Generalisable data-driven routing using Deep RL with GNNs

Oliver D. N. Hope

Jesus College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the Computer Science Tripos, Part III

University of Cambridge

Department of Computer Science and Technology

The Computer Laboratory

William Gates Building

15 JJ Thomson Avenue

Cambridge CB3 0FD

United Kingdom

Email: oliver.hope@cl.cam.ac.uk

June 2020

Declaration

I, Oliver D. N. Hope of Jesus College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11899

Signed: [illegible]

Date: 14th November 2020

Abstract

This project seeks to apply deep reinforcement learning (RL) techniques with graph neural networks to the space of intradomain traffic engineering in order to minimise link congestion. Recently, there has been work in this space from Valadarsky et al. in the paper 'Learning to route with Deep RL' where the authors achieved some success using RL with a multilayer perceptron (MLP) policy architecture. However, this approach suffers from a level of rigidity in that it cannot easily generalise to new networks or traffic types.

Aside from this work, there has been an explosion of research into graph neural networks (GNNs) which are a class of neural network the operates specifically on the structure of graphs. These make it easier to create only the intended relational inductive biases in policy design and have had much success in many fields, especially in terms of generalising solutions across different graphs.

In this project, we aim to combine these two ideas: the routing methods from Valadarsky et al. with GNNs to allow for generalisation. We provide a new environment in which different strategies can be implemented and tested, including different types of policy, traffic, and utility function. We also present the design of GNN-based policies that seek to solve this problem. Finally, we provide a comprehensive assessment of the different techniques, with a particular focus on their capability to generalise to different traffic patterns and network topologies.

Contents

List of Figures

CHAPTER 1

Introduction

Routing has always been an integral part of the functioning of the internet as it is required for data to traverse a network from source to destination successfully. Historically, strategies have mainly focussed on those that can be calculated in a distributed manner, are functionally correct, and are robust to network changes all while providing reasonable performance (both in the time to calculate routes and their effect on the traffic itself). More recently, and especially with the advent of software-defined networking (SDN), there has been a concerted effort to exert more control over how traffic is routed. This control has either been used to implement policies that favour types of traffic, particular customers, or simply aim to achieve particular notions of performance on the network.

In the area of intradomain routing where this kind of control is possible, one particular focus has been on minimising link over-utilisation, as congestion can have significant impacts on network performance for the end-user. To minimise congestion successfully, such protocols must take into account how one traffic flow on the network impacts the other traffic flows, which leads us to data-driven routing. It is already known how to route optimally given a set of traffic demands over a network. However, in general, these demands cannot be known in advance. Therefore, there have been efforts to create routing schemes that will generally give good congestion performance no matter the particular traffic demands on the network (called oblivious routing). Further research has aimed to make these routing strategies include some notion of the current traffic.

A recent paper: 'Learning To Route with Deep R' examined whether we can use reinforcement learning to produce better routing strategies than the oblivious approach. It worked under the assumption that there is some regularity in sequences of traffic demands on a network and that using this regularity we can get closer to the optimal performance. This work produced some impressive results. However, once it has learnt to route on a particular network, due to the rigid structure of the neural network used to make the policy, this cannot be applied to a different network. This issue may seem relatively small, that is until one takes into account the fact that networks often change, often due to temporary outages. If, for example, a link or node in the network were to be temporarily off-line, then such a system would be unable to provide a routing scheme.

This project seeks to take the problem specified in 'Learning to Route with Deep RL' as well as its solution and extend its domain to cover changing the structure of the network itself. In other words, the aim is to use reinforcement learning (RL) to enable the creation of close-to-optimal routings for a network, given a history of traffic demands on that network, and that this should be able to generalise both over different demand sequences for the same network and different demand sequences for entirely different networks. Graph generalisation is a useful goal as it allows a learned routing strategy to continue to work on networks suffering faults such as dropped links and also may mean that the model would not need to be retrained to be used on different networks. Finally, we aim to show that this is not just a toy problem and solution but leads to results on real-world datasets.

In summary, this study makes the following research contributions:

- Provides an environment for experimenting with RL in data-driven routing
- Designs a new mapping from edge weights to a fully specified multipath routing
- Introduces policy designs for approaching data-driven routing in a way that is generalisable to different network topologies
- Presents a comprehensive assessment of the performance of different techniques with a specific focus on generalisability over traffic patterns and network topologies.

The rest of the dissertation is structured as follows: chapter 2 introduces both previous research in this area and the research on which we base techniques in this work; chapter 3 presents the formal specification of the problem, how we designed the environment, and techniques that made the problem and routing feasible to be performed using RL; chapter 4 describes how the RL policy was designed and trained, and the structure of the graph neural networks (GNNs) used; chapter 5 explains the evaluation framework and examines the results of the experiments performed; and chapter 6 summaries the findings and contributions of the entire work as well as providing an insight into possible future work.

CHAPTER 2

Background and related work

In this chapter, we provide an overview and summary of the research that has been used by and influenced this work, as well as covering similar research that has been performed. This overview includes a look at traffic engineering, reinforcement learning techniques, the advent of graph neural networks and other approaches to optimising networking using reinforcement learning.

2.1 Traffic engineering

2.1.1 Intradomain routing

The field of internet traffic engineering contains the problem of how to control and manage computer networks. One such problem is that of intradomain routing. Intradomain routing focusses on which paths data should take on a single network entirely controlled a single entity which generally has complete knowledge of the structure of the network and control over how the routing is to be managed (e.g. which protocol to use). An example of what we mean by a network controlled by a single entity in this context is an autonomous system (AS). Autonomous systems can scale from the size of a small company to that of a large internet service provider (ISP).

The benefit of working inside an AS rather than the full internet is the complete knowledge and control, which gives considerably more scope for achieving optimality concerning network control. For example, a network administrator can decide their priorities for the network and internally apply policy and routing that reflects those aims. Common aims are minimising latency between source and destination or minimising link utilisation to avoid congestion and packet loss. Examples of the most common protocols used for intradomain routing are routing information protocol (RIP)[26] and open shortest path first (OSPF)[12] which both minimise distance, generally equating to latency.

2.1.2 Data-driven routing

In the previous section, we introduced intradomain routing along with the protocols commonly used for routing. These protocols have generally led to good performance and are deployed almost everywhere. However, they only take into account the structure of the network itself and not the traffic that goes over it. As different flows on the network can interfere with each other, the data on the network can have an impact on the utility of the routing used. Therefore, if possible, taking into account the traffic conditions when performing routing can have a significant positive impact on performance (whether this is measured by latency, throughput, or some other metric). These considerations lead us to data-driven routing[38].

Data-driven routing seeks to make routing decisions that take into account the impact of traffic on the network. This is often done to provide improved performance by reducing link congestion, which is the utility that we will focus on here.

2.1.3 Multicommodity flow

The problem of data-driven routing can be modelled as a multicommodity flow problem[6]. The multicommodity flow problem is simply a network flow problem with multiple demands between different source and sink nodes. It consists of a flow network $G(V, E)$ where each edge $(u, v) \in E$ has a capacity $c(u, v)$. Then we wish to place commodities $K_i = (s_i, t_i, d_i)$ on the network where s_i is the source, t_i is the sink and d_i is the demand (the amount of data to pass between source and sink). Finally, the following constraints must be satisfied:

1. Flow on a link must not exceed its capacity:
   $\forall (u, v) \in E : \sum_{i=1}^{k} f_i(u, v) \cdot d_i \leq c(u, v)$
2. Flow entering and exiting a node must be conserved:
   $\sum_{w \in V} f_i(u, w) - \sum_{w \in V} f_i(w, u) = 0 \quad \text{when} \quad u \neq s_i, t_i$
3. The entire flow of a commodity must exit its source:
   $\sum_{w \in V} f_i(s_i, w) - \sum_{w \in V} f_i(w, s_i) = 1$
4. The entire flow of a commodity must be absorbed at the sink:
   $\sum_{w \in V} f_i(w, t_i) - \sum_{w \in V} f_i(t_i, w) = 1$

In this framework, we can specify any valid routing we wish (including splitting one flow across multiple routes). However, importantly we can also optimise the routing under a given utility function using linear programming in polynomial time for a specified set of commodities (as long as we allow fractional flows)[11]. The existence of this method, in effect, means optimal data-driven routing for reducing link congestion is solved except for the critical fact that in most cases we do not have prior knowledge of the traffic demands on a network so the routing would always lag the commodities for which it was optimised.

2.1.4 Current approaches to reducing link congestion

Current approaches to traffic engineering often rely on the use of SDN, which separates the control plane and data plane and allows for more complicated and custom routing schemes to be easily deployed[8]. In addition to these SDN systems, many different methods have been used to achieve better network performance in the face of high traffic demands.

One such method is forms of oblivious routing[5] which seek to reduce congestion on links in a network without the knowledge of what the traffic demands may be. Early examples of approaches are that of Räcke[32] who showed that it is possible to have an optimal oblivious routing and Azar et al.[4] who proved that this can be calculated in polynomial time. Since then, small, incremental improvements have been made, improving calculation time or allowing for different utility functions to be specified[23].

However, it is still possible to do better than oblivious routing if one has some knowledge of the traffic. One seminal work in this area is SMORE[24] which combines oblivious routing and adaptive sending rates to improve reliable network performance under all kinds of operating conditions.

One will notice that while all of these methods do successfully improve network performance, in the sense of reducing link congestion, they are all fundamentally limited by their knowledge (or lack thereof) of the current operating conditions of the network for which they are creating a routing strategy.

2.2 Machine learning

2.2.1 Reinforcement learning

Reinforcement learning[41] is a subsection of machine learning, the third of the three paradigms, the others being supervised and unsupervised learning. It is concerned with training agents to take actions in some specified environment and achieve a goal in that environment as can be seen in Figure 2.1. In particular, deep RL[14] is a form of RL where

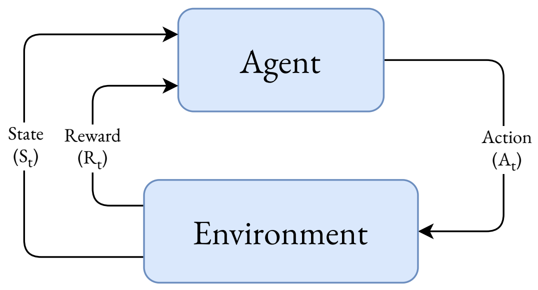

Figure 2.1: The structure of a reinforcement learning problem: the interaction of agent and environment

we apply deep learning techniques to make solutions to problems more efficient and feasible. Commonly the agent makes choices using a neural network which is trained using any one of multiple algorithms designed for this purpose (all with different trade-offs). Deep RL is powerful because it does not require an analytical model of the environment being used as it can find out how to act through exploration alone. RL has been used successfully in many fields approaching many different problems such as in robotics[15], game-playing[42], and general optimisation with one recent success being playing board games to a super-human level with AlphaZero[39].

Formally, a RL problem is generally framed with the environment modelled as a finite markov decision process (MDP) which can be defined as the 4-tuple (S, A, P_a, R_a) where:

- S is a finite set of states
- A is a finite set of actions
- $P_a(s, s')$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$
- R_a is the reward received after transitioning from state s to state s', due to taking action a

With a MDP defined as our environment, it is then necessary to create an agent to interact with it. Generally, such an agent is implemented by a policy which will return an action given an observation (a set of features probabilistically derived from the current state of the environment). This policy can take many forms, but in deep RL, as used in this research, it takes the form of a neural network.

The final piece of the RL puzzle is a way to ensure the policy acts how we want it to (or formally to maximise the reward it can achieve in an episode). Again, there are multiple ways to do this. One such method is the policy gradient method with the first prominent example being REINFORCE[48]. One branch of this to have gained traction, in particular, are actor-critic algorithms. These have two main parts:

- The *critic* which updates the value function ($V(s)$, the expected reward from a given state) given feedback from the environment.
- The *actor* which updates the policy ($\pi(a|s)$, the probability of choosing an action in a state) using feedback from the critic.

Proximal policy optimisation (PPO)[37] is an example of an actor-critic algorithm that achieves good sample efficiency and performance while maintaining a reasonably simple implementation (relative to similar algorithms). These characteristics are mainly due to how it restricts the maximum size of policy updates. It has had notable successes, such as beating world champions in a live game of Dota 2[30].

2.2.2 Graph neural networks

Graph neural networks (GNNs)[17, 36] are a form of neural network that has gained considerable traction over the last few years. This is because, in the same way that convolutional neural networks (CNNs) have been used with much success on images, as pixels form a grid and are often related to their neighbouring pixel, many other datasets are graph-structured with relations to neighbouring nodes being an essential part of this structure. GNNs effectively allow us to generalise the CNN model onto graphs. There are many different types of GNN with different trade-offs as described in countless surveys of the subject[51, 49].

Possibly the most general model, allowing for restrictions to define the different subtypes of graph network is that proposed by Battaglia et al.[7]. They propose that combinatorial generalisation is crucial to learning and that this relies on the integration of structured and unstructured approaches. All networks contain some level of structure leading to relational inductive biases. However, it is only graph networks that give fine-grained control over these biases to fully harness the structure present in the problem being solved and without implying structure that is not present.

The model presented by Battaglia et al. defines a graph network (GN) block which takes as input a graph and returns a graph. Here a graph is defined to be the 3-tuple $G = (u, V, E)$ where u is a global attribute vector, $V = \{v_i\}_{i=1:N^v}$ is the set of vertex attribute vectors, and $E = \{(e_k, r_k, s_k)\}_{k=1:N^e}$ is the set of edge attribute vectors and their source and destination vertices.

The graph block itself computes over an input graph as shown in Algorithm 1. This algorithm contains three ϕ functions which update attribute information and three ρ

Algorithm 1 Steps of computation in a full GN block. (Taken from [7])

```
function GraphNetwork($E, V, \mathrm{u}$)
    for $k \in \{1 \dots N^e\}$ do
        $\mathrm{e}'_k \leftarrow \phi^e\left(\mathrm{e}_k, \mathrm{v}_{r_k}, \mathrm{v}_{s_k}, \mathrm{u}\right)$        ▷ 1. Compute updated edge attributes
    end for
    for $i \in \{1 \dots N^n\}$ do
        let $E'_i = \left\{\left(\mathrm{e}'_k, r_k, s_k\right)\right\}_{r_k=i,\ k=1:N^e}$
        $\bar{\mathrm{e}}'_i \leftarrow \rho^{e \to v}\left(E'_i\right)$        ▷ 2. Aggregate edge attributes per node
        $\mathrm{v}'_i \leftarrow \phi^v\left(\bar{\mathrm{e}}'_i, \mathrm{v}_i, \mathrm{u}\right)$        ▷ 3. Compute updated node attributes
    end for
    let $V' = \left\{\mathrm{v}'\right\}_{i=1:N^v}$
    let $E' = \left\{\left(\mathrm{e}'_k, r_k, s_k\right)\right\}_{k=1:N^e}$
    $\bar{\mathrm{e}}' \leftarrow \rho^{e \to u}\left(E'\right)$        ▷ 4. Aggregate edge attributes globally
    $\bar{\mathrm{v}}' \leftarrow \rho^{v \to u}\left(V'\right)$        ▷ 5. Aggregate node attributes globally
    $\mathrm{u}' \leftarrow \phi^u\left(\bar{\mathrm{e}}', \bar{\mathrm{v}}', \mathrm{u}\right)$        ▷ 6. Compute updated global attribute
    return $(E', V', \mathrm{u}')$
end function
```

functions which pool attribute information to be used in the updates. It is the coefficients in these functions that are learnt and the ability to change the operations that these functions perform, which allows for the flexibility to implement many different types of GNN.

2.3 Approaches to routing using reinforcement learning techniques

There have thus far been multiple approaches to network-related problems using machine learning and especially reinforcement learning techniques[25]. There is a growing body of work covering areas from routing to deep packet inspection. Specifically, in the area of routing, reinforcement learning research has been taking place for quite some time with an early seminal paper introducing the idea of Q-routing in 1994[9] which attempts to use distributed on-device agents to route packets to minimise delay. Many pieces of work have followed a similar vein, gradually improving on Q-routing with ever more complex policies[50, 2].

A separate issue is the traffic engineering problem where one can make global decisions for a large scale network (such as an autonomous system) on a longer timescale. In this area, one major work: 'Learning to Route with Deep RL'[44] has recently been published, which is the paper that our work seeks to extend and improve. This paper introduces the problem of knowing the history of traffic demands for a given network and using it along with a neural network to predict what routing should be used on the network in the next timestep to minimise link over utilisation. This routing is performed using reinforcement learning with a novel translation from action to routing.

Similarly, another paper: 'A Deep-Reinforcement Learning Approach for Software-Defined Networking Routing Optimisation'[40] sought to use reinforcement learning to find good routes for a given traffic matrix. However, this means that the matrix must be known in advance, and we already have deterministic algorithms than can find optimal routings given a traffic matrix. Also, within the last few months, there has been further work on optimising routing, this time looking at link utilisation and using GNNs[35]. However, this approach learns to cope with the current traffic flow as it is occurring and is distributed rather than centralised.

CHAPTER 3

Problem specification and approaches

3.1 Introduction

In this chapter, we describe the formal specification of the problem to be solved whilst also introducing several tactics that we have used to make this more feasible. The contents include a description of methods to reduce the action space and how we can still end up with a fully defined routing. As part of this description, we introduce an algorithm to remove unwanted cycles from a network graph.

3.2 Work from 'Learning to route with Deep RL'

In chapter 2, we introduced the work of Valadarsky et al.[44] and that it was the basis for the work performed here. Specifically, that paper introduced a formal structure within which to define an RL routing policy, a specification for the generation of demand matrices (DMs) to be trained over, a translation from edge weights to routing strategies, and formulation of a reward function.

The policy that they introduced was multilayer perceptron (MLP)-based. The issue with this is that its input and output sizes are fixed, meaning that it will only be usable for a single graph. This means that it must be trained separately for each different network and cannot be used when there is a temporary fault such as a dropped link or node. It was also only evaluated on one type of sequence at a time, meaning that we cannot know if a model trained on one type of sequence generalises to others at all.

3.3 Routing formal specification

The introduction explained that this project seeks to perform routing that is closer to the optimal in the presence of knowledge of traffic demands than oblivious techniques. To do this, we must first define the routing model on which it acts. We take a similar model to that of Valadarsky et al. In this context we consider a static network upon which we can specify a routing, and which also has a set of traffic demands associated with it in the form of traffic matrices. Formally we have:

- The *network*, which is modelled as a directed graph where all the edges have a link capacity: $G = (V, E, c)$ where V is the set of vertices, E is the set of edges and $c : E \rightarrow \mathbb{R}^+$ is a function mapping each edge in the graph to its capacity.
- The *routing*, which for each flow (demand and source and destination pair) specifies at each vertex how much of that flow should be sent down each of its edges to each of its neighbours. Therefore, if we define $\Gamma(v)$ to be the set of all neighbours of vertex v then we can define a routing to be $\mathcal{R}_{v,(s,t)} : \Gamma(v) \rightarrow [0, 1]$ with $\mathcal{R}_{v,(s,t)}(u)$ as the proportion of the flow passing from s to t through vertex v that is forwarded to vertex u. Importantly, any routing specified must obey the two constraints:
    1. No traffic is lost between source and destination:
       $\sum_{u\in\Gamma(v)} \mathcal{R}_{v,(s,t)}(u) = 1 \qquad \forall s, t \in V \wedge v \neq t$
    2. All traffic for a destination is absorbed at that destination:
       $\sum_{u\in\Gamma(v)} \mathcal{R}_{t,(s,t)}(u) = 0 \qquad \forall s, t \in V$
- The *demands* which can be represented as matrix $D \in \mathbb{R}^{|V|\times|V|}$ called a DM where each element D_{st} is the traffic demand between the source s and destination t.

With these definitions, we can now fully describe the movement of traffic over a network. The rest of this work will use just this system and no more or less when talking about networks, routings or demands. In examples, we will assume that the network itself is fixed and that traffic is described by sequences of DMs, also fixed, each of which represents a discrete timestep. The only thing we are allowed to modify is the routing strategy. Given the above structure, we can define some notions of the goodness of a particular routing. In particular, as we aim to reduce congestion, we will make our utility function that of minimising link over-utilisation.

A linearisation of this utility can be defined to be minimising U_{max} in equation 3.1 where

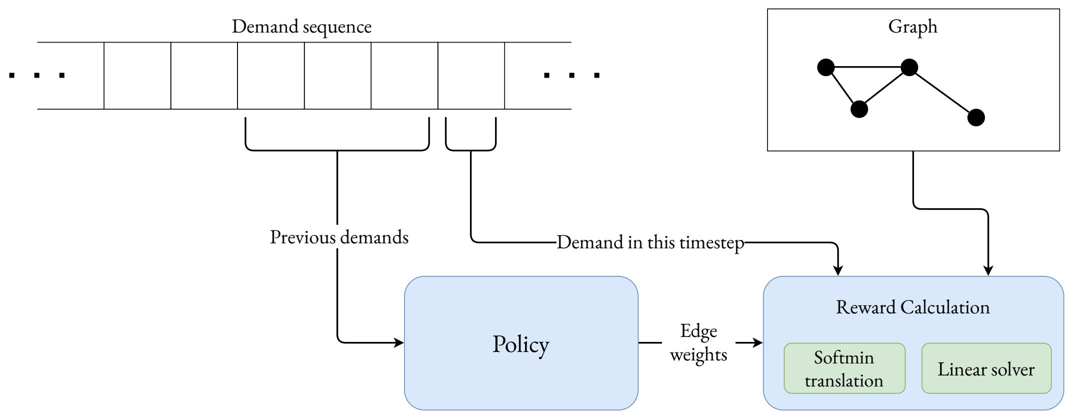

Figure 3.1: Environment dataflow in a single timestep. Top left is the generated demand sequence, and top right is the graph we are routing over. We can see that the previous n demands (here 3) are given to the policy as input while the reward is calculated using the new demand for this timestep.

$U(u, v)$ is the utilisation of the link (u, v).

$$\forall (u, v) \in E : U_{max} > U(u, v) \tag{3.1}$$

3.4 The environment

So that it would be possible to both train and evaluate approaches to the defined problem, it is necessary first to construct an environment to simulate the desired responses with which an RL agent could interact. For easy interoperability with existing libraries, we decided that this environment should have an OpenAI Gym[10] API. Alongside the internal operations of the environment being correct, its interface to the RL agent is crucial as it has a significant impact on how well the agent can learn. A diagram of the overall structure is given in figure 3.1. We will now proceed to discuss how the environment calculates rewards, the format of observations it gives to the agent and the format of actions it receives from the agent.

3.4.1 Reward calculation

As the optimal routing that can be achieved on a given graph varies for different demand matrices, the reward cannot be derived solely from the calculation of the maximum link utilisation. Fortunately, as described in section 2.1.3, we know that an optimal routing does exist and it can be found in polynomial time with linear programming (LP). To do this, we solve the problem using the standard LP formulation, minimising the utility function given in equation 3.1. The environment implements a linear solver for the optimal routing to calculate the optimal link utilisation[1]. Then, the reward is derived by comparing the routing produced by the RL agent to the calculated optimal routing. The original work used equation 3.2, where U_{max} is the maximum link utilisation. However, this reward does not well represent the increased difficulty in improvement the closer the routing is to the optimal, and so leads to the learning converging too early. To solve this problem, we instead used equation 3.3, which gives smaller increases in the ratio closer to the optimal congestion a higher relative reward.

$$\text{reward} = -\frac{U_{max_{agent}}}{U_{max_{optimal}}} \tag{3.2}$$

$$\text{reward} = e^{\frac{U_{max_{optimal}}}{U_{max_{agent}}}} \tag{3.3}$$

3.4.2 Observations

In the original 'Learning to route with Deep RL' paper, an observation is a history of traffic demands, which is presented as a list of traffic matrices and then flattened for input to the MLP. However, our work aims to make the solution generalisable over graphs of different shapes using GNNs. As GNNs work on graphs, it is possible to vary both the number of vertices and edges that are input to a GNN (something not possible with an MLP). However, this previous method of managing demands as observations no longer works. The reason for this is that a sensible way to input the demands to the GNN would be to place the demands associated with each vertex on that vertex as vertex attributes. The issue here is that the number of demands associated with a vertex scales with the number of nodes in the graph. Therefore, the size of node attributes in the GNN would have to grow as more vertices are added, which is unfortunately not possible within the structure.

To solve the above issue we had to find a way to associate the demands with the correct nodes such that the agent is still able to learn but only requires a constant amount of space per-vertex as the graph grows. The solution we decided upon for this problem was summing for each vertex the total outgoing flow and incoming flow, meaning that the observation size is now $O(|V|)$ as opposed to $O(|V|^2)$ and so can be used with a GNN.

One other important addition enabling this new structure to work was normalising the inputs, as otherwise the more vertices in a graph, the greater the size of the input features, which is unwanted behaviour.

[1]The solver is implemented on top of Google OR-Tools[31]

3.4.3 Action space

Following on from the definitions given in section 3.3, we can see that one valid way for the agent to assign a routing would be to provide splitting ratios for each edge under each flow. However, this would require an output of $|V| \times (|V| - 1) \times |E|$ separate values. Unfortunately, this size of action space is too big to enable successful learning. If we make some approximating assumptions about the routing (which will no longer allow us to achieve the optimal routing but still allow us to get closer than oblivious strategies), then we can reduce this output size.

A first method to reduce the action space size is to ignore the source of any packets, forming a destination-only routing. This method reduces the size to $|V| \times |E|$. However, this is still too large, so instead, we had to create a method of deriving routing strategies from setting weights of edges, creating an action space with only $|E|$ values. This space was finally small enough to achieve good results. The next section describes how the routing strategy works.

3.5 Softmin routing

The paper 'Learning to route with Deep RL'[44] also confronted the same issues of action space size and so created a method of deriving a routing strategy from edge weights which they called *softmin routing*. As part of our research, we attempted to use the same solution specified but found some issues and therefore had to modify it significantly. It is this modified softmin routing that we will present below.

Softmin routing is a way of deriving the splitting ratios on each edge from each vertex, per flow, given weights that have been set on each edge. These values are calculated using algorithm 2 which we shall now describe. We calculate the ratios per-flow, where a flow is a source-destination pair, (s, t). For each vertex, we calculate its distance to the destination vertex (along its shortest path using the weighted edges). Then, for each vertex, we add the weight of each outgoing edge to the distance of the neighbour at the end of that edge. Using the softmin function given in equation 3.4 with these summed numbers as input, we are returned splitting ratios to use on these edges for routing under this particular flow. We then repeat this process for all vertices and all flows.

$$\operatorname{softmin}(x) = \left(\frac{e^{-\gamma x_i}}{\sum_j e^{-\gamma x_j}} \right)_i \tag{3.4}$$

One will notice from this description that the routing derived from such a scheme does

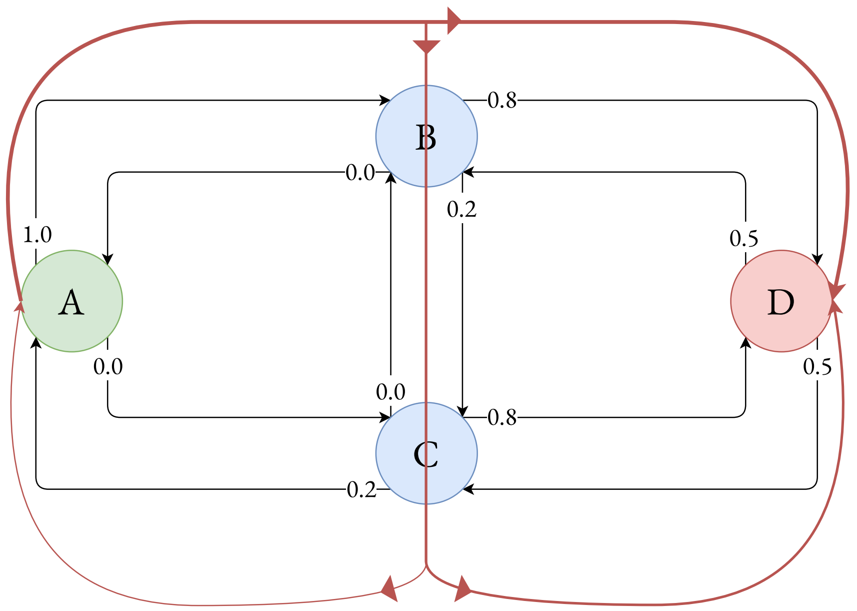

Figure 3.2: An example of a bad routing created by the softmin routing strategy. A loop between nodes A, B, and C causes excess link utilisation and increased latency. The circles are vertices with A being the flow source and D the flow destination. The black lines are edges with the numbers signifying their splitting ratios. The red line is the flow induced by this particular routing.

contain a potential flaw. Although it follows all the rules specified for a routing strategy in section 3.3 (no traffic is lost, and sources send the required demand which is all absorbed by the destination), there is nothing to stop routing loops occurring. An example of such a situation can be seen in figure 3.2. This situation is undesirable for two reasons. The first is that it wastes capacity as it means traffic traces the same route more than once, and the second is that it increases latency (this is bad but not under measurement here so does not impact our goals). Therefore, to achieve good results, we have to break routing loops.

3.5.1 Multipath DAG

The breaking of loops is a problem often faced in routing. In many protocols such as shortest-path based protocols, it is not usually an issue, as the shortest path by definition cannot contain any cycles, but this is not always true as distance vector protocols can produce issues when the network state has not yet converged. It would be easy to remove all loops between source and destination by only keeping the shortest path. However, this does not assist with load-balancing and so is not a useful strategy here. What we aim to do is take advantage of multipath with our routings, so the longer paths need to remain in the network.

An approach to this problem is looking into ways to convert the graph to a directed acyc-

Algorithm 2 Softmin routing algorithm: the steps taken to convert the learned edge weights given by the RL agent into a fully-defined routing strategy.

```
function SoftminRouting($(G, w, \gamma)$)
    for $(s, t) \in flows$ do
        $G \leftarrow$ PruneGraph($G$, $(s, t)$, $w$)    ▷ Convert to a DAG for the source-sink pair
        for $v \in V$ do
            $d[i] \leftarrow$ ShortestPath($(i, t)$)    ▷ Find the distance of each vertex to the sink
        end for
        for $v \in V$ do
            $out\_edges \leftarrow$ GetOutEdges($(v)$)
            $out\_weights \leftarrow \{w[(u, v)] + d[v] \mid (u, v) \in out\_edges\}$
                                                    ▷ Edge length + neighbour's distance
            $softmin\_weights \leftarrow$ Softmin($(out\_weights, \gamma)$)
            for $e \in out\_edges$ do
                $splitting\_ratios[e] \leftarrow softmin\_weights[e]$
            end for
        end for
    end for
    return $splitting\_ratios$
end function
```

lic graph (DAG) for each flow where the source and destination become source and sink. Similar work to this is in the use of destination-oriented directed acyclic graphs (DODAG) in the routing protocol for low-powered and lossy networks (RLP)[1]. There has been research in the space of constructing DODAGs for multipath applications such as work by Iova et al.[21]. However, being for RLP, this focussed on energy minimisation amongst other things. Therefore, we had to devise a new algorithm to efficiently retain as many paths from destination to source as possible whilst avoiding loops. One attempt was to use a weighted breadth-first search (BFS) from the sink to create a partial ordering on the graph and then only retaining edges that take us closer to the source. However, this still leads to the loss of too many usable links.

The final strategy we devised is shown in algorithm 3. The input to the algorithm is the graph with edges weighted by the agent. We begin by running Dijkstra's algorithm from the source, recording for each node its parent (in the case of the sink, multiple parents) and any locations where the frontier hits an already explored vertex, called *frontier meets*. Then we trace back from the sink to source, following parent links, and marking any vertex we pass through as *on path*. At this point, we have only recorded any shortest paths. Therefore, for every *frontier meet*, we find the distance to sink of the first ancestor from each side of the edge to the sink. We then update all the parent information and *on path* information for these vertices so that it becomes a valid path from the more distant ancestor to the closer ancestor. Finally, we remove all edges between nodes that are not *on path* and all edges that are *on path* but where the head of the edge is at the parent.

3.6 Traffic demand sequences

We are using RL to approach this problem, and the input to the agent is a history of previous demand matrices. The reason for this is that we have made the assumption that the sequence of DMs will have some sort of regularity which we can exploit to predict a routing strategy for the next timestep that is better than the optimal oblivious routing. Therefore, the demand sequences used for training such an agent must be built using some form of regularity. We try two different types of regularity, heavily inspired by those selected by Valadarsky[44].

The first kind of regularity is *cyclical*. Here we generate a cycle of demand matrices, and the sequence is simply some number of repetitions of this cycle. The second kind is an *averaging* sequence which is generated by taking a sequence of demand matrices and averaging over the last n when outputting the next demand. These scenarios make sense as temporal regularity is often seen in large networks such as those of ISPs[13].

The demands used to generate the sequences also come in two types. The first is the *bimodal*[28] demand where demands between pairs of nodes are drawn randomly from one of two normal distributions with different means to simulate standard network traffic with occasional elephant flows. The other is a *gravity* demand[33]. We generate this demand in a deterministic manner taking into account the structure of the network. Effectively we approximate the ingoing flow and outgoing flow to each node from the capacities of their edges. As this demand matrix is deterministic, to add variance allowing us to build sequences of different demands, we employ a technique called *sparsification*, whereby we stochastically remove demands between some nodes when building the sequence.

The formal definitions for the demand matrix sequences are as follows:

- Bimodal DM (parameters given by example values):
  $D_{ij} = p$ if $s > 0.8$ else q where $p \sim \mathcal{N}(400, 100), q \sim \mathcal{N}(800, 100), s \sim \mathcal{U}(0, 1)$.
- Gravity DM:
  $D_{ij} = \sum_{(u,v)\in E,\ u=i} c(u, v) \times \sum_{(u,v)\in E,\ v=j} c(u, v)$ where $c(u, v)$ is the capacity of the edge (u, v).
- Cyclical sequence:
  $x = \left\{D_{i \bmod q}\right\}_i$ where D is a sequence of q DMs.
- Averaging sequence:

$x = \left\{ \left(\sum_{j=i-q}^{i} D_{j \mod q} \right) / q \right\}_i$ where D is a sequence of q DMs.

- Sparsification:
  $\text{sparsify}(D, p)_{ij} = 0$ if $s > p$ else D_{ij} where $s \sim \mathcal{U}(0, 1)$

3.7 Training graphs

For training and evaluation purposes, the graphs trained on must be representative of real-world networks. We thought that potentially graphs from this set could be generated. However, it is difficult to define the set of graphs akin to real-world AS networks. Fortunately, there is an open dataset, 'The Internet Topology Zoo'[22] which contains a plethora of real-world networks, sufficient for both training and evaluation, so we decided to employ this instead.

Algorithm 3 DAG conversion algorithm retaining high path count from source to sink

```
function PruneGraph($(V, E, W), (s, t)$)
    $queue \leftarrow [(0, s, [])]$                ▷ Queue element is: (distance, vertex, parents)
    while $|queue| \neq 0$ do                     ▷ Run Dijkstra's algorithm
        $(d, v, p) \leftarrow \text{pop}(queue)$
        $parents[v] \leftarrow p$
        for $u \in \text{neighbours}(E, v) - p$ do
            if $u = t$ then                        ▷ Stop when reach sink
                $\text{append}(parents[t], v)$
            else if $u \in explored$ then          ▷ Record when see already explored node
                $\text{append}(frontier\_meets, (v, u))$
            else
                $\text{push}(queue, (d + W(u, v), u, [v]))$
            end if
        end for
        $\text{append}(explored, v)$
    end while
    $queue \leftarrow [t]$
    while $|queue| \neq 0$ do                     ▷ BFS to mark vertices on path source to sink
        $v \leftarrow \text{pop}(queue)$
        $\text{append}(on\_path, v)$
        for $p \in parents[v]$ do
            $d[p] \leftarrow d[v] + G[p][v]$
        end for
    end while
    for $(u, v) \in frontier\_meets$ do           ▷ Assign paths at frontier collisions
        $a \leftarrow \text{ancestor}(E, on\_path, u)$
        $b \leftarrow \text{ancestor}(E, on\_path, v)$
        if $d[a] = d[b]$ then
            continue
        end if
        $\text{ConnectPath}(a, u, v, b)$             ▷ Update parent pointers for new path
    end for
    for $(u, v) \in E$ do                         ▷ Remove edges not leading to sink
        if then$u \notin on\_path \land v \notin on\_path \land u \notin parents[v]$
            $E \leftarrow E - \{(u, v)\}$
        end if
    end for
    return $(V, E)$
end function
```

CHAPTER 4

Routing agent policy design

4.1 Introduction

In this chapter, we describe the basic policies used in previous work and their shortcomings, the policies that we have implemented, and their benefits. We also discuss how these policies were designed, and the structure of learning used to achieve the best results.

4.2 Environment interface

We describe he general interface between agent and environment for this problem in chapter 3. However, it omitted one important detail. When using the softmin routing strategy, one of the parameters for input to the softmin function was a value γ. In their paper, Valadarsky et al.[44] set this value to 2.0. They also only ran their experiments on a single graph. We quickly realised that the best value to use for γ would also depend on the topology of the graph used, and so although it was a hyperparameter in prior work, now that the graph changing is part of the problem space, γ also has to be output from our agent and so it is placed into the action space.

4.3 Standard policies

We use two policies as baselines against which we compare the new GNN-based method. The first of these is an multilayer perceptron (MLP)[34]. In this case, we use a fully-connected network as which is the type used in previous work. It is important to note that this policy does not take into account any of the structure of the problem, so it is, in effect, the simplest approach. However, it is also the least flexible as its input and output

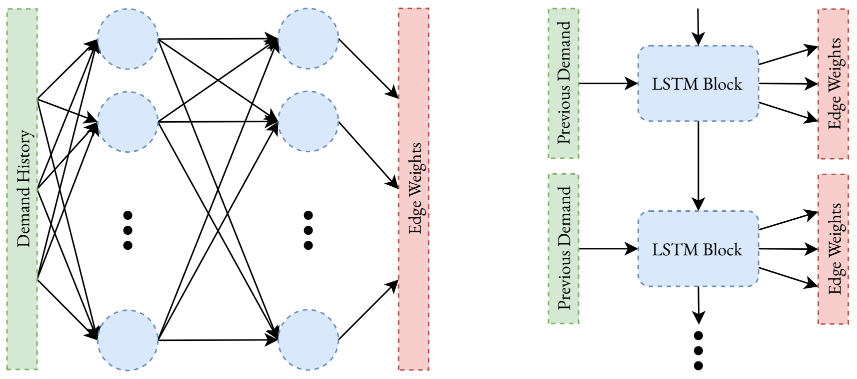

Figure 4.1: The structure of the MLP and LSTM policies for data-driven routing. On the left is a representation of the MLP policy and on the right is a representation of the LSTM policy.

sizes are both fixed. This lack of flexibility means that it can only be trained and used on one graph and with one length of history as the observations must fit the input.

We then extend this model to use an long short-term memory (LSTM)[20] as the history structure of the observations is limiting and only works if we know it is longer than any regularity we wish to find. The LSTM model is different in that at each timestep the observation it receives is only the previous DM with the hope that it will learn to interpret the history in some meaningful way using its recurrent memory. This is beneficial as it allows for flexibility of history length and means that less structure in the problem is assumed. However, it still suffers from the same issue as the MLP. Namely, it has a fixed size input and output, meaning that it can only be trained and used on a single graph. In fact, it is potentially possible to use an LSTM in a similar way to the iterative GNN model described in section 4.5, but that is beyond the scope of this research. A diagram of the MLP and LSTM policies can be seen in figure 4.1.

4.4 GNN policy

The new work is introduced here with the application of GNNs to the problem. The first stage of this was to make sure that a GNN can perform routing to the same level as the MLP. To achieve this, we began with the constrained space where we read out an entire routing from the GNN policy as a single action. Due to the nature of training frameworks, this, unfortunately, allows us only to train the network on one graph as the output size must be fixed and is the number of edges of the graph. However, it does still allow us to apply the trained model to different graphs.

When deciding what form the GNN should take, there are many different types from which to choose. For example, there is the graph-attention network (GAT)[46] which uses self-attention when propagating and is parallelisable, and message-passing neural networks (MPNNs)[16], which first generalised different types of GNN that pass data along edges and combine it in some way. In this specific problem, we need outputs on edges and inputs on nodes, so the model most derive edge features taking into account node features. It is also the case that the amount of demand to and from a specific node can have a global impact, as such a flow may be destined for the most distant vertex from this one. Therefore, the structure will have to be able to take into account global information when setting edge weights. Finally, the structure of the graph in the way its edges connect has a significant impact on how flows can be routed and their resulting impact on utilisation. To properly take advantage of this, our policy must be able to see how information flows and so must make use of the edges. Given these requirements, it makes sense to use one of the most general kinds of GNN so as not to place accidental restrictions on learning.

In keeping with the above observations, we settled on using a fully connected graph network block as defined by Battaglia et al.[7] and shown in section 2.2.2 as this is the most general method achievable with the framework and means that we have not constricted any information flows unnecessarily. We are also aware that constraining the feature vectors for edges and vertices to length one and two respectively, due to the input and output sizes, may make learning less efficient and results worse. Therefore, we expand from the core processing GN block to an 'encode-process-decode' model. This model consists of first mapping all the node features using a learned function to larger hidden size, then performing some number of computation steps by the core network and finally mapping all of the edge features to the correct output size using another learned function. The structure of this policy can be seen in figure 4.2.

Inside the graph network blocks, we use MLPs with learned parameters for all the attribute update ϕ functions (separate MLPs for updating the edge, vertex, and global attributes). For the pooling ρ functions, we use the `tf.unsorted_segment_sum` function to aggregate attributes across vertices and edges.

In summary, the inputs are ingoing and outgoing demands for each vertex:

$$V = \left\{ \left(\sum_{j=1}^{|V|} D_{ij}, \sum_{j=1}^{|V|} D_{ji} \right) \right\}_{i=1:|V|} \tag{4.1}$$

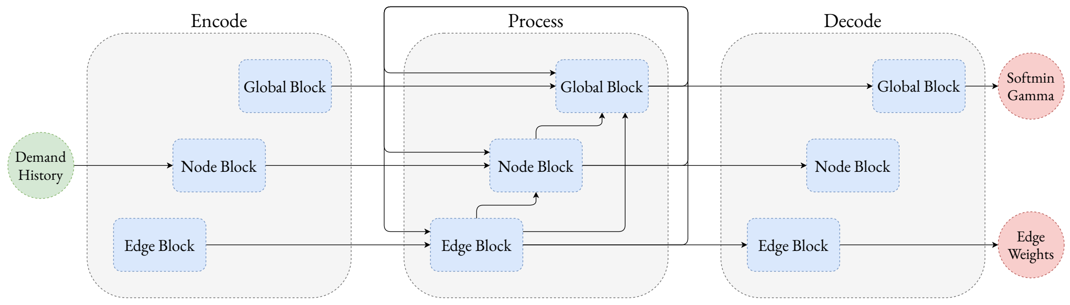

Figure 4.2: The encode-process-decode GNN structure. On the left are the inputs to the policy, followed by the attribute encoding block. Then we have the full graph network block which performs the message passing and the core of the computation. Finally on the right is the attribute decoding block and the policy outputs. We can see an extra loop from output to input on the central block, which is used in the case that we have multiple message-passing steps to carry back the updated state.

and the outputs are a weight for each edge:

$$E = \{w\}_{i=1:|E|} \tag{4.2}$$

4.5 Iterative GNN policy

Given the caveats of the basic GNN policy described above, it is necessary to design a policy that can be trained on different graphs as well as run on different graphs which hopefully leads to better generalisation. For this to work, we need a fixed-size action space. Fortunately, similar problems have been encountered before, so we were able to take inspiration from works such as Placeto[47] to create an iterative RL approach to routing.

In the iterative approach, we only output the value for a single edge as each action, meaning that the network has to be able to output the value for the right edge in the correct action step. The way that this is implemented is by providing extra information in the observation to help with the embedding. In Placeto, Ravichandra et al. use an recurrent neural network (RNN) with a hand-designed method of pooling and message-passing steps to build a graph embedding feature vector that can be passed as input to the RNN in each step. We instead harness the more recent research on GNNs letting a GNN learn the embedding itself. To do this, we modified the simple GNN policy introduced above.

Getting a set of edge weights for a routing strategy from the policy given a demand history takes place in the course of a set of iterations. This is because we can only set one edge value at a time and there are (generally) multiple edges. To enable this, we encode the information of which edge we wish to set on the inputs. As before, the node attribute inputs are the demand history, as shown in equation 4.1—however, this time we also create input attributes for the edges. The input attribute vector per edge becomes the 3-tuple defined in equation 4.3. That is, a value in the interval $[-1, 1]$ denoting the current value for this edge in the routing (and 0 if not set), a binary value from the set $\{0, 1\}$ denoting whether this edge's value has already been set in the routing, and a binary value from the set $\{0, 1\}$ denoting whether this is the edge to be set in the current iteration or not.

$$E = \left\{ (\text{weight}_i, \text{set}_i, \text{target}_i) \right\}_{i=1:|E|} \tag{4.3}$$

These observation inputs are passed into the same encode-process-decode model described in section 4.4. As the output is to be only for one edge, unlike the simpler GNN policy, we read it not from the output edge attributes but the output global attributes. In this policy, the global attribute output is the 2-tuple defined in equation 4.4 where the first element is the value to set on the edge specified and the second is the value of γ to use for the softmin routing (although this is only read on the last iteration of a particular demand history).

$$U = (\text{weight}, \gamma) \tag{4.4}$$

4.5.1 Training

As mentioned previously, for the iterative approach, it takes multiple actions from the agent to assign a single routing. This means we have to rethink how rewards are assigned. The previous approach of comparing to the optimal reward is still used. However, rather than giving rewards for partial routing assignments, we instead give a zero reward at each step and only give the real reward at the end of a set of actions when the routing for a demand history has been fully assigned. This method does work as long as the hyper-parameters of the learning algorithm are tuned because it is especially dependent on the value of γ used for discounting future rewards which allow for sensible policy updates to occur.

Another thing that we had to do was choosing an order to assign edge weights in the iterations. The simplest method would be using some sorted order. However, this could lead to the danger of the policy learning this fixed order and so overfitting and not being able to generalise. To avoid this issue, we always randomise the order that edges are assigned in training.

CHAPTER 5

Evaluation

5.1 Aims

So far, we have described the problem, approaches to a solution, and how to build different RL approaches that seek to solve it. The aim stated at the beginning of this work was to use RL to perform generalisable data-driven routing. Therefore, this evaluation seeks to assess:

1. Are the different policies able to learn to provide a near-optimal routing for a demand?

2. Are the policies able to generalise this technique to unseen sequences of the same type?

3. Do they still generalise to unseen sequences from a different distribution?

4. Are they able to generalise onto different graphs using sequences form the same and different distributions?

5. Are these routing schemes applicable to real-world scenarios?

Over the course of this chapter, these questions will be answered by a sequence of experiments testing all of the stated aims.

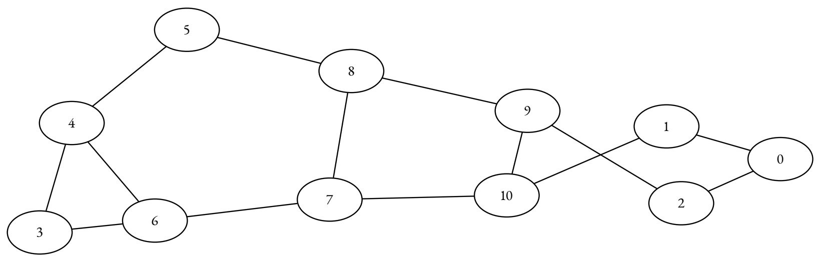

Figure 5.1: Abilene network graph topology.

5.2 Experimental setup

The experiments were all performed (both training and testing models) using the codebase available in the git repository for this project[1]. Following the instructions there, the experiments are fully replicable by running only one script. Prior to running the experiments, the hyperparameters for the learning algorithm were tuned using a custom OpenTuner[3] (which resides in the same repository) program to ensure that performance for all policies was at the highest levels we could achieve.

For the reinforcement learning, we used the implementation of PPO (PPO2) from the stable-baselines[19] library as the learning algorithm with its policy network replaced by the custom policies described in chapter 4. All training and evaluation were performed using this library. The GNN policies themselves were implemented using the Graph Nets[7] library from Deepmind on top of TensorFlow[27]. The entire environment was implemented in Python[45], with a heavy reliance on features from NumPy[29] and NetworkX[18] for numerical computation and working with graphs, respectively.

The hardware used for all the experiments was a single virtual machine running Ubuntu Linux 18.04 with access to 8 CPU cores and an NVIDIA Quadro RTX 8000 GPU.

5.2.1 Graph used

For the majority of the experiments, we used the Abilene graph from the Topology Zoo dataset. We chose this graph because it is relatively small, which reduces learning time while still being large enough to exhibit enough complexity to ensure the results are interesting and useful. Separately, we ran smaller tests on other graphs from the dataset and achieved similar results. The graph itself can be seen in figure 5.1. It has a total of 11 vertices, 28 edges (or 14 bidirectional edges) and is representative of the structure of other AS graphs, which are generally planar and of a similar scale.

[1]`https://github.com/odnh/gnn-routing`

5.2.2 Difficulties

Combining RL libraries with GNN libraries was more difficult than initially anticipated. This is because a GNN can have a variable-sized input and output whilst maintaining a fixed number of trainable variables. Most RL algorithms require a fixed-sized output to function to train correctly. The GNN iterative approach does have a fixed output size (although the non-iterative approach does not). However, for simplicity, most RL libraries assume a fixed-size input to the algorithm. Although PPO is on-policy and the policy itself (a GNN) can take a variable-sized input and return a fixed-size output, most RL libraries do not support this behaviour. Therefore, achieving interoperability between stable-baselines and Graph Nets required extensive work. This was chiefly because Graph Nets maintains the ability to batch while having variable sizes by flattening batches into linear inputs, whilst all RL libraries use an extra tensor dimension which does not allow for variable sizes. Fortunately, a work-around was possible, but it did have a performance impact on the training time.

5.3 Baselines

In order to be able to tell how good the results of the new methods are, we require a comparison. For this comparison, we decided to use two different types of baseline. The first of these is a simple MLP-only policy, as introduced by Valadarsky. We chose this as it is the method that we are seeking to improve upon by adding the ability to generalise over different graphs without impacting performance in predicting good routing strategies. We also add a simple LSTM strategy to examine if the fixed memory length imposed by how we have structured the policies does not overly weaken how the agent interprets and learns from temporal regularity.

The second baseline is to show that the routings the policies learn and produce are in fact useful. In chapter 2, we introduced the idea of an oblivious routing scheme, one which aims to provide a good routing strategy regardless of the particular demands on the network at any one time. As discussed, there is much research in this area with many examples of promising results. However, in the absence of any readily-available open-source code, it was hard to combine any of these complex algorithms with the environment. Therefore, as an oblivious baseline, we have used shortest-path routing as it is the most common method of routing within networks, is easy to implement without errors, and provides reasonable performance.

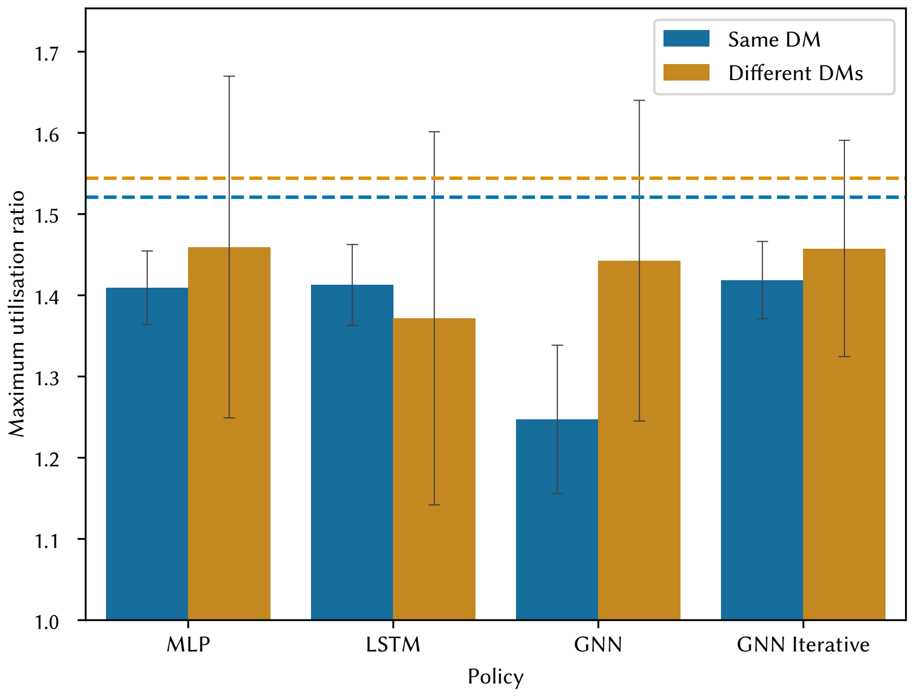

Figure 5.2: Learning to route given a DM. Bar heights are the mean ratio between achieved max-link-utilisation and the optimal for the given DM. Dotted lines across the graph are the ratios achieved by the oblivious routing scheme.

5.4 Experiments

5.4.1 Learning static routing

The first experiment was designed to discover if, given a demand matrix, each of the policies would be able to learn to route the traffic over the graph with a low link utilisation. The reason for this experiment is that the ability to route for a single DM is a prerequisite for predicting good routing strategies for the next step given a demand history, as that is the same problem but with less exact information.

We chose to use the Abilene graph from the Topology Zoo dataset for these experiments, and training was performed using 100 different DMs. The policy history length was one containing the same demand matrix as that being tested. We then performed two different types of test. The first training and test were using a single demand matrix. The second used a set of different demand matrices for training (all 100) and another set of different demand matrices for testing (a different 100). For each experiment, the tests were run 10 times.

The results of this experiment can be seen in figure 5.2. From this, we can see that all policies outperform the baseline, which was expected as the baseline does not allow for multipath routing. We can also see that for all policies, tests over the same single DM used in training perform better than when training and testing on multiple DMs. This result is as expected because we would assume that when using just a single DM, the policies can learn the most optimal routing that they can represent. Then, when we bring multiple DMs into the mix, this means the internal structure cannot be solely focussed on optimising for one DM, so there will be some loss of performance. However, this does also present the issue that there is a problem with generalisation, as if the policies could generalise perfectly, we would expect to see no difference; this is unfortunately not the case. This issue is explored further in section 5.5.

We also see quite wide error bars in this graph. This is a feature that repeats across the results of all the experiments. The error bars themselves represent the standard deviation and the reason that they are so wide is inherent in how the routing strategies are formed. The comparison we are making in these graphs is between the achieved utilisation to the optimal utilisation. However, this optimal utilisation is under a routing that has full control of per-flow splitting ratios, whereas the softmin routing is more constrained. This means that the best achieveable routing for a softmin-based strategy can be further from or closer to the optimal strategy depending on the particular DM being routed, thus leading to wide error bars.

5.4.2 Learning from temporal regularities

As one express aim of this project was to predict good routing strategies to use given a history of traffic information, the next step was to verify if the policies can learn to give good routings for sequences which exhibit some form of temporal regularity. For this, we decided to train and test on cyclical and averaging sequences of bimodal demand matrices and gravity demand matrices with varying sparsity (as defined in section 3.6). These sequences were chosen because they are both are good examples of realistic traffic patterns and the cycles and averaging are strong forms of regularity so are a good test for exploring whether learning to predict routing strategies based on regularity will work.

Again, we used the Abilene graph for these experiments. The sequences used to train on were 100 different cyclic sequences with cycle length 5, total sequence length 10, and the memory length for the policy was 10. For testing, sequences of the same cycle lengths were used but with different demand matrices creating the cycles. Tests were performed for both the bimodal and gravity DMs and over both cyclical and averaging sequences. For each experiment, the tests were run 10 times.

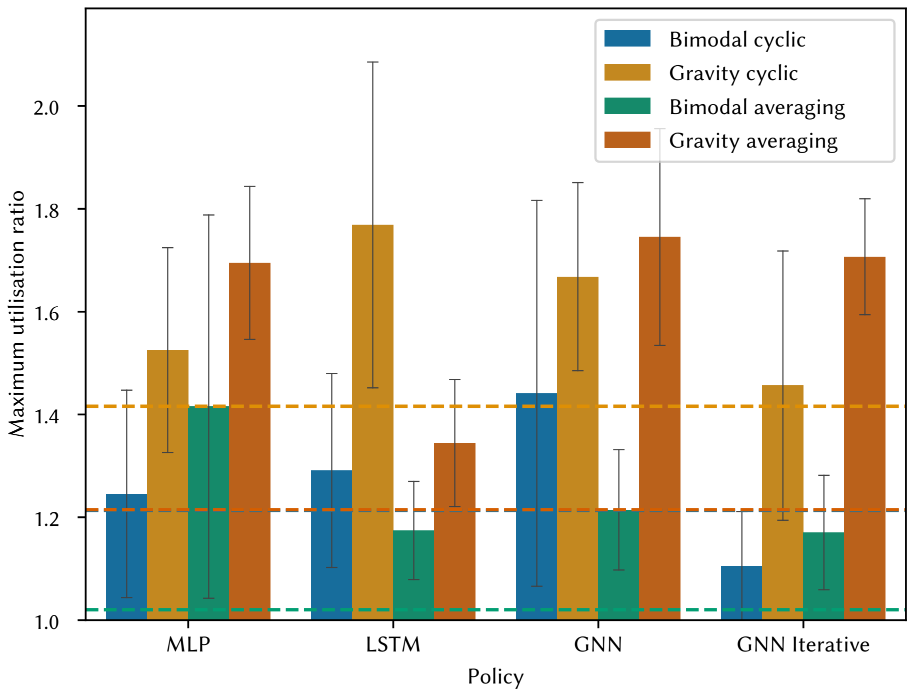

Figure 5.3: Learning to route temporal regularities. Bar heights are the mean ratio between achieved max-link-utilisation and the optimal for the given DM. Dotted lines across the graph are the ratios achieved by the oblivious routing scheme. The blue dotted line is hard to see: underneath and mostly obscured by the orange dotted line.

The results of this experiment can be seen in figure 5.3. From this figure, we can see that unfortunately, all of the policies perform relatively badly. This is likely due to the reasons discussed in detail in section 5.5. Beyond the relative performance of different policies we can draw further conclusions from this plot. The first is that even though, as expected, different types of traffic distribution will cause different levels of link utilisation, it is also the case that our softmin routing representation is able to achieve results closer to the optimum utilisation for some types of traffic than others. This is evident from the fact that the ratio between the achieved utilisation and optimal utilisation is always less for the sequences built using bimodal DMs than those built using sparsified gravity DMs for all policies. On the other hand, the type of regularity induced (a cyclic or averaging sequence) does not seem to have the same kind of impact, as it is not the case that all the policies perform better for one type of sequence than the other. However, this may be due to the reasons discussed in section 5.5.

5.4.3 Generalising to different cycle lengths

The paper by Valadarsky et al. looked into learning regularity but only when using a fixed cycle length. We sought to examine whether the policies can, in fact, learn to generalise to different cycle lengths or not. To perform this, we trained on 100 cyclical bimodal sequences with different cycle lengths (2, 3, 4 and 5) and then tested on a different 100 sequences generated in the same way. For each experiment, the tests were run 10 times.

The results of this experiment can be seen in figure 5.4. The first point we can notice is that all of the policies outperform the baseline, which is positive as it shows they are an improvement. Also interesting to note is that the mean ratio for all policies seems to be the same, apart from the LSTM policy. This suggests that, as expected, the structure of history for the non-LSTM policies is too rigid and therefore structured in such a way that makes it hard for them to learn to deal with different lengths of cyclical regularity. The LSTM policy benefits from the fact that it has no specific history length assumed in its design, meaning that it may be more flexible to interpreting different types of regularity (which is what we seem to be seeing here). This suggests that LSTMs would be interesting to combine with other techniques in further research.

5.4.4 Generalising to unseen graphs

As the second express aim of this work is to provide a policy that could generalise across different network graphs, it is crucial to test if this worked successfully. For this experiment, we anticipate two different types of graph generalisation. The first is generalising to graphs that are the same but with minor changes, as we would still want the policy to

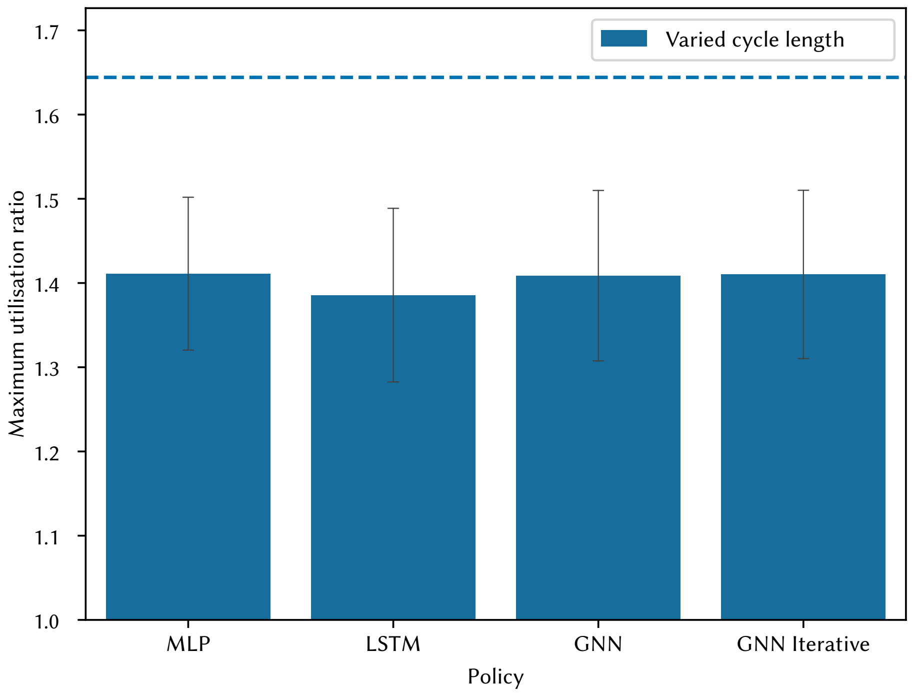

Figure 5.4: Generalising to different cycle lengths. Bar heights are the mean ratio between achieved max-link-utilisation and the optimal for the given DM. Dotted lines across the graph are the ratios achieved by the oblivious routing scheme.

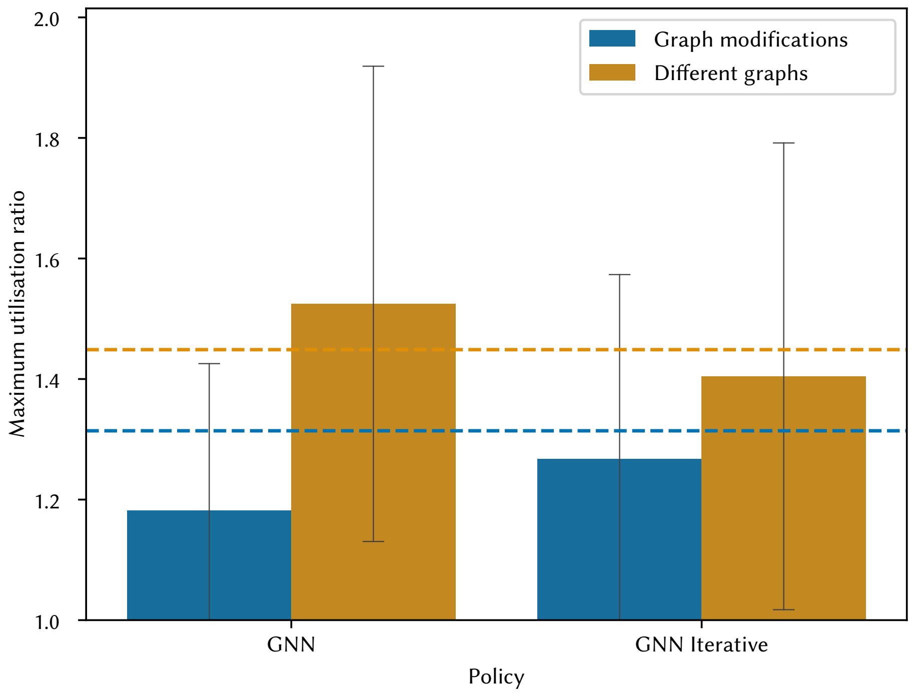

Figure 5.5: Generalising to unseen graphs. Bar heights are the mean ratio between achieved max-link-utilisation and the optimal for the given DM. Dotted lines across the graph are the ratios achieved by the oblivious routing scheme.

work on a network where, for example, an outage has caused a link or node to drop temporarily. Furthermore, adding a new node or link should similarly not cause the policy to misbehave. The second type of generalisation is to graphs that are unrelated to the original graph, for example, all drawn from different generating sets.

For this experiment, we test both types of generalisation. For the first, we train on the Abilene graph with a subset of vertices and edges randomly dropped and added (5 different modifications), and testing is performed using the same graph but a different set of small modifications (5 different modifications again). For the second type of generalisation, we train on a set of 5 graphs from the Topology Zoo dataset and test on a different set of 5 graphs from the same dataset. The tests were each run 10 times.

The results of this experiment can be seen in figure 5.5. We can see that in most cases, the policies perform better than the baseline. In fact, the GNN and iterative policies both perform better for small graph modifications, but only the iterative method performs better in the case of graphs that are very different from each other. The reason for this performance difference between the iterative and non-iterative methods is likely due to how the learning algorithm works. As the output size of the policies is fixed, this means that for smaller graphs for the GNN policy, some unused outputs had to be masked. However, the learning algorithm does not know this and can still use those outputs when updating the policy, which can lead to issues such as the lower performance we see here. The likely reason that the GNN policy performs better for more similar graphs is that the above issue does not cause a significant enough impact and the policy is simpler, therefore requiring less training and being more able to represent the complex relationships required easily.

5.4.5 Real world

Finally, as the problem we are attempting to solve is a real-world problem, it is essential to see if we can achieve any meaningful performance on a real-world dataset. Fortunately, there is a large dataset of real traffic matrices available from the Totem[43] project. Unfortunately, the graph is much larger than in previous experiments and the time period that each measurement is valid for is relatively short, meaning that the memory length for the policies had to be extended. However, we still proceeded with the experiment.

We trained on 5 sequences of length 20 with a memory length of 10 for the policies. Testing was performed in much the same way as for previous experiments, with the test sequences selected randomly from the dataset but being different sequences to those used for training. Each test was run 10 times.

The results of this experiment can be seen in figure 5.6. As with previous experiments, we can see that the policies seem to outperform the baseline. However, both the MLP and LSTM policies have better performance than the GNN policies. This better performance is likely down to reasons described in the next section and the small size of the learning set (which was restricted as the TOTEM graph was significantly larger than other graphs tested on, which greatly impacted computation time for calculating real and optimal link utilisation statistics) which result in a static oblivious routing being learned. It is probable that the oblivious routing learned by the two GNN policies was not as effective or general as that learned by the other two policies. However, further analysis and experimentation would be required to prove if this is the case.

5.5 Learning issues

From the results, we have seen that the RL policies generally do seem to outperform the baseline. However, it should be noted that as the baseline is shortest-path routing, this shows that the policies are learning to reduce max-link-utilisation but not how well they are doing this task compared to best-in-class oblivious routing strategies. In fact, upon further investigation, it seems that the main issue is that, given a high enough number of

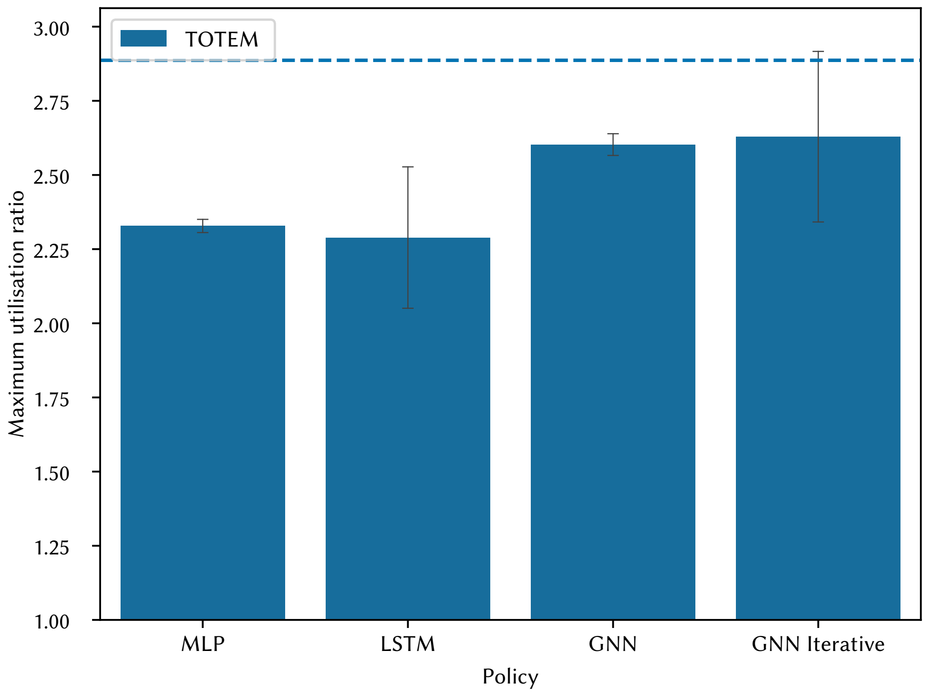

Figure 5.6: Learning to route from TOTEM dataset. Bar heights are the mean ratio between achieved max-link-utilisation and the optimal for the given DM. Dotted line across the graph is the ratio achieved by the oblivious routing scheme.

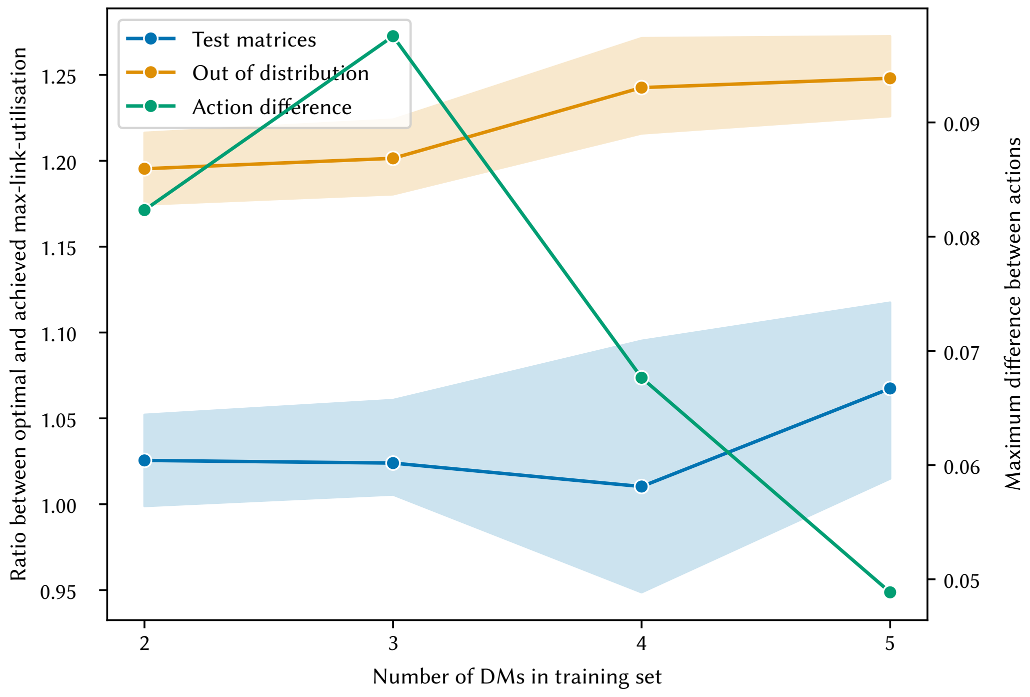

Figure 5.7: Exploring when learning collapses to a single value. Right-hand axis corresponds to the green points, and left-hand axis corresponds to the others. Blue plot is the result of testing models on DMs from the training set, whereas the orange line is from DMs outside of the training set. Error area is the standard deviation.

different sequences to train on, all the policies learn a fairly good oblivious routing rather than reacting to their inputs.

To explore the point where overfitting particular DMs turns into providing a static oblivious routing scheme, we undertook a few more experiments. We began by revisiting the experiment discussed in section 5.4.1. We performed this experiment for the MLP policy varying, the number of DMs in the training set and testing on those same DMs.

In figure 5.7, we can see both the overall performance of the trained models as the size of their training sets increases and the maximum difference between edge weights in different actions they output. From this we can gather two things: first that the performance gets worse the more different DMs we have to look at in training (for both training and when using the trained model) and second, that the variance decreases until the point that whatever the input, the output remains effectively the same. The error area does widen for the training set plot, but this is simply an effect of adding more DMs, which all have different relations to the optimal routing from each other.

The reason this occurs is probably due to how the learning is trading off between two extremes. In effect, it could either learn to provide an optimal routing for a predicted DM or just provide an optimal oblivious routing. The optimal routing for the DM is preferable, but if it gets the DM prediction wrong, then the result could be worse for link utilisation than the oblivious scheme which is guaranteed to behave well no matter the actual DM in the next timestep. The overfitting we are seeing is likely where the policy cannot find a good middle ground, and therefore an oblivious strategy performs better than a bad prediction, so that is what the learning converges on.

5.6 Discussion

We have seen various results in this chapter from experiments that sought to assess generalisability of the different policies. Overall, we have learnt that all the policies are able to outperform the baseline. We have also found that the policies struggle to fit the data well, in almost all cases overfitting as the size of the set of demands to route grows in the training set. This is unfortunate as it suggests that these policies are not structured in such a way that allows them to learn this information or that a different technique should be used to aid learning. However, even with this overfitting, it does seem to be the case that the graph network-based policies are still able to generalise to other graphs well which is positive as it is something we sought to show; even if what they are generalising is an oblivious routing as opposed to a strategy informed by demand histories.

In examining where overfitting occurs, it seems to be the case that either none of the policies are able to represent the relations required, or that the RL algorithm used in tandem with the construction of training sets was not able to explore the trajectories available to a sufficiently wide degree to find the best parameters for the models. When designing these experiments and the policies themselves, we tried many variations in the policy network architectures for all policy types and, as mentioned previously, we performed hyperparameter tuning with the aim of improving results. However, even with these measures, we were, unfortunately, unable to improve results further.

Overall, we have seen that the policies are able to fit small numbers of demand matrices and that graph neural networks are able to successfully generalise to different input graphs. This suggests that it is a viable area for further research, especially in using GNNs, but that different techniques should be explored in relation to the learning.

CHAPTER 6

Conclusions and future work

6.1 Conclusions

In this work, we have sought to provide a generalisable approach to data-driven routing using deep RL with GNNs. This has consisted of providing a detailed specification of the problem and presenting an environment which can be used to experiment with different techniques to ascertain how well they perform. We also took the approach introduced by Valadarsky et al. and modified it extensively so that it both worked to spread traffic across multipath routes and so that it could work with GNNs. In addition, we designed GNN policy architectures that could be used in conjunction with this problem and more generally provided a connection between an RL library and variable-sized graph network models.

After presenting previous work, as well as our new additions, we performed extensive experiments on the different policy architectures in different scenarios to assess how well they generalise to different demand matrices, followed by different kinds of regularity in demand sequences, before finally looking at generalisation to different network topologies. These experiments showed that all the policies can learn to efficiently route a demand matrix, in fact, multiple. They also showed that GNNs can generalise learned routings. Unfortunately, they showed that we were unable to make any of the policies learn to route demand matrices based on a history of previous demands beyond providing a good oblivious routing that did a good job of minimising the utility function (in this case maximum link utilisation).

Overall, we have presented new techniques and a new library, accompanied by a mixture of successes and failures in evaluation. Fortunately, we can draw on these failures to inspire new work.

6.2 Future work

As previous work has suggested the approaches taken in this paper can be successful, we feel it is important for these approaches to be explored further. One core element of the approach taken focussed on designing a good way to reduce the size of the action space. An important part of this was in designing a mapping from edge weights to a routing strategy. We feel that an exploration of different techniques in mapping edge weights, or indeed any intermediate structure, to a routing strategy could provide interesting results. Furthermore, the way that demand information is input to the network could be greatly improved through numerous techniques and possibly hierarchical RL would be an interesting area to explore here.

Beyond mappings of observations and outputs, there are many different ways in which the learning itself could be modified. These include using different learning algorithms to PPO and employing different techniques for graph generalisation, such as the iterative approach. Another aspect that could be looked into here are modifications to the reward function with different properties that could aid exploration. Finally, we saw some successes with varied sequence length from the LSTM approach so if this could be combined with a method allowing for generalisation to different graphs it could lead to an approach achieving the best of both worlds.

Assuming the success of the techniques presented here, this work can be expanded to explore optimisations of routing to perform different goals, such as in changing the utility function. A natural next step would be implementing these strategies in real-world SDN systems so that they could be tested and used on real-world networks. This would present new challenges, mainly centred around managing incomplete information and distributing control.

We did achieve considerable success with the generalisation ability of GNNs which suggests that outside of the area of data-driven routing there may be other useful application to explore. Other aspects of network control have been investigated with machine learning (ML) and particularly RL techniques such as resource scheduling and rate controls which may benefit greatly from the application of GNN-based techniques.

List of Abbreviations

API	application programming interface
AS	autonomous system
BFS	breadth-first search
CNN	convolutional neural network
DAG	directed acyclic graph
DM	demand matrix
DODAG	destination-oriented directed acyclic graphs
GAT	graph-attention network
GN	graph network
GNN	graph neural network
ISP	internet service provider
LP	linear programming
LSTM	long short-term memory
MDP	markov decision process
ML	machine learning
MLP	multilayer perceptron
MPNN	message-passing neural network
OSPF	open shortest path first
PPO	proximal policy optimisation
RIP	routing information protocol
RL	reinforcement learning
RLP	routing protocol for low-powered and lossy networks
RNN	recurrent neural network
SDN	software-defined networking

Bibliography

[1] Roger Alexander et al. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. Mar. 2012. DOI: 10.17487/RFC6550. URL: https://rfc-editor.org/rfc/rfc6550.txt.

[2] Ramy E. Ali et al. 'Hierarchical Deep Double Q-Routing'. In: *ArXiv* abs/1910.04041 (2019).

[3] Jason Ansel et al. 'Opentuner: An extensible framework for program autotuning'. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 303–316.

[4] Yossi Azar et al. 'Optimal oblivious routing in polynomial time'. In: *Journal of Computer and System Sciences* 69.3 (2004), pp. 383–394.

[5] Nikhil Bansal. 'Oblivious Routing'. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, 2008, pp. 585–588. ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_261. URL: https://doi.org/10.1007/978-0-387-30162-4_261.

[6] Cynthia Barnhart, Niranjan Krishnan and Pamela H. Vance. 'Multicommodity flow problemsMulticommodity Flow Problems'. In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos M. Pardalos. Boston, MA: Springer US, 2001, pp. 1583–1591. ISBN: 978-0-306-48332-5. DOI: 10.1007/0-306-48332-7_316. URL: https://doi.org/10.1007/0-306-48332-7_316.

[7] Peter W. Battaglia et al. *Relational inductive biases, deep learning, and graph networks*. 2018. arXiv: 1806.01261 [cs.LG].

[8] Kamal Benzekki, Abdeslam El Fergougui and Abdelbaki Elbelrhiti Elalaoui. 'Software-defined networking (SDN): a survey'. In: *Security and Communication Networks* 9.18 (2016), pp. 5803–5833. DOI: 10.1002/sec.1737. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1737. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1737.

[9] Justin A Boyan and Michael L Littman. 'Packet routing in dynamically changing networks: A reinforcement learning approach'. In: *Advances in neural information processing systems*. 1994, pp. 671–678.

[10] Greg Brockman et al. 'Openai gym'. In: *arXiv preprint arXiv:1606.01540* (2016).

[11] Thomas H Cormen et al. *Introduction to algorithms*. 2009. Chap. 29.

[12] Dennis Ferguson, Acee Lindem and John Moy. *OSPF for IPv6*. RFC 5340. July 2008. DOI: 10.17487/RFC5340. URL: https://rfc-editor.org/rfc/rfc5340.txt.

[13] Bernard Fortz and Mikkel Thorup. 'Optimizing OSPF/IS-IS weights in a changing world'. In: *IEEE journal on selected areas in communications* 20.4 (2002), pp. 756–767.

[14] Vincent François-Lavet et al. 'An Introduction to Deep Reinforcement Learning'. In: *Foundations and Trends® in Machine Learning* 11.3-4 (2018), pp. 219–354. issn: 1935-8237. doi: `10.1561/2200000071`. url: `http://dx.doi.org/10.1561/2200000071`.

[15] Andrey V Gavrilov and Artem Lenskiy. 'Mobile robot navigation using reinforcement learning based on neural network with short term memory'. In: *International Conference on Intelligent Computing*. Springer. 2011, pp. 210–217.

[16] Justin Gilmer et al. 'Neural message passing for quantum chemistry'. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1263–1272.

[17] Marco Gori, Gabriele Monfardini and Franco Scarselli. 'A new model for learning in graph domains'. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. IEEE. 2005, pp. 729–734.

[18] Aric Hagberg, Pieter Swart and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[19] Ashley Hill et al. *Stable Baselines*. `https://github.com/hill-a/stable-baselines`. 2018.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 'Long short-term memory'. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[21] Oana Iova, Fabrice Theoleyre and Thomas Noel. 'Using multiparent routing in RPL to increase the stability and the lifetime of the network'. In: *Ad Hoc Networks* 29 (2015), pp. 45–62.

[22] S. Knight et al. 'The Internet Topology Zoo'. In: *Selected Areas in Communications, IEEE Journal on* 29.9 (Oct. 2011), pp. 1765–1775. issn: 0733-8716. doi: `10.1109/JSAC.2011.111002`.

[23] M Kodialam, TV Lakshman and Sudipta Sengupta. 'Advances in oblivious routing of internet traffic'. In: *Performance Modeling and Engineering*. Springer, 2008, pp. 125–146.

[24] Praveen Kumar et al. 'Semi-oblivious traffic engineering: The road not taken'. In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, pp. 157–170.

[25] Nguyen Cong Luong et al. 'Applications of deep reinforcement learning in communications and networking: A survey'. In: *IEEE Communications Surveys & Tutorials* 21.4 (2019), pp. 3133–3174.

[26] Gary S. Malkin and Robert E. Minnear. *RIPng for IPv6*. RFC 2080. Jan. 1997. doi: `10.17487/RFC2080`. url: `https://rfc-editor.org/rfc/rfc2080.txt`.

[27] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. url: `https://www.tensorflow.org/`.

[28] Alberto Medina et al. 'Traffic matrix estimation: Existing techniques and new directions'. In: *ACM SIGCOMM Computer Communication Review* 32.4 (2002), pp. 161–174.

[29] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[30] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: `1912.06680 [cs.LG]`.

[31] Laurent Perron and Vincent Furnon. *OR-Tools*. Version 7.2. Google, July 2019. URL: `https://developers.google.com/optimization/`.

[32] Harald Racke. 'Minimizing congestion in general networks'. In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* IEEE. 2002, pp. 43–52.

[33] Matthew Roughan et al. 'Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning'. In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. 2002, pp. 91–92.

[34] David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. 'Learning representations by back-propagating errors'. In: *nature* 323.6088 (1986), pp. 533–536.

[35] Kaku Sawada, Daisuke Kotani and Yasuo Okabe. 'Network Routing Optimization Based on Machine Learning Using Graph Networks Robust against Topology Change'. In: *2020 International Conference on Information Networking (ICOIN)* (2020), pp. 608–615.

[36] Franco Scarselli et al. 'The graph neural network model'. In: *IEEE Transactions on Neural Networks* 20.1 (2008), pp. 61–80.

[37] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: `1707.06347 [cs.LG]`.

[38] Lei Shi et al. 'DDRP: an efficient data-driven routing protocol for wireless sensor networks with mobile sinks'. In: *International Journal of Communication Systems* 26.10 (2013), pp. 1341–1355.

[39] David Silver et al. 'A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play'. In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: `10.1126/science.aar6404`. eprint: `https://science.sciencemag.org/content/362/6419/1140.full.pdf`. URL: `https://science.sciencemag.org/content/362/6419/1140`.

[40] Giorgio Stampa et al. 'A deep-reinforcement learning approach for software-defined networking routing optimization'. In: *arXiv preprint arXiv:1709.07080* (2017).

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

[42] Gerald Tesauro. 'Temporal difference learning and TD-Gammon'. In: (1995).

[43] Steve Uhlig et al. 'Providing public intradomain traffic matrices to the research community'. In: *ACM SIGCOMM Computer Communication Review* 36.1 (2006), pp. 83–86.

[44] Asaf Valadarsky et al. 'Learning to route with deep rl'. In: *NIPS Deep Reinforcement Learning Symposium*. 2017.

[45] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[46] Petar Veličković et al. 'Graph attention networks'. In: *arXiv preprint arXiv:1710.10903* (2017).

[47] Shaileshh Bojja Venkatakrishnan et al. ‘Learning Generalizable Device Placement Algorithms for Distributed Machine Learning’. In: *Advances in Neural Information Processing Systems*. 2019, pp. 3983–3993.

[48] Ronald J Williams. ‘Simple statistical gradient-following algorithms for connectionist reinforcement learning’. In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[49] Zonghan Wu et al. ‘A Comprehensive Survey on Graph Neural Networks’. In: *IEEE Transactions on Neural Networks and Learning Systems* (2020), pp. 1–21. ISSN: 2162-2388. DOI: `10.1109/tnnls.2020.2978386`. URL: `http://dx.doi.org/10.1109/TNNLS.2020.2978386`.

[50] Xinyu You et al. ‘Toward packet routing with fully-distributed multi-agent deep reinforcement learning’. In: *arXiv preprint arXiv:1905.03494* (2019).

[51] Jie Zhou et al. *Graph Neural Networks: A Review of Methods and Applications*. 2018. arXiv: `1812.08434 [cs.LG]`.