Neural Operators for Efficient and Adaptive Network Traffic Optimization

Ardasher Nasriddinov
Department of Computer Science
Northwestern University
Evanston, IL, USA
ardasher@u.northwestern.edu

Abstract

Optimizing network traffic flow to minimize congestion is a fundamental challenge in communication networks. Traditional traffic engineering (TE) methods rely on solving optimization problems (e.g., linear programs) to distribute traffic and minimize the maximum link utilization (congestion). However, these methods can be computationally expensive and may not adapt quickly to changing traffic. In this work, we explore neural operator approaches for efficient and adaptive network traffic optimization. We generate a dataset of traffic demand matrices on a fixed backbone network (the 12-node Abilene network) and obtain optimal routing solutions via a min-max utilization solver. We then train two deep learning models to predict the per-link utilization under the optimal routing for any given traffic demand: (1) a Graph Convolutional Network (GCN) that uses spatial graph convolutions, and (2) a Graph Fourier Neural Operator (GFNO) that uses spectral graph convolutions (in the eigenbasis of the network Laplacian) to learn a global mapping from demands to network utilization. Our results show that the GFNO achieves significantly lower prediction error than the GCN, while converging faster, demonstrating the advantage of spectral neural operators in capturing global traffic patterns. This approach can produce near-optimal routing solutions in real-time once trained, enabling adaptive traffic optimization without repeatedly solving complex optimization problems. We discuss the methodology for dataset generation, model architectures, training procedure, and comparative evaluation, highlighting that neural operators offer a promising direction for efficient and adaptive network traffic engineering. Full code is available at: https://github.com/ArdasherN/497project.git

1 Introduction

Modern communication networks must handle ever-increasing demand driven by cloud computing, video streaming, real-time collaboration, and mobile edge applications. As traffic continues to grow, it becomes increasingly important to ensure that the underlying infrastructure is used efficiently. Traffic Engineering (TE) addresses this challenge by optimizing how data flows are routed through the network to meet performance objectives such as minimizing delay, avoiding link overloads, and maximizing throughput. One of the most common performance goals in TE is to minimize the maximum link utilization (MLU), since over-utilized links can lead to congestion, packet loss, and latency spikes [1].

Traditionally, TE problems are solved using optimization techniques such as multi-commodity flow formulations, where each traffic demand between a source and destination is modeled as a commodity. A linear program (LP) is used to minimize the worst-case (maximum) link utilization by

Preprint. Under review.

solving for the optimal flow distribution subject to capacity and conservation constraints [2]. While this method produces optimal solutions under fractional path splitting, solving an LP for each new traffic matrix is computationally expensive. In large networks or scenarios with frequently changing traffic demands, this can be too slow for real-time deployment. Although Software-Defined Networking (SDN) enables centralized traffic management and dynamic reconfiguration, the underlying optimization bottleneck remains, limiting how quickly and effectively network control can respond to changing conditions.

In response to this limitation, a growing body of work has explored machine learning-based TE, aiming to bypass traditional solvers by learning to approximate the optimization behavior. Early research such as RouteNet showed that Graph Neural Networks (GNNs) can learn to estimate key performance metrics like delay and jitter by modeling the network as a graph and applying message passing on nodes and edges [3]. Follow-up work has applied GNNs in combination with Deep Reinforcement Learning (DRL) to learn adaptive routing policies that adjust to current traffic loads and link states [4]. More recently, large-scale systems like Teal have demonstrated that learned models can match or exceed traditional optimizers in speed and scalability, using parallel GPU execution to make routing decisions on networks with thousands of nodes [5]. Other approaches, such as HARP, have focused on generalizing learned routing policies to unseen topologies, a key requirement for real-world deployment in evolving networks [6].

While GNN-based TE methods have shown strong results, they also come with challenges. Because message passing in GNNs is local by nature, capturing long-range dependencies across the network often requires many layers and significant tuning. Additionally, many GNN models are trained for a single topology or rely on careful encoding of structural information, which can limit generalization and robustness in dynamic environments. For example, a GNN trained on one set of link capacities or traffic conditions may not perform well if the topology experiences a temporary failure or sudden load spike.

To address these limitations, we propose exploring neural operator models, which aim to learn mappings between functions instead of fixed-length vectors. This makes them especially well-suited for generalizing across different input distributions and capturing global structure. In particular, the Fourier Neural Operator (FNO) has recently been introduced for learning solutions to families of partial differential equations by operating in the frequency domain, and has demonstrated strong performance in tasks that require learning long-range dependencies [7]. Variants of FNOs have also been applied to model traffic flow in transportation systems, showing promise for handling irregular, dynamic patterns [8].

Building on this idea, we investigate a Graph Fourier Neural Operator (GFNO) for traffic engineering. The GFNO works in the spectral domain of the network graph, using the eigenvectors of the graph Laplacian as a basis for learned global convolutions. This allows the model to learn global relationships between traffic demand and link utilization in a single operation, rather than relying on local message passing. By leveraging the full structure of the graph, the GFNO has the potential to be more resilient to temporary disruptions such as sudden demand spikes or link failures, and may generalize better to a variety of traffic conditions.

To evaluate this approach, we focus on the task of predicting per-link utilization for different traffic demand matrices on a fixed network topology. This is a common operational setting where the physical topology remains stable, but traffic patterns vary over time. Our goal is to learn a model that, given any demand matrix on the given topology, can quickly and accurately predict the corresponding link utilizations that would result from an optimal routing solution. As a baseline, we compare the GFNO to a standard Graph Convolutional Network (GCN) architecture that performs spatial graph convolutions using local message passing [9]. Both models take the same inputs and are trained using supervised learning on examples generated by solving the LP-based min-max utilization problem.

The rest of this paper is organized as follows. In Section 2, we describe the dataset generation process, the LP formulation used to compute optimal solutions, and the architecture of both neural models. Section 3 outlines our training procedure, hyperparameter settings, and evaluation metrics. Section 4 presents experimental results, including model accuracy, convergence behavior, and a discussion of the trade-offs between approaches. Finally, Section 5 concludes with limitations, practical implications, and directions for future work.

2 Methodology

2.1 Dataset and Topology Overview

Figure 1: Abilene network topology used for dataset generation. Source: SNDlib [11].

To study the traffic optimization problem in a controlled environment, we fix a single network topology and generate multiple traffic demand scenarios based on it. The topology we use is the Internet2 Abilene backbone, a well-known reference network used in research. The Abilene network includes 12 nodes, which represent major points of presence (PoPs) across the United States, and 28 directed links. Each physical link is bidirectional, but we treat them as directed edges to simplify modeling. All links are assumed to have the same capacity. In our dataset, we use 1000 Mbps per directed link, but since the model learns relative usage, any uniform capacity would work.

Traffic demand is represented using traffic matrices (TMs). A traffic matrix is an 12×12 matrix, where the entry at position (i, j) indicates the traffic demand, in Mbps, from node i to node j. We generate synthetic TMs using a gravity model. In this model, each node is assigned a random weight, and the demand between two nodes is set to be proportional to the product of their weights. This captures the intuition that nodes with more "activity" generate and attract more traffic.

To generate each traffic matrix, we assign each node a random weight drawn from a uniform distribution between 0 and 1. For each source–destination pair (s, d), we compute the initial demand as the product of their weights. After all demands are calculated, we scale the entire matrix so that the total demand sums to a fixed value, such as 10,000 Mbps. We also set all diagonal entries to zero, since we do not model self-demands (i.e., a node sending traffic to itself). This method results in a diverse set of traffic patterns, where some nodes consistently act as major sources or destinations. Using this method, we generated 100 different traffic matrices for the Abilene network, using different random seeds for variability.

To represent the network topology for input to our machine learning models, we use NetworkX to load the graph from a GraphML file. The Abilene topology is publicly available in the Internet Topology Zoo dataset. This gives us a directed graph with node and edge lists. When available, we also extract geographic coordinates for the nodes from the dataset. These can be included as input features to help the model understand the spatial structure of the network. If coordinates are not included, a graph layout algorithm like Kamada–Kawai can be used to assign positions.

Each example in our dataset consists of a traffic matrix and the corresponding network graph. For each of these scenarios, we compute the optimal per-link utilization using a solver, which we describe in the next section.

2.2 Traditional Solver for Min–Max Utilization

For each traffic matrix, we compute the optimal routing that minimizes the maximum utilization across all links. This problem is known as the Min-Max Traffic Engineering problem. We assume traffic can be split over multiple paths (multi-path routing), and the goal is to spread traffic so that no single link becomes a bottleneck. We solve this problem using a linear program (LP).

The LP includes the following decision variables: for every demand between a source node s and a destination node d, and for every directed edge (i, j) in the network, we define a flow variable $f_{s,d}(i, j)$, which represents how much traffic from s to d is routed through edge (i, j). We also define a special variable t, which represents the maximum utilization across all links. The solver's objective is to minimize t.

The LP includes several sets of constraints:

Capacity constraints: For each link (i, j), the total flow over that link (from all source–destination pairs) must not exceed t multiplied by the link's capacity. This ensures that t is at least as large as the highest utilization across the network. Formally:

$$\sum_{s,d} f_{s,d}(i, j) \;\leq\; t \times C_{ij},$$

where C_{ij} is the capacity of link (i, j).

Flow conservation constraints: For each demand (s, d), the flow must be conserved across intermediate nodes. This means:

- At the source node s, total outgoing flow minus incoming flow should equal the demand $D_{s,d}$.
- At the destination node d, incoming flow minus outgoing flow should equal $D_{s,d}$.
- At any other node v, incoming and outgoing flows must be equal.

Non-negativity constraints: All flow values must be greater than or equal to zero. The variable t must also be non-negative.

The LP solver minimizes t while satisfying all these constraints, resulting in a set of flow values that represent the optimal routing and the smallest possible maximum link utilization. We implement this solver using the PuLP library with the CBC backend, which is a free and open-source LP solver.

Once the LP is solved for a given traffic matrix, we compute the final utilization for each link as the total flow on that link divided by its capacity. These per-link utilizations are saved as the ground-truth outputs (labels) for our machine learning models.

While this LP is polynomial-time solvable and runs quickly for small networks like Abilene, it becomes expensive for larger networks or for use in real-time systems where traffic changes often. The number of variables grows with the number of demands and links (approximately $|E| \times |V|^2$), which adds to the computation time. For this reason, our goal is to train a model offline that learns to approximate the LP solver's outputs, so that at runtime we can predict near-optimal link utilizations in milliseconds using a forward pass through a neural network.

2.3 GNN Architecture (GCN Baseline)

Our first learning-based model is a Graph Convolutional Network (GCN), which serves as a baseline. A GCN is a type of Graph Neural Network (GNN) that performs spatial graph convolutions by aggregating information from each node's neighbors in the graph. After a few layers, each node's representation captures information not just about itself but also about its surrounding neighborhood, including the structure and the traffic entering and leaving nearby nodes [1].

Input Features To use a GCN, we must convert the current state of the network (topology and traffic demand) into node-level features. In our design, we use four features for each node:

- **Outgoing traffic sum:** For node v, this is the sum of all traffic it needs to send to other nodes. It is computed as the row sum of the traffic matrix:
$$\mathrm{Out}(v) = \sum_{d} T[v, d]$$
- **Incoming traffic sum:** For node v, this is the sum of all traffic destined for it, calculated as the column sum of the traffic matrix:
$$\mathrm{In}(v) = \sum_{s} T[s, v]$$

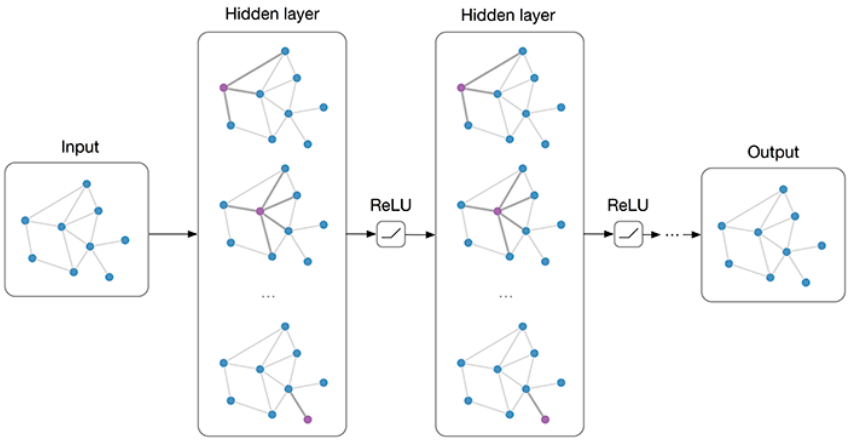

Figure 2: Illustration of a two-layer Graph Convolutional Network (GCN) processing a graph input through successive neighborhood-aggregation layers. Reprinted from Thomas Kipf's GCN tutorial [12].

- **Coordinate x:** This is the first coordinate value for the node, based on geographic or layout coordinates provided by the topology file.
- **Coordinate y:** This is the second coordinate value.

All features are normalized to the $[0, 1]$ range. Traffic features are divided by a constant (like total traffic) and node coordinates are min–max normalized. This results in a feature matrix $\mathbf{X}$ of size $N \times 4$, where N is the number of nodes (12 in the Abilene network).

GCN Layers We use a two-layer GCN as proposed by Kipf and Welling. The basic GCN layer formula is:

$$\mathbf{H}^{(l+1)} = \mathrm{ReLU}\big(\hat{D}^{-\frac{1}{2}}\,\hat{A}\,\hat{D}^{-\frac{1}{2}}\,\mathbf{H}^{(l)}\,W^{(l)}\big)$$

Here:

- $\hat{A} = A + I$ is the adjacency matrix with added self-loops.
- $\hat{D}$ is the diagonal degree matrix of $\hat{A}$.
- $\mathbf{H}^{(l)}$ is the node embedding matrix at layer l (with $\mathbf{H}^{(0)} = \mathbf{X}$).
- $W^{(l)}$ is the trainable weight matrix for layer l.

We precompute the normalized adjacency $\hat{D}^{-\frac{1}{2}}\hat{A}\,\hat{D}^{-\frac{1}{2}}$ (denoted $\mathbf{A}_{\mathrm{norm}}$) and use it as a fixed operation.

- **Layer 1:** Transforms the 4-dimensional input to a 64-dimensional hidden space via a 4×64 weight matrix, followed by ReLU and dropout (rate 0.3).
- **Layer 2:** Processes the 64-dimensional hidden features with a 64×64 weight matrix. No non-linearity is applied here; the output is the final embedding $\mathbf{H}$ of size $N \times 64$.

After two layers, each node's embedding contains information about nodes up to two hops away. In the Abilene topology, this often covers a large portion of the network, but the GCN still primarily captures local structure.

Edge Prediction Head To predict utilization for each directed link, we concatenate the embeddings of its endpoints. For edge (u, v):

$$\mathbf{z}_{uv} = \big[\mathbf{h}_u \,\|\, \mathbf{h}_v\big] \in \mathbb{R}^{128}$$

This is passed through an MLP:

- Dense: $128 \to 64$ with ReLU
- Output: $64 \to 1$ (linear)

The model is trained with mean squared error loss between predicted and true per-link utilizations.

2.4 GFNO Architecture (Graph Fourier Neural Operator)

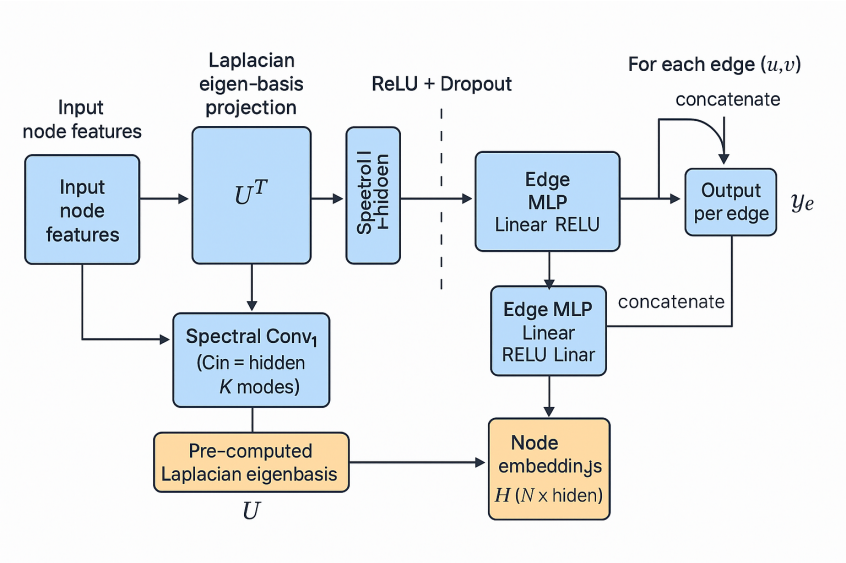

Figure 3: Architecture of the Graph Fourier Neural Operator (GFNO), showing the spectral convolution layers using the Laplacian eigenbasis U, followed by edge-wise MLP prediction.

Our second model is the Graph Fourier Neural Operator (GFNO), which builds on the Fourier Neural Operator concept [2]. The original FNO was designed for Euclidean grids with standard Fourier transforms. On graphs, we use the Laplacian's eigenvectors as a Fourier-like basis to learn global mappings.

Graph Fourier Basis Compute the normalized graph Laplacian:

$$L = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}},$$

where A is the undirected adjacency (without self-loops) and D the degree matrix. Eigendecompose:

$$L = U \Lambda U^T,$$

with $U \in \mathbb{R}^{N \times N}$ the eigenvectors and Λ the eigenvalues. The eigenvectors form an orthonormal basis for node functions. We use the first K eigenvectors (for Abilene, $K = 12$).

Transform node features $\mathbf{X} \in \mathbb{R}^{N \times 4}$ to spectral domain:

$$\mathbf{X}_{\text{spec}} = U^T \mathbf{X} \quad (\in \mathbb{R}^{K \times 4}).$$

Spectral Layers Each spectral convolution layer learns a weight tensor $W \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}} \times K}$. For each frequency k:

$$\mathbf{y}_k = \mathbf{X}_{\text{spec},k}\, W_k \quad (\mathbf{X}_{\text{spec},k} \in \mathbb{R}^{C_{\text{in}}},\ W_k \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}}})$$

Stack $\{\mathbf{y}_k\}$ into $\mathbf{Y}_{\text{spec}} \in \mathbb{R}^{K \times C_{\text{out}}}$, apply ReLU and dropout (rate 0.2), then invert transform:

$$\mathbf{H} = U\, \mathbf{Y}_{\text{spec}} \quad (\in \mathbb{R}^{N \times C_{\text{out}}}).$$

- **Layer 1:** Input $\mathbf{X}$, output $\mathbf{H}_1 \in \mathbb{R}^{N \times 64}$.
- **Layer 2:** Input $\mathbf{H}_1$, output final embeddings $\mathbf{H} \in \mathbb{R}^{N \times 64}$.

Spectral filters mix information globally, allowing the GFNO to capture long-range patterns in one operation.

Edge Prediction Head Identical to the GCN's head: for each edge (u, v), concatenate $\mathbf{h}_u$ and $\mathbf{h}_v$, then apply the same MLP architecture.

Model Complexity With $K = 12$:

- Spectral layer 1: $4 \times 64 \times 12 = 3{,}072$ parameters
- Spectral layer 2: $64 \times 64 \times 12 = 49{,}152$ parameters
- Edge MLP: $\sim 8{,}000$ parameters

Total $\approx 60{,}000$ parameters, compared to $\approx 13{,}000$ in the GCN. The spectral basis U is fixed and does not change during training.

3 Model Training and Evaluation

With the dataset prepared and both models defined, we proceed with training and evaluation. This section describes how we split the data, trained each model, and evaluated their performance on unseen traffic scenarios.

3.1 Data Splits

Our dataset consists of 100 traffic matrices, each representing a complete network state with a corresponding set of ground-truth link utilization values from the solver. We divide these into three subsets:

- 70 samples for training (70%)
- 15 samples for validation (15%)
- 15 samples for testing (15%)

The split is done randomly but with a fixed seed to ensure reproducibility. No overlap exists between the sets. Although 70 samples may seem small by deep learning standards, each sample here is a full graph-level instance with dozens of output labels (per-link utilization values), which helps increase the effective sample size. Since our focus is on a single topology and the goal is proof-of-concept, this data volume was sufficient to train both models without overfitting.

3.2 Training Procedure

Loss Function: We use Mean Squared Error (MSE) as the main loss function. It computes the average squared difference between the predicted link utilizations ($\hat{y}_{(u,v)}$) and the true values ($y_{(u,v)}$) from the LP solver. MSE heavily penalizes larger errors, which is important when high-utilization links are more sensitive. We also track Mean Absolute Error (MAE) during training, since it is easier to interpret (e.g., an MAE of 0.1 corresponds to 10% average error in predicted utilization).

Optimizer: We use the Adam optimizer with an initial learning rate of 0.001. Adam adapts the learning rate per parameter and tends to converge faster and more stably than standard SGD.

Batching: Each sample (one traffic matrix and its corresponding label set) is processed independently. Although it's possible to batch multiple samples together, we process one full graph at a time (batch size = 1), iterating through the full training set each epoch.

Epochs: We train for up to 100 epochs. One epoch means the model has seen all 70 training samples once. Both models usually converge well before 100 epochs, but we allow full training to ensure that validation performance is monitored throughout.

Validation and Early Stopping: After each epoch, we evaluate the model on the 15 validation samples using MAE as the main metric. We implement early stopping by tracking the best validation MAE across all epochs and saving the corresponding model state. At the end of training, we restore this best version of the model for final evaluation.

3.3 Evaluation Metrics

Mean Absolute Error (MAE): Primary metric. We compute the absolute error between predicted and true utilization values for each edge in each test sample, and then average them. This gives a straightforward sense of how close the predictions are, on average.

Mean Squared Error (MSE): Reported to understand the impact of large errors; we also refer to RMSE ($\sqrt{\text{MSE}}$) for interpretability.

Maximum Error: Worst-case absolute error across all links in the test set, to detect any major outliers.

Baseline Comparison: Core comparison between GCN and GFNO under identical data splits and conditions. The LP solver serves as the "oracle" ground truth.

3.4 Training Performance and Stability

We monitored the training loss (MSE) and validation MAE for both models across all epochs. The GCN, with fewer parameters, started with a higher initial error but showed stable convergence. The GFNO, which has more parameters due to its spectral layers, initially performed similarly but converged faster and to a lower error. In both cases, the use of dropout and a small learning rate contributed to smooth, stable training without divergence or instability.

We observed that the best validation MAE for both models typically occurred around the middle of the training schedule (between epochs 30 and 70), further justifying our use of early stopping based on validation performance.

3.5 Computational Efficiency

A major motivation for using a learned model is the ability to predict routing outcomes much faster than solving an optimization problem from scratch. Training both models took a modest amount of time-around 1–2 seconds per epoch on a CPU for the GCN, and slightly more for the GFNO. On a GPU, this could be sped up significantly, although our experiments were performed on standard CPU hardware.

The key benefit comes at inference time. After training, generating predictions with either model takes only a few milliseconds per sample, even on a CPU. In contrast, solving the LP using a traditional solver takes hundreds of milliseconds to over a second. On larger networks, LP solve times grow quickly, while neural models maintain constant-time inference. This makes the GCN and GFNO models suitable for use in real-time or near real-time SDN controllers, where fast response to changing traffic is critical."'

4 Results and Discussion

4.1 Quantitative Results

After training both the GCN and GFNO models, we evaluated their performance on the held-out test set and compared their predicted link utilizations to the optimal values computed by the traditional linear program solver. In addition to presenting the numerical results, we also examined how each model behaved in practice and what insights could be drawn from their comparative performance.

The GCN model, serving as our baseline, achieved a Mean Absolute Error (MAE) of 0.1435 on the test set. This translates to an average absolute difference of 14.35 percentage points between the predicted and true link utilizations. Its Mean Squared Error (MSE) was 0.0343, corresponding to a Root Mean Squared Error (RMSE) of about 0.185. While this result is a reasonable starting point given the simplicity of the model architecture and the modest size of the training set, the magnitude of the error shows that the GCN may struggle to reliably approximate optimal link utilizations, especially in more complex traffic scenarios.

In contrast, the GFNO achieved significantly better accuracy. Its test MAE was 0.0729, approximately half of the GCN's, and its MSE was 0.0091, with an RMSE of about 0.0958. These results indicate that the GFNO was able to make much closer predictions to the optimal values, with an

Table 1: Training/validation MAE at selected epochs and final test metrics for GCN and GFNO

Model	Epoch	Train MAE	Val MAE	Test MAE	Test MSE
GCN	1	0.402	0.304		
	10	0.305	0.289		
	20	0.272	0.240		
	50	0.188	0.157		
	100	0.159	0.124		
	Test			0.1435	0.0343
GFNO	1	0.419	0.252		
	10	0.170	0.139		
	20	0.100	0.082		
	50	0.073	0.071		
	100	0.058	0.061		
	Test			0.0729	0.0091

average error of only 7.3 percentage points. Throughout training, the GFNO consistently showed lower validation error compared to the GCN and demonstrated better convergence behavior. While the GCN's validation MAE began to plateau around epoch 80 and slightly worsened by the end of training, the GFNO's validation MAE steadily improved until it leveled off near epoch 90, which was when we saved the best-performing model.

4.2 Qualitative Analysis

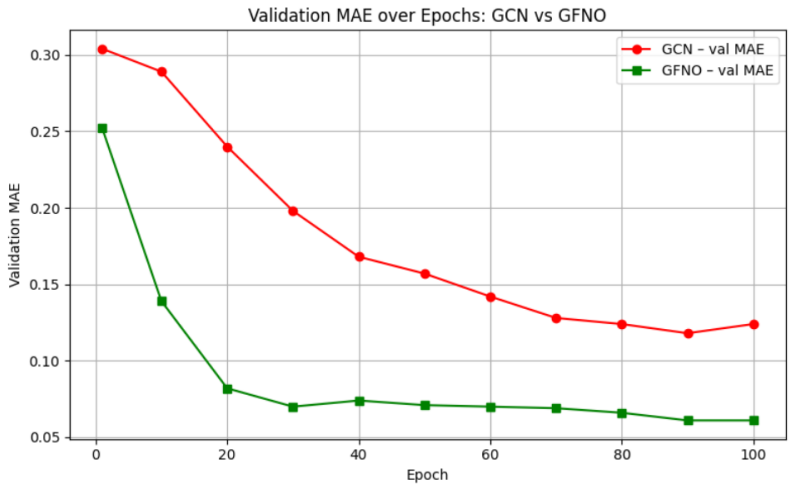

Figure 4: Validation MAE over epochs for the GCN and GFNO models on the validation set.

One of the key differences in performance between the two models seems to come down to their ability to represent and reason about global traffic patterns. The GCN, using only two layers, operates mostly locally. It aggregates information from neighbors and neighbors-of-neighbors but may miss longer-range dependencies, especially in cases where traffic from multiple distant nodes converges and stresses particular links. This limitation is particularly apparent in test cases where two or more heavy-traffic nodes are far apart in the topology. If those nodes generate flows that pass through common bottlenecks, the GCN may underestimate utilization due to its limited receptive field.

On the other hand, the GFNO works in the spectral domain using the eigenvectors of the graph Laplacian, which are inherently global. By projecting input features into this basis, the GFNO gains a way to recognize patterns that span the whole network. For instance, if the traffic matrix exhibits a structure that matches a particular spectral mode-such as a heavy flow from east to west-the corresponding coefficients in the transformed input will reflect that, and the model can adjust its

predictions accordingly. This allows the GFNO to detect how traffic imbalances or distributed load might affect links that lie between distant regions. The global nature of the spectral representation gives it a significant advantage for this type of task.

Beyond raw accuracy, the GFNO also showed more consistent performance across different test samples. The GCN's predictions had higher variance, with some individual test cases showing errors well over 30%, while the GFNO rarely exceeded 15% on any link. This level of reliability matters in network control applications where overestimating or underestimating link usage could lead to unstable routing decisions. For example, underpredicting congestion on a link might prevent proactive rerouting, while overpredicting might cause unnecessary changes that disrupt flow stability. A model that produces consistently accurate predictions is therefore more useful in practice, even if both models are fast.

To better understand this difference, we examined particular test cases. In one scenario, two nodes that had not previously generated much traffic during training suddenly became major traffic sources. The GCN, without sufficient training examples of that configuration, misjudged the link utilizations and predicted too low on a central link that carried a large portion of their traffic. The GFNO, however, successfully identified the correct bottleneck and predicted its utilization within 5% of the optimal value. This suggests the GFNO learned general patterns of flow distribution that hold across different traffic matrices, while the GCN was more reliant on specific examples it had seen.

Importantly, both models were trained and tested on the same fixed topology, so generalization here refers only to different traffic patterns on that topology. We did not evaluate performance on new topologies, which would require additional training or architectural adjustments. However, for scenarios like software-defined networks where the topology is relatively stable and traffic fluctuates frequently, this kind of generalization is extremely valuable. It means a trained model can adapt to new traffic conditions almost instantly without the need to re-solve a complex optimization problem.

The practical benefits of using a learned model become even clearer when considering runtime. The traditional LP solver, while optimal, takes between hundreds of milliseconds and a full second to compute a solution on our small 12-node network. In larger topologies, this delay could increase dramatically. In contrast, both the GCN and GFNO can make predictions in just a few milliseconds, even on a CPU. The extra computational cost of the GFNO's spectral transform is negligible at this scale, and with optimized libraries or GPU acceleration, both models could be used in real-time applications. In deployment, this enables near-instantaneous predictions for rerouting or load balancing, offering a powerful tool for adaptive traffic engineering.

4.3 Limitations and Future Work

While the Graph Fourier Neural Operator (GFNO) demonstrated strong performance and significant advantages over the baseline GCN model, there are still several important limitations and areas for future improvement worth considering.

One key limitation is that the GFNO is inherently tied to a specific network topology. Because the model relies on the eigenvectors of the graph Laplacian as its spectral basis, any change to the topology-such as adding or removing nodes or links-would alter the Laplacian and therefore change the spectral basis entirely. This means that the model, as trained, would not generalize across topologies without recomputing the eigen-decomposition and potentially retraining or fine-tuning the weights. This is acceptable for use cases where the topology is mostly stable, but in more dynamic environments, such as data center or multi-domain routing contexts, it might be a limiting factor. While GCNs also do not trivially generalize across graph structures without retraining, they are more flexible in principle if designed to be topology-agnostic or if trained on diverse graphs. In our fixed-topology setting, this was not an issue, but it points toward future work on making spectral models more robust to minor changes, possibly through basis adjustment techniques or meta-learning approaches.

Scalability is another concern. The Abilene network used in our experiments has only 12 nodes and 28 directed links, which makes the eigen-decomposition and spectral operations very efficient. However, for larger networks with 100 or more nodes, the cost of computing the full Laplacian spectrum can grow quickly. In theory, eigen-decomposition is an $O(N^3)$ operation, and using the full set of eigenvectors incurs storage and computational overheads of $O(N^2)$ per forward pass. A practical way to reduce this is to truncate the basis and use only the first K eigenvectors, where

$K \ll N$. This acts like a low-pass filter and may still capture most of the relevant structure, but selecting the right K and understanding how it affects performance is still an open question. In future work, it would be valuable to study how performance degrades (if at all) as K is reduced, and whether smarter basis selection methods can help retain accuracy while scaling up.

Another limitation is that our dataset was generated using a synthetic gravity model, which produces fairly smooth and balanced traffic matrices. Real-world traffic can be much more bursty, irregular, or influenced by external events, and our models have not been tested under such conditions. For example, some scenarios might involve a single dominant flow or exhibit highly skewed patterns that rarely occurred in training. While the GFNO showed some capacity to generalize to unseen traffic patterns, it remains unclear how robust it would be to extreme out-of-distribution cases. One way to improve robustness could be to augment the training set with artificially generated edge cases or incorporate historical traffic data if available. This would help the model better handle real-world deployment where unusual traffic behavior can be common.

Another direction for future work involves moving beyond utilization prediction toward generating actual routing decisions. Our models predict the expected per-link utilization under optimal routing, but this is not the same as determining how to route the traffic. In practice, a network controller needs to translate these predictions into concrete forwarding rules. One possible approach is to use the predicted utilizations to guide heuristic routing algorithms or to prune the search space of a secondary optimization step. Alternatively, more advanced models could be trained to output routing policies directly, though this would require a more complex output space and might introduce challenges related to combinatorial prediction. We focused on utilization prediction because it offers a tractable and informative proxy for network load, but future extensions could explore end-to-end learned routing schemes, particularly in architectures where latency or load balancing objectives are more dynamic.

5 Conclusion

This project demonstrated that neural operator models, particularly the Graph Fourier Neural Operator (GFNO), can be effectively applied to network traffic engineering tasks. By learning to predict optimal link utilization under varying traffic demands on a fixed topology, our model offers a fast, accurate alternative to traditional optimization solvers. The success of the GFNO also highlights a broader insight: techniques developed for solving partial differential equations in scientific computing can be adapted to optimization problems in communication networks. Just as PDEs describe the evolution of physical systems, traffic engineering seeks to balance flow across a network under global constraints. Neural operators, especially those based on spectral representations, provide a powerful framework for modeling these types of functional relationships. Going forward, this direction opens the door to faster, more adaptive traffic control systems, particularly in settings like SDN, and suggests future research opportunities in generalizing to dynamic topologies and integrating model predictions into real-time routing decisions.

References

[1] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," *Proc. IEEE INFOCOM*, 2000.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[3] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging graph neural networks for network modeling and optimization in SDN," *IEEE JSAC*, vol. 38, no. 10, pp. 2260–2270, 2020.

[4] A. Abrol, P. M. Mohan, and T. Truong-Huu, "A deep reinforcement learning approach for adaptive traffic routing," *Proc. IEEE ICC*, 2024.

[5] Z. Xu, F. Y. Yan, R. Singh, J. T. Chiu, A. M. Rush, and M. Yu, "Teal: Learning-accelerated optimization of WAN traffic engineering," *Proc. ACM SIGCOMM*, pp. 1–17, 2023.

[6] A. AlQiam, Y. Yao, Z. Wang, S. S. Ahuja, Y. Zhang, S. G. Rao, B. Ribeiro, and M. Tawarmalani, "Transferable neural WAN TE for changing topologies," *Proc. ACM SIGCOMM*, pp. 86–102, 2024.

[7] Z. Li, N. Kovachki, K. Azizzadenesheli, *et al.*, "Fourier neural operator for parametric partial differential equations," *Proc. ICLR*, 2021.

[8] B. T. Thonnam Thodi, S. V. R. Ambadipudi, and S. E. Jabari, "Fourier neural operator for learning solutions to macroscopic traffic flow models," *Transportation Research Part C*, 2024.

[9] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *Proc. ICLR*, 2017.

[10] Y. Zhang, Z. Ge, A. Greenberg, and M. Roughan, "Network anomography," *Proc. ACM IMC*, p. 30, 2005.

[11] SNDlib, "Abilene Network Topology," `https://sndlib.zib.de/home.action`, accessed 2025.

[12] T. N. Kipf, "Graph Convolutional Networks," online tutorial, 2017. Available: `https://tkipf.github.io/graph-convolutional-networks/`.