
Parameter	**Value**	**Parameter**	**Value**
Learning rate	*0.0005*	*Learning rate*	*0.0005*
Epsilon Start	*1*	*Discount*	*0.995*
Epsilon Midpoint	*0.1*	*Episode update frequency*	*1*
Epsilon End	*0.01*	-	-
Epsilon Start Decay	*0.0004*	-	-
Epsilon End Decay	*0.000001*	-	-
Discount	*0.995*	-	-
Batch Size	*15*	-	-
Memory Size	*25000*	-	-
Memory Start size	*5000*	-	-
Target network update Frequency	*6*	-	-
Step Update Frequency	*3*	-	-

Environment Parameter	**Value**
Colour type	*RGB*
Screen resize	*100, 90*
Screen process type	*Standard*
State Queue Size	*1*
Width Crop Percentages	*0-1*
Height Crop Percentages	*0.01-0.81*
Use Environment Reward	*False*
Lives Change Reward	*0*
End when life lost	*True*
Normalise Environment Rewards	*False*

CNN Layer Number	**Layer Type**	**Number of Neurons/Inputs**	**Kernel Size**	**Stride**
1	*Convoluted*	*32*	*8*	*4*
2	*Convoluted*	*58*	*4*	*2*
3	*Convoluted*	*64*	*3*	*1*
4	*Fully-Connected*	*600*	-	-

Enduro:

DQL/DDQL		REINFORCE	
Parameter	**Value**	**Parameter**	**Value**
Learning rate	*0.0005*	*Learning rate*	*0.0005*
Epsilon Start	*1*	*Discount*	*0.95*
Epsilon Midpoint	*0.1*	*Episode update frequency*	*1*
Epsilon End	*0.01*	-	-
Epsilon Start Decay	*0.0008*	-	-
Epsilon End Decay	*0.0001*	-	-
Discount	*0.995*	-	-
Batch Size	*40*	-	-
Memory Size	*25000*	-	-
Memory Start size	*5000*	-	-
Target network update Frequency	*2*	-	-
Step Update Frequency	*6*	-	-

Environment Parameter	**Value**
Colour type	*Gray*
Screen resize	*100, 90*
Screen process type	*Standard*
State Queue Size	*1*
Width Crop Percentages	*0.05-1*
Height Crop Percentages	*0.25-0.73*
Use Environment Reward	*True*
Lives Change Reward	*0*
End when life lost	*False*
Normalise Environment Reward	*False*

CNN Layer Number	**Layer Type**	**Number of Neurons/Inputs**	**Kernel Size**	**Stride**
1	*Convoluted*	*32*	*8*	*4*
2	*Convoluted*	*58*	*4*	*2*
3	*Convoluted*	*64*	*3*	*1*
4	*Fully-Connected*	*600*	-	-

Appendix E

External Figures

Figure 6 [45]: