The Mathematics of Reinforcement Learning and its Applications

by

Aadam Ul Haq

MA4K8 Scholarly Report

Submitted to The University of Warwick

Mathematics Institute

April, 2024

Contents

1 Introduction

Reinforcement Learning is a subset of Machine Learning in which an agent uses previous experiences and rewards via feedback and interaction with its environment to perform a task. The agent performs its task by trying to maximise its reward function. Figure 1 shows the basic structure of a reinforcement learning model. As the agent interacts with the environment via an action, a new state is created and the reward is updated.

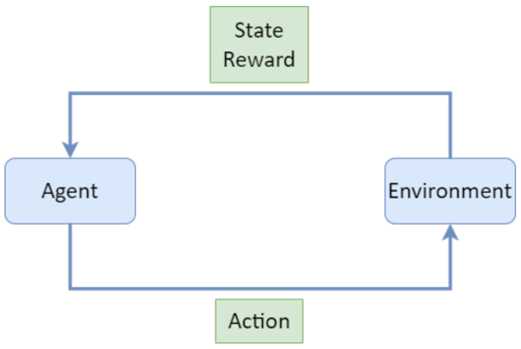

Figure 1: Reinforcement Learning Model. This and all other Figures, unless referenced, were created by the author.

The agent is not told directly what actions to take, however, given a state, it must discover what the best action to take is to yield the highest reward by trial and error. Often, the action must consider not just the immediate reward, but any future rewards too. The agent undergoes two key stages, exploration and exploitation, in which it learns more about the environment and how to make better action selections in the future.

Due to its general nature, reinforcement learning is studied within many disciplines. These include control theory, statistics, game theory and optimisation theory [1]. Some of these topics will be discussed within the report.

Although not as used currently as supervised and unsupervised learning, reinforcement learning is starting to build momentum. One of the earliest uses was to create an agent to beat a human player in a game of checkers [2]. Nowadays, there are uses in fields such as healthcare and robotics. These, and more applications will be explored in the report.

2 History and Motivations

There are three main 'threads' in which reinforcement learning was formed, the first two being found independently [1, pp. 16-25]. The first is based around the psychology of animals and investigating the innate trial–and–error processes they possess. The second examines the problem of optimal control and investigating the solution with techniques such as dynamic programming. The third investigates temporal–difference methods and is a more obscure thread. These three threads were combined in the late 1980's to form modern reinforcement learning.

Trial–and–error learning was first postulated as early as 1850. Expressed by Thorndike in 1911 as the 'Law of Effect' [3, p. 244], the combination of a stimulus with a situation for an animal led to behavioral changes. Specifically, negative and positive stimuli led to an association with an action and the stimulus, or reward. This led to the famous motivation of Pavlovian Conditioning.

Whilst investigating the behaviour of dogs in 1927, Pavlov discovered the phenomenon now known as Pavlovlian Conditioning (or Classical Conditioning) [4]. The experiment involved the salivation of dogs. When given an unconditioned stimulus, in this case food, the dogs had an unconditioned response, in this case salivating. In this stage, when given the neutral stimulus of the bell ringing, there was no response. During conditioning, the dogs were introduced to the food when the bell was rung at the same time. Due to the presence of the food, the dogs started salivating. Repeated pairing of food and bell enables the dogs to learn the association between food and the sound of the bell. Eventually this led to a conditioned response in which the dogs salivated when the bell was rung, even if there was no food present.

The principles of trial–and-error learning were then transferred to a machine learning context by Minsky in 1954 [5, p. 1-1]. In particular, he was interested in how rewards and punishments influence learning and decision–making in both biological and artificial systems.

The optimal control theory thread began later in the 1950's. Optimal control is a branch of control theory that aims to find a control for a dynamic system to optimise an objective function over a period of time. Bellman built upon previous work from the prior century to pioneer an approach specific to reinforcement learning which dynamically defines a functional equation, using a dynamic system's state, and returns an optimal value function. This is known as the Bellman Equation [6, p. 3]. Additionally, Bellman introduced another key concept – Markov Decision Processes (MDPs). The theory Bellman introduced was used further in the discovery of policy iteration methods and the use of dynamic programming.

Finally, the smaller and less distinct thread of temporal difference learning began from Minsky's work [5, §6.4]. Temporal difference methods are again related to animal learning psychology, however specifically to the notion of 'secondary reinforcers'. Secondary reinforcers, similar to trial–and–error learning, is when a stimulus or event is paired with a primary reinforcer and eventually adopts the properties of the primary reinforcer. Samuel built upon Minsky's work to implement a reinforcement agent to beat the game of checkers using the temporal difference techniques [2], although did not directly credit the prior work by Minsky.

As Klopf [7, p. 1] discovered links between trial–and–error and temporal difference methods, others such as Sutton [8, pp. 50-52] further investigated the links to animal learning theory. This all culminated when Watkins [9, p. 4] invented Q–Learning, tying together all the threads. This led to the convergence of trial–and–error learning and dynamic programming.

In the modern age, reinforcement learning techniques have been used to tackle traditional board games. As Tesauro implemented an agent to achieve 'Master–Level' in Backgammon [10], the race was on to create a competent chess agent. This was achieved by IBM's DeepBlue agent in 1997. Although it had dimensionality issues, the agent beat Kasparov, the World Chess Champion, creating a spotlight on reinforcement learning models and artificial intelligence [11]. Researchers then competed to create an agent to beat the game Go, as it was more complicated, with more possible moves so it was impossible to computationally brute–force an algorithm. This was subsequently made possible in 2016 by Google's AlphaGo [12] which defeated Ke Jie, the world's best player. These small achievements show the power of reinforcement learning techniques and gave a challenge in which new methods could be created and the field could be advanced. Currently reinforcement learning techniques are used over many applications, but the largest still are in trying to tackle different video games and board games.

3 Background

Before exploring the mathematics behind reinforcement learning, we must understand relevant background in machine learning. Some information is prior knowledge, and some are new concepts that are briefly explained to be used later in the report.

3.1 Paradigms of Machine Learning

Reinforcement learning is one of three paradigms of machine learning, the others being supervised learning and unsupervised learning. There are similarities in methods between all three types.

Supervised learning is perhaps the simplest of the three [13, pp. 21-23]. The key difference between supervised learning and the other types of learning is that the input data must be labelled so that the algorithm can learn the relationship between the input and the output. Precisely, given a training set of input–output data pairs, $\{\boldsymbol{x}^i, \boldsymbol{y}^i\}_{i=1}^n \subset \mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is the space of inputs (typically d–dimensional data) containing quantitative data and qualitative data and $\mathcal{Y}$ is the space of outputs (typically 1–dimensional data), the goal is to learn a function $h : \mathcal{X} \to \mathcal{Y}$ to predict an output given an input. The function h achieves its goal by minimising a loss function $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ which measures the mismatch between a prediction on a given input $\boldsymbol{x} \in \mathcal{X}$ and an element $\boldsymbol{y} \in \mathcal{Y}$.

Supervised learning models are typically simpler than the other two machine learning paradigms, with the most famous models being regression models or classification models.

This type of learning is inadequate for interaction learning as in reinforcement learning as we require the algorithm to learn well based off past experiences and quickly adapt to new situations. Additionally, it is difficult to tailor the data to a supervised learning setting, especially in larger contexts.

Unsupervised learning on the other hand does not use labelled data in its input, rather it tries to discover the underlying structure of the data to return an output. In this way, unsupervised learning is similar to reinforcement learning in its exploratory structure. Whilst unsupervised learning explores patterns in the structure, reinforcement learning explores an environment in order to discover the consequences of different actions. Examples of unsupervised learning include dimensionality reduction and clustering algorithms. Similar to supervised learning, the goal is to reduce the error in predicted results.

Although the nature of the objectives are different, reinforcement learning shares some similarities with the other paradigms. All three are object–driven learning techniques. Both supervised and unsupervised learning techniques aim to reduce a loss function and reduce the error in tasks. Meanwhile reinforcement learning aims to maximise cumulative rewards. All three also undergo a training period. Reinforcement learning undergoes this process when the agent learns how interactions with an environment affect its score, whilst the other two both have training data to improve its performance before evaluating a test set of data.

3.2 Neural Networks

Both supervised and unsupervised learning can take advantage of neural networks in their computation. A branch of reinforcement learning called deep reinforcement learning also takes advantage of neural networks.

The foundation of neural networks was set in the 1940's when McCulloch and Pitts laid the foundation for the MuCulloch–Pitts Neuron. This was heavily inspired by the biological neuron and laid inspiration for Rosenblatt in the 1950's to create the perceptron, a single-layer neural network capable of learning simple patterns. In 1969, Minsky and Papert wrote a book, 'Perceptrons', which was a harsh critique of the perceptrons, mainly arguing that there was no algorithm to train even a simple Boolean function, XOR. This led to an 'AI Winter' in which little research was conducted on multiple layered neural networks. It took ten years for the backpropagation algorithm to be made and as technology advanced, a resurgence in interest in neural networks occurred [14, pp. 4-9].

Neural networks are constructed from units known as perceptrons. Perceptrons activate when a specific linear combination of inputs, $\boldsymbol{w}^T\boldsymbol{x}$, surpasses a threshold $-b$, akin to tackling the problem of binary classification using a linear function.

Given a vector of weights $\boldsymbol{w} \in \mathbb{R}^d$, input vectors $\boldsymbol{x} \in \mathbb{R}^d$ and a bias term $b \in \mathbb{R}$, the binary classifier would give the output,

$$h_{\boldsymbol{w},b}(\boldsymbol{x}) = g(\boldsymbol{x}) = \mathbf{1}\left\{\boldsymbol{w}^T\boldsymbol{x} + b > 0\right\} = \begin{cases} 1 & \boldsymbol{w}^T\boldsymbol{x} + b > 0 \\ 0 & \boldsymbol{w}^T\boldsymbol{x} + b \leq 0 \end{cases}$$

In this case, $g(\boldsymbol{x})$ is chosen to be the the indicator function, $\mathbf{1}$. The choice of function, $g(\boldsymbol{x})$, is known as the activation function. There are many possible functions that can be chosen. One of the most common choices is the sigmoid function [15, p. 119].

In order to determine the weights and bias term from data, we will need to employ optimisation techniques. It is beneficial to approximate the indicator with a smooth and easily differentiable function such as the sigmoid:

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}.$$

The most desirable feature of activation functions are that they are easily differentiable and are smooth [15, p. 121]. This facilitates simpler calculation of weights and biases during backpropagation [16, pp. 150-152][17, pp. 861-862].

Neural networks are constructed from layers of perceptrons interconnected between the current and subsequent layers. We define, as in Higham and Higham [17, pp. 864], a linear map $\boldsymbol{W}^k : \mathbb{R}^{d_{k-1}} \to \mathbb{R}^{d_k}$ and a bias vector $\boldsymbol{b}^k \in \mathbb{R}^{d_k}$, to the k-th layer where d_k are the number of perceptrons in layer k. Applying an activation function componentwise (in this case the sigmoid activation function was chosen), we have a map, $\sigma\left(\boldsymbol{W}^k\boldsymbol{x} + \boldsymbol{b}^k\right)$ from layer $k-1$ to k.

A neural network with ℓ layers is a function of the form $\boldsymbol{a}^\ell(\boldsymbol{x})$, where the $\boldsymbol{a}^k$ are defined recursively as

$$\begin{aligned} \boldsymbol{a}^1(\boldsymbol{x}) &= \sigma\left(\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1\right) \\ \boldsymbol{a}^{k+1}(\boldsymbol{x}) &= \sigma\left(\boldsymbol{W}^{(k+1)}\boldsymbol{a}^k(\boldsymbol{x}) + \boldsymbol{b}^{k+1}\right), \quad 1 \leq k < \ell, \end{aligned}$$

and $\boldsymbol{W}^k \in \mathbb{R}^{d_k \times d_{k-1}}, \boldsymbol{b}^k \in \mathbb{R}^{d_k}$ for $1 \leq k \leq \ell$ (with $d_0 = d$) [18, p. 2102]. The initial layer serves as the input layer, and the final layer acts as the output layer. Any layers situated between the input and output layers are referred to as hidden layers. Different hidden layers may have different activation functions. These characteristics collectively constitute what is known as an architecture, shown in Figure 2.

We can train a neural network on data $(\boldsymbol{x}_i, \boldsymbol{y}_i), i \in \{1, \ldots, n\}$, by minimizing the empirical risk, $\hat{R}(h)$, with respect to a loss function, $L\left(h\left(\boldsymbol{x}^i\right), \boldsymbol{y}^i\right)$. The empirical risk of a function $h : \mathcal{X} \to \mathcal{Y}$ is the average loss over the training data, $\hat{R}(h) := \frac{1}{n}\sum_{i=1}^n L\left(h\left(\boldsymbol{x}^i\right), \boldsymbol{y}^i\right)$. L is a loss function, of which there are many possibilities [19, pp. 988-989].

To minimise the empirical risk, if L is almost everywhere differentiable, we utilise gradient

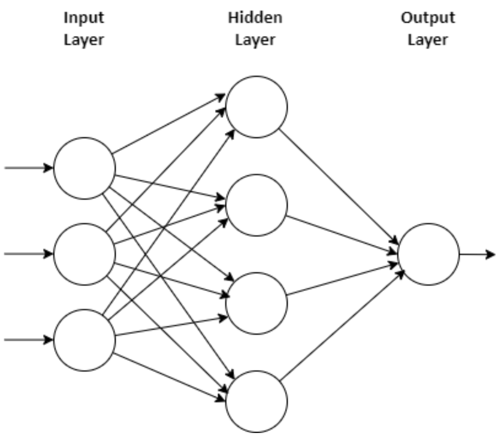

Figure 2: Neural Network Architecture. Every layer represents a linear mapping of the previous layer's outputs, succeeded by an activation function. Each arrow symbolizes a weight.

descent. This can be computed through the application of the chain rule, a process referred to as backpropagation. Firstly a few definitions must be made.

Letting $\boldsymbol{W}$ and $\boldsymbol{b}$ be the concatenation of all the weight matrices and bias vectors, we can subsequently denote w_{ij}^k the (i,j)-th entry of the k-th matrix, and b_i^k the i-th entry of the k-th bias vector. Empirical risk is then defined as $f(\boldsymbol{W},\boldsymbol{b}) := \frac{1}{n}\sum_{i=1}^{n} L\left(\boldsymbol{a}^{\ell}\left(\boldsymbol{x}_i\right), \boldsymbol{y}_i\right)$, which we are aiming to minimise.

One way of optimising a function is using gradient descent, which we will further define in Section 3.3. To calculate the gradient of a function in the following format: $f_i(\boldsymbol{W},\boldsymbol{b}) := L\left(\boldsymbol{a}^{\ell}\left(\boldsymbol{x}_i\right), \boldsymbol{y}_i\right)$, we must manipulate the equations.

We start with the forward pass [20, pp. 11-14]. Setting $\boldsymbol{x} = \boldsymbol{x}_i$ and $\boldsymbol{y} = \boldsymbol{y}_i$, and also writing $\boldsymbol{a}^0 := \boldsymbol{x}$, for $k \in \{1,\ldots,\ell\}$ we let $\boldsymbol{z}^k = \boldsymbol{W}^k\boldsymbol{a}^{k-1} + \boldsymbol{b}^k$, and $\boldsymbol{a}^k = \sigma\left(\boldsymbol{z}^k\right)$ as before. Our final output of the neural network is $\boldsymbol{a}^{\ell}$ for input $\boldsymbol{x}$.

Then we create a cost function, $J = J(\boldsymbol{W},\boldsymbol{b}) = L\left(\boldsymbol{a}^{\ell},\boldsymbol{y}\right)$ for the loss function with our desired parameters. For every layer k and coordinate $j \in \{1,\ldots,d_k\}$, the sensitivities are defined as

$$\delta_j^k := \frac{\partial J}{\partial z_j^k},$$

where z_j^k is the j-th coordinate of z^k. Denote $\boldsymbol{\delta}^k \in \mathbb{R}^{d_k}$ the vector of δ_j^k for $j \in \{1,\ldots,d_k\}$.

In context, sensitivities indicate how much the loss function will change with respect to small changes in the activations, $\boldsymbol{a}^k$, of the neurons in that layer. This is crucial in calculating the gradients of the loss functions in order to update the weights using optimisation techniques such as gradient descent. The partial derivatives of J can be computed in terms of these quantities.

Theorem 1 (Backpropogation)**.** *For a neural network with ℓ layers and $k \in \{1,\ldots,\ell\}$,* *we have*

$$\frac{\partial J}{\partial w_{ij}^k} = \delta_i^k a_j^{k-1}, \quad \frac{\partial J}{\partial b_i^k} = \delta_i^k$$

for $i, j \in \{1, \ldots, d_k\}$*. Additionally, the sensitivities* δ_i^k *can be computed as follows:*

$$\boldsymbol{\delta}^\ell = \sigma'\left(\boldsymbol{z}^\ell\right) \circ \nabla_{\boldsymbol{a}^\ell} L\left(\boldsymbol{a}^\ell, \boldsymbol{y}\right), \quad \boldsymbol{\delta}^k = \sigma'\left(\boldsymbol{z}^k\right) \circ \left(\boldsymbol{W}^{k+1}\right)^T \boldsymbol{\delta}^{k+1}$$

for $k \in \{1, \ldots, \ell - 1\}$*.*

The primary objective of conducting backpropagation is to update the weights and biases in a way that minimises the loss function. A proof can be found in Higham and Higham [17, pp. 869-873]. After acquiring the weights and biases, the neural network can undergo testing using test data.

3.3 Optimisation Methods

As illustrated in the backpropagation algorithm, gradient descent is an important tool in optimisation. We will investigate both gradient descent and stochastic gradient descent. It is important to note that there are other tools used in particular cases for optimisation in machine learning contexts, such as the Karush–Kuhn–Tucker conditions, but we will not focus on these.

An analytical (non–iterative) method to calculate the global minimum of a function or to calculate the weights for the cost function is to use the normal equation. This is primarily used in mulivariate linear regression problems.

If θ is the cost function minimization value, y is the vector of target values and X is the features matrix, we can evaluate the problem as follows. The aim is to minimise the cost function, or equivalently to make $X\theta \approx y$, so using mean squares error, we attain a cost function, $J(\theta) = ||y - X\theta||^2$, or similarly $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$, where m are the number of features [21]. By expanding and considering the partial derivatives with respect to θ, we achieve the formula for the normal equation:

$$\theta = (X^T X)^{-1} X^T y = X^\dagger y$$

where $X^\dagger = (X^T X)^{-1} X^T$ is the Moore–Penrose pseudoinverse.

There are several problems using this method in practice. The primary issue is the complexity time, with computations taking $O(n^3)$ time. For larger matrices this is not desirable. For large datasets, storing the matrices can also cause problems, and when multiplying the matrices, small perturbations in the data can lead to large errors. Additionally, the normal equation requires the feature matrix to be full rank for the matrix inversion to be well-defined which may not be the case [22, pp. 355-358].

For these reasons, gradient descent is more widely used. It is an optimization algorithm performed iteratively based on a convex function to find its local minimum. The intuitive idea is that given our cost function, we want to travel down the steepest path at a given point until we reach the minimum. Given a learning rate hyperparameter α, the minimum is found by iteratively performing:

$$\theta_{k+1} = \theta_k - \alpha \nabla J(\theta) = \theta_k - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x^{(i)}$$

or equivalently for each parameter, $\theta^{(i)}$, we perform the update formula $\theta_{k+1}^{(i)} = \theta_k^{(i)} - \alpha \frac{\partial}{\partial \theta^{(i)}} J(\theta)$. This method in particular is known as batch gradient descent [23, p. 2][22, pp. 356]. Algorithm 1 (Section 8, Appendix) is continued until we reach a threshold, ϵ in which the difference between succeeding values of θ are less than ϵ.

The learning rate, α, indicates the step size and must be fine tuned for this method to work successfully. A learning rate that is too aggressive may sacrifice stability for speed, possibly resulting in erratic behaviour, while a learning rate that is too conservative may sacrifice speed for stability and could lead to stagnation in optimisation. The value α is usually determined through experimentation. Some possible techniques are random sampling and systematically evaluating performance of different learning rates using cross–validation methods [21, p. 2].

Although gradient descent is useful, there are some limitations. The cost function must be differentiable almost everywhere in order for the algorithm to work. The method is also susceptible to getting stuck at plateaus where the gradient is close to 0, and saddle points in which the gradient is 0 but not optimal. Additionally the initial model parameters can strongly influence the performance, leading to sub-optimal or even divergent outcomes [24, pp. 10-11]. The algorithm is also computationally expensive to run, especially for large datasets and deep neural networks as every iteration of the algorithm uses every data point in the training data.

Intuitively, to solve specifically the last issue, we want an algorithm that would only iterate through some of the data points, yet arrives at a correct conclusion. Stochastic gradient descent aims to fix this issue by considering only one data point at a time [23, p. 2][25, pp. 123-130][22, p. 357]. At each step, a randomly chosen data point, $(x^{(i)}, y^{(i)})$, is chosen from which the gradient is calculated. The stochastic nature of this method is the way in which the data point is chosen as noise is injected through the variability of the random sampling. We then achieve a similar update rule:

$$\theta_{k+1} = \theta_k - \alpha \nabla J \left(\theta \mid x^{(i)}, y^{(i)} \right)$$

A proof of convergence can be found in Zhao [25, pp. 130-132]. As stochastic gradient descent (written in Algorithm 2, Section 8, Appendix) is more exploratory than gradient descent from the added noise, and explores solution space more extensively, there is higher chance of finding the global minimum and escaping saddle points. A large benefit of using stochastic gradient descent methods are that it can improve convergence speed and generalization, however due to the added variability in the optimisation process it can be harder to reproduce results, and is more sensitive to hyperparameters. Although stochastic gradient descent often attains a solution 'close' to the minimum much faster, it may not actually converge to the minimum and oscillate around it instead [26, pp. 2-6]. An alternative approach named mini–batch gradient descent can also be used [23, p. 3], which is similar to stochastic gradient descent. The main difference is that it considers a collection of samples, or batch, $\mathcal{B}$, chosen randomly rather than a singular data point at each step.

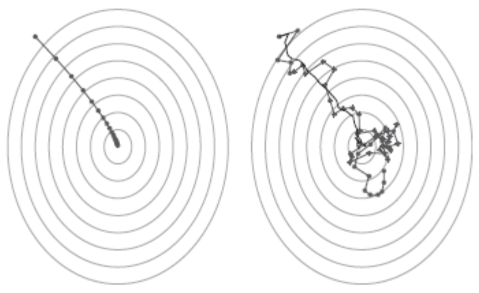

Figure 3: Example of Gradient Descent (left) and Stochastic Gradient Descent (right) [27, p. 191]

Many of these issues and themes such as analytic and iterative approaches, optimisation, and hyperparameter tuning all emerge when tackling issues related to reinforcement learning.

3.4 Transfer Learning

The final piece of background is regarding transfer learning. This is a useful tool in machine learning and in particular in reinforcement learning. We will go through a brief introduction into the topic and its definitions as it is a significant and rapidly growing branch of machine learning.

The concept, like other machine learning topics, is intuitively simple. The idea is to create a way in which the knowledge of one task can be used to help increase the performance in another related task. This has the potential of increasing the efficiency of learning a task, or for an agent to learn [28, p. 111]. The three main questions faced by researchers in regard to transfer learning are: what information is useful to be transferred to the target, what is the best way to transfer this information, and how do we avoid transferring information that is detrimental to the target [29, pp. 3-4] [30, p. 4]. To define transfer learning, we must introduce two concepts first.

Definition 2 (Domain). *[29, p. 3] A domain, $\mathcal{D} = \{\mathcal{X}, \mathbb{P}(X)\}$, is a space consisting of all the features, $\mathcal{X}$, and a marginal probability distribution $\mathbb{P}(X)$, where X consists of a sample of features $\{x_1, x_2, \ldots, x_n\} \in \mathcal{X}$.*

Domains differ if their feature spaces or their marginal probability distributions differ. We now define the next concept.

Definition 3 (Task). *[29, p. 3] Given a domain, we define a task, $\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$, that consists of a label space, $\mathcal{Y}$, and a predictive function $f(\cdot)$. This function is not observed but can be learned from training data pairs $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$.*

From a probabalistic perspective, we may write $f(\cdot)$ as $\mathbb{P}(Y|X)$. Two tasks are different if they have different label spaces or different conditional probability distributions.

Definition 4 (Transfer Learning). *[31, p. 3] Given a source domain $\mathcal{D}_S$ and source learning task $\mathcal{T}_S$, and given a target domain $\mathcal{D}_T$ and target learning task $\mathcal{T}_T$, transfer learning utilises knowledge learned in the source domains to aim to improve the learning of the target conditional probability distribution f_T or $\mathbb{P}(Y_T|X_T)$, where $\mathcal{D}_S \neq \mathcal{D}_T$ and $\mathcal{T}_S \neq \mathcal{T}_T$.*

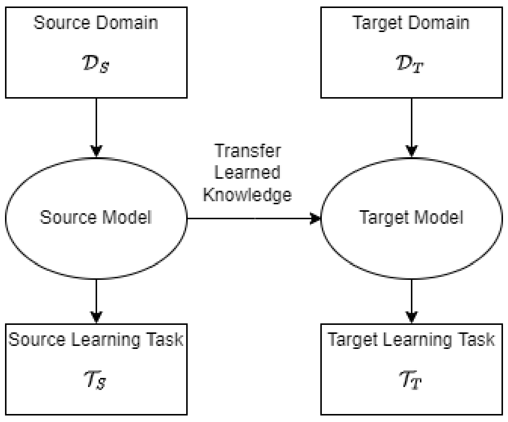

Figure 4: Transfer Learning definition expressed as a diagram.

From this definition, different types of transfer learning emerge. Homogeneous transfer learning is where the sample of features and samples of labels agree, however either or both of the marginal probability distributions and the predictive functions disagree. Heterogeneous transfer learning occurs when either or both of the sample of features and the sample of labels disagree.

We will not explore transfer learning further, however further information can be found in Weiss et al. [32] that explores different branches of transfer learning. Further information specific to transfer learning and its relation to reinforcement learning can be found in Torrey and Shavlik [33] and Zhu et al. [30].

4 Mathematics of Reinforcement Learning

Now we have the background knowledge, we will investigate some mathematical concepts behind reinforcement learning. We will begin by defining the framework of reinforcement learning. Following that, we explore different techniques to optimise the agent. Finally we will look at the exploration vs. exploitation problem.

4.1 Markov Decision Processes

Reinforcement learning is modelled as a sequential decision making process. Markov Decision Processes (MDPs) provide a framework for this type of learning. Before defining MDPs, we will remind ourselves of what a Markov Process is.

Definition 5 (Markov Process)**.** *[34, pp. 49,384] Given a state S_{t+1} at time $t+1$ in a finite sequence $S_0, S_1, S_2, \dots, S_N$, the sequence has the Markov property, if and only if*

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_0, S_1, S_2, \dots, S_t].$$

Moreover, we define the state transition probability $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$ where the state transition matrix $\mathcal{P}$ defines transition probabilities from all states s to s'. The sum of each row of $\mathcal{P}$ sum to 1.

Often we summarise this definition with the statement 'the future is independent of its past'. We now formulate Markov Decision Processes as follows.

Definition 6 (Markov Decision Processes)**.** *[35, §12.14] A Markov Decision Process is a 5–tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma)$ with*

- *$\mathcal{S}$ is a finite set of states*
- *$\mathcal{A}$ is a finite set of actions. $\mathcal{A}_{s_t} \in \mathcal{A}$ are the actions available given state $s_t \in \mathcal{S}$*
- *$\mathcal{P}$ is a state transition probability matrix, $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$*
- *$\mathcal{R}$ is a reward function, $\mathcal{R}^a_s = \mathbf{E}[R_{t+1} \mid S_t = s, A_t = a]$*
- *γ is a discount factor $\gamma \in [0,1]$ representing the difference in importance between present and future rewards*

At each time step, the agent must decide what action it must take given the current state. This mapping from the states to probabilities of selecting a possible action is known as a policy.

Definition 7 (Policy)**.** *[8, p. 68] A policy is a mapping from the state space to the action space, $\pi_t : \mathcal{S} \to \mathcal{A}$, where $\pi_t(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$.*

Sometimes we may see policies written as π_θ, where θ is a parameterisation of the weights and biases in a neural network. The goal of a MDP is to optimise a policy π, maximising a cumulative function of the random rewards. The policy therefore must balance whether to focus on immediate rewards or on potential future rewards. We define a return function, G_t to express this purpose using the discount factor [8, pp. 73-74].

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

If γ is close to 0 we have a myopic evaluation, and if it is close to 1, we have a far–sighted evaluation.

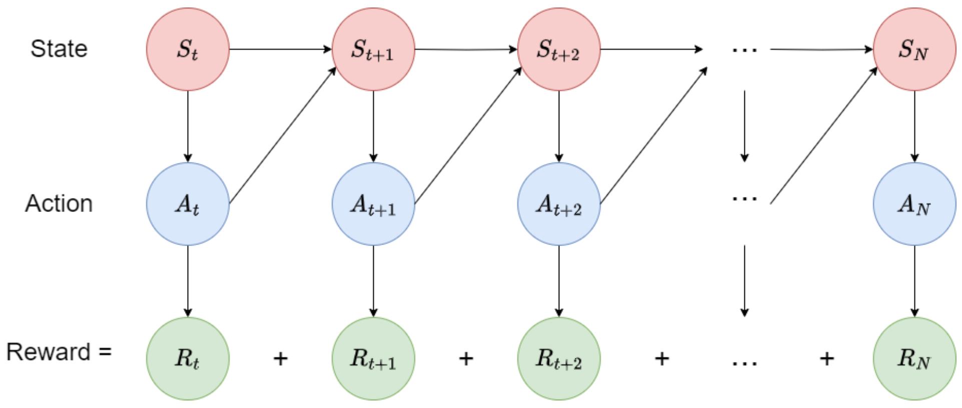

Figure 5: Markov Decision Processes

To calculate the expected return given a state, s, and following a particular policy, π, for an MDP we can use the following definition.

Definition 8 (State–Value Function for a policy, π). *[8, p. 84] The state–value function $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π*

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right]$$

In essence, the state–value function tells us how good it is to be in a certain state. Similarly, we can define the expected return if we take an action a.

Definition 9 (Action–Value Function for a policy, π). *[8, p. 84] The state–value function $v_\pi(s)$ of an MDP is the expected return taking an action a starting from state s, and then following policy π*

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid A_t = a, S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid A_t = a, S_t = s \right]$$

These are often called Q–Values. These tell us how good the actions we take are, given a state. Following the Backup diagrams in Figure 6 (Adapted from Sutton [8, pp. 84-86]), we see that we can also define $v_\pi(s)$ as the sum of two different Q–Values. We can express this as $v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a)$. Similarly we can write $q_\pi(s,a) = R_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s)$. These forms are useful as we have expressed our V–values in terms of our Q–values.

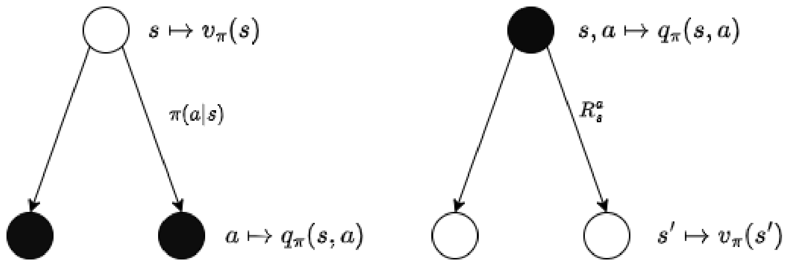

Figure 6: Backup Diagrams for a MDP. On the left for $v_\pi(s)$ and the right for $q_\pi(s,a)$. The diagrams show relationships that form the basis of updates in states and actions

Substituting the equations into one another we are left with two recursive equations that form the Bellman Equations.

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s) \right) \tag{1}$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a) \tag{2}$$

4.2 Bellman Equation and Optimal Value Functions

We have formulated the Bellman Equations for a given MDP to find its state-value function and state-action value function. We now need to utilise these to decide what the best way to behave in an MDP is.

Before continuing, we will first see another formulation of the Bellman Equations. We can rearrange Definition 8 to achieve an recursive formula which we can solve. Immediately we obtain (Adapted from [25, p. 31])

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] & (3) \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right]. & (4) \end{aligned}$$

Similarly we obtain $q_\pi(s,a) = \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid A_t = a, S_t = s]$. Both (3) and (4) are useful equations and are equivalent using the linearity of expectation. Equation 4 is known as the Bellman Expectation Equation.

Rewriting Equation 1 as $v_\pi(s) = r_\pi(s) + \gamma \sum_{s' \in \mathcal{S}} p_\pi(s' \mid s) v_\pi(s)$, where $r_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) R_s^a$ and $p_\pi(s' \mid s) = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a$, we may express the Bellman Equation in matrix form[25, p. 36]. Letting the states be indexed s_i for $i = 1, \ldots, n$, where $n = |\mathcal{S}|$, for state s_i we may write

$$v_\pi(s_i) = r_\pi(s_i) + \gamma \sum_{s_j \in \mathcal{S}} p_\pi(s_j \mid s_i) v_\pi(s_i) \tag{5}$$

which we may subsequently write in matrix form as

$$v_\pi = r_\pi + \gamma \mathcal{P}_\pi v_\pi \tag{6}$$

where $\mathcal{P}_\pi$ is our transition matrix.

We can reach a similar form for action–value functions. In this form we may calculate our state–value function in two possible ways. This process is known as policy evaluation.

One way is concisely expressing the state–value function as a closed form solution: $v_\pi = (I - \gamma \mathcal{P}_\pi)^{-1} r_\pi$. We may write it in this way as $(I - \gamma \mathcal{P}_\pi)$ is always invertible via the Gershgorin Circle Theorem (see [25, p. 38]). The main issue is that the time complexity is $O(n^3)$ so direct solutions would only be possible for smaller Markov reward processes. A solution is using an iterative method. By applying $v_{k+1} = r_\pi + \gamma \mathcal{P}_\pi v_k$ iteratively for $k = 0, \ldots,$ we eventually have convergence in polynomial time [36, pp. 2-3]. A more general proof of convergence, uniqueness and existence of this method will be shown in Section 4.3, however we will briefly prove convergence using an alternative strategy.

Lemma 10. *(Adapated from [25, p. 39]) The value iteration algorithm for the Bellman Equation converges.*

Proof. Setting $\delta_k = v_k - v_\pi$, our aim is to show $\delta_k \to 0$. By rearranging and substituting into our iteration, $v_{k+1} = r_\pi + \gamma \mathcal{P}_\pi v_k$, we obtain

$$\delta_{k+1} + v_\pi = r_\pi + \gamma \mathcal{P}_\pi (\delta_k + v_\pi)$$

This may be further rearranged to $\delta_{k+1} = \gamma \mathcal{P}_\pi \delta_k - v_\pi + (r_\pi + \gamma \mathcal{P}_\pi v_\pi) = \gamma \mathcal{P}_\pi \delta_k - v_\pi + v_\pi$. As a result, we have $\delta_{k+1} = \gamma \mathcal{P}_\pi \delta_k = \gamma^2 \mathcal{P}_\pi^2 \delta_{k-1} = \cdots = \gamma^{k+1} \mathcal{P} \mathcal{P}_{\pi_{k+1}} \delta_0$. Using properties of the transition matrix, we know $0 \leq \mathcal{P}_{\pi_{k+1}} \leq 1$ for all $k \in \mathbb{N}$, and as $\gamma \in [0, 1]$, $\gamma \to 0$ as $k \to \infty$. As a result, $\delta_k \to 0$ proving convergence.

□

Now we have the Bellman Equations to determine how well our policies are, we are faced with the question whether there is an optimal policy or not. To answer this question, we must define some terms.

Definition 11 (Optimal Policy and Optimal State–Value Function)**.** *[8, p. 75] A policy*

π^* is optimal if $v_{\pi^*}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$ and any policy π. If multiple π^* exist, they share the same state–value function, $v^*(s) = \max_\pi v(s)$.

Additionally, if we write $q^*(s,a) = \max_\pi q(s,a)$ it is clear that

$$q^*(s,a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v^*(S_{t+1}) \mid A_t = a, S_t = s\right].$$

Because our v^* and q^* must also follow our Bellman Equation (Equations 1 and 2), we obtain the following theorem.

Theorem 12 (Bellman Optimality Equations)**.** *[8, p. 90] The optimal values must satisfy the following:*

$$v^*(s) = \max_a R_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^*(s') \tag{7}$$

$$q^*(s,a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q^*(s',a') \tag{8}$$

Proof. The proof follows an argument similar to Equation 4 and using $\sum_a \pi(a \mid s) = 1$. $\square$

Moreover, we must intuitively have $v^*(s) = \max_{a \in \mathcal{A}} q^*(s,a)$. It becomes apparent that the best policy is greedy [8, p. 91]. Again using $\sum_a \pi(a \mid s) = 1$, we see

$$\sum_{a \in \mathcal{A}} \pi(a \mid s) q(s,a) \leq \sum_{a \in \mathcal{A}} \pi(a \mid s) \max_{a \in \mathcal{A}} q(s,a) = \max_{a \in \mathcal{A}} q(s,a) = q^*(s,a)$$

With equality achieved when

$$\pi^*(a \mid s) = \begin{cases} 1 & a = \arg\max_a q(s,a) \\ 0 & \text{otherwise} \end{cases}$$

This shows us that the key to the best policy is finding the best action–value function and state–value function. There are several ways to solve the Bellman Optimality Equations (Equations 7 and 8), which we shall explore. The two we will focus on are Dynamic Programming methods and Temporal Difference methods.

4.3 Dynamic Programming

Dynamic Programming refers to a collection of algorithms that aim to find the optimal policy by creating subproblems. Often these subproblems overlap, but previous computations are cached and are reused. These problems are usually recursive in nature, for example finding a certain value in the Fibonacci sequence [37, pp. 1-3], and often significantly reduce the time complexity of a problem.

Before we try to solve the Bellman Optimality Equations (Equations 7 and 8) with this method, it is crucial that we must first ensure there is only one unique solution to the problem. We must also ensure that a solution exists. To do this, we will utilise the Contraction Mapping Theorem (Banach Fixed–Point Theorem).

Theorem 13 (Contraction Mapping Theorem)**.** *[38, p. 50] If (X, d) is a complete metric space and $T : X \to X$ is a contraction mapping with modulus γ, then*

1. *T has exactly one fixed point, x^* in X*
2. *For any $x_0 \in X$, the sequence $\{x_i\}_{i\in\mathbb{N}}$ defined by $x_{k+1} = T(x_k)$ converges to x^**

If we apply this to theorem to the Bellman Optimality Equations, we have proven existence and uniqueness and shown an iterative method holds.

Defining $v_{k+1} = T_\pi(v_k) = r_\pi + \gamma\mathcal{P}_\pi v_k$, similar to Equation 6. T_π is known as the Bellman Operator. Using the infinity norm, $\|\cdot\|_\infty$, we must show T_π is a contraction mapping to prove our statement.

Theorem 14 (Uniqueness and Existence of an optimal policy)**.** *(Adapted from [25, pp. 55-56] and [39, §3 pp. 34-42]) The Bellman Operator T_π for a Markov Decison Process with policy π is a γ–contraction. Moreover, the sequence $\{v_i\}_{i\in\mathbb{N}}$ converges to a unique v^*, defined by iterative over the Bellman Operator.*

Proof.

$$\begin{aligned}\|T_\pi(u) - T_\pi(v)\|_\infty &= \|r_\pi + \gamma\mathcal{P}_\pi u - r_\pi - \gamma\mathcal{P}_\pi v\|_\infty \\ &= \|\gamma\mathcal{P}_\pi u - \gamma\mathcal{P}_\pi v\|_\infty \\ &= \gamma\|\mathcal{P}_\pi(u - v)\|_\infty \\ &\le \gamma\|\mathcal{P}_\pi\|_\infty\|u - v\|_\infty = \gamma\|u - v\|_\infty\end{aligned}$$

We have shown T_π is a contraction mapping, so applying the Contraction Mapping Theorem, $\{v_i\}_{i\in\mathbb{N}}$ converges to a unique v^*. □

As mentioned in Section 4.2, one such way to find the optimal policy is using value iteration [40, pp. 25-26]. The algorithm, as written in Algorithm 3 (Section 8, Appendix), is guaranteed to converge due to Theorem 14. In each iteration of the algorithm, there are two steps: a value update step, $v(s) \leftarrow \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma v(s')]$, and a policy update step, $\pi^*(s) \leftarrow \arg\max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma v(s')]$. The algorithm continues until it reaches a threshold, ϵ at which point the algorithm improves marginally and we can say $\pi \approx \pi^*$

Another prominent algorithm used is called policy iteration. The aim is to start by choosing an arbitrary policy π and iteratively evaluate and improve the policy until convergence. The Algorithm, shown in Algorithm 4 (Section 8, Appendix), is comprised of more steps than value iteration. The first step is a policy evaluation step, calculated from the last step of the Bellman Optimality Equation for $v(s)$. The second step is the policy improvement step in which we update the policy by $\pi(s) \leftarrow \arg\max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma v(s')]$, and see if this policy performs better than the previous step [25, pp. 72-73]. One way to visualise this algorithm is through the following diagram [8, p. 96]:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \dots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} v^*$$

where $\xrightarrow{\text{E}}$ represents the policy evaluation step, and $\xrightarrow{\text{I}}$ represents the policy improvement step. In order to use this Algorithm, we will check the policy improvement step to see if it achieves its desired goal.

Lemma 15 (Policy Improvement). *[25, pp 73-74] If $\pi_{k+1}(s) = \arg\max_\pi \sum_{s'\in\mathcal{S}}(r_\pi + \gamma\mathcal{P}_\pi v_{\pi_k})$, then $v_{\pi_{k+1}} \geq v_{\pi_k}$*

Proof. $v_{\pi_{k+1}}$ and v_{π_k} must satisfy the Bellman Equations:

$$v_{\pi_{k+1}} = r_\pi + \gamma\mathcal{P}_\pi v_{\pi_{k+1}} \qquad v_{\pi_k} = r_\pi + \gamma\mathcal{P}_\pi v_{\pi_k}.$$

As $\pi_{k+1}(s) = \arg\max_\pi \sum_{s'\in\mathcal{S}}(r_\pi + \gamma\mathcal{P}_\pi v_{\pi_k})$, we have

$$r_{\pi_{k+1}} + \gamma\mathcal{P}_{\pi_{k+1}} v_{\pi_{k+1}} \geq r_{\pi_k} + \gamma\mathcal{P}_{\pi_k} v_{\pi_k}$$

. From here we can see

$$\begin{aligned} v_{\pi_k} - v_{\pi_{k+1}} &= (r_{\pi_k} + \gamma\mathcal{P}_{\pi_k} v_{\pi_k}) - (r_{\pi_{k+1}} + \gamma\mathcal{P}_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq (r_{\pi_{k+1}} + \gamma\mathcal{P}_{\pi_{k+1}} v_{\pi_k}) - (r_{\pi_{k+1}} + \gamma\mathcal{P}_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq \gamma\mathcal{P}_{\pi_{k+1}}(v_{\pi_k} - v_{\pi_{k+1}}) \end{aligned}$$

Continuing in an argument similar to that in Lemma 10, we see that $v_{\pi_k} - v_{\pi_{k+1}} \leq \gamma\mathcal{P}_{\pi_{k+1}}(v_{\pi_k} - v_{\pi_{k+1}}) \leq \gamma^n\mathcal{P}^n_{\pi_{k+1}}(v_{\pi_k} - v_{\pi_{k+1}}) \leq \lim_{n\to\infty} \gamma^n\mathcal{P}^n_{\pi_{k+1}}(v_{\pi_k} - v_{\pi_{k+1}}) = 0$. □

Now we can prove convergence of this method.

Theorem 16. *The policy iteration algorithm converges.*

Proof. (Adapted from [25, p. 75]) The idea is to show that the policy iteration algorithm converges faster than the value iteration algorithm.

To prove the convergence of $\{v_{\pi_k}\}_{k=0}^\infty$, we introduce another sequence $\{v_k\}_{k=0}^\infty$ generated by $v_{k+1} = T(v_k) = \max_\pi (r_\pi + \gamma P_\pi v_k)$. This is the value iteration algorithm, and so we know v_k converges to v^* from Theorem 14.

We proceed using induction. For $k = 0$, we can always find a v_0 such that $v_{\pi_0} \geq v_0$ for any π_0. For $k \geq 0$, suppose that $v_{\pi_k} \geq v_k$. Then for $k+1$, we have

$$\begin{aligned}
v_{\pi_{k+1}} - v_{k+1} &= \left(r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}\right) - \max_{\pi} \left(r_\pi + \gamma P_\pi v_k\right) \\
&\geq \left(r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}\right) - \max_{\pi} \left(r_\pi + \gamma P_\pi v_k\right) && \text{(Using Lemma 15)} \\
&= \left(r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}\right) - \left(r_{\pi'_k} + \gamma P_{\pi'_k} v_k\right) && \text{(Letting } \pi'_k = \arg\max_{\pi} \left[r_\pi + \gamma P_\pi v_k\right]) \\
&\geq \left(r_{\pi'_k} + \gamma P_{\pi'_k} v_{\pi_k}\right) - \left(r_{\pi'_k} + \gamma P_{\pi'_k} v_k\right) \\
&= \gamma P_{\pi'_k} \left(v_{\pi_k} - v_k\right).
\end{aligned}$$

Since $v_{\pi_k} - v_k \geq 0$ and $P_{\pi'_k \in [0,1]}$, we have $P_{\pi'_k}\left(v_{\pi_k} - v_k\right) \geq 0$ and so $v_{\pi_{k+1}} - v_{k+1} \geq 0$. Therefore, we have shown $v_k \leq v_{\pi_k} \leq v^*$ for any $k \geq 0$. Since v_k converges to v^*, v_{π_k} also converges to v^*. $\square$

Both these dynamic programming techniques are incredibly useful for optimising our policies. Value iteration updates the state–value function, whereas policy iteration updates the policies to find the optimal policy. Value iteration can be seen as a truncated form of policy iteration, combining two phases of the algorithm, [8, p. 98] and is much simpler. The drawback is that it is slower to run and more computationally expensive in comparison to policy iteration. The main reason for this is that it runs every action for each possible state to achieve the optimal state–value function.

Other dynamic programming algorithms exist such as truncated policy iteration [25, § 4.3], however we will not explore these in this report.

4.4 Temporal Difference

Before moving onto the Temporal Difference methods to find the best policy, we will briefly introduce Monte Carlo methods [39, §4 pp. 6-11][8, pp. 114-115].

In most real–life scenarios, the environment is unknown so a method called Monte Carlo is used to sample data for estimating the state–value function. The state–value function can be approximated by the mean of the return instead of the expectation. If we let $S(S_t)$ denote the sum of our return function, G_t, at time t, and $N(S_t)$ the first visit number of state S_t, we can model $V(S_t)$ as $v(S_t) = \frac{S(S_t)}{N(S_t)}$. $V(S_t)$ tends to $v_\pi(s)$ as $N(S_t) \to \infty$. Each return is an independent, identically distributed estimate of $v_\pi(s)$ with finite variance and so this can be proven by applying the Law of Large Numbers [8, p. 115][25, p. 91].

Through simplification and rearrangement, we have the iterative Monte Carlo update formula: $v_{k+1}(s) = v_k(s) + \alpha(G_t - v_k(s))$, where α is the learning rate [8, p. 144].

Temporal Difference methods start from this point, combining the iterative Monte Carlo update formula with concepts from Dynamic Programming. With help from Equation 4, we may write an update formula

$$v_{k+1}(s) = v_k(s) + \alpha(R_{t+1} + \gamma v_\pi(s') - v_k(s)). \tag{9}$$

This update formula is known as TD(0) [40, p. 30], and is the basis of temporal difference learning. Because Equation 9 bases its update in part on an existing estimate from Equation 4, we say that it is a bootstrapping method. Dynamic programming is also a bootstrapping method, so often it is said that temporal difference learning is a combination of dynamic programming and Monte Carlo methods. The term $r_{t+1} + \gamma v_t(s_{t+1})$ is called the TD target [8, p. 144]. We can label all the terms as follows:

$$\underbrace{v_{t+1}(s_t)}_{\text{new estimate}} = \underbrace{v_t(s_t)}_{\text{current estimate}} + \alpha_t(s_t)\,[\overbrace{(\underbrace{r_{t+1} + \gamma v_t(s_{t+1})}_{\text{TD target } \bar{v}_t}) - v_t(s_t)}^{\text{TD error } \delta_t}].$$

To derive TD(0), we can utilise a generalisation of the stochastic gradient algorithm from Section 3.3.

Theorem 17 (Robbins–Monro Algorithm). *[41, p. 406][42, p. 3] Given a function $g(\theta)$, the goal to find $g(\theta^*) = 0$ where θ^* is the root of the function can be found using the following iterative algorithm:*

$$\theta_{n+1} = \theta_n + \alpha_n \cdot g_n(\theta_n).$$

θ_n is the estimate at iteration n, α_n is the step size at iteration n and $g_n(\theta_n)$ is the gradient at iteration n. The algorithm holds under the conditions that $\sum_{n=1}^{\infty} \alpha_n = \infty$ and $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$. The algorithm iteratively updates the estimate θ_n based on the gradient $g_n(\theta_n)$ and the step size α_n until convergence.

The proof of convergence is beyond the scope of this report. It uses Dvoretzky's theorems and can be seen in Vajjha et al. [42]. We can now derive Equation 9 by solving the Bellman Expectation Equation (Equation 4).

Derivation of TD(0). (Adapted from [25, p. 136]) For a state s_t, define $g(v_\pi(s_t)) = v_\pi(s_t) - \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$. The Bellman Expectation Equation is equivalent to $g(v_\pi(s_t)) = 0$. Clearly we can model $g_n(\theta_n)$ in terms of the TD error, δ_t. By setting η as our noise term (the difference between expected value of v_π and observed value), in our situation we may express $g_n(\theta_n) = g(\theta) + \eta$ [25, p. 113] as a specific case of the Robbins–Monro Algorithm. Following this, we may write:

$$
\begin{aligned}
g_n\left(v_\pi\left(s_t\right)\right) = &= \underbrace{\left(v_\pi\left(s_t\right) - \mathbb{E}\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) \mid S_t = s_t\right]\right)}_{g(v_\pi(s_t))} \\
&+ \underbrace{\left(\mathbb{E}\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) \mid S_t = s_t\right] - \left[r_{t+1} + \gamma v_\pi\left(s_{t+1}\right)\right]\right)}_{\eta} \\
&v_\pi\left(s_t\right) - \left[r_{t+1} + \gamma v_\pi\left(s_{t+1}\right)\right].
\end{aligned}
$$

Solving for $g(v_\pi(s_t)) = 0$, we have

$$
\begin{aligned}
v_{k+1}(s) &= v_k(s) + \alpha(g_n(v_\pi(s_t))) \\
&= v_k(s) + \alpha(R_{t+1} + \gamma v_\pi(s') - v_k(s))
\end{aligned}
$$

$\square$

TD learning is model–free, unlike dynamic programming techniques, in which we do not know $\mathcal{P}^a_{ss'}$ or $\mathcal{R}^a_s$. The benefit of TD learning compared to Monte Carlo methods are that the variance is much lower [8, p. 148-149]. For this reason, TD methods take the best of Monte Carlo methods as we do not need to know about the environment and also Dynamic Programming methods as we utilise the Bellman Optimality Equations.

We will explore two different TD learning algorithms, SARSA and Q–Learning. SARSA is an on–policy algorithm, whereas Q–Learning is an off-policy algorithm. The difference is that in off-policy learning, the agent learns about the optimal policy while following a different policy, called a behavioural policy, and on–policy learning follows the current policy.

The SARSA algorithm, as described in Algorithm 6 (Section 8, Appendix), utilises a method called the ϵ–greedy algorithm. We will investigate this more in Section 4.6. The algorithm iterates through the quintuple of events $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$, giving rise to the name SARSA [8, p. 155]. The heart of the algorithm, $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$, is similar to TD(0) in Equation 9, but for the action–value function. This particular version is also equivalent to the Bellman Equation (Equation 2). SARSA is particularly useful in non-stationary environments as the optimal policy will never be attainable. It is also useful if function approximation is used, because off-policy methods can diverge when this is used [40, p. 32].

The Q–Learning algorithm is an alternative popular approach, created by Watkins [9, p. 4]. Q-learning can directly estimate optimal action values and find optimal policies. The method is outlined in Algorithm 7 (Section 8, Appendix). Q–Learning looks deceptively similar to SARSA. The main difference is that the TD target of Q–Learning is $r + \gamma \max_{a'} Q(s', a')$ as opposed to $r + \gamma Q(s', a')$ in SARSA.

These algorithms are extremely useful in practice and will be seen in Section 6. A variation of Q–Learning will also be explored in Section 5.1 utilising neural networks. There are generalisations of these algorithms such as n–step TD learning and eligibility traces. More information can be found in Sutton [8, §7].

4.5 Policy Gradients

In Section 4.4 we saw that TD learning was model–free. Another model–free approach to optimising the Bellman Equations are policy gradients. We will briefly give an introductory insight into this family of solutions.

We will consider a stochastic, parameterized policy, π_θ, with the aim of maximising the expected return $J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$ [43], where $\tau \sim \pi_\theta$ denotes trajectories taken from the policy π_θ. Policy gradient methods assume the policy is represented using some function that is differentiable with respect to its parameters, θ.

The simplest way to optimise the policy is using gradient ascent. This is similar to gradient descent from Section 3.3, however instead of subtracting by the gradient function, we add it in order to maximise θ. The update rule is as follows: $\theta_{k+1} = \theta_k + \alpha\ \nabla_\theta J(\pi_\theta)|_{\theta_k}$. From this point, we must find what $\nabla_\theta J(\pi_\theta)|_{\theta_k}$ is.

Theorem 18 (Policy Gradient Theorem)**.** *(Adapted from [43] For any differentiable policy $\pi_\theta(s,a)$, for any of the policy objective functions $J(\theta)$, the policy gradient is*

$$\nabla_\theta J\left(\pi_\theta\right) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right) R(\tau)\right]$$

We will see a novel proof, based on [43] and [44, p. 325]

Proof. Utilising $\mathcal{P}_\theta(\tau)\nabla_\theta \log \mathcal{P}_\theta(\tau) = \nabla_\theta \mathcal{P}_\theta(\tau)$ and the gradient of environmental functions such as $R(\tau)$ are 0, we see that

$$\begin{aligned}
\nabla_\theta J\left(\pi_\theta\right) &= \nabla \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \sum_{\tau\sim\pi} \mathcal{P}_\theta(\tau) R(\tau) \\
&= \sum_{\tau\sim\pi} \nabla_\theta \mathcal{P}_\theta(\tau) R(\tau) \\
&= \sum_{\tau\sim\pi} \mathcal{P}_\theta(\tau) \nabla_\theta \log \mathcal{P}_\theta(\tau) R(\tau) + 0 \\
&= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \mathcal{P}_\theta(\tau) R(\tau)] \\
&= \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right) R(\tau)\right].
\end{aligned}$$

Where the final line comes from rewriting $P(\tau \mid \theta) = \rho_0\left(s_0\right) \prod_{t=0}^{T} P\left(s_{t+1} \mid s_t, a_t\right) \pi_\theta\left(a_t \mid s_t\right)$

as the probability of trajectory, for $\rho_0(s_0)$ being the probability of starting in state s_0. Clearly, $\log P(\tau \mid \theta) = \log \rho_0(s_0) + \sum_{t=0}^{T} (P(s_{t+1} \mid s_t, a_t) + \pi_\theta(a_t \mid s_t))$ and taking the gradient leaves us with $\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ since the gradient of environmental functions is 0. $\square$

An alternative proof that tackles the problem in a different way can be found in Sutton et al. [45, p. 1063]. Furthermore, a proof for the discounted reward case can be found in Zhao [25, pp. 210-212].

The simple gradient ascent method can now be utilised given we know how to calculate the gradient. The REINFORCE algorithm or Monte Carlo Policy Gradient algorithm utilises stochastic gradient ascent [44, p. 328]. The method is outlined in Algorithm 9 (Section 8, Appendix). One major benefit is that the algorithm balances the exploration–exploitation dilemma well. Letting $\beta_t = \frac{G_t}{\pi(a_t \mid s_t)}$, we see that β_t is proportional to G_t so is exploitative for greater reward values, but also β_t is inversely proportional to $\pi(a_t \mid s_t)$ when G_t is non–negative, meaning if the probability of conducting a_t is small, $\pi(a_t \mid s_t)$ is enhanced so the probability increases. This leads to an explorative nature for actions with low probabilities [25, p. 220]. More investigation of the exploration–exploitation dilemma is in Section 4.6. The time complexity of REINFORCE is $O(NT)$, where N are the number of episodes, and T is the length of each episode.

The major advantage of policy gradient methods are that they perform better in high–dimensional environments. Value based techniques such as SARSA and Q–Learning must iterate over all actions to determine $\arg\max_{a \in \mathcal{A}}$, which is harder when there are many states and actions. Additionally, in continuous environments it would be impossible to compute this, whereas it is possible to perform policy gradient methods [46]. On the other hand, policy gradient methods are much harder to explain and understand why the optimal policy is chosen compared to other methods. This is extremely undesirable in many cases and in some applications causes ethical problems, as outlined in the 'black–box problem' [47][48].

4.6 Exploration vs. Exploitation

The exploration–exploitation dilemma in reinforcement learning refers to the challenge of the balance between trying out new actions to discover potentially better strategies (exploration) and exploiting known strategies to maximize immediate rewards (exploitation). We will view the problem through the lens of a n–armed bandit problem.

Consider a situation in which you are repeatedly faced with a choice of n possible actions. After choosing an action, you receive a reward, determined by a probability distribution based on the selected action. The goal is to maximise the expected total reward [8, p. 32]. If we knew the reward for each action, the approach would be trivial and we would conduct the action that yields the highest possible reward. If we select the greediest possible action at each time step, we are exploiting. Otherwise, we are exploring. The challenge is that although exploiting may give us the largest expected reward in a time step, exploring may produce the greater total reward in the long run.

Definition 19 (Multi–Armed Bandit)**.** *[39, §9 p. 6] A multi–armed bandit is a tuple, $(\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ is the set of m actions and $\mathcal{R}^a(r)) = \mathbb{P}[r \mid a]$ is the unknown distribution of rewards. At each time step, t, the agent selects an action $a_t \in \mathcal{A}$ and recieves a reward, $r_t \sim \mathcal{R}^{a_t}$. The goal is to maximise $\sum_{k=1}^{t} r_k$.*

If we denote the true value of an action as $q(a)$, and the estimated value on the tth time step as $Q_t(a)$, we can estimate the rewards by averaging. If we let $N_t(a)$ be the total number of times action a is chosen, yielding rewards $R_1, \ldots, R_{N_t(a)}$, then define $Q_t(a) = \frac{R_1+\cdots+R_{N_t(a)}}{N_t(a)}$ [8, p. 33]. By the law of large numbers, it is clear that $Q_t(a) \to q(a)$. The greedy action therefore would be $a^* = \arg\max_{a \in \mathcal{A}} Q_t(a)$. There is a chance that this value will lock on to a sub–optimal action.

To determine how much should we exploit and how much should we explore, we define the notion of regret.

Definition 20 (Regret)**.** *[46] Given a policy, π and t number of arm pulls, regret is defined as:*

$$I_t = \mathbb{E}[\arg\max_{a \in \mathcal{A}} q(a) - Q_t(a)]$$

Similarly total regret is defined as:

$$L_t = \mathbb{E}[\sum_{\tau=1}^{t} \arg\max_{a \in \mathcal{A}} q(a) - Q_\tau(a)]$$

The goal of maximising reward is equivalent to minimising total regret. If an algorithm forever explores it will have linear total regret, and if an algorithm never explores it will also have linear total regret. Our aim is to achieve sublinear total regret.

The greedy algorithm proposed earlier, $a^* = \arg\max_{a \in \mathcal{A}} Q_t(a)$ may result in never finding the optimal policy. An alternative that solves this problem is the ϵ greedy algorithm, as shown in Algorithm 5 (Section 8, Appendix). There is a constant probability ϵ that a random action is taken and $1 - \epsilon$ probability that we take the current best strategy. As ϵ is constant throughout the policy, we still achieve linear regret. If T is the total number of time steps, the total regret is $\mathrm{O}(\epsilon \cdot T)$.

One other algorithm known as the decaying ϵ greedy algorithm solves this problem. The method shown in Algorithm 8 (Section 8, Appendix) shows that we pick an initial ϵ_0, and all subsequent $\epsilon_t = \frac{\epsilon_0}{t}$. This means there is a lot of exploring initially, but this decreases over time steps. The total regret over T time steps is $L_T \approx \epsilon_0 \sum_{t=1}^{T} \frac{1}{t}$. As the harmonic series grows logarithmically, so the regret for this method grows sublinearly. For this reason, epsilon decay is a popular method used in reinforcement learning agents.

In practice, we would not calculate $Q_t(a)$ the way we have described as it takes storage space in the computer memory. Instead, we use an incremental formula

$$Q_t(a) \leftarrow Q_{t-1}(a) + \frac{1}{N_t(a)}[R_{t-1} - Q_{t-1}(a)].$$

Using k instead of $N_t(a)$ for simplicity, the update formula is derived from:

$$\begin{aligned} Q_{k+1}(a) = \frac{1}{k}\sum_{i=1}^{k} R_i = \frac{1}{k}\left(R_k + \sum_{i=1}^{k-1} R_i\right) \\ &= \frac{1}{k}\left[R_k + (k-1)Q_k(a) + Q_k(a) - Q_k(a)\right] \\ &= Q_k(a) + \frac{1}{k}[R_k - Q_k(a)] \end{aligned}$$

If we let $\frac{1}{k} = \alpha$ be the step size, where $\alpha \in (0,1)$, we can neatly write this update formula as $Q_{k+1}(a) = Q_k(a) + \alpha[R_k - Q_k(a)] = \alpha R_k + (1-\alpha)Q_k(a)$. We can apply this recursively to obtain $Q_{k+1}(a) = \sum_{i=1}^{k}\left(\alpha(1-\alpha)^{k-i}R_i\right) + (1-\alpha)^k Q_1$ [8, p. 38]. As $(1-\alpha)^k + \sum_{i=1}^{k}\alpha(1-\alpha)^{k-i} = 1$, this is a weighted summation. As $1-\alpha$ is less than 1, the weight of R_i decreases exponentially according to $(1-\alpha)$, giving the most importance to the final reward, R_k. We also notice that the iterative algorithm is biased by the initial estimate $Q_1(a)$. As time continues this may not be a problem, and occasionally is helpful, but means we have another parameter [8, p. 40].

The final exploration–exploitation dilemma solution we will investigate is Upper Confidence Bounds (UCB1). The strategy involves taking the next action A_t according to $\arg\max_a \left[Q_t(a) + c\sqrt{\frac{2\ln(t)}{N_t(a)}}\right]$ [8, p. 55]. Intuitively, the $Q_t(a)$ term encourages exploitation, and the $c\sqrt{\frac{2\ln(t)}{N_t(a)}}$ term encourages exploration as this term is high for actions that have been explored less. With this method, all actions will eventually be performed as the natural logarithm is unbounded [46]. A novel way to show this approach is as follows.

The goal is to find an upper bound, $U_t(a)$ such that $q(a) \leq Q_t(a) + U_t(a)$. We consider the probability of the complement case: $\mathbb{P}[q(a) \leq Q_t(a) + U_t(a)] \leq e^{-2N_t(a)U_t^2(a)}$, utilising Hoeffding's Inequality. Picking a probability, p, we rearrange for $U_t(a)$ to achieve $U_t(a) = c\sqrt{\frac{-\ln(p)}{N_t(a)}}$. Finally, we choose a dynamic $p = t^{-4}$ such that we achieve the optimal policy as $t \to \infty$.

There are many other approaches that were not discussed such as gradient bandits and softmax. These approaches and others can be found in Bubeck et al. [49].

5 Varieties of Reinforcement Learning

There are numerous ways in which reinforcement learning is implemented. We will look at four different types of reinforcement learning and see why and when they are useful.

5.1 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) utilises neural networks from Section 3.2. The most popular methods for optimising the policy are policy gradient methods (Section 4.5) and a variant of Q-Learning (Section 4.4) known as Deep Q–Learning (DQL) [50]. We will focus on the latter.

DQL as outlined in Algorithm 10 (Section 8, Appendix) is Q–Learning in a neural network environment. Q–Learning uses a Q-Table to calculate the maximum expected future reward for each action at each state. For higher dimensional problems, this may take too long and so DQL uses a neural network that takes a state and approximates the Q-Values for each action based on that state. From there, we choose the action that has the largest Q–Value. Instead of updating the Q–Table at each step, as in Algorithm 7 (Section 8, Appendix), we update the weights of the neural network. This is a much more scaleable approach. The loss function,

$$\left[r + \gamma \max_{a'} Q(s', a'; \theta^{-}) - Q(s, a; \theta) \right]$$

is the square of the TD error in Q–Learning, δ_k (Section 4.4) [50]. Additionally, DQN utilizes an experience replay buffer to store past experiences observed by the agent. During training, batches of transitions are sampled randomly from the replay buffer to decorrelate the training data and improve learning stability [51, p. 24].

The method, created by DeepMind in 2015 compared previous algorithms with DQL on 49 Atari 2600 video games. Astonishingly, it was found that DQL surpassed all other algorithms and was comparable to professional game testers. This monumental feat has caused DQL to be one of the most used algorithms, with applications in gaming, robotics, natural language processing, finance, healthcare and more [51].

There are variations of the algorithm that utilise Convolutional Neural Networks (CNNs), Long–Short–Term–Memory (LSTM) and Recurrent Neural Networks (RNN). Using more complex architecture leads to more hyperparameters and choices in activation functions. The hyperparameters must be tuned and selected carefully. Currently, most hyperparameters are tuned using a grid search or a random search. Optimising the choice and developing new strategies to find the best hyperparameters are increasingly gaining attention within the research community [52].

5.2 Multi–Agent Reinforcement Learning

Multi–Agent Reinforcement Learning (MARL) deals with scenarios where multiple agents interact within the same environment. Each agent aims to learn to make decisions that optimize its own objectives while considering the actions and policies of other agents. In MARL, agents can be cooperative, competitive, or a combination of both, and they may have partial or full observability of the environment and other agents' actions [53, p, 8].

Some benefits of MARL include that the experience sharing can lead to faster outcomes and higher performance. If one or more policies fail, the remaining agents can take over some of their tasks. Additionally, the agents can work in parallel, leading to a decentralised structure of a task. The benefits come at a cost of dimensionality as adding more agents impacts computation exponentially. Specifying the goal can also cause issues since the agents' goals are correlated and cannot be independently maximised [54, pp. 7-8] [53, pp. 12-16].

MARL can be used in a wide array of applications. For cooperative settings, they can be used in autonomous vehicle driving in which the cars must collaborate in order to avoid collisions while trying to maximize traffic flow and possibly fuel efficiency [55] [56]. Competitive settings may include board games and video games such as using the technique to reach Grandmaster level in the video game Starcraft II [57]. A combination of both can be used in team games, such as for a coach–player dynamic [58].

5.3 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) is a form of reinforcement learning that utilises Semi Markov Decision Processes (SMDP). One of the main concepts is temporal abstraction. Tasks are decomposed into a hierarchy of subtasks or behaviors. Each level of the hierarchy represents a different level of abstraction, with higher-level subtasks encapsulating lower-level ones. Temporal abstraction therefore refers to the ability of agents to reason and make decisions and explore each sub task [59]. A way to conceptualise this is, for example, an agent would first decide to fill a kettle with water before boiling the water. These two problems can be modelled with different policies and broken into subtasks. This allows the agent to decompose complex tasks into simpler subtasks, making it easier to learn and execute efficient policies.

Some potential uses of HRL include disease diagnosis and robotics [60]. Disease diagnosis can be modelled as a sequential decision problem. At each time step, the agent chooses a symptom to inquire about. After a binary response, the agent presents a new inquiry based on prior knowledge. At the end of the diagnosis process, the agent predicts a disease that the patient may have, with the rewards being accuracy and minimal inquiry length [61].

5.4 Reinforcement Learning from Human Feedback

Reinforcement Learning from Human Feedback (RLHF) is a sub–field of reinforcement learning where the learning process is guided or enhanced by human feedback. Instead of relying solely on environmental rewards, the agent receives feedback from human trainers or users to accelerate the learning process or improve the agent's behavior. By leveraging human feedback, the learning agent can adapt more quickly to new tasks or environments, avoid undesirable behaviors, and achieve better performance compared to learning from environmental rewards alone. This type of reinforcement learning is the most popular currently as it forms the foundation of ChatGPT [62]. In this particular use–case, the policy is updated using Proximal Policy Optimisation (PPO), which is a type of policy gradient method. Further information regarding how PPO works can be found in Schulman et al. [63]. This type of reinforcement learning is often used in tandem with techniques in Natural Language Processing (NLP). Further information can be found in Stiennon et al. [64].

6 Applications of Reinforcement Learning

Reinforcement Learning is used in many contexts in the current day. We will explore a few of these, however there are numerous other uses, some of which were mentioned in Section 5. Some uses not discussed include uses in nuclear fusion [65], agriculture [66] and commercial cooling systems [67].

6.1 Video Games

As mentioned in Section 2, the development of reinforcement learning was accelerated through researchers creating agents for traditional board games that were transferred into a digital environment. For this reason, and due to its accessibility, it is to no surprise that the most well known area of reinforcement learning is the attempt to complete video games.

The OpenAI team have launched a tool called OpenAI Gym, comprising many tasks and environments for agents to learn in [68]. The aim is to provide numerous benchmark tests that an agent can use to compare against other techniques. Environments range from very simple games to complex, physics-based gaming engines. Some future additions that are planned for Gym include having one agent complete increasingly difficult games using transfer learning techniques, aiming to increase reinforcement learning and transfer learning development, and integration with real world hardware such as robotics.

The most famous benchmark, also included in OpenAI Gym are games from the Atari 2600 video game console due to the broad number of games, as alluded to in Section 5.1. Simulated environments such as video games are often used due to their accessibility, variability and lack of real–world implications. In particular, the variability of games enables many different types of reinforcement learning to be tested. Agents using MARL have been used in strategy games, beating 99.8% of human players in the game Starcraft II [57], DRL has been used in platformer games and multi–player online games [69] [70].

Game developers recently have also included agents in games in the form of Non Playable Characters (NPCs) and dynamic game balancing. HRL techniques have been used to model NPCs in simple ‘Capture the Flag’ games [71, p. 138], resulting in more complex characters that performed 52% better. Dyanmic game balancing has been used to fit the personal style of the player to provide a customised, challenging and smooth experience [70, p. 208].

Recent research has investigated robotic agents learning in a simulated video game environment before transferring the knowledge to a real life application using transfer learning techniques (Section 3.4). In a simulated environment, there are no real world impacts and it is often cheaper than running expensive and slow robot experiments [72].

We will now see a worked example of creating a reinforcement learning agent to play a video game.

6.1.1 Worked Example: Snake Game

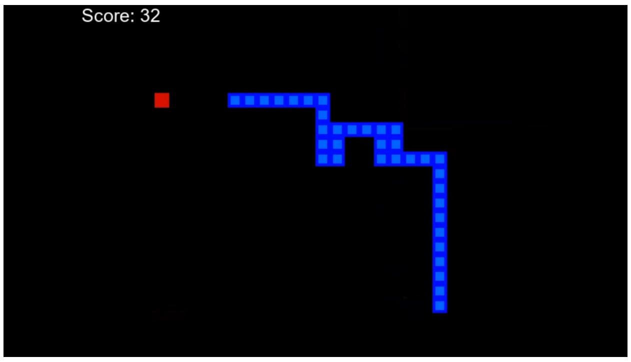

Figure 7: Reinforcement Learning agent attempting the Snake Game (140th Game of learning)

We will consider the example of the Snake Game. This is a classic computer game in which a snake moves in the four cardinal directions to ‘eat’ the fruit. If it does so, the length of the snake grows by 1 block and a new fruit appears in a random location. If the snake collides into itself or the walls of the game, it results in a ‘game over’. The game is shown in Figure 7.

Clearly when programming the agent, our rewards must be represented in the game structure. The agent is given a reward of -10 if it has a game over and $+10$ for every fruit eaten. Additionally, it is coded so that if the snake does not eat any fruit for a certain amount of time (coded as a function of its length), it gets a game over to incentivise the agent to eat the fruit.

There are 11 states and 3 actions coded. There are 4 states determining what direction the snake is moving, 3 to find immediate danger to the left, right or in front of the snake, and a final 4 to determine the relative direction of the fruit. The actions are to move left, right or straight. Actions must be calculated every frame of the game, which requires many fast computations due to the dimensionality of the state–action space. For this reason, we use DQL from Section 5.1 with the same loss function described earlier, and the Q–Learning iterative formula is hard coded into the agent. The learning rate α is 0.001 which is a standard value, and discount rate γ is 0.85, chosen through experimentation. The weights and biases are updated using backpropagation (Section 3.2) every frame and we receive 3 output actions from the neural network. We choose the action with the highest Q–Value at the end of each frame.

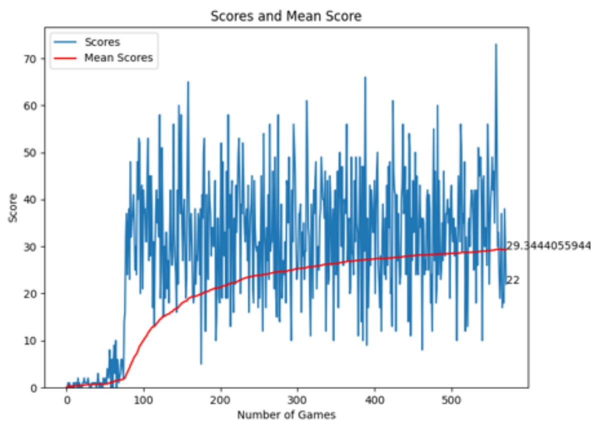

Figure 8: Scores from the agent in each game in blue. Rolling average is in red. This highlights the exploration and exploitation phases of the game.

Figure 8 shows the exploration and exploitation phases of the agent well. After roughly 80 games, the snake enters its exploitation phase and achieves much more impressive scores. The agent uses a more naive version of the ϵ–decay method shown in Algorithm 8 (Section 8, Appendix), however after 80 games, the randomness completely stops. The number 80 was chosen through trial and error, measuring performance.

The code for the agent can be found at :
`https://github.com/AadamHaq/Reinforcement-Learning-Dissertation/tree/main/Code`

6.2 Healthcare

We have already seen in Section 5.3 the use of HRL in disease diagnosis [61]. Recently, researchers have used a variety of reinforcement learning called ReLeaSE (Reinforcement Learning for Structural Evolution) which integrates both generative and predictive neural network models to help discover new drugs. The two networks are initially trained separately using supervised learning techniques, before jointly being used with ReLeaSE to create the generation of new chemical structures with desired physical and biological properties. The goal of the generative neural network is to maximise reward, and the predictive neural network acts as a critic estimating the agent's behavior by assigning a numerical reward to each generated molecule. The model utilises policy gradient methods, in particular the REINFORCE Algorithm (Algorithm 9, Section 8, Appendix) [73].

Another use of reinforcement learning is in dynamic treatment regimes, particularly for patients with evolving treatments and those with chronic disorders. The main challenge with a reinforcement learning approach is that treatment regimes are inherintly non-Markovian. The paper overcomes this issue by creating novel reinforcement learning strategies that efficiently leverage observational data, creating the UC-DTR algorithm. The approach incorporates observational samples so transtion probabilities and expectations can be calculated, enabling techniques such as Q–Learning [74].

7 Conclusion and Extensions

Reinforcement learning has developed rapidly since the 1950's and continues to accelerate in research. It is gradually becoming one of the most active research areas in machine learning, with increasing implementation in ambitious applications [75]. With the rise of applications such as ChatGPT, and its versatile utility in problems such as autonomous driving, air traffic control and healthcare, soon it will infiltrate our everyday lives in many different ways.

As reinforcement learning is implemented further, ethical questions arise. These include the lack of human interaction, the safety and reliability of agents, incentives for data collection, potential harms of reward optimisation and the future of work with more automation. As these questions are raised, these few fundamental issues must be rectified, specifically regarding its interpretability, accountability, and trustworthiness [76, pp. 1009-1015]. Inevitably however, there will be many benefits to further implementation and advancements in reinforcement learning, some of which have been highlighted in this report.

This report gives an insight into the mathematics of reinforcement learning, however there are many more topics and more depth that can be explored. In particular, this report focuses for the most part (with the exception of policy gradients and DQL) on discrete environments, however continuous environments and frameworks exist. Reinforcement learning can be viewed from the angle of continuous control theory as in Recht [77]. An excellent introductory video and playlist series can be found in Brunton [78]. Other branches of mathematics include stochastic differential equations as in Wang et al. [79] and game theory, in investigating the exploration–exploitation dilemma further or in situations such as MARL in Nowé et al. [80]. Additionally, reinforcement learning naturally heavily depends on complex tools and developments in computer science and statistics. Some of these perspectives can be found in Kaelbling et al. [81] and Dong et al. [82].

Bibliography

References

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, second edition, in progress edition, 2014.

[2] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[3] Edward L. Thorndike. *Animal intelligence; experimental studies.* New York, The Macmillan Company, 1911.

[4] P Ivan Pavlov. Conditioned reflexes: an investigation of the physiological activity of the cerebral cortex. *Annals of neurosciences*, 17(3):136, 2010.

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

[6] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[7] A Harry Klopf. *Brain function and adaptive systems: a heterostatic theory.* Number 133 in Special Reports. Air Force Cambridge Research Laboratories, Air Force Systems Command, United States Air Force, 1972.

[8] RS Sutton. Learning theory support for a single channel theory of the brain. Unpublished, 1978.

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

[10] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[11] Steven Levy and B Powell. Man vs. machine. *Newsweek*, 129(18):50–56, 1997.

[12] Christopher Moyer. How google's alphago beat a go world champion. *The Atlantic*, 28, 2016.

[13] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In *Machine learning techniques for multimedia: case studies on organization and retrieval.* Springer, 2008.

[14] Amirhosein Toosi, Andrea G Bottino, Babak Saboury, Eliot Siegel, and Arman Rahmim. A brief history of ai: how to prevent another winter (a critical review). *PET clinics*, 16(4):449–469, 2021.

[15] Barry J Wythoff. Backpropagation neural networks: a tutorial. *Chemometrics and Intelligent Laboratory Systems*, 18(2):115–155, 1993.

[16] Raul Rojas and Raúl Rojas. The backpropagation algorithm. *Neural networks: a systematic introduction*, pages 149–182, 1996.

[17] Catherine F Higham and Desmond J Higham. Deep learning: An introduction for applied mathematicians. *Siam review*, 61(4):860–891, 2019.

[18] H. Leung and S. Haykin. The complex backpropagation algorithm. *IEEE Transactions on Signal Processing*, 39(9):2101–2104, 1991. doi: 10.1109/78.134446.

[19] Vladimir N Vapnik. An overview of statistical learning theory. *IEEE transactions on neural networks*, 10(5):988–999, 1999.

[20] Shashi Sathyanarayana. A gentle introduction to backpropagation. *Numeric Insight*, 7:1–15, 2014.

[21] Fetty Fitriyanti Lubis, Yusep Rosmansyah, and Suhono Harso Supangkat. Gradient descent and normal equations on cost function minimization for online predictive using linear regression with multiple variables. In *2014 International Conference on ICT For Smart Society (ICISS)*, pages 202–205, 2014. doi: 10.1109/ICTSS.2014.7013173.

[22] Aatila Mustapha, Lachgar Mohamed, and Kartit Ali. An overview of gradient descent algorithm optimization in machine learning: Application in the ophthalmology field. In *Smart Applications and Data Analysis: Third International Conference, SADASC 2020, Marrakesh, Morocco, June 25–26, 2020, Proceedings 3*, pages 349–359. Springer, 2020.

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

[24] P. Baldi. Gradient descent learning algorithm overview: a general dynamical systems perspective. *IEEE Transactions on Neural Networks*, 6(1):182–195, 1995. doi: 10.1109/72.363438.

[25] S. Zhao. *Mathematical Foundations of Reinforcement Learning*. Springer Press, 2024.

[26] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.

[27] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[28] Thommen George Karimpanal and Roland Bouffanais. Self-organizing maps for storage and transfer of knowledge in reinforcement learning. *Adaptive Behavior*, 27(2):

111–126, 2019. doi: 10.1177/1059712318818568. URL https://doi.org/10.1177/1059712318818568.

[29] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010. doi: 10.1109/TKDE.2009.191.

[30] Zhuangdi Zhu, Kaixiang Lin, Anil K. Jain, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(11):13344–13362, 2023. doi: 10.1109/TPAMI.2023.3292075.

[31] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[32] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3:1–40, 2016.

[33] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

[34] Oliver Ibe. *Markov processes for stochastic modeling.* Newnes, 2013.

[35] Dan Marinescu. *Big Data, data streaming, and the mobile cloud*, pages 453–500. Morgan Kaufmann, 01 2023. ISBN 9780323852777. doi: 10.1016/B978-0-32-385277-7.00019-1.

[36] Davis Foote and Brian Chu. Polynomial time approximate solutions to markov decision processes with many objectives.

[37] David B Wagner. Dynamic programming. *The Mathematica Journal*, 5(4):42–51, 1995.

[38] Nancy L Stokey. *Recursive methods in economic dynamics.* Harvard University Press, 1989.

[39] David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.

[40] M. Wiering and M. van Otterlo. *Reinforcement Learning: State-of-the-Art.* Adaptation, Learning, and Optimization. Springer Berlin Heidelberg, 2012. ISBN 9783642276446. URL https://books.google.co.uk/books?id=T4wovQEACAAJ.

[41] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951. doi: 10.1214/aoms/1177729586. URL https://doi.org/10.1214/aoms/1177729586.

[42] Koundinya Vajjha, Barry Trager, Avraham Shinnar, and Vasily Pestun. Formalization of a stochastic approximation theorem. *arXiv preprint arXiv:2202.05959*, 2022.

[43] Josh Achiam and OpenAI Team. Part 3: Intro to Policy Optimization Spinning Up documentation — spinningup.openai.com. `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html`, 2018. [Accessed 27-03-2024].

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

[45] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[46] Tim Miller. Introduction to Reinforcement Learning — gibberblot.github.io. `https://gibberblot.github.io/rl-notes/intro.html`, 2022. [Accessed 27-03-2024].

[47] Warren J Von Eschenbach. Transparency and the black box problem: Why we do not trust ai. *Philosophy & Technology*, 34(4):1607–1622, 2021.

[48] Davide Castelvecchi. Can we open the black box of ai? *Nature News*, 538(7623):20, 2016.

[49] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

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

[51] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[52] Theresa Eimer, Marius Lindauer, and Roberta Raileanu. Hyperparameters in reinforcement learning and how to tune them. In *International conference on machine learning*, pages 9104–9149. PMLR, 2023.

[53] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences*, 11(11):4948, 2021.

[54] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221, 2010.

[55] Marco A Wiering et al. Multi-agent reinforcement learning for traffic light control. In *Machine Learning: Proceedings of the Seventeenth International Conference (ICML'2000)*, pages 1151–1158, 2000.

[56] Marc Brittain and Peng Wei. Autonomous air traffic controller: A deep multi-agent reinforcement learning approach. *arXiv preprint arXiv:1905.01303*, 2019.

[57] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[58] Bo Liu, Qiang Liu, Peter Stone, Animesh Garg, Yuke Zhu, and Anima Anandkumar. Coach-player multi-agent reinforcement learning for dynamic team composition. In *International Conference on Machine Learning*, pages 6860–6870. PMLR, 2021.

[59] Matthias Hutsebaut-Buysse, Kevin Mets, and Steven Latré. Hierarchical reinforcement learning: A survey and open research challenges. *Machine Learning and Knowledge Extraction*, 4(1):172–221, 2022. ISSN 2504-4990. doi: 10.3390/make4010009. URL `https://www.mdpi.com/2504-4990/4/1/9`.

[60] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 54(5):1–35, 2021.

[61] Hao-Cheng Kao, Kai-Fu Tang, and Edward Chang. Context-aware symptom checking for disease diagnosis using hierarchical reinforcement learning. *Proceedings of the AAAI conference on artificial intelligence*, 32(1), 04 2018.

[62] Open AI. Blog: Introducing chatgpt. `https://openai.com/blog/chatgpt`, 2022.

[63] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[64] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[65] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

[66] Romain Gautron, Emilio J. Padrón, Philippe Preux, Julien Bigot, Odalric-Ambrym Maillard, and David Emukpere. gym-dssat: a crop model turned into a reinforcement learning environment, 2022.

[67] Jerry Luo, Cosmin Paduraru, Octavian Voicu, Yuri Chervonyi, Scott Munns, Jerry Li, Crystal Qian, Praneet Dutta, Jared Quincy Davis, Ningjia Wu, et al. Controlling commercial cooling systems using reinforcement learning. *arXiv preprint arXiv:2211.07357*, 2022.

[68] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[69] Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, et al. Mastering complex control in moba games with deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):6672–6679, 04 2020.

[70] Xuyouyang Fan. The application of reinforcement learning in video games. In *2023 International Conference on Image, Algorithms and Artificial Intelligence (ICIAAI 2023)*, pages 202–211. Atlantis Press, 2023.

[71] Mostafa Al-Emran. Hierarchical reinforcement learning: a survey. *International journal of computing and digital systems*, 4(02), 2015.

[72] Janne Karttunen, Anssi Kanervisto, Ville Kyrki, and Ville Hautamäki. From video game to real robot: The transfer between action spaces. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3567–3571. IEEE, 2020.

[73] Mariya Popova, Olexandr Isayev, and Alexander Tropsha. Deep reinforcement learning for de novo drug design. *Science advances*, 4(7):eaap7885, 2018.

[74] Junzhe Zhang and Elias Bareinboim. Near-optimal reinforcement learning in dynamic treatment regimes. *Advances in Neural Information Processing Systems*, 32, 2019.

[75] Tie-Yan Liu, Tao Qin, Bin Shao, Wei Chen, and Jiang Bian. Machine learning: Research hotspots in the next ten years. `https://www.microsoft.com/en-us/research/lab/microsoft-research-asia/articles/machine-learning-research-hotspots/`, Dec 2023.

[76] Jess Whittlestone, Kai Arulkumaran, and Matthew Crosby. The societal implications of deep reinforcement learning. *Journal of Artificial Intelligence Research*, 70:1003–1030, 2021.

[77] Benjamin Recht. A tour of reinforcement learning: The view from continuous control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2:253–279, 2019.

[78] Steve Brunton. Reinforcement learning: Machine learning meets control theory. `www.youtube.com,youtu.be/0MNVhXEX9to?si=-E7wMyJl-MkhNm78`, Feb. 2021.

[79] Haoran Wang, Thaleia Zariphopoulou, and Xun Yu Zhou. Reinforcement learning in continuous time and space: A stochastic control approach. *Journal of Machine Learning Research*, 21(198):1–34, 2020.

[80] Ann Nowé, Peter Vrancx, and Yann-Michaël De Hauwere. Game theory and multi-agent reinforcement learning. *Reinforcement Learning: State-of-the-Art*, pages 441–470, 2012.

[81] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[82] Hao Dong, Hao Dong, Zihan Ding, Shanghang Zhang, and Chang. *Deep Reinforcement Learning.* Springer, 2020.

8 Appendix

8.1 Algorithms

Algorithm 1 Gradient Descent

```
Require: Training dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, Learning rate $\alpha$, Threshold $\epsilon$
Ensure: Optimal parameters $\theta$
  Initialize parameters $\theta$ randomly or using predefined strategies
  Initialize previous cost $J_{\text{prev}} = \infty$
  Set current cost $J_{\text{curr}} = 0$
  while $|J_{\text{prev}} - J_{\text{curr}}| > \epsilon$ do
    Set $J_{\text{prev}} = J_{\text{curr}}$
    Set $J_{\text{curr}} = 0$
    for $i = 1$ to $m$ do
      Perform forward pass to compute predicted output $\hat{y}^{(i)}$
      Compute loss function $L(y^{(i)}, \hat{y}^{(i)})$
      Compute cost function $J(\theta)$ and add it to $J_{\text{curr}}$
      Perform backpropagation to compute gradients
    end for
    for each parameter $\theta_j$ do
      Update parameter using gradient descent:
      $\theta_j = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$
    end for
  end while
```

Algorithm 2 Stochastic Gradient Descent

```
Require: Training dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, Learning rate $\alpha$, Threshold $\epsilon$
Ensure: Optimal parameters $\theta$
  Initialize parameters $\theta$ randomly or using predefined strategies
  Initialize previous cost $J_{\text{prev}} = \infty$
  Set current cost $J_{\text{curr}} = 0$
  while $|J_{\text{prev}} - J_{\text{curr}}| > \epsilon$ do
    Set $J_{\text{prev}} = J_{\text{curr}}$
    Set $J_{\text{curr}} = 0$
    Randomly shuffle the training dataset
    for $i = 1$ to $m$ do
      Select a single training example $(x^{(i)}, y^{(i)})$ randomly
      Perform forward pass to compute predicted output $\hat{y}^{(i)}$
      Compute loss function $L(y^{(i)}, \hat{y}^{(i)})$
      Compute cost function $J(\theta)$ and add it to $J_{\text{curr}}$
      Perform backpropagation to compute gradients using this single example
      for each parameter $\theta_j$ do
        Update parameter using stochastic gradient descent:
        $\theta_j = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$
      end for
    end for
  end while
```

Algorithm 3 Value Iteration Algorithm (Adapted from [8, p. 101])

```
  Input: Environment with states $\mathcal{S}$, actions $\mathcal{A}$, transition probabilities $\mathcal{P}^a_{ss'}$, rewards $\mathcal{R}^a_{ss'}$, discount factor $\gamma$, convergence threshold $\epsilon$.
  Output: Optimal value function $v_*(s)$ and policy $\pi_*(s)$.
  Initialize value function $v(s)$ arbitrarily for all $s \in \mathcal{S}$.
  repeat
    for each state $s \in \mathcal{S}$ do
      Update the value function using the Bellman optimality equation:
      $v(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma v(s')]$
    end for
    Update the policy based on the updated value function:
    for each state $s \in S$ do
      $\pi_*(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma v(s')]$
    end for
    Calculate the maximum change in value function: $\Delta \leftarrow \max_{s \in \mathcal{S}} |v_{\text{new}}(s) - v_{\text{old}}(s)|$
  until $\Delta < \epsilon$
```

Algorithm 4 Policy Iteration (Adapted from [40, p. 22])

```
**Input:** Environment with states $\mathcal{S}$, actions $\mathcal{A}$, transition probabilities $\mathcal{P}^a_{ss'}$, rewards $\mathcal{R}^a_{ss'}$, discount factor $\gamma$, convergence threshold $\epsilon$ and $\pi_*$
**Output:** Optimal policy $\pi_*$
Initialize policy $\pi$ arbitrarily
repeat
  Policy Evaluation:
  Initialize $v(s) = 0$ for all $s \in \mathcal{S}$
  repeat
    for each state $s \in \mathcal{S}$ do
      Update the value function using the policy evaluation equation:
      $v(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma v(s')]$
    end for
    Calculate the maximum change in value function: $\Delta \leftarrow \max_{s \in \mathcal{S}} |v_{\text{new}}(s) - v_{\text{old}}(s)|$
  until $\Delta < \epsilon$
  Policy Improvement:
  Policy stable $\leftarrow$ true
  for each state $s \in \mathcal{S}$ do
    old action $\leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma v(s')]$
    if old action $\neq \pi(s)$ then
      Policy stable $\leftarrow$ false
    end if
  end for
until Policy stable
```

Algorithm 5 Epsilon-Greedy Algorithm

```
Initialize action-value estimates $Q(s, a)$ and counter $N(s, a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
for each episode do
  for each step of the episode do
    With probability $\epsilon$, choose a random action $a$
    Otherwise, choose $a$ such that $a = \arg\max_{a'} Q(s, a')$
    Take action $a$, observe $r, s'$
    $N(s, a) \leftarrow N(s, a) + 1$
    $Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s,a)}[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  end for
end for
```

Algorithm 6 SARSA Algorithm (Adapted from [8, p. 155])

```
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialize $s$
repeat
  Choose action $a$ for $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
  Take action $a$, observe $r, s'$
  Choose action $a'$ for $s'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
  $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
  $s \leftarrow s'$, $a \leftarrow a'$
until episode terminates
```

Algorithm 7 Q-Learning Algorithm (Adapted from [40, p. 31])

```
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for each episode do
  Initialize $s$
  repeat
    Choose action $a$ for $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $a$, observe $r, s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  until episode terminates
end for
```

Algorithm 8 Decaying Epsilon-Greedy Algorithm

```
Input: Initial exploration rate $\epsilon_0$, number of time steps $T$
$t \leftarrow 0$
for $t = 1$ to $T$ do
  Calculate exploration rate $\epsilon_t = \frac{\epsilon_0}{t}$
  Choose action $a_t$ with probability $\epsilon_t$ (exploration) or $1 - \epsilon_t$ (exploitation)
  Execute action $a_t$, observe reward $r_t$
  Update action-value estimates
end for
```

Algorithm 9 REINFORCE Algorithm (Adapted form [44, p. 328])

```
Initialize policy parameters $\theta$
for each episode do
  Initialize list of gradients: grads = []
  Generate an episode $\{s_0, a_0, r_1, s_1, a_1, r_2, ..., s_T\}$ using policy $\pi(\theta)$
  for $t = 0$ to $T$ do
    Compute the return $G_t$
    Compute the policy gradient: $\nabla_\theta \log \pi(a_t|s_t) \cdot G_t$
    Append gradient to list: grads.append($\nabla_\theta \log \pi(a_t|s_t) \cdot G_t$)
  end for
  Update the policy parameters using stochastic gradient ascent: $\theta \leftarrow \theta + \alpha \cdot \text{mean(grads)}$
end for
```

Algorithm 10 Deep Q-Network (DQN) (Adapted from [51, p. 15])

```
 1: Initialize replay memory $D$ with capacity $N$
 2: Initialize action-value function $Q$ with random weights $\theta$
 3: Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
 4: **for** episode $= 1$ to $M$ **do**
 5:     Initialize state $s_1$
 6:     **for** $t = 1$ to $T$ **do**
 7:         With probability $\epsilon$ select a random action $a_t$
 8:         Otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
 9:         Execute action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
10:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
11:         Sample random minibatch of transitions $(s_i, a_i, r_i, s_{i+1})$ from $D$
12:         Set target value $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \theta^-)$
13:         Update action-value function by minimizing the loss:
14:         $\mathcal{L}(\theta) = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i; \theta))^2$
15:         Perform gradient descent on $\mathcal{L}$ with respect to $\theta$
16:         Every $C$ steps, update target network: $\theta^- = \theta$
17:     **end for**
18: **end for**
```