Reinforcement Learning For Telescope Optimisation

Curtly Blows

Supervisor: Prof. Bruce Bassett

September 10, 2019

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Reinforcement learning is a relatively new and unexplored branch of machine learning with a wide variety of applications. This study investigates reinforcement learning and provides an overview of its application to a variety of different problems. We then explore the possible use of reinforcement learning for telescope target selection and scheduling in astronomy with the hope of effectively mimicking the choices made by professional astronomers. This is relevant as next-generation astronomy surveys will require near real-time decision making in response to high-speed transient discoveries. We experiment with and apply some of the leading approaches in reinforcement learning to simplified models of the target selection problem. We find that the methods used in this study show promise but do not generalise well. Hence while there are indications that reinforcement learning algorithms could work, more sophisticated algorithms and simulations are needed.

Chapter 1

Machine Learning

Machine learning is a term that covers a wide variety of techniques in the areas of mathematics, statistics (Rajaraman and Ullman 2011). It is often incorrectly referred to as Artificial Intelligence since some parts of machine learning are basic statistics while other components utilise machine learning techniques to create intelligent systems (Shalev-Shwartz and Ben-David 2014).

Machine learning is the process of having a program receive data and provide insight into the data, through different algorithms (Marsland 2009). The objective with machine learning is to get the program to act as humans do but at a larger scale and a faster rate. Machine learning can be broken into three different fields, namely Supervised, Unsupervised and Reinforcement learning (Rajaraman and Ullman 2011).

Supervised learning is the objective of learning how input variables relate to output variables(Marsland 2009). The objective would then be to use this learned behaviour to make a new prediction based on new input variables. The better the model, the more accurately it predicts the output variable.
Unsupervised learning is the process of an algorithm deriving insight from data with no output variable. This usually includes clustering the data or performing some sort of dimensionality reduction (Rajaraman and Ullman 2011). These two methods usually do not make predictions based on how the prediction affects future predictions since the prediction will not have an impact on any future prediction.
Reinforcement learning is the branch of machine learning that deals with sequential decision making (Busoniu et al. 2010). It is one of the best methods for a program to use that resembles the way humans make decisions, thus it is closely related to artificial intelligence.

1.1 Supervised Learning

Supervised learning is still seen as the most common application of machine learning. A set of training data, from previously occurring events, is used to create a model which can explain the correlations in the data. The training of these models consists of features (input data) as well as a target variable (output data) (Pedregosa et al. 2011). Supervised learning methods use the features to attempt to predict the target variable. After a supervised model

is trained, it can be used to predict unseen data, where the instances in the data have not been used to train the model and the target variable is unknown (Shalev-Shwartz and Ben-David 2014; Marsland 2009). The aim of supervised learning is to build and fit a model that maps features to target variables, with the aim of accurately predicting the target variable or deriving insight into the relationship between features and target variables (James et al. 2014; Raschka 2015). Supervised learning can be broken into two algorithm categories, namely classification, and regression (Rajaraman and Ullman 2011). The choice of which supervised method to use is dependent on the type of problem since some algorithms can be applied to only regression or classification and not both.

1.1.1 Regression

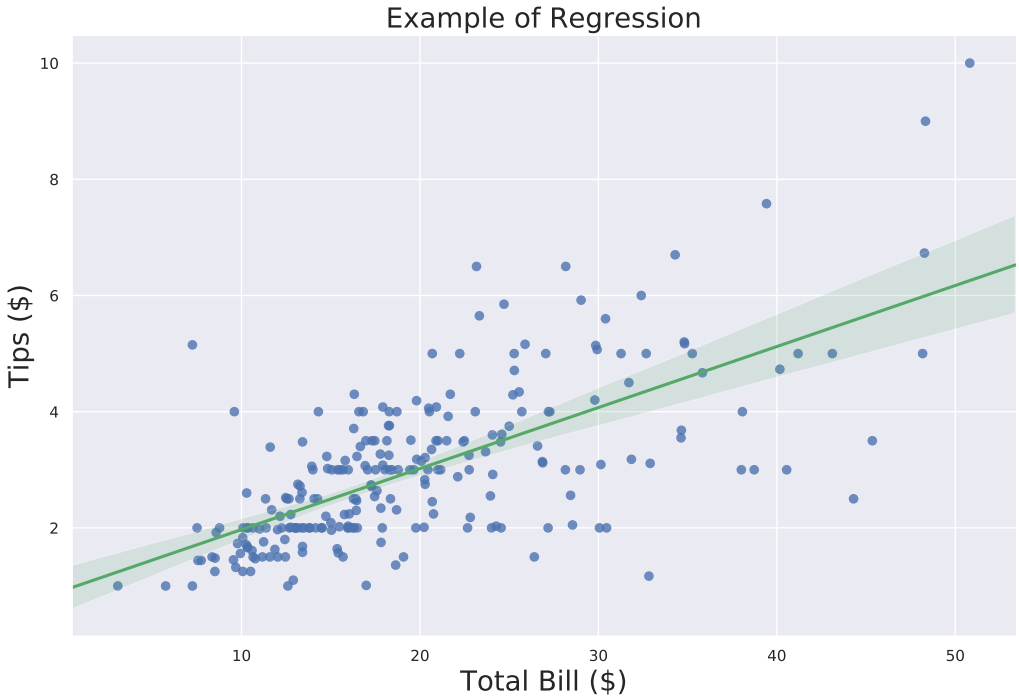

Figure 1.1: An example of a regression task. The data represents a particular restaurant's bills and the corresponding tip the waiter received for each bill. The green line represents the relationship the model has learned, mapping the bill total(feature variable) to tips (target variable).

When one looks toward regression as a solution, the target variable is of a quantitative nature. The target variable, in this case, is usually a singular continuous variable (Marsland 2009), i.e. it can be seen as any point on the real number line, $\in \mathbb{R}$. A simple example of regression is the model of linear regression (Pedregosa et al. 2011). Linear regression assumes that there is a linear relationship between a single feature variable (James et al. 2014).

Mathematically this is written as:

$$Y \approx mX + c \tag{1.1}$$

In layman's terms, the target variable, Y, is approximately equal to the input variable, X, multiplied by some weight, m, with the addition of an initial term/intercept, c. The objective is now to determine the unknown variables, m and c, such that they represent the equation above given the data (James et al. 2014). We use data to estimate these coefficients, let us assume a data set consisting of n data points,

$$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$$

The goal is to then obtain coefficients that result in the straight line representing the relationship between X and Y being as close as possible to the data instances (Rajaraman and Ullman 2011). It can be said that the coefficients fit the equation,

$$\hat{y}_i \approx \hat{m}x_i + \hat{c} \tag{1.2}$$

for $i = 1, ..., n$. There are multiple ways of measuring the closeness of the fit, the most common approach is the least squares approach (James et al. 2014). Consider a prediction for the ith prediction, $\hat{y}_i = \hat{m}x_i + c$. The ith

residual is then defined as

$$e_i = y_i - \hat{y}_i$$

where y_i is the actual observed target variable and $\hat{y}_i$ the models predicted output. This way we have a comparison between the target and predicted variable, known as the error or squared error. The equation can be expanded to all data points using the *mean squared error*, given by

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{1.3}$$

Most regression problems use the mean squared error or absolute error, or some variation of this, as an evaluation as to how well the model fits the data and a metric used to adjust any weights or parameters of the model. A model with a low error is said to be a better model, thus the task is to adjust the model parameters such that we *minimise* the error. In linear regression, this then becomes an optimisation problem to find the point where the gradient of the MSE, w.r.t each coefficient, is zero. Using algebraic expansion and calculus, it can be shown that the coefficients for linear regression are

$$m = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{1.4}$$

$$c = \bar{y} - m\bar{x} \tag{1.5}$$

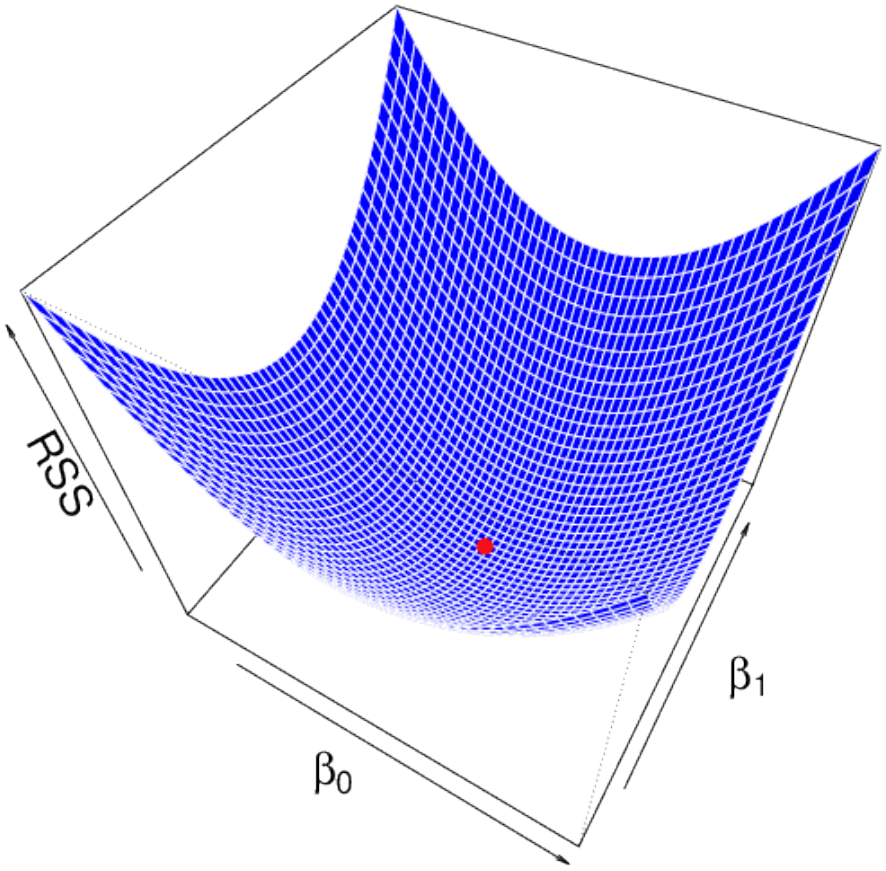

Figure 1.2: A 3D render of the RSS function (similar to the MSE) and the relationship to the coefficients β_0 and β_1 (James et al. 2014). The redpoint marks the minimum RSS, this reflects the best fitting coefficients for the linear regression model.

The figure is an example of the combinations of the parameters and the RSS that is returned using these parameters (James et al. 2014). The RSS

refers to the residual sum of squares, which is similar to the MSE and can be defined as:

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{1.6}$$

1.1.2 Classification

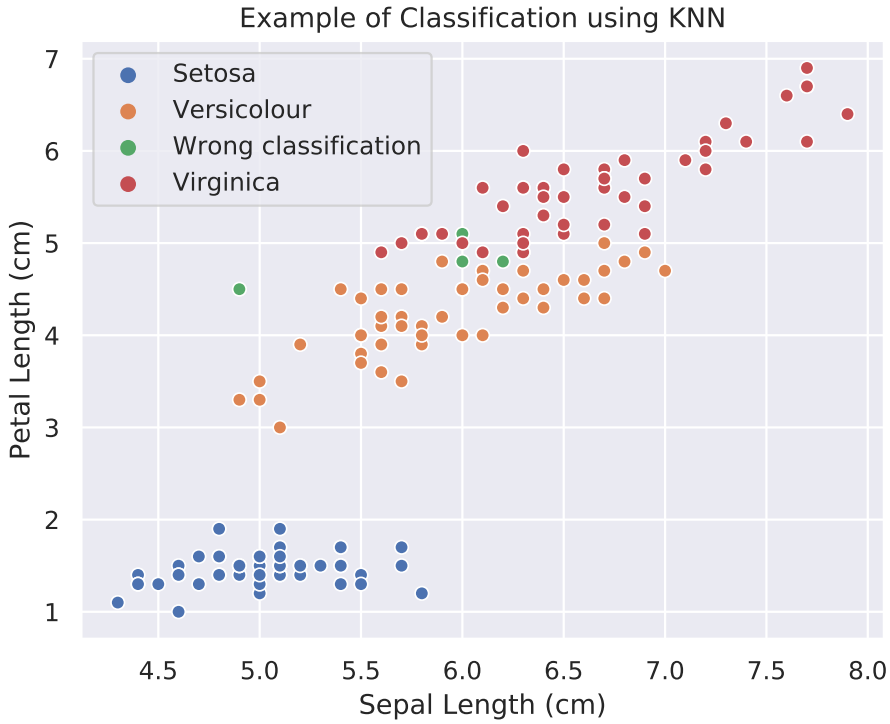

Figure 1.3: An example of a classification task. The data is a sample of measurements for irises, the two variables considered are the petal and sepal length. The colours represent the model correctly or incorrectly classifying the data. Note how the incorrect classification lies in the region where two classes overlap. This is due to the algorithm which classifies a new point by considering the points closest to it.

Classification is an approach whereby the target variable is of a qualitative nature, taking on a distinct value of K classes. Gender, Nationality and credit defaults (yes or no) are all examples of qualitative variables (James et al. 2014). As an example, we will focus on K-Nearest Neighbours (KNN) to get the general idea behind training a classification model. KNN is considered a simple technique but can be powerful in the correct circumstances (Hauck 2014). The model takes in a set of features and stores the data instances in feature space (Marsland 2009). Once a new data point needs to be classified, the input features are compared to the points closest to the new point. The number of points to consider is determined by the factor k (Raschka 2015). The k nearest stored data points (data that was used to train the model) is used as a majority classification. In this way, a data point with similar features to the points closest will be classified similarly (Rajaraman and Ullman 2011).

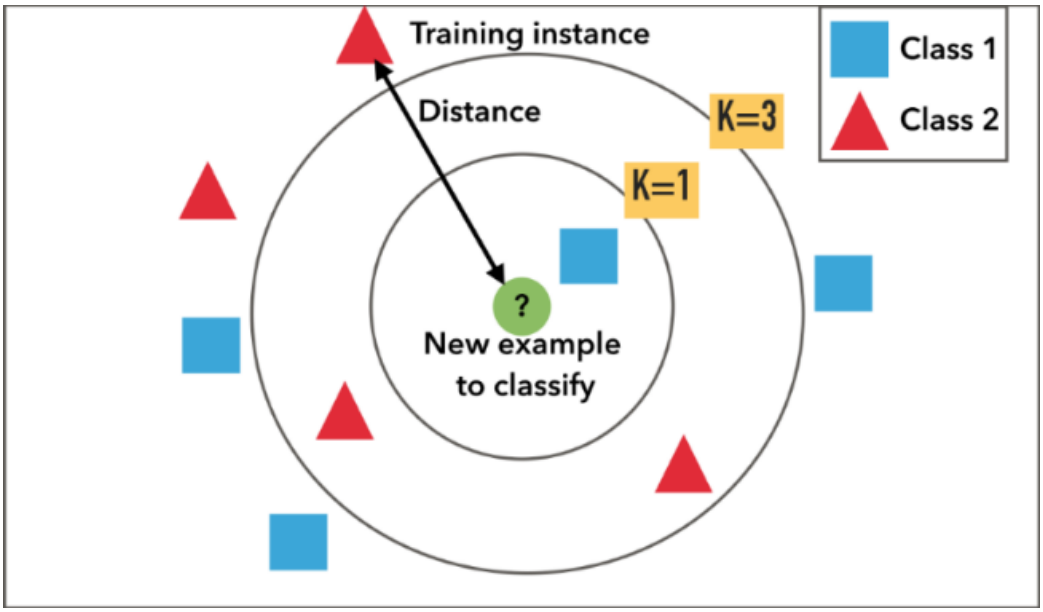

Figure 1.4: An illustration showing the effect of K on classification. The image shows data consisting of 2 classes and a new instance of data to be classified (Bronshtein 2017).

Figure 1.4 shows how the k factor changes the output of the model. A k value of 1 can lead to the new data point being classified as class 1. While a $k = 2$ leads to a classification of class 2 (Bronshtein 2017).

While this particular algorithm has no training or optimisation needed, the most common metric used to optimise the parameters of a classifier is the accuracy of prediction (Raschka 2015). This is also the metric used to assess the model.

A popular application of classification is image classification. Before the implementation of machine learning, the problem of classifying images received poor performing algorithms. The implementation of machine learning and

neural networks caused a great advancement of the task. Recent implementations can be said to have an accuracy higher than human ability. The high accuracy coupled with the speed at which computers can compute means that large volumes of images can be classified in short time spans (Krizhevsky, Sutskever, and Hinton 2012).

1.2 Unsupervised Learning

Unsupervised learning is a more challenging problem since, for every instance/observation in the data, there is a set of features but no target variable (Rajaraman and Ullman 2011). In this case, we seek to understand the relationships between features or to find an underlying structure or pattern within the data (James et al. 2014). This, therefore, means that unsupervised learning does not rely on previously known or labelled data to be trained (Raschka 2015). Unsupervised learning often allows users to gain a deeper understanding of their data, even if it is not obvious at the outset quite what will be learned (Shalev-Shwartz and Ben-David 2014). The most common algorithms in unsupervised learning revolve around doing some sort of cluster analysis or a dimensionality reduction (ibid.). However this does not mean unsupervised methods are limited, the lack of a target variable makes unsupervised methods applicable to a range of problems. (Vincent et al. 2008) has applied unsupervised learning methods to encode data into a format that is easier to train a supervised model on.

Unsupervised clustering is the method of analysis of the structure and distribution of the data and attempting to group data in distinct classes (Marsland 2009). This is similar to classification but the classes and the number of classes are unknown. K-means clustering is a common clustering algorithm. The model is trained by randomly assigning the centre of a class (in feature space) (ibid.). Each data point is then assigned a class based on the distance to the closest class centre. The centre is then adjusted based on the data assigned to it; this is repeated until the centres are relatively static.

Dimensionality reduction is a method of reducing the number of features of a data set. A common occurrence is the inclusion of useless data, this is usually data which does not help in training a model. By reducing the dimensions of the feature variables, the algorithm can improve performance and most likely decrease the time taken to train (Rajaraman and Ullman 2011). In some cases, the reduction of dimensions can often lead to data which is less interpretable.

1.3 Reinforcement Learning

Reinforcement learning is the branch of machine learning that deals with learning optimal behaviour (Sutton and Barto 2017). It is a way of learning what to do in a new environment, how to map a situation to an action, in order to achieve the desired result, i.e. maximise the reward received (Bu-

soniu et al. 2010). The learner must explore its new environment, by trying different actions, and learn which actions yield the best results based on each situation. In general, this learned behaviour can be referred to as the policy, a way to map the state seen to an action (Oliehoek 2012).

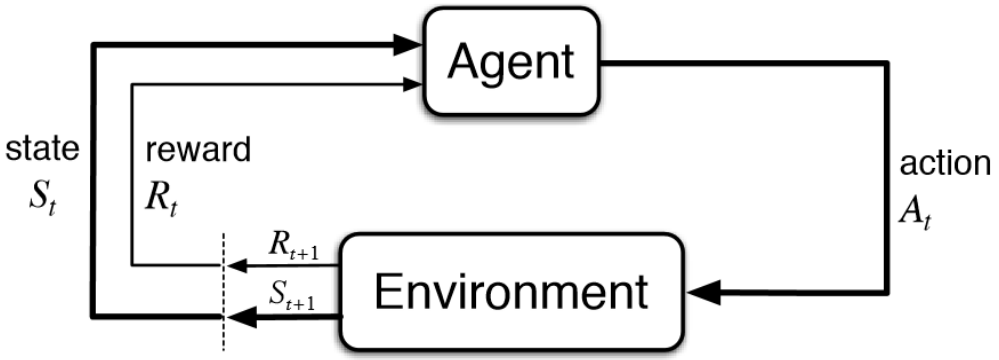

Figure 1.5: The Agent Environment Loop: the cycle of the agent taking actions and the environment telling the agent how the state changed and how good the action was (reward value) (Bhatt 2018).

The machine/algorithm is called the agent and the environment consists of distinct states (Sutton and Barto 2017). In reinforcement learning, the agent is given the state and performs an action. The agent then receives a reward based on how good or bad the action was. The agent then adjusts the policy for that specific action and proceeds to the next state (Busoniu et al. 2010). This is repeated until the policy converges to optimal values. The set of all possible state configurations is named the state space and similarly the set of all actions that can be taken in the environment is named the action

space. An episode consists of transitions starting at the initial state and ending at the terminal state.

The basis of reinforcement learning stems from the Markov Decision Process (MDP) which is a formal description of the reinforcement learning environment (Sigaud and Buffet 2010). MDP is dependant on the Markov property, which states "The future is independent of the past given the present" (Sutton and Barto 2017). This means that the present state captures all relevant information about the past and other past history can be forgotten, thus the present state is a sufficient statistic for the future (Sigaud and Buffet 2010). In terms of Markov decision processes, the agent is presented with a state, s, and takes an action, a. The state changes by some transition function, $T(a)$, into the next state, s' (Sutton and Barto 2017). The mapping of state to action is called the policy, π. There are two main methods in reinforcement learning that make use of MDPs, namely, value-based and policy search methods (Arulkumaran et al. 2017). In some cases, the action taken results in an immediate reward and also affects the rewards received in the future. The agent's goal is to maximise the expected cumulative reward, so in some cases, the agent can take an initial poor action, but the final cumulative reward is larger than if it took an initial good action (Oliehoek 2012). We can model the transition from state to state using probability, i.e.

$$T(s_t, a_t) = P(s_{t+1}|s_t, a_t), \tag{1.7}$$

where s_t and a_t refer to the state and action at time t. This can be defined as the probability of moving into a new state based on the current state and the action taken in that state. This is known as a probabilistic transition function. When the transition function is known, there is no need for probabilities and the transition can instead be written as

$$T(s_t, a_t) = s_{t+1}, \tag{1.8}$$

a deterministic transition function. There are different types of learning in the reinforcement learning field. Model-based reinforcement learning tries to infer the environment to gain the reward while model-free reinforcement learning does not use the environment to learn the action that results in the best reward (Arulkumaran et al. 2017). With model-free learning, the agent must first learn the transition and reward functions, it can then learn a policy with trial and error (exploration and exploitation) (Oliehoek 2012).
There have been many advancements and applications of reinforcement learning, especially with the advancements in neural network architecture.

There are many examples of reinforcement learning outperforming human capabilities, while other examples include reinforcement learning mimicking human behaviour at a large scale. The recent popularity of reinforcement learning is owed to (Mnih et al. 2015). In this study, reinforcement learning was applied to simple 2D Atari games. The general algorithm was able to

achieve human-level performance and in some cases outperform humans at the games presented.

Figure 1.6: An example of an Atari game. The objective is to break all the bricks using the paddle and ball. The RL agent which was trained on this was able to discover its own strategy. The agent would break through the bricks on the far left, which allowed the ball to bounce around in the gap between the bricks and the above wall (Wikipedia contributors 2019).

A more recent study, trained a reinforcement learning agent on the game of Go (Silver, Schrittwieser, et al. 2017). The game has relatively simple actions, but the number of strategies that arise is vastly complex. The agent was trained with no prior knowledge of the game and in some cases discovered new unknown strategies. The agent was able to beat the world Go champion. The team then further improved the agent by making it even more general,

which was then able to outperform the previous iteration (Silver, Hubert, et al. 2018). This new AlphaZero agent was further applied to chemistry and was able to optimise the way in which chemicals are made. Currently, this research team has tackled the Starcraft 2 environment, a strategy game (Vinyals et al. 2017). This is a highly complex problem and the research shows that a reinforcement learning agent is able to play as well as a human. The reinforcement learning algorithm produced, named AlphaStar, was able to outplay two top Starcraft players. The agent did have the advantage of a larger field of view of the play area, as well as being able to do actions at a rate much faster than human capabilities. This does, however, mean that AlphaStar is able to act faster and coordinate attacks that require many actions in a short time frame.

Figure 1.7: The interface of Starcraft 2, the player must be able to control larger armies as well as buildings. The figure shows a battle between AlphaStar and a top human player (Komincz). AlphaStar's Stalkers (blue) attack Komincz's Immortals, Archons, and Zealots (red) on three sides. The stalker units require immense attention, since the units are weak but move fast due to a teleport ability. Top players find it difficult to manage one set of stalkers, but AlphaStar was able to coordinate a three-way attack, which shows the high speed at which it can take actions. AlphaStar's Stalkers ultimately won this battle and went on to destroy Komincz's base (Lee, T. 2019).

There are many other applications like optimising internet traffic flow (Wolpert, Tumer, and Frank 1999) and robotics (Lin 1992).

1.4 Neural Networks

A neural network is somewhat inspired by the structure of the brain (Lippmann 1987). Based on the idea that the brain receives some sort of input, which sets off a chain reaction of signals activating neurons through synapses after which there is some form of output. Similarly to neurons, a neural network comprises of nodes which connect and transfer data via pathways or edges (ibid.). A series of connections and nodes leads to a sort of computational graph(Goodfellow, Bengio, and Courville 2016). Using neural networks one can approximate many functions, linear and nonlinear. The ability to approximate almost all functions is the reason neural networks are so popular and widely adopted.

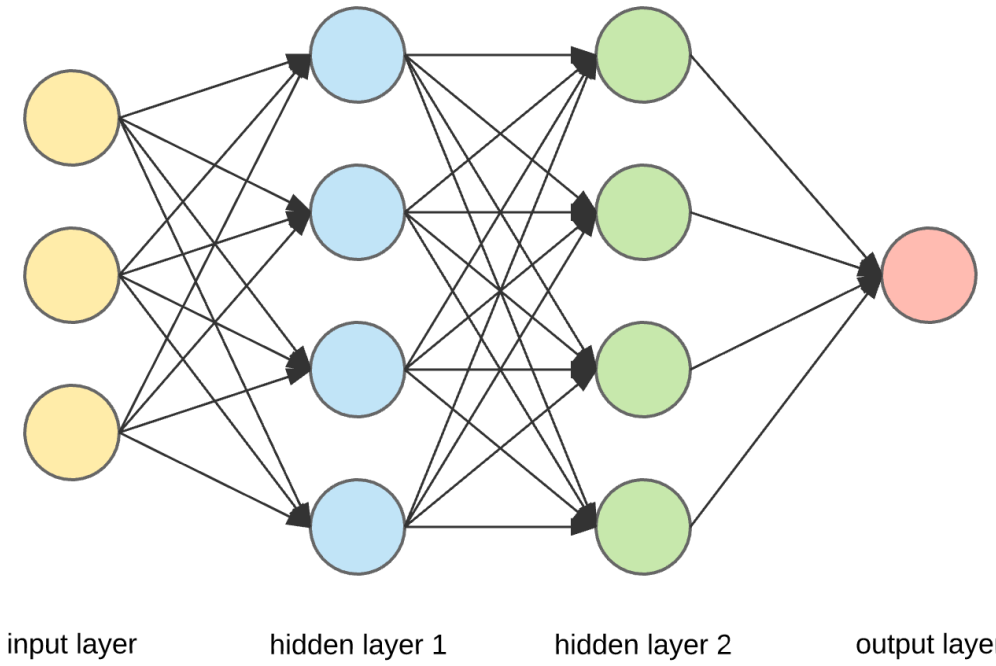

Figure 1.8: A graphical representation of a neural network. It shows a typical feed forward neural network, that maps a 3 dimensional vector mapped to a single output, using the two hidden layers (Dertat, A. 2017).

These neural networks have been applied to numerous fields of research like self-driving cars, voice generation, chat-bots and many more.

1.4.1 Perceptron

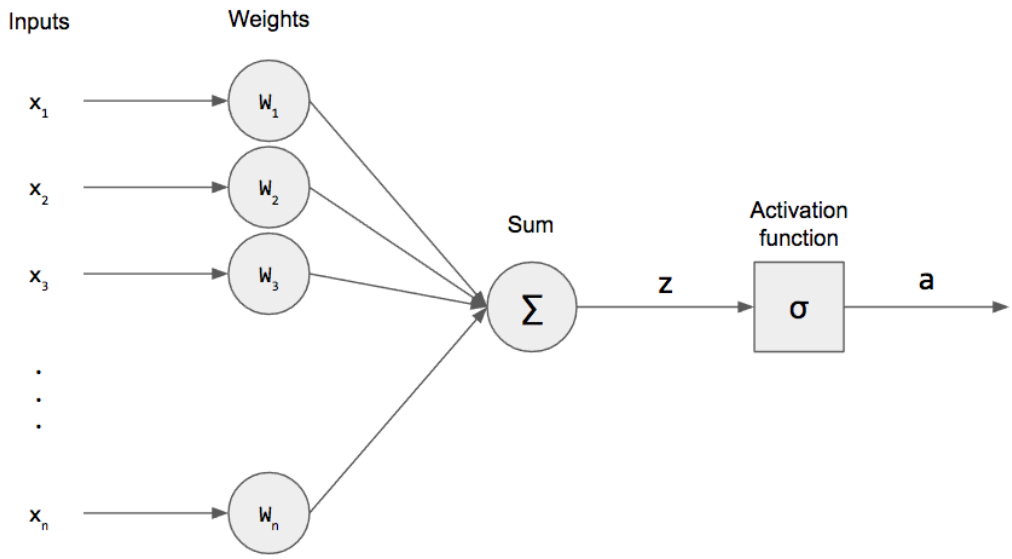

Figure 1.9: Illustration of a single perceptron. The perceptron computes the dot product of the input vectors and corresponding weights (ignoring bias) and outputs a single value (Deshpande, M. n.d.).

A perceptron/node of a neural network is the simplest unit and it is essentially a transformation of input. It maps some input to a single output value. This output is in a binary format, either 0 or 1(Goodfellow, Bengio, and Courville 2016). The mapping is some function $f(x)$ of the form:

$$f(x) = \begin{cases} 1 & \sum w_i x_i > 0 \\ 0 & else \end{cases} \tag{1.9}$$

The input is in the form of a vector, which is dot product multiplied to a weight vector, along with some bias factor added. The weight vector is initialised with random real values. The objective is to optimise these weight values, such that the network learns the importance of each input (Goodfellow, Bengio, and Courville 2016). This then means, using the equation above, that a larger weight corresponds to an input having a larger impact on the output value and more importance (ibid.). Conversely, an input with a small or 0 weight has a smaller contribution. The bias term is added to aid the perceptron in approximating larger numbers.

$$f(x) = \begin{cases} 1 & \sum w_i x_i + b > 0 \\ 0 & else \end{cases} \tag{1.10}$$

1.4.2 Activation functions

In the previous equation, the perceptron contained some activation function. This function acts to convert the sum of the weighted vector inputs into some binary output. This was a simple step function at some threshold. There are different activation functions that can be used like the sigmoid, soft-max and rectified linear unit(ReLU).

The sigmoid function maps the input value to a value between 0 and 1, which can be seen as a probability of sorts(ibid.). Equation 1.11 is an example of

a sigmoid function.

$$f(x) = \frac{1}{1 + \exp^{-x}} \tag{1.11}$$

The soft-max function is typically used for classification problems, since all the outputs of this function sum to 1 and are between 0 and 1(Goodfellow, Bengio, and Courville 2016). This makes the output a good indication of the probability of each class

$$f(x) = \frac{\exp x_i}{\sum_{j=1}^{N} \exp^{x_j}}, \tag{1.12}$$

where N = Number of classes.

The ReLU function outputs 0 if the input is less than 0 but if the input is greater than 0, the input becomes the output(ibid.). The equation for this can be seen below:

$$f(x) = \max(x, 0) \tag{1.13}$$

There have been additional activation functions introduced as neural networks became exposed to more problems, but are not important to discuss for this study. The activation function is important especially for the output of the neural network, e.g. you would not use a soft-max function for a regression problem.

1.4.3 Deep Feed-Forward Neural Network

So far the discussion has been around a single layer network. With the advancements in computation power, multi-layer networks have become more useful when the function to approximate becomes more complex . This complexity arises when the function contains multiple functions that are dependent and interconnected. This lead to the use of multi-layer perceptrons. Each layer of perceptrons can be seen as a function approximator and by linking these layers, it becomes possible to approximate a more complex function(Goodfellow, Bengio, and Courville 2016). A set of perceptrons that are not connected to each other and feed to a further perceptron can be seen as a layer. The first layer is usually the input layer and conversely, the last layer is the output layer (ibid.). Every layer in between the input and output layer is referred to as the hidden layers. Since the weights of every perceptron are not immediately correct, there needs to be some form of adjustment made to each weight (ibid.). As an example, consider a neural network with

three layers, an input, hidden and output layer, one can represent this as:

$$y = NN(x) \tag{1.14}$$
$$= f(w_2 \cdot f(w_1 \cdot x)) \tag{1.15}$$
$$= f(w_2 \cdot f(z_1)) \tag{1.16}$$
$$= f(w_2 \cdot h_1) \tag{1.17}$$
$$= f(z_2) \tag{1.18}$$
$$= h_2 \tag{1.19}$$

In this case, x is the input to the network, w_2 is the weights of the output layer and w_1 is the weight vector for the hidden layer. As a result, h_1 is the output of the hidden layer and h_2 the output of the output layer. f is an arbitrary activation function. Note that the bias term is ignored in this example for simplicity.

A forward pass is a process during training, which takes the input and computes the values at each computation point, i.e. the dot product of weights, bias, and application of the activation function. This then produces an out value or vector. When training a neural network, one has the input values, x, actual output values, y, and the neural network computed output, $\hat{y}$. In order to adjust the weights of the neural network, there needs to be an indication of how close the actual output value is to the computed output value. This is called the loss function. In general regression tasks, the loss function is some form of the mean-squared error or absolute error. For classification,

the loss function can be of a binary or categorical cross-entropy function. Given this loss function $J(\theta)$, where θ can be seen as the weights of the neural network, we can apply *gradientdescent* in order to update the weights in order to minimise the loss function, thereby reducing the difference between y and $\hat{y}$. By taking the negative of the loss function, $-\nabla_\theta J(\theta)$ as well as some learning rate factor, it is now possible to adjust the weights using the equation:

$$\theta' = \theta - \gamma \times \nabla_\theta J(\theta) \tag{1.20}$$

The learning rate dictates the step size of the weight adjustment. If the step is too large, it is possible to avoid the local minimum and conversely, a small step size will increase the number of steps needed to reach the local minimum, which increases the training time. In a neural network, the number of weights can become quite large, making the gradient computation quite expensive.

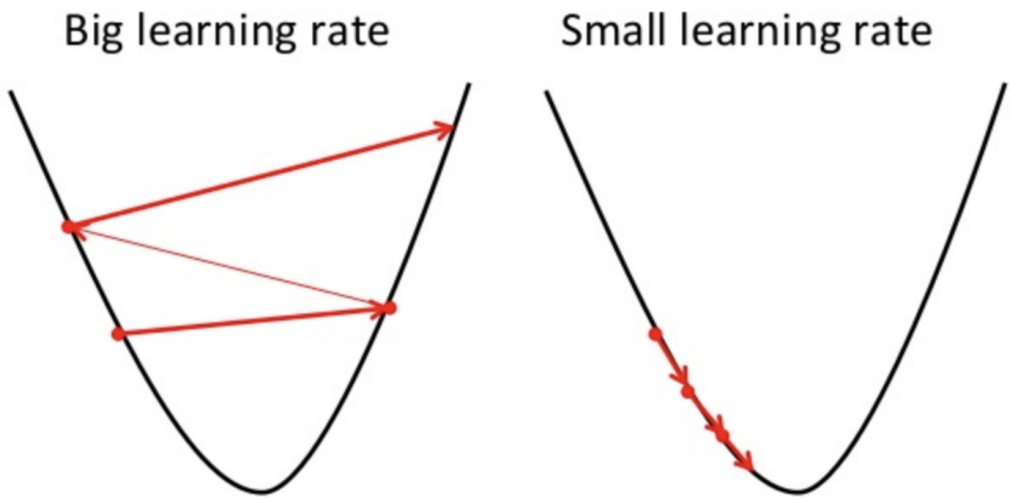

Figure 1.10: A graphical representation of different learning rates. This shows how a large learning rate can cause the neural network to avoid the optimal solution. Whereas, a small learning rate would require more steps to reach an optimal solution. (Donges, N. 2018)

Thankfully the number of computations have been greatly reduced, by the use of the chain rule. The chain rule is a powerful tool used for solving derivatives. It states that if $F(x)$ is the composition of functions f and g i.e $F(x) = f(g(x))$. Then the derivative of F(x) is $F'(x) = f'(g(x))g'(x)$. This is useful in the case of neural networks since computing the gradient of a node/perceptron requires the gradient of the layer before, which in turn requires the layers before. The chain rule simplifies the computation by avoiding multiple computations of the same derivative (Goodfellow, Bengio, and Courville 2016). If one were to have a neural network of two hidden layers, G and H, of two nodes, with one output layer 0. To find the derivative,

$\frac{\partial O}{\partial \theta}$, using the chain rule:

$$\frac{\partial O}{\partial \theta} = \frac{\partial O}{\partial H_1}\frac{\partial H_1}{\partial \theta} + \frac{\partial O}{\partial H_2}\frac{\partial H_2}{\partial \theta} \tag{1.21}$$

$$= \frac{\partial O}{\partial H_1} \cdot (\frac{\partial H_1}{\partial G_1}\frac{\partial G_1}{\partial \theta} + \frac{\partial H_1}{\partial G_2}\frac{\partial G_2}{\partial \theta}) + \frac{\partial O}{\partial H_2} \cdot (\frac{\partial H_2}{\partial G_1}\frac{\partial G_1}{\partial \theta} + \frac{\partial H_2}{\partial G_2}\frac{\partial G_2}{\partial \theta}) \tag{1.22}$$

In this way the gradient requires the computation of derivatives of subsequent layers (Goodfellow, Bengio, and Courville 2016). This way the computation time is greatly reduced, since the number of times derivatives need to be computed is reduced.

Chapter 2

Reinforcement Learning

2.1 Value Based Methods

In this approach, the agent's objective is to learn to predict the value of the current state of the environment which is then used to take an action. Once an optimal mapping of state to values is found, the policy can then be inferred. This is done by looking at states adjacent to the current state and then greedily choosing the state with the highest value (Busoniu et al. 2010). The focus is then to obtaining this accurate mapping of state to value, known as the optimal value function, $V(s)$ (Sutton and Barto 2017).

Value-based methods seek to define a function that describes the desirability or value of different states. The optimal policy would then be to move from the current state to the most desirable state (Busoniu et al. 2010). The basis of valuation methods is derived from the Bellman equation. Bell-

man devised a way to represent dynamic programming iteratively (Sutton and Barto 2017). Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (ibid.). The Bellman equation writes the value of a decision problem at a certain point in time in terms of the payoff from some initial choices and the value of the remaining decision problem that results from those initial choices (Reviews 2016). The Bellman equation breaks up the large dynamic programming problem into smaller sub-problems (ibid.). A Bellman equation can be used to rewrite the value function in recursive form so that it can be solved sequentially.

$$V(s) = R(s) + \alpha \cdot \sum_{s'} (T(s'|s, \pi(s)) \cdot V(s')) \tag{2.1}$$

The $R(s)$ in this case is the reward the agent receives for reaching the state. There are different value based reinforcement learning methods, Value iteration and Q-learning are the most used methods. Value iteration assigns values to states. This value is proportional to the reward for that state and next state value (Busoniu et al. 2010). The values for each state are found by iteratively exploring all states until the values have converged. The optimal behaviour for the agent is then to take the action which leads to the highest value state (ibid.). Value iteration can be seen as model-based learning since it attempts to learn the environment and not necessarily the actions to take. Fortunately, the agent is able to interact with the environ-

ment and experience sample data from a sub-optimal policy. As the agent explores the environment it gathers state, action, reward, new state data tuples, (s, a, r, s_0) (Sutton and Barto 2017).

The temporal difference can be defined as the prediction error between the current and previous iterations of the value function (ibid.). We introduce the temporal difference in the value function as follows:

$$V^{i+1}(s) \leftarrow V^i(s)\alpha \cdot (\underbrace{R(s) + \gamma \cdot V(s')}_{A} - V^i(s)) \tag{2.2}$$

which now contains the temporal difference. A is termed the learned value, the entire bracketed term is the temporal difference error, R is the immediate reward, γ is the term that governs the impact of future values, α is the learning rate and the superscript i, is the iteration number (ibid.). The difference between the learned value and the initial $V(s_0)$ term is known as the temporal difference. A zero temporal difference would then result in no update to the value function. This happens when the value function has converged to the optimal policy (ibid.). We pick the action that maximises the sum of the immediate reward, $R(s_0)$, and the sum of the values of all adjacent state, $V(s_1)$, weighed by γ (Watkins and Dayan 1992). It is important to note that the final state is not determined by 2.2, but instead by the final reward. As the values for each state converge to optimal values, the final reward propagates through all the states by the $V(s_1)$ value. The algorithm below shows the method for value iteration.

Algorithm 2.1 Value Iteration

1: V ← Initialise Values for all states
2: **while** V not converged **do**
3: Observe state
4: Perform action under policy π
5: Receive reward and new state
6: Update V for current state using $V(s_1) \leftarrow \alpha \cdot (R(s_0) + \gamma \cdot V(s_1) - V(s_0))$
7: **end while**

2.2 Q-Learning

Q-learning is similar to value iteration, but instead of values for each state, there are state-action values. It tends to be a more robust and intuitive method for reinforcement learning. It is also the method which is widely used in the field of reinforcement learning.

We defined a value function 2.2, where $V(s)$ and $R(s, a)$ are the value and reward values respectively. This value function does not take into account the actions the agent makes. In order to make use of information we introduce the action-value function, Q:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (R(s_t, a_t) + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (2.3)$$

This function assigns a value to state-action pairs (Sutton and Barto 2017; Arulkumaran et al. 2017). It is important to note that the action the agent performed in the current time step, a_t, as well as the action the agent wants to take in the next time step, a_{t+1}, is needed. The agent chooses these

actions under its current policy, thus, this is an on-policy approach (Sutton and Barto 2017). This is different to an off-policy approach, where future actions are not selected under the current policy of the agent (Hasselt, Guez, and Silver 2015).

The advantage of an on-policy approach is the fact that the agent learns a value function that considers future actions (Sutton and Barto 2017). The disadvantage is that the agent only learns from its current policy, this does not make for very effective learning. The agent would then need more time to train and converge to an optimal policy. This is due to the fact that the agent needs to have future Q-values to converge before it can result in earlier Q-values converging, which is why an off-policy method is favourable. Using an off-policy approach, the agent can sample data (to train on) while the agent explores the environment. This also implies that direct access to the environment is not necessary. The agent simply needs observations from the environment to learn. To incorporate the off-policy method the Q-function is updated to:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (R(s_t, a_t) + \gamma \cdot \max_{Q} \cdot Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (2.4)$$

The future policy is now to choose the action that leads to the most desirable value (Oliehoek 2012). This action does not need to be the next performed action (Watkins and Dayan 1992). Since the Q-function is defined recursively, it does not account for a terminal state. In order to begin updating Q-values,

the value for the terminal state must be defined. Thus, to adjust for the ending of an episode in the environment, the Q-function becomes:

$$Q(s_t, a_t) \leftarrow \begin{cases} R(s_t) & A \\ Q(s_t, a_t) + \alpha \cdot (R(s_t, a_t) + \gamma \cdot \max_Q \cdot Q(s_{t+1}, a) - Q(s_t, a_t)) & else \end{cases} \tag{2.5}$$

Where A is the condition: terminal state (Mnih et al. 2015). The following is a formal implementation of Q-learning (Sutton and Barto 2017):

Algorithm 2.2 Q-learning

```
Q ← Initialise Q-Values for all state action pairs
Define γ and α
for Episode in training do
    Reset environment
    while State is not terminal do
        Observe state
        Perform epsilon greedy action
        Receive reward and new state
        Update Q for current state and action
    end while
end for
```

In general Q-learning algorithms, the Q-values are often stored in a matrix (Watkins and Dayan 1992). The rows of the matrix would then represent the states and the columns represent the actions that can be taken. Thus, the Q update pseudo-code is the following:

Algorithm 2.3 Q Update

1: **function** Q UPDATE(Q-values, state, reward, next state, terminal, gamma, alpha)
2: **if** terminal **then**
3: Q-values $\leftarrow$ reward
4: **else**
5: Q-values $\leftarrow Q(s_t, a_t) + \alpha \cdot (R(s_t, a_t) + \gamma \cdot \max_Q \cdot Q(s_{t+1}, a) - Q(s_t, a_t))$
6: **end if**
7: **end function**

This method generally works in a wide variety of problems. However, as the state and action space grows, the matrix becomes larger. The matrix then becomes computationally expensive to converge to the optimal policy. This is to allow the Q-values to propagate through the matrix and to allow convergence. This implementation does, however, not allow the agent to generalise well. A never before seen state would result in the agent taking a near random action. This problem, coupled with the issues surrounding high dimensional state and action spaces, leads to the need for a more advanced algorithm.

2.2.1 Exploration vs Exploitation

One of the common challenges that arise in reinforcement learning is the trade-off between exploration and exploitation (Sutton and Barto 2017). In order to obtain a high reward, the agent must take actions that it has tried, during training, that produces the most effective reward. This action does not necessarily yield the highest immediate reward, but leads to a higher total

reward for the entire episode (Busoniu et al. 2010). With this structure, we can also allow for ϵ-greedy decision making:

Algorithm 2.4 ϵ-greedy selection

```
1: function ε-GREEDY(Q-values, state, epsilon)
2:     if random() < epsilon then
3:         action ← random action
4:     else
5:         action ← action with highest Q value given the state
6:     end if
7: end function
```

The algorithm above represents the most common tool for exploration or exploitation. Intuitively it is the probability an agent explores a random action or exploits the current best action (Sutton and Barto 2017). A random number is produced and if this number is below a threshold, ϵ, the agent must explore a random action. Otherwise, the agent exploits the best action currently.

2.3 Policy Gradient Methods

This is a relatively new field in reinforcement learning with the advancement of neural networks in the field of machine learning. In the previous value based methods the policy needs to be inferred from the values,$(V_\pi(s))$, where the values are dependant on the policy function $\pi(s) = a$. In this way, the policy can be inferred using the value function.

This approach centres around defining a set of parameters, θ that can be

used to parameterise the policy function, which is of the form:

$$\pi_\theta(s) = P(a|s, \theta) \tag{2.6}$$

The policy is now a probability distribution over possible actions (Sutton and Barto 2017). In layman's terms, it is the probability that action a is taken given that the environment is in state s with parameters θ (Oliehoek 2012). There are many ways to model this function, most popular is neural networks. In this case, θ represents the weights of the network. This can now be treated as an optimisation problem and to have θ converge, such that it would result in the optimal policy (Lillicrap et al. 2015). The problem is to define how the parameters or weights of the function approximator of the neural network are optimised. There are many algorithms that seek to optimise the policy the agent acts over, but not all of these were investigated. If a similar approach to optimisation as equation 1.20 is used, assuming the weights of the neural network as the parameters of policy function.

$$\theta' = \theta + \gamma \cdot \nabla_\theta \hat{J(\theta)} \tag{2.7}$$

In this case, $\hat{J(\theta)}$ is a measure of policy performance. The performance of the policy is then maximised through equation 2.7. In this case, $\nabla_\theta \hat{J(\theta)}$ is a gradient approximation, with respect to arguments θ, of the performance measure (Sutton and Barto 2017). This is a common concept shared by all policy methods. This method works particularly well in large or continuous

state and action spaces (Busoniu et al. 2010). The disadvantage is that this method does not always converge and can take longer to converge.

A popular and intuitive policy gradient method is the REINFORCE algorithm, also known as the Monte-Carlo method. Using the gradient ascent equation above, we define the update function as:

$$\theta' = \theta + \gamma \cdot G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{2.8}$$

$$= \theta + \gamma \cdot G_t \nabla_\theta \ln \pi(A_t|S_t,\theta) \tag{2.9}$$

by the identity, (2.10)

$$\nabla \ln x = \frac{\nabla x}{x} \tag{2.11}$$

The learning increment is proportional to the return G_t and the gradient of the probability of taking an action, A_t, divided by that same probability. The term $\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$, gives a direction in the parameter space, θ, which increases or decreases the probability of taking action A_t when presented with state S_t (Sutton and Barto 2017). The return G_t, determines how much to adjust the parameters by. A large positive return will increase the probability of taking the action that will result in the same return(ibid.). The converse is then true for a negative reward. The return, G_t, is the total reward which is weighted based on the time at which action, A_t, is taken. An action taken early in an episode is weighted in such a way that the return is smaller than an action at the end of an episode.

Algorithm 2.5 REINFORCE

Initialise parameters, θ
Define policy, π_θ using parameters
Define learning rate, γ
while Policy not converged **do**
    Generate data on episode using current policy
    Discount reward throughout episode, G
    for Every data couplet in episode data, G_t, S_t, A_t **do**
        $\theta \leftarrow \theta + \gamma \cdot G_t \nabla_\theta \ln(A_t|S_t)$
    end for
end while

In this algorithm, a policy is initialised with random parameters. An episode is then run under the policy, of which the state and action probability action vectors are stored (Busoniu et al. 2010; Sutton and Barto 2017). The total reward for the episode is then propagated through the episode data by the discount factor. The parameters of the policy are then updated using the equation 2.8. This process is then repeated until the policy reaches optimal parameters.

2.4 Deep Q-Learning

Deep Q-learning is the technique, whereby a deep neural network is used to approximate the action-value function from the Q-learning technique (Mnih et al. 2015). Traditional Q-learning does not perform well in problems with large state dimensions since they do not generalise well (Arulkumaran et al. 2017).

This is due to the use of matrices to encode the Q-values, but the use of a

function approximator, like a neural network, can help overcome these issues. The use of neural networks to approximate the Q-function now becomes:

$$Q(s_t) = NN(s_t, \theta) \tag{2.12}$$

where NN represents the function that the neural network applies (Hasselt, Guez, and Silver 2015). The policy in Q-learning (and deep Q-learning) is in most cases for discrete action spaces. The actions are labelled as discrete values, like integers or letters. These labels represent each action available to the agent e.g. action 0 can is moving forward etc. This becomes useful since action-value pairs can be stored in a position in the vector, y. For example $y = 0$ would be the value associated with taking action 0 in the current state. The structure of the neural network is then to take the state as input and to predict the action-value vector (Mnih et al. 2015). This is done by having n neurons as the input layer, where n is the dimensions of the state vector and m neurons as the output layer of the neural network, where m is the dimensions of the action space. In order to train the network, the NN produces Q values for each action in the current state (Busoniu et al. 2010). The Q value for the action taken is then updated similarly to 2.5, but using the neural network as:

$$y_{a_t} := r_t + \gamma \cdot \max_a y' \tag{2.13}$$

We can now express this in terms of a neural network update function,

$$NN(s_t, \theta^{i+1}) := r_t + \gamma \cdot \max_a NN(s_{t+1}, \theta^i) \tag{2.14}$$

The α term, from equation 2.5, is missing in this equation since the neural network has its own inherent learning rate which acts as a α term. θ is the parameters/weights of the neural network and updating the weights, θ^{i+1}, depends on the previous weights, θ^i (Li 2017).

If the environment gives a state s_t to the agent, the agent only has two actions that it can perform (action 0 and action 1). Before training $\alpha = 1$ and $\gamma = 0.9$ are defined. The state is then given to the neural network and a action-value vector $y = [0.7, 0.2]$ is produced. Under the current policy of choosing the action with the largest Q value, the agent decides on $a_t = 0$. The agent takes action $a_t = 0$ at the current state s_t, and the environment then responds with s_{t+1} and $r_t = 0.75$. The neural network then gives a vector for s_{t+1} to obtain $y' = [0.5, 0.6]$. We can now substitute values into the above equations:

$$\begin{aligned} NN(s_t, \theta^{i+1}) &= r_t + \gamma \cdot \max_a NN(s_{t+1}, \theta^i) && (2.15)\\ &= (0.75) + (0.9) \cdot \max_a [0.5, 0.6] && (2.16)\\ &= (0.75) + (0.9) \cdot (0.6) && (2.17)\\ &= 1.29 && (2.18) \end{aligned}$$

We then substitute this value, into the vector generated by the neural network, $y = [0.7, 0.2]$, to give $\hat{y} = [1.29, 0.2]$. The neural network then uses this new vector and the loss function to update its parameters, θ, until it has reached an optimal policy. In this way performance is improved, since one forward pass of the neural network produces the action-value pairs that need to be updated. Recall the Q-function 2.5 and the algorithm 2.2, when using deep Q-Learning the algorithm remains roughly the same. The real difference becomes obvious in updating the Q-function.

Algorithm 2.6 Deep Q Update

```
function Q UPDATE(Q NN, state, reward, next state, terminal, gamma)
    Predict Q-values for the current state using Q network
    if terminal then
        Q-values ← reward
    else
        maxQ ← max_Q(Q_NN.predict(next_state))
        Q-values ← Q(s_t, a_t) + γ · max_a Q
    end if
end function
```

This approach offers a way to approximate functions very well without the need for a matrix to store(Busoniu et al. 2010). In this way, only the weights of the neural network need to be optimised. The Q-function now becomes generalised, and able to give somewhat plausible actions to unseen states (ibid.).

2.4.1 Experience replay

Q-Learning and other deep reinforcement learning methods are said to be inefficient since the state action pair is used to adjust the weights of the neural network but then become useless. This is not good practice since, with the exploration, the agent experiences a state action pair but, due to exploitation, this experience becomes rare. These rare events mean that the neural network does not have an optimal policy once it is exposed to the state again, thus the introduction of *Experience replay* (Lin 1992). Experience replay repeatedly exposes the agent to past experiences to the agents learning algorithm, as if the agent experienced the state over and over again. The advantage of this is that training the agent requires fewer iterations and the agent would be exposed to these rare events and update its parameters accordingly (ibid.). While training the agent/neural network if a state has not been seen for a while, typically, the network will forget the corresponding Q-values and need to be exposed to that state again, in order to re-learn the corresponding Q-values. A condition for experience replay to be applicable is that the basic rules of the environment do not change over training. Otherwise, past experiences are not applicable to learn and can be misleading to the agent. Consider the example, the agent has two actions that it can make, A and B. The agent is currently exploiting action A and is not choosing action B. The agent thus forgets the Q-values for action B, but because of experience replay the agent can re-experience actions where it considers B and thus

refresh what it has learned.

Chapter 3

Experimental Investigation

3.1 Purpose of Study

The main objective of this thesis is to investigate an alternative to the current method astronomers use to manage observing the night sky with a telescope. Currently, a singular telescope requires an astronomer to determine which objects are most important to observe throughout the night. The astronomer does this by receiving a list of projects and the requirements for those objects, the clarity of the observation and type of observation, and determining the best object based on the conditions of the night. The schedule is determined by the operator, that takes into account the weather, visibility and project requirements. The problem comes with the introduction of future telescopes like LSST (Large Synoptic Survey Telescope) which produces millions of new potential objects to observe. The objective is to further observe these objects

and adapt to any new objects that have appeared if it is an important object. These new objects, as well as the already existing/discovered objects will make it difficult to choose which objects to observe further with the limited telescopes in existence.

The weather can affect the choices astronomers make during an observation night. The overall weather is somewhat predictable on average, an automated system could then learn this predictable pattern and adjust such that the return from observations is maximised. Although, in some cases, random cloud cover can cause the astronomer to adjust the observation plan. An object that is covered by clouds or any other weather effects can render the observation useless, this can be seen in (Helminiak 2009). A replacement for the astronomer would have to be able to adapt to these weather conditions. The recent observation of gravitational waves (B. P. Abbott et al. 2016), drove a need for multiple observations of the same event with different types of telescopes. The next observation of gravitational waves was then observed by many different telescopes around the world (B. P. Abbott et al. 2019; Evans et al. 2017). By introducing some sort of automated system one could have managed the telescopes around the world better, it could have possibly reduced the delay in reaction and reduced the duplicity of observations. The recent advancements in reinforcement learning could be a way to alleviate these issues. In order to get to a collaborative system, research needs to be applied to a single system, such that one reinforcement learning agent can take the place of an astronomer. The next step would then be to have a

reinforcement learning agent that can control multiple telescopes and work in collaboration.

There are telescopes which are fully automated and only require a queue of objects to observe but these telescopes cannot interpret transient events like (B. P. Abbott et al. 2016). There are also manual telescopes which require astronomers to operated but the trade off is that astronomer can only evaluate choices between a small number of objects with little variables that can be considered. The purpose of this study is to investigate a method which can be autonomous as well as being able to deal with unseen circumstances (weather and transient objects) and also take in to account many variables. The aim is to combine the queuing system of automated telescopes with the "intuition" and freedom of an astronomer.

3.2 Problem Setup

We found that the best way to approach the problem was from a scheduling perspective. The night sky viewing would be separated into even distinct time slots and the agent would have to choose an object to observe in each time slot. It can be difficult for computer programs to compute analytic solutions, so by dividing the problem in to distinct parts the program can solve the problem numerically, similar to the way integrals are computed. The objects would be available for observation during different intervals of the night. The agent would then have to choose the best object based on

the state. The value for each object/action in a state is defined by a reward function. The object that has the highest reward is not necessarily the highest value object/action since the Q-value is a combination of the current reward as well as the future rewards.

3.3 Telescope Theory

In order to understand the study one needs to learn some background on how telescopes work. In the field of astronomy, there are many different telescopes with different types of detectors. For the purpose of this study, we will rather be focusing on the detectors, how the telescope returns an image/observation from a source of light, rather than the actual mechanics of a telescope. For a general understanding, modern telescopes use a mirror to reflect light or a medium (usually glass) to refract light to a central focal point. At this focal point lies a detector of some sort, which uses the light to produce a signal, which is then converted into an image/observation.

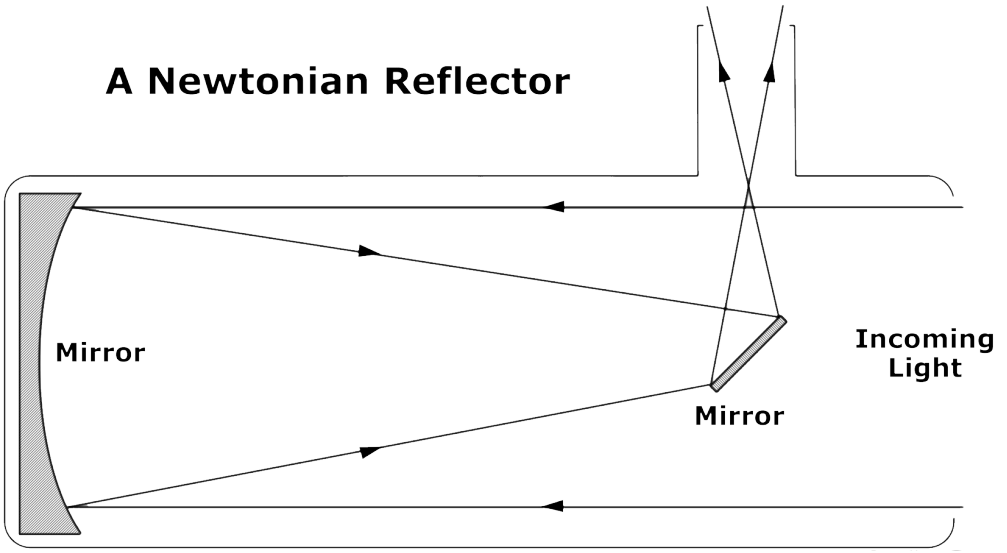

Figure 3.1: A representation of how light is reflected in a Newtonian reflector telescope. The light is reflected off a primary mirror towards the secondary mirror. This then further reflects the light to an eyepiece, situated at the focal point of the light (*Reflecting Telescopes* n.d.).

Photon detectors produce a signal that depends on an individual photon altering the quantum-mechanical state of one or more detector electrons (Chromey 2010). A change in electron energy in a photoconductor or photodiode can produce a change in the macroscopic electrical properties like conductivity, voltage or current(ibid.). These changes lead to a signal being produced. Photon detectors are particularly suited to shorter wavelengths (infrared and shorter), where the energies of individual photons are large compared to the thermal energies of the electrons in the detector (ibid.). In typical astronomy, objects are valued higher or lower than others based on the type of object, e.g. a galaxy might have more scientific value than a star, this is mostly as a result of the direction of popular research. This is usually

based on the current direction of astronomy and cosmology. This "value" allows us to rank the objects in the night sky based on their importance to the field of astronomy. Another value to observation is the quality of the observation. When working with telescopes, this is known as the signal to noise ratio (SNR) (Schroeder 2000). An object of higher brightness will produce more "signals" than a fainter object in the same time interval. This means that a brighter object will take less time to reach a desired SNR than a fainter object, since the noise is normally constant since it is as a result of equipment and environment. The typical format of this SNR function is,

$$SNR = \sqrt{n}, \tag{3.1}$$

where n is the number of photons per time (ibid.). But n is proportional to the exposure time t. So the function can be written as,

$$SNR \approx \sqrt{t}. \tag{3.2}$$

Intuitively, the SNR is a ratio of true signal to the noise. Noise can be seen as photons that are observed, which are not from the source object. Many times this is instrumental noise which comes from imperfect components in the telescope. Other times it can be more complex, like the way the atmosphere absorbs, emits and scatters light. A high SNR is an indication of a high-quality observation. Inversely, a low SNR indicates the observation will be mostly noise. A noisy observation becomes difficult to derive insight

from since the statistical inference will have large uncertainties.

3.4 Reward Function

An astronomer would know the brightness of the objects to observe before and as a result know the best SNR for the objects. The objective of the reinforcement learning agent is instead to learn this best SNR for each object and the relationship to the objects brightness through exploration of the environment. In order to do this one would need to define a reward function that rewards the agent for choosing the optimal observation time of an object and penalises any less optimal behaviour. The most challenging aspect of reinforcement learning is defining a reward function that yields the desired results.

The issue with the function 3.2, is that the reward would only be given once the entire night-sky viewing is over. With the slower learning policy gradient reinforcement learning, it would not be a problem, but Q-learning seemed to have some issues with this approach. In this case a more immediate reward would be best, i.e. a reward after an action is taken.

The reward function would then have to be a function whose cumulative function was a square root function. Initially, this led to the derivative of the function, which is $\frac{1}{2\sqrt{n}}$, but this proved difficult when trying to stretch the function, i.e. change the rate of 'decay'. This way some objects would need different observation time to reach the same reward point. The decision was

to then use an exponential decay function, which has a factor that changes the decay rate. The rate of decay would then be related to how long the object would need to be observed. In typical astronomy the brighter an object, the larger the photons per time step, so a bright object would reach a particular SNR in a shorter amount of time than a dim object. This means that the decay rate can be said to be related to the flux of an object, i.e. the brighter an object the quicker the exponential decay function approaches zero and the less time needed to receive the total reward.

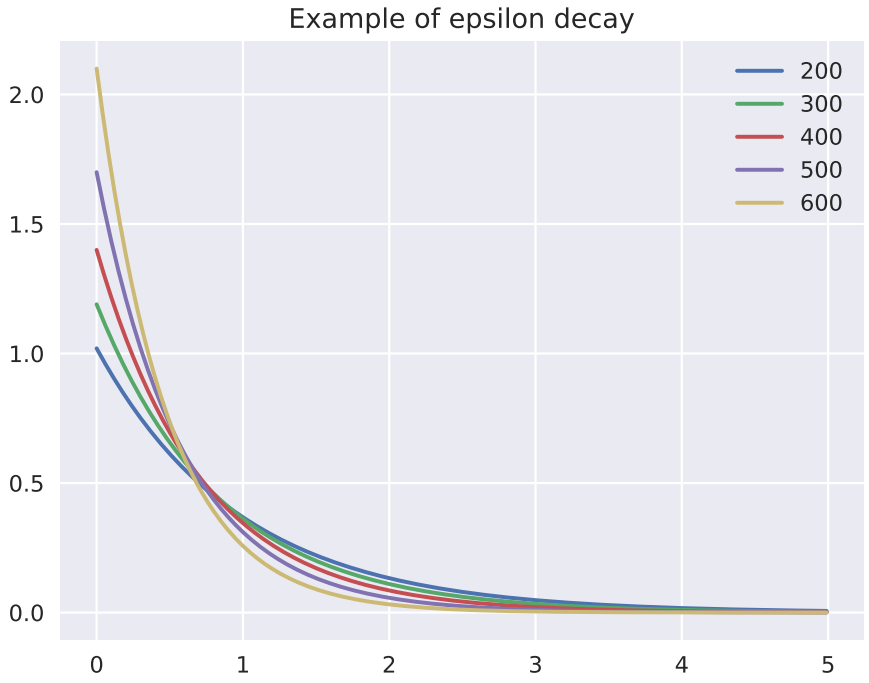

Figure 3.2: An example of epsilon decay for different decay rates. Different decay rates correspond to what can be considered a flux of an object. An object with a larger flux decays at a faster rate than an object with a smaller flux

The figure above is an example of the exponential decay for different "fluxes", the values are arbitrary. An object with a flux of 600 needs to be observed for roughly 2.4 seconds to get most of the reward, while on the other spectrum, an object with a flux of 200 needs to be observed for roughly 4 seconds. This is roughly how telescopes work, the observation time needed is inversely proportional to the flux of the object, more specifically $t \propto \frac{1}{\sqrt{flux}}$. This is an indication of the reward the agent receives as it observes an object,

the final reward is the sum of all the rewards received over time.

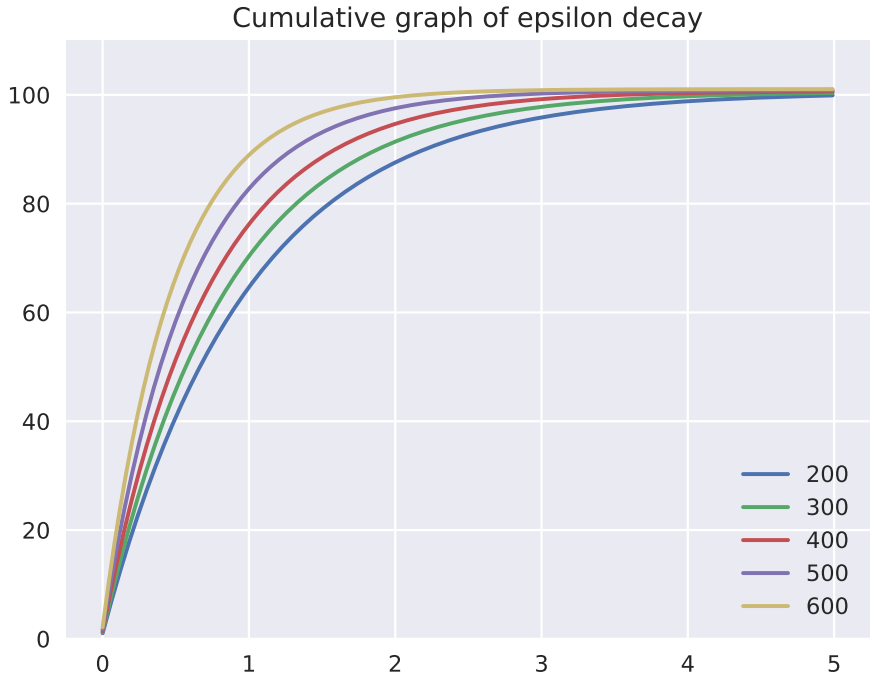

Figure 3.3: An example of the cumulative function of the exponential decay function. It shows that even though the functions decay at different rates, the cumulative functions converge to the same value.

The figure 3.3 shows the cumulative reward over time, it is an indication of the final reward for each flux. There are small differences between each function's value towards the end, but it will be negligible.

The final reward function is an exponential decay function, where the flux determines the rate of decay multiplied by some factor, which is related to the class of the object. Since the epsilon decay function continuously decays and never reaches 0, the agent could essentially always receive a positive

reward. In order to avoid this, a discontinuity is introduced whereby the reward becomes a penalty at 0.01. This occurs before the type constant is multiplied. This way the agent learns to avoid continuously observing the same object for little to no reward. The stopping criteria are not necessarily the correct amount of time to observe, it merely serves to aid the agent in the learning process by removing extreme observation times.

$$R(t) = \begin{cases} T(F \times e^{-Ft}) & F \times e^{-Ft} > 0.01 \\ -1 & F \times e^{-Ft} <= 0.01 \end{cases} \tag{3.3}$$

Where in this case,

$T \propto$ The class of the object

$F \propto$ The flux of the object

and $R(t)$ is the reward received when an object is observed for t time.

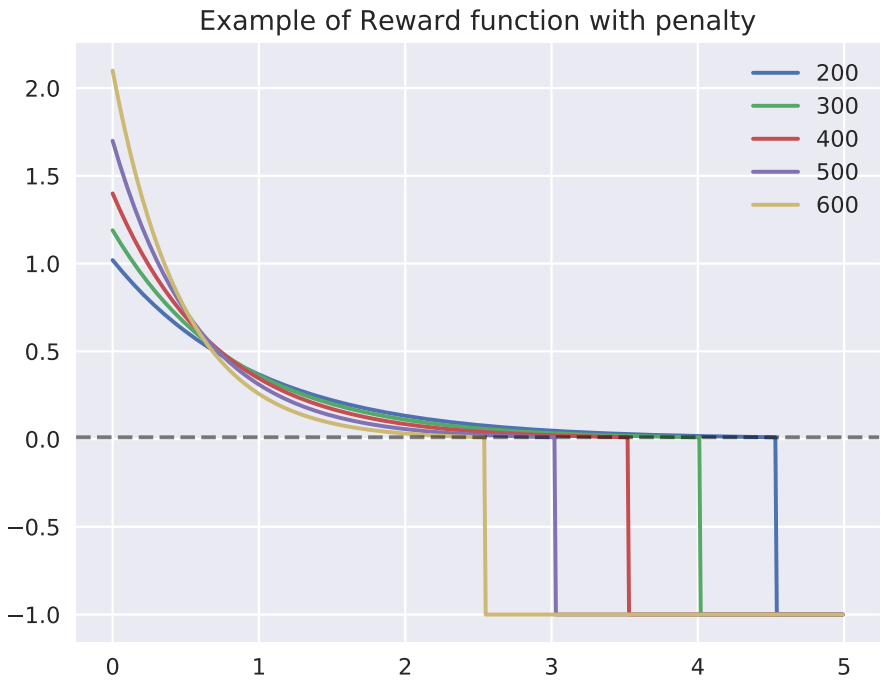

Figure 3.4: Example of final reward function with penalty, without the class multiplier factor. This represents different fluxes but with a fixed class. The dashed line represents the point where the reward switches to a penalty.

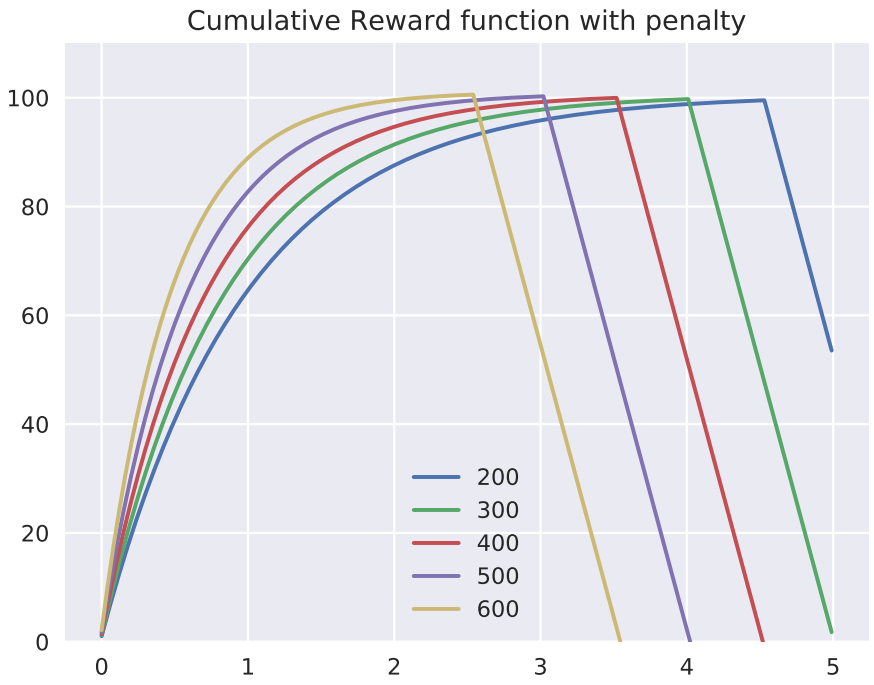

Figure 3.5: The final cumulative reward function with the penalty, without the class multiplier factor. This represents different fluxes but a fixed type of object. One can see that the penalty switch is roughly at the same value, this is a good indication for the optimal time to observe an object.

3.5 Simulation

In order to use a reinforcement learning algorithm, one needs to take in a state and map that to action. The choice of what to feed in as the state is essentially what the algorithm is learning.

The environment is set up in such a way, that it is almost like a real-world simulation. The environment is made up of multiple objects of different type-

s/class. The episode is divided into time steps and the objective of the agent is to choose an object for each time step. In this way, the agent attempts to choose the best object to observe and by dividing the episode into time steps, it also decides how long to observe an object. It is important to note that the agent might not choose the best object in a particular state, since it may be choosing an object in order to achieve an even better state, i.e maximise the returned reward. Another issue that the agent will have the freedom to switch between objects, this behaviour (while allowed) is not optimal and the agent would have to be steered away from this behaviour.

Each object has a particular set of attributes that form part of the state. These attributes, with the action the agent takes for the current state, determine the reward the agent receives. These attributes are the fluxes of the object, which determine the slope of the exponential decay, the total time the object has been observed, which determines the time for the exponential decay function, the object type (which translates to the constant), which is multiplied with the exponential decay, the time the object is available for in the current state, in some cases to avoid switching between objects the previous action is also included in the state. Some parts in the state are constant throughout the episode and do not add differences between states, which some may argue are unnecessary, but it is included in hopes that the neural network learns the relationships between flux, type and observation time to reward received.

3.6 Algorithm

The training of the agent requires many steps, the pseudo-code can be seen in the appendices. Algorithm A.1 is the function that handles the setting up the environment. The static components of each object are defined here i.e. flux, class type. This also handles the random availability and time of each object for the duration of the game (the time is only random when the environment is first started).

Algorithm A.2, updates the time observed for the object that was observed by adding a time step. This is important in determining the reward for the action that the agent receives. It most importantly does not update the time for an object that is not available for observation.

Algorithm, A.3, determines the reward for the action taken, the reward function is based on the function explained above. The other processes, that do not necessarily require a break down of the algorithm in pseudo-code, are explained below:

- **NN.initialize:** This sets up the configuration of the neural network with a particular learning rate, as well as the number of nodes and layers.
- **Env.reset:** This resets the environment in such a way that the dynamic parts of the state, the time observed, is set to zero for each

object. It also sets the time available for each object and is set to 0 at the beginning state of the episode.

- **epsilon_action:** As with algorithm 2.4, this function generates a random number, if the number is lower than epsilon the action is random, if not the action becomes the best action the agent chooses.

- **update_Q:** this takes the current Q values for the agent and replaces a new Q value at the action that was taken in this state. This new Q value is determined using the Q function, similar to algorithm 2.6.

- **Store_data:** This stores the state and Q value vector for training purposes. There is also a store process for all the data over all the episodes.

- **train_episode:** this takes the values stored through the episode and does one batch of training over the whole episode. One training run means determining the Q values for each state and then adjusting weights using backpropagation and some gradient descent method.

- **replay_memory:** This process takes a random subset of all the data currently and trains the model for multiple epochs. Generally, the subset is chosen from all the data available, but we have adjusted this to take a subset that gets incrementally closer to the most recent data. We have found that using the total data set can lead to the model adjusting weights in a negative way. This happens early on in the

training when an action for a state has not yet been properly learned, but as training happens the model adjusts and learns the relationship. If the model were to consider this earlier action state data point, it would (although in a small increment) unlearn the relationship. The model instead, in the beginning, considers all the data but as training progresses the data considered becomes a smaller and smaller subset of the full data set, this subset is always the most recent data points.

The epsilon factor, which can be seen as the probability to explore a random action or exploit the action with the largest Q-value, is not constant. The epsilon factor starts at a high value and decays throughout the training of the agent. The function is a epsilon decay function of the form

$$\epsilon(t) = \begin{cases} e^{-\frac{t}{0.4 \times n}} & t < 0.9 \times n \\ 0 & t >= 0.9 \times n \end{cases} \tag{3.4}$$

Where in this case,

t = The episode number

n = The total number of episodes

This is then a function which starts at an epsilon and becomes smaller as the episode number becomes larger. This encourages the agent to have a high exploration rate and low exploitation at the beginning of training and vice versa at the end of training. Below is a graphical representation of the function,3.4.

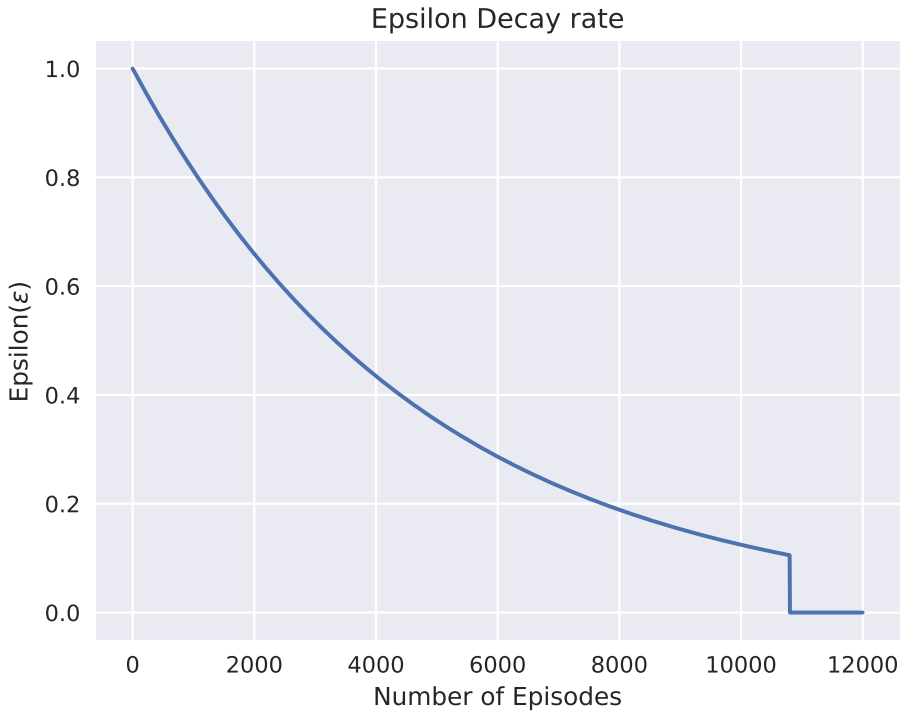

Figure 3.6: Epsilon function over the number of episodes. This shows how the probability of taking a random action decays as training progresses. At the beginning of training an agent, the chance of random actions is high and vice versa at the end of training. The step indicates where the exploration is switched off and validation has begun

3.7 Improvements

The algorithm and simulation discussed so far showed promise, but due to some issues, there was an improvement that needed to be made. In the outcomes section, table 4.1, one may notice that the agent switches between objects. This is in order to receive the most reward out of the episode. This

is due to the reward function for two similar objects reaching a similar value. Once the agent observes object A, object B's reward is larger, thus in the next time-step the agent will observe object B, this then repeats until a new more valuable object becomes available.

In order to combat this, an adjustment was made to the reward function and time update algorithm. This, in turn, led to an update to the state that the agent received. The reward function was adjusted, such that if an agent decided to switch to another object that would result in a positive reward, it instead receives a zero reward.

The time update algorithm was adjusted in a similar way, such that the observation time is not updated if the previous action is not the same as the current action. The previous action would then need to be incorporated into the state, such that the agent would be able to differentiate between similar states with different previous action.

3.8 Network architecture

A particular issue arises when working with neural networks. The issue of the architecture of the network i.e. how many layers, how many nodes. In some cases, it can be said that the architecture could be related to the number of states. This could be that certain weights are activated when presented with a particular state. The method for choosing the architecture was roughly intuition based on the dimensionality of the state space, but could also be

interpreted as random. In this study, the number of nodes chosen was a constant of 500 and the number of layers was then increased based on how the network performed(trained) on a couple of training episodes.

3.9 Previous algorithms

In order to get to a viable environment simpler simulations of the problem were first investigated before reaching the simulation discussed below.

The first of these simulations was an environment in which the agent was given five actions, which represented five objects and had five different constant rewards. The environment consisted of three-time steps, the agent would then have to take an action at each time-step but was not able to take an action that it had taken in a previous time-step. This might seem like a trivial environment but it helped to understand the Q-Learning algorithm and validate that it could work with a scheduling approach.

The approach above seemed to have promise, but it lacked automation. This was due to the fact that once an object was observed, the Q value was removed when choosing a new action ($argmax(Q(s_t))$. A new approach was then proposed, whereby once an object is observed, the reward for that object is decreased. This was very similar to the approach in the final algorithm but instead of a function calculating the reward, the reward for each object would decrease by some constant after being observed.

In the next iteration, a weather effect was also introduced with the schedul-

ing approach. During an episode, a random object was "switched off" and no longer available for observation. If the agent observed the object while not available, it would receive a penalty. The weather effect was introduced randomly throughout training. The agent was able to adapt in these random conditions and able to receive the maximum reward in these episodes.

Once the scheduling approach seemed viable, we investigated giving the agent a set amount of time. The agent then has to take actions to divide the time between rewards and maximise the reward. In this case, the reward function was either a sigmoid or square root function, similar to the signal to noise ratio 3.1, and was given at the end of an episode. This approach was unpredictable and did not behave the way we intended. This approach could have worked well with the Policy Gradient method 2.3.

Policy Gradient methods were investigated for the final simulations but seemed to take much longer to train and did not give promising results. Further investigations could have yielded better results but would have required a completely new simulation of the environment and a new reward function.

Chapter 4

Experimental Outcomes

4.1 Comparison

In order to find out if the agent is a good model, other algorithms were used to compare to reinforcement learning. For the purpose of this study, the algorithms considered for comparison are the randomised and greedy search algorithm. Each of these agents would be dropped in the same simulation as stated above and use their particular algorithm to decide the actions to take.

A randomised agent would take a random action at each time step. Since this algorithm performs poorly most of the time, it is not the best algorithm to compare to in this case, thus the introduction of the greedy search algorithm.

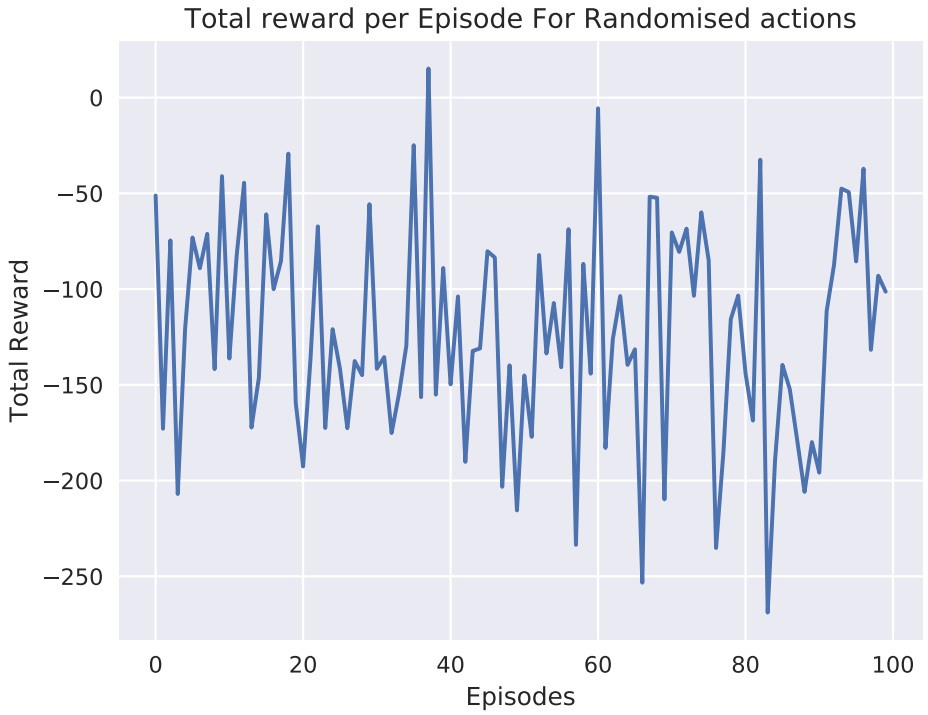

Figure 4.1: Total reward per episode for an agent acting with random actions. The total reward varies due to the random action and peaks at roughly 30. The peak is what is used as a comparison for the final algorithm

The greedy search algorithm makes a locally optimal choice at each stage, in the hopes, it will find a global optimum. This means that the algorithm picks the best solution at the moment, without regard for consequences. It picks the best immediate output but does not consider the big picture, hence it is considered greedy. In the case of the study, the greedy search alternative would be to find the object with the highest value at each time step and follow the action that observes the object with the highest value. The highest value object, in this case, would be the object with the highest class value,

of all objects available. This does seem similar to Q-learning, but recall equation 2.5 the term, $\max_Q Q(s_{t+1}, a)$, allows us to choose a Q-value with a high immediate reward, as well as a high future value. This future value propagates through all Q-values once the optimal policy has been learned. Using the same reward function previously discussed, the reward per episode can be seen in figure 4.2.

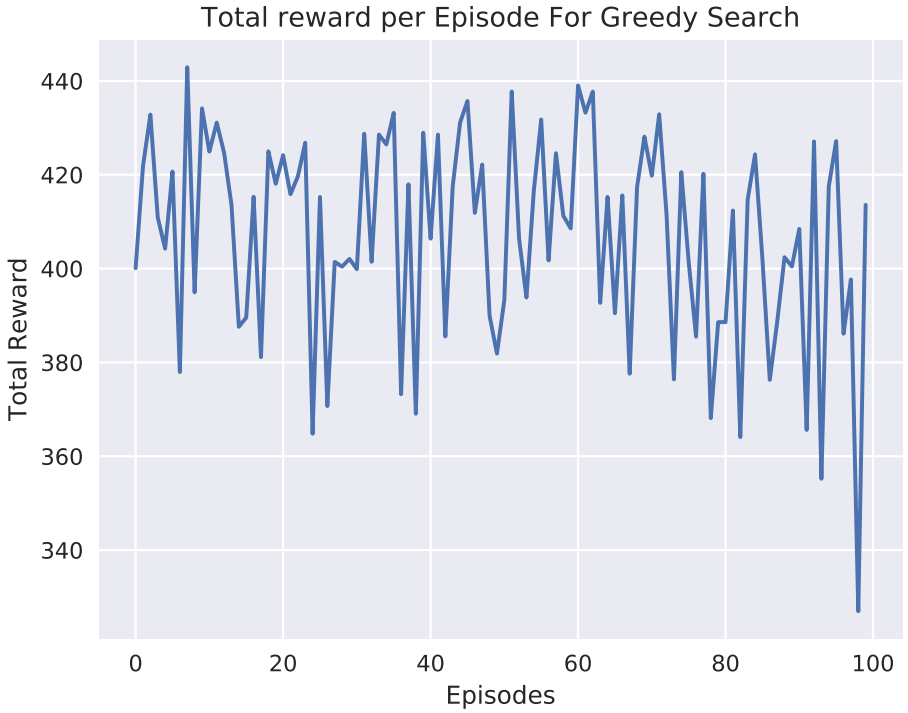

Figure 4.2: Total reward per episode for the greedy search algorithm. The variability in the total rewards is due to the algorithm choosing between two objects of the same class

The total rewards for the greedy search algorithm have a high variance

between rewards. This is due to the algorithm randomly selecting between multiple objects of the same high-value class. This then causes a difference in rewards since the actions for each episode are not the same

4.2 Deep Q-Learning Algorithm before adjustments

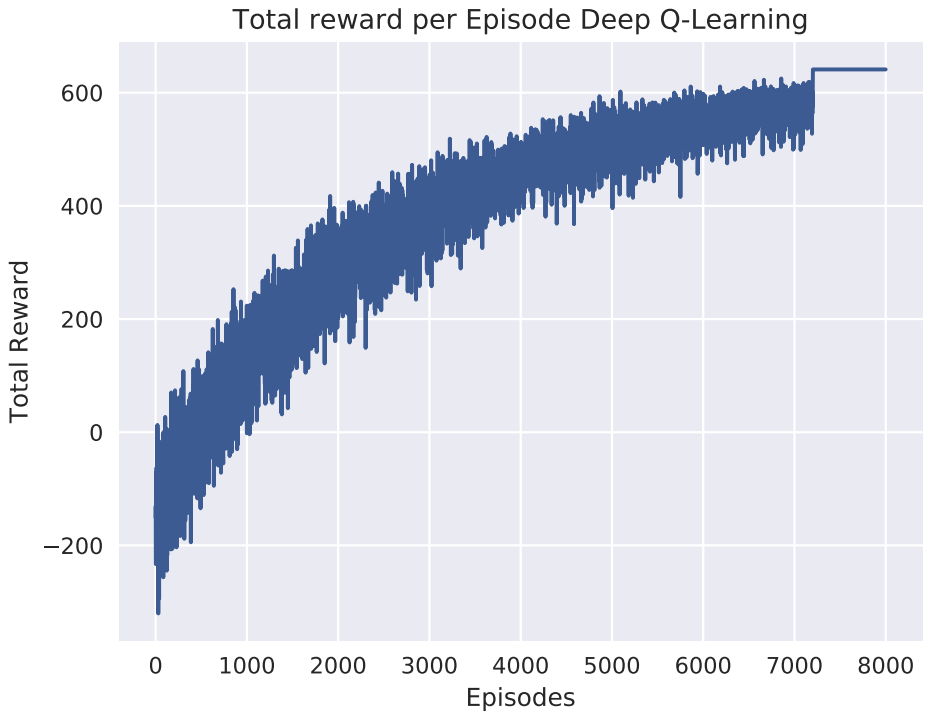

Figure 4.3: Total reward per episode for Deep Q-Learning Algorithm before adjustments. The graph shows how the algorithm steadily learns, by looking at the increase of total reward over episodes. The step corresponds to the epsilon factor switching off. At this point the agent is purely acting on the policy without any randomised action

The figure above, 4.3, shows the training process of the Deep Q-Learning Algorithm. The agent begins by having a high epsilon, ϵ, causing a high ex-

ploration. This means that the agent has a high chance of acting randomly throughout each episode. As training goes on the epsilon factor diminishes and the agent then chooses to exploit actions that have the highest Q-value. The variance of the total reward of each episode is also due to the epsilon factor. If the agent chooses actions with a high Q-value throughout the episode, but then due to epsilon, chooses a random action. This then vastly affects the state that the agent receives, the agent then adjusts and chooses actions that may not give the highest reward, but instead, the highest reward was given the new state.

The plateau, after 7200 episodes, is due to the epsilon factor, 3.6, and the training switching off. The agent is said to now be in a testing or validation phase. The agent has learned enough about the environment to choose actions that maximise the final reward.

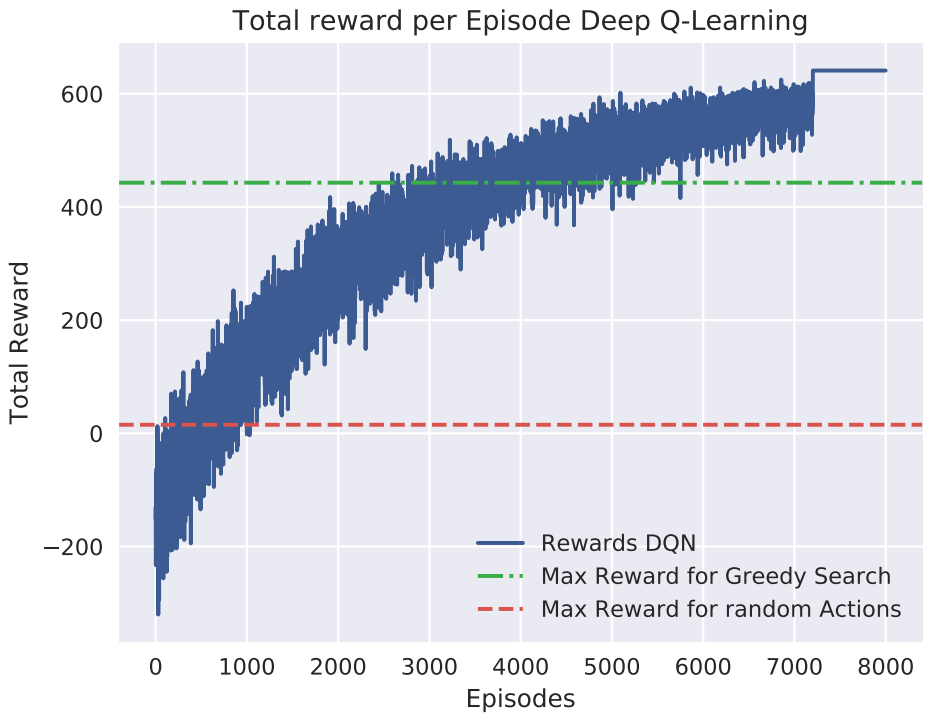

Figure 4.4: Total reward per episode for Deep Q-Learning Algorithm (before adjustments) compared to other algorithms. It shows how at initial training the agent acts comparable to a randomised agent, but gradually obtains a larger reward than the greedy search

Once we compare the Deep Q-Learning algorithm to the greedy search and random algorithm, in the same environment, we see that the Deep Q-Learning algorithm does indeed yield a higher total reward. It is important to note that the environment that each of these algorithms used, is exactly the same, i.e. the object availability, the type of objects and fluxes were all the same. This is because changing any of these factors, changes the maximum reward of the environment. Looking at figure 4.4, we could say

that the agent was able to achieve the task it was trained to do. In order to say the agent has achieved the task, we must analyse the actions it took during the different states.

	Object availability										
Time(s)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Action**
28.2	0	0	3.4	16.2	8.6	0	0	8.8	0	18.2	10
28.4	18	0	3.2	16	8.4	0	0	8.6	0	18	1
28.6	17.8	0	3	15.8	8.2	0	0	8.4	0	17.8	1
28.8	17.6	0	2.8	15.6	8	0	0	8.2	0	17.6	4
29	17.4	0	2.6	15.4	7.8	0	0	8	0	17.4	10
29.2	17.2	0	2.4	15.2	7.6	0	0	7.8	0	17.2	1
29.4	17	0	2.2	15	7.4	0	0	7.6	0	17	10
29.6	16.8	0	2	14.8	7.2	0	0	7.4	0	16.8	4
29.8	16.6	0	1.8	14.6	7	0	0	7.2	0	16.6	1
30	16.4	0	1.6	14.4	6.8	0	0	7	0	16.4	10

Table 4.1: A snippet of an episode after training. It shows the time that the objects and the corresponding action taken by the agent at each time-step. One can see the agent chooses between two high class objects.

The table above, 4.1, is a section of an episode between times 28.2 and 30 seconds. The colours of each object refers to the class/type of object, with green, yellow and red being the most, middle and least valuable type of object. From this table, one can see that the agent carried out the best

action in each state. We also need to see whether the agent observed the objects for a reasonable amount of time. Since the reward function matches a signal to noise ratio function, the agent could observe as many objects as possible for short amounts of time. This would still yield a high reward but in practice would make these observations useless, since images that have been observed for a short amount of time are very noisy.

Object	**Flux**	**Optimal Time (s)**	**Agent Observation time (s)**
1	300	4	0.8
2	200	4.5	4.6
3	600	2.5	6.8
4	500	3	0.4
5	300	4	0.8
6	600	2.5	2.6
7	600	2.5	1.8
8	500	3	1.8
9	200	4.5	9.8
10	200	4.5	0.8

Table 4.2: Table of objects and corresponding best observation time by reward function. As well as the total observation time for each object, decided by the RL agent. This shows how the agent prioritised observing as many objects as possible, until a better object is available

The agent has learnt to observe objects when they are available and learnt the optimal observation time for each object. Therefore the agent has learnt

to designate the best observation times for each object.

From table 4.1, one can see that the agent has learnt the more valuable objects over less valuable objects. The agent does this by observing objects, belonging to the higher valued class, first. However, the agent has also learnt that there is value in the lesser classed objects, thus it learns to observe these objects once the reward is comparable to objects already observed.

4.3 Deep Q-Learning Algorithm after adjustments

The same simulation was then run using the adjustments to avoid the agent switching between two of the same objects. The new environment meant that the comparison graphs had to be adjusted as well. These graphs can be found in the appendices to reduce repetition.

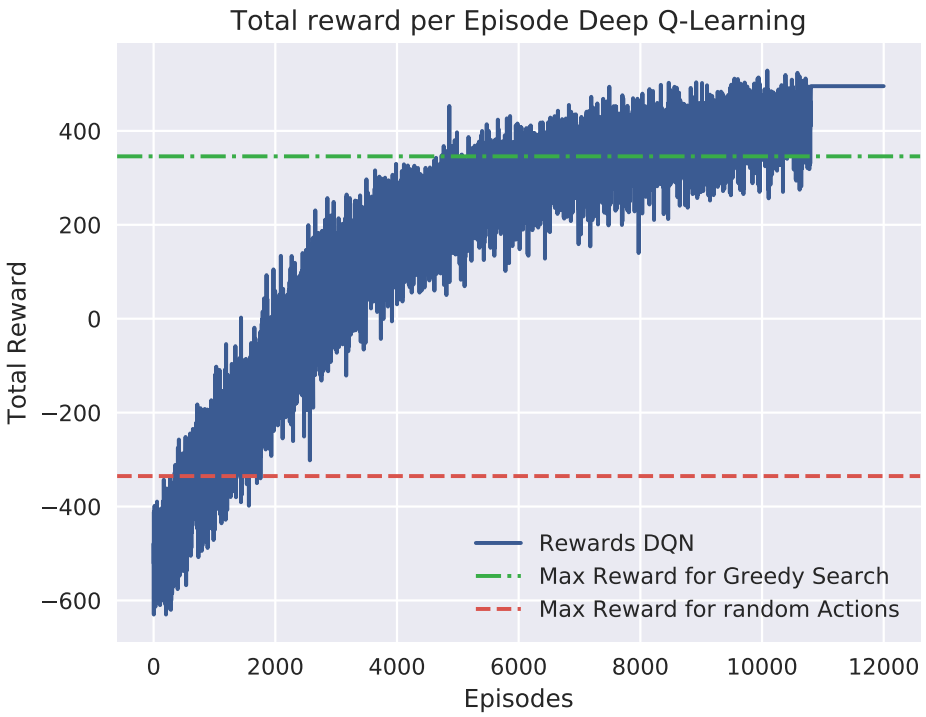

Figure 4.5: Total reward per episode for Deep Q-Learning Algorithm (after adjustments) compared to other algorithms. In this case the agent takes longer to train due to larger state space, but is still able to gain a larger total reward than the greedy search algorithm

The figure 4.5 above, shows how well the Deep Q-learning agent performed against other algorithms. In order to compare the agents actions we can use the table 4.1 of previous actions.

	Object availability										Actions	
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
28.2	0	0	3.4	16.2	8.6	0	0	8.8	0	18.2	10	8
28.4	18	0	3.2	16	8.4	0	0	8.6	0	18	1	8
28.6	17.8	0	3	15.8	8.2	0	0	8.4	0	17.8	1	8
28.8	17.6	0	2.8	15.6	8	0	0	8.2	0	17.6	4	8
29	17.4	0	2.6	15.4	7.8	0	0	8	0	17.4	10	10
29.2	17.2	0	2.4	15.2	7.6	0	0	7.8	0	17.2	1	10
29.4	17	0	2.2	15	7.4	0	0	7.6	0	17	10	10
29.6	16.8	0	2	14.8	7.2	0	0	7.4	0	16.8	4	10
29.8	16.6	0	1.8	14.6	7	0	0	7.2	0	16.6	1	10
30	16.4	0	1.6	14.4	6.8	0	0	7	0	16.4	10	10

Table 4.3: A snippet of an episode after training, comparing the actions from the two agents. It shows the time that the objects and the corresponding action taken by the agent at each time-step. It is clear that the agent is avoiding switching between multiple objects and instead observes one object to completion.

The table above shows the actions made by both agents, before and after adjustments were made.

			Agent Observation time	
Object	**Flux**	**Optimal Time (s)**	**Before (s)**	**After (s)**
1	300	4	0.8	0
2	200	4.5	4.6	5
3	600	2.5	6.8	2.8
4	500	3	0.4	0
5	300	4	0.8	0
6	600	2.5	2.6	3.4
7	600	2.5	1.8	2.6
8	500	3	1.8	2.6
9	200	4.5	9.8	12.6
10	200	4.5	0.8	1.2

Table 4.4: Table of objects and corresponding best observation time by reward function. As well as the total observation time for each object, decided by the RL agents. The new agent has now prioritised high value objects and aims to observe these objects for as long as possible. The agents observation time after adjustments is a rough estimate, ± 0.2 seconds, since the exact times could not be retrieved, due to the new time update method.

The penalty introduced when the agent observes an object for too long, is to help train the agent faster. In a simulation comparable to a real night sky viewing, the time at which a penalty is introduced can be tedious to keep track of. Also, logically there is no penalty for observing an object over the allotted time, the reward becomes smaller the longer an object is observed

(approaching 0). It would then make sense to remove this penalty.

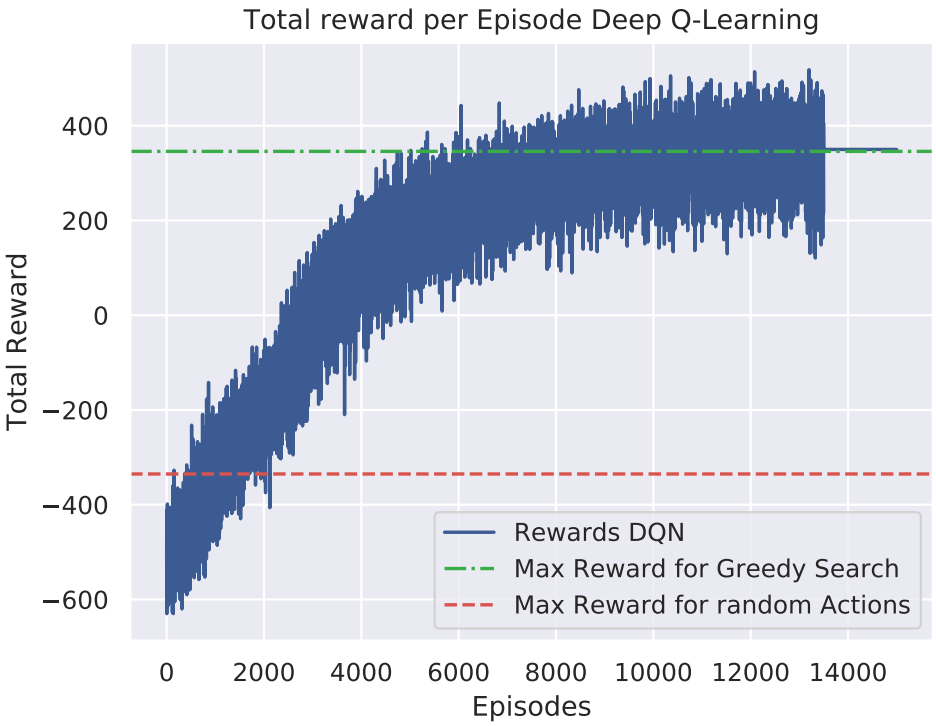

Figure 4.6: Total reward per episode for the Deep Q-Learning Algorithm (after adjustments) with the observation penalty removed. The agent is trained for significantly more, roughly 15 000, training episodes

The figure above shows the difference in total reward when the penalty for observing an object over the optimal time is removed.

4.4 Analysis

Looking at both figure 4.4 and 4.5, it is clear that in both cases the deep Q-learning model has done better than the other algorithms used as compar-

isons. In the case of figure 4.4, the model was able to continuously achieve the maximum reward in the environment, during validation. The deep Q-learning model was able to achieve a reward of roughly 640 while the greedy algorithm was achieved a maximum of 440. A similar argument can be made for figure 4.5, which gained a reward of 520, while the greedy search algorithm achieved 345. Even though the algorithm required more training to achieve the maximum reward, the reward is larger than the greedy algorithm during validation. This means that in both cases the deep Q-learning algorithm has achieved the goal described by the reward function.

The snippet of training in table 4.1, displays the actions the agent has made given the state of the environment. We can from this small snippet that the agent switches between two high valued class objects and a low valued class object. This happens despite there being two other high valued class objects available. This is due to the objects being observed to completion earlier on in the episode, we can see this in table D.1. Ignoring the object 3 and 5's availability, the agent then has to decide between objects 1, 4, 8 and 10. The agent chooses to switch between objects 10 and 1 initially since these objects are of the highest value. Once the reward from object 1 is comparable to object 4, the agent switches to the lower value object. In a more proper simulation of the telescope, there would be a time delay between switching between objects. In this way, an astronomer would avoid switching between objects. The astronomer would instead observe one object and observe to

completion before switching to another, the astronomer would rarely switch back to an object.

After adjustments were made, it is clear from table 4.3 that the objective of the adjustments was achieved. The agent no longer switches between objects multiple times. The agent instead observes one object for a collection of time-steps. Table 4.3, does show that the agent is not observing the most valuable objects. At time 28.2, the agent is observing object 8, clearly, there are more valuable objects to switch to observing. Looking at the rewards toward the end of training, from figure 4.5, there is a more optimal total reward that has been achieved during training. This is an indication that the policy has not converged to an optimal policy. It is possible that further training could result in a policy that improves the total reward and the observations toward the end of the episode.

The table of observation times, 4.2, shows that the agent has attempted to observe objects for the optimal amount of time (before a negative reward is received). Although some objects were observed for a very short amount of time, these objects were available towards the end of the episode, so the agent attempted to get as much reward as possible from these objects. The table also shows that the agent observes some objects more than the time required, this is simply due to these objects being the only object available. Observe an object for longer than required results in a smaller penalty than observing an unavailable object, essentially forcing this behaviour. In true astronomy

nature, it is impossible to observe all the available objects in one-night sky viewing. There is the factor of time taken to switch between objects, this would significantly reduce the available observation time. The agent would then have to allocate time more wisely and reduce the time that it observes less valuable objects. This is the objective of adjusting the reward function such that this behaviour is avoided.

Due to the adjustments, the agent has not observed some objects. The new reward function, coupled with the availability of the objects, has forced the agent to choose between objects. In some cases, only one object is available, which gives the agent an easy choice. In other cases, especially towards the end of an episode, the agent has to make the choice between objects that it is already observing or to switch to another object which has more value. In terms of rewards, this is an obvious choice between a small positive reward or no reward at all. However, if the current object being observed is of a lower class, the agent should switch to an object with a higher value class, even if it means an initial reward of 0. This would result in a maximised total reward for the episode. This could also be due to a sub-optimal policy discussed above. The agent has also learned to observe objects for a more reasonable amount of time, from table 4.4

Table D.1, shows that there are still artefacts of a sub-optimal policy. From time 21.6 to 23.4 seconds, the agent switches between objects, this would mean that the agent receives little to no reward during this time pe-

riod. This might seem like the agent is again switching between objects, but it is in actual fact a method to avoid negative penalty. Looking at table 4.4, one can see that the agent has observed object 2 and 6 for more than the optimal time. This would mean that the agent would receive a negative penalty for continuously observing these objects. By switching between objects, the agent receives a zero reward instead of a penalty, in this way the agent maximises the final reward. This may seem like unexpected and unwanted behaviour, but it is in actual fact a direct result of the reward function.

As one can see from figure 4.6, the total reward is roughly the same as a greedy search algorithm. Towards the end of the training phase, the agent does achieve a significantly larger reward than what is achieved during validation. This shows that the agent is capable of achieving a higher reward but would need many more training episodes to converge to this value. The number of training episodes is already significantly larger than the original 7200 episodes. This shows how the penalty has helped reduce the number of episodes required for training. But in a true simulation, this penalty is not realistic and should be excluded.

Chapter 5

Conclusion

Reinforcement learning is still a relatively new field of study, while this study has many shortcomings, there is some evidence to suggest that reinforcement learning could be a possible solution to the problem of building observing schedules in astronomy.

It is clear that the deep Q-Learning algorithm worked well on the task, due to the discrete action space. From the results, one can see that the agent has achieved the task of scheduling objects during a night of observing. The agent also outperforms other generic algorithms, specifically greedy search. Whether or not the algorithm mimics astronomer like behaviour is subjective. Given the simulation, the agent is indeed able to act similar, if not mimic, the choices an astronomer would make. This simple simulation shows that there is a promise in investigating this approach further. With more resources and research, it is possible that deep reinforcement learning could be a replace-

ment for human astronomers. The future algorithm would require much more knowledge in the fields of astronomy and computer science. This scheduling approach could be further implemented in other fields that require resources to be allocated. A typical example is a scheduler for high-performance cluster computing (Staples 2006). The objective here is to allocate computational jobs to certain parts of the cluster. A good scheduler is able to keep the number of resources in use high, while also making sure each job gets the correct amount of resource for the correct amount of time.

The agent is able to observe the most valuable available objects throughout the episode. The agent is also able to observe the object for a reasonable amount of time. Although weather effects were not introduced during the final iteration of the investigation, a previous iteration showed that the agent is able to adapt to these random effects. This could mean that if random weather effects were introduced in the simulation above, the agent would be able to adapt.

By using a function approximator like neural networks as the agent, one could say that the neural network has learned the relationship between object flux and type, and the time to observe the object. If this is true then the agent would be able to be placed in a similar environment and need little to no training to be able to find an optimal policy. The same can be said for weather effects and transients, the neural network's ability to infer the relationship means it would be able to adapt to objects that are no longer

available or appear suddenly. This would mean that the agent would be able to generalise to multiple environments, which would mean the solution would only need to be solved once and applied to many different telescopes.

The most challenging aspect of developing a reinforcement learning algorithm is deciding on a reward function for the value of observing each target. In this work, we assumed it as given (e.g. by a committee of humans) but doing this automatically is an unsolved problem in general. A reward function that does not explain the optimal solution well, will result in an agent that achieves a high reward but does not achieve the objective.
In order to do a thorough investigation into a worthwhile solution, one would need to investigate many different algorithms within reinforcement learning and compare these algorithms. It also might not be possible to properly compare the algorithms since the environments presented to the algorithms might have to differ.

5.1 Down falls

Although the reward graph, as well as the table of objects, shows that the agent is indeed doing the correct actions, there are many shortcomings in the results.
When using a neural network, the state which is presented to the network is

of fixed size. In this case, it is possible to encode a fixed state since the number of objects is small. In an environment with a larger number of objects, it would be ideal that the state and action dimensions decrease, such that it represents only the available objects.

Another issue that arises is the training time factor. For this small simulation of 10 objects, the number of training episodes is roughly 8000. One can assume that the number of training episodes is related to the number of states, in order to find an optimal policy and converged Q-values the agent would need to explore all possible states. Due to the use of a neural network, one can greatly reduce the number of training episodes, since the neural network would infer some relations of state and actions for unseen states. Even though the neural network reduces the number of training episodes, as the study progresses to a more complex model the number of states grows proportional to the number of objects as well as the properties of these objects. There has been research in reinforcement learning where states are of a continuous nature, typically these include policy gradient methods or a tree-based approach (Ernst, Geurts, and Wehenkel 2005). Although not continuous, the state space will be large when applied to actual telescopes. In a real-world simulation, the amount of objects is vast so the training time for a reinforcement learning technique could take a large amount of time. Since a training episode is dependent on the episodes before, it seems unlikely to have the training run in parallel. One could reduce training time by reducing the number of times that the neural network is updated. In this way, one

could run parallel training episodes under the current policy and train the neural network on all the data from these episodes.

A similar argument can be made for the action space since in this approach the action space is directly proportional to the number of objects.

A major problem with the solution presented is the reward function. The issue of presenting rewards at each time-step is not representative of the true nature of the signal to noise ratio. A more general approach would be to have a reward at the end of an episode, similar to the cumulative reward function 3.3. This would mean that the agent could possibly need to explore states more. This is because the agent will not immediately know which actions affect the final reward.

The network architecture has not been properly investigated, it could be said that the number of parameters could be reduced and further increase the time taken to train the network. Although, once the environment becomes more complex naturally the number of parameters in the network will need to be increased.

The reward function is based on a perfect signal to noise ratio. In a real-world simulation, there are many factors that affect the quality of the observation. These factors were not taken into consideration when developing the simulation and would be needed when taking this project further.

5.2 Further Investigations

There has been work in applying reinforcement learning to telescopes. A recent paper has investigated the use of reinforcement learning as a scheduler for the Large Synoptic Survey Telescope (LSST) (Naghib et al. 2018). This is very similar to the approach taken in this study, but LSST is a survey telescope and observes patches of the sky instead of singular objects. The paper also takes into account many more factors in training the agent. The reward function is based roughly on the time taken to move the telescope and the airmass of the current observation. The objective, however, is slightly different, since the agent is responsible for choosing parameters that handle the scheduling for the night. These parameters are highly complex and dependant, due to the agent's responsibility of choosing these parameters such that the telescope observes as much as possible in a night sky viewing.

A progression of this work would be to use a policy gradient approach. A policy based approach would allow for a larger action space. This is because policy gradient methods thrive in high dimensional states and large actions spaces. The policy-based approach would be a better approach to a larger simulation with many objects. An approach like this is discussed in (Lillicrap et al. 2015).

The agent presented in this study can only perform optimally based on

the reward function. If the actions taken by the agent are not as expected, the reward function needs to be adjusted in order to get an optimal policy for the environment. A more complex reward function could be investigated in order to retrieve an agent that performs well at the given task. A more complex reward function and environment coupled with Policy Gradient Methods could be a further improvement on the algorithm already discussed. In planning this study a further progression would be to add a choice of the observation type for each action. In astronomy, there are in general two major observation methods, namely photometry and spectroscopy. Photometry is used to find magnitudes of objects, which can be used to find brightnesses and distances of objects, while spectroscopy can be used to infer the chemical composition and distances of objects. These two types are often related and would need two (possibly) linked reward functions.

One could take a supervised learning approach by having a model that learns the state as feature vectors and the actions an astronomer took in each state as a target variable. In this way, a well-trained model would yield astronomer-like behaviour.This does however require significant amounts of labelled data which would not be easy to obtain since astronomers do not generally the circumstances which lead to an observation made, and where this is recorded the amount of data would be too little.

Bibliography

Rajaraman, A. and J. D. Ullman (2011). *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press.

Shalev-Shwartz, S. and S. Ben-David (2014). *Understanding Machine Learning: From Theory to Algorithms*. New York, NY, USA: Cambridge University Press.

Marsland, S. (2009). *Machine Learning: An Algorithmic Perspective*. 1st. Chapman & Hall/CRC.

Busoniu, L. et al. (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators*. 1st. Boca Raton, FL, USA: CRC Press, Inc.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12, pp. 2825–2830. ISSN: 1532-4435. URL: `http://dl.acm.org/citation.cfm?id=1953048.2078195`.

James, G. et al. (2014). *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated.

Raschka, S. (2015). *Python Machine Learning*. Packt Publishing.

Hauck, T. (2014). *Scikit-Learn Cookbook*. Packt Publishing.

Bronshtein, A. (2017). *A Quick Introduction to K-Nearest Neighbors Algorithm*. URL: https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

Vincent, P. et al. (2008). "Extracting and Composing Robust Features with Denoising Autoencoders". In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: ACM, pp. 1096–1103. URL: http://doi.acm.org/10.1145/1390156.1390294.

Sutton, R. S. and A. G. Barto (2017). "Reinforcement learning: an introduction." In: *UCL,Computer Science Department, Reinforcement Learning Lectures*, p. 1054. ISSN: 1045-9227. arXiv: arXiv:1011.1669v3. URL: http://incompleteideas.net/book/bookdraft2018jan1.pdf%7B%5C%%7D0Ahttp://incompleteideas.net/sutton/book/bookdraft2017june.pdf.

Oliehoek, F. A. (2012). "Decentralized POMDPs". In: *Reinforcement Learning: State of the Art*. Ed. by M. Wiering and M. Otterlo. Vol. 12. Adaptation, Learning, and Optimization. 10.1007/978-3-642-27645-3$_1$5. Berlin,

Germany: Springer Berlin Heidelberg, pp. 471–503. URL: http://dx.doi.org/10.1007/978-3-642-27645-3_15.

Bhatt, S. (2018). *5 Things You Need to Know about Reinforcement Learning*. URL: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html.

Sigaud, O. and O. Buffet (2010). *Markov Decision Processes in Artificial Intelligence*. Wiley-IEEE Press.

Arulkumaran, K. et al. (2017). "A Brief Survey of Deep Reinforcement Learning". In: arXiv: 1708.05866. URL: http://arxiv.org/abs/1708.05866%20http://dx.doi.org/10.1109/MSP.2017.2743240.

Mnih, V. et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518, pp. 529–33.

Wikipedia contributors (2019). *Breakout (video game) — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Breakout_(video_game)&oldid=880028750.

Silver, D., J. Schrittwieser, et al. (2017). "Mastering the game of Go without human knowledge". In: *Nature* 550, pp. 354–. URL: http://dx.doi.org/10.1038/nature24270.

Silver, D., T. Hubert, et al. (2018). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419, pp. 1140–1144. ISSN: 0036-8075. eprint: http://science.sciencemag.org/content/362/6419/1140.full.pdf. URL: http://science.sciencemag.org/content/362/6419/1140.

Vinyals, O. et al. (2017). "StarCraft II: A New Challenge for Reinforcement Learning". In: *arXiv e-prints*, arXiv:1708.04782, arXiv:1708.04782. arXiv: 1708.04782 [cs.LG].

Lee, T. (2019). *An AI crushed two human pros at StarCraft—but it wasn't a fair fight*. URL: https://arstechnica.com/gaming/2019/01/an-ai-crushed-two-human-pros-at-starcraft-but-it-wasnt-a-fair-fight/.

Wolpert, D. H., K. Tumer, and J. Frank (1999). "Using Collective Intelligence to Route Internet Traffic". In: *arXiv e-prints*, cs/9905004, cs/9905004. arXiv: cs/9905004 [cs.LG].

Lin, L.-J. (1992). "Reinforcement Learning for Robots Using Neural Networks". UMI Order No. GAX93-22750. PhD thesis. Pittsburgh, PA, USA.

Lippmann, R. (1987). "An introduction to computing with neural nets". In: *IEEE ASSP Magazine* 4.2, pp. 4–22. ISSN: 0740-7467.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Dertat, A. (2017). *Applied Deep Learning - Part 1: Artificial Neural Networks*. URL: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6.

Deshpande, M. *Perceptrons: The First Neural Networks*. URL: https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/.

Donges, N. (2018). *Gradient Descent in a Nutshell*. URL: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0.

Reviews, C. (2016). *Artificial Intelligence*. Cram101. URL: https://books.google.co.za/books?id=xL8jXVao3xIC.

Watkins, C. J. C. H. and P. Dayan (1992). "Q-learning". In: *Machine Learning* 8.3, pp. 279–292. ISSN: 1573-0565. URL: https://doi.org/10.1007/BF00992698.

Hasselt, H. van, A. Guez, and D. Silver (2015). "Deep Reinforcement Learning with Double Q-learning". In: arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

Lillicrap, T. P. et al. (2015). "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971. arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

Li, Y. (2017). "Deep Reinforcement Learning: An Overview". In: arXiv: 1701.07274. URL: http://arxiv.org/abs/1701.07274.

Helminiak, K. G. (2009). "Impact of the Atmospheric Refraction on the Precise Astrometry with Adaptive Optics in Infrared". In: *New Astron.* 14, pp. 521–527. arXiv: 0805.3369 [astro-ph].

Abbott, B. P. et al. (2016). "Observation of Gravitational Waves from a Binary Black Hole Merger". In: *Phys. Rev. Lett.* 116.6, p. 061102. arXiv: 1602.03837 [gr-qc].

Abbott, B. P. et al. (2019). "Properties of the Binary Neutron Star Merger GW170817". In: *Physical Review X* 9, 011001, p. 011001. arXiv: 1805.11579 [gr-qc].

Evans, P. A. et al. (2017). "Swift and NuSTAR observations of GW170817: Detection of a blue kilonova". In: *Science* 358.6370, pp. 1565–1570. ISSN: 0036-8075. eprint: http://science.sciencemag.org/content/358/6370/1565.full.pdf. URL: http://science.sciencemag.org/content/358/6370/1565.

Reflecting Telescopes. URL: https://lco.global/spacebook/reflecting-telescopes/.

Chromey, F. R. (2010). *To Measure the Sky: An Introduction to Observational Astronomy*. Cambridge University Press.

Schroeder, D. J. (2000). "Chapter 17 - Detectors, Signal-to-Noise, and Detection Limits". In: *Astronomical Optics (Second Edition)*. Ed. by D. J. Schroeder. Second Edition. San Diego: Academic Press, pp. 425–443. URL: http://www.sciencedirect.com/science/article/pii/B9780126298109500189.

Staples, G. (2006). "TORQUE Resource Manager". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: ACM. URL: http://doi.acm.org/10.1145/1188455.1188464.

Ernst, D., P. Geurts, and L. Wehenkel (2005). "Tree-Based Batch Mode Reinforcement Learning". In: *J. Mach. Learn. Res.* 6, pp. 503–556. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1046920.1088690.

Naghib, E. et al. (2018). "A Framework for Telescope Schedulers: With Applications to the Large Synoptic Survey Telescope". In: *arXiv e-prints*, arXiv:1810.04815, arXiv:1810.04815. arXiv: 1810.04815 [astro-ph.IM].

Appendix A

Pseudo-code for algorithm

The algorithms used to train the episode can be found below. The main algorithm used is A.4, the functions that are used in the algorithm can be found before.

Algorithm A.1 Initialise Environment

```
def initialise():
    num_objects, num_classes = obj_params()
    timestep, maxtime = env_params()
    objects = dataframe.init()
    objects['Flux'] = random.choice([800, ..., 200], num_objects
    )
    objects['TimeObs'] = 0
    objects['Type'] = random.integer(0, num_classes, num_objects
    )
    obj_availability = generate_times(maxtime, timestep)
    default_state = convert_state(objects, obj_availability[0])
    state_length = default_state
    return state_length
```

Algorithm A.2 Update Time observed

```
def update_time(objects, action, timeavails):
    time_observered = objects['TimeObs']
    time_avail = timeavails[action]
    if time_avail != 0:
        time_observered += timestep
    objects['TimeObs'] = time_observered
    return objects
```

Algorithm A.3 Retrieve reward for action

```
def reward(objects, action, timeavails):
    time = objects['TimeObs'][action]
    type = objects['Type'][action]
    flux = objects['Flux'][action]
    flux_factor = convert_f(flux)
    type_factor = convert_t(type)
    func_value = exponenctial_decay(time, flux_factor)
    timeavail = timeavails[action]
    if timeavail == 0:
        rew = -15
    else:
        if func_value <= 0.01:
            rew = -1
        else:
            rew = type_factor * func_value
    return rew
```

Algorithm A.4 Training Agent

```
state_length, max_time, timestep = env.initialise()
num_episodes = get_number_episodes()
gamma, epsilon, lr = get_params()
Q_NN = NN.initialise(lr, state_length)
for i in range(num_episodes):
    env.reset()
    for j in range(0, max_time, timestep):
        state, objects, timeavails = env.get_state()
        Q_NN_values = Q_NN.predict(state)
        action = epsilon_action(Q_NN_values)
        reward = env.reward(objects, action, timeavails)
        next_state = env.update_time(dataframe, action,
    timeavails)
        Q_val_vector = update_Q(Q_NN_values, action, reward,
    gamma, next_state)
        episode_data = store_data(state, Q_NN_values)
    all_data = store(episode_data)
    Q_NN.train_episode(episode_data)
    if i % (0.1*num_episodes) == 0:
        replay_memory(all_data)
```

Appendix B

Parameters

Hyper-Parameter	**Value**
Gamma (γ)	0.8
Episode Time and Time-step	30 and 0.1
Neural Network learning rate	1×10^{-4}
Neural network architecture	(500, 500, 500, 500)
Optimiser	Adam
Experience Replay Frequency	Every 10% of the number of episodes
Number of Training episodes	12000
Number of Validation episodes	10800

Table B.1: Hyper-Parameters used when training the DQN algorithm.

Appendix C

Additional Results for comparison algorithms

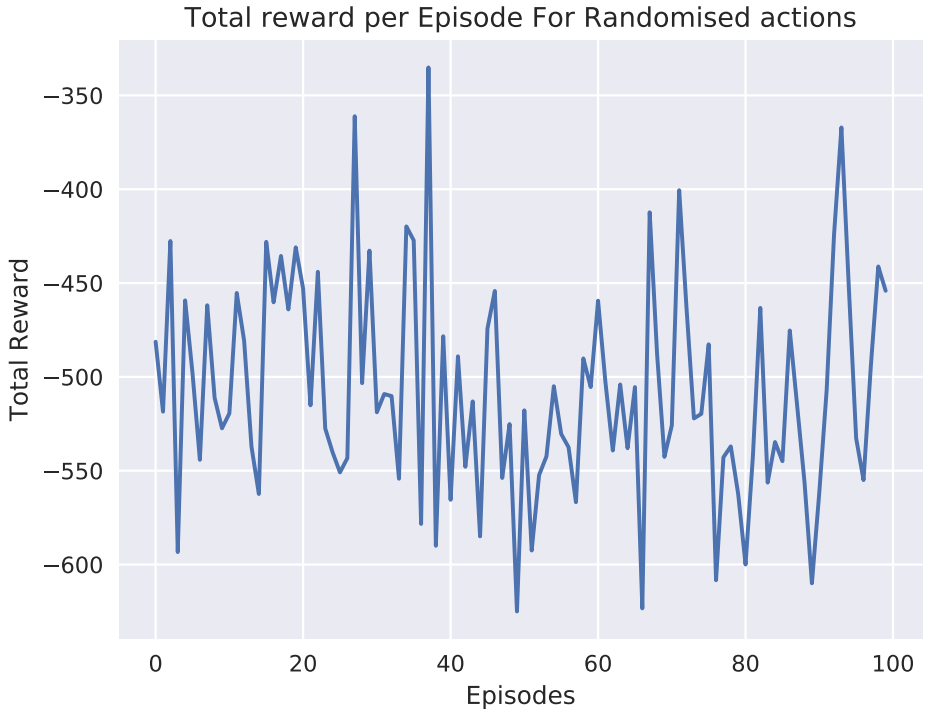

Figure C.1: Total reward for an agent acting randomly, after adjustments to the reward function was made. The total reward varies due to the random action and peaks at roughly -350. This is due to the new reward function penalising switching between objects. The peak is what is used as a comparison for the final algorithm

Figure C.2: Total reward for an agent acting greedily, after adjustments to the reward function was made. The new reward function causes the greedy search algorithm to have a higher peak since it is less likely to switch between objects

			Agent Observation Time	
Object	**Flux**	**Optimal Time (s)**	**Random (s)**	**Greedy (s)**
1	300	4	3.6	0.2
2	200	4.5	1.8	0.6
3	600	2.5	2.6	1.4
4	500	3	3.0	1.2
5	300	4	3.0	1.6
6	600	2.5	3.4	12.6
7	600	2.5	3.8	0.8
8	500	3	2.8	0.8
9	200	4.5	3.6	10.8
10	200	4.5	2.6	0.2

Table C.1: Table of objects and corresponding best observation time by reward function. As well as the total observation time for each object, decided by the comparison agents.

Appendix D

Full Table of objects and Corresponding actions

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
0	0	0	0	0	0	0	0	0	0	0	3	9
0.2	0	0	0	0	0	0	0	0	0	0	3	9
0.4	0	0	0	0	0	0	0	0	0	0	3	9
0.6	0	0	0	0	0	0	0	0	0	0	3	9
0.8	0	0	0	0	0	0	0	0	0	0	3	9
1	0	0	0	0	0	0	0	0	0	0	3	9
1.2	0	0	0	0	0	0	0	0	0	0	3	9
1.4	0	0	0	0	0	0	0	0	0	0	3	9

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
1.6	0	0	0	0	0	0	0	0	12.8	0	9	9
1.8	0	0	0	0	0	0	0	0	12.6	0	9	9
2	0	0	0	0	0	0	0	0	12.4	0	9	9
2.2	0	0	0	0	0	0	0	0	12.2	0	9	9
2.4	0	0	0	0	0	0	0	0	12	0	9	9
2.6	0	0	0	0	0	0	0	0	11.8	0	9	9
2.8	0	0	0	0	0	0	0	0	11.6	0	9	9
3	0	0	0	0	0	0	0	0	11.4	0	9	9
3.2	0	0	0	0	0	0	0	0	11.2	0	9	9
3.4	0	0	0	0	0	0	0	0	11	0	9	9
3.6	0	0	0	0	0	0	0	0	10.8	0	9	9
3.8	0	0	0	0	0	0	0	0	10.6	0	9	9
4	0	0	0	0	0	0	0	0	10.4	0	9	9
4.2	0	0	0	0	0	0	0	0	10.2	0	9	9
4.4	0	0	0	0	0	0	0	0	10	0	9	9
4.6	0	0	0	0	0	0	0	0	9.8	0	9	9
4.8	0	0	0	0	0	0	0	0	9.6	0	9	9
5	0	0	0	0	0	0	0	0	9.4	0	9	9
5.2	0	0	0	0	0	0	0	0	9.2	0	9	9
5.4	0	0	0	0	0	0	0	0	9	0	9	9

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
5.6	0	0	0	0	0	0	0	0	8.8	0	9	9
5.8	0	0	0	0	0	0	0	0	8.6	0	9	9
6	0	0	0	0	0	0	0	0	8.4	0	9	9
6.2	0	0	0	0	0	0	0	0	8.2	0	9	9
6.4	0	0	0	0	0	0	0	0	8	0	9	9
6.6	0	0	0	0	0	0	0	0	7.8	0	9	9
6.8	0	0	0	0	0	0	0	0	7.6	0	9	9
7	0	0	0	0	0	0	0	0	7.4	0	9	9
7.2	0	0	0	0	0	0	0	0	7.2	0	9	9
7.4	0	0	0	0	0	0	0	0	7	0	9	9
7.6	0	0	0	0	0	0	0	0	6.8	0	9	9
7.8	0	0	0	0	0	0	0	0	6.6	0	9	9
8	0	0	0	0	0	0	0	0	6.4	0	9	9
8.2	0	0	0	0	0	0	0	0	6.2	0	9	9
8.4	0	0	0	0	0	0	0	0	6	0	9	9
8.6	0	0	0	0	0	0	0	0	5.8	0	9	9
8.8	0	0	0	0	0	0	0	0	5.6	0	9	9
9	0	0	0	0	0	0	0	0	5.4	0	9	9
9.2	0	0	0	0	0	0	0	0	5.2	0	9	9
9.4	0	0	0	0	0	0	0	0	5	0	9	9

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
9.6	0	0	0	0	0	0	0	0	4.8	0	9	9
9.8	0	0	0	0	0	0	0	0	4.6	0	9	9
10	0	0	0	0	0	0	0	0	4.4	0	9	9
10.2	0	0	0	0	0	0	0	0	4.2	0	9	9
10.4	0	0	0	0	0	0	0	0	4	0	9	9
10.6	0	0	0	0	0	0	0	0	3.8	0	9	9
10.8	0	0	0	0	0	0	0	0	3.6	0	9	9
11	0	0	0	0	0	0	0	0	3.4	0	9	9
11.2	0	0	0	0	0	0	0	0	3.2	0	9	9
11.4	0	0	0	0	0	0	8.2	0	3	0	7	9
11.6	0	0	0	0	0	0	8	0	2.8	0	7	9
11.8	0	0	0	0	0	0	7.8	0	2.6	0	7	9
12	0	0	0	0	0	0	7.6	0	2.4	0	7	9
12.2	0	0	0	0	0	0	7.4	0	2.2	0	7	9
12.4	0	0	0	0	0	0	7.2	0	2	0	7	9
12.6	0	0	0	0	0	12.4	7	0	1.8	0	6	7
12.8	0	0	0	0	0	12.2	6.8	0	1.6	0	6	7
13	0	0	0	0	0	12	6.6	0	1.4	0	6	7
13.2	0	0	0	0	0	11.8	6.4	0	1.2	0	6	7
13.4	0	0	0	0	0	11.6	6.2	0	1	0	6	7

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
13.6	0	12.6	0	0	0	11.4	6	0	0.8	0	2	7
13.8	0	12.4	0	0	0	11.2	5.8	0	0.6	0	2	7
14	0	12.2	0	0	0	11	5.6	0	0.4	0	2	7
14.2	0	12	0	0	0	10.8	5.4	0	0.2	0	2	7
14.4	0	11.8	0	0	0	10.6	5.2	0	0	0	2	7
14.6	0	11.6	0	0	0	10.4	5	0	0	0	2	7
14.8	0	11.4	0	0	0	10.2	4.8	0	0	0	2	7
15	0	11.2	0	0	0	10	4.6	0	0	0	2	7
15.2	0	11	0	0	0	9.8	4.4	0	0	0	6	2
15.4	0	10.8	0	0	0	9.6	4.2	0	0	0	2	2
15.6	0	10.6	0	0	0	9.4	4	0	0	0	7	2
15.8	0	10.4	0	0	0	9.2	3.8	0	0	0	2	2
16	0	10.2	0	0	0	9	3.6	0	0	0	6	2
16.2	0	10	0	0	0	8.8	3.4	0	0	0	2	2
16.4	0	9.8	0	0	0	8.6	3.2	0	0	0	7	2
16.6	0	9.6	0	0	0	8.4	3	0	0	0	6	2
16.8	0	9.4	0	0	0	8.2	2.8	0	0	0	2	2
17	0	9.2	0	0	0	8	2.6	0	0	0	2	2
17.2	0	9	0	0	0	7.8	2.4	0	0	0	7	2
17.4	0	8.8	0	0	0	7.6	2.2	0	0	0	2	2

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
17.6	0	8.6	0	0	0	7.4	2	0	0	0	2	2
17.8	0	8.4	0	0	0	7.2	1.8	0	0	0	2	2
18	0	8.2	0	0	0	7	1.6	0	0	0	2	2
18.2	0	8	0	0	0	6.8	1.4	0	0	0	2	2
18.4	0	7.8	0	0	0	6.6	1.2	0	0	0	2	2
18.6	0	7.6	0	0	0	6.4	1	0	0	0	2	2
18.8	0	7.4	0	0	0	6.2	0.8	0	0	0	2	2
19	0	7.2	0	0	0	6	0.6	0	0	0	2	2
19.2	0	7	12.4	0	0	5.8	0.4	0	0	0	3	6
19.4	0	6.8	12.2	0	0	5.6	0.2	0	0	0	3	6
19.6	0	6.6	12	0	0	5.4	0	0	0	0	3	6
19.8	0	6.4	11.8	0	0	5.2	0	0	0	0	3	6
20	0	6.2	11.6	0	0	5	0	0	0	0	3	6
20.2	0	6	11.4	0	0	4.8	0	0	0	0	3	6
20.4	0	5.8	11.2	0	0	4.6	0	0	0	0	3	6
20.6	0	5.6	11	0	0	4.4	0	0	0	0	3	6
20.8	0	5.4	10.8	0	0	4.2	0	0	0	0	3	6
21	0	5.2	10.6	0	0	4	0	0	0	0	6	6
21.2	0	5	10.4	0	0	3.8	0	0	0	0	3	6
21.4	0	4.8	10.2	0	0	3.6	0	0	0	0	6	6

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
21.6	0	4.6	10	0	0	3.4	0	0	0	0	6	6
21.8	0	4.4	9.8	0	0	3.2	0	0	0	0	3	2
22	0	4.2	9.6	0	0	3	0	0	0	0	6	2
22.2	0	4	9.4	0	0	2.8	0	0	0	0	3	6
22.4	0	3.8	9.2	0	0	2.6	0	0	0	0	3	2
22.6	0	3.6	9	0	0	2.4	0	0	0	0	6	6
22.8	0	3.4	8.8	0	0	2.2	0	0	0	0	2	2
23	0	3.2	8.6	0	0	2	0	0	0	0	3	6
23.2	0	3	8.4	0	0	1.8	0	0	0	0	3	2
23.4	0	2.8	8.2	0	0	1.6	0	0	0	0	3	6
23.6	0	2.6	8	0	0	1.4	0	0	0	0	3	3
23.8	0	2.4	7.8	0	0	1.2	0	0	0	0	3	3
24	0	2.2	7.6	0	0	1	0	0	0	0	3	3
24.2	0	2	7.4	0	0	0.8	0	0	0	0	3	3
24.4	0	1.8	7.2	0	0	0.6	0	0	0	0	3	3
24.6	0	1.6	7	0	0	0.4	0	0	0	0	3	3
24.8	0	1.4	6.8	0	0	0.2	0	0	0	0	3	3
25	0	1.2	6.6	0	0	0	0	0	0	0	3	3
25.2	0	1	6.4	0	0	0	0	0	0	0	3	3
25.4	0	0.8	6.2	0	0	0	0	0	0	0	3	3

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
25.6	0	0.6	6	0	0	0	0	11.4	0	0	8	3
25.8	0	0.4	5.8	0	0	0	0	11.2	0	0	8	3
26	0	0.2	5.6	0	0	0	0	11	0	0	8	3
26.2	0	0	5.4	0	0	0	0	10.8	0	0	8	3
26.4	0	0	5.2	0	0	0	0	10.6	0	0	8	8
26.6	0	0	5	0	0	0	0	10.4	0	0	8	8
26.8	0	0	4.8	0	0	0	0	10.2	0	0	8	8
27	0	0	4.6	0	0	0	0	10	0	0	8	8
27.2	0	0	4.4	0	0	0	0	9.8	0	0	8	8
27.4	0	0	4.2	0	9.4	0	0	9.6	0	0	5	8
27.6	0	0	4	0	9.2	0	0	9.4	0	0	5	8
27.8	0	0	3.8	0	9	0	0	9.2	0	0	5	8
28	0	0	3.6	0	8.8	0	0	9	0	0	5	8
28.2	0	0	3.4	16.2	8.6	0	0	8.8	0	18.2	10	8
28.4	18	0	3.2	16	8.4	0	0	8.6	0	18	1	8
28.6	17.8	0	3	15.8	8.2	0	0	8.4	0	17.8	1	8
28.8	17.6	0	2.8	15.6	8	0	0	8.2	0	17.6	4	8
29	17.4	0	2.6	15.4	7.8	0	0	8	0	17.4	10	10
29.2	17.2	0	2.4	15.2	7.6	0	0	7.8	0	17.2	1	10
29.4	17	0	2.2	15	7.4	0	0	7.6	0	17	10	10

	Object availability											
Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Before**	**After**
29.6	16.8	0	2	14.8	7.2	0	0	7.4	0	16.8	4	10
29.8	16.6	0	1.8	14.6	7	0	0	7.2	0	16.6	1	10
30	16.4	0	1.6	14.4	6.8	0	0	7	0	16.4	10	10

Table D.1: A full episode using Deep Q-Learning(before and after adjustments). Green, yellow and red being the highest to lowest value objects respectively. The value for each object indicates the time availability of the object