Efficient Web Crawling with Reinforcement Learning

Daniel Friar

University College London
Department of Computer Science
Supervisor: David Barber

This report is submitted as part requirement for the MSc Degree in CSML at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

August 30, 2017

Abstract

This thesis applies value-based reinforcement learning algorithms to the problem of finding specific pages on the world wide web. Several Q-learing based approaches are used to find a small number of company pages within a large web graph, with word embeddings and convolutional neural networks applied to URL strings to represent the value function of a web page. We demonstrate that these approaches significantly outperform a random crawler using only the URL strings as input, with an asynchronous deep Q-network giving the best performance. Additionally, this algorithm is used to find a small number of company pages on the entire UK web beginning from a chosen set of pages and extensions are proposed to make practical use of the method.

The full code for the project is available in the repository here:
https://github.com/dannyfriar/msc-project.

Contents

1 Introduction **1**

1.1 Specific Problem and Objectives 1

1.2 Summary of the Thesis 2

2 Literature Review **3**

2.1 Reinforcement Learning Introduction and MDPs 3

2.1.1 Solving an MDP 5

2.2 Value-based model-free prediction and control 5

2.2.1 Monte Carlo Learning 6

2.2.2 Temporal Difference Learning - TD(0) 6

2.2.3 TD(λ) 6

2.2.4 Policies - exploration vs exploitation 7

2.2.5 Sarsa and Q-learning 7

2.2.6 Function approximation 8

2.3 Neural Networks and Optimization 10

2.3.1 Introduction 10

2.3.2 Optimization in Neural Networks 11

2.3.3 Parameter Initialization in Neural Networks 13

2.3.4 Convolutional Neural Networks (CNNs) 14

2.4 Deep Reinforcement Learning 16

2.4.1 Deep Q-learning 16

2.4.2 Experience replay buffer 17

2.4.3 Target Network 18

2.4.4 Double Q-learning 18

2.4.5 Asynchronous Methods for Deep Reinforcement Learning . . . 19

2.4.6 Debugging Deep RL 20

2.5 Alternatives to Value-based Methods 21

2.5.1 Tree-based Methods 21

2.5.2 Policy-based Methods 21
2.6 Application to Finding Pages of Interest 21
2.6.1 Focused Crawling with Reinforcement Learning 21
2.6.2 Other Relevant Work 23
2.7 CNNs for Text Classification 23
2.7.1 Word Embeddings 23
2.7.2 CNNs for Classifying Text 24

3 Methods **26**
3.1 Available Data 26
3.2 Framing as an RL Problem 26
3.2.1 Function Approximation 27
3.2.2 Starting Problem 27
3.2.3 Dealing with Repeat Rewards 29
3.2.4 Evaluating Against the Optimal Value Function 29
3.3 Motivating Q-learning 30
3.4 Initial Approach: Q-Learning with Linear Function Approximation . 30
3.4.1 Inclusion of Prioritized Experience Replay Buffer 31
3.4.2 Linear Q-Learning: Results and Analysis 32
3.5 Extensions of the Linear Q-learning Approach 34
3.5.1 Using Anchor Text and Page Titles 34
3.5.2 Penalizing Long Episodes with Negative Rewards 35
3.6 DQN with Word Embedding 35
3.7 Testing the Representation and Benchmarking 36
3.7.1 Benchmarking 38
3.8 Asynchronous DQN 38
3.9 Summary of Results 40
3.9.1 Running Time 41
3.10 Possible Extensions 42
3.10.1 A3C: Advantage Actor-Critic Methods 42
3.10.2 Using LSTMs to Improve State Representation 42

4 Application to the UK Web **43**
4.1 Expansion of Previous Problem 43
4.1.1 Results and Analysis 44
4.1.2 Results on Held-out Rewards 45
4.2 Practical Application 46

4.2.1 Starting Pages 46
4.2.2 Using the Common Crawl Data in Practice 46

5 Conclusion and Suggestions for Further Work 48
5.1 Summary 48
5.2 Critique and Suggestions for Further Work 49

Bibliography 50

A 56
A.1 Eligibility Traces for Online TD(λ) 56
A.2 Tree-based Methods 56
A.2.1 Forward Search 57
A.2.2 Simulation Based Search 57

B 58
B.1 Distribution of Number of Outgoing Links 58
B.2 Computation Graphs for Text Classification CNN 59

C 61
C.1 Distribution of Number of Outgoing Links in UK Web Graph 61

Chapter 1

Introduction

The number of indexed pages in the world wide web is estimated to exceed 4 billion [1]. This enormous volume of information makes finding pages relevant to a specific topic incredibly difficult. In order to obtain the data necessary to maintain its search engine, Google crawls an enormous number of web pages every day. Clearly, very large scale crawling is not feasible with limited infrastructure and obtaining very specific information from the web requires alternative methods.

In reinforcement learning (RL), an agent considers how to sequentially make decisions based on interactions with their environment in order to maximize a reward signal [2]. In recent years, reinforcement learning has achieved very impressive results in a number of domains, notably achieving super-human performance in the game of Go [3] in early 2016. In this thesis we attempt to use current reinforcement learning techniques to solve the practical problem of finding pages of interest in a large network of web pages.

This problem is well suited to the reinforcement learning framework. Consider browsing the web to find specific information and randomly clicking links starting from a given URL. After several links have been visited, we may begin to realize which pages are more relevant and start to adapt our behaviour accordingly. In this thesis we aim to do the same with an RL agent. We aim to find an efficient way to navigate a web graph in order to find specific pages, using RL methods to refine our crawling policy to achieve this.

1.1 Specific Problem and Objectives

We consider the problem of finding information on UK companies from the web. Given a list of companies within a specific industry, we're interested in finding pages

that contain information on these companies. Finding a company page or a company name within a web page would constitute a reward and these pages are considered relevant. Our aim is then to maximize the number of rewards found per pages crawled.

In order to concretely define this problem, we begin with labelled data on company web pages within a specific industry and construct a web graph from these pages. We then attempt to use RL methods to efficiently find these company pages within the graph, inspired by the work presented in the literature review.

1.2 Summary of the Thesis

In section 2, we begin by reviewing the reinforcement learning literature, starting with basic concepts before building up to more advanced and recent methods and then discussing previous work on similar problems.

In section 3, we use this theory to frame our problem as an RL problem and discuss the motivation behind using specific approaches. We build a web graph from the labelled companies data and apply these approaches to find company pages within the graph. An asynchronous DQN with a state representation that uses CNNs and word embeddings on URL strings is found to achieve the best performance, finding an average of 45% more rewards than a random crawler after crawling approximately 10% of pages in the web graph. A pre-trained classifier is used to form an upper benchmark on performance, with the DQN obtaining just 7% fewer rewards.

This problem is extended in chapter 4, with the asynchronous DQN applied to the larger problem of finding company pages within the UK web. Performance is again much better than random, with the DQN finding 40% more rewards after crawling 1% of pages. A further test is performed using a set of held-out company pages, with the asynchronous DQN obtaining almost twice as many rewards as a random crawler.

Chapter 2

Literature Review

Throughout the following chapter, we present a brief introduction to both reinforcement learning (RL) and neural networks before explaining how neural networks can be used in conjunction with RL algorithms in order to achieve better practical results. The theoretical motivation behind some practical deep RL "tricks" are explained and research papers relevant to the given problem are presented. In accordance with the method presented in chapter 3, we then discuss how convolutional neural networks can be used for text classification. Please see Sutton and Barto's *Reinforcement Learning: An Introduction* [2] for a more in-depth discussion on reinforcement learning basics.

2.1 Reinforcement Learning Introduction and MDPs

Reinforcement learning is a branch of machine learning which focuses on learning to make optimal decisions from experience. Generally, we wish to understand how an agent should act within an environment in order to maximize future reward. We can formalize this definition as a Markov Decision Process (MDP) by defining the following.

- A set of states S that follow the Markov propery, i.e. the transition probability depends only on the current state and not the previous history e.g. $P(S_{t+1}|S_1, ..., S_t) = P(S_{t+1}|S_t)$.
- A set of actions A.
- A probability transition matrix that describes the probability of transitioning between states after an action is taken: $P^a_{ss'} \equiv P(S_{t+1} = s'|S_t = s, A_t = a)$.

- A reward function $R_s^a \equiv \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$ that describes the expected reward after taking an action from a particular state.
- A discount factor $\gamma \in [0, 1]$ that is used to weight the value of immediate rewards against delayed rewards.

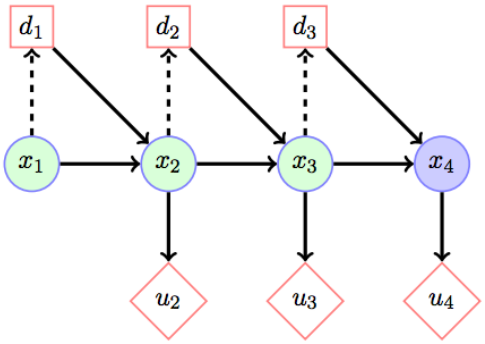

Figure 2.1: A diagram of a Markov Decision Process represented as an influence diagram. Here states, actions and rewards are defined by x, d and u respectively [4].

We define the return, G_t, as the discounted sum of rewards from a particular state. The MDP may have an infinite time-horizon i.e. there may not be a terminating state and the agent may continue taking steps indefinitely. For $\gamma < 1$, the discount factor ensures that the return remains finite as $T \to \infty$.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{t=0}^{T} \gamma^t R_{t+1} \tag{2.1}$$

The behaviour of an agent within the MDP is defined by its *policy* π, formally this is the distribution over actions in each state:

$$\pi(a|s) = P(A_t = a|S_t = s). \tag{2.2}$$

This allows us to define the value function, $v_\pi(s)$, which is the expected return from state s following policy π. Similarly, we define the action-value function, $q_\pi(s, a)$, which is the expected return after taking action a from state s and subsequently following policy π. The action-value function is particularly useful in cases where the underlying dynamics of the MDP are not known, as discussed in the next section.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.3}$$

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.4}$$

The goal of reinforcement learning is then to find an optimal policy π^* that achieves the optimal value and action-value functions $v^*(s) = \max_\pi v_\pi(s)$ and $q^*(s,a) = \max_\pi q_\pi(s,a)$. Here we also define a greedy policy as one which takes the action that maximizes the action-value function at each step i.e. $max_a q(s,a)$.

2.1.1 Solving an MDP

When the transition and reward dynamics of the MDP are known, we can decompose the definitions of the value and action-value functions in order to obtain the Bellman expectation equations. Using the definition of $v^*(s)$ and $q^*(s,a)$ we can also form the Bellman optimality equations. These are given for the value function in equations 2.5 and 2.6.

$$v_\pi(s) = \sum_{a\in A} \pi(a|s)\left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_\pi(s')\right) \tag{2.5}$$

$$v^*(s) = \max_{a\in A}\left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v^*(s')\right) \tag{2.6}$$

The recursive nature of these expressions allows us to use dynamic programming in order to solve them. Policy iteration and value iteration are popular techniques that use the Bellman expectation and optimality equations respectively (see [2] for more details).

2.2 Value-based model-free prediction and control

In general, the underlying dynamics of the MDP are not known and these techniques cannot be used. Generally, only sampled experience from the MDP is available - the agent chooses an action $a \in A$ from its current state s and the environment returns a reward R and a new state s'. Here we focus on value-based RL methods that attempt to learn the value or action-value functions for an MDP and then derive an implicit policy (e.g. by acting greedily - selecting the action with the largest value) based on these values (in contrast to policy based methods, which are discussed briefly in section 2.5.2). In general, the action-value function is used instead of the value function as MDP knowledge is *not required* for greedy improvement. Concretely, given a learned action value function $q(s,a)$, we can simply choose the optimal action

as $max_a q(s,a)$ which does not require $P^a_{ss'}$ to be known (whereas the value function requires $max_a(R^a_s + \gamma \sum'_s P^a_{ss'} v_\pi(s))$).

2.2.1 Monte Carlo Learning

In Monte Carlo (MC) reinforcement learning we sample full episodes from our environment (i.e. the agent continues taking steps until a terminal state is reached) and update the action-value function incrementally after each episode, using the observed returns, G_t, as target values. An example update rule for MC learning (where α is a learning rate):

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha(G_t - q(S_t, A_t)). \tag{2.7}$$

2.2.2 Temporal Difference Learning - TD(0)

TD(0) is an alternative algorithm that works online i.e. we make updates after each step of an episode rather than waiting until the episode terminates as in the MC case. In many cases, this makes TD learning preferable as more frequent updates are made and the algorithm can be used in continuing environments, i.e. MDPs with no terminal state. An estimate of the return is made using the observed reward, R_{t+1}, and the estimate of the value or action-value function, $V(S_{t+1})$ or $q(S_{t+1}, A_{t+1})$. This is referred to as *bootstrapping*. An example update rule for $q(S, A)$:

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)). \tag{2.8}$$

The $R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$ term is referred to as the *TD target* (since we update towards this value) and what's contained within the bracket is referred to as the *TD error* and is the discrepancy between the expected value and the estimated value based on the reward signal.

2.2.3 TD(λ)

There is a bias/variance trade-off between the algorithms presented in the previous two subsections. TD(0) is biased since it *bootstraps* its estimate of the value function, however it is low variance since it only incurs noise from a single step. MC learning incurs noise from several steps so has higher variance but uses exact returns so is unbiased. The TD(λ) algorithm generalizes the two approaches.

Define the n-step return $G^{(n)}_t$ and λ-return G^λ_t as follows:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n}) \tag{2.9}$$

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}. \tag{2.10}$$

TD(λ) then updates towards the λ-return as:

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha(G_t^{\lambda} - q(S_t, A_t)). \tag{2.11}$$

Naively TD(λ) appears to require full episodes in order to compute the λ-return, however we can implement the algorithm online using eligibility traces, which are discussed in appendix A.

2.2.4 Policies - exploration vs exploitation

The only policy discussed so far is the greedy policy, one in which we choose the action that gives the highest expected return from a given state. Since we learn an *estimate* of the action-value function, this policy may be far from optimal as the agent will not *explore* new states. A simple improvement (that is still used in many state-of-the-art applications) is to use an ϵ-greedy policy in which a random action is chosen with some small probability ϵ and the optimal action is chosen with probability $1 - \epsilon$. Clearly, larger values of ϵ will encourage more exploration.

Decaying ϵ-greedy is a slightly more sophisticated alternative, where an initial value of ϵ is chosen and this value is decayed (e.g. linearly) at each step, such that the policy converges to a greedy policy. This encourages greater exploration initially when the estimate of the action-value function is poor and it exploits the more accurate action-value function after several timesteps.

2.2.5 Sarsa and Q-learning

RL algorithms can be further divided into on- and off-policy methods. On-policy algorithms choose actions according to a policy π and update the action-value function according to the same policy π, i.e. they learn "on the job". Off-policy algorithms choose actions according to a behavioural policy μ but update the action-value function according to a target policy π. This approach allows us to use an exploratory policy for choosing actions (e.g. ϵ-greedy) and an exploitative policy for updating the action-value functions (e.g. greedy).

Sarsa is an on-policy algorithm that uses TD(0) updates, as illustrated in algorithm 1 below. Clearly this can also be extended to use the n-step or λ-return rather than the 1-step return.

Algorithm 1 Sarsa

Initialize $q(s, a)$ arbitrarily
for each episode (until s is terminal) **do**
  Initialize state s
  Choose action a from s using policy π
  for each step of episode **do**
    Take action a, observe r, s'
    Choose a' from s' using policy π
    $q(s, a) \leftarrow q(s, a) + \alpha(r + \gamma q(s', a') - q(s, a))$
    $s \leftarrow s'$; $a \leftarrow a'$
  end for
end for

Q-learning is a similar algorithm that uses off-policy learning. Actions are chosen according to an ϵ-greedy behavioural policy μ and the action-value function is updated according to a greedy target policy π. Variants of Q-learning have been used in many impressive applications, notably in Google Deepmind's application to Atari games [5].

Algorithm 2 Q-learning

Initialize $q(s, a)$ arbitrarily
for each episode (until s is terminal) **do**
  Initialize state s
  Choose action a from s using ϵ-greedy policy μ
  for each step of episode **do**
    Take action a, observe r, s'
    $q(s, a) \leftarrow q(s, a) + \alpha(r + \gamma max_{a \in A} q(s', a) - q(s, a))$
    $s \leftarrow s'$; Choose a from s'
  end for
end for

2.2.6 Function approximation

In many cases, representing the action-value function explicitly for each state-action pair is computationally in-feasible due to the size of the table required. Many applications may have continuous state or action spaces which can also not be represented

exactly in a tabular way. As a result, many practical applications use *function approximation* to represent the action-value function. The state and/or action is represented by a feature vector and an estimate of the action-value function is learned based on this feature vector (usually using a neural network, as discussed in section 2.4). An example feature vector is given in [6], where the authors created 1.5 million features based on board configurations in order to create an RL agent able to play Othello. Similarly, IBM's famous chess-playing Deep Blue algorithm used more than 8000 hand-crafted features [7].

Linear function approximation is the simplest case, in which the value function for a particular state is represented as $\mathbf{w}^T\mathbf{x}$ where $\mathbf{x}$ is a feature vector and $\mathbf{w}$ is a learned vector of coefficients. The standard tabular (i.e. no function approximation) case can be considered a special case of linear function approximation with a binary feature vector $\mathbf{x} = (\mathbb{1}[S_t = s_1], \mathbb{1}[S_t = s_2], ..., \mathbb{1}[S_t = s_N])$.

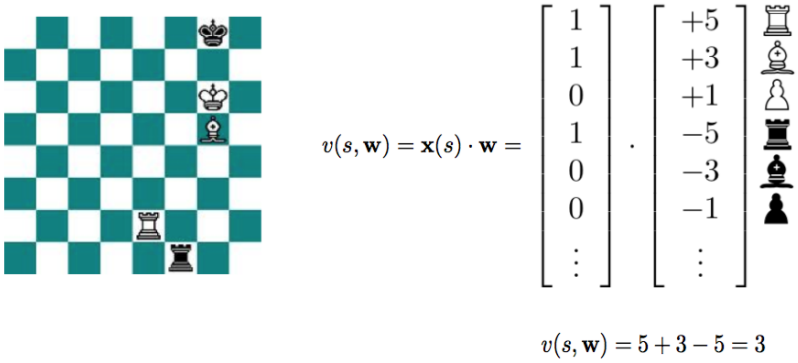

Figure 2.2: Simple linear function approximation used in chess - value is the weighted sum of the pieces left on the board. *Credit: David Silver* [8].

Many of the theoretical guarantees of common reinforcement learning algorithms break down when function approximation is used, leading to a possibility that the algorithm will diverge. This is particularly true in the case of off-policy, bootstrapping algorithms such as Q learning, especially when non-linear function approximators are used [9]. The 1997 paper from Tsitsiklis and Van Roy [10] provides a rigorous, mathematical example of a TD algorithm diverging under non-linear function approximation. In practical applications, several tweaks are used to prevent this happening and to improve the stability of the algorithm [11], as discussed in section 2.4.

2.3 Neural Networks and Optimization

Here we present a brief introduction to neural networks and the optimization techniques used to train them. The Deep Learning Book [12], Convex Optimization, Boyd and Vandenberghe [13] and Bayesian Reasoning and Machine Learning, David Barber [4] appendices 4-6 are used as general references.

2.3.1 Introduction

Neural networks are layered, graphical representations of functions. They may consist of several layers, each of which computes a simple function of the inputs. These simple functions are composed to allow complex relationships to be learned. In simple, "feed-forward" neural networks, each layer corresponds to a linear transformation of the inputs followed by some non-linear *activation function*, commonly *sigmoid*, *tanh* or *ReLU* [14]. Each layer is made up of several neurons, where each neuron simply takes inputs $\mathbf{x}$ and outputs a value y where $y = f(\mathbf{w}^T\mathbf{x} + b)$, $\mathbf{w}$ and b are parameters of the network and f is an activation function. A vector of inputs is passed to the network which may contain several *hidden* layers before a final output layer which produces an output $\mathbf{y}$. Networks with several hidden layers are generally referred to as *deep neural networks*.

Neural networks are powerful function approximators; under certain conditions a single-layer neural network with enough neurons can approximate any continuous function, as shown in Cybenko's 1986 paper [15]. They have rapidly increased in popularity in recent years due largely to improvements in GPU technology that makes training deep networks with many layers feasible [16]. Deep learning has achieved start-of-the-art performance in several fields, notably in image and speech recognition [17] and to master the game of Go as a component of Google's AlphaGo algorithm [3]. While results can be very good, training deep neural networks can be difficult and hyper-parameters must be carefully tuned. Some training techniques are discussed in the following sub-sections, with a more detailed discussion provided in [18].

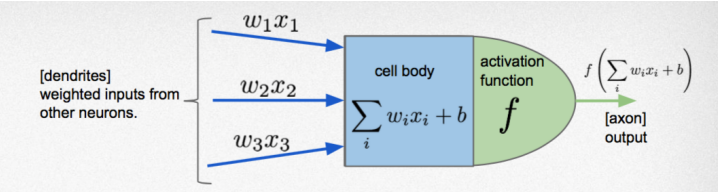

Figure 2.3: A single neuron, where f is some activation function. *Credit: Simon Osindero* [14].

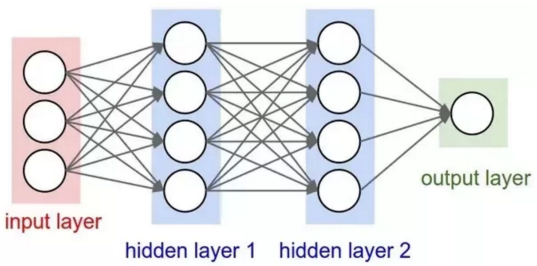

Figure 2.4: A simple neural network with two hidden layers. *Credit: Stanford CS231n* [19].

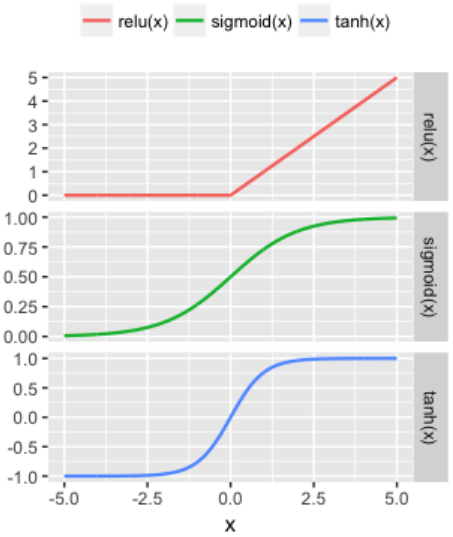

Figure 2.5: Common activation functions used in neural networks

For supervised learning problems, a differentiable loss function, L, that measures the discrepancy between the network output and the true labels is used in order to train the network. Convex loss functions are often used since they make optimization easier. The *backpropagation* algorithm [20] is used to propagate gradients through the network in order to compute gradients of the loss function with respect to the network parameters and thus perform parameter updates.

2.3.2 Optimization in Neural Networks

We consider first-order optimization approaches (approaches based on the first derivative of the loss function) to the problem of finding the optimum of a neural network loss function, $L(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the parameters of the network. While second-order approaches such as the Gauss-Newton method can also be successful, these can be computationally difficult to implement for deep networks with several parameters (as discussed in [12] section 8.6) and are not discussed in this section.

2.3.2.1 Gradient Descent

A simple and popular method to train a neural network is via gradient descent. The entire training set is used to compute the gradient of the empirical loss function with respect to the network parameters $\boldsymbol{\theta}$, which are then updated in the direction of steepest decrease in the loss function i.e. in the direction of its gradient.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_\theta L(\boldsymbol{\theta}_t) \tag{2.12}$$

These updates are repeated until a convergence criterion is met. The learning rate must be tuned in order to achieve optimal performance which can be difficult since gradient descent with too small a learning rate can be very slow to converge and using too large a learning rate can lead to divergence. Better performance may be achieved by varying the learning rate throughout training e.g. using a higher learning rate initially to make larger jumps and decreasing this as we approach the optimum.

In practice, mini-batch gradient descent (or *stochastic gradient descent* for batches consisting of a single input) is often used in which a batch of inputs is used to compute the updates, in place of the entire training set, allowing for faster, more frequent updates.

2.3.2.2 Momentum

While gradient descent is simple and easy to implement, progress at each step can be slow and alternative methods can achieve much faster convergence. Figure 2.6 illustrates the "zig-zag" performance of gradient descent on a simple 2d quadratic function.

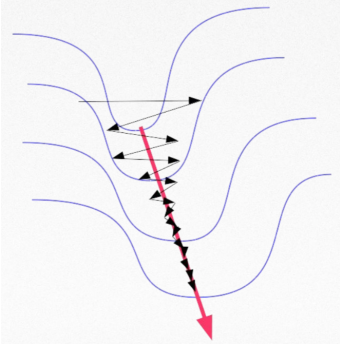

Figure 2.6: Gradient descent with a small step size applied to a 2d quadratic function. *Credit: James Martens, Google Deepmind* [21].

Momentum attempts to solve this issue by making updates in the weighted average direction of previous updates, allowing us to accelerate along directions that consistently decrease the loss function and steadily make progress towards the optimum.

$$\begin{aligned} \boldsymbol{\eta}_{t+1} &= \mu \boldsymbol{\eta}_t - \epsilon \nabla_\theta L(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta_t} + \boldsymbol{\eta}_{t+1} \end{aligned} \tag{2.13}$$

Nesterov's accelerated gradients is a commonly used variant of momentum that computes the gradient with respect to the *future* parameters [22], as shown in equation 2.14 below, that often works better than the standard momentum approach [23].

$$\begin{aligned} \boldsymbol{\eta}_{t+1} &= \mu\boldsymbol{\eta}_t - \epsilon\nabla_\theta L(\boldsymbol{\theta} + \mu\boldsymbol{\eta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \boldsymbol{\eta}_{t+1} \end{aligned} \tag{2.14}$$

2.3.2.3 RMSProp and Adam

The above methods use the same learning rate for each of the parameters (each component of $\boldsymbol{\theta}$) and make updates in proportion to the gradient vector. RMSProp is an adaptive learning rate method, originally proposed by Geoffrey Hinton [23], which adjusts the learning rate separately for each parameter by considering the sign of its previous gradients, making results significantly less dependent on the chosen initial learning rate. It is an extension of the earlier RProp method [24] to use mini-batches. Adam (Adaptive Moment Estimation) [25] is an alternative method that also adapts learning rates individually for each parameter and uses a decaying average of previous gradients, in a similar way to momentum [22].

2.3.3 Parameter Initialization in Neural Networks

Deep neural networks with several hidden layers can be prone to vanishing gradients. The backpropagation [20] algorithm uses the chain rule to propagate gradients through the network, meaning parameter gradients in deep networks consist of several products of gradients of activation functions with respect to their inputs. Gradients of commonly used activation functions, such as *sigmoid* and *tanh*, can saturate to zero for very large or very small inputs which can lead to vanishing gradients.

Careful initialization of parameters can help to prevent this from occurring. Here we present a simple, popular strategy as discussed in detail in [26], which provides a full mathematical justification. For standardized D-dimensional inputs $\mathbf{x}$, it's possible to show that initializing the weights of the first layer by drawing them from a zero mean and variance $\frac{1}{D}$ distribution keeps the activations of the first layer (i.e. the inputs to the activation function, $\mathbf{w}^T\mathbf{x}$) in a similar range to the inputs $\mathbf{x}$, mitigating against very large or small values that will saturate the activation function. We can use a similar technique for future hidden layers by drawing weights for layer l from a zero mean, $\frac{1}{D_{l-1}}$ variance distribution where D_{l-1} is the output dimension of the previous layer.

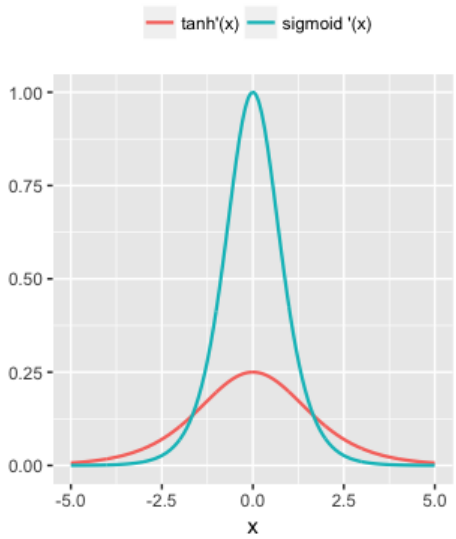

Figure 2.7: Gradients of *tanh* and *sigmoid* functions

2.3.4 Convolutional Neural Networks (CNNs)

CNNs are neural networks designed to take grid-like inputs [12]. They are popular in image classification tasks where an image is represented as a 2-dimensional array of real numbers. CNNs consist of a series of convolutional and pooling layers followed by fully connected layers, each described briefly in the following sub-sections. Stanford's CS231n course notes [19] are used as a general reference.

2.3.4.1 Convolutional Layer

A convolutional layer slides a filter over the input image, computing dot products with each element in order to produce an output. The size of the filter simply refers to the number of input elements included in the convolution operation. Figure 2.8 shows a convolution with a 2×2 filter applied to a 3×4 array. Adding a bias term is common, and activation functions (such as those described above) may be used to allow non-linear relationships to be learned. The convolution operation is applied with a *stride*, which simply determines how many elements are skipped between successive filters and affects the size of the output image, with larger strides used to produce smaller outputs. Figure 2.8 has a stride of 1 i.e. no elements are skipped and the filter moves one unit at a time. A stride of 2 is shown for the pooling operation in figure 2.9 below.

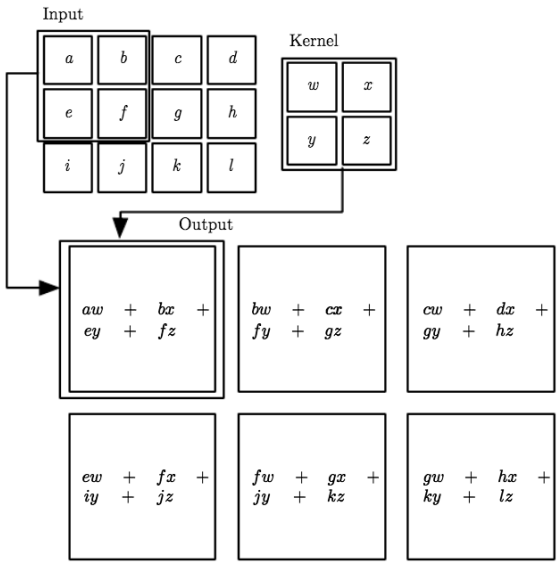

Figure 2.8: Convolution operation, here 'filter' is referred to as a 'kernel' [12].

2.3.4.2 Pooling Layer

Convolutional layers are often followed by pooling layers, which summarise the information in the resulting output by performing aggregation operations such as *mean* and *max*. Similarly to a convolution, a filter slides over the grid and computes the operation over the elements in the filter. A *max-pooling* operation with a stride of 2 is shown in figure 2.9.

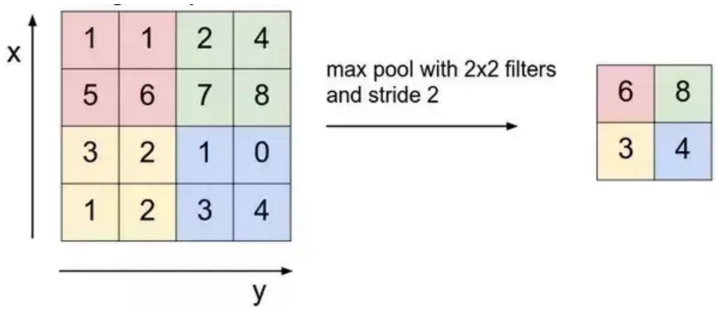

Figure 2.9: Max-pooling operation with a stride of 2. *Source: Stanford CS231n* [19].

2.3.4.3 Fully-connected Layer

In order to produce a vector of outputs, for example the probabilities of an image belonging to a series of classes, convolutional and pooling layers are generally followed by fully-connected layers. This 'flattens' the 2-dimensional matrix into a vector and then feeds this as in input into a standard feed-forward neural network which may

have several hidden layers before returning an output vector.

2.4 Deep Reinforcement Learning

In this section, we discuss the use of neural networks as function approximators in model-free, value-based reinforcement learning algorithms, with a focus on deep Q-learning in accordance with the method presented in chapter 3. Yuxi Li's 2017 paper [9] provides an up-to-date and comprehensive overview of the topic, including brief discussions of several applications of deep RL.

2.4.1 Deep Q-learning

Deep Q-learning works very similarly to the Q-learning algorithm presented in algorithm 2, however the action-value function is now represented by a neural network with parameters $\boldsymbol{\theta}$ as $q(s, a; \boldsymbol{\theta})$. The neural network is generally trained by minimizing the squared loss between the Q-learning target and the current action-value function after observing a sample from the environment (s, a, s', R).

$$L = \frac{1}{2}\big(R + \gamma max_{a'} q(s', a'; \boldsymbol{\theta}) - q(s, a; \boldsymbol{\theta})\big)^2 \tag{2.15}$$

In order to update the parameters such that $q(s, a; \boldsymbol{\theta})$ moves towards the target, the gradient is prevented from propagating through the target (implemented in the Tensorflow library [27] with the `stop_gradient()` function).

$$\nabla_{\boldsymbol{\theta}} L = \big(R + \gamma max_{a'} q(s', a'; \boldsymbol{\theta}) - q(s, a; \boldsymbol{\theta})\big)\nabla_{\boldsymbol{\theta}} q(s, a; \boldsymbol{\theta}) \tag{2.16}$$

This general algorithm with various heuristic tweaks to improve stability has achieved state-of-the-art performance in many notable domains [28]. While practical performance can be very good, training is often unstable and mistakes can be difficult to debug. Figure 2.10 is taken from Google Deepmind's original Atari paper [29] and illustrates how performance throughout training can be very noisy. Common adjustments used to stabilize training along with techniques to highlight problems in deep RL are discussed in the following sections, with John Schulman's *Nuts and Bolts of Deep RL* [30] providing a comprehensive reference.

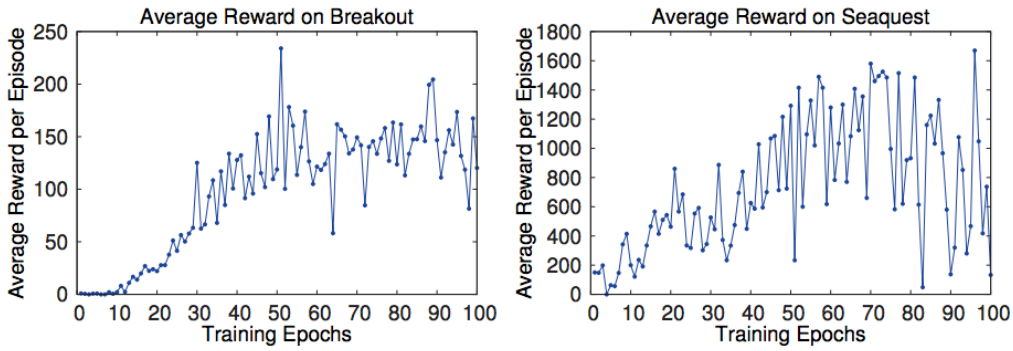

Figure 2.10: Deepmind Atari performance throughout training [29].

2.4.2 Experience replay buffer

Deep reinforcement learning differs from a standard supervised learning task in which the data is assumed independent and identically distributed (i.i.d) since successive inputs are highly correlated. This can cause problems for many of the gradient descent based algorithms used to train neural networks that rely on this i.i.d data assumption.

Many deep RL implementations use an *experience replay buffer* [31] to help solve this issue by storing each experienced transition and reward in a buffer. A random sample is taken from the buffer at each step and fed into the neural network in order to learn the action-value function. This helps to break correlations between the inputs and takes us closer to the familiar i.i.d setting from supervised learning. It also means we sample from a wider range of experiences at each step, so we can learn about parts of the environment that the agent is not currently in. Since the policy will change significantly throughout training, the buffer is generally cleared after a fixed number of steps to ensure the agent is learning from recent experience.

2.4.2.1 Prioritized Experience Replay

Schaul et al. present an alternative to random sampling in their 2015 paper [32] to make faster learning progress by selecting transitions with a larger TD error. Naively selecting transitions based on the size of the TD error causes a lack of diversity in the agent's experience which in turn can lead to overfitting. This is overcome by using a sampling method which interpolates between a greedy (highest TD error) prioritization and uniform random sampling by drawing from a distribution

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.17}$$

where p_i is simply calculated according to the rank as $p_i = \frac{1}{rank(i)}$ and α is treated as a hyper-parameter, with $\alpha = 0$ corresponding to uniform random sampling.

Moving away from random sampling introduces bias, since values correspond to the expected return under the distribution that generated them, which is altered by using prioritized replay. Weighted importance sampling is used to overcome this issue with weights calculated as

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta} \tag{2.18}$$

where N is the mini-batch (sample) size and again β is treated as a hyper-parameter and optimized. Updates are then made in proportion to these weights.

Prioritized replay combined with the DQN algorithm (see next subsection) [5] achieved a new state-of-the-art in the Atari domain [32].

2.4.3 Target Network

Another potential problem that may arise in deep Q-learning is that the algorithm will "chase its own tail" because it updates by boostrapping towards an action-value function that changes on every step of the algorithm (since a change in the neural network parameters will change the action-value estimate for *all* states).

In order to fix this, two neural networks (with the same architecture) are used instead of just one. One of the neural networks, with parameters $\boldsymbol{\theta}$, is updated at every step while the parameters of the other network, $\boldsymbol{\theta}^{-}$, are kept constant for a fixed number of steps. The parameters from the most up-to-date network are then copied over at regular intervals. The fixed neural network is used to compute the Q-learning target, $R + \gamma max_a q(s', a, \boldsymbol{\theta}^{-})$, in order to break correlations between the Q-network and the target, stabilizing the training process. Deep Q-learning with an experience replay buffer and a target network gives the Deep Q-Network (DQN) algorithm [5] shown in figure 2.11 [33].

2.4.4 Double Q-learning

The 2010 paper from Van Hasselt [34] attempts to deal with bias in the DQN algorithm. Re-writing the Q-learning update from algorithm 2 in terms of the $argmax$ (equation 2.19) makes it clear that the algorithm uses the same values to select and to evaluate actions.

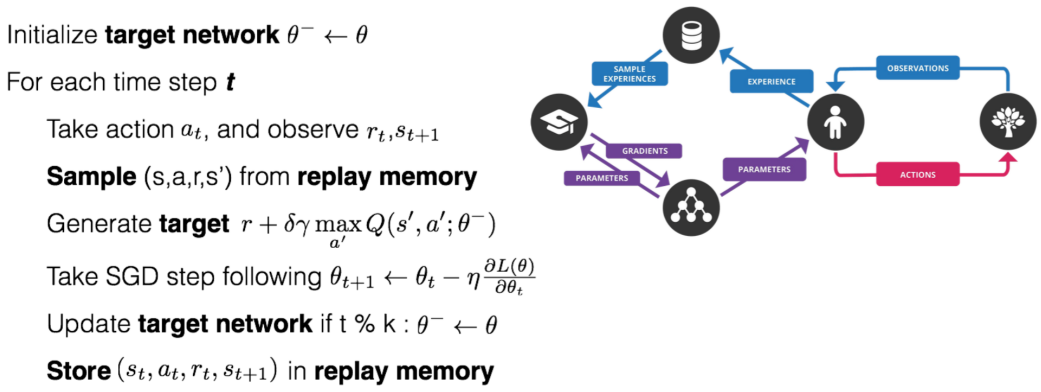

Figure 2.11: DQN algorithm [33].

$$R_{t+1} + \gamma q(S_{t+1}, argmax_a q(S_{t+1}, a)) \tag{2.19}$$

This can lead to overestimation and upward bias since the algorithm is more likely to select actions with over-estimated values. We can remedy this by using a second neural network to decouple the selection and evaluation and by updating one at random after each experience. We update either q or q' after each step and use the following as the Q-learning target.

$$R_{t+1} + \gamma q(S_{t+1}, argmax_a q'(S_{t+1}, a)) \tag{2.20}$$

This method is used in conjunction with the DQN algorithm in the 2015 paper from the same author [35].

2.4.5 Asynchronous Methods for Deep Reinforcement Learning

This late 2016 paper from Minh et al. [36] provides an alternative to experience replay, in which several RL agents are asynchronously executed in parallel and the experiences of all the agents are used to update the parameters of a common neural network. Since different agents will explore different parts of the environment at a given time step, this achieves a similar goal to experience replay by de-correlating the inputs to the network. Since a buffer is not required and training does not need to be carried out in batches, the approach can be faster and more data-efficient than using experience replay. The approach is applied within several RL algorithms, including Sarsa and Q-learning, achieving impressive results in the Atari domain.

2.4.6 Debugging Deep RL

Deep RL algorithms can be notoriously difficult to debug; a recent article from John Schulman of OpenAI [28] found subtle bugs in many popular research implementations. Given the relatively large number of hyper-parameters involved, it can be difficult to distinguish errors in the code from poorly tuned hyper-parameters.

As mentioned in section 2.4.2, samples are not generated from a static underlying distribution as in the supervised learning case since the agent's policy evolves throughout training. The neural network training loss is therefore likely to be noisy and it's important that we also monitor the value or action-value function throughout training. Simple general advice suggests starting with a smaller test problem and visualizing the learned action-value function and corresponding policy (and comparing with a random policy) at frequent intervals [30].

A more sophisticated technique is used in the 2016 paper from Zahavy et al. [37] in which the authors use a visualization technique called t-SNE [38] (which forms lower-dimensional representations of high-dimensional spaces for visualization purposes) to visualize the state space, in order to gain a better understanding of the policy learned by a DQN agent on various Atari games. Figure 2.12 below shows t-SNE applied to the game Seaquest, from which the Authors were able to determine the agent's strategy.

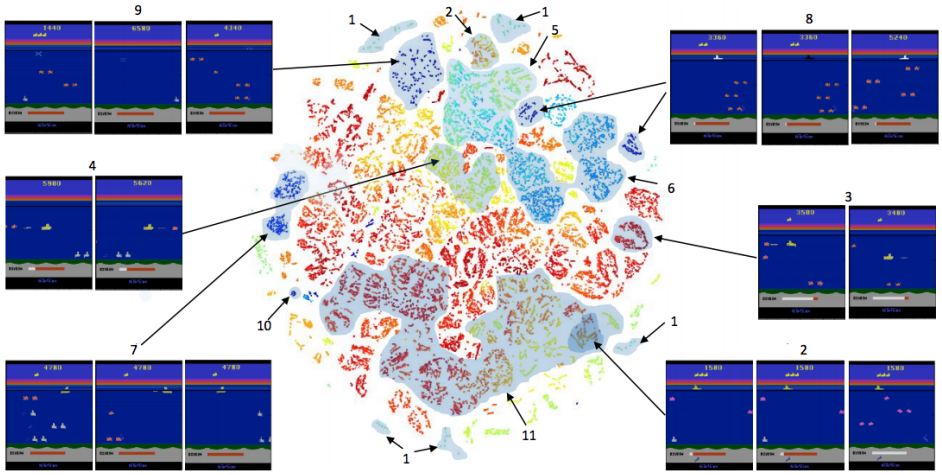

Figure 2.12: t-SNE applied to the state space of Seaquest Atari DQN [37].

2.5 Alternatives to Value-based Methods

2.5.1 Tree-based Methods

Tree-based reinforcement learning methods, particular those based on Monte Carlo Tree search have been used to achieve impressive results in recent years, most notably as part of the *AlphaGo* program [3]. Tree-based methods were initially considered for the given problem and were researched in detail, however were ultimately not used. A justification for this is presented in section 3, with an introduction to tree-learning based approaches provided in appendix A.

2.5.2 Policy-based Methods

Rather than learn the value function explicitly and derive a policy implicitly, the policy $\pi(a|s)$ can be parameterised as $\pi_\theta(a|s)$ and learned directly. This approach can be advantageous for problems with continuous action spaces and in cases where the environment or value function is complex but a simple policy can be used to solve the problem [2]. Variants of the actor-critic algorithm, which combines value-based and policy-based methods, are the current state-of-the-art in many RL domains [9]. While policy-based and actor-critic methods are not applied to the given problem, they may form a valuable extension of the work, as discussed in the following chapters. Chapter 13 of Sutton and Barto [2] provides a detailed overview.

2.6 Application to Finding Pages of Interest

While there is a relatively small number of papers applying reinforcement learning to our specific problem, this section presents similar pieces of research that apply RL to related tasks.

2.6.1 Focused Crawling with Reinforcement Learning

In the 1999 paper from Rennie et al. [39], the authors consider the task of exploring the web to find pages on a specific topic. The state is represented as the binary vector of relevant documents that remain to be discovered and the action space as the set of hyperlinks that can be followed. Rewards are received whenever relevant documents are found and the agent is able to jump to *any* hyperlink that is included in a previously visited page. It's worth noting that this state space representation is

exponential in the number of relevant documents which is likely a poor choice for our problem. Performance is measured by considering the proportion of links followed before 75% of relevant documents are found.

The 2004 paper from Grigoriadis et al. [40] considers a similar problem but the authors use the much simpler representation in which each page corresponds to a state and the set of actions from each state are just the hyperlinks from the current page. Function approximation with a neural network is used, with the state represented by a 500-dimensional binary vector indicating the presence of a series of keywords within the page. The TD(λ) algorithm is used to apply updates and the method is evaluated on a few small datasets (that were used in an earlier focused crawling paper [41]), with the largest consisting of a few thousand web pages of which around 20 are relevant.

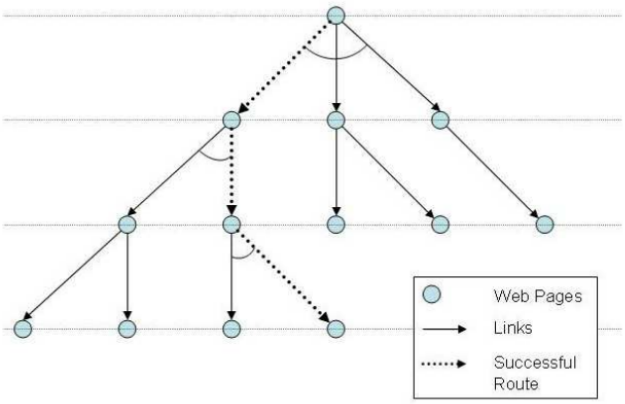

Figure 2.13: A figure showing the optimal path through a small network of web pages [40].

A similar method is used at a much larger scale in the 2010 paper 'Efficient Deep Web Crawling Using Reinforcement Learning' [42] to find particular keywords by crawling parts of the deep web. The method significantly outperforms baseline methods on five chosen experiment sites including *Yahoo Movies* and *Google Music*.

In the 2009 work from Bidoki et al. [43], a Q-learning based approach is used to find "important" pages within a web graph, with important pages judged by first running the *PageRank* algorithm on the graph. The authors note that in their initial version of the proposed algorithm, pages are not revisited which makes it difficult to frame exactly as a reinforcement learning problem. In a second version of the algorithm, web pages are revisited and this is completely mapped to the RL problem. Performance is evaluated on 18 million UK pages, in a similar way to [39], by

considering the number of important pages found after 25% of total pages have been crawled.

2.6.2 Other Relevant Work

This more recent 2016 paper from Narasimhan et al. [44], applies a DQN to an information extraction task. The model learns to improve its extractions from a source article by making use of similar relevant articles. At each step of the algorithm, the agent decides which document within a collection of documents to choose and the reward function is related to the extraction accuracy, with negative rewards given at each time step to penalize longer episodes. A state feature vector is built based on confidence scores of extracted text, *tf-idf* similarity between the source article and the new article and counts of certain context words in relevant parts of the documents [44], with experience replay and a target network used to stabilize performance. The results are impressive, out-performing alternative information extractors in several domains.

2.7 CNNs for Text Classification

In this section we provide a very brief introduction to word embeddings before discussing how these can be used in conjunction with CNNs for sentence classification.

2.7.1 Word Embeddings

In natural language processing, we may simply represent a sentence with a "bag-of-words" representation, that consists of a binary, vocabulary-sized vector with elements taking on value 1 when a word appears in the sentence and 0 otherwise. For a large vocabulary, this representation is large, likely to be sparse and does not capture relationships between particular words.

Word embeddings offer a more sophisticated alternative, in which each word in a vocabulary is represented as a real-valued vector with a much smaller size than the vocabulary. A matrix $\boldsymbol{W}$ with dimensions (*vocab size*, *embedding size*) is learned which maps from the bag-of-words representation to the embedding representation. Generally, $\boldsymbol{W}$ is learned by training on a large corpus for a specific task such as predicting future words in a sentence. Since training can be costly, pre-trained embeddings such as Google's *Word2Vec* [45] and Stanford's *GloVe* [46] are popular.

Figure 2.14 is taken from Mikolov's 2013 paper [45] and illustrates how the model has learned some information about the relationship between linguistic concepts without explicitly being given supervised examples. In this case, the figure illustrates the closest words found from the vector addition of the word embeddings.

Czech + currency	Vietnam + capital	German + airlines	Russian + river	French + actress
koruna	Hanoi	airline Lufthansa	Moscow	Juliette Binoche
Check crown	Ho Chi Minh City	carrier Lufthansa	Volga River	Vanessa Paradis
Polish zolty	Viet Nam	flag carrier Lufthansa	upriver	Charlotte Gainsbourg
CTK	Vietnamese	Lufthansa	Russia	Cecile De

Table 5: Vector compositionality using element-wise addition. Four closest tokens to the sum of two vectors are shown, using the best Skip-gram model.

Figure 2.14: *Source: Mikolov, 2013* [45].

2.7.2 CNNs for Classifying Text

Yoon Kim's 2014 paper [47] applies CNNs and word embeddings to a sentence classification problem. Consider the task of classifying a set of sentences into two classes. We first represent each sentence as a series of binary vectors based on a bag-of-words approach, and "pad" each sentence to the maximum sentence length with zeros to ensure consistently sized inputs with dimension (*sentence length*, *vocab size*). As a concrete example, a set of two sentences {"hello world", "simple test example""} with vocabulary ("hello", "world", "simple", "test", "example") would be represented by the 3×5 matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ respectively (where the number of rows, 3, corresponds to the maximum sentence length).

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \boldsymbol{B} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.21}$$

A pre-trained word embedding matrix is then used to convert the inputs to a matrix with dimension (*sentence length*, *embedding size*). A convolutional layer is applied to this matrix, with multiple different filter sizes corresponding to sliding over different numbers of words in the sentence. The length of the filter is just the size of the embedding and the width corresponds to the number of words. The outputs of the convolutional layer are fed into a ReLU activation function followed by a max-pooling layer and are then flattened and combined into a single vector. For example, if filter widths of 3, 4 and 5 were used, the filter would perform convolutions over 3, 4 and 5 words at a time, followed by ReLU and max-pooling, before flattening and

combining three outputs. This vector is then fed into a feed-forward neural network which outputs a length 2 vector corresponding to the probability of each of the two classes, as illustrated in figure 2.15. In practice, each of the filters may be applied several times, to create a longer feature vector.

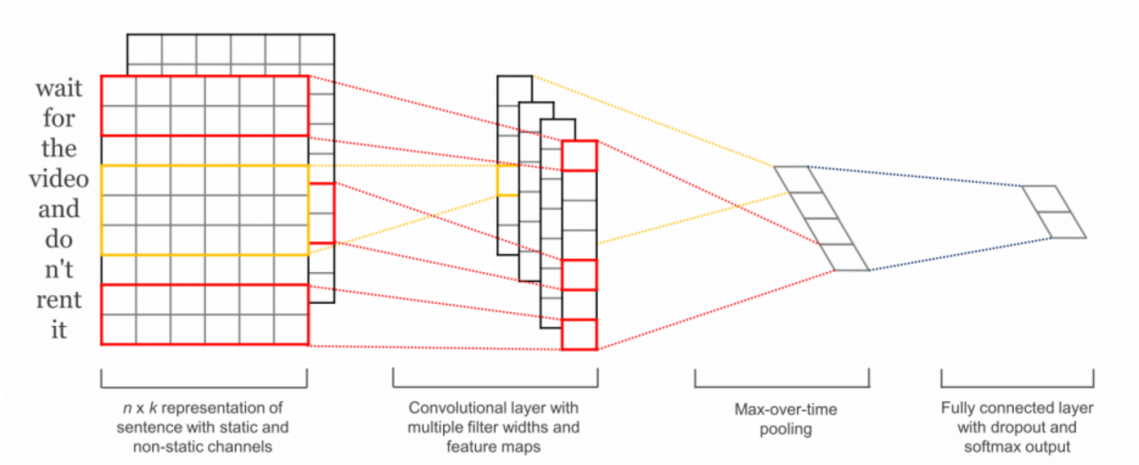

Figure 2.15: CNN for classifying sentences [47].

In the following chapter, we apply a similar method in conjunction with a Q-learning algorithm in order to estimate the value function of a web page from a URL string, using this excellent blog post [48] to help guide the implementation.

Chapter 3

Methods

In this chapter we being by outlining the available data before explicitly framing the general problem of finding specific pages within a web graph as a reinforcement learning problem. We then create our own web graph from the labelled *companies* data in order to test different approaches and compare results.

3.1 Available Data

- *Companies* dataset - 170,000 UK company names, URLs and areas of business. 3,000 of these companies are accountancy, tax and law firms of which 1,850 have *.uk* web pages. A subset of 850 of these 3,000 pages are used to build the web graph in this chapter, with the rest held out for further testing in chapter 4. These 850 pages are all verified to be in the Common Crawl. *Source: Evolution AI.*

- *Common crawl* data [49] - A publicly available dataset of over 1.5 billion crawled pages, including page text, headers and hyperlinks from the page. We use a subset of this data in order to avoid crawling a large number of web pages in real time to find links. The data is publicly available on Amazon S3 and we use Apache Spark to query this data, which is then stored locally in an LMDB [50] database.

3.2 Framing as an RL Problem

Similarly to the approach presented in [40], we let each webpage represent a state and the hyperlinks from each page represent the possible actions. The RL agent will

begin at a particular page and follow hyper-links according to it's policy π, obtaining rewards whenever a company page is found. These rewards are defined specifically in section 3.2.2 below.

In contrast to many RL implementations, the set of actions is not the same for different states since the out-going links will clearly not be the same for different pages. Consider using deep RL to solve a game with a fixed set of actions {up, down}. Given an input state vector, a neural network would output a 2d vector, with the elements representing the action-values of taking the two possible actions from the current state. This approach will not work in our case, since the number of output units would have to be different for each state. Instead, we train a neural network to output a single value, representing the *value* (rather than the action-value) function of a state given the input feature vector. A greedy policy would clearly then select the hyperlink for which the estimated value was largest. In practice, we feed a batch of N input feature vectors, corresponding to the N hyperlinks from a given URL, into the neural network which outputs a length N vector of value estimates. This works because the transition dynamics are known - if taking action a from page s corresponds to following a link to page s', we arrive at s' with probability 1 i.e. $P^a_{ss'} = 1$.

3.2.1 Function Approximation

Clearly for a large number of web pages, the state space will be very large and representing the value function explicitly is not possible. Some of the similar approaches outlined towards the end of section 2 [40] use feature representations of states based on the presence of particular words either in the entire web page or in the vicinity of hyperlinks. While these approaches are successful in the papers presented in the previous chapter, there is a significant computational cost incurred to check whether particular words appear in each of the linked pages at every step of the algorithm. To avoid incurring this cost, we use the URL string of the web page in order to construct a feature vector and begin with a simple bag-of-words approach before using word embeddings as discussed in section 2.7.

3.2.2 Starting Problem

Following the advice in [30], we first build a simpler starting problem in order to test different approaches. One significant difficulty that may be faced is sparse rewards. Even if just the *.uk* web pages are used, the resulting web graph is extremely large and

company pages are likely to comprise just a small fraction of this space. Few reward signals can make learning a meaningful value function difficult, as was noted for the game *Montezuma's Revenge* after the Atari DQN was first introduced, prompting alternative methods aimed at dealing with sparse and delayed rewards [51] [52].

We address this problem by beginning with the labelled *companies* data. In particular we start with the URLs of just under 850 known accountancy, tax and law firms and use the *Common Crawl* data to build a web-graph which includes pages that link to these 850 pages in 2 hops or fewer. We also include *.uk* outgoing links from these pages, giving a state space of approximately 2 million URLs. A simple illustration of the web graph, starting from just 2 company (reward) URLs is shown in figure 3.1 below, with counts given in table 3.3.

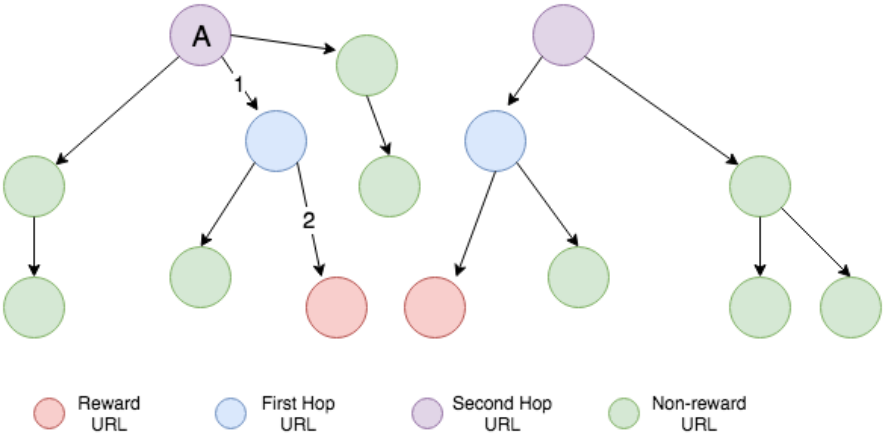

Figure 3.1: An example web graph, starting from 2 reward URLs and branching out to pages that are 2 hops away. The green nodes are the outgoing links from these pages. While the graph shown is acyclic, it's likely that a larger web graph will contain cycles. The optimal path from node A to a reward, following edges 1 and 2, is also shown.

		Number of Pages
●	Company (reward) URLs	843
●	First Hop URLs	113,174
●	Second Hop URLs	769,765
●	Outgoing (non-reward) URLs	1,149,853
-	**Total**	**2,033,635**

Table 3.1: Count of URLs by type in the web graph.

At each state, we only consider hyperlinks that link to pages within this web graph and the episode is terminated whenever a page with no out-going links in the graph

is encountered. Reaching any page in a company domain is considered a reward and the reward frequency is sufficiently high so we're able to learn relatively quickly from our environment, with a random crawler picking up rewards on approximately 1% of steps. A reward of 1 is given whenever a company domain is found and no rewards, positive or negative, are given otherwise.

3.2.3 Dealing with Repeat Rewards

Since the web graph may contain several pages for a given reward domain, repeat rewards are likely to be a problem. If the crawler agent reaches a reward state *lawfirm.com*, it is then reasonably likely to transition to another page in this domain on the next step (e.g. *lawfirm.com/contact-us*) and double count the reward. Not allowing repeat rewards also presents a problem because the agent receives conflicting signals if it reaches rewarding domains multiple times. As noted in [43], this makes the problem more difficult to map to a RL problem.

This is dealt with by allowing repeated rewards but terminating episodes whenever a reward is found, similarly to the approach in [40]. Since the web graph is large, it is reasonably unlikely that a reward will be found twice and we avoid giving conflicting reward signals to the agent. When evaluating performance we then always consider the number of *unique* rewards found by the agent.

Whenever a terminal state is encountered or a reward is found, we choose a new URL at random from the state space and continue. In order to avoid getting stuck in cycles, we maintain a list of the 50 previously visited URLs and ignore any links back to these pages.

3.2.4 Evaluating Against the Optimal Value Function

Given that the web graph is of a manageable size, it is computationally possible to calculate the exact value function under the optimal policy, $v^*(s)$, for a given URL just by using a depth-first search. For example, a web page which is one hop away from a reward URL will have value γ and a web page two hops away will have value γ^2. Any value-based RL method can be tested this way just by taking a sample of states, and comparing the estimated values learned by the RL agent with the optimal values.

3.3 Motivating Q-learning

Throughout the following sections we apply variations of Q-learning to the given problem. As discussed in the previous chapter, Q-learning based approaches, particularly the DQN algorithm, have been successful in recent years and have achieved great popularity as a result. There is a large amount of research in this area, with several variations to improve stability and practical performance in specific scenarios. Using a value-based method also allows us to directly compare the agent's learned representation to the optimal state value function and to compare results with a supervised learning approach as discussed in section 3.7 below.

Tree-based methods were also considered (discussed in appendix A). However, since some web pages will contain links to a very large number of URLs, this makes it difficult to implement tree-based methods efficiently and this method was ultimately not used for the problem. The distribution of the number of outgoing links from the pages in the web graph is shown in appendix B.

Applying actor-critic methods to the problem may be a valuable extension of the work, as discussed in section 3.10.

3.4 Initial Approach: Q-Learning with Linear Function Approximation

As an initial approach we begin by implementing Q-learning, using linear function approximation to represent the value of each state. A bag-of-words feature vector is used, with a vocabulary of words built by running a word segmentation algorithm on the URL strings using the wordsegment package in Python [53] and keeping the 10,000 words that correspond to 80% of words found, with URL endings such as 'co' and 'uk' included in this. This approach is very transparent, allowing us to determine exactly which words correspond to larger values.

A linearly decaying ϵ-greedy behavioural policy is used, starting at a value of 0.1 and decaying to 0 over the first 100,000 steps. This is observed to give slightly better performance than fixed ϵ-greedy in our case, encouraging more exploration when the value function estimate is poorer. $\gamma = 0.75$ is used to prevent myopic behaviour and encourage the agent to find the most efficient path to a reward from the starting state i.e. a state should have a relatively high value if a relevant URL can be reached within one hop. Larger and smaller values of γ were tested in the range 0.5 to 0.95, with $\gamma = 0.75$ observed to give the best performance.

The linear function approximation is implemented in Tensorflow [27] and a vocabulary size weight vector $\boldsymbol{w}$ is learned, with $\boldsymbol{w}$ initialised by drawing components from an $\mathcal{N}(0, 0.001)$ distribution. A square loss is used as shown in equation 3.1, where S refers to the set of pages that page s links to and the `stop_gradient()` function is used to prevent the gradient from propagating through the target. Both *RMSProp* and an *AdamOptimizer* are tested with an *AdamOptimizer* observed to give slightly better performance. Several learning rates are tested, in the range 0.5 to 0.0005, with 0.001 giving the best performance.

$$L(\boldsymbol{w}) = \left(R + \gamma * \texttt{stop\_gradient}\big(\max_{s' \in S} v'(s'; \boldsymbol{w}) \big) - v(s; \boldsymbol{w}) \right)^2 \tag{3.1}$$

A crawler agent following a uniform random policy is used as a baseline, this starts at a given URL and randomly follows links until either a reward or a terminal state is found, before repeating from a random URL. We use a similar evaluation metric to [43], running both the random and Q-learning agents for 200,000 steps, visiting approximately 10% of the pages in the state space, with performance evaluated by the *number of unique rewards found*.

3.4.1 Inclusion of Prioritized Experience Replay Buffer

While the mean episode length is short (with a mean length of 5.3 steps for the random crawler) and we start new episodes from a random URL, many of the successive states will still be highly correlated. In an attempt to combat this we repeat the above but include an experience replay buffer. We add our sampled experience to the buffer after each step and randomly sample 10 experiences from the buffer and train on this batch of examples. Only the most recent 5,000 experiences are stored to ensure we're training on recent examples. This is not observed to lead to an increase in performance. It's also interesting to note that altering the sample size results in very little change in performance.

Instead, we follow the method discussed in section 2.4.2.1 [32] and use a prioritized replay buffer in an attempt to make faster learning progress. A value of $\alpha = 0.5$ is used to interpolate between uniform random sampling and greedy prioritization (always choosing the samples with the largest loss). We also use $\beta = 1$ in accordance with section 3.4 of [32], fully accounting for any bias introduced from the prioritized sampling. As shown in figure 3.2 this gives very similar performance to the standard Q-learning case.

Since linear function approximation is less likely to suffer from stability issues, it's

likely that as a result using a replay buffer does not affect performance. For non-linear functional representations of the state space, the buffer may be necessary.

3.4.2 Linear Q-Learning: Results and Analysis

Figure 3.2 compares the performance of the random crawler, the linear Q-learning agent and the same agent when prioritized experience replay is used. Performance is clearly better than random for the linear Q-learning case and using prioritized replay does not further increase performance. The following analysis in this section focuses on the Q-learning agent without experience replay.

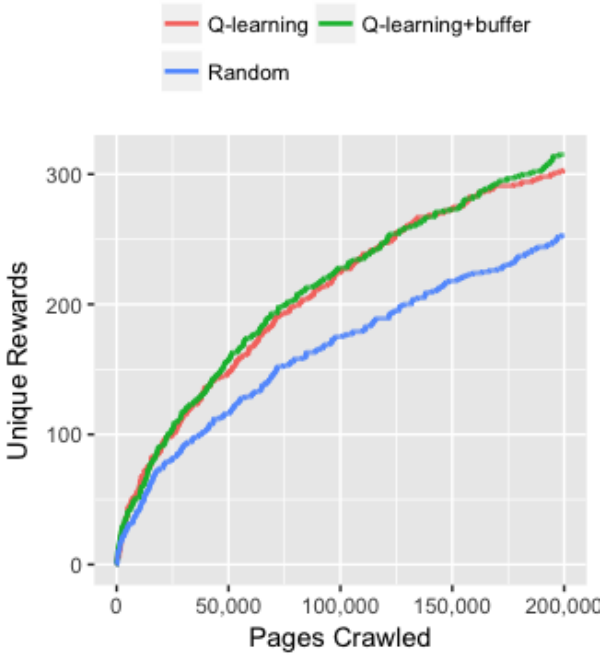

Figure 3.2: Q-learning with linear function approximation and an experience replay buffer.

3.4.2.1 Learned Coefficients

Figure 3.3 shows the coefficients of the top 20 words by magnitude of coefficient, while table 3.2 shows the coefficients of words which may be expected to have large coefficients for the specific domain.

Clearly, some of the words with large coefficients correspond exactly to reward domains e.g. ‘fca’ and ‘intuit’. These companies have very large websites with many incoming and outgoing links and as a result are reached relatively frequently by the

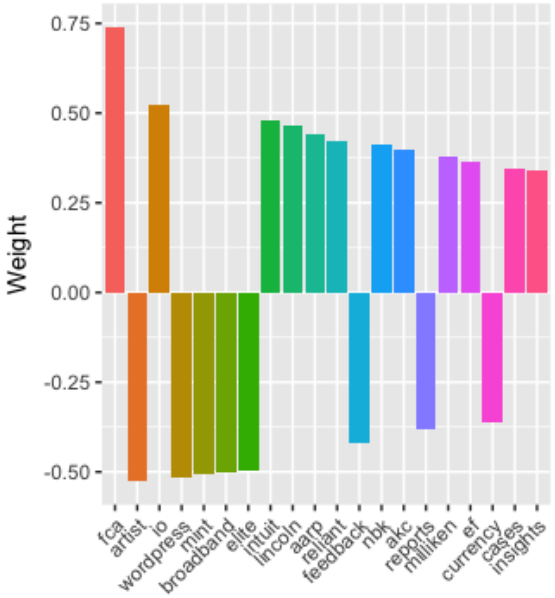

Figure 3.3: Top 20 words and weights by size of coefficient.

Word	Coefficient
taxes	-0.12
lawyer	-0.11
accountants	0.09
accountant	0.06
accounting	0.06
laws	-0.06
account	-0.05
taxation	0.05
legal	-0.04
accounts	-0.03
tax	0.03
law	-0.01
lawyers	0.01

Table 3.2: Weights of Common Words

crawler. Interestingly, large negative coefficients are also learned for particular words. Initially, the crawler will often get 'stuck' on a blog and take many steps just jumping between articles in the blog's domain before ultimately learning negative coefficients for 'wordpress' (as shown in the figure) and also 'blogspot' (-0.0745).

Table 3.2 shows the coefficients for words related to accountancy, tax and law. In some of these cases, reasonably large positive coefficients are learned, but the table indicates some of the limitations with the bag-of-words approach, e.g. positive coefficient for 'accountants' and a negative coefficient for 'account'.

3.4.2.2 Learned Value Function

In order to test the learned representation, we take a sample of 20,000 states that were *not* visited during training and compute the value function for these states, using the learned weight vector. We then use a Depth First Search to exactly compute the value function for these states under an optimal policy and compare to the estimated values.

Figure 3.4 illustrates this comparison for $\gamma = 0.75$. The plot label gives the optimal value function v^*, which can take on the values 1, $\gamma = 0.75$, $\gamma^2 = 0.5625$ or 0. The histograms give the distribution of the estimated value function for each of these values of v^*. While there is some distinction between the different values, the

predicted values are large underestimates in all cases except $v^* = 0$. This plot, along with table 3.2 suggest a richer functional representation of the state space is needed in order to accurately represent the value of different states.

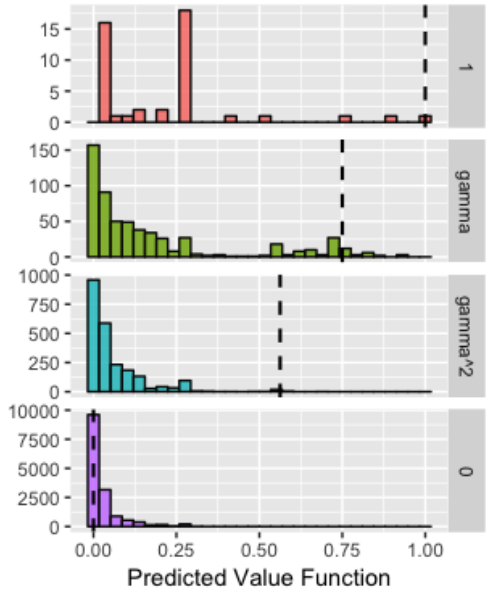

Figure 3.4: Comparing estimated value function with v^* - the dotted line indicates the optimal value.

3.5 Extensions of the Linear Q-learning Approach

3.5.1 Using Anchor Text and Page Titles

In order to enrich the state representation, we try using the anchor text of each link and the title of the current web page in addition to the URLs. Similarly to the URL strings, we build a vocabulary of words that occur frequently in the anchor text and page titles of our web graph and use a bag-of-words approach. A *separate* set of coefficients (to those learned for the URLs) is learned since the vocabularies are different.

This extension is *not* observed to improve performance. This is likely due to the fact that anchor text is not present in approximately 25% of cases, with many links occurring in the form of a logo or image rather than text on the page. This is particularly true of company pages, where links to the page are likely to be in the form of a company logo. There is also significant similarity between words in the URL string and words in the anchor text and page titles. Alternative approaches

that attempt to enrich the state representation, such as a word embedding approach, may yield better results.

3.5.2 Penalizing Long Episodes with Negative Rewards

As discussed in 2.6.2, we follow the approach of [44] and penalize longer episodes by giving a small negative reward of -0.05 on each step. This leads to a small decrease in mean episode length from 4.50 for the standard Q-learning agent to 4.29 for the Q-learning agent with the penalty term, however this does not lead to an improvement in performance. Different values were used for the negative rewards, in the range -0.01 to -0.1, without having a significant impact on performance.

3.6 DQN with Word Embedding

In an attempt to enrich the state representation we use a DQN approach with a word embedding. Google's pre-trained *word2vec* [45] model is used to represent the words in the vocabulary, producing a 50×300 matrix for each URL, where 50 is the maximum number of words in the URL string and 300 is the size of the embedding. This input matrix is normalized and fed into a convolutional neural network similarly to the approach described in section 2.7 [47] [48]. A single convolutional layer with a $ReLU$ non-linearity followed by a max-pooling operation is used with 4 filters each of size 1, 2, 3 and 4. The outputs are then combined and flattened and fed through a single-layer neural network with a *sigmoid* activation to output an estimate of the value function. Again, an *AdamOptimizer* is used with a learning rate of 0.001 and weights are initialized according to the discussion in 2.3.3. Linearly decaying ϵ-greedy and $\gamma = 0.75$ are used as in the linear Q-learning case. Similarly to [44], a target network is used in order to improve training stability, with parameters copied over to the target network every 100 steps and an experience replay buffer is also used, training with batches of size 10, with larger batch sizes not observed to give an increase in performance. Prioritized replay is also not observed to improve performance. The computation graph is shown in appendix B.

The performance of the agent is compared to the linear Q-learning approach from the previous subsection, along with a random crawler. As shown in figure 3.5, performance is improved over the linear Q-learning case, achieving an average of 11% more rewards over 200,000 steps. The plot from figure 3.4 is repeated using the embedding state representation and these results (for 20,000 unseen web pages) are shown in

figure 3.6 below. While there is some distinction between the different values, the learned value function is still clearly an underestimate of the true value in all cases except $v^* = 0$.

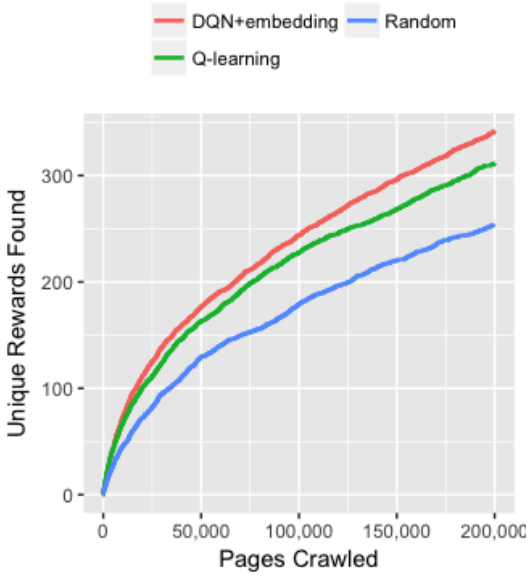

Figure 3.5: DQN with embedding compared to linear Q-learning agent (labelled as 'Q-learning'). Results are averaged over 10 runs.

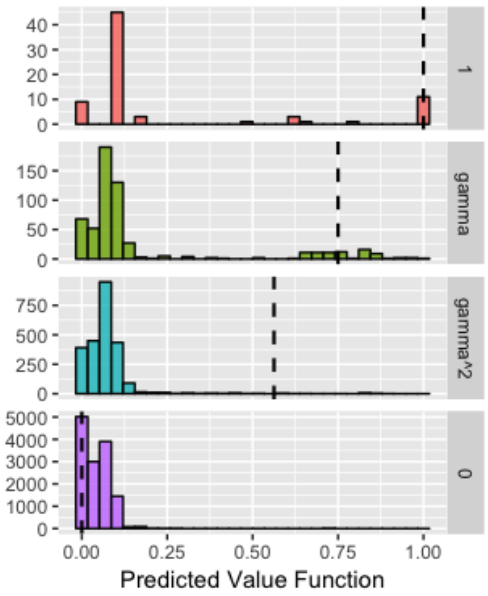

Figure 3.6: Comparing estimate for $v*$ for DQN with embedding.

3.7 Testing the Representation and Benchmarking

While performance is much better than random, figure 3.6 suggests that the feature representation may still not be sufficient to accurately learn the optimal value function (and thus an optimal policy) and we may require additional data to the URL string alone e.g. the full text or description of the web page. In order to test this, we turn this into a supervised learning problem and use both the bag-of-words and embedding representations to attempt to predict the optimal value function for a given state.

We begin with the 2 million URLs in the web graph and split this data 50:50 into train and test sets before splitting aside a further 10% of the training set to use as validation. We re-balance the classes in the training set and using each of the two functional representations discussed above (bag-of-words and embedding with CNN) train the network to predict the optimal value function by feeding in 2,000 mini-batches of size 1,000, training on a total of 2 million examples. The squared loss between the optimal and predicted value function is used (as shown in equation 3.2)

with an *AdamOptimizer* and training and validation loss plots are given in figure 3.7.

$$L(\boldsymbol{\theta}) = \sum_{s \in mini-batch} \frac{1}{2}\big(v(s;\boldsymbol{\theta}) - v^*(s)\big)^2 \tag{3.2}$$

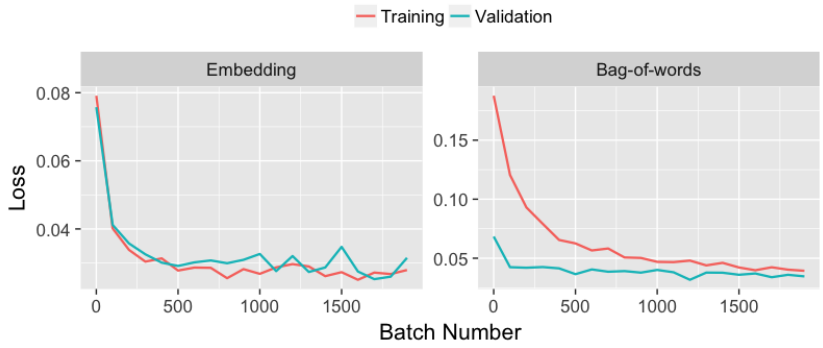

Figure 3.7: Training and validation loss for classifiers.

Similarly to above, a sample of 20,000 URLs is taken from the test set and the learned model is used to predict the optimal value function. The results of both approaches are given in figures 3.8 and 3.9, indicating that the embedding approach does a good job of predicting the value function and suggesting that using the URL string alone is likely sufficient for learning to navigate the web graph. The bag-of-words classifier performance is clearly worse, as may be expected from the results presented in section 3.6.

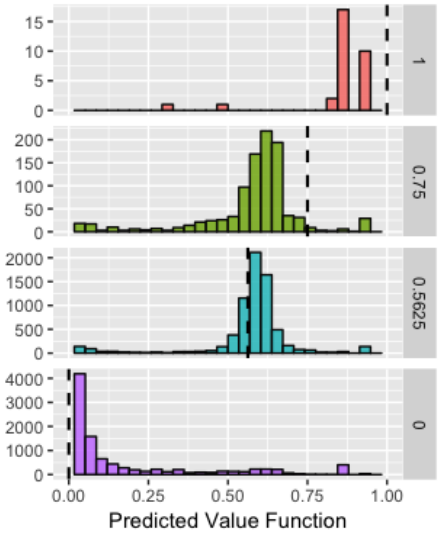

Figure 3.8: Value function for embedding classifier.

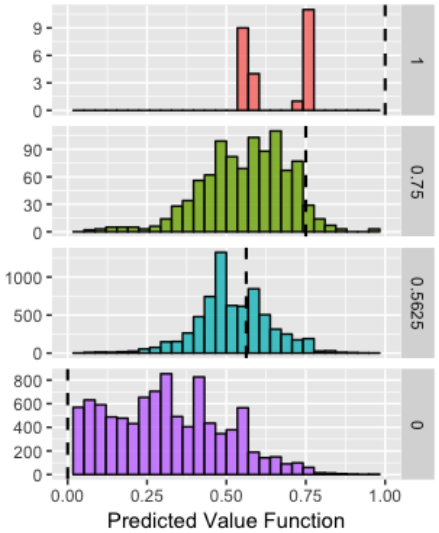

Figure 3.9: Value function for bag-of-words classifier.

While both the DQN agent and supervised embedding classifier are trained on 2 million examples (via experience replay in the DQN case), the classifier receives 5 times as many unique examples and directly targets the classification problem (rather than bootstrapping with value function estimates in the DQN case) which explains the improved performance.

3.7.1 Benchmarking

We also use the weights learned from the trained embedding classifier to choose actions inside a crawler, giving an upper benchmark on performance, which is shown alongside previous results in figure 3.10, achieving an average of 55% more rewards than the random crawler. While the performance of this crawler may be expected to be much better (when compared with random), its performance is likely to be hindered by cycles in the web graph, which make it possible to take several steps in the graph before finding a reward.

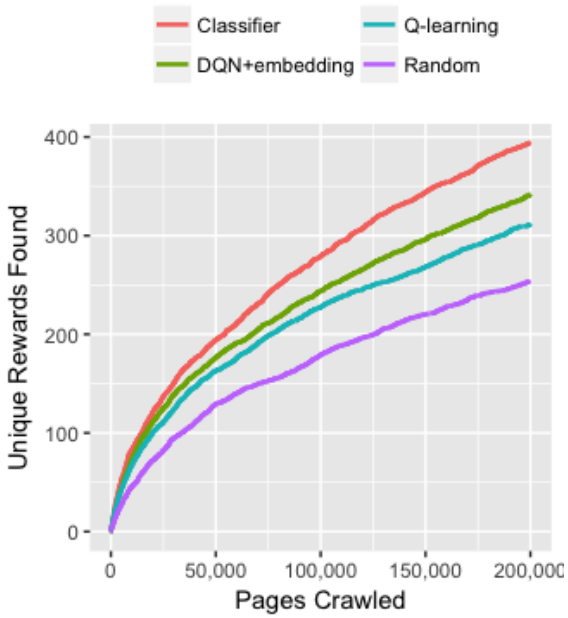

Figure 3.10: Benchmarking performance against embedding classifier (again averaged over 10 runs).

3.8 Asynchronous DQN

In order to increase the diversity of experience, we remove the experience replay buffer from the embedding DQN and instead try an asychronous approach, in accordance

with the method presented in section 2.4.5. The following blog post [54], which uses an asynchronous actor-critic method to learn to play the video game *Doom*, along with the original paper [36] are used to guide our implementation.

We implement several DQN agents across the available CPU threads (using 10 in total so we train the agent on 2 million steps, allowing comparison with previous methods) and train the CNN-embedding neural network from previous sections using the combined experience of the agents. Another thread is set aside to evaluate performance throughout training using a greedy policy. Psuedo-code is shown in algorithm 3 below and is adapted from algorithm 1 in the original paper [36].

Algorithm 3 Asychronous Deep Q-learning - for single agent thread [36]

// Parameters $\boldsymbol{\theta}$, $\boldsymbol{\theta}^-$ and T are shared globally
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
Initialize network gradients $d\boldsymbol{\theta} \leftarrow \mathbf{0}$
Choose web page, s, at random
repeat
  Check for reward r and check if s is terminal (no outgoing links)
  $$y \leftarrow \begin{cases} r, & \text{if s is terminal} \\ r + \gamma \max_{s'} v(s'), & \text{otherwise} \end{cases}$$
  Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \nabla_\theta (y - v(s))^2$
  if s is not terminal **then**
    Choose link s' from page according to ϵ-greedy policy
  end if
  $T \leftarrow T + 1$; $t \leftarrow t + 1$
  if T % $I_{target} == 0$ **then**
    Update target network $\theta^- \leftarrow \theta$
  end if
  if t % $I_{async} == 0$ or s is terminal **then**
    Perform asynchronous gradient update for θ using $d\theta$
    Clear gradients $d\theta \leftarrow 0$
  end if
until $T <$ max_steps

The results are shown in figures 3.11 and 3.12, indicating improved performance over the replay buffer case. Figure 3.12 shows that using an asynchronous approach which increases the diversity of experience allows the agent to learn a much more accurate estimate of the value function for unseen web pages, with just a small increase in running time per step (see table 3.4).

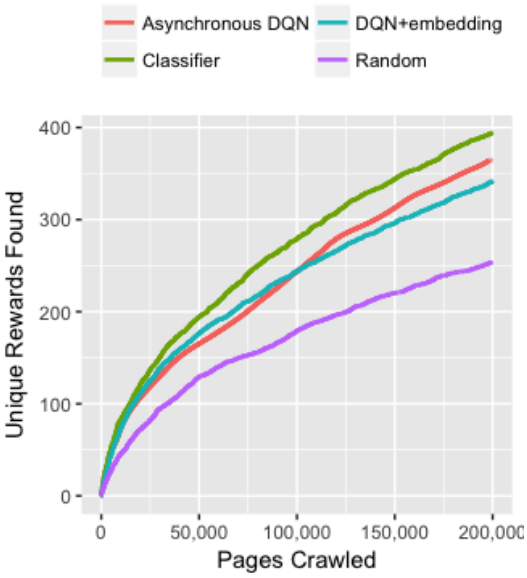

Figure 3.11: Aysnchronous DQN compared with previous algorithms. The upper benchmark given by the classifier is also shown and results are averaged over 10 runs.

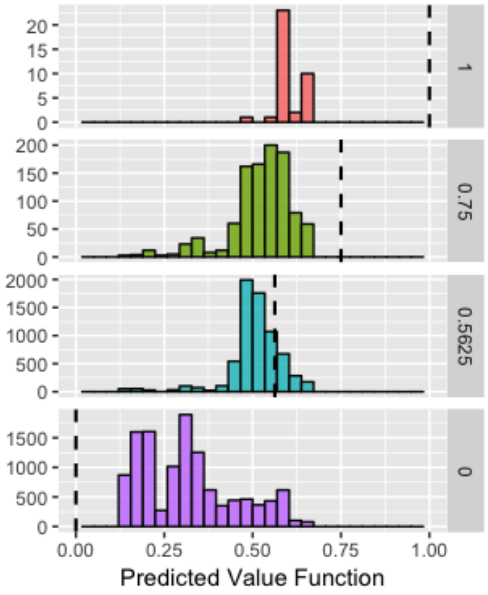

Figure 3.12: Learned value function for asynchronous DQN.

3.9 Summary of Results

Table 3.3 summarizes the mean and standard deviation in the number of unique rewards (over 10 runs) obtained after 200,000 steps for each of the methods above, with the performance of the random crawler and the crawler which uses the state representation learned from the classifier (upper benchmark) also shown. Figure 3.13 shows the mean performance throughout training along with the minimum and maximum over the 10 runs for each of the methods. These results clearly indicate that the asynchronous DQN with the CNN-embedding state representation achieves the best performance, which is reasonably close to that of the pre-trained classifier.

Method	Mean unique rewards	SD	Mean increase over random
Random Crawler	253.6	8.5	-
Bag-of-words linear DQN	311.4	11.1	22.8%
Embedding DQN	341.8	4.6	34.8%
Asychronous Embedding DQN	367.0	5.2	44.7%
Embedding Classifier	394.2	11.4	55.4%

Table 3.3: Comparison of the different methods.

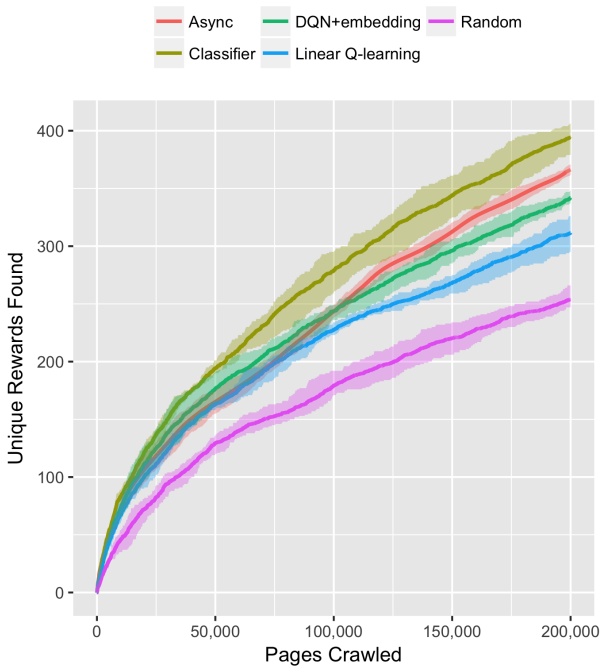

Figure 3.13: Performance throughout training.

3.9.1 Running Time

Method	Mean run time per step (s)
Random Crawler	0.0020
Bag-of-words linear DQN	0.0063
Embedding DQN	0.0158
Asychronous Embedding DQN	0.0218
Embedding Classifier	0.0110

Table 3.4: Run times for each of the methods.

As mentioned in section 3.1, the data for the web graph (URLs and links from each page) is obtained from the *Common Crawl* [49] using Spark and parsed and stored in an LMDB database in order to avoid real-time crawling. For practical application (that does not use a pre-defined web graph), the time required to download pages from the web is likely to be significantly slower than the run times presented in table 3.4 and by far the most costly step in the algorithm, with times on the order of 1s per page for a simple web crawler. As a result we are ultimately concerned with the number of rewards obtained per timestep, similarly to the work presented in section 2.6.

It is still worth noting that the bag-of-words linear DQN is more than twice

as fast as the embedding DQN and the random crawler is (as would be expected) much faster than any of the other methods. While the increase in performance of the other methods over a random crawler may not appear to justify the increased running times presented in table 3.3, for practical application this time will likely be negligible compared to the time taken to scrape the content of a web page.

3.10 Possible Extensions

In this section, we briefly discuss possible extensions of the algorithm that could be used in an attempt to improve performance. A discussion on the practical implementation of the work is deferred until chapter 4.

3.10.1 A3C: Advantage Actor-Critic Methods

While the DQN algorithm performs reasonably well, it may be possible to obtain better results with an actor-critic method, that combines the value-based and policy-based approaches. In particular, the A3C algorithm [36] applies the asynchonous approach used in 3.8 within the actor-critic framework and may be a valuable extension of the work presented above.

3.10.2 Using LSTMs to Improve State Representation

While the CNN state representation achieves reasonably good results, it may be possible to make faster learning progress by feeding the URL strings into a recurrent neural network such as an LSTM (for brevity, these are not discussed in section 2 but this blog post [54] provides a good introduction), in order to capture sequential relationships between words in the string.

Additionally, better results may be achieved by combining the two approaches, and using the output from the CNN as input to an LSTM in order to learn sequences in the output feature vectors. A similar approach is used in the 2016 paper [36] from Mnih et al. within the A3C algorithm.

Chapter 4

Application to the UK Web

In the previous chapter, we defined a web graph of approximately 2 million pages in order to tune and test deep RL methods for finding company pages. In this chapter, we apply these methods to the much larger and more realistic problem of finding information on the UK web i.e. all 57 million web pages in the *Common Crawl* [49] ending in *.uk*, comprising just over 400GB of data. This new web graph is again stored in an LMDB database to avoid real time crawling.

4.1 Expansion of Previous Problem

As an expansion of the chapter 3 problem, we attempt to find company pages within the much larger UK web graph. We use the 1.1 million UK pages from the graph defined in section 3.2.2 as starting pages and always begin episodes by random selecting one of these pages, such that we always start within 2 hops of a reward. The the same reward set of just under 850 pages is used but we filter on UK pages only, leaving 450 reward URLs. Rewards are then given whenever a page is found in one of these 450 domains, with repeat rewards handled in the same way as the previous chapter (as discussed in section 3.2.3).

The graph is much larger and more connected than in the previous problem, which allows for considerably longer episodes without finding a reward, with a random crawler taking an average of 15.9 steps per episode (compared with 5.3 for the web graph from chapter 3). The distribution of number of outgoing links in the graph is shown in appendix C, with a median of 90 outgoing links per page.

We apply the asynchonous embedding DQN to this problem and compare performance with a random crawler, using the same set of hyperparameters as the previous chapter. 11 of the 12 available CPU threads are used, with one set aside to evaluate

performance using a greedy policy.

4.1.1 Results and Analysis

Similarly to the results in chapter 3, figure 4.1 and table 4.1 show the results over 200,000 steps, averaged over 10 runs. Asynchronous DQN performance is much better than a random crawler, finding an average of 36% more rewards over the 200,000 steps. It is worth noting that the asynchronous DQN is approximately 10 times slower (per step) than the random crawler, however the mean run time of 0.027s is likely to be negligible compared to the time taken to crawl a web page (on the order of 1s).

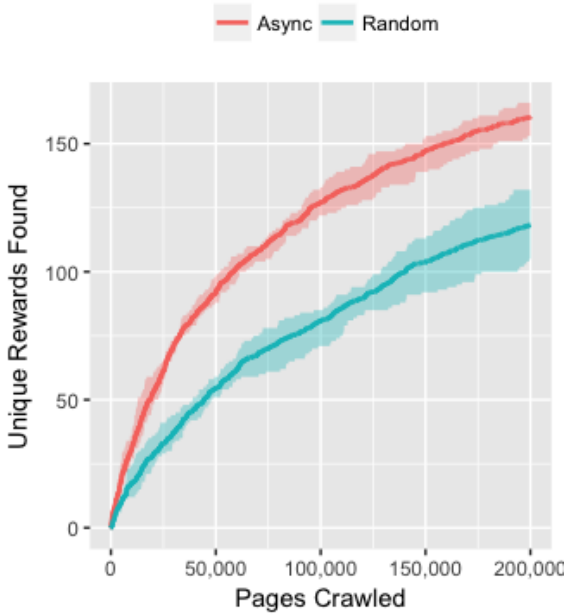

Figure 4.1: Mean and minimum and maximum performance throughout training, over 10 runs.

Method	Mean unique rewards	SD	Mean run time per step (s)
Random Crawler	118.2	10.2	0.0022
Asychronous Embedding DQN	160.4	5.6	0.0268

Table 4.1: Final performance vs random crawler for 200,000 steps, averaged over 10 runs.

Since the web graph is now much larger, figure 4.2 and table 4.2 show results for a single run over 1 million steps, visiting approximately 2% of pages in the web graph, with the DQN finding 39% more reward URLs than the random crawler.

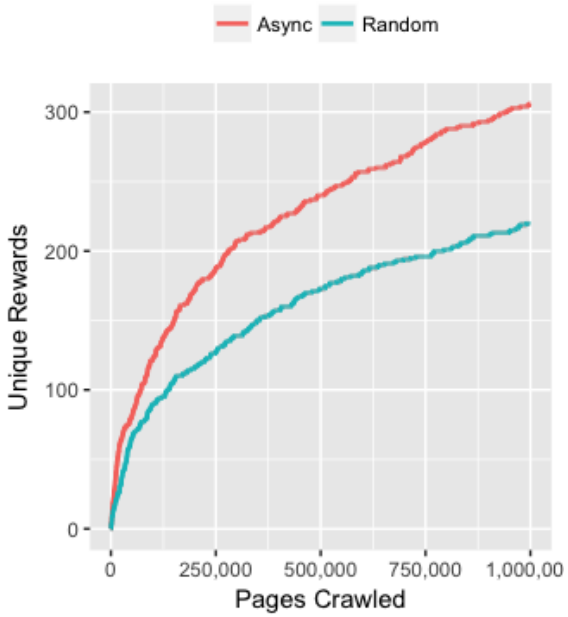

Figure 4.2: Performance over 1m steps.

Method	Unique rewards
Random Crawler	220
Asychronous Embedding DQN	306

Table 4.2: Final performance after 1 million steps.

4.1.2 Results on Held-out Rewards

As mentioned in section 3.1, there are an additional 1,400 UK company URLs which were not used to build the web graph in chapter 3 and as a result may not be within two hops of the set of starting pages. As a further test, we compare the number of these held-out reward pages found by the asynchronous DQN compared with the random crawler. Since these pages are not necessarily within two hops of a starting page, a crawler that randomly follows links is much less likely to find them.

Method	Mean held-out rewards 200k	Held-out rewards 1m
Random Crawler	19.6	106
Asychronous Embedding DQN	45.3	200

Table 4.3: Number of held-out reward pages found for the 200k and 1m step runs.

The DQN significantly outperforms the random crawler on this test, finding almost twice as many rewards over 1 million steps. This result, along with with figure 3.12, indicates that the asyncronous DQN is able to predict a reasonably accurate value

function for unseen pages.

4.2 Practical Application

In a business context, our method may be extended to find information on companies within a specific area of business, beginning from a particular set of starting pages on the web. In the more realistic case where company pages are not known *a priori*, rewards may be given when a company name or particular keywords are found in the text of a web page.

In practical applications, web crawlers are typically implemented in parallel, with several different processes downloading pages from the web. The asynchronous method is clearly very compatible with this approach, making this more useful than a replay buffer for practical applications.

4.2.1 Starting Pages

The problem above uses the pre-defined web graph from chapter 3 to choose the starting pages, which is clearly not available in practice. Instead, the crawler may be implemented by starting at common business pages (e.g. *Bloomberg*), which may be expected to link to company pages within a reasonably small number of hops, to avoid very sparse rewards.

4.2.2 Using the Common Crawl Data in Practice

In the current problem, the RL agent will only follow a link to a page if that link is contained in the UK *Common Crawl* i.e. if it is in the web graph of 57 million UK pages. However, if the value of a particular link is high, we may consider extending this approach and crawling the page anyway. In this case we have to weigh up the predicted value of following that link with the time taken to actually crawl the web page. One possible solution may be to give the agent a negative reward proportional to the time taken to crawl the page, allowing the agent to consider the predicted value of the page against the time taken to obtain its data. This extension would allows us to incorporate the saved *Common Crawl* data in practice, without limiting the scope of the web graph.

In this case, the running time of the RL algorithm becomes more of a consideration and the bag-of-words approach may be preferred, despite poorer per-step

performance, as it is more than 3 times faster than the asynchronous embedding approach (see table 3.3).

Chapter 5

Conclusion and Suggestions for Further Work

5.1 Summary

In this thesis, we have sought to use reinforcement learning to efficiently find specific pages in a web graph. In chapter 3, we demonstrated that a DQN algorithm is able to obtain significantly more rewards per timestep than a random crawler within a reasonably large graph, using only the URL string as input. We found that using a CNN-based text classification algorithm allowed us to outperform a bag-of-words approach and demonstrated using a supervised learning method that this approach is sufficient for representing pages in the web graph. Additionally, an asynchronous DQN approach was shown to improve the learned estimate of the value function and improve performance by increasing the diversity of experience.

In chapter 4, we extended this problem to the much larger web graph defined by the *.uk* web pages within the *Common Crawl*. The asynchronous DQN was applied to this graph, achieving 40% better performance over a random crawler. Additionally, the asynchronous DQN found almost twice as many held-out test pages, indicating the state representation learned by the DQN generalizes to unseen pages.

While RL for focused crawling is clearly not novel (see section 2.6), many earlier methods require significantly more data (i.e. the text of the web page as in [39], [40] and [43]) in order to achieve good performance. Additionally, much of the research in this area is out-dated and applied to much smaller web graphs. In this thesis, we have a shown that a fast, data-efficient approach (using only the URL string) in combination with modern RL algorithms is able to achieve good performance on a

very large network of web pages.

5.2 Critique and Suggestions for Further Work

In the problems considered, the pages are queried from a database rather than downloaded from the web in real time. In this case, the increased number of reward pages found by the RL algorithms are not sufficient to warrant the increased run time. However, for practical application, downloading pages from the web will be costly meaning the RL methods are likely to be preferred in practice.

As discussed in chapter 4, the most valuable extension of the work presented is to modify the method in order to obtain useful results in practice, in particular considering how to implement the method when reward pages are not pre-defined, for example giving rewards when a company name is found within a web page. Additionally, the obtained *Common Crawl* data may be able to be used in practice, by giving negative rewards when a page has to be crawled in real-time rather than queried from the database.

As mentioned in chapter 3, the algorithm may be improved upon by using Actor-Critic methods (particularly A3C) and faster learning progress may be achieved by enriching the state representation using a recurrent neural network. Both of these may be valuable theoretical extensions of the work presented.

Bibliography

[1] World Wide Web Size. http://www.worldwidewebsize.com/. [Cited on page 1.]

[2] Richard S Sutton, Barto, and Andrew G. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition, 2016. [Cited on pages 1, 3, 5, 21, and 56.]

[3] Aja Huang David Silver, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 2016. [Cited on pages 1, 10, 21, and 57.]

[4] D Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012. [Cited on pages 4 and 10.]

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Alex Graves Marc G. Bellemare, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Dharshan Kumaranand Helen King, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015. [Cited on pages 8 and 18.]

[6] Michiel van der Ree and Marco Wiering. Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play. *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2013 IEEE Symposium on*, 2013. [Cited on page 9.]

[7] Murray Campbell, Joseph Hoane Jr, and Hsu Feng-hsiung. Deep Blue. *Artificial Intelligence*, 134, 2002. [Cited on page 9.]

[8] David Silver. Lecture notes on Reinforcement Learning, University College London, 2017. [Cited on page 9.]

[9] Yuxi Li. Deep Reinforcement Learning: An Overview. 2017. [Cited on pages 9, 16, and 21.]

[10] John N Tsitsiklis and Benjamin Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. *IEEE Transactions On Automatic Control*, 42, 1997. [Cited on page 9.]

[11] Joseph Modayil. Lecture notes on Reinforcement Learning, University College London, 2017. [Cited on page 9.]

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. [Cited on pages 10, 11, 14, and 15.]

[13] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. [Cited on page 10.]

[14] Simon Osindero. Lecture notes on Deep Learning, University College London, 2017. [Cited on page 10.]

[15] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989. [Cited on page 10.]

[16] The Economist. From not working to neural networking. *The Economist*, 2016. [Cited on page 10.]

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521, 2015. [Cited on page 10.]

[18] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010. [Cited on page 10.]

[19] Stanford CS231n Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/convolutional-networks/. [Cited on pages 11, 14, and 15.]

[20] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back propagating errors. *Nature*, 323:533–536, 10 1986. [Cited on pages 11 and 13.]

[21] James Martens. Lecture notes on Deep Learning, University College London, 2017. [Cited on page 12.]

[22] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. [Cited on page 13.]

[23] Geoffrey Hinton. Coursera: Neural Networks for Machine Learning Lecture Notes, University of Toronto, 2012. [Cited on page 13.]

[24] Martin Riedmiller and Heinrich Braun. RPROP - A Fast Adaptive Learning Algorithm. *Proc. of the Int. Symposium on Computer and Information Science*, VII., 1992. [Cited on page 13.]

[25] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014. [Cited on page 13.]

[26] Christopher Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995. [Cited on page 13.]

[27] Tensorflow. https://www.tensorflow.org/. [Cited on pages 16 and 31.]

[28] Szymon Sidor and John Schulman. OpenAI Baselines: DQN. *OpenAI*, 2017. [Cited on pages 16 and 20.]

[29] V Mnih, K Kavukcuoglu, D Silver, A Graves, I Antonoglou, D Wierstra, and M Riedmiller. Playing Atari with Deep Reinforcement Learning. *NIPS 2013*, 2013. [Cited on pages 16 and 17.]

[30] John Schulman. The Nuts and Bolts of Deep RL Research. *OpenAI*, 2016. [Cited on pages 16, 20, and 27.]

[31] Long-Ji Lin. Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching. *Machine Learning*, 1992. [Cited on page 17.]

[32] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *The International Conference on Learning Representations (ICLR).*, 2016. [Cited on pages 17, 18, and 31.]

[33] Volodymyr Mnih. Lecture notes on Practical Deep RL, University College London, 2017. [Cited on pages 18 and 19.]

[34] Hado van Hasselt. Double Q-learning. *NIPS 2010*, 2010. [Cited on page 18.]

[35] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. [Cited on page 19.]

[36] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR. [Cited on pages 19, 39, and 42.]

[37] Tom Zahavy, Nir Baram, and Shie Mannor. Graying the black box: Understanding DQNs. *ICML*, 2016. [Cited on page 20.]

[38] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 2008. [Cited on page 20.]

[39] Jason Rennie and Andrew Kachites McCallum. Using Reinforcement Learning to Spider the Web Efficiently. *ICML '99 Proceedings of the Sixteenth International Conference on Machine Learning*, 1999. [Cited on pages 21, 22, and 48.]

[40] Alexandros Grigoriadis and Georgios Paliouras. Focused Crawling using Temporal Difference-Learning. *Methods and Applications of Artificial Intelligence. SETN 2004. Lecture Notes in Computer Science, vol 3025. Springer, Berlin, Heidelberg*, 2004. [Cited on pages 22, 26, 27, 29, and 48.]

[41] Konstantinos Stamatakis, Vangelis Karkaletsis, Georgios Paliouras, James Horlock, Claire Grover, James R Curran, and Shipra Dingare. Domain-Specific Web Site Identification: The CROSSMARC Focused Web Crawler. *Proceedings of the Second International Workshop on Web Document Analysis*, 2003. [Cited on page 22.]

[42] Lu Jiang, Zhaohui Wu, Qian Feng, Jun Liu, and Qinghua Zheng. Efficient Deep Web Crawling Using Reinforcement Learning. *Lecture Notes in Computer Science, page 428-439. Springer*, 6118, 2010. [Cited on page 22.]

[43] Ali Mohammad Zareh Bidoki, Nasser Yazdani, and Pedram Ghodsnia. FICA: A novel intelligent crawling algorithm based on reinforcement learning. *Web Intelligence and Agent Systems*, 7, 2009. [Cited on pages 22, 29, 31, and 48.]

[44] Karthik Narasimhan CSAIL, Adam Yala, and Regina Barzilay. Improving Information Extraction by Acquiring External Evidence with Reinforcement Learning. *CoRR*, abs/1603.07954, 2016. [Cited on pages 23 and 35.]

[45] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems 26*. [Cited on pages 23, 24, and 35.]

[46] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. [Cited on page 23.]

[47] Yoon Kim. Convolutional Neural Networks for Sentence Classification. *EMNLP 2014*. [Cited on pages 24, 25, and 35.]

[48] Denny Britz. Implementing a CNN for Text Classification in TensorFlow. http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/. [Cited on pages 25 and 35.]

[49] Common Crawl - Data. https://commoncrawl.org/the-data/. [Cited on pages 26, 41, and 43.]

[50] Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/. [Cited on page 26.]

[51] Tejas D Kulkarni, Karthik R Narasimhan, Ardavan Saeedi CSAIL, and Joshua B Tenenbaum BCS. Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation. *CoRR*, abs/1604.06057, 2016. [Cited on page 28.]

[52] Alexander Vezhnevets, Volodymyr Mnih, John Agapiou, Simon Osindero, Alex Graves, Oriol Vinyals, and Koray Kavukcuoglu. Strategic attentive writer for learning macro-actions. *CoRR*, abs/1606.04695, 2016. [Cited on page 28.]

[53] PyPi WordSegment Module. https://pypi.python.org/pypi/wordsegment. [Cited on page 30.]

[54] Arthur Juliani. Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C). [Cited on page 39.]

[55] Richard S Sutton, Barto, and Andrew G. *Reinforcement Learning: An Introduction, Chapter 7*. The MIT Press, 1st edition, 1998. [Cited on page 56.]

[56] Hal Id. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. 2006. [Cited on page 57.]

Appendix A

A.1 Eligibility Traces for Online TD(λ)

At each step of the algorithm, the value function for *each* state is updated according to the TD error in proportion to its eligibility trace, $E_t(s, a)$.

$$E_t(s, a) = \gamma\lambda E_{t-1}(s, a) + \mathbb{1}(S_t = s, A_t = a) \tag{A.1}$$

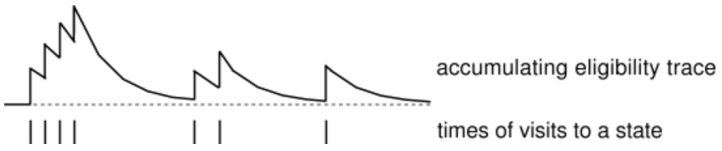

Figure A.1: Accumulating eligibility trace [55].

Depending on the chosen value of λ, more recently visited states will have more significant updates. An example update for the action-value function is shown below:

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t) \\ q(s, a) &\leftarrow q(s, a) + \alpha\delta_t E_t(s, a). \end{aligned} \tag{A.2}$$

A.2 Tree-based Methods

A model of the environment allows an agent to predict the results of taking an action from a given state, with a perfect model corresponding to knowing the transition and reward dynamics of the MDP exactly. Consider the case where we have a model of the environment, this may either be a perfect model or one that has been learned from experience (see chapter 8 of Sutton and Barto [2] for how this may be done). In contrast to the model-free algorithms (such as Q-learning and Sarsa) discussed

in chapter 2, in this section we discuss model-based approaches to reinforcement learning, specifically within a tree-based framework.

A.2.1 Forward Search

Starting at a specific state s in the environment, with forward search we look ahead at the *sub-MDP* from the current state in order to choose the best action, by doing a full-width backup to the current state. This corresponds to solving the sub-MDP exactly.

A.2.2 Simulation Based Search

Clearly the forward search approach is computationally intractable in many cases. Simulation-based search methods instead use the model to sample experience from the current state in order to estimate value or action-value functions.

In simple Monte-Carlo search, for a current state s, we sample a number of episodes, K, after taking each of the possible actions A from this state and then evaluate the actions by their mean return in order to estimate the action-value function (where $\hat{Q}$ is used to denote an estimate of the true value function).

$$\hat{Q}(s,a) = \frac{1}{K}\sum_{k=1}^{K} G_t^k \tag{A.3}$$

We then choose the best action according to a chosen policy (e.g. greedy or ϵ-greedy) and then repeat from the next state. Learning to play a board game is a natural way to think about this approach. If the current state of the board is represented by s, we can use a model of the environment to sample a series of episodes from this current state and evaluate actions by the proportion of the sampled episodes that were won by the agent, using these results to update the action-value function and learn a better policy as a result. A relatively recent extension of this method, Monte-Carlo Tree Search [56], was used in the well-known *AlphaGo* algorithm [3].

An equivalent temporal difference approach would run a fixed series of steps from the current state (for each action) and then use the current estimate of the value function to *boostrap* from a resulting state deeper in the tree. Concretely, from a starting state, we take a fixed number of steps to reach a new state, and use the value function at this new state to help learn an estimate of the value function of the starting state.

Appendix B

B.1 Distribution of Number of Outgoing Links

Table B.1 and figure B.1 show the distribution in the web graph from chapter 3.

Min	1st Quartile	Median	Mean	3rd Quartile	Max
0.0	6.0	22.0	53.9	67.0	3455.0

Table B.1: Summary statistics for number of outgoing links in the web graph.

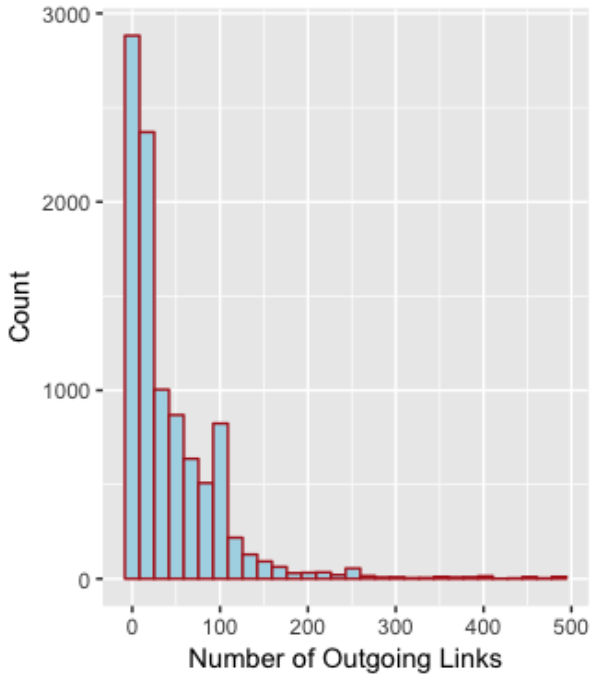

Figure B.1: Distribution of the number of outgoing links in the web graph. The tail of the distribution is removed to aid visualization.

B.2 Computation Graphs for Text Classification CNN

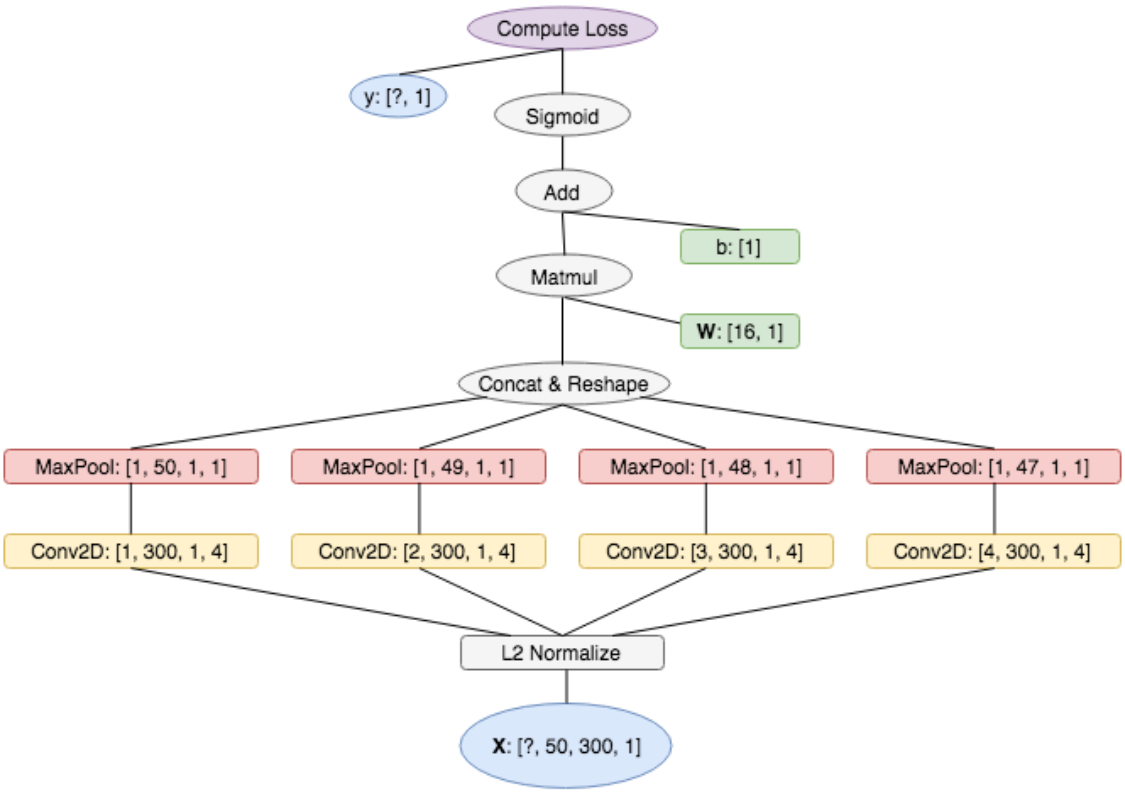

Figure B.2: Computation Graph, with dimensions shown. The input is the word embedding representation of the (padded) URL strings, where 300 is the embedding size.

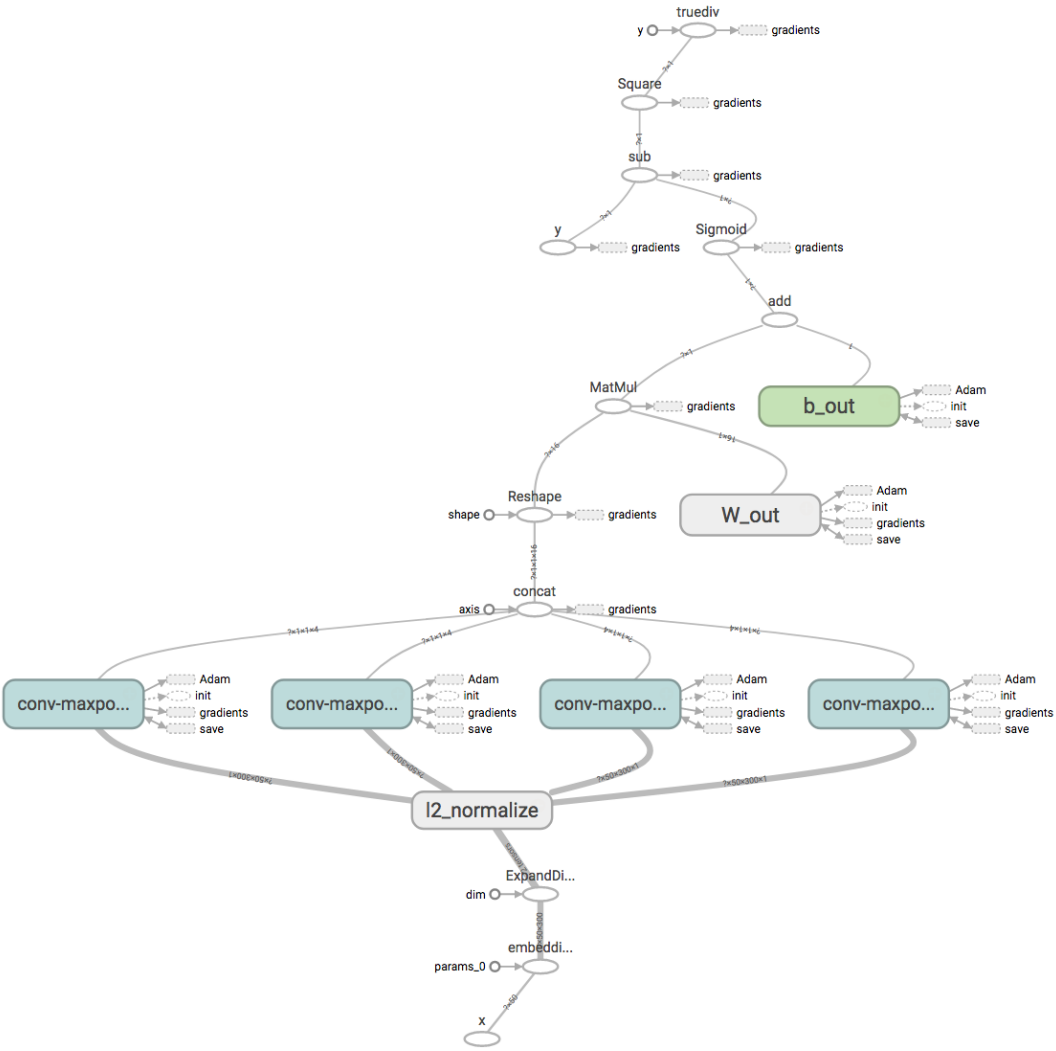

Figure B.3: Tensorflow Computation graph.

Appendix C

C.1 Distribution of Number of Outgoing Links in UK Web Graph

Table C.1 and figure C.1 show the distribution of number of outgoing links in the UK web graph from chapter 4.

Min	1st Quartile	Median	Mean	3rd Quartile	Max
0.0	6.0	90.0	225.5	228.0	2982.0

Table C.1: Summary statistics for number of outgoing links in the web graph.

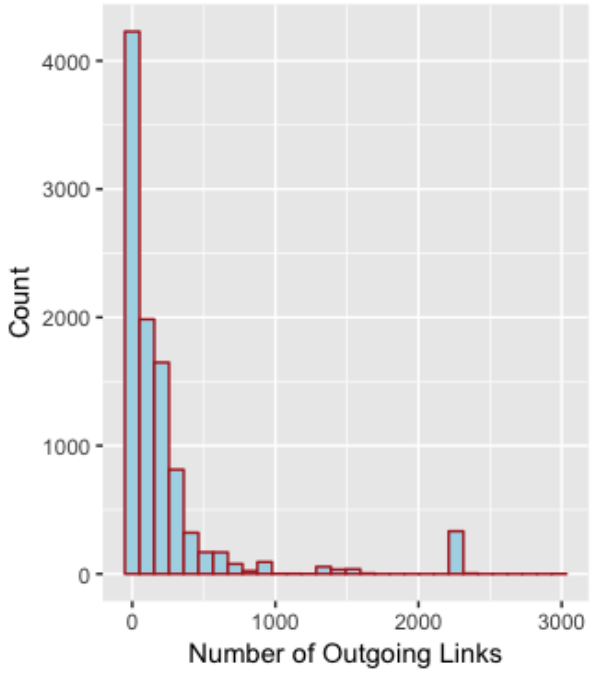

Figure C.1: Distribution of the number of outgoing links in the web graph.