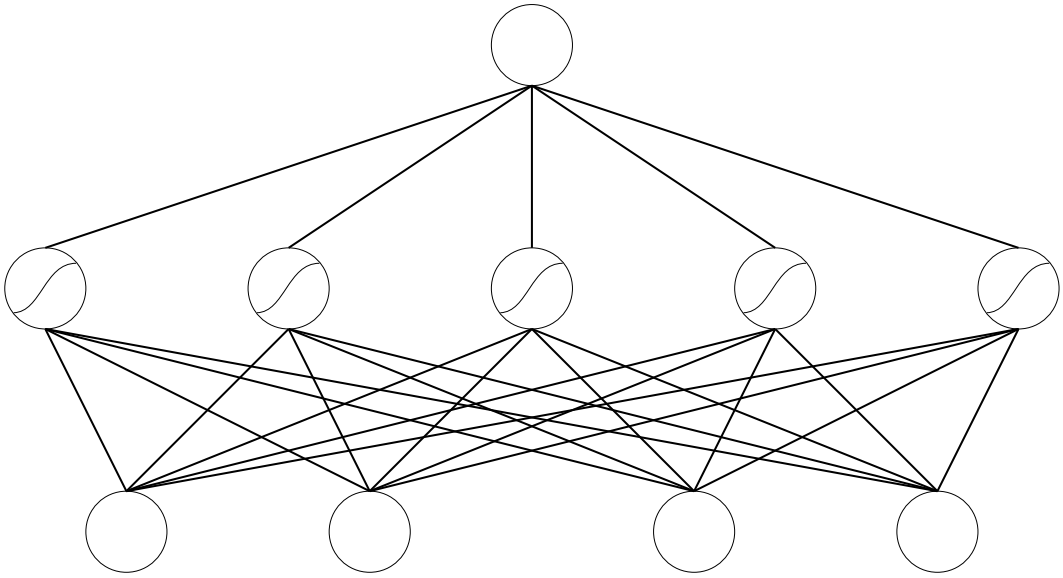
A Neural Network Algorithm for Internetwork Routing



Will Newton

u8wrn@dcs.shef.ac.uk

Supervisor: Dr. Si Wu

This report is submitted in partial fulfilment of the requirement for the degree of Bachelor of Engineering with Honours in Software Engineering by Will Newton.

All sentences or passages quoted in this dissertation from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this dissertation have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

Name: Will Newton

Signature:

Date: May 5, 2002

Abstract

Efficient routing of packets in computer networks is a prerequisite for wide deployment of network-enabled devices, especially mobile and distributed computing. Current approaches to this problem have been partially successful, but are liable to breakdown when under heavy load. Boyan and Littman presented the Q-routing algorithm, a network routing algorithm based on Q-learning, a method from the emerging field of reinforcement learning. Q-routing learns to route packets in an adaptive manner allowing slow spots in the network to be routed around, and has performed well in simulation. Neural networks have been used with some success to perform Q-learning, and would seem to be a possible method to allow Q-routing to scale well beyond it's initial table-based implementation. The results achieved are disappointing, but some insight is gained into the difficulties of using neural networks as a function approximator in reinforcement learning tasks.

Acknowledgements

I would like to thank the following people who have helped me with this project:

- Tom Newton, my brother, for proof-reading my work and support.
- Dr. Si Wu, my supervisor, for suggestions and advice.
- Gary Flake, author of NODElib, for his timely help and suggestions.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Rise of Pervasive Networking

Computer networks play an important and ever increasing role in the modern world. The development of the Internet, the corporate intranet, and mobile telephony have extended the reach of network connectivity to places that ten years ago would have been unthinkable. The result of these trends is that the performance of network hardware and software is being tested by the increasing load placed upon them, and new ways are having to be found to solve the problems that the original ARPANET engineers faced nearly 50 years ago.

Modern networking applications are often based upon frequently changing ad hoc network topologies and require that the network protocols running these applications are able to withstand outages of parts of the network. The emphasis on peer-to-peer technologies of such applications as Napster © and Gnutella demonstrates the desire to move away from centralised network architectures towards more loosely defined, distributed ones.

In this paper a routing algorithm that attempts to solve some of the problems faced in dynamic, unreliable, and congested networks is suggested. Such an algorithm aims to have the following properties:

- It should be able to “bootstrap” itself without any need for operator intervention.
- It should be able to “route around” failed links or nodes.

- It should be able to find optimal or close to optimal routes even when the network is heavily congested.

1.2 Neural Networks and Function Approximation

Neural networks are a biologically inspired approach to machine learning that can learn to approximate complex mathematical functions. Neural networks have been used in proposed solutions to all sorts of problems, from financial forecasting to automated control systems, and have achieved some very significant successes in many of these areas. Neural networks are a very active area of research. Whilst it is obvious that no one technology can be a solution for every problem, neural networks have proved to be extraordinarily versatile and effective.

This report attempts to apply a neural network as a function approximator in an online reinforcement learning task, a field where neural networks have been used with varying degrees of success in the past. A discussion of the factors involved in neural network function approximation in reinforcement learning is provided.

Chapter 2

Literature Review

2.1 Machine Learning

2.1.1 Reinforcement Learning

Reinforcement learning is a relatively new and emerging area of machine learning theory. Reinforcement learning aims to develop successful techniques for learning complex strategies from limited data in a goal-directed manner. The definition of reinforcement learning given by Sutton and Barto is "Reinforcement learning is defined not by characterising learning methods, but by characterising a learning problem" [31].

A reinforcement learning problem is a problem where supervised learning cannot easily be used because there are no training sets, insufficient data or external knowledge that can be applied, a reward signal. Maximising this reward signal is the goal of reinforcement learning. Reinforcement learning algorithms develop a *policy*, usually defined as a mapping of states and subsequent actions to expected reward.

The field is still quite young, and there are problems that are still being researched which are proving difficult to solve. For example, the problem of temporal credit assignment, when given a reward determining which set of long or short term actions were responsible for the reward, is a very difficult one. In some cases it will obviously be better to take a lower short term reward for a greater long term reward. However, the reinforcement learning approach is very promising even in

these early stages, and seems a good fit for the network routing problem as Boyan and Littman [7] showed.

Temporal difference (TD) learning, is a major part of reinforcement learning theory, and covers a number of methods, such as $TD(0)$, $TD(\lambda)$, Q-learning and Sarsa [31]. What these methods have in common is the concept of accumulating the effect of differences in reward over time, hence "temporal difference".

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.1}$$

The equation in 2.1, sometimes referred to as the "Q-function" for quite obvious reasons, is the policy update rule used in Q-learning. Confusingly, it is also the same function used by the Sarsa algorithm, although the two techniques have differences in other areas. The update rule states how the value of the Q-function is updated for each state s_t, and for every action a_t that can be taken from that state. r_{t+1} is the immediate reward from this action, and $\gamma Q(s_{t+1}, a_{t+1})$ is the discounted future reward from this state. The value γ, known as the *discounting rate* determines whether the algorithm is greedy in the short term or longer term.

A greedy policy is a policy that always takes the action that is estimated to have the highest reward. This is sometimes the best course of action, for example in a static environment. An online algorithm must keep trying other actions to ensure that the its reward estimates are accurate. To do this a probability, known as epsilon (ϵ), is given that in any state there is a probability of epsilon that a random action will be taken. A policy like this is known as ϵ-greedy.

Q-learning is an off-policy reinforcement learning method, which means it learns a policy, called an estimation policy, that is separate from the policy which it acts upon, the behaviour policy. This allows, for example, an estimation policy that is greedy while the behaviour policy may be able to sample other actions, e.g. epsilon-greedy. An algorithm closely related to Q-learning, but working in an on-policy manner, is Sarsa. Sarsa is close enough to Q-learning to be able to be substituted directly in some cases.

2.1.2 Neural Networks

Neural networks are a technique for pattern recognition and function approximation based originally on ideas from biology and the study of neurons [17],

although the theory behind neural networks is based on statistical foundations [6]. Neural networks seek to mimic the apparently simple, but extremely effective, structure of the human brain to able to approximate functions that would be far too complex to program by hand. Neural networks consist of units, corresponding to neurons, and weighted connections between them. The values of each unit are "fed forward" via the weighted connections to other units. The artificial neurons have an output value based on a specific "activation function" of which they are many, some more biologically realistic, and others better suited for other tasks. The most common are linear units and sigmoid units. Neural networks may also be made up of building blocks of smaller networks, for example perceptrons. When two layers of units are combined, such a neural network can approximate any continuous function. When three layers of units are combined any arbitrary function that may be required can be approximated [24].

Learning in neural networks is commonly achieved using the backpropagation [6] algorithm. Backpropagation takes the overall error of the output of a neural network, as measured against training data, and propagates it back through the network, calculating the error of each individual weight. The weights of the network can then be updated according to a weight update rule. Backpropagation has proved successful at solving some quite complex problems [24], including complex control tasks such as engine control, and speech and image recognition. The typical use of backpropagation is in multiple iterations over static data sets, but it may be used in online applications as a means of calculating error gradients.

Neural networks have been used before in network routing type problems, for example, using radial basis functions to optimise call set-up in a telephone network with a static topology [36]. This situation could be seen as analogous to a computer network, but the two situations are quite different. A telephone network has a relatively static topology, and quite predictable usage patterns. A computer data network often has a changeable, ad hoc topology, greatly varying usage patterns and a lack of control over bandwidth allocation. Because of this, a controller trained on prior data cannot be guaranteed to perform well under the conditions, and must be rebuilt when the topology is altered.

2.1.3 Neural Networks and Reinforcement Learning

Neural networks have often been used in reinforcement learning. The use of neural networks in this situation is sometimes called "neuro-dynamic programming" because of the use of neural network and dynamic programming techniques together. There have been notable successes with these methods, such as TDGammon, a reinforcement learning approach to playing backgammon. TDGammon uses temporal difference learning and a neural network which approximates the value function of the various board states in a game of backgammon. The current version of TDGammon, 3.0, is on a par with the best human players in the world today and easily the best computer backgammon player [31, 33].

Neural networks and the Sarsa algorithm were also used to good effect by Bazen et. al. to extract minutiæ information from fingerprint data [4]. Success with neural network function approximation and temporal difference learning was also shown with a control problem similar to network routing by Crites and Barto. Crites and Barto used a neural network function approximator in conjunction with temporal difference learning to control a system of elevators (lifts) and which was able to outperform all known heuristic elevator control algorithms [12].

Unfortunately the Q-learning algorithm is known to be unstable when used in conjunction with a linear function approximator [5, 16]. Baird demonstrated an example that clearly shows this with the "star problem" a six-state Markov decision process (MDP) that has an exact solution, but when using a linear function approximator is guaranteed to fail to converge successfully [5]. This means there is no guarantee that a linear approximator will be able to safely approximate the Q-function, and if convergence can not be guaranteed for the weaker condition of a linear approximator, then a non-linear approximator is also not guaranteed. A neural network therefore cannot guarantee to safely approximate the Q-function, although in some cases it may. Tsitsiklis and Van Roy showed that all Temporal Difference algorithms are liable to become unstable when used with a non-linear function approximator, such as a neural network [34].

The solution Baird used was a technique known as *Residual Algorithms*, which is essentially a gradient descent approach as opposed to a simple value maximisation algorithm [5]. Residual algorithms tend to be quite slow to learn and are really more suited to episodic tasks rather than infinite horizon tasks.

Eligibility traces, sometimes termed $TD(\lambda)$ methods, are another technique

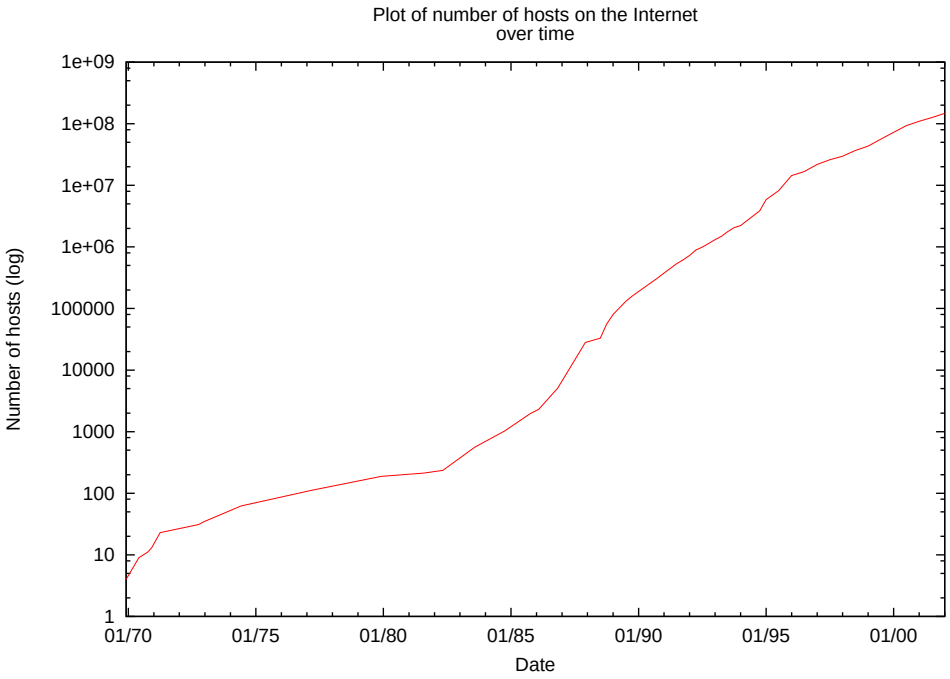
that can make function approximation more stable. Eligibility traces try to bridge the gap between temporal difference methods and Monte Carlo methods [31]. Discussion of Monte Carlo methods and dynamic programming are deemed outside the scope of this project due to the huge quantity of unnecessary detail that would require. Eligibility methods basically amount to another step of "back up" in the Q-function, so for example the last two known rewards are used plus the discounted sum of the remaining rewards, rather than the last known reward plus the discounted sum.

2.2 Computer Networks

2.2.1 The Internet

The Internet, with a capital "I", as opposed to the more general term internet or internetwork, is a global network of computer systems that originated in the United States as part of the military funded ARPANET which began in the early 1970s [10]. The Internet has pushed networking technology into the mainstream and it is without doubt the most important network, both in terms of technology advances and social impact, in the world. The number of hosts on the Internet is growing at an incredible rate and shows no sign of slowing down [21]. This growth, illustrated in Figure 2.1, has placed strain on the network infrastructure that was built on what were, at the time ARPANET was created, experimental technologies.

The Internet uses packet switching technology to transmit data, that is, data that is to be transmitted over the Internet is split into small chunks, known as packets. These packets are then transmitted one at a time across the Internet where they are reassembled at the receiver. Packets on the Internet may not necessarily arrive in order, or at all, and may follow different routes to their destinations. Packet switching allows the Internet to be flexible in what physical media it can use to transmit data, as well as to be fairly resilient when faced with small amounts of data loss caused by line noise or other factors. The basic building blocks of the Internet are the TCP/IP suite of protocols [27], which may be modelled as a stack of protocols split into several layers [32]. The underlying protocol at the network layer, Internet Protocol or IP is a connection-less best effort protocol, meaning it



Data from Hobbes Internet Timeline:
http://www.zakon.org/robert/internet/timeline/

Figure 2.1: Number of hosts on the Internet since 1970

has no connection establishment phase or authentication, and it does not provide a guarantee that the data sent will reach it's destination [27]. Reliable delivery is provided by the Transmission Control Protocol, or TCP.

However, the properties that make the Internet so effective and successful also make it vulnerable to "Internet Meltdown" or "congestion collapse" [8]. Several aspects of the underlying Internet technology are showing their age and reaching the point where other approaches may soon have to be explored if the growth rate and stability of the Internet is to be maintained. These areas include address allocation [27], routing and congestion [8].

2.2.2 Network Congestion

Network congestion occurs when a part of a computer network is asked to transfer more data than it's resources allow in a period of time. This results in data loss and a degradation in performance. The problem of congestion occurs when the bandwidth of a specific link is exhausted by an upsurge in traffic, if packets cannot be sent down a congested link then the machine sending packets onto that link has to wait for the link to become free before sending any more. This means the sender has to buffer the packets it still has to send until the link is free, which requires that the sender have memory available to buffer the packets. If the backlog of packets gets too large for the sender to buffer, then any additional packets must be discarded or dropped. In reliable network protocols the response to a dropped packet is to resend the packet in question, and because of this, the effect of a dropped packet is multiplied. For every packet that is dropped from a reliable protocol it will be sent again at least once, adding to the congestion and therefore more packets will be dropped, and so on [27, 32].

Network congestion is increasingly becoming a problem on the Internet. In the mid-1980s the Internet was beginning to suffer from congestion problems and the possibility of congestion collapse [26]. TCP congestion control measures [8] such as Tahoe and Reno extensions have attempted to alleviate this problem, and TCP performance under load is much better than it was when the Internet cut over to TCP/IP on the 1st January 1983. The continuing growth of the Internet and the proliferation of high-bandwidth applications means this issue is far from resolved however.

2.2.3 Congestion Control

Much progress has been made in the field of congestion control in recent years, and problems such as congestion collapse [26] have largely been avoided [29] in the context of TCP, the most widely used transport protocol on the Internet. Congestion problems are by no means confined to TCP based applications however, and the area is still being very actively researched.

Congestion control can be approached at every level of the network protocol stack. At the data link layer, the hardware will do its best to avoid congestion, for example Ethernet will do exponential backoff in an attempt to avoid packet collisions on an Ethernet segment [27]. Congestion control at this level is usually very simple because it is usually implemented in hardware. Ethernet is simple enough to be implemented in very small integrated packages and Ethernet hardware is accordingly very cheap.

At the network layer the queueing policy and routing algorithm come into play along with the possibility of a virtual circuit switching technology. Virtual circuit switching technology will not be discussed here for space reasons. Active queue management is an active research topic, and algorithms such as Random Early Detection, or RED, [3] and BLUE [9] have proved very successful in simulation, although practical use of these technologies is still quite limited [23]. Active Queue Management seeks to drop packets not simply based on the order they are received at a router. For example, RED drops packets randomly with a specific probability before congestion becomes a problem, in an attempt to prevent congestion occurring. These algorithms have also been extended to implement fair queueing, an attempt to make sure every traffic flow gets access to network resources fairly, and differential services, which allows different types of network packet to be given different priorities in network resource allocation. There have also been proposed algorithms that use cost metrics to price the routing of a packet through a network, so a user willing to pay more could route packets over a faster or more reliable network. Routing algorithms are discussed in the next section.

Active queue management may also be applied at the transport layer, for example the Explicit Congestion Notification, ECN [19] extension to TCP. This adds a single bit to TCP packets so congestion can be signalled explicitly rather than implicitly by dropping packets at the network layer. This approach works best with connection based protocols because rate control is more easily applied to a

connection than single packets, rate control for TCP is performed by altering the transmission window size of the sender [27]. The other main Internet transport layer protocol, the User Datagram Protocol or UDP [27] must be treated separately by an active queue management algorithm, because it is best effort and connection-less much like the underlying IP. UDP tends to be bandwidth greedy and can be a major factor in network performance degradation. An ECN bit has also been proposed for IP, but it is a more recent proposal and so not many TCP/IP stacks implement it today.

The setting of timeout values is also very important because it impacts directly on how much excess traffic is put onto the network when congestion occurs. Timeout values determine how long a packet may go unacknowledged by the receiver, after a certain length of time the sender will assume the packet is lost and resend it. Getting the timeout values perfect is very difficult and depends to a large extent on the characteristics of the network involved. TCP has received a lot of attention, and there are many extensions proposed that may improve it further, including Selective Acknowledgements, or SACK, which sends less acknowledgement packets and Fast Retransmit, which attempts to guess when a packet has been lost and retransmit it before it timeouts [15].

Applications may also attempt their own congestion control. If an application uses TCP it is usually counter-productive to try and implement another layer of congestion control at the application level unless the application author has some extremely domain-specific knowledge. UDP applications, in particular streaming applications like Internet radio and video conferencing, can benefit enormously from being able to handle their own congestion control due to their greedy nature. However, it is a very difficult problem to calculate the bandwidth available across a wide variety of links and routers between two hosts. A lot of the research in this area seems to be proprietary and quite experimental.

2.2.4 Routing Algorithms

Routing algorithms are methods for finding the best way to get from a host A to another host B. This may be via a large number of other machines or it may be in the next room. On a small, simple network the problem is almost trivial, statically allocating routes and defining them by hand, but when dealing with a

huge internetwork such as the Internet this is not possible. A heavily interconnected network such as the Internet has many routes from one host to another, and these routes span many different types of link with different bandwidth and latency characteristics. Calculating the best route through such a complex system is computationally intractable and impossible to do by hand. A routing algorithm first seeks to deliver a packet successfully, and if possible deliver it by the quickest route available. If many packets are routed through the same router a bottleneck occurs, and the whole network slows down, a good routing algorithm will try to route around a bottleneck router to minimise the effect of network congestion.

There are two types of routing algorithms, inter-domain routing algorithms and intra-domain routing algorithms. Inter-domain routing algorithms route packets between domains, that is, they find routes from one large area of the Internet, for example JANET, to another large area such as LINX. These large areas are called domains or autonomous areas. Inter-domain routing algorithms include the Exterior Gateway Protocol or EGP which is now considered obsolete, and the Border Gateway Protocol or BGP [28] which the modern Internet backbone runs on. There are many problems in the field of inter-domain routing, the sheer size of the routing tables for the Internet today is close to overwhelming some of the backbone routers and some have raised questions over whether BGP is a stable enough protocol for running such a large and complex network [11].

Intra-domain routing algorithms find routes within autonomous areas. An autonomous area could be as small as 2 or 3 machines or as large as tens of thousands on a corporate network. Intra-domain routing algorithms include Routing Information Protocol or RIP, version 1 and 2 [22], and Open Shortest Path First or OSPF [25].

RIP is a distance vector routing protocol, and very simple to implement. Distance vector routing, sometimes known as Bellman-Ford, after it's inventors, is based on the idea of each router in a network keeping a vector of distances to every other node in the network. This vector is then distributed to its nearest neighbours, which update their distance vector on the basis of the information they are sent by their neighbours. In this way routing data propagates throughout the network and eventually converges to a stable routing policy.

RIP and distance vector routing suffer from some quite serious drawbacks unfortunately. Under certain circumstances the routing tables for a network will

not stabilise, a problem often called the "count to infinity" problem [27]. Various partial solutions have been proposed to this but none are particularly elegant. RIP also has scalability problems, it can only handle networks with a maximum of 16 hops, and it generates a considerable amount of network traffic in larger networks. In 1979 RIP was retired from ARPANET.

OSPF is a link-state routing algorithm. Link-state routing attempts to build up a graph representing the network at each router and uses a shortest path algorithm, usually Dijkstra's algorithm, to find the shortest path to a route according to the graph it has built up. The graph representation is built up by routers exchanging link-state advertisements, or LSAs. These packets contain link-state information, that is, what links a router has, and to what machines. OSPF adds several advanced features that RIP didn't have, such as message authentication, load balancing and a hierarchical structure [25]. OSPF also uses metrics to calculate the cost of taking a route, rather than the RIP approach which is merely based on hop counts, i.e. the number of routers on the route. This allows, for example, a 128Kbps link to be preferred over a 9.6Kbps link. The metric used is a very important factor as to the optimality of the routes chosen, and ideally should be able to take into account how congested a link is.

There are several other routing algorithms that have been proposed. Many of these are applicable only to specific routing problems such as Massively Parallel Processor, or MPP networks [18] or virtual circuit switched networks, and do not work well in a complex, dynamic environment such as a packet switched computer network.

2.2.5 Q-routing

Of more interest is Q-routing, a routing algorithm that uses a technique known as Q-learning from the field of reinforcement learning to learn the best routes and constantly update them online [7]. Boyan and Littman present very impressive results in their paper. The algorithm is dynamic and suited to networks with changing topologies and traffic levels, it aims to be as stable as possible under high loads whilst still performing well in less extreme situations. The Q-routing algorithm has been developed further by others [20], but the extensions proposed are rather incidental to the underlying algorithm.

Chapter 3

Design

3.1 Simulation Environment

3.1.1 Simulator

The simulation environment was built on OMNET++, a discrete event simulator written in C++ [35]. OMNET++ uses a mixture of C++ and it's own custom Network Description (NED) language to specify network components. It is quite a general package and can simulate any type of discrete event system, so it does not have built-in support for testing network protocols specifically.

Using the a mixture of C++ and the NED language a number of modules were specified to implement various parts of the network. The relationships between these modules are shown in the UML class diagram Figure 3.1.

The `Sink` module simply receives packets and deallocates their storage, whilst keeping track of how many packets have been received, the amount of time they were on the network, and the distance they have travelled. The `Generator` module generates packets at a random interval within a bound. The `Generator` keeps track of how many packets have been sent and adds a time-stamp to each one. A custom loader was also written, the `TestNetwork` module, to load network topology from an XML based specification, this avoids the need to write extremely long hard-coded network definitions in the NED language and allows generation of networks with scripts more easily.

The network used for testing the algorithm was a 36 node 6x6 irregular grid

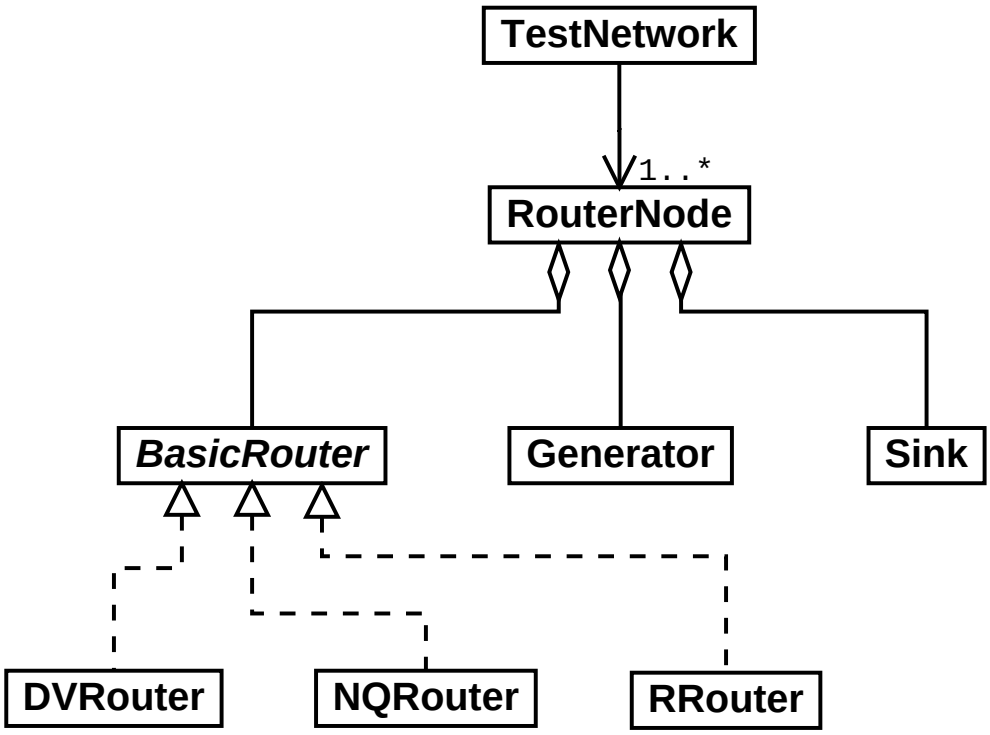


Figure 3.1: Class diagram of simulation environment

like the one used by Boyan and Littman in their paper. The layout of the network is shown in Figure 3.2. The network is complex enough to avoid accidental convergence and small enough to provide tractable simulation times. The neural network based algorithms took approximately ten times real time to calculate on my 700MHz AMD PC which is quite close to making it impractical to simulate many runs. The simple random router, for example, takes approximately 1/30 of real time.

The `omnetpp.ini` file is the initialisation file for the OMNET++ simulator, and allows several runs to be setup with different parameters for testing each type of router. The results are written to a file and can later be analysed with gnuplot and Octave.

3.1.2 Routers

In order to be able to accurately judge the performance of the routing algorithm, two other routers were written. These were the `DVRouter`, a router based on the Distance Vector algorithm [22], and the `RRouter`, a router that routes packets via a random output port. These two can be seen as examples of average practical performance and worst case performance respectively.

3.2 Neural Network Router

3.2.1 Background

The algorithm presented by Boyan and Littman [7] is a Q-learning type table-based algorithm, meaning the values computed at each step for the Q-function are stored in their method in a table. Their actual simulation implementation uses a statically allocated multi-dimensional C array for this purpose. The use of a table is the simplest method for representing the Q-function, but unfortunately tables are not a good way to represent state-spaces of high dimensionality, sometimes called the “curse of dimensionality” [6], and because of this it is attempted to replace the table in the Q-routing algorithm with a function approximator, namely a neural network.

A neural network Q-routing algorithm should aim to perform similarly to the

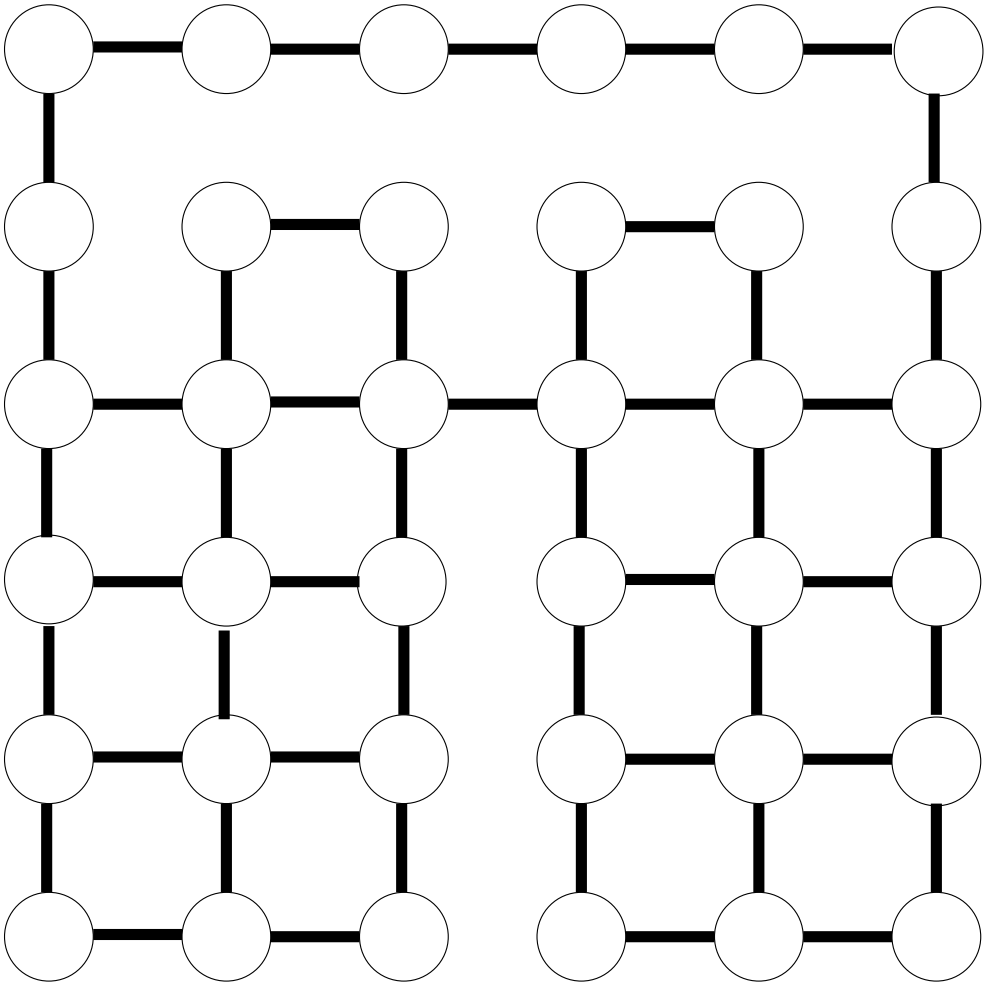


Figure 3.2: Network topology used in the simulation

table based version, but have much better scalability characteristics, instead of requiring $O(n)$ storage space on a network of n nodes at each node, a neural network of a fixed size is used.

At first the requirement of the Q-routing algorithm for accurate timings of each packet after every hop [7] may seem unrealistic. In practice this could be implemented using acknowledgement packets that reliable transport algorithms such as TCP require to return timing information.

3.2.2 Reinforcement Learning Algorithm

Both off-policy Q-learning and on-policy Sarsa variants of the neural Q-routing algorithm were implemented. It is largely unknown under which conditions the Sarsa algorithm can be guaranteed to converge, and as it is a temporal difference algorithm it has no guarantee for non-linear function approximators such as neural networks, but may however prove to be more successful than Q-learning which is known to be unstable. A single step eligibility trace [31] was also implemented in an effort to make the algorithm more stable.

The weight update rule for the Q-learning algorithm and the Sarsa algorithm is identical, the weights of the network were updated using Equation 3.1, which may be derived from Equation 2.1.

$$\Delta w = \eta[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]\nabla_w Q(s_t, a_t) \tag{3.1}$$

3.2.3 Neural Network

The architecture of a neural network has a large influence on the performance of the network during training and in use. It was chosen to use probably the most common neural network architecture, the multi-layer perceptron (MLP). In this case a three layer, fully connected architecture, with a hidden layer of sigmoidal units.

There are many choices that can be made when picking a neural network architecture. It is possible to use a radial basis function [31], and self organizing maps or Kohonen maps [13] have also been used for reinforcement learning. Due to a lack of familiarity with either of these technologies the multi-layer perceptron

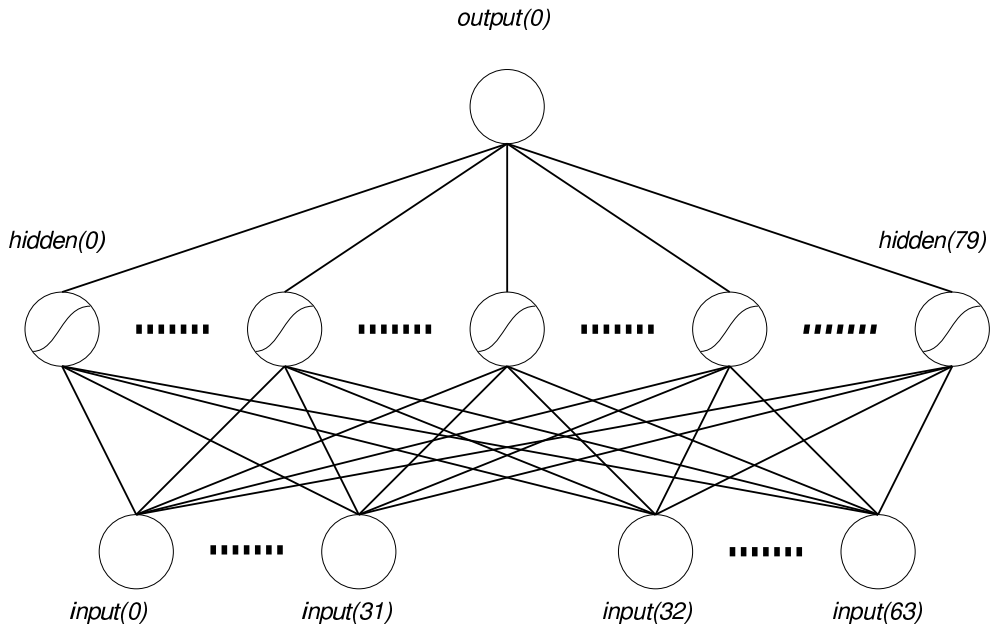


Figure 3.3: Neural network architecture

was chosen, which has already been proven successful in this field and is a relatively well understood technology [33, 12]. The activation function used may also affect the results obtained. The library used made it trivial to change the activation function used, and while experiments were made with other functions, such as *tanh*, it was decided to use the sigmoid function. This was due to the similar performance of the activation functions in tests and the lack of familiarity with some of the other functions.

The neural network implemented was based on the structure in Figure 3.2.3. It is comprised of 64 linear input units, 80 hidden sigmoid units and a single linear output unit. The 64 linear unit width of the input layer was chosen on the basis of the data that is to be processed, Discussion of how and what input is passed to the neural network is later. The 80 hidden unit width of the middle layer was chosen almost arbitrarily, 80 units is a considerable number, greater than the number of inputs, and errs on the side of expressiveness over speed, this is to minimise the chance of the neural network architecture being the restricting factor in the performance of the algorithm. The linear output unit gives an estimate of

for i = [0..32] **do**
  $input_i$ = bit i of neighbour address
  $input_{i+32}$ = bit i of destination address
end for

Figure 3.4: Algorithm for preprocessing input

the delay given the inputs, which is essentially an "inverse value" function, i.e. lower is better when using this function.

The 64 inputs to the neural network are the preprocessed values of the addresses of the neighbouring node via which a packet has been sent, and the final destination node of the packet. The addresses are numeric, and are comprised of 32 bit integers. In this respect the implementation requires an underlying 32 bit system architecture, but this is not a theoretic requirement of the routing algorithm. The addresses are split up into individual bits. The way this is done is using "bitwise shift" and "bitwise and" instructions to interpret each bit of an address as an input. The algorithm is presented in pseudo-code in Figure 3.4.

The aim of this algorithm is to extract the most fine grained information possible from the inputs. The inputs could have, for example, been treated as a pair of reals. The way a network address is constructed, two addresses that are numerically similar may be located on very different parts of the network, so the perceived similarity of the two addresses could make separating their different routing requirements difficult, and the simplicity of such a network would make it unlikely to be able to separate the inputs effectively. The inputs after preprocessing become either ones or zeros depending on the value of the address. The addresses that will be used in the experiments are all small numbers, between 1 and 100, but hopefully this scheme could be used to scale up to an almost fully populated address space, without preprocessing this would surely be impossible. The algorithm is also designed to be practical, logical operations like "shift" and "and" are very fast on hardware processors.

3.2.4 Drop Packets

A weakness of the Q-routing algorithm as presented in the paper by Boyan and Littman is that it never has to deal with dropped packets, which are commonplace in real world situations. This can cause routing loops as a packet can be routed indefinitely irrespective of whether it is getting any closer to its destination. During the bootstrapping phase of the routing algorithm this can be particularly problematic, as the network load may be increased greatly. To prevent routing loops "drop packets" were implemented which are essentially similar to ECN [15], sending a notification back to the sender of a packet when that packet is dropped. Packets are dropped when they have travelled a certain number of "hops", defaulting to 16 hops which is a reasonable value for a small network. This does add some complexity to the problem however, with adjusting the penalties for a dropped packet to be not unduly harsh or lenient.

3.2.5 NODElib

NODElib is a library of neural network functions for C and C++, written by Gary William Flake [1]. It supports many neural network and other machine learning features, including multi-layer perceptrons, radial basis functions and support vector machines. This library was chosen after looking at several others, including Torch [2], because it is well-tested and stable, and allowed easy manipulation of the neural network weights at a low-level. The MLP used in the experiments was built from the basic NODElib `NN` object.

Chapter 4

Results

4.1 Aims

The experiments performed aimed to give an accurate measurement of the performance of the neural network based algorithm. All tests were carried out on the network topology given in Figure 3.2 with identical packet transmission rate characteristics.

4.2 Parameters

There were several parameters that can be tuned to alter the performance of the neural Q-routing algorithm.

- *The learning rate* (α)
  The most successful value for the learning rate or α, was approximately 0.7. Lower learning rates led to no sense of convergence at all, whereas rates higher than 0.7 seemed to induce rapid divergence.

- *The likelihood of taking a non-greedy action* (ϵ)
  The value of epsilon had little effect on the simulation, as the algorithm was mostly diverging. Values of around 0.02, i.e. one in every fifty packets is sent by a randomly selected output port, seemed to be reasonable. Using a decreasing value of epsilon over the length of the simulation was also tried, to little effect on the result.

Parameters	*Behaviour*
Q, No Drop Packets	No convergence
Q, Drop Packets	No convergence
Q, Eligibility Traces	Divergence
Sarsa, No Drop Packets	Rapid divergence
Sarsa, Drop Packets	Rapid divergence

Table 4.1: Performance of the function approximator under various parameters

- *The penalty associated with dropped packets*
  The penalty associated with dropped packets was intended to be detrimental to a routes probability of being chosen, but not unduly so. The penalty has to be tailored to the network it is running on, for example, using a penalty of 2.0 on a network with an average round-trip time (RTT) or 0.01 is too harsh. On a network with an average RTT of 300.0 it is too lenient. Three times the average RTT for the network was the value used in the simulation.

- *The threshold within which the approximator is considered stable*
  Timing errors and small fluctuations in system load can make the RTT of a packet fluctuate slightly. To prevent constant retraining of the neural network when an acceptable route has been decided upon, an error bound is set so small fluctuations in the RTT are simply discarded. This value must also be tailored to the network the algorithm is used upon, and would ideally be expressed as a function of average RTT. One hundredth of an average RTT was the value used in the simulation.

4.3 Performance

The performance of the neural network function approximator under different parameters is shown in Table 4.1. The neural network did not converge in any of the circumstances, and diverged in some. The performance measurements were obtained by measuring the error of the function approximator at each node.

In these experiments, Sarsa performed seemingly worse than Q-learning. The error values at each node tended to diverge rapidly with Sarsa and within a rel-

atively short period of time exceeded the precision of the type used to measure it (48bit IEEE floating point). This does not prove that Q-learning is in any way "better" than Sarsa for this task, as the net result was still a failure to converge.

The use of drop packets did not seem to be an important factor in the failure of the algorithm, indeed it was difficult to tell if they had an overall positive or negative effect in the absence of convergence.

Eligibility traces were implemented in an effort to improve performance, but had surprisingly little effect on the result.

The performance of the routing algorithm was poor, even when the function approximator appeared at first to be close to convergence. For comparison, the results from the random router, `RRouter`, that routes packets in a random fashion, and the neural network (with a typical parameter setting) router can be seen in Figure 4.1 and Figure 4.2 respectively. Clearly the random routing algorithm is *more* successful in correctly routing packets than it's neural network counterpart, so even though the error values stay approximately stable, there is no actual convergence to a useful policy at all.

The simulation was slow to complete, taking approximately 10 times real time to simulate. However, this should not be a problem for implementing the algorithm in practice, as this implementation is totally without optimisation, and designed to be easily debugged and understood rather than high performance.

The neural network failed to converge in all situations, although in some it appeared to exhibit chaotic behaviour, as if fluctuating around a convergence fixed-point [14], but in others it diverged completely.

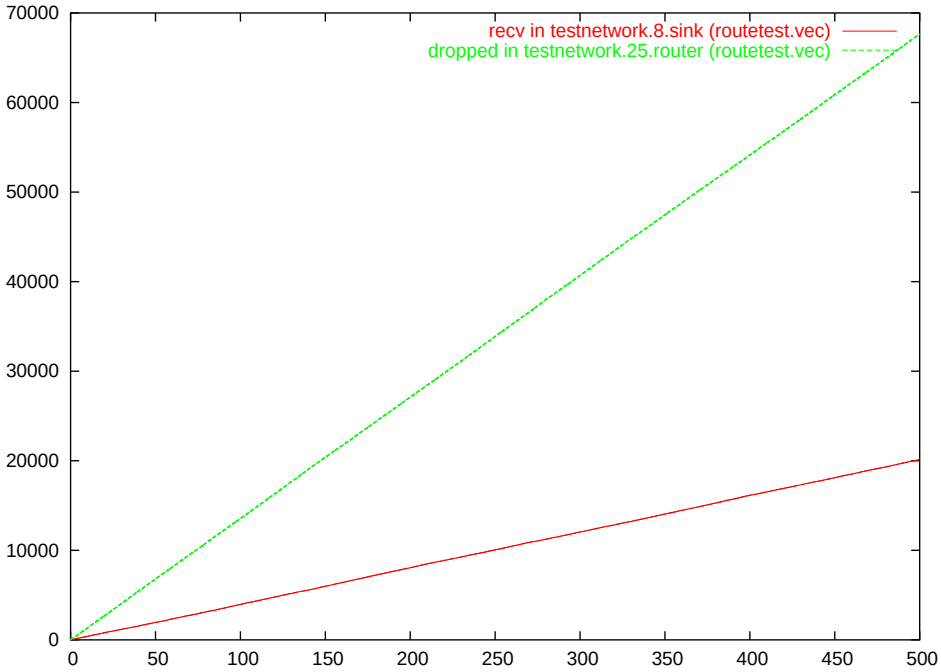


Figure 4.1: Received packets and dropped packets over time using the random router

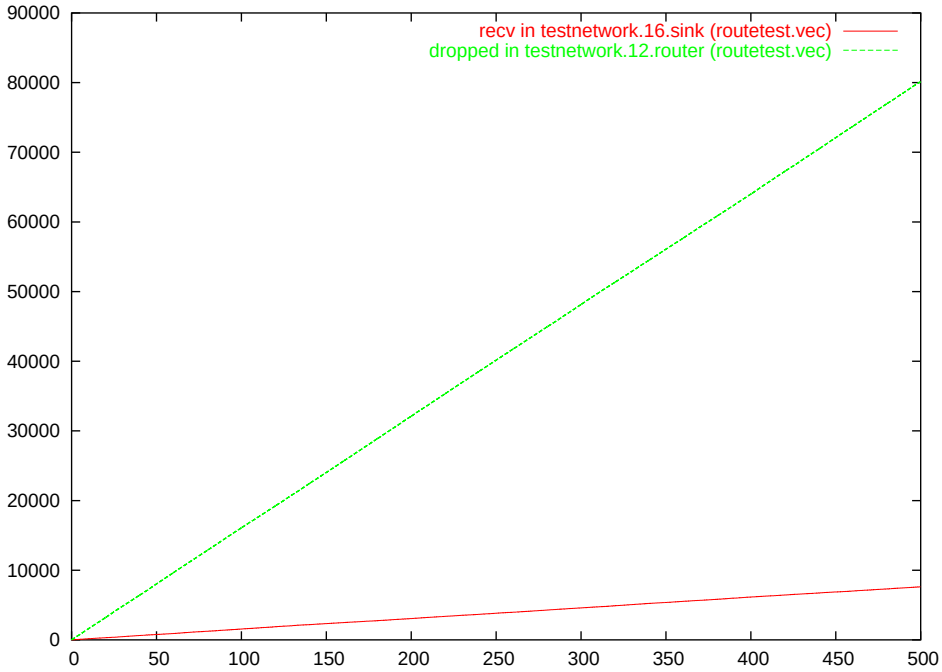


Figure 4.2: Received packets and dropped packets over time using the neural Q router

Chapter 5

Conclusion

In this case, the application of neural network function approximation to a reinforcement learning task was not successful. The algorithm was unable to converge to a stable routing policy, so clearly failed in its aims. This result does not cast doubt upon the utility of neural networks as a function approximators in a reinforcement learning tasks, but does highlight the difficulties that are involved in such a task. The inherent unstability of Q-learning when combined with a non-linear function approximator makes it hard to predict whether or not a particular algorithm will work.

Reinforcement learning and neuro-dynamic programming are both relatively new fields in computer science. The techniques involved still have some rough edges and are not always fully understood, this is particularly the case with the application of function approximation to reinforcement learning. Many papers have been written proposing techniques to overcome the problems faced [5, 16, 30], but as yet the definitive solution does not appear to have presented itself.

That the network failed to converge was disappointing, and while there has certainly been some success achieved combining neural networks and reinforcement learning, the Reinforcement Learning FAQ [30] contains the following passage:

> It is a common error to use a backpropagation neural network as the function approximator in one's first experiments with reinforcement learning, which almost always leads to an unsatisfying failure. The primary reason for the failure is that backpropation (sic) is fairly tricky to use effectively, doubly so in an online application like re-

> inforcement learning. It is true that Tesauro used this approach in his strikingly successful backgammon application, but note that at the time of his work with TDgammon, Tesauro was already an expert in applying backprop networks to backgammon. He had already built the world's best computer player of backgammon using backprop networks. He had already learned all the tricks and tweaks and parameter settings to make backprop networks learn well. Unless you have a similarly extensive background of experience, you are likely to be very frustrated using a backprop network as your function approximator in reinforcement learning.

Indeed, Sutton himself, a leading researcher in reinforcement learning rarely uses neural networks as function approximators, precisely because of the problems of convergence. The results that Boyan and Littman described as "disappointing" [7] with a neural network function approximator have been replicated here. It was perhaps over-ambitious to attempt to tackle this problem with my relatively limited knowledge of neural networks and on such a limited time-scale. The use of the C++ language for implementation probably made the task harder than it could have been due to it's complex nature and general unsuitability for numeric programming.

A different function approximator may have shown more success. A radial basis function and tile coding, such as proposed by Sutton and Barto [31] could be a good approach to try, as could self-organising maps [13]. Residual algorithms [5] were considered as a solution but would require the Q-routing algorithm to be re-phrased as an episodic learning task, which may not be feasible. A more thorough and time consuming analysis of the failures of the methods proposed in this project could lead to a successful application of neural network function approximation to the Q-routing problem, and it seems unlikely that such an aim could not be achieved eventually.

Reinforcement learning is a promising and fascinating approach to learning solutions to complex, dynamic problems in a variety of situations, however, care has to be taken when using such techniques with non-linear function approximators such as neural networks.

Appendix A

Source Code

A.1 Location

The source code for the simulation can be found at the following places.

- `http://www.dcs.shef.ac.uk/~u8wrn/project/`
- `http://www.misconception.org.uk/will/project/`

A.2 Requirements

To compile or run the code a copy of OMNET++ is required, and libxml2 is required for the topology loading code. The code has only been tested under Linux.

- *OMNET++*
  - `http://www.hit.bme.hu/phd/vargaa/omnetpp.htm`
- *libxml2*
  - `http://xmlsoft.org`

A.3 Files

Filename	**Purpose**
6x6.xml	6x6 irregular grid test network definition.
basicrouter.ned	Router interface implemented by all router types.
dvrouter.cc	Implementation of the distance vector router.
dvrouter.h	Header definitions for distance vector router.
dvrouter.ned	NED description of distance vector router.
generator.cc	Implementation of a packet generator.
generator.h	Header definitions for the packet generator.
generator.ned	NED description of packet generator.
links.ned	Link type definitions (Ethernet and Dial-up)
nqrouter.cc	Implementation of the neural Q router.
nqrouter.h	Header definitions for neural Q router.
nqrouter.ned	NED description of neural Q router.
omnetpp.ini	Initialization file for OMNET++.
routernode.ned	Network node that ties together a router, generator and sink.
rrouter.cc	Implementation of the random router.
rrouter.h	Header definitions for random router.
rrouter.ned	NED description of random router.
sink.cc	Implementation of a packet sink with statistics.
sink.h	Header definitions for packet sink.
sink.ned	NED description of packet sink.
testnetwork.cc	Implementation of XML topology loader.
testnetwork.h	Header definitions for XML topology loader.
testnetwork.ned	Network setup NED description.
util.cpp	Utility functions common to all code.
util.h	Utility function prototypes.

Appendix B

Colophon

B.1 Tools

This report was typeset with the LaTeX document formatting system using XEmacs and AUCTeX. The typeface used is Times New Roman from the freely available txfonts Postscript font package. The neural network and network topology diagrams were drawn using Kontour, part of the KOffice project. The UML diagram was drawn using Dia, part of the GNOME project. All graphs were produced with the aid of gnuplot.

This project was produced with 100% Free Software on a Debian GNU/Linux system.

B.2 Statistics

This document is comprised of around 7500 words and well over 1000 lines of TeX markup. The source code weighs in at over 3000 lines of code.

Bibliography

[1] NODElib Machine Learning Library. URL `http://www.neci.nec.com/homepages/flake/nodelib/html/`.

[2] Torch Machine Learning Library. URL `http://www.torch.ch`.

[3] Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 2(1):397–413, 1993.

[4] Sabih H. Gerez Asker M. Bazen, Martijn van Otterlo and Mannes Poel. A Reinforcement Learning Algorithm for Minutiæ Extraction From Fingerprints. Technical report, Department of Computer Science, University of Twente, 2001.

[5] Leemon Baird. Residual Algorithms: Reinforcement Learning with Function Approximation. Technical report, Department of Computer Science, U.S. Air Force Academy, 1995.

[6] Chris M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[7] Justin Boyan and Michael Littman. Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach. Technical report, School of Computer Science, Carnegie Mellon University, 1993.

[8] Robert Braden. RFC 2039: Recommendations on Queue Management and Congestion Avoidance in the Internet. Technical report, Internet Engineering Task Force, 1998.

[9] Wu chang Feng. Improving Internet Congestion Control and Queue Management Algorithms. Technical report, 1999.

[10] Tami Schuylerm Christos J. Moschovitis, Hilary Poole and Therese M. Senft. *History of the Internet*. ABC CLIO (Reference Books), 1999.

[11] Abhijit Bhose Craig Labovitz, Abha Ahuja and Farnam Jahanian. An Analysis of BGP Convergence Properties. *Computer Communication Review*, 29 (4), 1999.

[12] Robert Crites and Andrew Barto. Improving Elevator Performance Using Reinforcement Learning. *Advances in Neural Information Processing Systems*, 1996.

[13] Maria Gini Dean F. Hougen, John Fischer and James Slagle. Self-organizing Maps with Eligibility Traces: Unsupervised Control-Learning in Autonomous Robotic Systems. Technical report, Department of Computer Science, University of Minnesota, 1996.

[14] Gary William Flake. *The Computational Beauty of Nature*. MIT Press, 1998.

[15] Sally Floyd. A Report on Some Recent Developments in TCP Congestion Control. Technical report, ACIRI, 2000.

[16] Geoffrey J. Gordon. Stable Function Approximation in Dynamic Programming. Technical report, Computer Science Department, Carnegie Mellon University, 1995.

[17] Kevin Gurney. *An Introduction to Neural Networks*. UCL Press Limited, 1997.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1996.

[19] Sally Floyd K. Ramakrishnan and D. Black. RFC 3168: The Addition of Explicit Congestion Notification (ECN) to IP. Technical report, Internet Engineering Task Force, 2001.

[20] Shailesh Kumar. Confidence based Dual Reinforcement Q-Routing: An On-line Adaptive Network Routing Algorithm. Technical report, University of Texas, 1998.

[21] Mark K. Lottor. RFC 1296: Internet Growth (1981-1991). Technical report, Internet Engineering Task Force, 1992.

[22] G. Malkin. RFC 1721: RIP Version 2 Protocol Analysis. Technical report, Internet Engineering Task Force, 1994.

[23] Christophe Diot Martin May, Jean Bolot and Brian Lyles. Technical report.

[24] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[25] J. Moy. RFC 2178: OSPF Version 2. Technical report, Internet Engineering Task Force, 1997.

[26] John Nagle. RFC 896: Congestion Control in IP/TCP Internetworks. Technical report, Internet Engineering Task Force, 1984.

[27] Larry L. Petersen and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, 2 edition, 2000.

[28] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4 (BGP-4). Technical report, Internet Engineering Task Force, 1995.

[29] W. Richard Stevens. RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. Technical report, Internet Engineering Task Force, 1997.

[30] Richard Sutton. Reinforcement Learning Frequently Asked Questions. URL `http://www-anw.cs.umass.edu/~rich/RL-FAQ.html`.

[31] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[32] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall Inc.

[33] Gerald Tesauro. TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. Technical report, IBM Corp., 1993.

[34] John N. Tsitsiklis and Benjamin Van Roy. An Analysis of Temporal Difference Learning with Function Approximation. Technical report, Massachusetts Institute of Technology, 1996.

[35] András Varga. The OMNET++ Discrete Event Simulation System. *Proceedings of the European Simulation Multiconference (ESM'2001)*, 2001.

[36] Si Wu and K. Y. Michael Wong. Dynamic Overload Control for Distributed Call Processors Using the Neural-Network Method. *IEEE-NN*, 9(6), 1998.