Using Message Passing Networks to Predict Train Arrival Times

Daniel Melcer
melcer.d@northeastern.edu

Jonathan Corzo
corzo.j@northeastern.edu

Samuel Steiner
steiner.s@northeastern.edu

April 2021

1 Objectives and Significance

Each week, hundreds of thousands of riders rely on the the timely operation of the MBTA's rapid transit network, one of the busiest systems in North America. A single train can carry hundreds of people, meaning that even small delays can result in many wasted person-hours. This led us to analyze the data that the MBTA publishes, and attempt to predict train arrival times and delays accurately.

We would like to use Machine Learning to perform these predictions, but we believe that standard neural networks do not adequately represent the complex temporal dependencies between successive trains on the same line. To overcome this, we developed a message passing neural network (MPNN), which allows us to model the state of every active train and station. We trained and tested our model, as well as several baseline models, on MBTA data from 2019. Our architecture achieves about the same mean absolute error as most of the baseline methods, but outperforms the baselines on the root mean square error.

2 Background

2.1 Predicting Train Delays

Several studies have been conducted in an attempt to correlate common delay factors with actual train arrival times. Wang and Zhang [1] used weather records, historical train delay records, and train schedule data across 75 stations along the Beijing–Guangzhou Chinese railway line, and applied gradient-boosted regression tree analysis to predict delays. While their model was able to somewhat accurately predict which trains would be extremely delayed—by 80 minutes or more—it did not produce accurate results for short and moderate-length delays.

Nilsson and Hanning compared several methods of predicting train delays in Stockholm [2]. They tested a neural network, as well as several variants of decision tree models. The neural network models outperformed all other prediction methods by a nontrivial margin.

2.2 Graph Neural Networks and Message Passing

The idea of a Message Passing Neural Network first originated in the context of graphs. It is difficult to represent an entire graph as the input to a neural network. One approach proposed to tackle this problem, inspired by the architecture of a Recurrent Neural Network (RNN), is to store a hidden state for each node. Then, repeatedly run an algorithm: at each time step, "messages" are passed between graph nodes until the values of the hidden states converge [3].

More specifically, a Graph Neural Network (GNN) algorithm typically consists of three operations. First, each graph node may generate a "message" vector. The contents of this vector are generated by a neural network which accepts the hidden state of the graph node. Second, the node's neighbors receive this message. A second neural network takes as input the receiver's hidden state, and the contents of the message vector. It outputs a new hidden state for the receiver. Lastly, after some convergence criterion is met, the values of each node's hidden state are fed through a third neural network (a "readout function") to obtain the final result for the classification or regression task. Recently, GNNs have been used to model queuing communication networks, obtaining accurate delay predictions for both synthetic and real-life networks [4].

Graph Neural Networks have continued to evolve. For example, the Gated Recurrent Unit (GRU) update function was developed to improve performance on RNNs [5]. Li et al. responded by creating Gated Graph Geural Networks (GG-NNs), which incorporate GRUs into the update function upon receiving a message [6]. This architecture reflects that the message should merely update the hidden state, not overhaul it entirely.

However, GNNs still only took as input a static graph, and resulted in a single prediction. Battaglia et al. [7] generalized the message passing functionality of GNNs to allow for node-level effects at different times, and complex interactions between different graph nodes . Gilmer et al. compared several variations of this idea to predict the properties of molecules in quantum chemistry [8]. They refer to the general idea as Message Passing Neural Networks (MPNN), and generalize such networks further to allow for some message and readout functions that were not previously explored. We use this concept of a MPNN as a starting point for our train delay prediction system.

3 Proposed Methodology

3.1 Intuition and Overview

The idea behind our architecture is that each train and each station carries some hidden state that is not present in the data. In the real world, this hidden state may include the crowdedness of a train or platform, the inflow of passengers from a nearby event, the proximity of a train to the one in front of it, the operator's driving style, the effect of weather conditions on a track, or any number of other (possibly stateful) factors that may influence the train's dwell time at a station and the travel time betweeen stations. We cannot include all of these as explicit features, so we hope that our method is able to learn any important factors automatically.

The question is then how can we model these complex interactions between the 20 stations on the line, and the variable number of trains that are running. We chose to use a MPNN with two node types, as we can quite naturally translate the interactions between every component into a concrete model. Each time that a train departs from a station, the train and the station exchange "messages" that update both of their hidden states. These messages reflect the fact that when a train stops at a station, the real-world state of the train and station both change in interconnected ways. Additionally, we occasionally send a message to the station with a weather overview and crowding information (the number of fare gate entries), allowing it to update its hidden state to incorporate this if it turns out to be useful.

3.2 Datasets

For this project we used data published by the MBTA, containing information about train arrivals and departures [9]. We focused specifically on the Orange line northbound route, from Forest Hills to Oak Grove. The data was collected from 2016 to the present. Each year of data contains approximately 47,000 trips, totalling approximately 1.7 million individual arrival and departure events. Additionally, the MBTA publishes the number of fare gate entries at each station, with 30 minute granularity. We chose to use the data from 2019, thereby excluding the irregularities in ridership caused by the COVID-19 pandemic. We also augmented the dataset with the local weather conditions, collected by the weather station at Boston Logan International Airport, using the Meteostat API [10].

3.3 Message Passing Neural Network

Our prediction pipeline is split into two major components: the "State Vector System" (SVS) and the "Prediction Vector System" (PVS).

Note that we use the following terms in this section:

- Difference from mean (DFM): for a given feature, we normalize the data so that the mean of this feature is zero.
- Train timestamp: The number of seconds that have passed since a given train left Forest Hills.
- Dwell time: The number of seconds between a train's arrival at and departure from a station.
- Headway: For a given departure, the number of seconds that have passed since the last train's departure from the same station.

3.3.1 State Vector System

SVS contains two main entities: trains, and stations. A train is a vector in $\mathbb{R}^t$, and a station is a vector in $\mathbb{R}^s$.

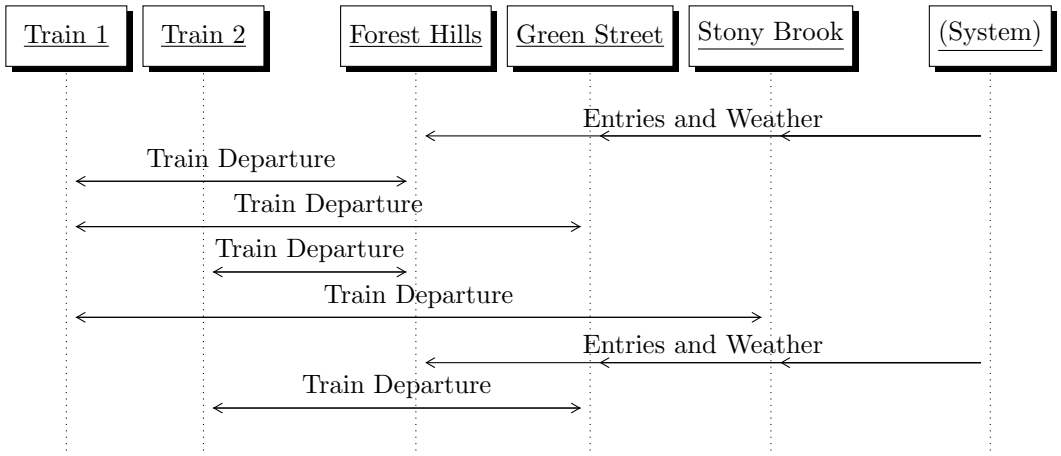

Figure 1: An illustration of SVS over time. At the beginning of the day, all stations start off with their (learned) initial state vector. Every 30 simulated minutes, the current weather and the number of fare gate entries are sent to each station as a "message". The station's state vector and the message are fed through a neural network to obtain a new station state vector (internally, this network uses a GRU architecture, where the station's state is used as the hidden GRU state). New trains are initialized with the initial train vector. When the train departs a station, information related to the train's departure is encoded as another message. The train's state vector, the station's state vector, and the message are used as the input to another neural network. This network outputs a new train vector, and a new station vector.

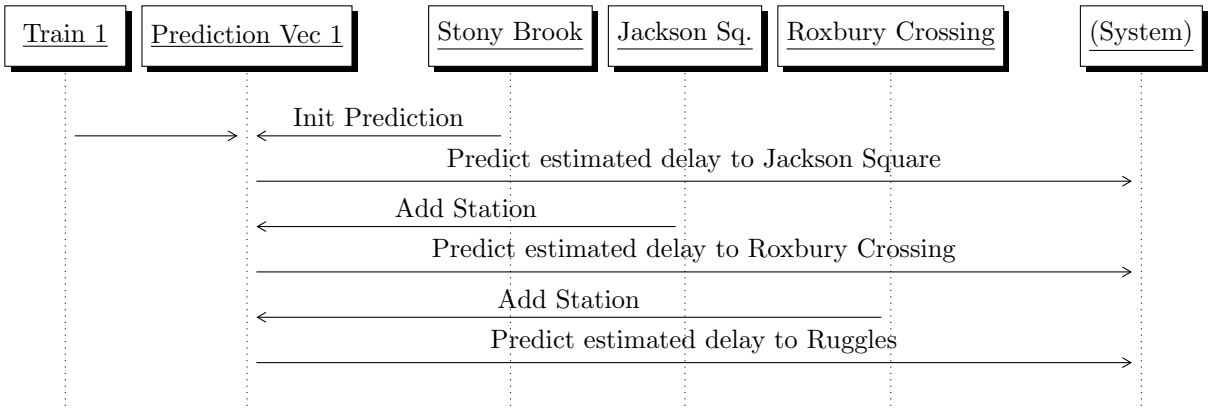

Figure 2: An overview of how PVS predicts the delays to a given station. Train 1 has just now departed from Stony Brook. We first call P_{init} with the train vector, and Stony Brook's station vector to obtain a prediction vector. $P_{predict}$ is used on this prediction vector to obtain the estimated delay to the next station, Jackson Square. Then, $P_{station}$ is called with the prediction vector and Jackson Square's station vector, "adding" any accumulated delays that are likely to occur as the train passes this station. When $P_{predict}$ is called on this new prediction vector, the result will be the estimated delay from Stony Brook to Roxbury Crossing. The process is repeated until the end of the line, where $P_{station}$ is used to include Malden Center's station vector, and then $P_{predict}$ is called one last time to predict the overall delay between Stony Brook and Oak Grove.

We define the vector $\theta_T \in \mathbb{R}^t$ to represent the initial train vector. For all stations s, we define $\theta_{S,s} \in \mathbb{R}^s$ to be the initial station vectors for a given day. Lastly, we define θ_{depart} and $\theta_{entries\_and\_weather}$ to be neural network weights.

SVS uses a number of functions to build up state vectors:

- $M_{depart}(t : \mathbb{R}^t, s : \mathbb{R}^s, ts : \mathbb{R}, dwl : \mathbb{R}, hd : \mathbb{R} | \theta_{depart}) \to \mathbb{R}^t \times \mathbb{R}^s$. When a train departs from the station, M_{depart} is called with the current train vector, current station vector, the train timestamp (DFM for this station), the train's dwell time (DFM for this station), and the headway (also DFM). This outputs a new train vector, and a new station vector.
- $M_{entries\_and\_weather}(s : \mathbb{R}^s, p : \mathbb{R}, w : \{\text{“nice”, “fog”, “rain”, “freezing rain”, “sleet”, “snow”, “hail”}\} | \theta_{entries\_and\_weather}) \to \mathbb{R}^s$. Every 30 simulated minutes, $M_{entries\_and\_weather}$ is called with the current station weather, the number of fare gate entries (DFM), and the approximate weather conditions at the station. It returns a new station vector.

For a given simulated day, all station vectors are initialized to $\theta_{S,s}$. Throughout the simulated day, $M_{entries\_and\_weather}$ is called with the current weather conditions and the number of gate entries. The station vector is replaced with the output of this function.

When a new train is dispatched at the beginning of the line, it is initialized with θ_T. As it encounters each station, the M_{depart} function is called for this train and each successive station. Both the train and the station vectors are updated to the output of this function. This process is visualized in Figure 1. In practice, this process is structured like the following pseudocode:

```
input: initial train vector t0
       initial station vectors s[0]..s[19]
       train departure events
       weather and fare_gate_entries datasets
       neural network weights theta_depart, theta_entries_and_weather

n = number of unique trains seen in the train departure events
t[0]..t[n-1] = initial train vector t0

time = first train departure event of the day, rounded down to the half hour
while time < last train departure event of the day:
    for each station s[i]:
        s[i] = M_entries_and_weather(s[i], fare_gate_entries[i, time],
                                     weather[time], theta_entries_and_weather)

    for each train departure event t_event between time and (time + 30 minutes):
        t[t_event.train_id], s[t_event.station] =
```

```
            M_depart(t[t_event.train_id], s[t_event.station], t_event.timestamp,
                     t_event.dwell_time, t_event.headway, theta_depart)

    time += 30 minutes
```

Thus, at any given point during the simulated day, we have a station vector for every station, and a train vector for each currently moving train. However, these vectors are semantically meaningless without PVS.

3.3.2 Prediction Vector System

A prediction vector is defined as a member of $\mathbb{R}^p$. We introduce three new neural network weights, θ_{p0}, θ_{ps}, and θ_{pt}.

PVS consists of three functions, all implemented as neural networks:

- $P_{init}(t : \mathbb{R}^t | \theta_{p_0}) \rightarrow \mathbb{R}^p$. It maps a train to a prediction vector.
- $P_{station}(p : \mathbb{R}^p, s : \mathbb{R}^s | \theta_{ps}) \rightarrow \mathbb{R}^p$. Given a prediction vector and a station, update the prediction vector to include this station.
- $P_{predict}(p : \mathbb{R}^p | \theta_{pt}) \rightarrow \mathbb{R}$. Given a prediction vector, output the total travel time (DFM).

If a train is at Ruggles, and we want to predict its time to get to both Mass Ave and Back Bay, we would use the following pseudocode (visualized in Figure 2):

```
input: train vector t0 that just departed from ruggles,
       station vectors s_ruggles, s_mass_ave, and s_back_bay,
       neural network weights theta_p0, theta_ps, and theta_pt
pred0 = P_init(t0, s_ruggles, theta_p0)
pred1 = P_station(pred0, s_mass_ave, theta_ps)

P_predict(pred0, theta_pt) // (Estimated time from ruggles to mass ave
                               - average time from ruggles to mass ave)
P_predict(pred1, theta_pt) // (Estimated time from ruggles to back bay
                               - average time from ruggles to back bay)
```

3.3.3 Loss Function and Training

There are 20 stations. Thus, for a given train, there are $\frac{20(20-1)}{2} = 190$ possible predictions. When a train leaves Forest Hills, we can predict the delay time to get to Green Street, delay to Stony Brook, etc. Then, when the train leaves Green Street, we can predict the delay to Stony Brook, delay to Roxbury Crossing, and so on.

Let $y_{t,s,d}$ represent the delay time for train t between station s and station d, and $\hat{y}_{t,s,d}$ represent the predicted delay, calculated through the process described above. If $\mathcal{T}$ represents all trains in a batch, and $\mathcal{S}$ represents all pairs of stations (where the second station is north of the first station), the loss function that we are trying to minimize is:

$$L = \sum_{t \in \mathcal{T}} \sum_{(s,d) \in \mathcal{S}} (\hat{y}_{t,s,d} - y_{t,s,d})^2$$

All parameters $\theta_T, \theta_{S,s}, \theta_{depart}, \theta_{entries\_and\_weather}, \theta_{p0}, \theta_{ps}$, and θ_{pt} are jointly optimized via gradient descent. We use Pytorch [11] to automatically track the data dependencies and calculate the gradients.

3.4 Implementation Details

3.4.1 SVS Networks

Train, station, and prediction vectors each have 10 dimensions.

The neural network that processes the periodic message containing information about gate entries and weather conditions first concatenates the station vector with a representation of the message. This vector is then fed through 3 linear layers with 64 hidden neurons. The first two use leaky ReLU activations, while the last uses a sigmoid activation. This combined vector is used as the input to a GRU cell, with the station vector as the hidden state. The output hidden state becomes the new station vector.

When a train departs from a station, the train vector, station vector, and the message containing information about the train departure are concatenated together. This message is fed through a network that has a nearly identical architecture to the linear layers mentioned during the entries and weather message processing (only differing in the number of dimensions for the first layer's input). Then, this summarized vector is used as the input to two independent GRU cells. One of these cells uses the station vector as its hidden state; the other uses the train vector as the hidden state. The new hidden states become the resulting train and station vectors, respectively.

3.4.2 PVS Networks

The network that transforms a train and station vector to a prediction vector first concatenates the two vectors. Then, 3 linear layers are used. Similarly to SVS, the first two have leaky ReLU activations, and the last has a sigmoid activation. When a station is added, the station vector is directly used as the input to a GRU cell with the prediction vector as the hidden state. Lastly, to get a concrete time prediction, another 3-layer linear network is used, where the last layer has only a single output and no activation function. As seen in Figure 3, this architecture resembles a RNN that processes each station in order as a "character" in a "sentence".

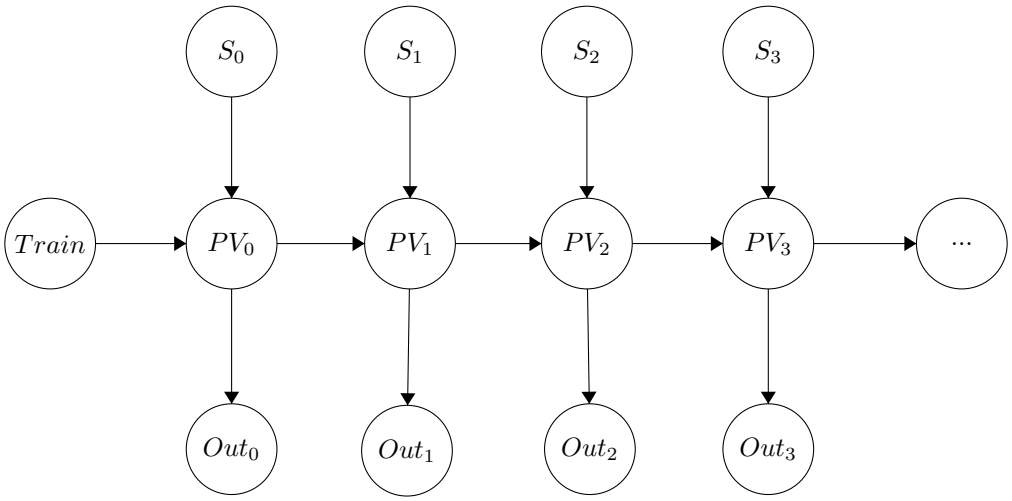

Figure 3: The architecture of PVS mirrors that of a traditional RNN. Each successive station that the prediction vector "visits" can be thought of as an input character; the predicted delays can be viewed as an output character.

3.4.3 Batching

The MPNN architecture is challenging to implement efficiently due to the temporal dependencies among various components. We batch the data in two ways to speed up training. First, we calculate batches of non-interfering trains offline. For example, if a train departs from Ruggles, and a different train departs from North Station, we can calculate the new train and station vectors in parallel. Second, once we obtain train and station vectors at each moment, we do not calculate the predicted delays right away. Instead, we store these vectors, with their gradients intact, until we obtain a large number of train and station vectors. Then, once we have sufficiently many of these, we run 19 iterations of PVS (using masks to prevent trains with fewer than 19 remaining stations from affecting the final loss computation). This is able to run with somewhat reasonable efficiency on a standard GPU.

3.4.4 Data Processing

We use the Python library, Pandas [12], to pre-process the data prior to training. For example, since the MBTA train data only includes arrival and departure events in relative isolation, we must correlate arrivals and departures at the same station to find the dwell times, events at different stations to calculate the travel times, and events at the same station between successive trains to determine the headways. Similarly, we need to find the averages for several of the features per station, as many of our model's inputs are normalized using this average.

3.5 Baseline Models

While the MPNN runs a "simulation" of all the trains and stations during an entire day to obtain the predicted delays, the baseline models require input-output pairs that are completely independent of each other.

Since we cannot include the entire history of a train in the feature space for a single input sample, we must somewhat recharacterize the training task. Each pair of stations along a train's journey becomes a data point, with discrete features for start and end station in the pair and the weather at the time. We also include continuous features (all DFM) for the train timestamp, dwell time, headway, and the number of station entries in the 30 minutes prior to the train's departure. The task is then to predict the delay (travel time, DFM) between these two stations.

We adapt a set of pre-implemented baseline models [13] to our exact feature set and task. These models had already been used to predict delays on the Green and Red lines, but not on the Orange line. The OLS model and the XGBoost tree regression model [14] performed the best on these lines, so we focus on them in our comparison. We also included a Generalized Linear Model. However, in the evaluation of the pre-implemented models on the Green and Red lines, GLM performed poorly. It is possible that the pre-implemented model library does not adequately scale the output of the GLM; we decided to leave it as-is, since extensive evaluation of the baseline models is outside the scope of our project.

As a reference baseline, we also calculate the error for a predictor which always predicts the average delay between the start and end stations. Since the delay is *defined* as the actual travel time minus the average delay between two stations, this corresponds to a predictor which always outputs zero.

3.6 Training

We partitioned 70% of the days in the dataset for training, used 10% as validation, and reserved 20% for our test set. We trained three instances of our network; two using only 14 days from the training set, and one using the full 70%. One version of the 14-day network was trained such that gradient updates considered the entire set; another version batched per day of training data. Due to resource constraints, we were only able to train the full version using daily batches. All three networks were trained for approximately 38 hours; the 14-day versions were trained on a NVIDIA 1050M, and the full version was trained on a NVIDIA 3090.

4 Results

A summary of our results is shown in Figure 4. The full-dataset model performs about at par with most of the baselines on the mean absolute error (MAE) metric, but better than all on the root mean square error (RMSE). This indicates that while the average prediction that our model generates has about the same deviation from the true value on average, our model is less likely to be extremely far off in either direction.

Model	MAE (s)	RMSE (s)
MPNN (full training dataset)	77.6	141.3
MPNN (14 days, full batch)	80.8	142.6
MPNN (14 days, daily batches)	81.1	147.4
Naive Average	78.2	171.9
OLS	77.7	160.5
GLM	89.4	3169.7
XGBoost (tree regression)	71.3	147.1

Figure 4: Comparison of the mean square errors for different models. When trained on the full dataset, MPNN performs about the same as the other baseline models when considering mean absolute error, only behind XGBoost. However, MPNN performs significantly better on the mean square error metric, indicating that it handles outliers better.

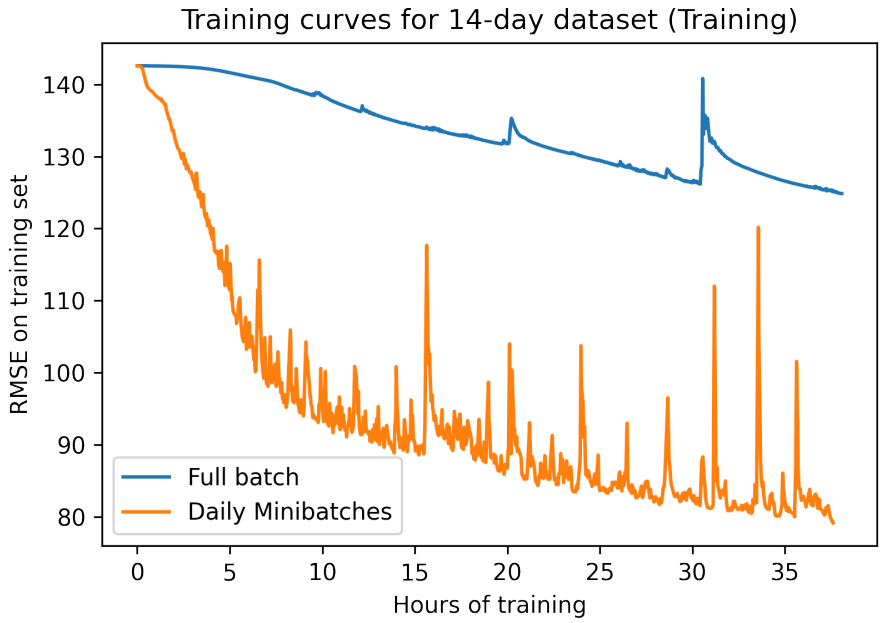

Figure 5: We ran the model with a 14-day training set with both full-batch gradient updates and daily minibatch gradient descent. The daily minibatch approach showed greater instability during the training process, even when evaluated on the training set.

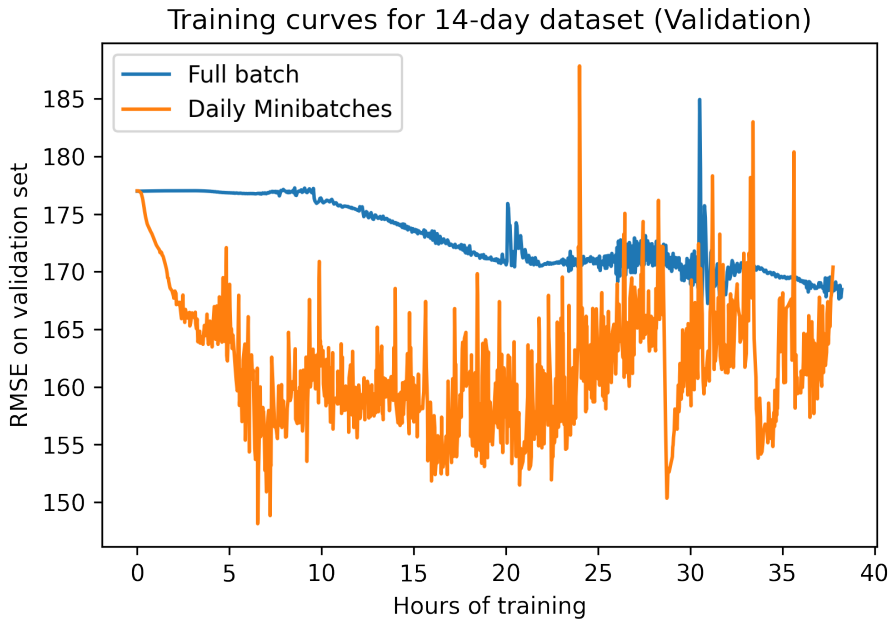

Figure 6: The daily minibatch approach also did better on the validation set initially, but its performance decreased as training continued, while the full batch model continued to improve for a longer period of time.

As indicated by Figures 5 and 6, the version that performed gradient descent on daily minibatches improved much quicker than the version that used the entire batch. However, its training became unstable as time went on. The full-batch version performed better on the test set—the better performance of the daily minibatches on the validation set may have also been due to random fluctuation—but it was infeasible to scale the full-batch approach up to the full dataset due to technical limitations. Nevertheless, the version trained on the full dataset performed better than either of the 14-day versions, beating all the baselines in RMSE and only losing to XGBoost for MAE.

5 Conclusion and Future Work

While the MPNN architecture's performance was not significantly better in terms of absolute error, our model beat all of the baselines' performance for the squared errors. One possible explanation could be that during non-exceptional operating conditions, most train delays may be more or less random. The MPNN may be better able to predict the outliers, resulting in the lower RMSE. However, if it is "overeager" to predict an outlier in the data that does not exist, this may lead to a higher MAE.

The MSE on the training set was significantly lower than the validation error, indicating that the model may be overfitting to the training set. This may be the result of model with too many parameters; however, this is unlikely, due to the small number of neurons in the hidden layers of our model. It is possible that the model was trained for too many iterations on the same dataset. Future ways to prevent overfitting may include the

use of a regularization term, early stopping when the change in validation error falls below an empiracally determined delta, or stopping if the magnitude of the gradient decreases past a certain amount. More comprehensive cross-validation techniques such as k-fold cross-validation may also improve the performance of future models.

One limitation of the study was the restricted time frame. Due to the time required to train the model, we were unable to perform an exhaustive hyperparameter search. Further hyperparameter tuning, including the sizes of the train, station, and prediction vectors, the architectures of the neural networks used to process messages, and alternative loss functions could all be subjects for future study. Additionally, we only used one year of data, while the MBTA publishes data going back to 2016. Given additional time, this data could also be included for training and testing.

Lastly, the Orange Line does not exist in isolation. Besides the three other subway lines that connect with the train, Boston has a commuter rail network, an extensive bus system, and a public bikeshare system. The riders that use these systems may connect to the Orange Line, potentially influencing delay times. Every train, train station, bus, bus stop, and BlueBike station could potentially be modeled as nodes in a MPNN, with messages passing between stations near each other as events occur to potentially model complex interactions. A bus unloading a passenger or a rider docking a bike leads to an additional person waiting for a train; while these individual events may not contribute much to the final delay, the cumulative effect may be meaningful. Data such as traffic on road networks, or major events occurring in Boston can be integrated into the system, and a MPNN can be used to predict more than just train delays; one only needs to be able to model everything in the system as nodes that can send and receive messages.

6 Individual Tasks

Samuel adapted the baseline models for the data and ran the training and testing of the baselines on his machine as well as the training of the full MPNN. Jonathan did data preprocessing on the data. And Daniel Designed both portions of the MPNN and trained the 14 day model for them. Each member contributed to the documentation of the study.

References

[1] P. Wang and Q.-p. Zhang, "Train delay analysis and prediction based on big data fusion," *Transportation Safety and Environment*, vol. 1, no. 1, pp. 79–88, Jul. 2019, ISSN: 2631-4428. DOI: 10.1093/tse/tdy001. [Online]. Available: https://doi.org/10.1093/tse/tdy001 (visited on 03/10/2021).

[2] R. Nilsson and K. Henning, *Predictions of train delays using machine learning*, eng. 2018. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-230224 (visited on 03/10/2021).

[3] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, Jan. 2009, Conference Name: IEEE Transactions on Neural Networks, ISSN: 1941-0093. DOI: 10.1109/TNN.2008.2005605.

[4] K. Rusek and P. Chołda, "Message-Passing Neural Networks Learn Little's Law," *IEEE Communications Letters*, vol. 23, no. 2, pp. 274–277, Feb. 2019, ISSN: 1558-2558. DOI: 10.1109/LCOMM.2018.2886259.

[5] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," en, *arXiv:1406.1078 [cs, stat]*, Sep. 2014, arXiv: 1406.1078. [Online]. Available: http://arxiv.org/abs/1406.1078 (visited on 03/12/2021).

[6] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated Graph Sequence Neural Networks," Sep. 2017, arXiv: 1511.05493. [Online]. Available: http://arxiv.org/abs/1511.05493 (visited on 03/12/2021).

[7] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu, *Interaction networks for learning about objects, relations and physics*, 2016. arXiv: 1612.00222 [cs.AI].

[8] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural Message Passing for Quantum Chemistry," en, *arXiv:1704.01212 [cs]*, Jun. 2017, arXiv: 1704.01212. [Online]. Available: http://arxiv.org/abs/1704.01212 (visited on 03/10/2021).

[9] *MBTA Blue Book Open Data Portal*, en. [Online]. Available: https://mbta-massdot.opendata.arcgis.com/datasets/ (visited on 03/12/2021).

[10] *Meteostat Developers*. [Online]. Available: https://dev.meteostat.net/ (visited on 03/12/2021).

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[12] J. Reback, W. McKinney, jbrockmendel, J. V. den Bossche, T. Augspurger, P. Cloud, gfyoung, Sinhrks, A. Klein, M. Roeschke, S. Hawkins, J. Tratner, C. She, W. Ayd, T. Petersen, M. Garcia, J. Schendel, A. Hayden, MomIsBestFriend, V. Jancauskas, P. Battiston, S. Seabold, chris b1, h vetinari, S. Hoyer, W. Overmeire, alimcmaster1, K. Dong, C. Whelan, and M. Mehyar, *Pandas-dev/pandas: Pandas 1.0.3*, version v1.0.3, Mar. 2020. DOI: 10.5281/zenodo.3715232. [Online]. Available: https://doi.org/10.5281/zenodo.3715232.

[13] A. Radhakrishnan, *Subway time prediction*, 2021. [Online]. Available: https://github.com/amalrkrishna/subway_time_prediction.

[14] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *CoRR*, vol. abs/1603.02754, 2016. arXiv: 1603.02754. [Online]. Available: http://arxiv.org/abs/1603.02754.