Exploring Graph Neural Networks for Real-Time Transit Arrival Prediction

Nick Gable

Submitted under the supervision of Alireza Khani to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science, *summa cum laude* in Computer Science.

December 19, 2023

Acknowledgements

There are several people I would like to acknowledge who have had a significant impact on me and my work on this thesis:

1. Alireza Khani, my thesis and research advisor, who has graciously had me as a member of his research lab for several years, and has provided wonderful and sound guidance in all my work in his lab, especially this thesis.
2. Yao-Yi Chiang and Jack Kolb, my thesis readers, whose feedback and insights on my work were valuable and appreciated.
3. Nick Racz, my honors advisor, who has provided helpful insights in all aspects of my undergraduate career, including the thesis writing process.
4. Various close friends and family, especially my roommates at The Forge, who have supported me through this process, constantly asking good questions, providing insights, and keeping me grounded.
5. My parents Jeff and Dori Gable, for supporting me and providing for me throughout my academic career, and so much more.
6. The Lord Jesus, for his overflowing grace; for the gifts of academics, learning, and research; for guiding me and providing for me in college and all of life.

Abstract

Real-Time Transit Arrival Prediction is an important problem to solve in the public transportation world, as users of these systems increasingly rely on real-time up-to-date information to plan their use of buses and trains worldwide. Various approaches to this problem exist in literature, and this thesis explores the use of a Graph Neural Network, a special type of Neural Network specifically designed to operate on graphs. The model is trained on two subsets of the Metro Transit network in the Minneapolis-St. Paul area. The results show that the model is inferior to the existing predictions in place, but still reasonably accurate, showing the value of a GNN implementation to approach this problem.

Table of Contents

1 Introduction

For transit riders, an accurate understanding of when a vehicle is going to arrive at their stop is extremely important. In theory, this would be as simple as using the transit schedule, but the reality is that countless factors often cause vehicles to deviate from the schedule. Thus, in order to provide accurate vehicle arrival times, transit agencies must implement some sort of real-time prediction system to determine schedule deviances and inform riders of the resulting delays, or early arrivals.

Prediction of these deviations is no easy task, as the data required to do so is not always available or accurate, and some schedule deviations are the result of unpredictable events (traffic accidents, excessive demand, mechanical failure, etc). Nonetheless, most modern transit agencies provide some form of real-time arrival prediction, and major navigation services like Apple Maps and Google Maps tie into this, sometimes doing their own predictions in addition to what the agency provides [1].

A variety of Machine Learning models have been implemented to approach the problem of vehicle arrival prediction in real-time. This thesis seeks to specifically explore the application of Graph Neural Networks (GNN) within this domain. As the name implies, a GNN is a Neural Network designed to operate on data that can be represented as a graph. Transit networks are very easily represented in this way, leading a GNN to be a seemingly appealing option in making real-time predictions across a network.

One additional challenge in real-time prediction is making predictions across a network with a diverse variety of routes and modes of transit (such as local bus, express, Bus Rapid Transit (BRT), Light Rail Transit (LRT), etc). Much existing research focuses on predictions made along a single route or across a handful of routes, but less research has been done on making predictions across such a diverse network, without creating route specific models. In this vein, GNNs seem particularly promising, with techniques such as message passing between graph nodes being useful in cultivating the sharing of information across the transit network. With that in mind, this thesis specifically seeks to explore the effectiveness of a GNN at making predictions across portions of the Metro Transit network in the Minneapolis-St. Paul area, which contains a large number of local and express bus routes, as well as BRT and LRT lines.

The upcoming section explores existing work done in the transit and transportation world relating to real-time arrival prediction, as well as work done in these spheres specifically using Graph Neural Networks. We then outline a data pipeline and GNN model design designed to approach this problem, and analyze its results on two subsets of the broader Metro Transit network.

2 Related work

Real-time transit vehicle arrival prediction has been studied in depth, and a volume of solutions exist in literature. Historically, a number of straightforward solutions to this problem have been proposed or implemented by transit agencies, such as historical-data based models or regression models. Jeong and Rilett (2005) discusses how these models rely on simple mathematical relationships between historical travel times, scheduled times, and actual travel times, but discusses how in practice, these relationships are often more complicated [2]. Thus, recent literature has focused more closely on machine learning models as a means to more adequately address this problem.

2.1 Machine learning applications

Jeong and Rilett (2005) implements an Artificial Neural Network (ANN) as a means to predict bus arrivals on a single route in Houston, making predictions based off of input features of known arrival time, dwell time, and schedule adherence at previous stops. They showed that their model performs better than historical and regression-based models, showing the immediate benefit to using machine learning on this problem. Yin et al. (2017) does similar, implementing both an ANN and Support-Vector

Classifier (SVM) based model and showing that the ANN model was superior [3]. Their model takes inputs of proceeding bus arrival times along a route, as well as the travel speed of target vehicles, to produce an arrival time prediction. However, a number of more advanced techniques have arisen in recent years that have shown the value of more complex model architectures in making these predictions.

One such example is seen in Xie et al. (2021), which implements seven Recurrent Neural Network (RNN) models of varying complexity, ultimately finding that its ConvLSTM model performed best [4]. They used a large number of input features such as route, direction, driver, holidays, and weather, to output predictions in the form of incremental time offsets from the previous stop in the sequence, for each stop. Unfortunately, a lack of consistent metrics, as well as this model's application to a different route in China makes it difficult to compare these results to those found by Jeong and Rilett (2005). However, Agafonov and Yumaganov (2019) implement an RNN with Long Short Term Memory (LSTM) and shows its superiority to a Multi Layer Perceptron (MLP) model similar to the model proposed by Jeong and Rilett (2005), showing the value of the more complicated RNN and LSTM architectures [5]. Agafonov and Yumaganov (2019) uses the current date and time, as well as a variety of travel time related metrics as input features for their model.

Chondrodima et al. (2022) shows the value of intentional data cleaning and data-preprocessing combined with strategic model implementation. It tackles the real-time prediction problem using Radial-Basis Function (RBF) Neural Networks, a special type of ANN that uses an RBF as its activation function [6]. Combined with a "novel pre-processing pipeline" and a Particle Swarm Optimization (PSO) method for training the model, Chondrodima et al. (2022) creates a model that outperforms numerous other models while still simply being powered by an ANN. This paper also trained their model on a dataset of collected Metro Transit data, making its results more relevant when compared to those of this thesis. Their model takes in as input information about prior stop visits as well as other information related to future stops, and predicts the arrival time for a given future stop.

Google Research has also used ML techniques with Neural Networks to predict bus arrival times, and this technology is integrated into their Google Maps predictions [1]. The exact architecture is not disclosed, but it appears to be sufficiently robust so as to be used on transit systems where reliable real-time data is not available.

2.1.1 Similar ML applications in traffic/transportation

ML models similar to the ones used above have also been used for a number of applications in transportation or traffic prediction. One example relevant to the real-time transit arrival prediction problem is implemented by Jenelius and Koutsopoulos (2013), which uses a MLP to predict vehicle locations across an urban road network using low-frequency GPS reporting [7]. Another example is by Qiu and Fan (2021), which studies a variety of common ML models including Decision Trees, Random Forest, Extreme Gradient Boosting, and LSTM networks to predict short-term travel time [8].

We also see in literature examples of more complicated RNN or LSTM architectures in use in this domain. One example of this is by Wang et al. (2019), which uses a bi-directional LSTM network to predict traffic speed across the central district of Xuancheng, China. Li et al. (2020) shows something similar, using a Bi-LSTM model combined with a CNN to predict traffic congestion [10]. A third example is by Choi et al. (2018), which uses RNN modeling to predict individual vehicle trajectories [11].

2.2 Kalman Filters

Rather than implementing a machine learning model, Metro Transit uses *TheTransitClock*, which implements a Kalman filter, an estimation algorithm that makes predictions using weighted averages of estimates built upon past observations [12]. Achar et al. (2022) also uses a Kalman filter combined with a Neural Network (NN) to solve the prediction problem [13].

While Metro Transit in 2020 ultimately decided on a Kalman filter implementation due to its optimal performance against competing algorithms [12], it is worth noting that existing literature does *not* seem to view Kalman filtering as state-of-the-art, with Jeong and Rilett (2005) and Yin et al. (2017) citing existing literature that found Kalman filtering inferior to even ANN modeling. Additionally, Agafonov and Yumaganov (2019) views Kalman filters as simple but restrictive due to their inability to forecast "complex nonlinear space-time relations". Nonetheless, it does seem clear that Kalman filtering is a commonly implemented algorithm in this domain due to these papers, and others such as Chondrodima et al. (2022) and Elliott and Lumley (2020).

2.3 Graph Neural Networks

As seen in section 2.1, a number of different ML-based approaches to the real-time transit vehicle prediction problem exist in literature. However, many of these papers are relatively narrow in scope, being applied and tested along a single route, or a small sampling of routes with similar characteristics. Thus, in order to apply these models at a network level, one would need to create many different versions of the same model. This may be impractical and inefficient, and also misses the opportunity for the model to learn from information from other routes.

Yin et al. (2017) tackles this problem by explicitly feeding information from other routes on the same road into their model. This is a useful technique, but it still fails to adequately consider the complex spatio-temporal relations that exist within a complex, multi-modal transit network. Rather, Zhang et al. (2022) discusses how one may instead turn to a Graph Neural Network (GNN) in order to capture these characteristics.

As previously mentioned, Graph Neural Networks (GNN) are a type of NN Deep Learning model that are designed to operate on graph data structures. An extension of Convolutional Neural Networks (CNN), GNNs are able to take advantage of graph topology to extract meaningful information from the graph as a whole that other models could not. Within this sphere, Sahili and Awad (2023) discusses spatio-temporal GNNs, GNNs that specifically take into account spatial and temporal dimensions of the data [16]. Thus, a GNN has the potential to resolve many of the issues faced in previous real-time models.

2.3.1 GNN applications in transit

There are a number of applications of GNNs related to transit vehicle time prediction, as well as transit in general, in literature. One such example is by Zhang et al. (2022), which predicts train station delay across a rail network in China using a "Multiattention Graph Convolution Network", showing its results to be superior to a number of other models used in other papers, including ANN and LSTM models [15]. However, this model only serves to predict a single number of delayed trains, requiring more complexity to make predictions network-wide. Li et al. (2023) is similar, applying a "Heterogeneous GNN approach" that combines two existing GNN models to predict station delays across two sub-networks in the China railway system, creating a model that takes in an entire "heterogeneous graph" representing the network state, and outputting delays for all stations [17]. In the urban transit sphere, Ma et al. (2022) proposes a "Multi-Attention GNN" that they use for city-wide bus travel time estimation (essentially the same problem as arrival-time prediction), evaluating it using data from Xi'an, China [18]. Their model has the additional benefit of being robust when trained with limited or sparse data, and feeds in location, timestamp, speed, bus ID, and weather information as inputs. Finally, Li et al. (2022a) uses a GNN to predict public transit demand, testing their model on data from the Greater Sydney area [19].

2.3.2 Similar GNN applications in traffic/transportation

GNNs have also been applied to solve a broad number of traffic or transportation problems. In the realm of traffic prediction, Zhou et al. (2021) proposes a Bayesian framework using "variational graph recur-

rent neural networks" for traffic forecasting [20]. Ji et al. (2023) and Li et al. (2022b) do similar, both applying Spatial-Temporal GNNs to this problem [21], [22]. Jiang and Luo (2022) surveys a number of existing papers that apply GNNs to solve traffic prediction [23]. In industry, Google and DeepMind have partnered to use GNNs for traffic prediction, informing route selection [24].

Outside of traffic prediction, Wu et al. (2023) applies a "Spatio-Temporal Heterogeneous GNN" to estimate travel time [25]. Singh and Srivastava (2022) uses a GNN with an RNN for trajectory prediction, the intent being for the model to be used in autonomous vehicles to predict the location of other vehicles on the road [26]. Liang et al. (2022) applies a Spatio-Temporal GNN to recover missing data used in general transportation operations, such as sensor data from roads and vehicles [27].

2.4 Conclusion

It is clear that there are a number of very mature and powerful ML models in use throughout literature to predict transit vehicle arrival time. However, many of these models are lacking in scope or scale, often covering only a single route or lacking predictive ability for all stops on the route. To overcome these concerns and also take advantage of relationships between routes across the network, using a GNN to approach this problem appears promising, and can be seen in Ma et al. (2022). While less research has been done using GNNs in the transit sphere, the multitude of applications it has seen in similar problems in the transportation domain suggests that it is a worthy target for trying to approach the transit arrival problem.

For all of these reasons, this research will explore the use of Graph Neural Networks as the primary model to predict transit vehicle arrivals. While some papers have done this already, there appears to be a lack of published results applying a GNN to a transit network as diverse as Metro Transit is (with local/express bus, BRT, and LRT), or within a more localized urban area in general. Thus, it provides a good opening into research for this thesis.

3 Model design

3.1 Data pipeline

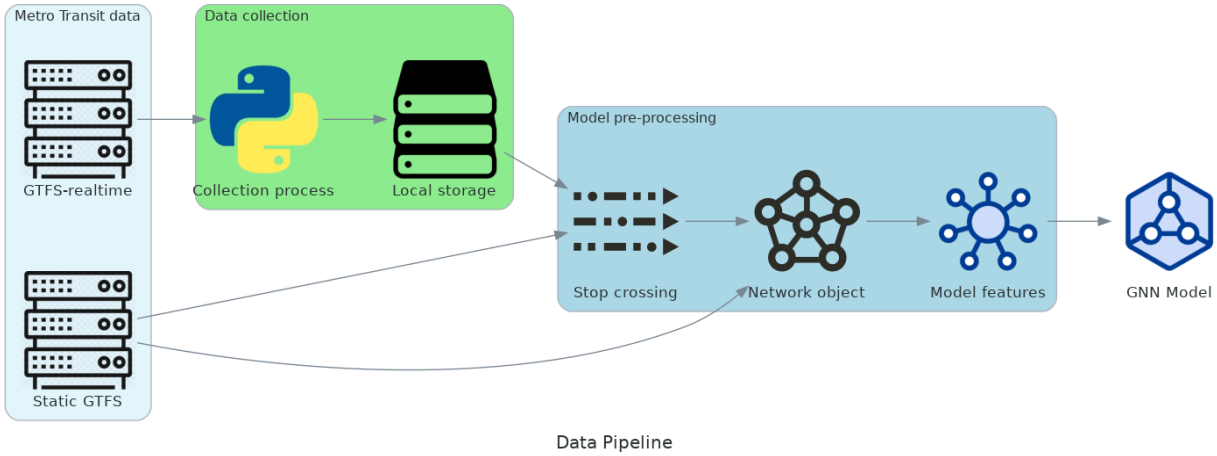

Figure 1: Schematic outlining data pipeline process, from initial data collection to final input into the model.

This section outlines the process of collecting data for use on this model, converting it, and processing it before ultimately feeding it into the model training / testing loop.

3.1.1 Metro Transit data

Data from Metro Transit (MT), the principal transit operator in the Minneapolis-St. Paul Metropolitan area, was collected and used for this project. Both real-time and static General Transit Feed Specification (GTFS) data was collected:

- Metro Transit provides three GTFS realtime feeds: **TripUpdate**, **VehiclePosition**, and **ServiceAlerts**. For this project, data was collected from VehiclePosition (real time GPS locations of vehicles), and TripUpdate (predicted vehicle arrival times from MT). The vehicle position data is used to train the model, and the trip update data as a means of comparison to the current predictions in use at MT.
- Static GTFS data (stops, trips, scheduled stop visits, etc) was also used throughout the data pipeline. Because this data doesn't change as frequently, static data was pulled as needed from MT servers and archives rather than having a more detailed collection procedure.

3.1.2 Data collection process

Realtime data was collected over a three week period from September 10th, 2023 (Sunday) at 4 AM to October 1st, 2023 (Sunday) at 4 AM. This period was selected to minimize disturbances and schedule deviances from local holidays and events - notably, it is after the Minnesota State Fair, as well as after the University of Minnesota and most other local universities start their fall semester. Data collection was started at 4 AM since Metro Transit schedules start the day at 4 AM rather than midnight.

Two separate processes periodically polled the two feeds (vehicle positions and predicted times). In both cases, data was pulled from the GTFS realtime feed, converted into tabular data (Pandas DataFrame), and then appended to the existing data for that day. Vehicle position data was polled every 15 seconds, and prediction data every 60 seconds, consistent with the approximate frequency of those feeds being updated.

3.1.3 Pre-processing steps

After being collected, the real-time vehicle location data had to go through three main steps before it was ready to be used by the model:

1. **Stop crossing**, where the raw vehicle location data was converted into stop arrival data.
2. **Network object / high level graph representation**, where a high-level graph object containing both the topography of the transit network and stop visit data is created.
3. **Model features / low level graph representation**, where the high-level graph is used to generate a vectorized representation of its information that can then be fed into the GNN model.

Graph representation implementations are described in the Network Representation and Model Implementation sections.

3.1.4 Stop crossing algorithm

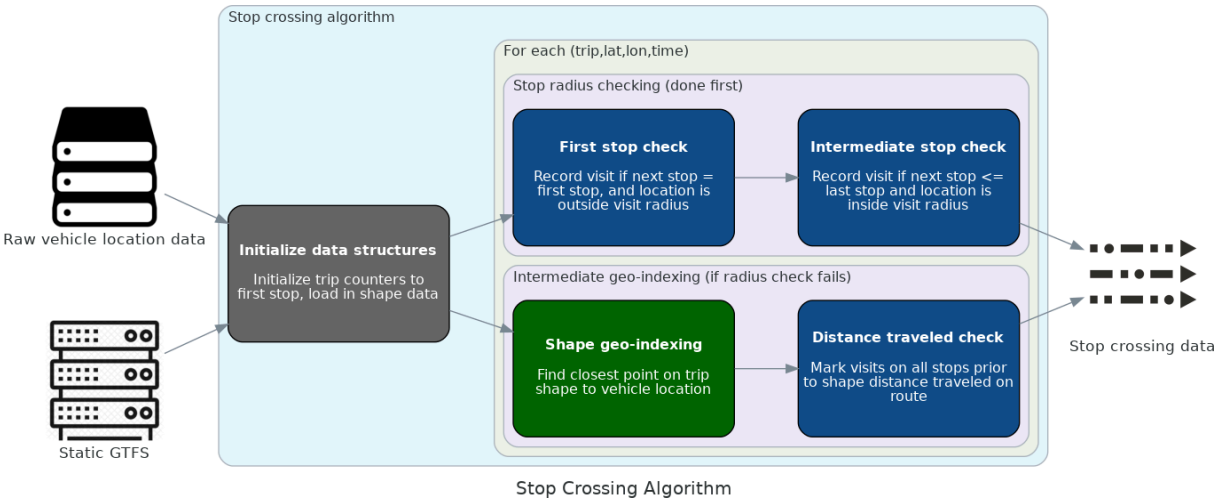

Figure 2: Stop crossing algorithm schematic.

The goal of the stop crossing algorithm is to convert the raw data taken from realtime GTFS feeds into more meaningful stop visit data. Implementing this algorithm robustly is important, as it directly impacts the quality of the data that the model will train on.

3.1.4.1 Input and initialization

Table 1: Input data format for stop crossing algorithm. Data is tabular, and produced by the data collection process.

Field	Description
`trip_id`	Unique string identifier for the trip this entry is reporting on
`route_id`	Integer route ID value
`latitude`	Latitude coordinate
`longitude`	Longitude coordinate
`bearing`	Float bearing value (not used)
`speed`	Float speed value (not used)

The stop times algorithm also makes use of the following static GTFS files:

- `stops`, a table containing `stop_id` and stop location values, used in stop radius checking to determine proximity to stops.
- `stop_times`, a table containing scheduled arrival times at each stop, per unique `trip_id`. When the algorithm is initialized, these values are separated by trip so that an order of stop visits can be established.
- `trips`, a table providing information about each trip, used to retrieve `shape_id` and `route_id` values for each trip.
- `shapes`, a table containing a large number of latitude/longitude coordinates per trip that can be used to trace the precise route a vehicle travels along during the course of a trip. This is loaded into a GeoPandas GeoDataFrame and then separated by `trip_id` for easy access later.

After loading in these files, the stop crossing algorithm initializes for each trip:

- A `next_stop` counter representing the next stop in the sequence that has yet to be recorded as a visit. This is initially equal to 1, indicating that the first stop in the sequence (in GTFS files, stop sequences are 1 indexed) has yet to be visited.
- A `crossing_times` list, used to keep track of the times that a visit is recorded.

The algorithm then loops through every data point in the input data set, populating the `crossing_times` list as appropriate.

3.1.4.2 Stop radius checking

The first check that is done for each data point is a simple distance check, checking whether or not the recorded latitude/longitude location is within a certain visit radius $r = 0.1$ miles. This is done in two phases:

1. If `next_stop` = 1, record a stop visit if the recorded vehicle location is *outside* the radius from the first stop. This is seen as indicating the start of a trip (since vehicle location data often exists prior to the start of a trip).
2. If `next_stop` $\leq (n = \text{number of stops})$, record a stop visit if the recorded vehicle location is *inside* the stop radius of the next stop to be visited.

Stop radius checking is the only way that a first stop or last stop visit is recorded by this algorithm. In both cases, it is because vehicle location data before and after the start of a route is not particularly meaningful, and can include random data such as driving to/from bus depots. Additionally, because location data sometimes ends before the last stop is reached, the algorithm will autofill the last 2 stops of data with the final recorded time of the vehicle being online, *if all other visit data up to the last two stops is recorded.* This was added after many initial tests of the algorithm found the last 1-2 stop visits missing (likely due to an early end to location tracking).

3.1.4.3 Intermediate geo-indexing

If $1 <$ `next_stop` $< n$, and the vehicle location was found to be *outside* of the stop radius, then an intermediate process is used to attempt to determine if the vehicle has since passed the next stop, as well as potentially future stops. This process is needed since vehicle location data often does not record the precise instant that a vehicle is at a stop, but can often get relatively close to it, although lack of data sometimes leads to gaps that make this harder.

This process works in the following manner:

1. **Index the shapes GeoDataFrame by the current vehicle location, returning the closest point on the trip shape.** This process (and indeed, the entire algorithm) relies on the fast implementation of a *geo-spatial index* by GeoPandas, which allows for the algorithm to very efficiently search through a table of points along the route the vehicle is taking and determine the closest one.
2. **Determine the distance traveled along the shape, and mark all stops prior to this point along the shape as being visited.** This is also able to be done very efficiently as it relies on both the GTFS shape data including pre-computed distance traveled values for each point on the shape, as well as GTFS stop data including *shape distance traveled* values for each stop along the route. Thus, the algorithm can easily determine if the vehicle has passed the next stop *or future stops* simply by comparing these two values.

   Occasionally, the algorithm will determine that multiple stops have been passed between the previous and current data points. When this happens, the crossing times between the previous stop and the furthest reached stop are linearly extrapolated, meaning that the visit times for the intermediate stops are spaced out evenly as an approximation of the vehicles visits along that stretch of the route. Rather than using a more precise method, linear extrapolation was chosen here as a way to approximate and fill in smaller data holes that were causing the algorithm to fail to complete routes, most notably seen on routes with high numbers of stops in close proximity (such as local bus routes). Additionally, on a one day sample of the training data for the high frequency network (see Section 3.4.2), only 4% of output data points were the result of linear extrapolation, with an average of 1.4 stops extrapolated *when linear extrapolation was needed.* Thus, we can see that the

practical impact of linear extrapolation was minimal, while providing a decent approximation for cases where multiple stops need to be recorded as visited from a single location point.

3.1.4.4 Output crossing data

Table 2: Stop crossing data output format.

Field	Description
`trip_id`	Unique string identifier from input data
`route_id`	Route integer from input data
`stop_id`	Stop ID integer representing stop being visited
`stop_name`	String name of stop being visited
`date`	Date/time value indicating when this trip arrived at this stop

This data is then loaded into network representations later in the data pipeline which can then be used to generate training data for the model.

3.1.4.5 Optimizations

Transit location data is not always the highest quality, and gaps or errors in the data can lead to problematic or unrealistic results from this stop crossing algorithm. To address some of these problems, three different optimizations / checks were used:

1. **First stop visit capping**: One frequent problem that occurs with this dataset is location data for trips being reported long before the scheduled start of a route (sometimes up to 30 minutes!). This could lead the algorithm to falsely recording very early starts to trips. To address this, the stop crossing algorithm will never record a trip as having started earlier than two minutes before its scheduled time.

   While it is possible that some trips are actually starting more than two minutes early, this seems unlikely based off of observed stop crossing patterns for stops after the first stop. The cumulative benefit of data accuracy far outweighs this small loss of data accuracy.

2. **Missed stop limiting**: Occasionally, erroneous data points could lead the stop crossing algorithm to recording very large volumes of stops as being visited from a single location report (sometimes upwards of 20!). When this occurred, the algorithm would devolve to linearly extrapolating the entire stretch of stops, which quickly diverges from the actual visit times if the erroneous data point occurs substantially earlier in the trip than expected. These errors were likely the result of either a lack of precision within the input data, or reported location data coming from outside the trips scheduled time frame (also addressed by first stop visit capping). In every observed instance, these simultaneous visits were obviously not reflective of reality, so the algorithm will discard a data point if it would lead it to mark more than 5 consecutive stops as visited from a single data point.

   One problem with this approach is that prevents the crossing algorithm from recovering from a data gap if the time-frame is long enough to miss 5 stop visits. While this is important to consider, in practice this limit did not impact the number of reported crosses consequentially. Rather, trips where gaps occurred usually would lose their location data for all remaining stops in a trip, which will always be unrecoverable.

3. **Speed differential capping**: As an additional protection against erroneous data points, the crossing algorithm imposes a limit on the ratio between the computed speed of the vehicle on the newly-visited stretch of stops, compared to that of the schedule. If this ratio exceeds 2 (effectively, the vehicle traveled twice as fast as scheduled), the data point is dropped. The following formula is used to compute this ratio:

$$\frac{(d_f - d_p) \div (t_{dp} - t_p)}{(d_f - d_p) \div (t^s_f - t^s_p)}$$

$$\begin{aligned} d_f &= \text{Shape distance traveled for furthest reached stop} \\ d_p &= \text{Shape distance traveled for previously furthest stop} \\ t_{dp} &= \text{Reported time at current data point} \\ t_p &= \text{Reported visit time for previously furthest stop} \\ t^s_f, t^s_p &= \text{Scheduled visit times for current \& previously furthest stops} \end{aligned} \tag{1}$$

It is important to note that this cap is not intended to disallow the model from reporting trips that are actually very ahead of schedule. Rather, this cap is intended to reduce false positives in the crossing algorithm where a stretch of stops are all reported as visited simultaneously, leading to an unrealistically large increase in the average vehicle speed during that stretch of the route. To that end, this differential cap is only ever calculated and used when more than one stop is crossed at once - single stop crossings are always trusted and reported even if the speed differential is above 2.

Similar to the missed stop limiting, this cap could theoretically result in the algorithm being unable to complete trips, but in practice this does not appear to occur beyond situations where large amounts of data loss were present.

3.2 Abstract network representation

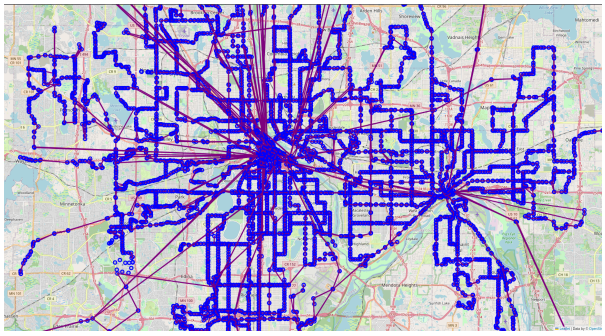

Figure 3: Map view of Metro Transit network graph representation.

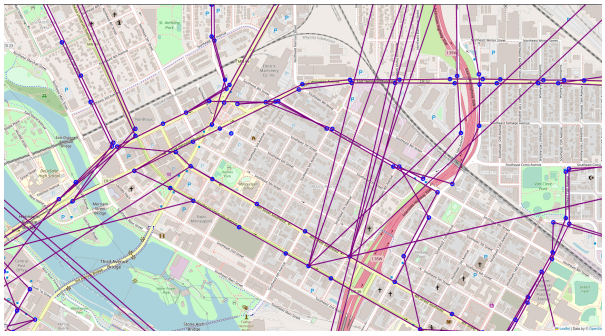

Figure 4: Close up view of individual stop connections.

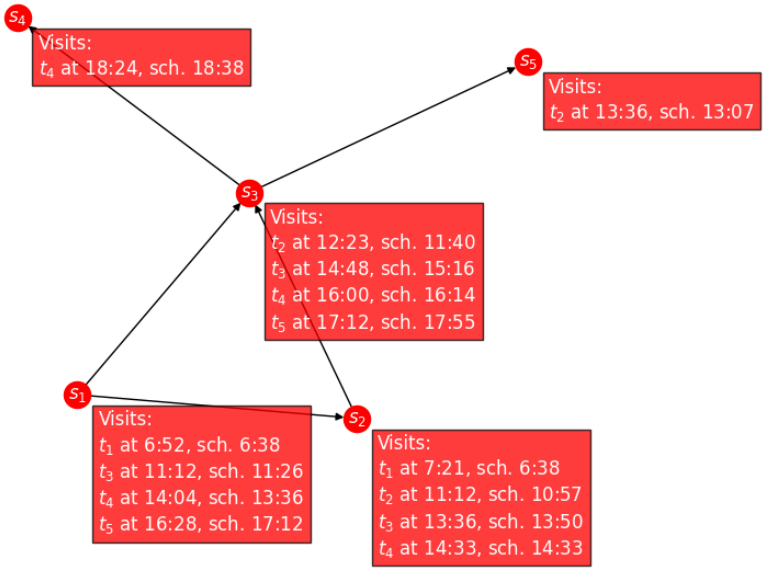

Figure 5: Example abstract graph representation of 5 trips and 5 stops. This graph contains all of the information needed to train the model, and will be transformed into an intermediate representation before being used to directly generate model features.

Before the data is converted into a format the model can recognize, a high-level graph representation of the transit network is created. Graph nodes and edges are implemented as simple Python classes for the purpose of having an abstract representation of the network the model will be making predictions on.

In this directed graph, stops are represented as nodes on the graph, and edges are made from one node to another if any trip within the network passes between those two nodes. Thus, the topology of the network represents all possible routes that transit vehicles can follow within it. One intentional consequence of this is that stops which are shared by multiple routes may have multiple edges to different stops across the network.

In this directed graph, stops are represented as nodes on the graph, and edges are made from one node to another if at least one trip in the network passes between those two stops. Thus, the topology of the graph represents all possible routes that transit vehicles can follow within it. One intentional consequence of this is that distinct routes which share sections of stops and stop connections in the actual network are represented using the same stops and edges in the graph, reducing graph complexity and facilitating message passing across trips and routes in the GNN model.

Each node/stop in the network stores its stop ID, stop name, and any trips that visit it, along with the visit times matched with their scheduled visit times. In the pre-processing pipeline, a single network is initially created with no reported trip visits using only static data. Then, for each day, the network is populated with the stop crossing data for that day, converted into the model-level feature representation, and then reset for the next day of data.

3.3 Model implementation

start_times	0.12	0.3	0.3	0.42	0.52

end_times	0.14	0.4	0.45	0.6	0.55

visit_times	$t_{1,k}$	$t_{2,k}$	$t_{3,k}$	$t_{4,k}$	$t_{5,k}$
s_1	0.12	0	0.3	0.42	0.52
s_2	0.14	0.3	0.4	0.44	0
s_3	0	0.35	0.45	0.5	0.55
s_4	0	0	0	0.6	0
s_5	0	0.4	0	0	0

sched_times	$t_{1,s}$	$t_{2,s}$	$t_{3,s}$	$t_{4,s}$	$t_{5,s}$
s_1	0.11	0	0.31	0.4	0.55
s_2	0.11	0.29	0.41	0.44	0
s_3	0	0.32	0.47	0.51	0.58
s_4	0	0	0	0.61	0
s_5	0	0.38	0	0	0

Figure 6: Start times, end times, stop visit, and schedule tensors, with trips within the window defined by $p = 0.35$ highlighted. This is the complete windowed data structure, generated from the graph shown in Figure 5. In the `visit_times` and `sched_times` tensors, $t_{i,k}$ and $t_{i,s}$ represent, respectively, the known and scheduled stop visit times for trip i.

X	$t^w_{1,p}$	$t^w_{2,p}$	$t^w_{1,k}$	$t^w_{2,k}$	$t^w_{1,s}$	$t^w_{2,s}$
s_1	0	0	0	0.3	0	0.31
s_2	0	1	0.3	0	0.29	0.41
s_3	0	1	0.35	0	0.32	0.47
s_4	0	0	0	0	0	0
s_5	1	0	0	0	0.38	0

Y	$t^w_{1,p}$	$t^w_{2,p}$
s_1	0	0
s_2	0	0.4
s_3	0	0.45
s_4	0	0
s_5	0.4	0

Figure 7: Example X and Y (input and output) feature matrices, generated from windowed data structure in Figure 6 when $p = 0.35$. In this example where $t^w_{max} = 2$, the first two columns of X are binary indicator values telling the model which stop/trip pairings to predict, the 2nd two columns are the known crossing times at time p, and the final two columns the schedule. $t^w_{i,k}$ and $t^w_{i,s}$ represent the known and scheduled stop visit times for *windowed* trip i, i.e. the ith trip contained in the current window. $t^w_{i,p}$ represents the prediction mask for trip i described later in this section.

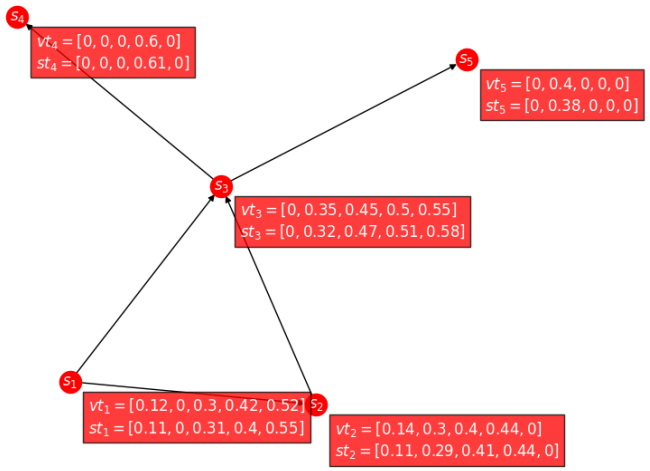

Figure 8: Abstract graph from Figure 5, showing each node's contributions to the windowed data structure in Figure 6. Each stop s_i contains a vector vt_i representing row i in the visit times tensor, and st_i representing row i in the stop times tensor. In both of these vectors, a given index j represents the time that trip j visited / was scheduled to visit the stop. Times are also converted from regular 24 hour time to be within the range of $(0, 1)$, offset by 4 hours to account for the transit day starting at 4 AM.

3.3.1 Model feature implementation

This model uses a "windowing" method as a means to efficiently store the features used to train and test the model, while also reducing the dimensions of the features fed into the model. The key to this implementation is the concept of a "window", defined as follows:

Let p be a given time within the transit day, $0 < p \leq 1$ (e.x. $p = 0.5$ represents a time 12 (out of 24) hours into the transit day). Let T be the set of all trips that are currently active (i.e. are between their first stop and their last stop) at the time p. The window ω is defined as the time period during which at least one trip in T is active.

Stop visits for trips in T that take place during ω are split by whether they fall before or after p. Visits before p are considered "known" values, as they represent visit information that would be known by a live model operating only on data available up to p. Known values are provided to the model as input (described in Section 3.3.1.2). Visits after p are "unknown" future values that the model is tasked with predicting. Thus, the model's task given the known values from ω (stop visits from trips in T that occur prior to p) is to predict all future values in ω that occur after time p.

3.3.1.1 Windowed data structure

Each day of stop crossing data is converted into a windowed data structure (Figure 6 above), which the training and testing loops can then use to dynamically generate model features based off of a given window time t. The Window data structure has the following fields which are generated from a day of stop crossing data:

- `start_times` and `end_times` are, respectively, vectors which contain the lower and upper bound for when a given trip t should be included in a given window. The index value of the vector represents the trip, so `start_times[i]` is when trip t_i is first included in a window. These are computed from the crossing data as the first and last time that a vehicle has crossing data in a given day. So, for a given trip t_i, t_i will be in the window generated by time p if `start_times`$[t_i] \leq p \leq$`end_times`$[t_i]$.

  One useful thing to note is that, since `start_times` and `end_times` are computed by the actual crossing data instead of the schedule, holes in the stop crossing data (due to algorithm limitations, or more likely missing feed data) will not propagate into the model features - instead, the model will be prompted to make predictions until data is lost, in which case the trip will be removed from future windows. This is because the stop crossing algorithm only marks stops as visited consecutively (meaning a hole in the crossing data will always be through the end of a trip, leading to that trip ending early in `end_times`).

- `visit_times` is a matrix where each row represents a stop, and each column represents a unique trip's (with index matching `start_times` and `end_times`) stop visits within the window. So, `visit_times` at row i and column j represents the time that trip t_j visits stop s_i, if such a visit occurs, zero otherwise.

- `sched_times` matches the structure of `visit_times`, but with scheduled arrival times instead of the ones generated by the stop crossing algorithm.

3.3.1.2 Conversion to model features

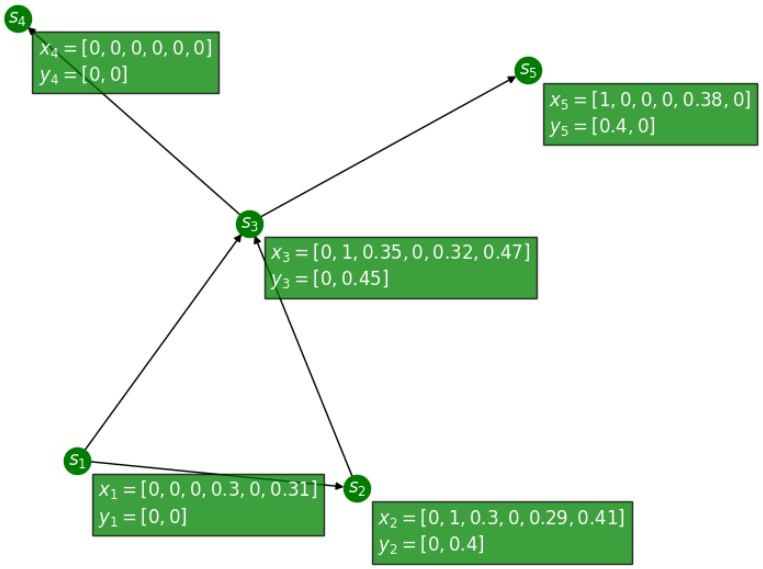

Figure 9: Graph representation of model input and target output features after the data has been windowed with time value $p = 0.35$. This graph shows each nodes contributions to the input matrix X and target output matrix Y as seen in Figure 7. See Figure 7 and below for explanations of each feature.

Once created for each day, the windowed data structure can be used to dynamically generate node features for a window at a given time p. This process is done within the training loop, and results in the creation of the node input feature matrix X and output feature matrix Y. In both of these matrices, a given row i represents the input/output features for stop s_i.

In the conversion process, trips from the windowed data structure are re-indexed so that they can be compactly fit into the (smaller) feature vectors. This dimension reduction is possible because of the windowing process, since only active trips in the window (as opposed to all trips that occur during the day) are passed into the model.

The input feature vectors contain three sub-matrices of information, each of length t^w_{max}, the maximum number of trips present in any given window across all days of training data. These matrices are concatenated left to right:

- A prediction matrix, where at a given row i and column j, $t^w_{j,p}$ is 1 if t^w_j visits stop s_i, 0 otherwise.
- A "known visits" matrix, recording stop visit times that have already occurred at the window time p - essentially, recording the known stop visits of vehicles that are currently moving in the window. This uses the same indexing scheme as the prediction matrix, and is 0 for trip/stop visits that do not exist, or for trip/stop visits that are intended to be predicted.
- A schedule matrix, recording scheduled visit times, or zero if a trip/stop visit does not exist.

The output feature matrix Y is essentially a copy of the prediction matrix in X, but with the predicted visit times from the model instead of a binary flag indicating a visit.

3.3.1.3 Special feature options

Some slight modifications to the input features were explored throughout the training process:

1. **Prediction only schedule**: Some models were trained with the input vector only providing schedule information for the future visits that the model was trying to predict, as opposed to providing both the past and future schedule information.

2. **Schedule deviation**: Some models were trained with both the predicted and known values reported as the *deviation from the schedule* instead of the absolute time of stop arrival. For example, if a trip had values $t^w_{i,k} = 0.5$ and $t^w_{i,s} = 0.55$ without schedule deviation, then with schedule deviation $t^w_{i,k} = -0.05$.
3. **Outlier cut**: To account for various sources of errors throughout the processing pipeline that could lead to erroneous output, some models were trained with ground truth values outside of a certain prediction threshold removed. This was only used with schedule deviation also enabled. For example, if this was set to 30 minutes, that would mean that a ground truth value claiming that a vehicle was more than 30 minutes outside of its scheduled time would not be fed into the model.

3.3.2 Model internals

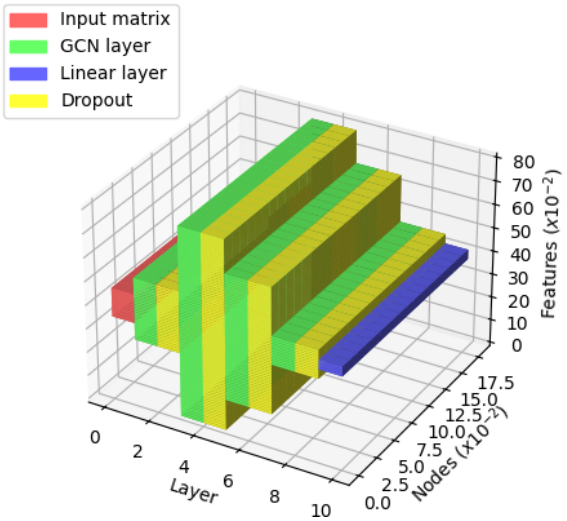

Figure 10: 3D plot of Model A8 structure, the best performing model on the high-frequency dataset. Layer sizes in the plot correspond to the output shape of that layer. Activation functions not shown.

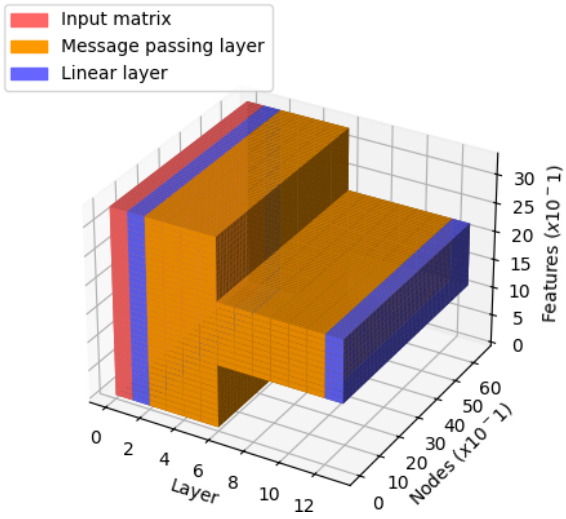

Figure 11: 3D plot of Model SD2 structure, the best performing model on the small U of M dataset.

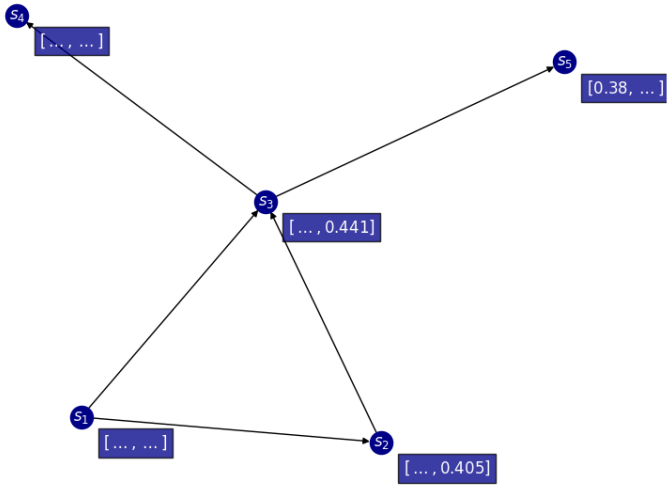

Figure 12: Example graph representation of model output, given input seen in Figure 7 matrix / Figure 9 graph. Irrelevant output features (past or no visit, so no prediction needed) are marked with (...).

The models implemented for this project are feed-forward Graph Neural Networks with straightforward architectures. The exact architecture varied trial to trial (see training and results sections), but each model would generally include a combination of simple linear layers, followed by a series of message passing layers, and a final output linear layer.

All models were implemented using PyTorch Geometric, an extension library to PyTorch that provides resources for implementing Graph Neural Networks [28].

In the following sections, X represents the feature matrix where each row represents a node in the graph (i.e. a stop).

3.3.2.1 Linear layers

PyTorch Geometric linear layers are simply linear transformations, implementing a standard fully connected neural network layer [28]:

$$X' = XW^T + B \tag{2}$$

3.3.2.2 Message passing layers

One of the unique features of Graph Neural Networks is *message passing*, the process where node features are propagated across edges in the graph structure, using an aggregation function such as a sum or mean.

Two message passing layers are used within models on this project, including a simple message passing layer and Graph Convolutional layer.

3.3.2.2.1 Simple message passing (`GraphConv`)

The simple message passing layers used in many of the proposed models are taken from Morris et al. (2021) and implemented by PyG as `GraphConv` [29]. They work by applying layer weights to the existing features, and then adding in the aggregation of features from connected nearby nodes to the output tensor, learning weights for those as well.

Let x_i be the ith row of the feature matrix X, i.e. the feature vector for node i. Additionally, let $\mathbb{E}(i)$ be the set of all nodes that have an incoming edge from node i. x'_i, the modified feature vector after the layer pass, is:

$$x'_i = W_1 x_i + W_2 \sum_{j \in \mathbb{E}(i)} x_j \tag{3}$$

We can see from this that the model both learns a set of weights directly related to the feature vector, as well as weights relating to the significance of the gathered values from nearby nodes.

3.3.2.2.2 Graph Convolutional layers (`GCNConv`)

Some of the proposed models use Graph Convolutional layers as proposed by Kipf and Welling (2017), implemented by PyG as `GCNConv` [30]. This layer implements what they describe as a "fast approximate convolution" on the graph in the following manner:

- Let A be the graph adjacency matrix and I be the identity matrix with the same shape as A. Note that I can be seen as an adjacency matrix where every node is connected with only itself.
- Let $A' = A + I$, i.e. the graph adjacency matrix with self connections.
- Let D be a diagonal matrix where each entry d_{ii} is equal to $\sum_j A'_{ij}$.

Then, the graph convolutional layer operation can be defined as follows:

$$X' = D^{-\frac{1}{2}} A' D^{-\frac{1}{2}} XW \tag{4}$$

3.3.2.3 Activation functions

If schedule deviation was not used, then the rectified linear unit (ReLU) activation function was used between hidden layers in the model (ReLU is simply $y = x$ if $x > 0$, 0 otherwise). When schedule deviation was used, tanh (hyperbolic tangent) was chosen as the activation function instead due to the need to propagate and return negative values in the range $(-1, 1)$, which is the range of tanh.

Note: Model A8 (Section 4.2.2) uses *LeakyReLU* between hidden layers, with no activation function immediately prior to output. *LeakyReLU*, implemented by PyTorch, is a modification of ReLU that multiplies negative values by a constant 0.01 instead of setting them to zero.

3.4 Training

From the collection process, 21 days of data are available for training, from the transit day starting on 9/10/2023 at 4 AM to the day starting on 9/30/2023 and ending at 9/31/2023 at 4 AM. Of this data, the first 14 days were used to train the model, and the last 7 to test. This test split was intentionally chosen to evaluate the model's forecasting performance, emulating how the model would perform in a scenario where it was trained on past data, and then applied to make predictions on real-time data. During training, the data is treated as one large dataset of feature matrix pairings, which is shuffled before each epoch. Adam was the optimizer used throughout the training process.

The training process for this model was an iterative one, as many different model / training features as well as hyper-parameters were explored as the model was fine-tuned in an effort to reach an optimal solution (see results section). The following sections describe the features and processes used to varying degrees throughout the training process.

3.4.1 Training system setup

Training was done on a single machine with a Tesla T4 GPU, two Intel Xeon Gold 6148 CPUs @ 2.40 GHz (40 cores total), and 96 GB of RAM. The machine runs Ubuntu 22.04, and training and evaluation scripts used Python 3.10.12, PyTorch 2.1.0, and PyTorch Geometric 2.4.0.

3.4.2 Metro Transit subset networks

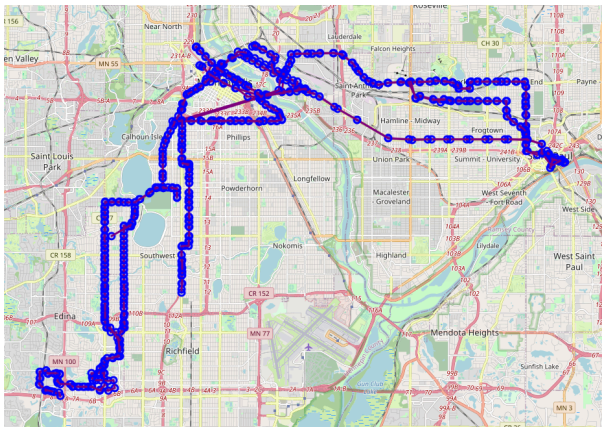

Figure 13: Map view of small network subset of the Metro Transit network.

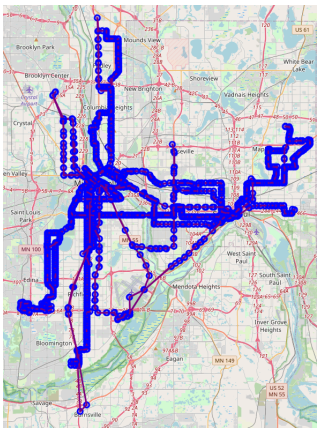

Figure 14: Map view of the Metro Transit High Frequency network.

Due to time and computational constraints, training on the entirety of the Metro Transit network proved to be outside of the scope of this project. Instead, the models outlined here are trained on two subsets of the Metro Transit network:

1. **A "small network"** (left): Specifically, local bus routes that serve the University of Minnesota, Twin Cities (Routes 2,3,6,113, and 114), plus the METRO Green Line LRT. Training typically took less than 30 minutes on this network, so it was used for a lot of experimentation.

2. **The High Frequency network** (right): This is a subset of the Metro Transit network specifically identified by the agency as having at most 15 minutes delay between vehicles during most times of the day. These 16 routes include all but one of the agency's BRT lines, both LRT lines, and several local bus routes of varying design throughout Minneapolis and St. Paul. Many of the most popular routes are in this subset, so it serves as a helpful benchmark for the system as a whole. Training on this network could take several hours, so most trials ran on this were done after being thoroughly tested on the small network.

3.4.3 Loss functions

Mean Absolute Error (MAE) and Mean Squared Error (MSE) were the two loss metrics used throughout the training and testing process for this model. MAE was selected primarily for its simplicity, as an intuitive and accurate measurement of delays with a linear penalty. MSE was also sometimes used as the loss function as a way to have a quadratic penalty, penalizing largely inaccurate predictions at a greater rate.

Because the output vectors of the model are sparse, loss values during training only fed features that were supposed to be non-zero (i.e. features that the model was aiming to predict). This was also always done when operating on the test set, since it is more reflective of the model's actual predictive performance.

Let y_{nz} be all non-zero values in the output matrix Y, and p_{nz} be all values in the prediction matrix P returned by the model that match the indices of the non-zero values from y_{nz} in Y. The loss formulas are then as follows:

$$\begin{aligned} \text{MAE} &= \frac{\sum |p_{nz} - y_{nz}|}{\|y_{nz}\|} \\ \text{MSE} &= \frac{\sum \left(p_{nz} - y_{nz}\right)^2}{\|y_{nz}\|} \end{aligned} \tag{5}$$

3.4.4 Regularization

Most models used dropout in an effort to prevent overfitting. This is the process where, during training, certain values between layers in the neural network are randomly set to zero (with a fixed probability) to prevent the model from becoming too reliant on any one feature or weight.

Instead of dropout, some models used L2 regularization, as implemented by the optimizer. Rather than a random process like dropout, L2 regularization adds an additional penalty to the loss value equal to the sum of the squares of all weights in the network, multiplied by a tuning parameter λ. This formula is:

$$\text{Loss} = \text{Error (MSE or MAE)} + \lambda \sum_{l_i \in L} \sum_{w_j \in l_i} w_j^2 \tag{6}$$

where L is the set of all layers in the model, and $w_j \in l_i$ represents the set of all weights in layer l_i.

3.4.5 Uniformity penalty

Many of the models were showing a tendency to make overly uniform predictions during training, often converging to predictions that amounted to predicting the exact same value for every trip regardless of stop. To counter this, a "uniformity penalty" was implemented where the model would be penalized when the standard deviation of predictions for a given trip was lower than the standard deviation of the ground truth arrival values. The formula for this penalty is as follows:

$$\begin{gathered}\text{penalty} = \sum_t \min\big(0, \sigma(t_t) - \sigma\big(t_p\big)\big) \\ \sigma(t_t) = \text{standard deviation of target values for trip t} \\ \sigma\big(t_p\big) = \text{standard deviation of actual model predictions for trip t}\end{gathered} \tag{7}$$

This penalty is multiplied by a penalty multiplier m and then added to the loss function, in addition to L2 regularization (if used).

3.4.6 Batching

The optimizer was incremented in mini-batches of various sizes throughout the training process.

3.4.7 Learning rate scheduling

An exponential learning rate decay was sometimes used in the training process. If used, the learning rate would be decayed by a fixed constant value after each epoch.

A cyclical learning rate scheduler was also used in some training trials, as first described by Smith (2017) and implemented by PyTorch [31]. When used, the learning rate would be oscillated between the initially defined learning rate and a new "max learning rate", cycling completely between the two every 4 complete epochs, modified with every batch.

3.4.8 Early stopping

Early stopping was implemented and used by some models in the training process. The early stopping implementation works by stopping training if the model has not improved performance on a randomly selected validation set (20% of the training set) within a specified number of epochs ("patience" parameter). The model that performed best on the validation set is returned.

3.4.9 Normalization

When schedule deviation was used, many model features tended to be quite small. To adjust for this, some models had schedule deviation values normalized to try to approximately fit into the range (-1,1). For simplicity's sake, with this feature enabled, deviation values in the input and output feature vectors were normalized so that predictions that were between –15 minutes and 15 minutes off schedule would fall in the range (-1,1), and offset so that zero was the equivalent of being 30 seconds behind.

Note that this was only done in an effort to aid the model in weighting features accurately and making predictions - all loss values and statistics in the results will have consistent metrics.

4 Results

As discussed above, the process of training this model was an iterative one, exploring many different subtle changes to the model architecture, parameters, hyperparameters, and training features (as is often the case with ML research). This section documents the meaningful results from this process, as well as discusses some of the training process and the insights it gave into the model's ability to operate on this problem.

4.1 Notation

Table 3: Name and description of properties identified in tables throughout the results section.

Field	Description
Model	Type of model (see Section 4.2)
MAE	Mean Absolute Error of model evaluated on test set, measured in minutes (i.e., MAE = 5 indicates a model which is on average 5 minutes off from the actual stop visit time at any given stop) (See Section 3.4.3)
LR	Training learning rate
BS	Training batch size
E	Number of training epochs
OPS	Prediction only schedule (see Section 3.3.1.3)
SD	Schedule deviation (see Section 3.3.1.3)
EXP	Exponential decay on learning rate scheduler, if used (see Section 3.4.7)
CYC	Max LR parameter on cyclical learning rate scheduler, if used (see Section 3.4.7)
N	Whether normalization was used (see Section 3.4.9)
ES	Early stopping patience parameter, if used (see Section 3.4.8)
WD	Weight decay parameter for L2 regularization, if used (see Section 3.4.4)
OC	Outlier cut parameter, in minutes (see Section 3.3.1.3)
UP	Uniformity penalty weight, if used (see Section 3.4.5)

4.2 Models

A number of different model architectures were explored during training, 13 of which are included in the results report. Their architectures and layers used are outlined in the table below.

4.2.1 Layer key

Table 4: Description of layer identifiers used in the model table (Section 4.2.2).

Layer ID	Description
L	Simple linear layer (Section 3.3.2.1)
GC	Simple message passing layer (Section 3.3.2.2.1)
GCN	Graph Convolutional layer (Section 3.3.2.2.2)

4.2.2 Model table

The following table notes the architectures of the various models explored in training. Each layer is denoted by its type, as well as the number of input and output elements per node in that layer - for example, $L(100, 50)$ indicates a linear layer where each input matrix had 110 elements per node and each output matrix had 50 elements.

Table 5: Structural description of models shown in results. Models with prefix 'S' use simple message passing layers (Section 3.3.2.2.1), models with prefix 'A' use Graph Convolutional layers (Section 3.3.2.2.2), and models with prefix 'SD' are similar to prefix 'S' but of greater depth.

Model	Description
S1	L(110,110)⟶GC(110,110)⟶GC(110,110)⟶GC(110,110)⟶GC(110,110)⟶L(110,110)
S3	L(330,678)⟶GC(678,678)⟶GC(678,678)⟶GC(678,678)⟶GC(678,678)⟶L(678,110)
S6	GC(110,110)⟶L(110,110)
S7	L(1290,2673)⟶GC(2673,2673)⟶GC(2673,2673)⟶L(2673,430)
S8	L(1290,2673)⟶GC(2673,2673)⟶GC(2673,2673)⟶GC(2673,2673)⟶GC(2673, 2673)⟶L(2673,430)
S11	L(330,678)⟶GC(678,678)⟶GC(678,678)⟶L(678,110)
S13	L(1062,2673)⟶GC(2673,2673)⟶GC(2673,2673)⟶L(2673,354)
S14	L(110,110)⟶GC(110,110)⟶GC(110,110)⟶L(110,110)
S18	L(330,678)⟶GC(678,678)⟶L(678,110)
A2	GCN(1062,2673)⟶GCN(2673,2673)⟶GCN(2673,2673)⟶GCN(2673,1062)⟶L(1062,354)
A8	GCN(1290,2673)⟶GCN(2673,8019)⟶GCN(8019,5346)⟶GCN(5346,1290)⟶L(1290,430)
SD1	L(1290,1290)⟶GC(1290,1290)⟶GC(1290,1290)⟶GC(1290,1290)⟶GC(1290, 1290)⟶GC(1290,430)⟶GC(430,430)⟶GC(430,430)⟶GC(430,430)⟶GC(430, 430)⟶GC(430,430)⟶L(430,430)
SD2	L(330,330)⟶GC(330,330)⟶GC(330,330)⟶GC(330,330)⟶GC(330,330)⟶GC(330, 110)⟶GC(110,110)⟶GC(110,110)⟶GC(110,110)⟶GC(110,110)⟶GC(110,110)⟶L(110, 110)

4.3 Small network

In total, 111 training trials were ran on the small network dataset. Of these trials, the most notable ones (highlighting insights learned as different training features were explored) are shown below. Results are sorted by MAE (in minutes).

Table 6: Small network notable results.

Model	MAE	LR	BS	E	OPS	EXP	SD	CYC	N	ES	WD	OC	UP
SD2	3.38	0.001	8	100	N	0.95	Y	N	Y	5.0	0.01	30.0	N
S12	3.56	0.001	8	100	N	0.95	Y	N	Y	10.0	0.05	30.0	N
S12	3.59	0.001	8	100	N	0.95	Y	N	Y	10.0	0.01	30.0	1.0
S12	3.94	0.001	256	100	N	0.95	Y	N	Y	5.0	0.01	30.0	0.1
S17	6.18	0.001	8	100	N	0.95	Y	N	Y	10.0	0.005	N	N
S7	6.18	0.001	8	100	N	0.9	Y	N	Y	10.0	N	N	N
S16	6.2	0.001	8	100	N	0.9	Y	N	Y	10.0	N	N	N
S11	6.23	0.0001	8	100	N	N	Y	N	Y	10.0	N	N	N
S11	6.24	0.0001	8	100	N	N	Y	0.001	Y	10.0	N	N	N
S11	6.25	0.001	8	50	N	0.9	Y	N	Y	N	N	N	N
S15	8.38	0.001	8	50	N	0.9	Y	N	N	N	N	N	N
S15	8.84	0.001	2	50	Y	0.9	N	N	N	N	N	N	N

4.3.1 Takeaways

From the data above, we can observe the following:

1. **Outlier removal helped model convergence significantly.** Many of the stop crossing optimizations outlined in Section 3.1.4.5 were found to significantly improve convergence performance of the model. Nonetheless, the additional step of avoiding training the model on unreasonable data was very helpful in increasing model performance.
2. **The benefits of weight decay vs. dropout were unclear.** Weight decay (L2 regularization) was explored as a less stochastic alternative to dropout (due to the model being able to learn which features didn't matter). However, model performance did not improve meaningfully as a result of this change.
3. **The uniformity penalty did not make a significant impact on MAE**. However, knowing the penalty is present with a similar MAE value does place a higher confidence in the predictions made, because it ensures that the model has done an additional level of learning compared to falling into a simple prediction of the same deviation value for every stop.

Additionally, some observations not seen here but encountered throughout the training process include:

1. **Benefits of hyperparameter tuning were noticeable, but limited.** After some experimentation, a learning rate of 0.001 and batch size of 8 were generally determined to be optimal for this problem. Nonetheless, model performance was generally relatively stable even with different batch sizes. Regarding the learning rate, 0.001 appeared to be optimal, with higher and lower learning rates struggling to remain stable in convergence, but nonetheless not offering an extremely significant difference in performance. Number of epochs also did not usually matter, and early stopping would often end model training prior to epoch 50 regardless.
2. **Only providing predicted schedule information did not significantly impact performance, but schedule deviation did.** Schedule deviation was likely helpful for the model as it made the model input and output features more uniform when it came to time of day. However, restricting the schedule information to only the predicted features did not appear to significantly improve model performance, although the model was still able to learn similarly without the full schedule information.

4.4 High frequency network

As the small network was being trained and refined, 9 high frequency network tests were ran based off of the lessons learned from the small network training process. These results are shown below.

Table 7: High frequency network results.

Model	MAE	LR	BS	E	OPS	SD	EXP	CYC	N	ES	WD	OC	UP
A8	3.71	0.001	8	100	N	Y	0.95	N	Y	7.0	0.01	30.0	1.0
S3	4.18	0.001	8	150	N	Y	0.95	N	Y	15.0	0.05	30.0	N
S3	4.87	0.001	8	100	N	Y	0.95	N	Y	7.0	0.01	30.0	0.15
S3	5.07	0.001	8	100	N	Y	0.95	N	Y	7.0	0.01	30.0	0.25
SD1	5.35	0.001	8	100	N	Y	0.95	N	Y	7.0	0.01	30.0	0.25
S6	7.39	0.001	8	100	N	Y	0.9	N	Y	5.0	N	N	N
S6	8.72	0.0001	8	100	N	Y	N	0.001	Y	10.0	N	N	N
S14	21.9	0.001	8	50	N	Y	0.9	N	Y	N	N	N	N
A1	77.65	0.001	32	35	Y	N	N	N	N	N	N	N	N

4.4.1 Takeaways

In general, high frequency network performance matched that of the model on the small network, with only slight increases in average error values. Worth noting is that the highest performing model on this dataset was with a high uniformity penalty value, which may lead to more meaningful predictions, and possibly indicates an increased value in the uniformity penalty on larger networks.

4.5 Comparison to schedule, MT prediction performance

As a benchmark, it is helpful to compare the performance of our model against the existing Metro Transit predictions, as well as against a simple model that always predicts zero schedule deviation for all stops (in effect, predicting that the schedule is always 100% correct). These results were computed for both the small and high-frequency network subsets of the transit network.

Table 8: MAE values (in minutes) for predictions made by the existing Metro Transit system, as well as the schedule-based model and the best proposed model for each network.

Network	MAE, current predictions	MAE, schedule	MAE, best proposed model
Small	2.685	6.502	3.38
HF	2.831	6.988	3.71

In the interest of fairness, the Metro Transit predictions are evaluated based off of the approximate stop crossing times as interpreted from their feed, instead of based off of the stop crossing algorithm used to train the model. This approach is done so that the Metro Transit prediction system performs as well as possible when compared to the results from this thesis. Stop crossing times are approximated from the Metro Transit prediction feed by recording the last time a prediction at a stop for a given trip is reported, discounting events where the predicted times are in the past (kept in the feed to note that a vehicle was early). This is consistent with the realtime GTFS feed specification for TripUpdates [32].

From these results, we can see that the best proposed GNN models have successfully managed to make predictions that are better than predicting the schedule, and close to the accuracy of the existing predictions (albeit still less accurate).

5 Analysis

Based off of MAE values, it is clear that the best models for both the small and high-frequency network perform meaningfully better than simply predicting the schedule, but are marginally yet non-trivially less accurate than the existing Metro Transit prediction method using a Kalman filter. However, a single error metric across all predictions on all days of the test set does not give an accurate understanding of the models performance. Thus, this section seeks to more fully analyze the models performance across days and other factors, as well as more fully compare it to the Metro Transit existing implementation.

All analysis is done here on the best model trained on the high frequency network. For cases where error is signed, error is computed here as the difference between the target arrival time and the predicted arrival time ($\text{target} - \text{predicted}$), i.e. that positive error indicates a prediction that was too early, and negative error too late.

5.1 Overall performance, comparison to Metro Transit

5.1.1 Error distribution

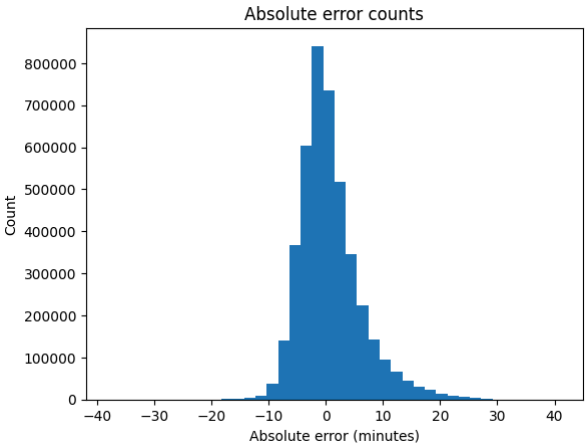

Figure 15: Best model error distribution.

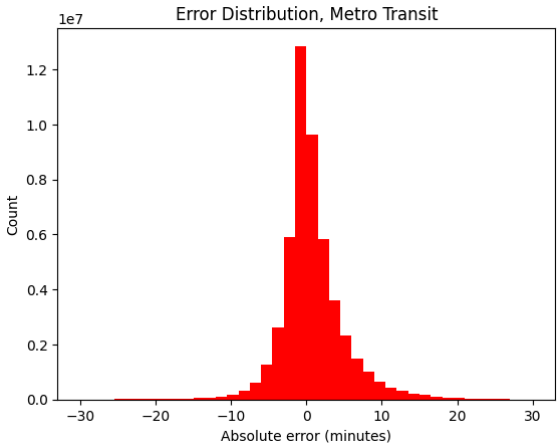

Figure 16: Metro Transit prediction error distribution.[1]

From the error distributions in Figure 15 and Figure 16, we can see that both our model and the existing Metro Transit predictions roughly follow a normal distribution centered around 0, with the bulk of predictions occurring within a few minutes accuracy, and the vast majority ± 10 minutes. Tracking with the overall MAE values, the Metro Transit predictions are more tightly distributed around 0 minutes of error compared to our models.

5.1.2 Prediction quality

When evaluating the performance of a real-time prediction system such as this, it is helpful to see the proportions of ranges of prediction accuracy in a way that models the perceived quality of predictions that a human user would have when using your system. This is especially helpful for arrival prediction, where a prediction being early is not as bad as being late (when that could potentially lead to someone relying on the prediction and missing their bus or train). To that end, the following two charts showcase our model and Metro Transit's proportional prediction accuracy (matching the error distributions in Section 5.1.1), with error ranges and colors selected to approximate prediction quality.

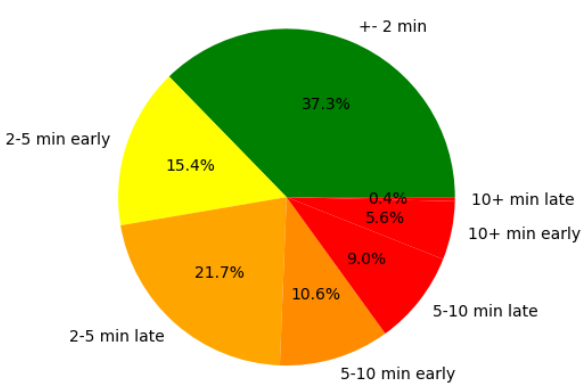

Figure 17: Model prediction quality proportions.

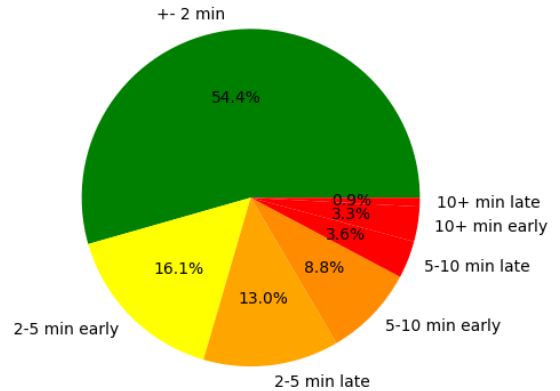

Figure 18: Metro Transit prediction quality proportions.

[1]The y axis on this chart is larger than Figure 15 because Metro Transit predictions were sampled more frequently than our model was evaluated, as well as likely due to differences in how edge cases and missing predictions are handled. Nonetheless, comparing the error distributions is still valuable.

From Figure 17 and Figure 18, we can see that the Metro Transit predictions solidly outperform our own, especially when it comes to very good quality predictions (within two minutes of the actual arrival). Our model also seems to be more likely to make late predictions than the Metro Transit model, which is obviously not helpful. That being said, our model still produces reasonable predictions a majority of the time, with only a reasonably small portion of predictions falling into the worst category.

5.2 Model performance by day, time of day

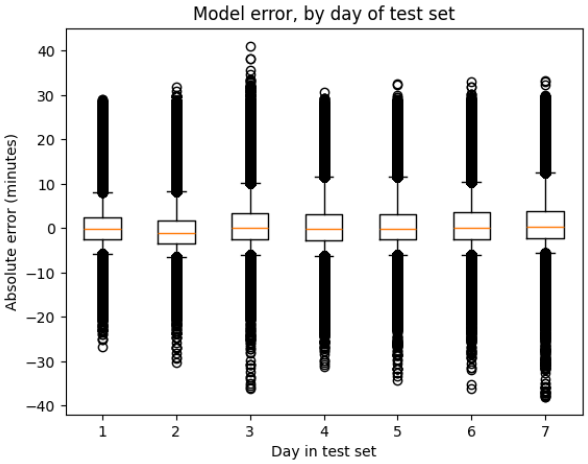

Figure 19: Model error, by day of test set. Whiskers for this and future box plots are set at the 5th and 95th percentile, meaning that 90% of values fall between the two.

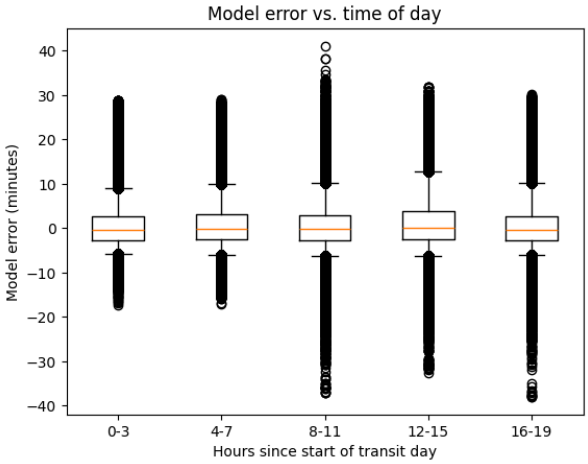

Figure 20: Model error, by time of day.

From Figure 19 and Figure 20, we can see that model performance did not vary substantially by day within the test set, or by time of day, save for marginally higher positive error in the later hours of the transit day. It is interesting to note that the distribution of the worst 5% of predictions appears to be better earlier in the day when it comes to early predictions.

5.3 Model performance by prediction difficulty

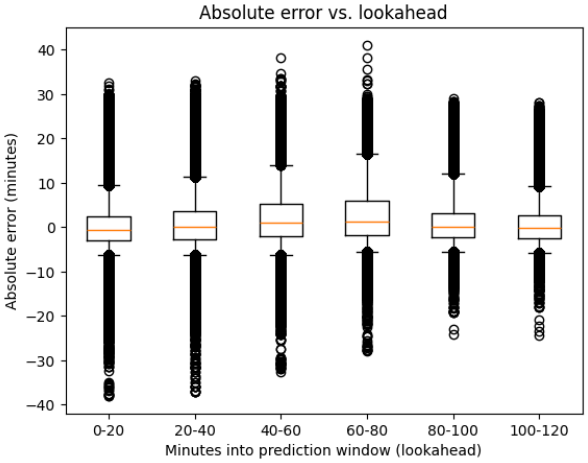

Figure 21: Model error, by minutes into the prediction window (lookahead).

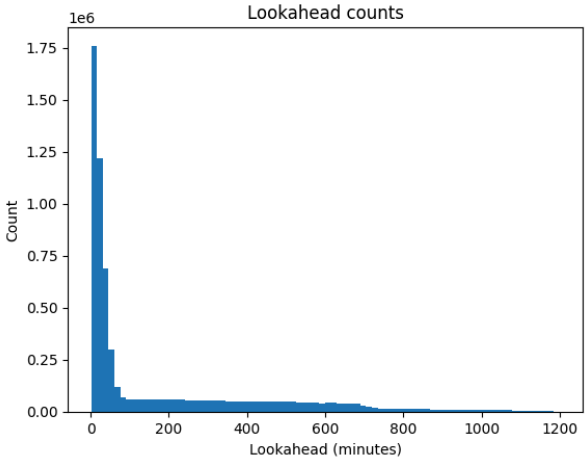

Figure 22: Distribution of lookahead values (minutes into the prediction window).

Making predictions further into the future is necessarily more difficult than closer predictions. Because of this, it is worth looking into the model's performance compared to the amount of time until the prediction actually comes to pass (referred to here as "lookahead"). In terms of the windowing tech-

nique used by this model, this amounts to the amount of time between the current time t and the actual departure time $t_{i,k}$.

From Figure 21, we can see that the model does seem to perform better on predictions that are coming sooner / earlier in the prediction window. Specifically, we can see that predictions tend towards greater positive error (early prediction) as we go further into the prediction window. This is the case until 80 minutes into the prediction window, where distributions appear to tend towards something similar to the predictions for the first 20 minutes of the window.

5.3.1 Prediction difference error

From Figure 22, we can see that the vast majority of values fall within 50-60 minutes of the prediction window. However, there is a significant amount that falls outside of that, most of which is likely erroneous, a result of errors earlier in the data pipeline. This is likely leading to some model error across the board, and also possibly explains the strange behavior of predictions 80+ minutes into the prediction window in the left chart (considering that it is likely there are very few trips present for more than 80 minutes in the network).

5.4 Bias and variance compared to training set

Table 9: Bias, variance and standard deviation values for training and test set on best high frequency model. Bias is computed as the average signed error of all predictions across the dataset. Variance and standard deviation are similarly calculated on average signed error.

Dataset	Bias (minutes)	Variance (minutes^2)	Standard deviation (minutes)
Training set	0.715	27.693	5.262
Test set	0.686	27.140	5.210

Examining the bias and variance values of the model's error on the training and testing sets can provide insights into its ability to generalize well and fit to the data. As we can see in Table 9, there is no substantial difference in the bias, variance, or standard deviation values between the training and testing sets. This likely indicates that the model is not over-fitting to the training set, which is also consistent with results observed during training where test set loss values were not significantly worse than the training set.

Looking at the values themselves, the bias for both sets is low (under one minute). This indicates that the model is not severely under-fitting to the data, something that the uniformity penalty was implemented to reduce. However, the variance and standard deviation values are larger, which is likely reflective of the models sensitivity to input data noise and error, as discussed in Section 5.3.1 and Section 6.2.1 and generally seen throughout the analysis section.

In summary, the consistent values across the training and testing sets, as well as the low bias values, seem to indicate that the model is generally generalizing well to new data, without significant over or under-fitting. High variance (or standard deviation) values are likely reflective of other issues seen throughout model analysis, including input noise.

6 Conclusion

Based off of the results and discussion, it is clear that a GNN model is a promising application towards making transit arrival time predictions in real-time. However, the performance of the model implemented in this thesis on subsets of the Metro Transit network was acceptable, but subpar compared to the existing Kalman filter implementation. With these realities in mind, we can draw a few conclusions about the nature of this model and the use of Graph Neural Networks to solve this problem.

6.1 Successes

One obvious success seen in the model implemented for this thesis is the ability of the model to generalize to different routes and subsets of the network without additional tuning. This is evidenced by the similar performance on the small network and high frequency network with the same model design and parameters. Additionally, much of the model design is relatively Metro Transit agnostic, meaning that the general principles used here could be applied to make predictions on another network entirely.

Another benefit of the model implemented here is the ease by which it is able to simultaneously make predictions across the network it is trained on. This graph-wide operation is a major advantage of Graph Neural Networks, as it allows for the training and fine-tuning of a single model with minimal to no need to tune for specific routes or peculiarities in the network. The result of both of these benefits is that the GNN model is able to effectively make predictions on the network in a single operation, taking advantage of the graph structure of the transit network to do so.

As a whole, the performance of this model shows that the benefits that come from directly applying a Graph Neural Network to a transit network are worthwhile, even if the current results of applying it are not as accurate as the existing solution for Metro Transit. With greater exploration and care, a GNN based system would certainly prove even more valuable, especially in generalizing to the transit network as it grows and changes with time.

6.2 Difficulties

Applying a GNN model to this problem did not come without drawbacks and difficulties, which should be considered when implementing a GNN model to solve this problem:

1. **Model size**: While certainly far from the largest, the GNN models used in this thesis proved to be relatively large, and the size of them scaled with the size of the network used, and with that training time. While the training times and model sizes used here should be well within capacity for a large agency like Metro Transit, it is something to be considered, especially in its ability to increase development complexity.
2. **Model complexity**: While the application of a GNN to the transit network is relatively simple from a conceptual standpoint (considering that it is intuitive to apply a Graph Neural Network to a transit network easily represented as a graph), in practice this very general approach also leads to a more complex model designed to handle such large operations simultaneously. This complexity brings with it the difficulties of training such large, deep neural networks, and the training results are evidence of the difficulty in making good parameter and design choices. Also, while this model proved to be relatively stable when it came to hyperparameter tuning, the model was also very sensitive to outliers and problems within the training set, increasing the difficulty in training the model effectively.

6.2.1 Sources of error

It is worth noting that sources of error from the implementation of components in the data pipeline likely contributed to difficulties in model convergence (such as discussed in Section 5.3.1). Additionally, while Metro Transit feed data was generally very reliable throughout this project, errors in that dataset certainly contribute to difficulties in both our model and the existing Metro Transit predictions. With that being said, more careful implementation and testing of data pipeline components, especially involving time data, would likely help to refine model data quality and improve performance.

6.3 Future work

Beyond care and refinement of processes used in this thesis, exploring and experimenting with some of the specific advantages of Graph Neural Networks (specifically message passing) would be a promising future area of research on this topic. This thesis did not explore this in much detail, but it is likely that

a refined and streamlined message passing approach would lead to predictions with greater accuracy and responsiveness to network conditions. In a similar vein, exploring changes to the network structure to promote communication between nodes in the network (such as connecting all stops in a route to all other stops along the route) would be an interesting future area of study. In addition to all of these areas, exploration of training on a larger subset of the Metro Transit network, or on a larger system altogether, would be valuable in further testing the model's performance at scale.

Bibliography

[1] Google, "Predicting Bus Delays with Machine Learning". Accessed: Feb. 07, 2023. [Online]. Available: https://ai.googleblog.com/2019/06/predicting-bus-delays-with-machine.html

[2] R. Jeong and L. R. Rilett, "Prediction Model of Bus Arrival Time for Real-Time Applications", *Transportation Research Record*, 2005.

[3] T. Yin, G. Zhong, J. Zhang, S. He, and B. Ran, "A prediction model of bus arrival time at stops with multi-routes", *Transportation Research Procedia*, vol. 25, pp. 4623–4636, Jan. 2017, doi: 10.1016/j.trpro.2017.05.381.

[4] Z.-Y. Xie, Y.-R. He, C.-C. Chen, Q.-Q. Li, and C.-C. Wu, "Multistep Prediction of Bus Arrival Time with the Recurrent Neural Network", *Mathematical problems in engineering*, vol. 2021, pp. 1–14, 2021, doi: 10.1155/2021/6636367.

[5] A. A. Agafonov and A. S. Yumaganov, "Bus Arrival Time Prediction Using Recurrent Neural Network with LSTM Architecture", *Optical Memory and Neural Networks*, vol. 28, no. 3, pp. 222–230, Jul. 2019, doi: 10.3103/S1060992X19030081.

[6] E. Chondrodima, H. Georgiou, N. Pelekis, and Y. Theodoridis, "Particle swarm optimization and RBF neural networks for public transport arrival time prediction using GTFS data", *International Journal of Information Management Data Insights*, vol. 2, no. 2, p. 100086, Nov. 2022, doi: 10.1016/j.jjimei.2022.100086.

[7] E. Jenelius and H. Koutsopoulos, "Travel time estimation for urban road networks using low frequency probe vehicle data", *Transportation Research Part B: Methodological*, vol. 53, pp. 64–81, Jun. 2013, doi: 10.1016/j.trb.2013.03.008.

[8] B. Qiu and W. (. Fan, "Machine Learning Based Short-Term Travel Time Prediction: Numerical Results and Comparative Analyses", *Sustainability*, vol. 13, no. 13, p. 7454, Jan. 2021, doi: 10.3390/su13137454.

[9] J. Wang, R. Chen, and Z. He, "Traffic speed prediction for urban transportation network: A path based deep learning approach", *Transportation Research Part C: Emerging Technologies*, vol. 100, pp. 372–385, Mar. 2019, doi: 10.1016/j.trc.2019.02.002.

[10] T. Li, A. Ni, C. Zhang, G. Xiao, and L. Gao, "Short-term traffic congestion prediction with Conv–BiLSTM considering spatio-temporal features", *IET Intelligent Transport Systems*, vol. 14, no. 14, pp. 1978–1986, 2020, doi: 10.1049/iet-its.2020.0406.

[11] S. Choi, H. Yeo, and J. Kim, "Network-Wide Vehicle Trajectory Prediction in Urban Traffic Networks using Deep Learning", *Transportation Research Record*, vol. 2672, no. 45, pp. 173–184, Dec. 2018, doi: 10.1177/0361198118794735.

[12] "TheTransitClock.github.io". Accessed: May 01, 2023. [Online]. Available: https://thetransitclock.github.io/

[13] A. Achar, A. Natarajan, R. Regikumar, and B. A. Kumar, "Predicting public transit arrival: A non-linear approach", *Transportation Research Part C: Emerging Technologies*, vol. 144, p. 103875, Nov. 2022, doi: 10.1016/j.trc.2022.103875.

[14] T. Elliott and T. Lumley, "Modelling the travel time of transit vehicles in real-time through a GTFS-based road network using GPS vehicle locations", *Australian & New Zealand Journal of Statistics*, vol. 62, no. 2, pp. 153–167, 2020, doi: 10.1111/anzs.12294.

[15] D. Zhang *et al.*, "Prediction of Train Station Delay Based on Multiattention Graph Convolution Network", *Journal of Advanced Transportation*, vol. 2022, p. e7580267, Feb. 2022, doi: 10.1155/2022/7580267.

[16] Z. A. Sahili and M. Awad, "Spatio-Temporal Graph Neural Networks: A Survey". Accessed: Apr. 15, 2023. [Online]. Available: http://arxiv.org/abs/2301.10569

[17] Z. Li, P. Huang, C. Wen, and F. Rodrigues, "Railway Network Delay Evolution: A Heterogeneous Graph Neural Network Approach". Accessed: Apr. 17, 2023. [Online]. Available: http://arxiv.org/abs/2303.15489

[18] J. Ma, J. Chan, S. Rajasegarar, and C. Leckie, "Multi-attention graph neural networks for city-wide bus travel time estimation using limited data", *Expert Systems with Applications*, vol. 202, p. 117057, Sep. 2022, doi: 10.1016/j.eswa.2022.117057.

[19] C. Li, L. Bai, W. Liu, L. Yao, and S. T. Waller, "Graph Neural Network for Robust Public Transit Demand Prediction", *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 5, pp. 4086–4098, May 2022a, doi: 10.1109/TITS.2020.3041234.

[20] F. Zhou, Q. Yang, T. Zhong, D. Chen, and N. Zhang, "Variational Graph Neural Networks for Road Traffic Prediction in Intelligent Transportation Systems", *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2802–2812, Apr. 2021, doi: 10.1109/TII.2020.3009280.

[21] J. Ji, F. Yu, and M. Lei, "Self-Supervised Spatiotemporal Graph Neural Networks With Self-Distillation for Traffic Prediction", *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 2, pp. 1580–1593, Feb. 2023, doi: 10.1109/TITS.2022.3219626.

[22] H. Li, D. Jin, X. Li, H. Huang, J. Yun, and L. Huang, "Multi-View Spatial–Temporal Graph Neural Network for Traffic Prediction", *The Computer Journal*, p. bxac86, Jul. 2022b, doi: 10.1093/comjnl/bxac086.

[23] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey", *Expert systems with applications*, vol. 207, p. 117921--, 2022, doi: 10.1016/j.eswa.2022.117921.

[24] "Traffic prediction with advanced Graph Neural Networks". Accessed: Apr. 17, 2023. [Online]. Available: https://www.deepmind.com/blog/traffic-prediction-with-advanced-graph-neural-networks

[25] L. Wu, Y. Tang, P. Zhang, and Y. Zhou, "Spatio-Temporal Heterogeneous Graph Neural Networks for Estimating Time of Travel", *Electronics (Basel)*, vol. 12, no. 6, p. 1293--, 2023, doi: 10.3390/electronics12061293.

[26] D. Singh and R. Srivastava, "Graph Neural Network with RNNs based trajectory prediction of dynamic agents for autonomous vehicle", *Applied Intelligence*, vol. 52, no. 11, pp. 12801–12816, Sep. 2022, doi: 10.1007/s10489-021-03120-9.

[27] W. Liang *et al.*, "Spatial-Temporal Aware Inductive Graph Neural Network for C-ITS Data Recovery", *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–12, 2022, doi: 10.1109/TITS.2022.3156266.

[28] M. Fey and J. E. Lenssen, “Fast Graph Representation Learning with PyTorch Geometric”, in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[29] C. Morris *et al.*, “Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks”. 2021.

[30] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks”. 2017.

[31] L. N. Smith, “Cyclical Learning Rates for Training Neural Networks”. 2017.

[32] “Trip Updates - General Transit Feed Specification”. [Online]. Available: https://gtfs.org/realtime/feed-entities/trip-updates/