Scalable Spatio-Temporal Arrival Time Estimation for Public Transit

by

Karan Dhingra

Under the supervision of

Dr. Pravesh Biyani, IIIT Delhi

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

NEW DELHI– 110020

September 2021

Scalable Spatio-Temporal Arrival Time Estimation for Public Transit

by

Karan Dhingra

Submitted

in partial fulfillment of the requirements for the degree of

Master of Technology in

Computer Science Engineering (CSE-AI)

to

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

September 2021

Certificate

This is to certify that the thesis titled **Scalable Spatio-Temporal Arrival Time Estimation for Public Transit** being submitted by **Karan Dhingra** to the Indraprastha Institute of Information Technology Delhi, for the award of the Master of Technology in Computer Science & Engineering (CSE-AI), is an original research work carried out by her under our supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

September, 2021

Dr. Pravesh Biyani

Indraprastha Institute of Information Technology Delhi

New Delhi 110020

Acknowledgement

After three years at IIITD, I would like to pay my hearty gratitude to several individuals who contributed in many ways. Firstly I give special thanks to Dr. Pravesh Biyani, who gave me freedom, guidance, and faith to pursue my ideas. I am incredibly fortunate that I have had the support even before I started my M.Tech. Next, I would like to thank Charul for her contribution, constant support, and making sense of my ideas. I would also like to thank Kshitij for helping me with the ETA data.

In the end, I would like to express my gratitude towards my family and friends for their faith, co-operation, and guidance, which have been a constant source of motivation. The last appreciation would be towards my nephews and niece; their presence and naivety helped me stay calm and happy throughout this journey.

Abstract

The problem of the ETA prediction of public transit has an essential role in improving the rider's experience. While, it is challenging to ensure the timeliness of bus, especially during the rush hours. This thesis provides a heads-up on estimated arrival time for better planning using the open-transit data.

The first step of providing a real-time scalable ETA is to design an algorithm that can preprocess the raw GTFS data of a day into a tensor. The representation aims to decouple the information about the bus, thereby enabling scalability across routes and reducing variance.

The second step is to design a Spatio-temporal model (SSTG) for scalable and robust ETA prediction. In the proposed SSTG framework, we will provide answers to the following open problems. Firstly how can we exploit the spatio-temporal correlation in the ETA data? Secondly, how to scale the spatial-temporal ETA prediction framework on a large network effectively? Thirdly, How to handle sparsity in the data? Fourthly, the prediction of the ETA for the cold start stops is an unexplored problem. i.e., stops that are absent from the training dataset, how can we predict ETA for a cold start-stop? Moreover, a user would prefer waiting a bit longer than missing the bus because of underestimation. Therefore, for better customer satisfaction, we need to reduce the underestimation.

The proposed framework captures the Spatiotemporal structure in the ETA data using recurrent neural networks modified with a graph convolutional. The input to the network can be sub-sampled, thereby ensuring scalable learning and further providing a solution to the cold start stops ETA prediction. The first layer of the encoder integrates GRU-D for the missing data imputation. Moreover, we use a MSLE-Weighted loss function to overestimate the ETA and fine-tune the penalty on overall performance compared to the regression loss(MSE) function. We finally conclude that the SSTG model is computationally efficient and outperforms the state-of-the-art methods on ETA and traffic datasets.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traffic Congestion impacts our society adversely on the economy, environment, and mental health. On an average, a Delhi resident wastes approximately seven days [1] due to traffic congestion in a year. Another study [2] highlighted that the traffic congestion amounted to approximately ₹54000 crores in Delhi in 2013. One way to resolve this problem is by strengthening the public transportation services, providing end-to-end mobility and ease of use in accessing the public services.

However, the share of non-captive riders, who are formally employed and earn more than ₹50000 per month, is less in the Delhi bus system as compared to Mumbai [3], because of punctuality and travel time. Another study [4] highlighted the reason behind the transition of commuters from bus to the Metro. While 83% of the commuters moved to reduce their travel time, approximately 37.4% bus riders believe that buses are less punctual when compared with the Metro.

The recent initiative of open-data [5] has provided real-time access to the bus movement in the General Transit Feed Specification (GTFS) format, which enables the development of services to facilitate a better experience for an average rider and thereby improve the usability of the buses. One of the primary reasons behind the poor share of non-captive riders in Delhi buses is the punctuality of the buses; ensuring timeliness is challenging, especially during rush hours in Delhi traffic.

It is possible to provide riders with a heads-up on the amount of time taken by the bus to reach their location. Providing estimation of arrival time can aid a user in deciding whether to opt for public transport or use private depending on convenience. According to a study[6] on the bus service in Chicago, an increment in ridership, approximately 126 rides per day was observed after providing real-time information of the buses.

Estimating the time of arrival is a challenging problem, especially in Delhi. Key features like $speed$, $time\ taken$ exhibit complex Spatio-temporal relationship. E.g.,

Two vehicles might take drastically different times to travel the same distance in two neighboring streets, while it might take a similar time to travel farther located expressways. It is very common that traffic during rush hours is smooth in one direction but congested in the opposite. Thus, it is essential to model the temporal relationship based on the vehicle's path and the spatial relationship with different locations.

It is preferable to provide over-estimated results. Let's say the estimated arrival time for the bus is 10:30, but it arrived at 10:32 this incurs discomfort as the rider would have to wait for 2minutes but what if it came at 10:28 and left even before 10:30. The penalty incurred by a rider with over-estimation is minor as compared to under-estimation.

1.1 Existing Work

There are various methods available to model the temporal sequences. ARIMA (Autoregressive Integrated Moving Average) and SARIMA (Seasonal ARIMA) have been used to model univariate traffic data [7] and predict short-term traffic at a fixed location. ARIMA and SARIMA rely on Wold Decomposition, i.e., any stationary time series can be written as the sum of two time series, 1) deterministic and 2) stochastic. Thus, if we can transform a series into a stationary, it would become simpler to predict the output as the stationary sequence becomes independent of time. If a time series is not stationary, it can be converted by computing the difference of contiguous elements which are k distance apart. SARIMA is a special case of ARIMA, in which the difference is modeled based on seasonal information of the time series.

Kalman filter is an algorithm that aims to estimates a function which fits the noisy input. Kalman filters are used to smooth the jerk in any real-world process, like GPS data fetching e.t.c. It models the internal state space using a Markov chain to learn the state-transition matrix given the observations. Kalman Filters has been used to model the spatiotemporal data [8] and ETA prediction [9], [10].

A hybrid system [11] has been proposed in which SVM is used to model the ETA

on key points based on historical metadata about the trip such as start-time, weather, route condition, travel time for a road segment to predict the ETA for key points between the trip. Kalman filters are used for interpolation between the key points in order to make adjustment if required based on the real-time input.

Deep Learning techniques are also used for future prediction tasks. In seq2seq [12], the authors proposed a deep neural network that can estimate the output for the next k steps using the input features of k_1 sequence length. It uses LSTMCell for temporal modeling and a full connected layer for spatial modeling.

In [13], the authors proposed FC-LSTM, which shares the weight between vertices, thereby reducing the overall footprint of the network, but it implies that the vertices are independent of each other. ConvLSTM [14] improves upon the FC-LSTM by using convolutional layers for spatial modeling.

The ConvLSTM uses spatial convolutional operator and works well for ordered data, which is not the gurranteed with graphs. To model the Spatio-temporal relationship, the spatial convolutional operator is replaced with spectral convolutional operator [15], [16], [17]. DCRNN improves upon the T-GCN by modifying GRUCell to compute graph convolution instead of matrix multiplication.

Generative models [18] have also been used for estimation of the time, a generative model aims at modeling the distribution of the input data, i.e. spatio-temporal. It works in autoregressive configuration i.e. the output at k^{th} time interval is not just conditioned on the input feature but also on the output of $k_{-1}....k_{-p}$ steps where p is the overall receptive field of the network.

In [19], [20], the distance between the vertices is used to represent the weighted adjacency, which works well. But it would award less score to far-away highways that are more likely to operate similarly or awards a very high score to the opposite direction of the road. Graph Attention Networks [21] aims at learning the adjacency on the fly by modeling the relationship of the vertices with themselves. Correlation between the vertices can also be used to build the adjacency matrix using relationship between the vertices. The techniques discussed in this thesis extent these works on Simplified and Scalable Spatio Temporal Modeling.

1.2 Problem Statement

The time taken by a bus to travel between the two stops can be used to provide a head's up on approximate travel time between stops at a given time. In a given transit system, we have multiple vehicles running between stops following a pre-assigned route. The arrival time for a given set of routes needs to be processed in a single operation instead of sequential operations for each route. As not every vehicle is expected to cover the edge at every time interval. In some cases, the data get lost because of poor connection, leading to missing entries. We need to consider the effect model has when it over-estimates the arrival time vs. under-estimation.

The input data is a list of entries, such that each entry consists of trip id, route id, GPS coordinates, speed, and the time stamp. We have over 20 lakh entries for a given day, such that each trip has, on average more than 160 entries sampled at a frequency of 10sec, where some entries are lost due to signal error. The thesis solves the following two problems:

- Modeling the raw GPS sequences into a tensor, $T \in \mathbb{R}^{N \times T \times V}$, where N is the number of days, T denotes the sampling frequency per day, and V represents the edge, i.e., transition from one stop to another.

- Estimation the arrival time in advance using a compact yet scalable model that learns the Spatio-temporal relationship and handles the missing data. It provides a mechanism to finetune the overestimation rate.

1.3 Contribution

The objective of this thesis is to model the Spatio-temporal relationship for arrival time Estimation of buses using the open data of Delhi. We will first refine the raw data and define the structure as it is challenging to model raw sequences of GTFS data considering we have an average of 26 trips on 1085 routes every day. We will then decouple the data from the buses, as running on individual buses independently is counterproductive as the experience of a nearby bus may aid in a better relationship

than the previous time-step. There are multiple buses traveling between single hop at a given time, thereby reducing the overall variance of input data.

After that, we will define neural network based framework and training techniques that can be used for modeling, followed by the proposed framework, which scales up efficiently on large databases,

The main contribution of the thesis can be summarized as follows.

1. We propose a scalable Spatio-temporal ETA prediction framework that predicts the future ETA of a large network in the presence of missing data.

2. The proposed framework models the spatiotemporal characteristic in the ETA data using a Graph convolution-based framework.

3. To efficiently train the ETA framework to a large network, we employ sampling during training.

4. The proposed framework process the missing data to improve the prediction performance in the presence of high missing entries.

5. To improve the overestimation, we have shifted from the MSE loss function to a modified MSLE loss function.

6. We conduct comprehensive experiments on real-world ETA and traffic speed datasets that show the efficacy of the proposed method over the other state-of-the-art methods.

1.4 Structure

The thesis is organized as follows:

- Chapter 2 presents the preprocessing techniques require to model the raw GTFS data into Spatio-temporal tensor.
- Chapter 3 overviews the concepts and framework used in the thesis.
- Chapter 4 overviews the algorithms used for comparison and our proposed framework.

- Chapter 5 presents the Experimental Results and Analysis.
- Conclusion and Future work are provided in Chapter 6.

Chapter 2

Modeling Raw Data to Tensor

2.1 Introduction

Open transit data enables an exciting opportunity to aid passengers in improving their travel experience. The open transit data follows GTFS format [22], General transit feed specification, which consists of two components 1) static and 2) feed. The static data consists of routes information, stops details, and other information which does not change over time. In contrast, the feed data consists of real-time information consisting of bus metadata, timestamp, and spatial coordinates.

Each entry in the data consists of coordinates of a bus at a given time following a predefined route. While it is a very efficient way to store data for real-time exchange, but requires iteration through the complete dataset to get details of a specific trip, which is computationally expensive. Similarly, given a trip, multiple entries along the route are available, but it is required to filter out entries that are not nearby from a stop to estimate the arrival time. Thus, modeling not just reduces the system's complexity but also removes any irregularities from the data.

2.1.1 Background

Each entry in the dataset consists of GPS coordinates at a given time. As the points are represented in the spherical coordinates system, the calculations need to incorporate the spherical nature of the earth.

1. **Havestine Distance between two points**: As the earth is spherical, the straight line distance needs to be calculated along the arc, while the Euclidean distance is a circle segment. $H_d(BC) = \alpha * R$, where α is the angle of the arc, and R is the radius. α can be computes as $sin(\alpha) = \frac{d}{2R}$, where d is the eucledian distance between the points.

2. **Mid Point of two points**: Earlier, we were calculating the distance BED (Fig.

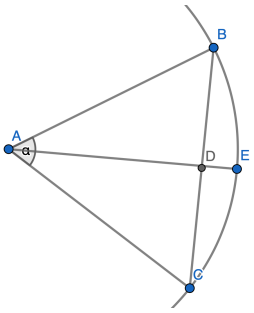


Figure 2.1: Havestine Formula, R is the radius, BC is the eucledian distance, and BEC is havestine distance

2.1), we also require coordinates of E point. We use the value of D as E by assuming that the earth is locally flat, where D is calculated as the midpoint in cartesian system.

2.1.2 Motivation

The aim is to design an algorithm that can preprocess the data of a given day into a matrix, $M_f \in \mathbb{R}^{K \times T}$ where T is the sampling frequency on a given day, and K is the number of locations edges. Here an edge represents the time taken by bus to go from a stop, s to the next connected stop. This representation decouples the information about the bus, enabling 1) improving the scalability of the system and 2) reducing the variance as the time taken by multiple buses are averaged.

2.2 Modeling

The process of modeling M_f has been broken down into four independent sub-processes to reduce the complexity of the overall system and improve the debugging capabilities.

2.2.1 Entries to Tree

In this subprocess, we restructure(Alg. 1) the input data into a tree format such that the first child of the root contains all the trips on a particular route, and its child stores a list consisting of individual trip information. The tree structure (Fig. **??**) reduces the complexity of trip querying from $O(N_R * N_T * T_E)$ to $O(N_R + N_T)$,

where N_R denotes the total number of routes, N_T denotes trips on a given route, and T_E denotes the number of entries in a trip.

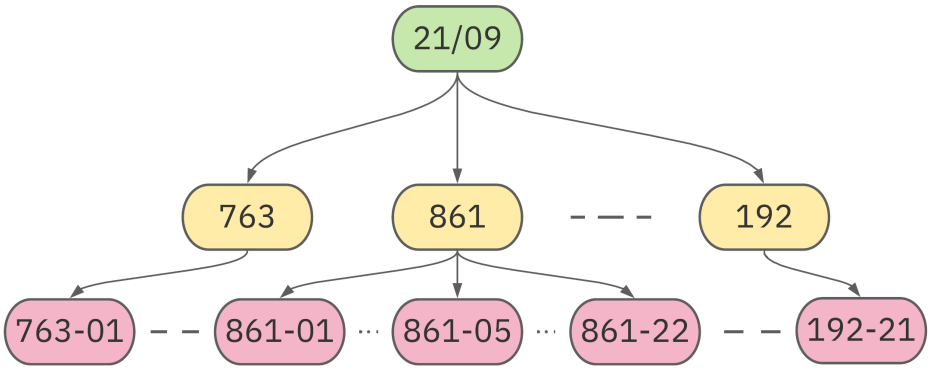


Figure 2.2: Entries Tree Structure - Root node is date, with its children describing data from the individual routes and the leaf nodes consisting of individual trip data.

While adding to a given trip, we assert that there are no duplicate entries. It reduces the overall entries by 20%, which is huge considering more than a million entries each day.

Algorithm 1: Restructuring the entries into tree

Result: T, tree as described in 2.2.1
Initialization : $T = \{\}, I_d = Input\ Data$;
for $Entry \in I_d$ **do**
  $route, trip, x = Entry$
  if $route \notin tree$ **then**
    $tree[route] = \{\}$
  else
    if $trip \notin tree[route]$ **then**
      $tree[route][trip] = x$
    else
      $t_{-1} = tree[route][trip][-1].time$
      if $x.time! = t_{-1}$ **then**
        $tree[route][trip].add(x)$
      end
    end
  end
end

2.2.2 Assigning entries to stops

A trip consists of entries present all over the route path, and this sub-process filter out entries that are not in proximity to a given stop.

The brute force approach assigns an entry to a stop with the minimum distance. The alignments are accurate, but the complexity is $O(T_E * T_N)$, where T_N is the number of stops in a trip. The computation would scale up as this process needs to be repeated $O(N_R, N_T)$ times. Also, it might introduce alignment with extra stops which the bus never covers.

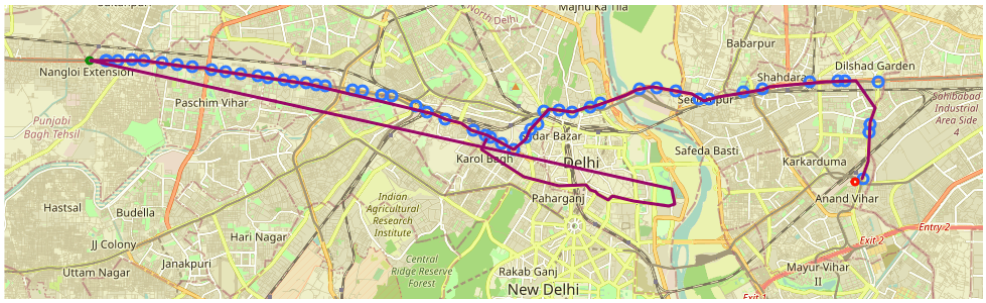


Figure 2.3: Bus starts from Anand Vihar stop(red), and should end at the Nangloi Extension(green).

Given an ordered list of stops, $[S_1, S_2, ...S_N]$ on a route, for an entry, e_i we search for the stop, S_i,

$$S_i : \big(H_d(S_i, e_j) < H_d(S_{i+1}, e_j)\big) \wedge \big(H_d(S_i, e_j) < \epsilon\big) \tag{2.1}$$

where ϵ is the threshold. The alignment assigns an entry to a particular stop if it is closer than the next stop and the distance between the stop and entry is less than a threshold, ϵ. It is done, because not all trips starts from their source' stop (Fig. 2.4).

The thresholding is helpful in scenarios where a stop's GPS location is wrong. If S_{i+1} is very far, then all of the entries would be assigned to the stop S_i only. We used the value of $250m$ as $epsilon$ and observed an increment of 5% in stop alignment of the data.

The value of $250m$ was taken based on multiple factors, like the percentage of stops aligned vs average speed required to cover a distance in 10 seconds, which is the frequency of our data. On average, more than 80% of the entries are under

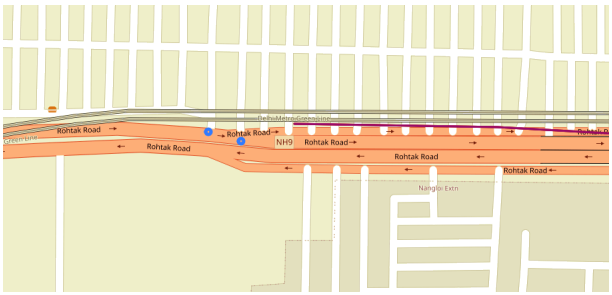
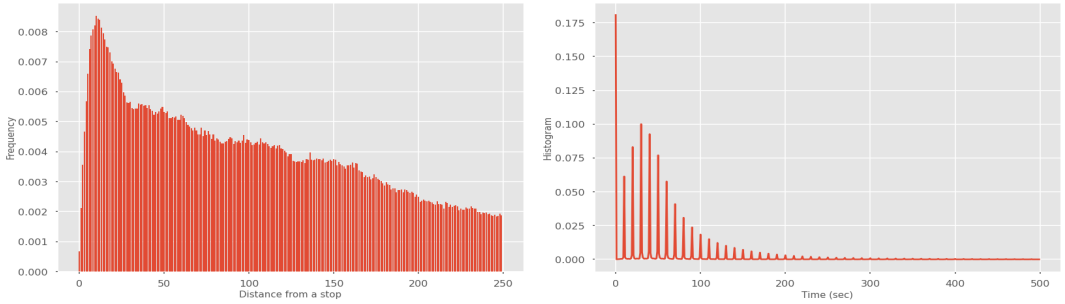


Figure 2.4: Stops : Blue circle, and Entries: purple line; there is no entry before the second stop.

50m distance from the stop, but 6% of the entries above $150m$ are the only readings aligned for the given stop, thereby reducing the missing alignment data from 46% to 42%.



(a) Distance from a stop

(b) Time taken at a stop

Figure 2.5: Histograms

2.2.3 Time interpolation

Now, we have multiple entries in proximity with a given stop; still, we need to assign a specific time to a given stop as it could take anywhere between 10sec to 4 minutes to travel $250m$. We take few assumptions to assign the time,

- We set the ideal displacement of the bus from a stop's as $-32m$, i.e., we assign the time to any stop as the time taken by a bus to reach 32m before the stop's location. Here, 32m is equivalent to the length of two buses, and to promote over-estimation.

- We only consider entries that are fetched within 2.5 minutes of bus reaching the closest point from both directions. The threshold automatically increases

to 5 minutes when there are fewer entries available. Here, the value of 2.5 minutes is based on analysis of bus waiting time(Fig. 2.5b) at a given stop. This threshold removes any outliers on the terminal stops.

Given the aligned entries, we need to estimate the time at which the bus is $32m$ before the stop. We first need to mark entries based on their location relative to the stop, i.e. whether the entry is logged before or after the stop. Assigning direction based on the distance from neighbouring stops lead to in-correct assignment due to missing entries in one or another direction as shown in Fig. 2.6.

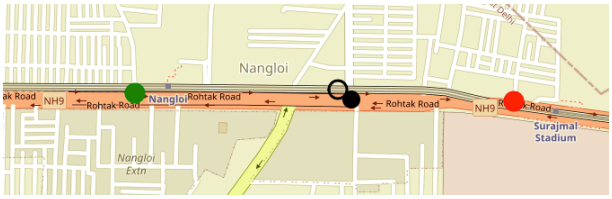


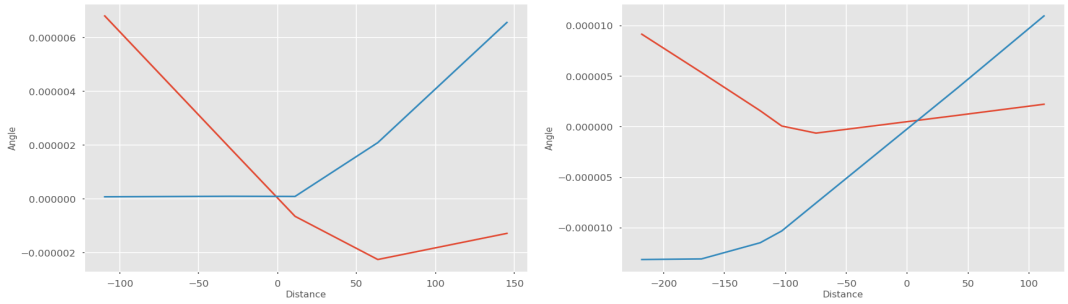
Figure 2.6: Red: Previous, Black: Assigned, Green: Next Stop

Even though the entry is after the stop, it would be assigned as before because it is closer to the previous stop than the next stop. Also, distance for direction could introduce errors in stops that are not in the straight line. Thus, we compute the angle between entry and neighboring stops to assign direction.

Algorithm 2: Assigning Direction to an entry

Result: D, boolean; 0 refers to before, and 1 as after the stop
Initialization: s: stop id; d_e : direction of an entry; d_n : direction of stop;
if $len(d_n)$ is 1 **then**
  #Edge condition, start and end stop;
  #Assigns opposite direction;
  if s *is 0* **then**
    $d_n = (-d_n, d_n)$;
  else
    $d_n = (d_n, -d_n)$;
  end
end
$D = d_n[1].dot(d_e) > d_n[0].dot(d_e)$

Once the direction is assigned, we use the longest increasing subsequence(LIS)



(a) Example 1 (b) Example 2

Figure 2.7: Direction Alignment; red - towards stop; blue - away from the stop

algorithm(Eq. 2.2) to ensure monotonicity in the displacement around the stop.

$$L(i) = \begin{cases} 1 + max(L(j)) & (x[j] < x[i]) \wedge (0 < j < i) \\ 1 & otherwise \end{cases} \tag{2.2}$$

We also tried Kalman filters to smoothen the coordinates and avoid removing any entry, but 1) Kalman filters are computationally expensive, and 2) these entries have less than $15m$ of distance, highlighting the stoppage of bus and should have no effect on the time estimation. The LIS algorithm operates with $O(T_E)$, and we observed a 5% reduction in the entries, but it was less than 2.5% when the entries count were less than five and increased up to 40% when the count reached 30 as shown in Fig. 2.8

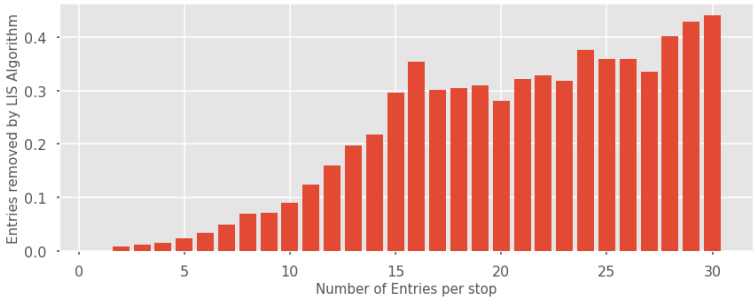


Figure 2.8: Entries removed using LIS Algorithm

The monotonicity was introduced, 1) to cross-check if the alignment of direction is correct, 2) remove entries when the driver forgets to switch off the recording device or waits for longer duration of time. The monotonicity also enables easier

interpolation as distance can now be used as an independent variable.

We used interpolation to estimate the time at the $-32m$ mark, assuming that the bus travels with constant speed as cubic spline and other polynomial interpolater did not perform well. The linear interpolation is directly used if there are entries before and after the mark. We did not use linear interpolation for extrapolation as the results were not robust. For extrapolation, we break it down into three cases. If multiple entries are present, we compute the speed between consecutive entries and use the closest to the required speed. If there is only one entry present, then we use the corrected instantaneous speed (scaled up by the factor of 3.6) if the speed is not zero, or else we assume the speed as $2.7m/s$.

The quality of time estimation can be computed based on how much time a bus takes to reach the stop B from A. We labelled the entries as mis-alignment if the time taken is less than 30s or more than 30min.

Time Mis-Alignment			
Interpolator	Cubic Spline	Linear	OUR
% Error	28%	2.26%	2.24%

Algorithm 3: Time Interpolation Algorithm for a stop

Result: t, time taken to reach $32m$ mark.

Initialization : $I_d =$ list of entries, and t, s = stop ;

diff returns difference of consecutive elements

if *(diff(I_d) < 0)* **then**

$t = unique(I_d.t)$;

$I_d = I_d[t]$;

end

havestine distance between a point and stop.

$I_d.d = H_d(s, I_d)$;

$C_p = argmin(I_d.d)$;

Rescale the origin of time to the closest entry.

$t_{ref} = I_d.t[C_p]$;

$I_d.t = I_d.t - t_{ref}$;

$I_p = zeros(len(I_d))$;

2.5minutes thresholding

$I_p[-15 + C_p : 15 + C_p] = 1$;

5minutes thresholding

$I_p = I_p \wedge (I_d.t < 5 * 60) \wedge (I_d.t > -5 * 60)$;

$I_d = I_d[I_p]$;

direction assignment using Alg. 2.7

$I_D = direction(I_d)$;

if $(I_d[0] < -32 \wedge I_d[-1] > -32)$ **then**

$t = interpolate(I_d.x, t)(-32)$

else

if $len(I_d) > 1$ **then**

disprequired = $\begin{cases} I_D.x[0] + 32 & if I_D.x[0] > -32 \\ I_D.x[-1] + 32 & otherwise \end{cases}$;

$displacement = diff(I_D.x)$;

$p = argmin(|displacement - disprequired|)$;

$\mathrm{V}_c = \dfrac{displacement[p]}{I_d.t[p+1] - I_d.t[p]}$

else

$\mathrm{V}_c = \begin{cases} 3.6 * I_d[0].v & I_d[0].v > 0 \\ 2.7 & otherwise \end{cases}$

t = $\begin{cases} time[0] - drequired/V_c & I_D.x[0] > -32 \\ time[-1] + drequired/V_c & otherwise \end{cases}$

$t = t + t_{ref}$

2.2.4 Tree to Tensor

Given the stop time-alignment, this subprocess aims at generating tensor, T for a day of shape $\mathbb{R}^{K \times T}$, where K is the number of edges and T is the time. The tensor representation decouples the time estimation from trips, which increases over time and reduces the overall variance as different buses cover the same edge with approximate; we observed each non-zero entry is sampled 2.7 times.

We discretize the time into buckets of 10minutes; a reduction in the bucket size increases the sparsity. We observed that 32% of the time bus repeats its trip within the 10minutes. This number reduces to 16% if buckets of 5minutes are used, but the sparsity increases from 76.7% to 84%.

Algorithm 4: Tree to Tensor Algorithm

Result: Tensor, T

Initialization: $t, tree$; $h, dict()$ stores index for an edge alignment;

for *route in tree* **do**

  for *trip in tree[route]* **do**

    for *stop in range(*$N_s - 1$*)* **do**

      $data = tree[route][trip][stop]$;

      $data_{+1} = tree[route][trip][stop + 1]$;

      if *data is None OR* $data_{+1}$ *is None* **then**

        #mis-alignment

        continue;

      end

      $time = (data_{+1} - data)$;

      if $time < 0$ *OR* $time > 1800$ **then**

        continue;

      end

      $T[h[stop][stop + 1], time//600] += time/60$;

    end

  end

end

Chapter 3

Preliminaries

Understanding the behavior of a process requires analyzing the underlying relationship. While the relationships between the independent variables of a quadratic polynomial is simple, most real-world strategies follow a complex pattern. In recent times, neural networks have achieved the state of the art performances in modeling real-world processes. This chapter will focus on different neural network models and training strategies for scalable and efficient Spatio-temporal modeling.

3.1 Temporal Modeling

A recurrent Neural Network (or RNN) is a type of neural network which contains a self-loop (Fig. 3.5) on a temporal sequence. The self-loop enables the modeling of change in the input over a constant time interval.

We can use a multi-layer perceptron for temporal modeling but it would require $O(t * k)$ memory, where t is the sequence length and k feature size. Also, would requires more permutations of data because trainable parameters are independent of the time domain.

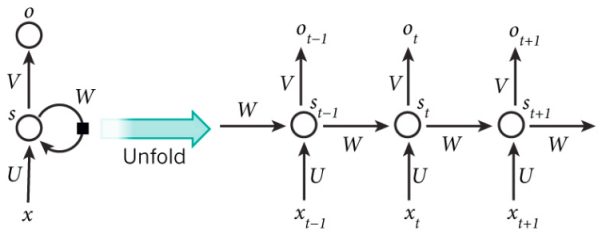


Figure 3.1: Recurrent Neural Network

As the trainable parameters are shared across time steps, it only requires $O(k)$ memory during inference. However during training, It unrolls the RNN to make t copies of the weights as part of gradient backpropagation through time, BPTT [23]. While the sequence length is one of the issues limiting its scalability, the other significant problems are Vanishing and Gradients Explosion.

3.1.1 Vanishing Gradient

Vanishing Gradient is a condition in which gradients are not able to propagate back to the initial layers. It occurs in neural networks which consists of many layers as,

$$\frac{\partial l}{\partial O_{k-1}} = \frac{\partial O_k}{\partial O_{k-1}} * \frac{\partial l}{\partial O_k}, \text{ acc to chain rule} \tag{3.1}$$

where O_{k-1}, O_k are the output of $k-1^{th}$, and k layers respectively.

Thus, gradient of k^{th} layer,

$$\frac{\partial l}{\partial O_k} = \frac{\partial O_{k+1}}{\partial O_k} * \frac{\partial O_{k+2}}{\partial O_{k+1}} * .... * \frac{\partial l}{\partial O_{-1}} \tag{3.2}$$

is directly proportional to the gradients of a layer's output wrt to input, if the gradients in-between layers become less than 1, then the effect on k^{th} layer would increase exponentially. It eventually saturates the learning curve, hence leading to the vanishing gradient. If the value of the gradients becomes more than 1, it might lead to the problem of exploding gradients, i.e., Gradients become too large (∞ or undefined).

It is more prominent in Recurrent Neural Networks with large sequence length, as the gradient for i^{th} time step is,

$$\frac{\partial l}{\partial O^i} = \frac{\partial O^{i+1}}{\partial O^i} * \frac{\partial O^{i+2}}{\partial O^{1+1}} .... * \frac{\partial l}{\partial O^{-1}} \tag{3.3}$$

thus,

$$\begin{aligned} & h^t = f(Wh^{t-1} + Ux_t + b), \\ \Rightarrow \quad & \frac{\partial h^t}{\partial h^{t-1}} = f'(h_{t-1}) * W \\ \therefore \quad & \frac{\partial l}{\partial h^t} = W^{T-t} \prod_{k=t}^{T} f'(h_k) \end{aligned} \tag{3.4}$$

$\propto W^k$, where k is the number of steps to the end of the sequence; it introduces instability, either vanishing gradients or gradient explosions.

3.1.2 LSTM and GRU

To mitigate the gradient stability issues, Hochreiter [24] proposed a gating mechanism inside the RNN Cell, known as Long Short Term Memory (LSTM) which controls the flow of information, thereby improving the stability of the network.

$$v = \sigma(Wh_{t-1} + Ux_t) \tag{3.5}$$

$$[i_t, f_t, o_t] = v \tag{3.6}$$

$$C_t^* = \tanh(W^g h_{t-1} + U^g x_t) \tag{3.7}$$

$$C_t = f_t \circ C_{t-1} + i_t \circ C_t^* \tag{3.8}$$

$$h_t = \tanh(C_{t-1}) \circ o_t \tag{3.9}$$

where U, W are the learnable weights, x_t is the input at time step t, and h_t, c_t are the hidden states.

The gradient of cell at time t,

$$\begin{aligned}
\frac{\partial C_t}{\partial C_{t-1}} &= \frac{\partial(f_t \circ C_{t-1} + i_t \circ C_t^*)}{\partial C_{t-1}} \\
&= \frac{\partial(f_t \circ C_{t-1})}{\partial C_{t-1}} + \frac{\partial(i_t \circ C_t^*)}{\partial C_{t-1}} \\
&= \frac{\partial(f_t \circ C_{t-1})}{\partial C_{t-1}} \quad + \epsilon \\
&= f_t \circ \frac{\partial C_{t-1}}{\partial C_{t-1}} + C_{t-1} \circ \frac{\partial f_t}{\partial C_{t-1}} \quad + \epsilon \\
&= f_t + C_{t-1} \circ \frac{\partial f_t}{\partial C_{t-1}} \quad + \epsilon \\
&= f_t \quad + \epsilon \\
\therefore \frac{\partial l}{\partial C_{t-1}} &= \boxed{\prod_{k=t}^{T} f_k + \epsilon}
\end{aligned} \tag{3.10}$$

where ϵ stores the gradient which is susceptible to vanishing gradient due to the presence of W^k (as highlighted in Eq. 3.4).

Thus the gradient back propagating through lstm cell is atleast $\prod_{k=t}^{T} f_k$ (from Eq. 3.10). The value of f_k is also controlled between $[0, 1]$ using sigmoid gate, and also

depends on the output value at time t, f_k.

A simplified version of LSTM was proposed by Kyunghyun [25] known as Gated Recurrent Network, GRU, which reduces the number of gates required, thereby reducing the computation and providing similar performance.

$$\begin{aligned} v &= \sigma(Wx_t + Uh_{t-1} + b) \\ [r_t, z_t] &= v \\ h_t^* &= tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h) \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ h_t^* \end{aligned} \tag{3.11}$$

Similarly to lstm, GRU ensures minimum gradient backprop of $\prod_{k=t}^{T}(1 - z_k)$.

LSTM and GRU reduce the effect of vanishing gradient but do not handle the issue of gradient explosion, and for that, gradients are clipped if required.

3.2 Spatial Modeling

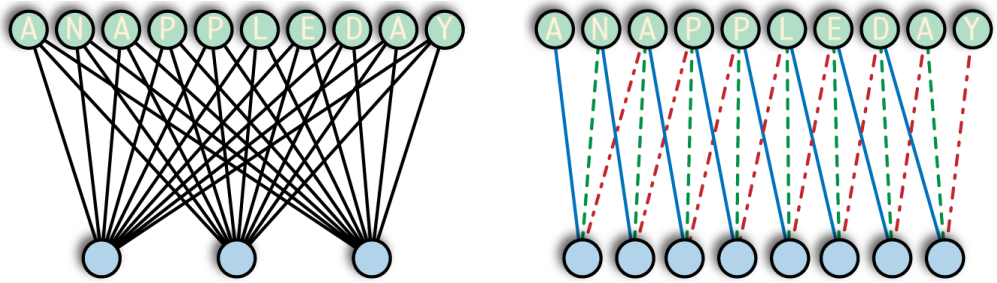
Convolutional Neural Network (CNN) is a type of neural network that aims to model a signal's spatial relationship. It consists of $f_c(x)$ which computes convolutional of input $X \in \mathbb{R}^{S_1 \times S_2 ... S_n \times N}$ with learnable weights, $W \in \mathbb{R}^{K_1 \times K_2 ... K_n \times M}$, where $S_1, S_2, ...S_n$ are the spatial dimension over which convolution takes place, $K_1, K_2 ... K_n$ are the spatial dimension of W, or the receptive field of CNN, and M, N are input and output feature size respectively.

$f_c(x)$ for an ordered data is given by,

$$f_c(x, i) = \sum_K x_{i:i+K} \circ W \tag{3.12}$$

where K denotes the summation of features across the spatial dimensions, and i is the vector representing a location of x.

CNN's are inspired by the concept of convolution in signal processing, but the mathematical operation of CNNs computes correlation. Some of the key benefits provided by CNNs over fully connected layers(FCN) are,



(a) Fully Connected Network (b) Convolutional Neural Network

Figure 3.2: Circle represents an embedding, while line represents learnable weights. In FCNs, weights are different (even for three As or two Ps), while in CNNs, weights are shared (shown with different color), and hence the network can generalize the feature for three As based on its neighbourhood. Source: Modified; originally from Chapter 4: Convolutional Neural Network by Oreilly.com

- Size of the learnable parameter is independent of the input, x, and is controlled as receptive field hyperparameter. In case of FCNs, size is directly proportional to x (Fig. 3.6a).
- CNNs requires less training data when compared to FCNs, especially when the data is repetitive because of parameter sharing (Fig. 3.6b).
- CNNs are also known as a regularized version of FCNs as the scale of connectivity and complexity is less than or equal to FCNs depending upon the receptive field.

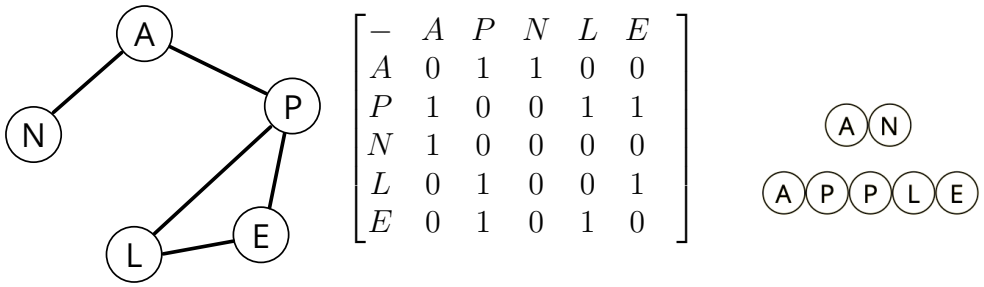
3.3 Graph Convolutional Network

The convolution operator defined in Eq. 3.12 models the relationship between data based on the spatial ordering of the data. i.e., for a given location, the output feature would depend on the following k elements. But, a graph is not guaranteed to be ordered; for example, in Fig. 3.3a we can not say whether P comes before L or after, as Apple and LEP are possible and valid structures.

We can illustrate the importance of ordering in CNN with the help of adjacency matrices, which is one of the fundamental ways to represent a graph (Fig. 3.3b).

If we use a CNN with the receptive field of size 3, the relationship of vertex A with $\{P, N\}$ will be captured. But for P, it can either be with $\{A, N\}$, $\{L, E\}$, or $\{N, L, E\}$ even though P has $\{A, E, L\}$ as adjacent.

Thus, vanilla CNN is not a correct approach for modeling the spatial relationship of a graph, and would not be scalable as the number of vertices in the graph increases, or the sparsity of the graph increases.



$$\begin{bmatrix} - & A & P & N & L & E \\ A & 0 & 1 & 1 & 0 & 0 \\ P & 1 & 0 & 0 & 1 & 1 \\ N & 1 & 0 & 0 & 0 & 0 \\ L & 0 & 1 & 0 & 0 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

(a) (b) (c)

Figure 3.3: a) Graph, G where Each vertex represents one alphabet, with edges representing their relationships, b) Adjacency Matrix of G, and c) highlights one possible structure of interest from the graph.

Graph Convolution aims at computing convolution on graphs efficiently and correctly. If we compute the Fourier transform of Eq. 3.12, we can rewrite convolution as,

$$f_c(x, i) = \sum_K x_{i:i+K} * W \tag{3.13}$$

$$f_c^f(x, i) = x^f \circ W^f \tag{3.14}$$

where x^f, x^{-f} denotes the Fourier and inverse-Fourier transform of x wrt graph, G respectively. $x^f = U^T x,\ and\ x^{-f} = Ux$. U is the eigen vector of Laplacian, L of G. This transformation is known as convolution property.

We can write the convolutional of a graph signal with a learnable parameter as,

$$\begin{aligned} f_c(x, i) &= (x^f \circ W^f)^{-f} \\ f_c(x, i) &= U(U^T x \circ U^T W) \end{aligned} \tag{3.15}$$

We can write f_c as,

$$f_c(x, i) = U(U^T W \circ U^T x) \tag{3.16}$$

as hadamard product is commutative.

Now replacing $diag(U^T W) = W'$,

$$f_c(x, i) = U(W' U^T x) \tag{3.17}$$

because dot product and matrix multiplication with same for diagonal matrix,

The Eq. 3.17 computes the convolution of graph signal with a learnable parameter, but the Fourier transformation brings few challenges,

- Time Complexity to compute the Eigenvector is $O(V^3)$, where V is the number of vertices.
- The Convolution operation is not localized in space and computes the feature vectors of each vertex with respect to all the vertices, requiring memory proportional to graph size and affecting the distant and less-important vertices.

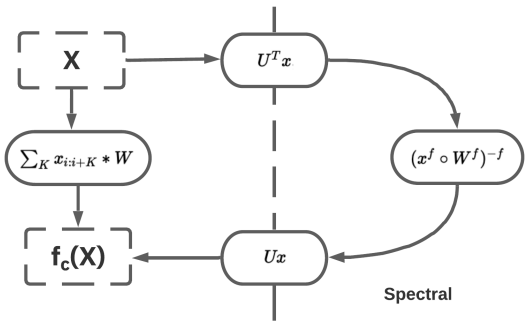
3.3.1 ChebNet

To handle the above-mentioned issues, Hammond [16] proposed ChebNet, which uses Chebyshev Polynomials [26] to approximate the graph convolution and reduce time complexity. Using Chebyshev Polynomials, we can approximate any function $f(x)$ as,

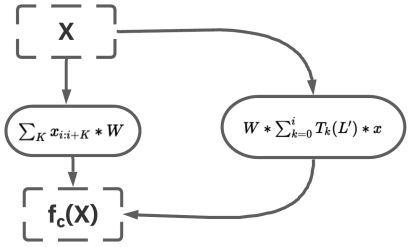
$$f(x) = \sum_{i=0}^{\infty} A_i T_i(x) \tag{3.18}$$

where $T_i(x) = xT_{i-1}(x) - T_{i-2}(x),\ given\ T(0) = 1,\ T(1) = x$, and $A_0, A_1 ... A_i$ are the learnable parameters. Also, $x \in [-1, 1]$, and $f(x)$ needs to be continuous and single value.

Thus, using Eq. 3.18, we can approximate $W' = diag(U^T W)$ as a $f(\lambda')$, where $\lambda' = 2\frac{\lambda}{max(\lambda)} - 1$. It is done to normalize the eigen value between $[-1, 1]$ as eigen value of Laplacian is always greater than or equal to 0.



(a) Computing Graph Convolution using spectral transformation



(b) Approximating the Graph Convolution using Chebyshev polynomials.

Figure 3.4: Graph Convolution Approximation - Block Diagram

$$W' = \sum_{i=0}^{k} \theta_i T_i(\lambda') \tag{3.19}$$

where, W_i is the approximation upto order k, and θ_i is the learnable parameter. Now, $f_c(x, i)$ can be written as,

$$f_c(x, i) = UW'U^T x$$

Putting the value of W' from 3.19

$$f_c(x, i) = U \sum_{i=0}^{k} \theta_i T_i(\lambda') U^T x$$

Rearranging

$$f_c(x, i) = \sum_{k=0}^{i} \theta_k (U T_k(\lambda') U^T) x \tag{3.20}$$

$$f_c(x, i) = \boxed{\sum_{k=0}^{i} \theta_k T_k(L') x}$$

where $L' = 2 * \frac{L}{max(\lambda)} - I$. The proof of Eq. 3.20 is present in A.9.

$\therefore f_c(x, i) = \sum_{i=0}^{k} \theta_i T_i(L') x$ is not just time efficient with $O(V^2)$ complexity but is space localized as θ_i can be scalar or vector of constant size. Also, k represents the receptive field of the convolution as $T_k(L')$ would compute the laplacian upto k distant vertices from a given vertex thereby reducing the effect on distant vertices.

3.3.2 GCN

ChebNet suffered from the problem of overfitting and over smoothening, and to reduce that Kipf [15] proposed few changes to the ChebNet. Instead of learning a k receptive fold using single convolutional layer, GCN proposes k convolutional layers such that each layer is a linear function of Laplacian followed by non-linearity.

Setting k = 1;

$$f_c(x,i) = (\theta_0 T_0(L') + \theta_1 T_1(L'))x$$

$$f_c(x,i) = (\theta_0 I + \theta_1 L')x$$

$$f_c(x,i) = (\theta_0 I + \theta_1(\frac{2L}{max(\lambda)} - I))x$$

Approximating $max(\lambda) = 2$;

$$f_c(x,i) = (\theta_0 I + \theta_1(L - I))x$$

$$(L = D - A \implies L = I - D^{-1/2}AD^{-1/2})$$

$$f_c(x,i) = (\theta_0 I + \theta_1(I - D^{-1/2}AD^{-1/2} - I))x$$

$$f_c(x,i) = (\theta_0 I - \theta_1 D^{-1/2}AD^{-1/2})x$$

Assuming $\theta_1 = -\theta_0 \implies \theta$;

$$f_c(x,i) = \theta(I + D^{-1/2}AD^{-1/2})x$$

Rearranging for numerical stability

$$f_c(x,i) = \boxed{\theta D^{-1/2}(A + I)D^{-1/2}x} \tag{3.21}$$

The approximation of $max(\lambda)$ is from the assumption that neural networks will be able to scale with the input, and $\theta_1 = -\theta_0$ is made to reduce the parameter in a single layer. K separate layers over a single layer with k convolutions are used to increase the overall non-linearity.

3.3.3 SGN

In GCN, the underlying assumption is that the learning would improve due to presence of non-linear activation between many layers with less parameters, as shown

with CNNs but SGN [17] hypothesise that presence of non linearity between layers is not of importance as much of the benefit arise from averaging with the neighbourhood vertexs using Laplacian matrix. Thus, the network becomes,

$$\text{GCN k layer network,} \tag{3.22}$$

$$O_k(x) = \sigma(f_c(..k....\sigma(f_c(\sigma(f_c(X_i))))))) \tag{3.23}$$

$$\text{Removing Non linearity,} \tag{3.24}$$

$$O_k(x) = f_c(..k....(f_c((f_c(X_i))))) \tag{3.25}$$

$$\text{Putting } f_c(x) \text{ as } L^*\theta x, \text{ where } L^* = D^{-1/2}(A+I)D^{-1/2} \tag{3.26}$$

$$O_k(x) = (L^*...k...L^*)(\theta_k....\theta_0)X_i \tag{3.27}$$

$$O_k(x) = L^{*^k}\theta X_i \tag{3.28}$$

Thus, we can collapse the k layers as a single layer. The network can learn the feature representation up to the k layer using a single pass while avoiding additional parameters and improving numerical stability compared with ChebNet.

3.4 Sub Sampling

Graph Convolutional Network enables scalable modeling for small to medium graphs, but as the size of the graph increases (5.1.3), the computational requirements increase too.

With the ordered data, it is easier to crop the input images w.r.t. to the overall receptive field of the network. Cropping (or subsampling) is not simple as each layer model's k neighbor relationships and different vertices may have distant neighbors.

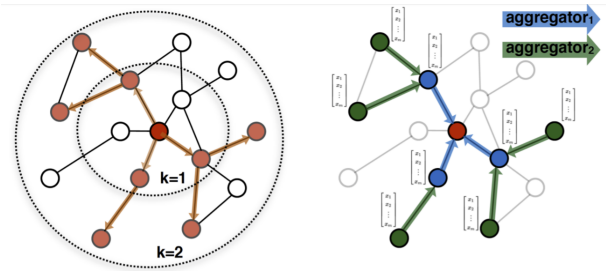


Figure 3.5: Sub Sampling of Graph for training, Source: towardsdatascience.com

In GraphSaint [27], a subsampling technique was proposed, which aims at selecting subgraphs to 1) to efficiently train large graphs and 2) regularize large graphs (similar to dropout) to reduce over smoothening. Thus, sub-sampling enables training on a portion of the dataset, accompanied by improvement in generalization.

3.4.1 Random Sampling

The most naive technique is to sample the vertices randomly, i.e.

$$S_N = random(V, N) \tag{3.29}$$

where $random(x, k)$ function uniformly samples k integers from $0 \rightarrow x$ without repetition. While the random sampler adds minimal computational penalty, it is not used as it might ignore the neighbors of a given vertex and select un-related vertices altogether, especially when working with sparse graphs.

3.4.2 Node Sampling

The Node sampling techniques sample the vertices based on the importance of a given vertex, as shown in algorithm 5. The vertices are initially sampled with repetition, and once the vertices are sampled, only unique vertices are used as sampling copies of the same vertex introduces instability in the system and is not correct.

Algorithm 5: Node based Subsampling Algorithm

Result: $S_N = set(V_1, V_2.....V_k)$

initialization : $P(v_i) = \frac{\sum_j W_{ij}}{\sum_j P(v_j)} \forall i, N_s = 0, S_N = set()$;

while $N_s < k$ **do**

    # where $|=\sim$ samples one element from v with repetition

    $S_N| =\sim P(v)$;

    N_s++;

end

Thus, NodeSampler returns vertices that are less than or equal to k, which is the size of a sub-sampled graph.

NodeSampling, on the one hand, adds an extra computation cost but reduces the

randomness in the sampled graph. It is still not well suited for sparse graphs as the probability of selecting an edge is directly proportional to the likelihood of choosing both vertices. If one vertex is less connected, it will result in under-training those relationships and introduce biases. In order words, if we have sparse graph then a node-sampled graph is more probable to include most densely connected vertices of each cluster than neighbours of a particular cluster.

3.4.3 Ripple Walk SubSampling

Ripple Walk SubSampling [28], RWS is a modified random-walk sampling algorithm [27]. Random Walk Sampling has two key advantages, 1) S_n is more probable to contain the neighbor vertices than vertex-sampling, and 2) It samples the neighboring vertices uniformly, thereby reducing the bias. Initially, the samples select one vertex to S_N, followed by recursive addition of f neighbors of S_n, until k vertices are sampled.

Algorithm 6: Ripple Walk Subsampling Algorithm

Result: $S_N = set(V_1, V_2.....V_k)$

Initialization : $S_N = set(), k \in I$;

$S_N| = \sim U(V, 1)$; # $U(V, f)$ randomly selects f values $\in [0, V]$.

while $len(S_N) < k$ **do**

  $N_C = len(S_N)$;

  $S_C = set()$;

  for $N_i \in S_N$ **do**

    $S_C| = neighbours(N_i)$; # $|=$ is in-place logical OR.

  end

  $S_N| = U(S_C, f)$;

  if $len(S_N) == N_C$ **then**

    # No new neighbours are added. Adding another root node.

    $S_N| = \sim U(V, 1)$;

  end

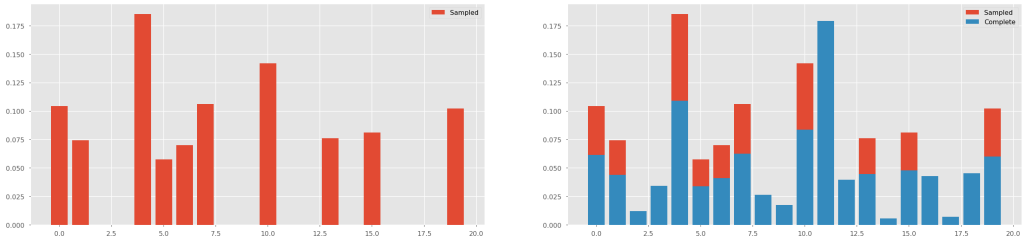
end

RWS algorithm is biased towards neighbors of a vertex and vertices within a cluster rather than the center of multiple clusters. Also, the number of sampled vertices is always equal to k, improving the computational efficacy during training.

3.4.4 Bias from SubSampling

A sampler algorithm introduces two kinds of biases, 1) the probability with which a particular vertex is selected and 2) computing vertex features from selective neighbors. While the former is mitigated with the help of the RWS algorithm. For the latter, a normalization technique was proposed in GraphSaint [27]. It aimed at rescaling each edge by dividing it with a factor of $\frac{P(E_{ij})}{P(v_j)}$. The factor reduces the effect of more probable edges on the training as the probability of selecting edge wrt to given vertex, $P(E_{ij}|v_j)$.

For the NodeSampling algorithm, we can compute edge and vertex probability analytically. It is much more complex for the RWS algorithm and required repeated sampling for numerical estimation, increasing the computation cost. We independently observed that computing the Laplacian on the subsampled graph improves the overall stability of the system, as later found in the implementation of GraphSaint [27] later on. Thus, we compute the Laplacian of the subsampled and do not perform any further normalization on it.



(a) Vertex features when 50% sampled, (b) Rescaled with complete graph,

Figure 3.6: Handling Bias by recomputing Laplacian for each sub-graph

3.5 Sequence Learning

The aim is to predict ETA for the next k' minutes given the input features of previous k minutes. We use the seq2seq [12] model, which accomplishes it in two

phases. In the first phase, known as encoder, last k sequences are modeled, 2) followed by the second phase known as decoder, which consumes the embedding from the encoder to predict the output for the next k' steps in auto-regression fashion.

In autoregressive settings, an algorithm not just depends on the encoder's embedding but also on the output of the previous time step too,

$$X_t = f(X_e, X_{t-1}) \tag{3.30}$$

where X_e is the encoder embedding, and X_{t-1} is the output from the previous time-step. Input, X_{t-1} to the f can be noisy, especially during the early stages of the learning, and to mitigate that, Williams [29] proposed teacher force learning,

$$\begin{aligned} X_t &= f(X_e, X^{"}_{t-1}), \text{during training} \\ X_t &= f(X_e, X_{t-1}), \text{during evaluation} \end{aligned} \tag{3.31}$$

where $X^{"}_{t-1}$ is the ground truth value at $t - 1^{th}$ time step. With the help of teacher force training, we can remove the noisy estimate to f.

3.5.1 Curriculum Learning

The distribution learned through teacher force training expects ground truth as input, which is not available during evaluation, thereby less generalizable. Bengio [30] proposed curriculum learning, an extension to the teacher force training(*TTF*) in which with ϵ probability, input to f is the previous output and ground truth otherwise. Initially, the value of $\epsilon = 0$, and as the training progress, its value is increased to 1. It introduces a decay curve, which controls the threshold, ϵ known as teacher-training threshold (TTR).

$$\begin{aligned} X_t &= f(X_e, X^*_{t-1}) \\ \text{where } X^*_{t-1} &= \epsilon * X^{"}_{t-1} + (1 - \epsilon) * X_{t-1} \end{aligned} \tag{3.32}$$

We also performed experiments on more complex techniques for Autoregressive learning but did not yield much improvement over curriculum learning and can be found in A.2.

3.6 Missing Data

The Recurrent Neural Networks assumes that every input feature is available at a given time, but it can not be guaranteed for real-world datasets. Thus handling missing data is essential to ensure consistency and generalizable performance.

3.6.1 GRU-D

GRU-D [31] extends the GRU cell to handle missing data. At first, it learns the mask which imputes the missing data with the linear combination average and last seen value depending upon a learned mask (γ_1). After that, it rescales the state matrix based on γ_2.

$$X = X \circ m + (1 - m) \circ (X_1 \circ \gamma_1 + (1 - \gamma_1) \circ X_2) \tag{3.33}$$

where $\circ$ is the element wise multiplication, X is the input, m is the mask of X; $m_{i,j}$ is 0 if an entry is missing. X_1 is the input value available at any previous time step. X_2 is the average value of input feature. γ_1 is the learnable parameters,

$$\gamma_1 = \begin{cases} e^{-(W_1 * d_t + b_1)} & W_1 * dt + b_1 >= 0 \\ 0 & otherwise \end{cases} \tag{3.34}$$

where W_1, and b_1 are the learnable parameters, and d_t stores the time at which previous entry was available. Similarly, γ_2 with W_2 and b_2 learnable weights are used to rescale h, $\boxed{h = \gamma_2 \circ h}$.

GRU-D also adds the missing data information to the recurrent output of GRU as

$$\begin{aligned} v &= \sigma(Wx_t + Uh_{t-1} + b + \boxed{\beta_1}) \\ [r_t, z_t] &= v \\ h_t^* &= tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h + \boxed{\beta_2}) \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ h_t^* \end{aligned} \tag{3.35}$$

where $\beta_1 = W_1^\beta * m$, and $\beta_2 = W_2^\beta * m$; W_1^β, W_2^β are the learnable parameters.

Chapter 4

Scalable ETA prediction Framework

We have analyzed multiple algorithms, such that each algorithm increases the overall complexity of the system. Each algorithm takes an input, $I \in \mathbb{R}^{V \times T \times K}$, where V represents vertices (location for traffic data, edges for ETA-Data), T represents the time sequence, and K represents the number of features. It returns an output, $O \in \mathbb{R}^{V \times T}$, which is the speed for METR-LA, SP-DT, and time elapsed for ETA-DT.

4.1 Mean

As stated in Occum's razor [32], or parsimony principle, the simplest explanation is the best explanation. Thus, the mean model returns an average value for a vertex at a given time over training samples. Given an input data consisting of time elapsed $I \in \mathbb{R}^{N \times V \times T}$,

$$O_{v,t} = \frac{1}{N} \sum_{N} I_{v,t}, \tag{4.1}$$

where v represents vertex, t represents time, N represents the number of samples. On the one side, it does not model any form of relationship, but it is also very fast as just indexing is required during evaluation.

4.2 LSTM

LSTM uses seq2seq architecture [12] (Fig. 4.1). The encoder consists of Conv1D and GRU layers to generate the embedding for the input sequence. It takes an input, $I \in \mathbb{R}^{T \times (V*K)}$, which has features of all the vertices concatenated across the last dimension, and the encoder returns a latent embedding, $L_e \in \mathbb{R}^N$ where N is the size of latent embedding. The decoder comprises of GRU cells to autoregressively estimate the output for T' future steps.

LSTM, on the one hand, enables the relationship modeling of the data, but on the

other hand, it is not flexible with the changes in the number of vertices. It is also not possible to evaluate the model on specific vertices.

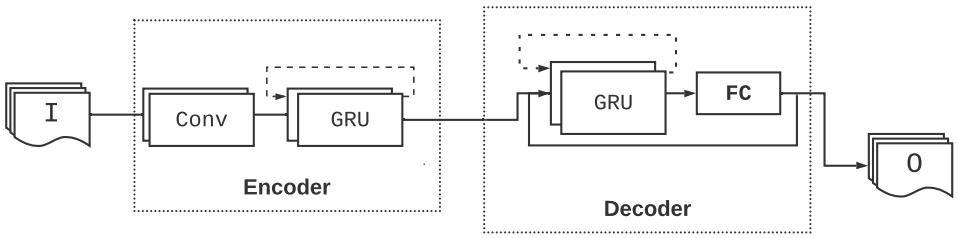
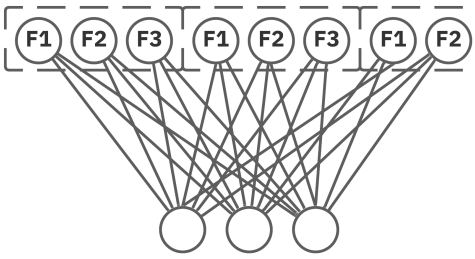


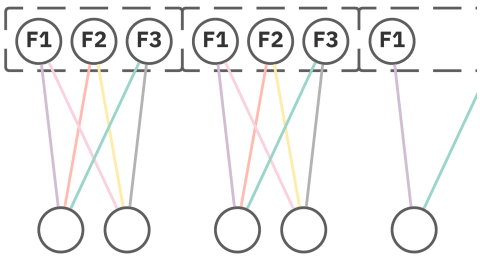
Figure 4.1: Seq2Seq Architecture

4.3 ConvLSTM

ConvLSTM [14] builds upon the work of FC-LSTM [13] in which the parameters across vertices (Fig. 4.2) are shared i.e. the temporal relationship across vertices is modeled independently. In ConvLSTM, a Conv2D block is used before the encoder to incorporate the relationship between the vertices represented by the adjacency matrix.



(a) LSTM: Independent parameters for all the vertices



(b) FC-LSTM: Parameters *F1*, *F2*, *F3* are shared across vertices

Figure 4.2: LSTM vs FC-LSTM

It takes an input, $I \in \mathbb{R}^{T \times V \times K}$, without any concatenation. The output from the Convolutional block, $L_c \in \mathbb{R}^{T \times V \times M}$ is transposed to $\mathbb{R}^{V \times T \times M}$. After that, the GRU layer in the encoder is used to generate the latent embedding, $L_e \in \mathbb{R}^{V \times N}$. The

autoregressive decoder then consumes the latent embedding to estimate the output for T' future steps.

4.4 DCRNN

Diffiusion Convolutional Recurrent Neural Network [19], DCRNN is one of the state of the art architecture to model the Spatio-temporal relationships. It uses Diffusion Convolutional to model the spatial relationships and GRU for temporal relationships. The GRUCell has been modified to compute diffusion convolution before applying any fully connected layer, as shown in Fig. 4.3

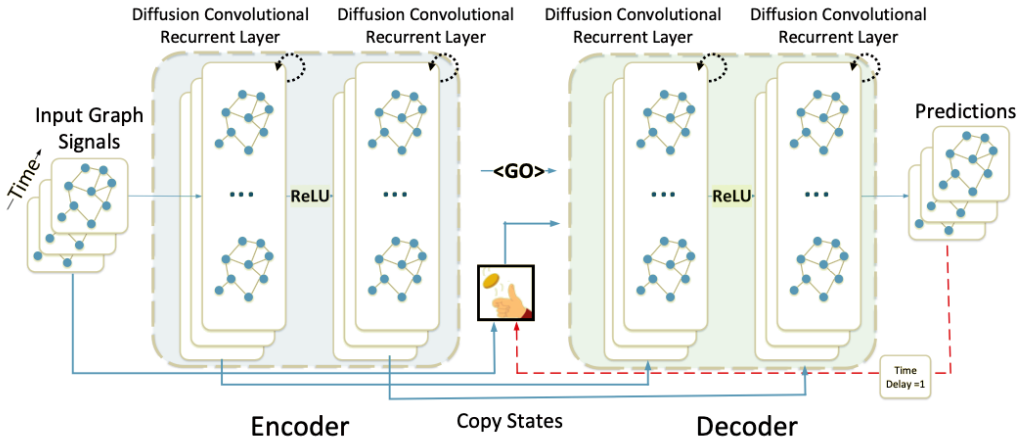


Figure 4.3: DCRNN : Taken from the original source [19]

Diffusion Convolution layer uses the stationary distribution of the graph for modeling,

$$D_s = \sum_{i=0}^{\infty} \alpha(1-\alpha)^i (D^{-1}W)^i \tag{4.2}$$

where D_s, represents the Stationary distribution, D^{-1} represents the degree matrix, and W matrix represents the adjacency matrix. They have used ChebNet approximation(3.3.1), and truncate the layer upto k^{th} order of Laplacian. Thus, the Diffusion convolution of X wrt θ is,

$$D_c = \sum_{i=0}^{k} (\theta_k^1 (D^{-1}W)^i + \theta_k^2 (D^{T^{-1}}W)^i) \tag{4.3}$$

where D^T, aims at modelling the in bound traffic to a particular vertex.

The modifications in GRUCell (from 3.11) are,

$$v = \sigma(\boxed{D_c^0}([x_t, h_{t-1}]) \tag{4.4}$$

$$[r_t, z_t] = v \tag{4.5}$$

$$h_t^* = tanh(\boxed{D_c^1}([x_t, r_t \circ h_{t-1}])) \tag{4.6}$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ h_t^* \tag{4.7}$$

4.5 Proposed Framework

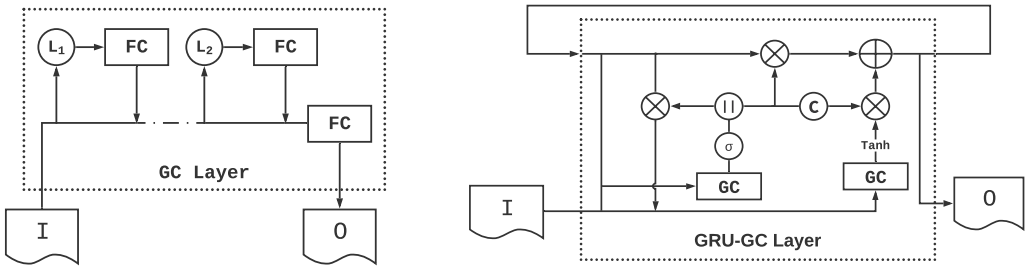
Simplified Spatio-Temporal Graph (SSTG) is our proposed framework, with an emphasis on simplifying Spatio-temporal modeling. It is based on seq2seq architecture and takes an input of shape $I \in \mathbb{R}^{V \times T \times F}$, where V are the number of vertices, T is the time-sequence length, and F is the number of input features. It returns the output of shape $O \in \mathbb{R}^{V \times T}$.

The input is first processed by the encoder, which consists of two Spatio-Temporal Graph(STG) Blocks. Each STG block is a modified GRUCell which models the relationship using graph convolution, GC operating in residual fashion [33]. Each GC layer, GC_k consumes output from the previous layer and input to the block using k^{th} order of Laplacian. The structure of the GC layer and STG Block is shown in 4.4.

The encoder processes input tensor sequentially, such that input, I_e and h_e at each time $t = t_i$ is

$$\begin{aligned} I_e &= I[:, t_i] \\ h_e &= \begin{cases} 0 & t_i = 0 \\ E_{t_{i-1}} & otherwise \end{cases} \end{aligned} \tag{4.8}$$

where E_{t_i} is the embedding of the encoder at time, t_i. The embedding of encoder at time t_e is fed into decoder as the initial hidden state, where t_e is the time of last input sequence. The input to the decoder, I_d and h_d at each time $t = t_i$ is,



(a) **GC-Layer**: I and O are input and output respectively. FC is a fully connected layer, comprises of dropout, single layer Neural Network, and LeakyReLU activation. $\textcircled{L_i}$ takes dot product with i^{th} order laplacian.

(b) **GRU-GC Layer**: σ and $Tanh$ are the activation functions. The self loop on top represents the hidden state. $\textcircled{||}$ splits the output into two, across the latent dimension. $\bigoplus$ and $\bigotimes$ performs addition and multiplication respectively. $\textcircled{C}$ computes the complement.

Figure 4.4: Block Diagram of GC Layer and STG

$$I_d, h_d = \begin{cases} I[:, t_e, :1], E_{t_e} & t == t_e \\ O[:, t], D_{t_{i-1}} & otherwise \end{cases} \tag{4.9}$$

where D_{t_i} is the embedding of decoder at time, t_i. We use ground truth input at time t_e acting as the initial input, I_e. The ETA for the further time steps are generated in autoregressive manner (3.5), as shown in the Fig. 4.5.

We use GRUD-GC as the first cell of the encoder to handle the sparsity in the input data (Fig. 4.5). GRUD-GC integrated GRUD (3.6.1) updates with the GRUCell and takes four extra inputs, $\{I_1, I_\mu, M, D_t\}$, where $I_1 \in \mathbb{R}^{V \times T \times F}$ is the last non-missing value, $I_\mu \in \mathbb{R}^{V \times F}$ is the average value of a feature, $M \in \{0,1\}^{V,T,F}$ is the binary mask where 0 denotes missing-entry, $D_t \in I^{V,T,F}$ is the total time for which a particular feature has been missing.

GRU-D transforms the input at any time step t,

$$I \rightarrow I \circ M + (1 - M) \circ (\gamma_1 \circ I_1 + (1 - \gamma_1) \circ I_\mu) \tag{4.10}$$

The hidden state is transformed as $\gamma_2 \circ h_d$. We do not use the GRU-D transformations on recurrent output as we did not observe any considerable performance boost.

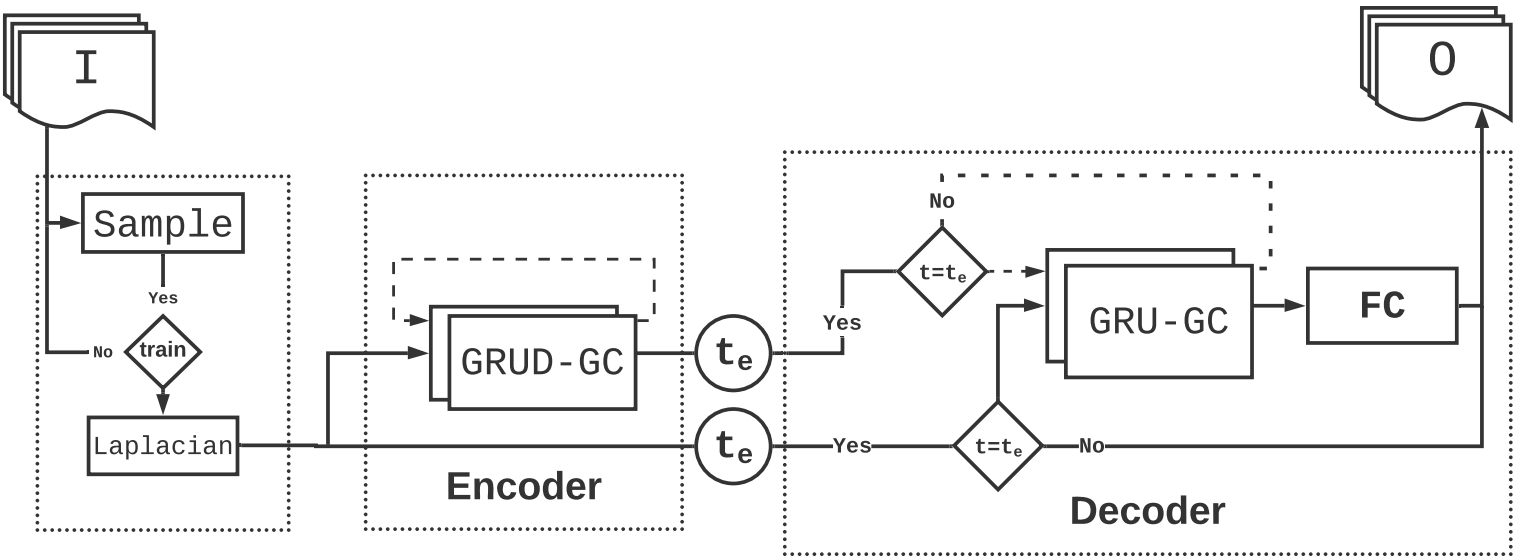


Figure 4.5: Architecture- SSTG : Condition $t = t_e$ is used to feed different inputs at the initial time of the decoder. $\textcircled{t_e}$ returns the input at time, $t = t_e$.

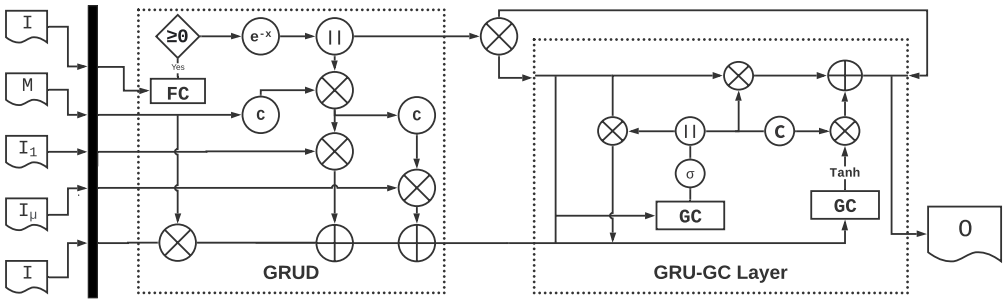


Figure 4.6: GRUD-GC : Condition $\boxed{\geq 0}$ is used to ensure the output after $\boxed{e^{-x}}$ transformation stays in the range $[0, 1]$.

4.5.1 Training Procedure

During training, input to the network is sampled using the RWT sampling algorithm (3.4.3); the sub-graph corresponding to the sampled vertices is used to compute Laplacian. We used random-walk normalized Laplacian because the sum of all edges for a given vertex is 1 for random-walk Laplacian, and is $\sum_{i=1}^{N} \sqrt{\frac{D_v}{D_i}}$ for symmetric normalization. Thus, the symmetric Laplacian requires modeling of degree for a given vertex, which poses a challenge as the degree of vertices changes when sampling.

The input to the decoder can be the ground truth sample or the previous output. We use curriculum learning (3.5.1) in which the network gradually switches from ground truth to the previous output. After testing multiple decay functions, we settled with the decay function similar to $Tanh$ over the number of training steps.

4.5.2 Evaluation Procedure

During the evaluation, the network can receive input based on three different policies:

1. **RWT sampling**: Sampler can used to sample a fixed amount of vertices; it is of little utility during evaluation as we are looking for ETA prediction of all the vertices or a specific set of vertices.

2. **Complete Graph**: We can input the complete graph to the network; it does not require any normalization or fine-tuning because the Laplacian is random-walk normalized.

3. **Specific Vertex**: If we are looking for output for a specific set of vertices, we can set those vertices as initial for the RWT sampler and evaluate at a much faster rate when compared to the complete graph. It operates without any modification, but to get the stable output, we need to ensure a good percentage of neighbors are present in the set of sampled vertices.

The output at the previous step acts as input for the decoder, and thus the decoder entirely operates in AutoRegressive Fashion.

Chapter 5

Experimental Results and Discussion

5.1 Datasets

To quantitatively analyze our framework, we evaluated the performance on a standardized dataset, METR-LA, and then extended them into our dataset, ETA-DT.

5.1.1 METR-LA

METR-LA [19] is a traffic dataset on Los-Angeles County, USA and has 207 sensors with each sensor collecting the data for the duration of 4 months. We used this dataset to analyze and iterate over different ideas. It uses gaussian kernel to compute the weighted matrix,

$$a_{i,j} = \begin{cases} 0 & ||X_i - X_j|| < k, \\ e^{\frac{-||X_i - X_j||^2}{\sigma^2}} & otherwise, \end{cases} \tag{5.1}$$

where X_i and y are geolocation of two sensors, σ is the standard deviation of the distances, and k is the constant to control sparsity.

5.1.2 Delhi Traffic data

SP-DT is the speed traffic dataset on Delhi, India, and uses 519 sensors across different road segments. The data was collected for over 60 days with a sampling frequency of 15min from 7:00 AM to 11:00 PM. The data can be organized as a tensor, $T \in \mathbb{R}^{519 \times (60*67)}$. This dataset was initially used, for a minimal use case of testing the performance on missing data.

5.1.3 Delhi ETA data

ETA-DT[2.2] is a time-velocity dataset on DTC buses, Delhi, India. Each time step consists of time taken to travel between an edge, i.e. from one stop to another.

There are 6639 such edges. We only take edges with at most 70% sparsity for training and evaluation, which turns out to be 1169. For training and evaluation, data of the five months has been used. For more details on the development and cleaning of the dataset, please check out section 2.2. To generate the adjacency representation between the edge's, we computed the spearson [34] correlation between a pair of edges and selected the top 9% of the coefficients to ensure sparsity in the Graph. We used constant seed for batch sampling across all of our experiments to increase the training consistency.

5.2 Training Details

To ensure fair analysis between models, we used similar hyperparameter and learning settings when possible.

5.2.1 Model Parameters

A deep neural network can compose of several hyperparameters, and it is not possible to validate every permutation of the parameters. For the same reason, we went with standard values of parameters and only updated them when required. In all of our tests, the number of units in a GRU layer is 64 or 128 for the METR-LA dataset and 128 to 512 for the ETA-DT dataset. We used two RNN layers at the time of encoding and two layers when decoding with the same hidden dimension. It was done to ensure the final state of the encoder can be directly used as the initial state of the decoder.

5.2.2 Learning Rate

Our Initial Experiments used a learning rate of 10^{-3}. Later we employed the learning rate decay after 15 (or 20) epochs by the factor of 0.1 every 10 epochs. It worked well for LSTM based networks, but with GCNs, we observed that imcrement in the decay factor to 0.5 improves the numerical stability and better convergence.

5.2.3 Optimization

We used Adam optimizer for training analysis on METR-LA dataset but switched to AdaGrad optimizer when performing experiments on ETA-DT dataset. It was done to reduce the effect of missing data on gradient explosion and stabilize the training. The value of ϵ was also increased from 10^{-7} to 10^{-3} to stabilize the training.

The gradient explosion was leading to overfitting and instability on very primitive models. We added batch normalization before every convolutional layer, ensuring that the input distribution to each convolutional layer is standardized. With RNN, we employed dropout as implementing vanilla batch normalization before every time step is incorrect because batch normalization would standardize the input distribution across time steps, which is not guaranteed to be true.

5.2.4 Loss Function

Initially, we used mean square error to minimize the error and solve the problem as a regression problem. The ground truth of two vertices could differ by a huge margin, and thus using the absolute loss would have less effect on few vertices,

- MSLE loss:

$$MSLE(X_i, X_i') = log^2\left(\frac{X_i + 1}{X_i' + 1}\right) = \left(log(X_i + 1) - log(X_i' + 1)\right)^2 \quad (5.2)$$

where X_i' is the predicted value, and X_i is the ground truth.

Not only it is relative, but it also penalizes the underestimates more than the overestimates. If the value of X_i is 20, and of $X_i' = 10$ then MSLE loss would be

$$log^2\left(\frac{21}{11}\right) = log^2(1.90) \implies 0.07$$

but if the value of $X_i' = 30$ then the MSLE loss would be

$$log^2\left(\frac{21}{31}\right) = log^2(0.67) \implies 0.03$$

- Weighted Loss: We weight the loss for underestimates more than that of over-estimates,

$$W_h(X_i, X_i') = \begin{cases} L(X_i, X_i') & X_i' >= X_i \\ \alpha L(X_i, X_i') & otherwise \end{cases} \tag{5.3}$$

  where α is a constant.

  The Weighted loss is absolute in nature, while the MSLE is not defined in some cases, for example, $X_i' \leq -1$.

- MSLE-Weight Loss: We compute the MSLE loss to ensure relative error but uses different weights for under-estimates as we observed MSLE significantly underestimates.

$$C_h(X_i, X_i') = \begin{cases} MSLE(X_i, X_i') & X_i' >= X_i \\ \alpha MSLE(X_i, X_i')^2 & otherwise \end{cases} \tag{5.4}$$

5.3 Metrics

To quantitatively analyze the performance of different algorithms, we defined five metrics with contrastive qualitative attributes,

5.3.1 Mean Square Error

Mean Square Error (or MSE) calculates the expected value of square different between ground truth and output.

$$MSE(X, X') = \frac{1}{N} \sum_{i=0}^{N} (X_i - X_i')^2 \tag{5.5}$$

5.3.2 Mean Absolute Error

Mean Absolute Error (or MAE) calculates the expected value of absolute difference between ground truth and output.

$$MAE(X, X') = \frac{1}{N} \sum_{i=0}^{N} |X_i - X'_i| \tag{5.6}$$

5.3.3 Root Mean Square Error

Root Mean Square Error (or RMSE) takes the square root of Mean Square Error.

$$RMSE(X, X') = \sqrt{\frac{1}{N} \sum_{i=0}^{N} (X_i - X'_i)^2} \tag{5.7}$$

Outliers have less effect on RMSE than MSE; MAE is a linear score and weighs each input equally.

5.3.4 Mean Relative Error

Mean Relative Error (or MRE) calculates the expected value of the ratio of the absolute difference between ground truth and output and ground truth. MRE measures the relative error, enabling comparison between different datasets.

$$MRE(X, X') = \frac{1}{N} \sum_{i=0}^{N} \frac{|X_i - X'_i|}{|X_i|} \tag{5.8}$$

On the one side, MSE is more biased towards higher values in error; MAE highlights the average error present. It is possible that between two observations, MAE stays the same while MSE reduces e.g.

$$G, O_1, O_2 = [0, 0]; [1, 1]; [2, 0];$$

MAE after $O_1 = \frac{|1-0|+|1-0|}{2} \rightarrow 1$, MSE after $O_1 = \frac{(1-0)^2+(1-0)^2}{2} \rightarrow 1$,
MAE after $O_2 = \frac{|2-0|+|0-0|}{2} \rightarrow 1$, MSE after $O_1 = \frac{(2-0)^2+(0-0)^2}{2} \rightarrow 2$.
While the MSE has increased, the value of MAE stays the same, highlighting an

increment in the variance over bias.

5.3.5 Overestimate Percentage

When the model is trained with MSLE loss function, another metric is used which checks for percentage of time the network over estimates the results.

$$O_P(X, X') = \frac{100}{N} \sum_{i=0}^{N} (X'_i \geq X_i) \tag{5.9}$$

5.4 Experimentations

We have designed multiple experiments to understand each moving part of this system to develop a simple, scalable yet robust framework. There are four experiments, Graph Convolution, Sub Sampling, AutoRegression, Missing Data. The first three studies were extensively performed on the METR-LA dataset. In the meantime, we used SP-DT dataset to study the missing data problem. The learnings helped in designing a framework for the ETA-DT dataset. We used ETA-DT for detailed study on missing data because of higher sparsity and experiments on arrival time estimation.

5.4.1 Convolution layer

In this experiment, we used METR-LA (5.1.1) dataset and compared the performance of DCRNN [19] with other Graph Convolutional techniques. The aim is to observe the performance penalty if we replace the Diffusion Convolution Layer in DCRNN with standard GCN networks. In DCRNN, authors have claimed that Diffusion Convolution improves performance. But, when we compared the performance with similar specification GCN, we observed very similar results. There was only one change,

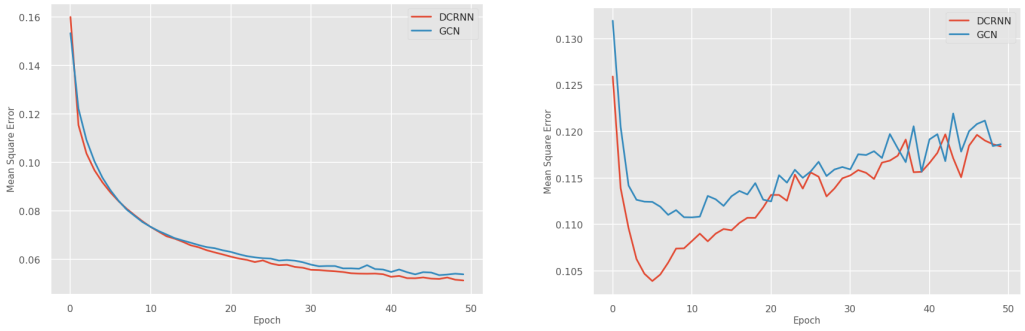
- DCRNN uses diffusion convolution, which is random-walk normalized Lapla-

cian (L_U), while GCN uses symmetric normalized Laplacian (L_N).

$$L_N = D^{-1/2} A D^{1/2}$$
$$L_U = D^{-1} A \tag{5.10}$$

For each vertex, L_U represents the relationship with other vertices in the form of unit vector such that the sum for all the edges is 1; while it is $\sum_{i=1}^{N} \sqrt{\frac{D_v}{D_i}}$ for symmetric normalized Laplacian. The intuition is that L_N carries more information about the Graph and should perform better if not equally.

We observed similar performance(Fig. 5.1) during training, with a 4% increase in error during validation, which reduces down to ϵ as the training progressed and the model over-fitted.



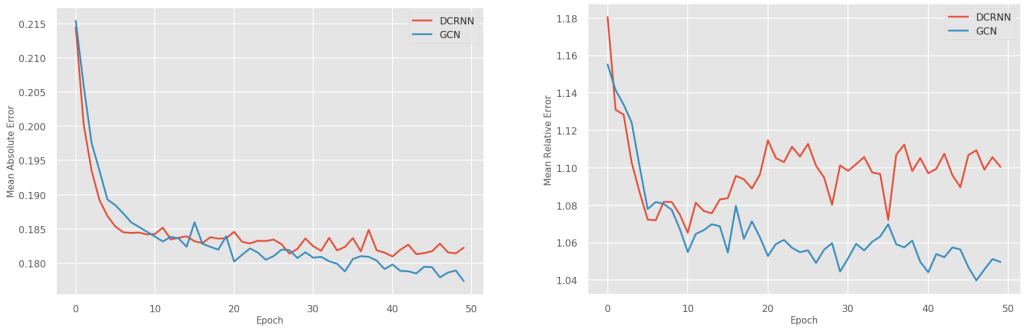
(a) Training Error (b) Validation Error

Figure 5.1: GCN vs DCRNN

The key benefits from this experiment were that GCN layer performed at par with DCRNN in terms of minimization of MAE error (Fig. 5.2), highlighting the overall reduction in the variance of the model. The gap between the model increased even further when we compare percentage error.

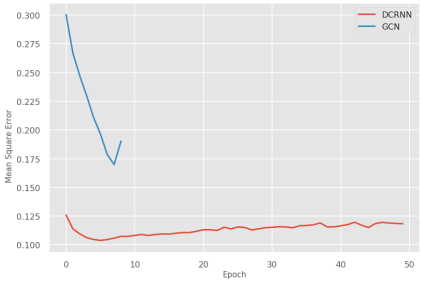
The next step towards simplification was to use the SGN network in place of GCN; when we used a single layer of order k, we observed a dip in performance(Fig. 5.3), we think it is because of over-smoothing in single order GCN.

The results from GCN motivate us to simplify Spatio-temporal modeling. We used vanilla GRU cells in the encoder, followed by multiple SGC layers stacked in residual learning configuration i.e. input to the k^{th} layer is the output of previous



(a) MAE Validation Error (b) MRE Validation Error

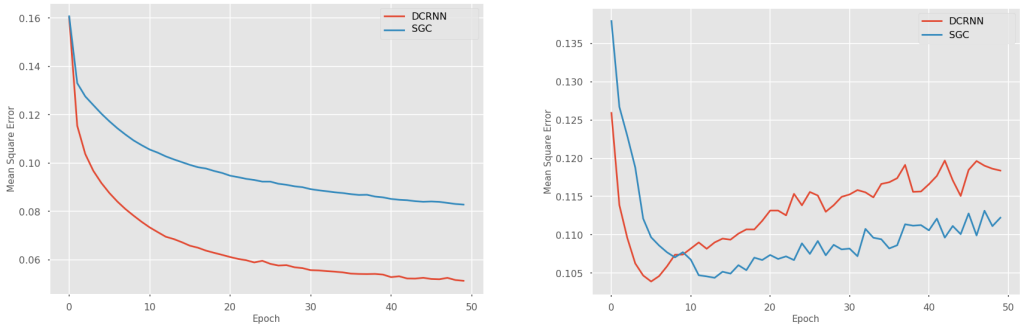
Figure 5.2: GCN vs DCRNN



(a) Validation Error

Figure 5.3: Single Order Convolution

layer and input. The embedding is then fed to the decoder. We obtained 1) better performance during validation and increment in training error which might be due to less overfitting(Fig. 5.4, 5.5).

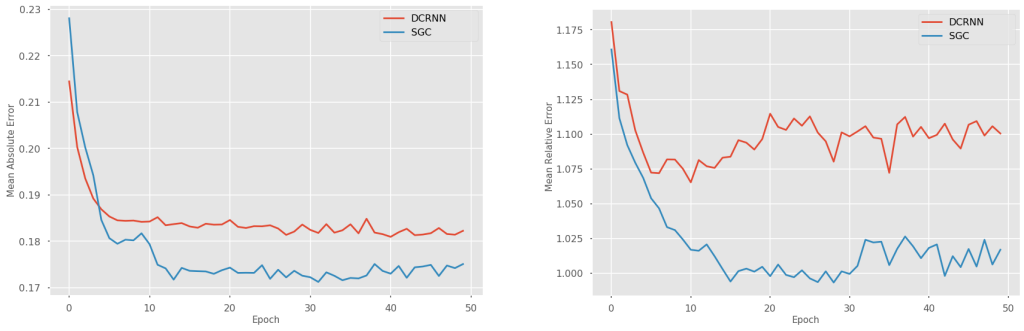


(a) Training Error (b) Validation Error

Figure 5.4: DCRNN vs SGC : Training and Error Loss Minimization

Also, the simpler model is 60% faster when compared with DCRNN and 53% faster when compared with GCN. The immediate performance boost comes because we are no longer multiplying the hidden embedding with normalized Laplacian at

every time step.



(a) MAE Validation Error (b) MRE Validation Error

Figure 5.5: DCRNN vs SGC : Performance on Validation set

Results for each convolution technique, along with the time taken are shown in Table. 5.1. The results are generated based on average performance over last 5 epochs if not specified.

	Training Error				Validation Error				Training Time
	MSE	MAE	MRE	RMSE	MSE	MAE	MRE	RMSE	Epoch/sec
DCRNN	**0.0461**	**0.1309**	**0.9726**	**0.2143**	0.0905	0.1612	0.9967	0.2986	231.4s
GCN	0.0534	0.1424	1.0293	0.2303	0.111	0.1774	1.0398	0.3304	267.09s
SGC	0.0828	0.1665	1.1006	0.2861	*0.1044*	*0.1712*	**0.9933**	*0.3201*	112.74s

Table 5.1: Results for Experiment : Graph Convolution

5.4.2 Subsampling

In this experiment, the METR-LA dataset is used to evaluate different graph sampling techniques. All of the tests were performed on the DCRNN model and further extended to our model.

5.4.2.1 Learning - 1

In the literature, node sampling has been shown to perform better than without sampling. Initially, we did not get good results as the loss would reach *NaN* (Fig. 5.6) even before training of first epoch is finished, highlighting gradient explosion even after gradients clipping over 0.5 is used. Upon looking at the available implementation, we observed that unique vertices were used for training once the vertices were sampled with repetition. As at each layer, we are aggregating the features w.r.t.

normalized laplacian, it controls the amount of neighborhood information clubbed with the given feature. But, when we allow multiple copies of a vertex, the same information is added multiple times, introducing the possibility of value explosion especially when clubbed with RNNs. Taking unique vertices ensures the control propagation during training and does not change the probability distribution, which would happen had we sampled the elements without replacement.

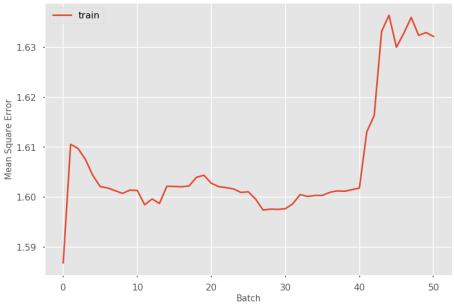
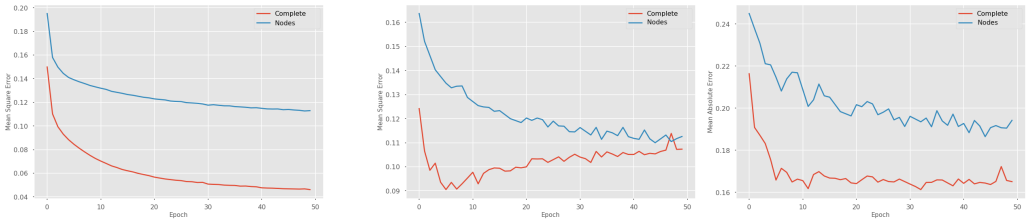


Figure 5.6: Instability due to multiple copies of vertices during node-sampling

We observed (Fig. 5.7) that network no longer suffers from parameter explosion and trains well. Still, the subsampling introduces an extra 175% error during the training, expected to a certain extent as the model is regularized. Still, it performed poorly in generalizing well with the Validation data as it has 20% more error when comparing the best performance. While it reduced the training time by 35%, the loss in performance was too significant to ignore.



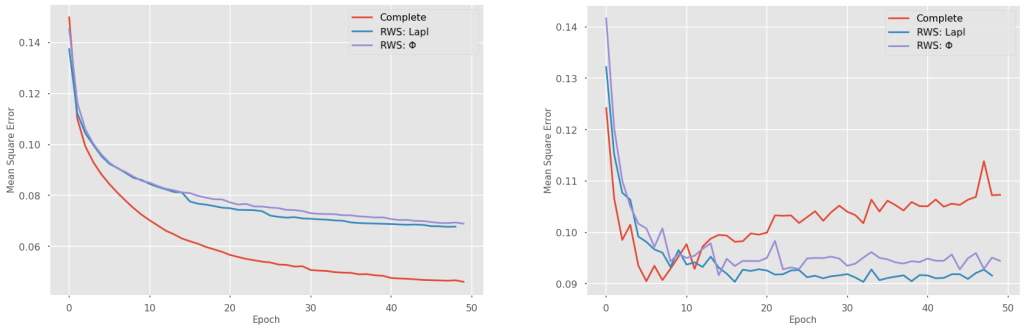
(a) MSE Training Error (b) MSE Validation Error (c) MAE Validation Error

Figure 5.7: Node Sampling on DCRNN model

5.4.2.2 Ripple Walk Sampler

Using Ripple Walk Sampler [3.4.3], we observed much better performance as compare to node sampler (RWS Sampling:ϕ) (Fig. 5.8). It underperforms in train-

ing with an extra error of 47.8% when compared to the complete model, which is way less than that of node sampling. The most significant improvement was in the validation score, highlighting that the increment in training error should improve generalization. It only had an extra 2% error in terms of best performance when compared to the complete model. It did not overfit and reduces the training time by 25% and uses just 50% of the data. The main reason behind the increase in training time compared to node sampling is preprocessing ahead of time, while RWS requires walks during every batch.



(a) MSE Training Error

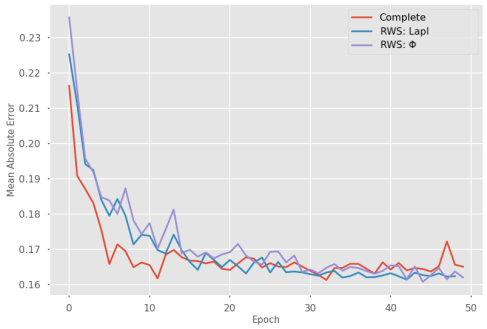
(b) MSE Validation Error

Figure 5.8: Sub Sampling

5.4.3 Bias from SubSampling

Upon implementing the GraphSaint normalization (RWS Sampling:Norm), by estimating the edge and node numerically. We observed an increment in the training error by 19%, and 11% during validation. After that, we computed the Laplacian of each subgraph (RWS Sampling:Lapl) and observed the increment in performance by 2%. Also, the best performance of the sub-sampled Graph was similar when compared with the Complete Graph, and it did not overfit as the training proceed. It also took 20% less time, as laplacian computation during every batch took 5% extra time.

Results for each subsampling technique along with time taken is shown in table 5.2,



(a) MAE Validation Error

(b) RMSE Validation Error

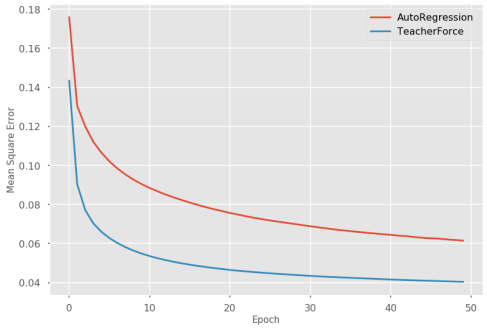
Figure 5.9: Random Walk Sub Sampling

	Training Error				Validation Error				Training Time
	MSE	MAE	MRE	RMSE	MSE	MAE	MRE	RMSE	Epoch/sec
Complete Graph	**0.0461**	**0.1309**	**0.9726**	**0.2143**	0.0905	0.1612	0.9967	0.2986	231.4s
Node Sampling	0.1121	0.1906	1.137	0.3327	0.1094	0.1860	0.9985	0.3283	153.5s
RWS Sampling: ϕ	0.0689	0.1490	1.0496	0.2613	0.0916	**0.1607**	**0.9647**	0.3006	172.79s
RWS Sampling: $Norm$	0.08215	0.1610	1.0770	0.2849	0.1023	0.1743	1.0017	0.3175	176.49s
RWS Sampling: $Lapl$	0.06770	0.14806	1.0476	0.2588	**0.0903**	0.1613	0.9764	**0.2977**	185.66s

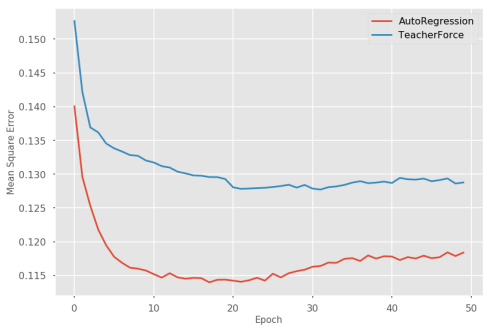
Table 5.2: Experiment : Sub Sampling

5.4.4 AutoRegression

In this experiment, we used METR-LA dataset and compared the performance of the Teacher force training mechanism with curriculum learning. Teacher force training introduces multiplicative error during validation (Fig. 5.10b). Even though the model performance is optimal during training, it underperforms during the validation.



(a) Training Error

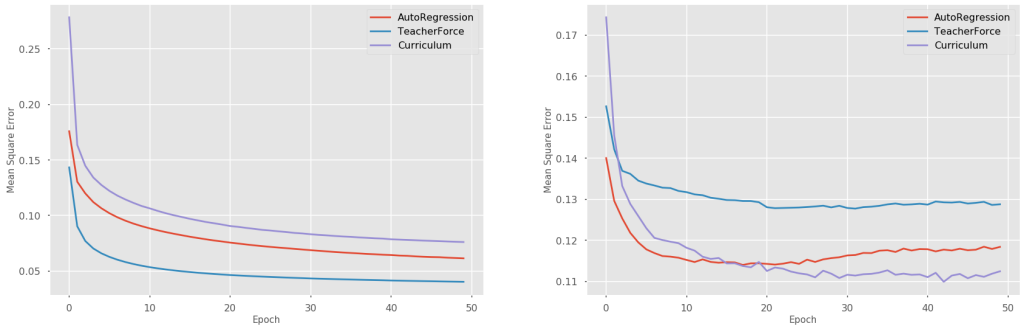


(b) Validation Error

Figure 5.10: Teacher Force vs Auto Regressive Training

5.4.4.1 Curriculum Learning

With *Curriculum learning*, we observe network underperform in comparison with TTF and outperform in comparison to AR during training. As expected, it outperformed against both of the training strategies during evaluation. We repeated the test such that the model starts from an autoregressive setting and slowly decays down to the TTF setting and observed a similar performance.



(a) Training Error (b) Validation Error

Figure 5.11: Curriculum Training

Results for each subsampling technique along with time taken is shown in the Table. 5.3,

	Training Error			Validation Error			Training Time
	MSE	MAE	RMSE	MSE	MAE	RMSE	Epoch/sec
Teacher Force	**0.0404**	**0.1291**	**0.2007**	0.1277	0.1906	0.3541	102.21s
Auto Regression	0.0615	0.1502	0.2473	0.114	0.1898	0.3351	99.55s
Curriculum	0.0762	0.1672	0.2749	0.1099	0.1808	0.3289	163.4s

Table 5.3: Results for Experiment : AutoRegression

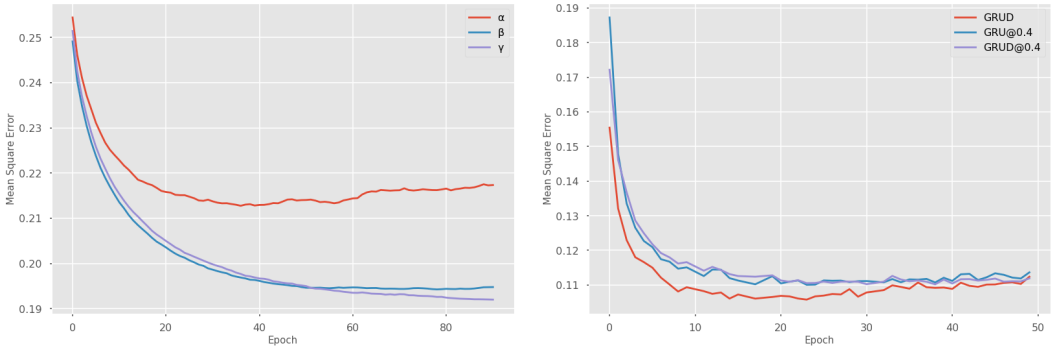
The model in teacher force and autoregression configuration is the fastest in terms of computation. While all of the other models have similar performance, this performance penalty is mainly due to additional conditions required to switch between AutoRegression and Teacher Force training Configuration.

5.4.5 Missing Data

We used SP-DT dataset for the initial testing on missing data configuration on the LSTM model. We observed a considerable performance improvement in the validation set.

5.4.5.1 Lesson 2

To test the GRUD model's performance on different sparsity of data. We designed a random function generator that sets the sparsity of the batch data based on hyper-parameter. We observed an improvement in performance on the validation set when compared to the original data. Ideally, it should have been the opposite. We repeated the experiment on the METR-LA dataset to cross-verify this finding and observed the same conclusions, as shown in Fig. 5.12.



(a) SP-DT Dataset: The sparsity of γ is higher than that of β, and α. Performance of GRU-D model is analysed over three different level of sparsity.

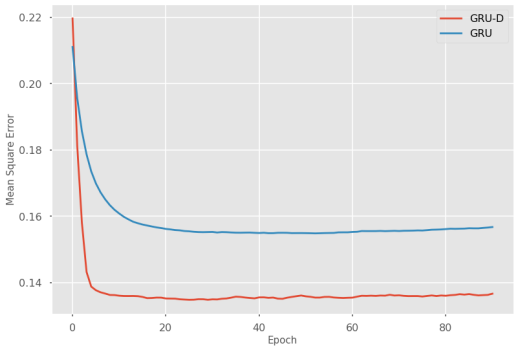
(b) METR-LA Dataset: Sparsity is set to 60% for both models. For comparison, GRUD is trained without any sparsity imputation.

Figure 5.12: Mean Square Error during Validation

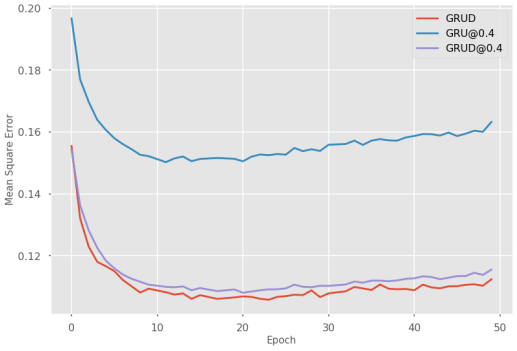
Initially, we hypothesized that for both the dataset, missing data has been acting as a regularization mechanism to improve the generalization capability. But upon further studies, we found the bug. We were masking the data independently for every batch, and as the batch shares the same sequence with other batches at different time steps. Independent batch introduces a similar effect as of dropout because the probability of never seeing input data input features is reduced.

When we preprocessed the training dataset before stacking the entries over time sequence length, we observed it underperforms as illustrated by validation loss in Fig. 5.13.

Thus, we observe that GRU-D effectively handles the missing data using a learnable parameter that controls the amount of dependence on previous and average feature values.

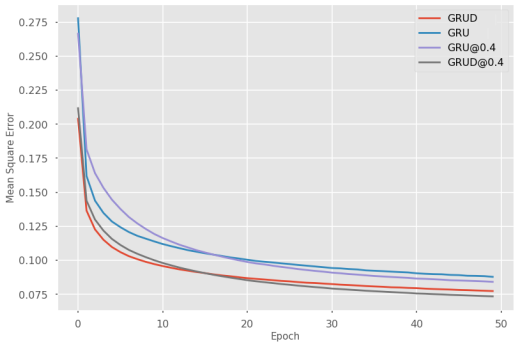


(a) SP-DT Dataset: Disabled Independent Batch Sampling

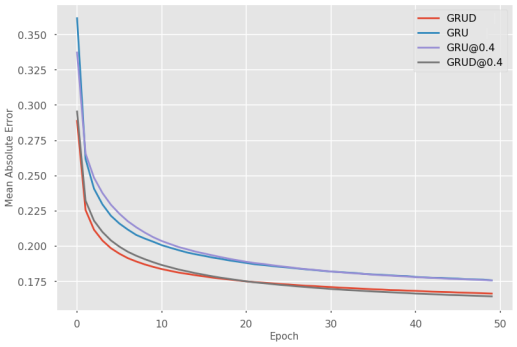


(b) METR-LA Dataset: Legend with @ implies the percentage of sparsity

Figure 5.13: Corrected: Validation Error



(a) MSE Error



(b) MAE Error

Figure 5.14: Corrected: Training Error on METR-LA Dataset

Results for GRU-D vs GRU for SP-DT and METR-LA dataset is shown in Table. 5.4,

Dataset	**Algorithm**	Training Error			Validation Error		
		MSE	MAE	RMSE	MSE	MAE	RMSE
SP-DT	GRU	0.1385	0.2542	-	0.1542	0.2667	-
	GRU-D	0.1225	0.245	-	0.1333	0.2543	-
METR-LA	GRU	0.0841	0.1758	0.2889	0.1502	0.2219	0.3846
	GRU-D	0.0735	0.1645	0.2701	0.108	0.1863	0.3262

Table 5.4: Results for Experiment : Missing Data

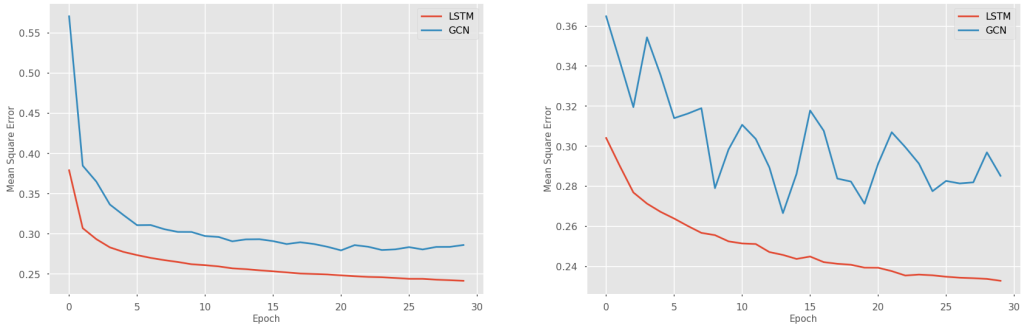
5.5 Extension to ETA-DT dataset

After observing the behavior of different spatial-temporal models on the METR-LA dataset and missing data on SP-DT dataset. We aim to implement a robust and scalable model on our ETA-DT dataset. It is not just large compared to the METR-

LA dataset in terms of vertices but is more sparse and thus introduces the effect of missing data on the optimization of the network.

5.5.1 Choosing the Architecture

Similar to METR-LA, we first trained the LSTM model on our dataset and compared its performance with a similarly configured GCN model with sub-sampling. Ideally, GCN should have outperformed LSTM without any issues, but we observed LSTM beat GCN with a considerable margin (Fig. 5.15). To ensure we have not made any programming error, we compared the performance with Mean Model, and GCN was outperforming the Mean model without any hiccups.



(a) MSE Training Error (b) MSE Validation Error

Figure 5.15: LSTM vs GCN

We could not understand the reason behind this behavior initially, primarily because of performance on validation data. Thus, to iron out any errors in the data modeling, we added more assertions. Still, the behavior did not change, highlighting some mistakes during the training phase.

The reason behind our belief on the training anomaly is because LSTM with $1/10^{th}$ of the parameters outperformed. This is when we found the first mistake; in some forums, Adam [35] optimizer was discouraged when working with the missing data, and upon studying more in this direction, we observe that Adam optimizer

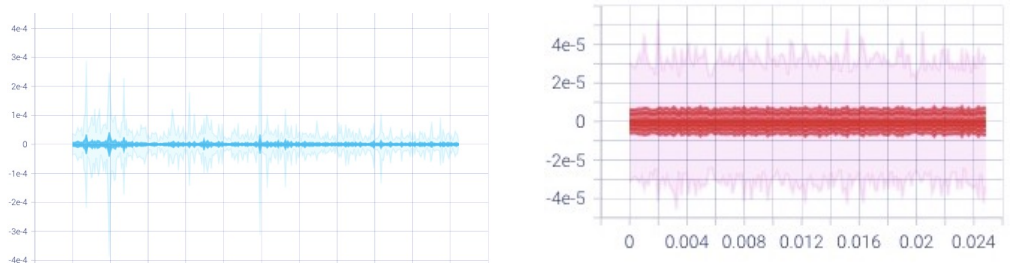
updates the gradients based on,

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
m &= \frac{m_t}{1-\beta_1^t} \\
v &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{v}+\epsilon} m
\end{aligned}
\tag{5.11}
$$

If we observe the update equation, the value in the denominator is controlled w.r.t. to the gradients and β_2 constant, which is different from β_1, which can set the denominator to be zero. At the same time, the numerator is non-zero and thereby very large gradient updates. As soon as we switched to the Adagrad [36], the spikes in the gradients stop appearing (Fig. 5.16) because in Adagrad v is replaced with L_2 norm of gradients(G_t) for each parameter.

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t}+\epsilon} g_t \tag{5.12}
$$

where g_t is the gradient at time t.

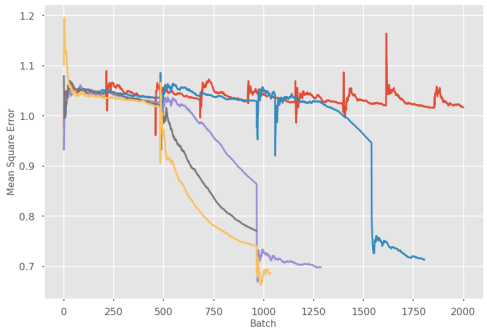


(a) Using Adam Optimizer (b) Using Adagrad Optimizer

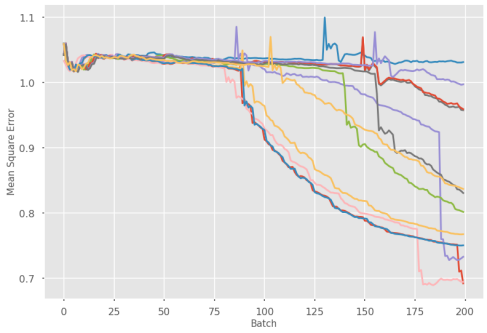
Figure 5.16: Gradients for LSTM Kernel

After switching to Adagrad optimizer, LSTM starts saturating at a loss of 1.0 for a few epochs but then suddenly begins training and again reaches the result we observed with Adam optimizer. The performance of the model varied a lot with the change in ϵ (Fig. 5.17).

We performed one final experiment such that the parameters for the network are



(a) Different Learning Rates



(b) All of the experiments are done in same setting; in some experiments the value of ϵ is 0.01 and in other 0.011

Figure 5.17: Batch Loss on different Value of epsilon

the same, except ϵ. In some of the experiments, the value of epsilon is 0.01, and in some, it is 0.011. The loss curves have a very distinct boundary as the models with 0.01 epsilon decaying earlier when compared with 0.011. We believe it is due to the missing data present in the data, leading to sparsity in the parameter and gradient explosion. The effect of gradient explosion is more on LSTM when compared with the rest because all the other networks share parameters across vertices while LSTM does not. We also observed that all of the states of the art algorithms like T-GCN [20], DCRNN uses FC-LSTM [13] for comparison.

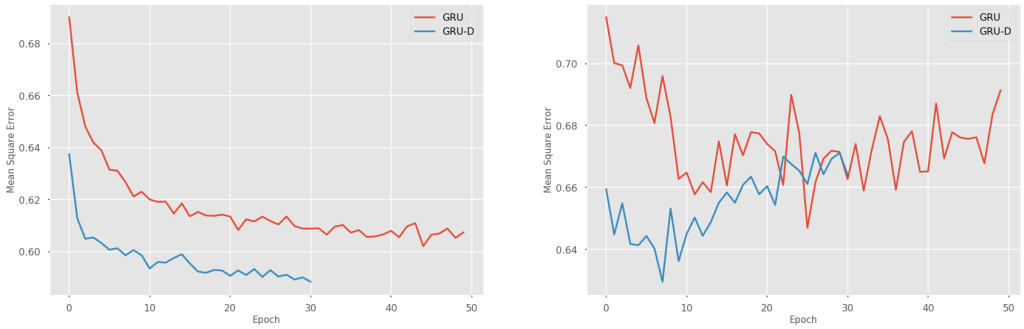
Results for each model, along with the time taken are shown in Table. 5.5. All models were trained for at max 50 epochs, with early stopping enabled.

Algorithm	Training Error				Validation Error				Time Taken
	MSE	MAE	MRE	RMSE	MSE	MAE	MRE	RMSE	Epoch/sec
FC-LSTM	0.7292	0.4492	3.0636	0.8527	0.7291	0.4476	3.0677	0.8527	135.8s
DCRNN	0.6499	0.4015	**2.8141**	0.8038	**0.6502**	**0.4022**	2.8338	**0.8041**	949.7s
SSTG	**0.6061**	**0.3916**	2.9014	**0.7751**	0.6726	0.4211	**2.5243**	0.8164	297s

Table 5.5: Results for Experiment: ETA-DT Dataset Integration

5.5.2 Integrating the Missing Data

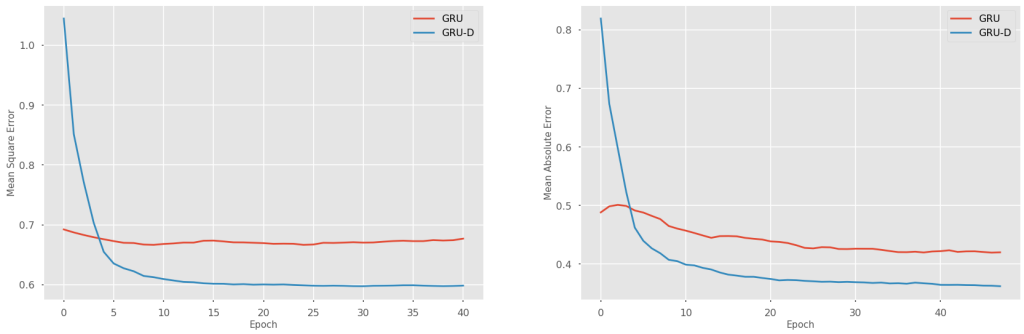
The initial testing of missing data on ETA prediction was done on a simpler model, SGC(5.4.1), and observed improvement in terms of training loss, but the updated model overfitted and had performance drop in terms of the validation loss (Fig. 5.18).



(a) Training Error (b) Validation Error

Figure 5.18: SGC: Overfitting during Validation

It was a great sign, as it highlights that the network is saturating. We then switched to our proposed network, SSTG. We observed a better-generalized performance across the parameters, as shown in Fig. 5.19).



(a) Validation MSE (b) Validation MAE

Figure 5.19: SSTG-GRUD: Performance

Results for each model, along with the time taken are shown in table 5.6. All the models were trained for at max 50 epochs, with early stopping if required.

Algorithm	Training Error				Validation Error				Time Taken
	MSE	MAE	MRE	RMSE	MSE	MAE	MRE	RMSE	Epoch/sec
SSTG-GRU	0.6061	0.3916	2.9014	0.7751	0.6726	0.4211	**2.5243**	0.8164	297s
SGC-GRUD	0.5885	0.3776	2.8909	0.7641	0.6295	0.4041	2.9717	0.7901	199s
SSTG-GRUD	0.475	0.3304	2.5778	0.6876	**0.5954**	**0.364**	2.7649	**0.7696**	297s

Table 5.6: Results for Experiment: Missing Entry in ETA-DT Dataset

SSTG-GRUD is trained with the **curriculum** training process, while others have been with AutoRegression.

5.5.3 Relative Loss Function

MSE loss function penalizes all of the entries equally whether the ground truth is 10 and prediction is 9.5 or ground truth is 1 and prediction is 0.5. The error for both outputs is the same as 0.25, but their percentage error is 0.05 and 0.5, respectively. Also, the MSE loss penalizes the output equally irrespective of greater or less than the ground truth.

Thus, we switched the loss function to $MSLE$, which aims at computing the relative error. We observed that the model has a lower overestimation percentage as compared to MSE. However, it outperformed in all other metrics—with the most significant improvement in terms of MRE error, highlighting that relative loss function improves the model's overall functioning, and the reduction in over-estimation percentage is because of improvement in generalization. Also, with MSLE loss, our model performs desirably in TTF and AR autoregressive configuration enabling better performance with curriculum learning.

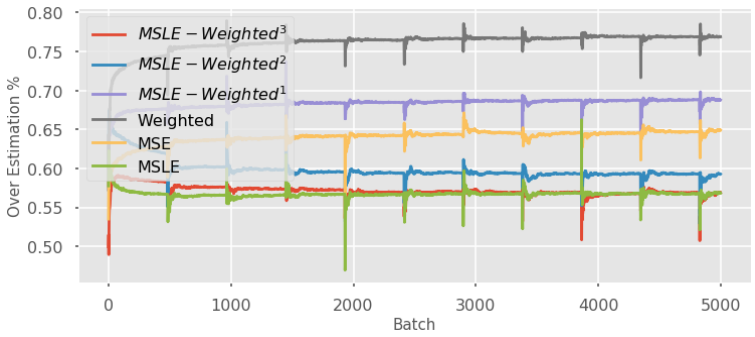


Figure 5.20: Overestimation Percentage

After, that we tried weighted loss and observed improvement in the overestimation percentage (Fig. 5.20), but it comes at the cost of the other metrics.

Our final model uses MSLE-Weight loss, 1) to increase the penalty on the underestimation results, and 2) to ensure relative error minimization. We observe a better overestimation percentage compared to MSLE and less performance penalty compared to Weighted loss.

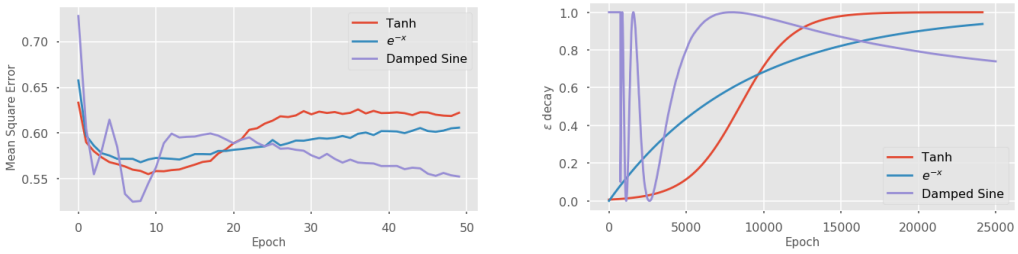
Results for all of the models has been shown in Table. 5.7,

Algorithm	**TTF/AR**	Training Error					Validation Error				
		MSE	MAE	MRE	RMSE	OE%	MSE	MAE	MRE	RMSE	OE%
MSE	AR	0.5822	0.3731	2.8463	0.7612	65.51	0.6307	0.4294	3.3009	0.7927	64.4
MSLE	AR	0.6073	0.361	2.5615	0.7769	57.17	0.6117	0.3642	2.5637	0.7803	55.61
MSLE	TTF	0.5344	0.3454	2.4732	0.729	55.45	0.6527	0.3846	2.7021	0.806	48.13
Weighted*	TTF	0.613	0.4302	3.6458	0.7816	77.19	0.6177	0.4273	3.4976	0.7842	70.9
MSLE-Weighted[1]	TTF	0.5965	0.3893	3.0368	0.7706	69.47	0.6154	0.403	2.8732	0.7827	70.01
MSLE-Weighted[2]	Curr	0.6205	0.3689	2.6008	0.7859	58.29	0.6287	0.3966	2.6333	0.7914	66.19
MSLE-Weighted[3]	Curr	0.6035	0.3673	2.65	0.7752	60.19	0.6187	0.369	2.7331	0.785	55.45

Table 5.7: Results for Experiment: Over-Estimation on SSTG-GRUD model

Training the naive SSTG model in GRUD configuration is difficult when AutoRegression training processed is used; for the same reason, curriculum learning was used. Weighted* was early stopped, and MSLE-Weighted[1], MSLE-Weighted[2], MSLE-Weighted[3] have different 2, 1.5, and 1.2 weight factors, respectively. Also, MSLE-Weighted[2] only penalizes if the bus is more than a minute early.

It is crucial to utilize the curriculum learning as the network is susceptible to the input distribution as shown by the training loss tracking the decay curve (Fig. 5.21). Thus with the help of curriculum learning, we can ensure that the input during the evaluation will not be entirely unexpected for the network.



(a) Training Error (b) Decay Curve

Figure 5.21: SSTG-GRUD: Training Error vs TTR Decay

The SSTG-GRUD, with the help of MSLE-weighted loss function, is easily tunable to the specific needs of over-estimation and performance.

5.6 Results

In this section, we have compiled all the significant models that we used in experimentation.

The results on METR-LA data has been shown in Table. 5.8,

	Training Error				Validation Error				Training Time
	MSE	MAE	MRE	RMSE	MSE	MAE	MRE	RMSE	Epoch/sec
LSTM	0.0811	0.1874	1.1571	0.2829	0.1433	0.2319	1.2341	0.3757	121.1s
ConvLSTM	0.1121	0.1906	1.137	0.3327	0.1094	0.1860	0.9985	0.3283	153.5s
DCRNN	0.0461	0.1309	0.9726	0.2143	0.0905	0.1612	0.9967	0.2986	231.4s
SGC	0.0828	0.1665	1.1006	0.2861	0.1044	0.1712	0.9933	0.3201	112.74s
Node Sampling	0.1121	0.1906	1.137	0.3327	0.1094	0.1860	0.9985	0.3283	153.5s
RWS Sampling: ϕ	0.0689	0.1490	1.0496	0.2613	0.0916	0.1607	0.9647	0.3006	172.79s
RWS Sampling: $Norm$	0.08215	0.1610	1.0770	0.2849	0.1023	0.1743	1.0017	0.3175	176.49s
RWS Sampling: $Lapl$	0.06770	0.14806	1.0476	0.2588	0.0903	0.1613	0.9764	0.2977	185.66
RWS Sampling: $Lapl$@0.6	0.0841	0.1758	-	0.2889	0.1502	0.2219	-	0.3846	”
RWS GRUD: $Lapl$@0.6	0.0735	0.1645	-	0.2701	0.108	0.1863	-	0.3262	”
SSTG@0.6	0.0829	0.1616	1.8842	0.2863	0.1027	0.172	1.8378	0.3178	-

Table 5.8: Results: METR-LA Dataset

In all of the experiments, the latent feature size of Encoder and Decoder was constant, 64. We used two layers in both encoder and decoder. The initial configuration was referred from the implementation of DCRNN. The ConvLSTM model underperformed when compared with LSTM because the vertices are modeled independently in ConvLSTM. While, during validation ConvLSTM outperforms LSTM as it is more regularized due to parameter sharing. SGC outperformed DCRNN, especially in terms of MAE error, highlighting that using residual structure improves the overall learning, and modifying GRU to compute the graph convolution is not required in every case. The RWS Sampling and RWS Sampling:$Lapl$ performed similarly, but in SSTG, we compute Laplacian at every batch because it is the correct way to process the sub-graph. The experiments marked with RWS Sampling and SSTG are the same, except RWS Sampling does not utilize the residual connection, while SSTG does. We did not perform intermediate experiments on SSTG; because we were parallelly experimenting on sub-sampling and simpler methods for graph convolution. The last three experiments were done to highlight the effect of missing data; here, 0.6 implies 60% of the data is available. Our proposed framework, SSTG, outperformed RWS Sampling by a considerable margin, and it has a very marginal improvement compared with the GRUD version of RWS Sampling.

The results on ETA-DT data has been shown in Table. 5.9,

	Training Error				Validation Error				Training Time
	MSE	MAE	MRE	RMSE	MSE	MAE	MRE	RMSE	Epoch/sec
FC-LSTM	0.7292	0.4492	3.0636	0.8527	0.7291	0.4476	3.0677	0.8527	135.8s
DCRNN	0.6499	0.4015	2.8141	0.8038	0.6502	0.4022	2.8338	0.8041	949.7s
SSTG-GRU	0.6061	0.3916	2.9014	0.7751	0.6726	0.4211	2.5243	0.8164	297s
SGC-GRUD	0.5885	0.3776	2.8909	0.7641	0.6295	0.4041	2.9717	0.7901	199s
SSTG-GRUD	0.475	0.3304	2.5778	0.6876	0.5954	0.364	2.7649	0.7696	297s

Table 5.9: Results: ETA-DT Dataset

Loss fn	Policy	Training Error					Validation Error				
		MSE	MAE	MRE	RMSE	OE%	MSE	MAE	MRE	RMSE	OE%
MSE	AR	0.5822	0.3731	2.8463	0.7612	65.51	0.6307	0.4294	3.3009	0.793	64.4
MSLE	AR	0.6073	0.361	2.5615	0.7769	57.17	0.6117	0.3642	2.5637	0.780	55.61
MSLE	TTF	0.5344	0.3454	2.4732	0.729	55.45	0.6527	0.3846	2.7021	0.806	48.13
MSLE-W^1	TTF	0.5965	0.3893	3.0368	0.7706	69.47	0.6154	0.403	2.8732	0.783	70.01
MSLE-W^2	Curr	0.6205	0.3689	2.6008	0.7859	58.29	0.6287	0.3966	2.6333	0.791	66.19
MSLE-W^3	Curr	0.6035	0.3673	2.65	0.7752	60.19	0.6187	0.369	2.7331	0.785	55.45

Table 5.10: Results: Over-Estimation on ETA-DT Dataset

As with experiments on the METR-LA dataset, we used a two-layer encoder and decoder but 128 hidden units instead of 64. We also performed tests with 256 and 512 units and observed marginal improvement. DCRNN outperforms FC-LSTM (as the vertex size is very high, ConvLSTM would have required 5x more layers to spatially cover based on ordering provided by adjacency matrix) but is computationally expensive. SSTG-GRU outperformed DCRNN and takes less than $1/3^{rd}$ amount of time. To handle the missing data, we switched to the GRUD version. The final SSTG-GRUD has been trained with Curriculum Learning policy, and for that reason, there is a massive difference in the training error. The simple SGC network overfits when merging with GRUD. The experiment in the Table. 5.10 highlights the effect of over-estimation, and for the same reason, we have an additional metric, Over-estimation percentage ($OE\%$). We have tested different policy to maximize the performance on the validation dataset. When the MSLE loss function is used, our model improves in all of the metrics except the Over-estimation percentage. We believe the improvement in generalization could be behind the dip. After that we tried the weighted loss function, which improved the Overestimation percentage but performed poorly in other metrics (that's why it stopped early). The last three experiments were performed using the MSLE-weighted loss function with different weight factors, enabling us to finetune the penalty on estimation vs. Over-estimation per-

centage. The first experiment uses factor of 2, while the the second experiment uses the factor of 1.6 and we only penalize if the under-estimation is more than 1min. In the last experiment, we have used a very small factor of 1.2 only.

The results on ETA-Dataset while evaluating different strategies of sampling are shown in Table:5.11,

Strategy	Sample Size	Algorithm	MSE	MAE	RMSE	$OE\%$	Epoch/sec
Complete		Mean	3.92m	62.504s	118.8s	48.53	0.05s
		FC-LSTM	2.99m	54.671s	103.6s	68.24	2.1s
		DCRNN	2.645m	48.689s	97.3s	67.08	10.72s
		SSTG	2.651m	49.142s	97.6s	70.18	16.5s
RWS Sampling	512	SSTG	2.511m	48.529s	94.9s	70.16	4.8s
	256	SSTG	2.551m	49.007s	95.6s	69.75	3.86s
Predefined@1	All	Mean	3.832m	69.307s	117.5s	51.81	0.05s
	All	FC-LSTM	3.701m	58.43s	115s	66.22	0.12s
	All	DCRNN	2.357m	45.208s	91.3s	66.6	10.72s
	32	SSTG	2.507m	46.544s	78.3s	69.14	0.23s
	256	SSTG	2.503m	46.629s	78.3s	69.84	1.95s
Predefined@2	All	Mean	3.089m	55.933s	105.5s	47.59	0.05s
	All	FC-LSTM	2.398m	50.246s	92.5s	71.58	0.12s
	All	DCRNN	2.262m	48.934s	89.5s	73.26	10.72s
	16	SSTG	2.227m	45.661s	72.9s	72.58	0.13s
	64	SSTG	2.226m	45.918s	73s	73.01	0.41s

Table 5.11: Results: ETA-DT Test Dataset

There are three evaluation strategies; in the first block, we have discussed how our proposed framework performs when evaluating the results on the complete graph. When evaluating on the complete data, the proposed algorithm has a mean square error of 2.635m as compare to 3.93m for the mean model and 2.64m for the DCRNN. DCRNN out performs the proposed algorithm in other metrics too; but our algorithm was trained with a penalty in the loss to increase over-estimation (4% higher) and is also trained on 25% of the data. In the predefined vertex evaluation strategy, we sampled out a fixed node and estimated the output for those; we repeated these experiments for two types of vertices, 1) Predefined@1 in which average neighbors are more than 100, and 2) Predefined@2 in which the average neighbors are between 10-20. Both of the experiments are averaged over 25 trials. Our proposed algorithm outperforms DCRNN when tested on vertices with moderate connectivity, while it underperformed when the vertices are very highly connected. We observe that the performance of our proposed model is very similar when compared with DCRNN, while taking only 4% time during evaluation.

Chapter 6

Conclusion and Future Work

The problem of the ETA prediction plays a very important role in improving the rider's experience. While ensuring timeliness is challenging, especially during rush hours. It is possible to provide riders with a heads-up on the amount of time taken by the bus to reach their location. A stream of GTFS data enables the estimation of the arrival time for the future time steps. In this thesis, we developed an algorithm to model the raw GTFS data into Spatio-Temporal tensors, which are further used by our proposed algorithm, SSTG, based on recurrent neural networks and graph convolutional networks future prediction. The proposed model captures the Spatio-temporal structure defined by the adjacency matrix of the data. To incorporate the absence of data for certain vertices at times, we modify the first layer to impute the data dynamically.

We obtain better results for arrival time estimation with our proposed algorithm when compared to the state-of-the-art technique. We also evaluated our algorithms on an additional traffic speed dataset, METR-LA. The two points, which we would look to develop in the future, 1) dynamically learn the adjacency matrix by using single-attention for graph modeling and temporal prediction, and 2) would be to develop techniques that can help us define the graph size for a pre-defined vertex during evaluation.

Bibliography

[1] Tom Tom Index. New delhi traffic report.

[2] Neema Davis, Harry Joseph, Gaurav Raina, and Krishna Jagannathan. Congestion costs incurred on indian roads: A case study for new delhi. 08 2017.

[3] Hemant K. Suman, Nomesh B. Bolia, and Geetam Tiwari. Comparing public bus transport service attributes in delhi and mumbai: Policy implications for improving bus services in delhi. *Transport Policy*, 56:63–74, 2017.

[4] V. Chauhan, H. K. Suman, and N. Bolia. Binary logistic model for estimation of mode shift into delhi metro. *The Open Transportation Journal*, 10:124–136, 2016.

[5] Open Transit Data Delhi. https://opendata.iiitd.edu.in/.

[6] Lei Tang and Piyushimita (Vonu) Thakuriah. Ridership effects of real-time bus information system: A case study in the city of chicago. *Transportation Research Part C: Emerging Technologies*, 22:146–161, 2012.

[7] Billy M. Williams and Lester A. Hoel. Modeling and forecasting vehicular traffic flow as a seasonal arima process: Theoretical basis and empirical results. *Journal of Transportation Engineering*, 129(6):664–672, 2003.

[8] Noel Cressie and Christopher K Wikle. Space-time kalman filter. *Encyclopedia of environmetrics*, 4:2045–2049, 2002.

[9] Hao Liu, Henk Van Zuylen, Hans Van Lint, and Maria Salomons. Predicting urban arterial travel time with state-space neural networks and kalman filters. *Transportation Research Record*, 1968(1):99–108, 2006.

[10] Hao Liu, Hans van Lint, Henk van Zuylen, and Ke Zhang. Two distinct ways of using kalman filters to predict urban arterial travel time. In *2006 IEEE Intelligent Transportation Systems Conference*, pages 845–850, 2006.

[11] Bin Yu, Zhong-Zhen Yang, Kang Chen, and Bo Yu. Hybrid model for prediction of bus arrival times at next station. *Journal of Advanced Transportation*, 44(3):193–204, 2010.

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[13] Alex Graves. Generating sequences with recurrent neural networks, 2014.

[14] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai kin Wong, and Wang chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting, 2015.

[15] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[16] David K Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory, 2009.

[17] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6861–6871. PMLR, 09–15 Jun 2019.

[18] Charul and Pravesh Biyani. To each route its own eta: A generative modeling framework for eta prediction, 2019.

[19] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Graph convolutional recurrent neural network: Data-driven traffic forecasting. *CoRR*, abs/1707.01926, 2017.

[20] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-gcn: A temporal graph convolutional network for

traffic prediction. *IEEE Transactions on Intelligent Transportation Systems*, 21(9):3848–3858, Sep 2020.

[21] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[22] General Transit Feed Specification.

[23] M. Mozer. A focused backpropagation algorithm for temporal pattern recognition. *Complex Syst.*, 3, 1989.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[25] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[26] Prof. J. R. Culham. Chebyshev polynomials. Accessed: 15-05-2021, Reading Material on Chebyshev Polynomials.

[27] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020.

[28] Jiyang Bai, Yuxiang Ren, and Jiawei Zhang. Ripple walk training: A subgraph-based training framework for large and deep graph neural network. *CoRR*, abs/2002.07206, 2020.

[29] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[30] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks, 2015.

[31] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David A. Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. *CoRR*, abs/1606.01865, 2016.

[32] John Laird. The law of parsimony. *The Monist*, 29(3):321–344, 1919.

[33] Jiawei Zhang and Lin Meng. Gresnet: Graph residual network for reviving deep gnns from suspended animation. *CoRR*, abs/1909.05729, 2019.

[34] S. Kokoska and D. Zwillinger. Crc standard probability and statistics tables and formulae, student edition. 1999.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[36] Agnes Lydia and Sagayaraj Francis. Adagrad - an optimizer for stochastic gradient descent. Volume 6:566–568, 05 2019.

[37] Alex Lamb, Anirudh Goyal, Ying Zhang, Saizheng Zhang, Aaron Courville, and Yoshua Bengio. Professor forcing: A new algorithm for training recurrent networks. *arXiv preprint arXiv:1610.09038*, 2016.

Appendix A

Appendix

A.1 Induction Proof

Induction Proof: $UT_i(\lambda')U^T = T_i(L')$

$\Diamond$ *Base Case*: $T_0(\lambda') = 1 \implies UT_0(\lambda')U^T = UU^T = 1$

$\implies UT_0(\lambda')U^T = T_0(L')$

$\Diamond$ *Induction Step(s)*: Assumption for $T_s(L') = UT_s(\lambda')U^T$,

$\Diamond$ *Proof for Step(s+1)*:

$$T_{s+1}(L') = L'T_s(L') - T_{s-1}(L') \tag{A.1}$$

$$\because \text{Eq. 3.18} \tag{A.2}$$

$$T_{s+1}(L') = L'UT_s(\lambda')U^T - UT_{s-1}(\lambda')U^T \tag{A.3}$$

$$\because \text{Induction Assumption} \tag{A.4}$$

$$T_{s+1}(L') = U\lambda'(U^TU)T_s(\lambda')U^T - UT_{s-1}(\lambda')U^T \tag{A.5}$$

$$T_{s+1}(L') = U(\lambda'T_s(\lambda') - T_{s-1}(\lambda'))U^T \tag{A.6}$$

$$\because \text{Distributive property} \tag{A.7}$$

$$T_{s+1}(L') = UT_{s+1}(\lambda')U^T \tag{A.8}$$

$$\text{Q.E.D} \tag{A.9}$$

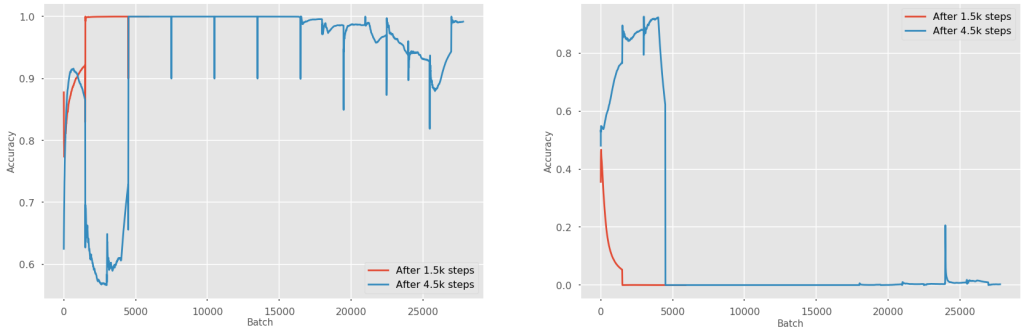
A.2 Further Study on AutoRegressive Training

A.2.1 Professor Training

In Professor Training [37], The decoder generates two outputs; the first has ground truth as the input D_f, and the second has output from the previous steps D_s. It adds another neural network, known as a discriminator, distinguishing D_f from D_s. D_f is marked as the correct class in the discriminator cycle, while D_s is marked

during the generator cycle. During each cycle, only the gradients of the respective modules are updated. The generator's parameters are only updated when the discriminator is confident well trained (above 75% accuracy) but not too successful (below 99% accuracy). It is done because the distribution of real-input (D_f) depends on the decoder output, and Professor training does not pre-train the network in teacher-force configuration.

After multiple attempts, we could not get it to work. We are aware that adversarial domain adaption is difficult but, even after setting all of the parameters correctly and following the methodology, it did not work. We also tried pre-training it initially. Still, as the pretraining stops, the generator updates the distribution of the decoder such that the accuracy of TTF decoded becomes 1. In contrast, that of AR decoded gets reduced down to 0. (Fig. A.1).



(a) Discriminator Loss for TTF Decoded (b) Discriminator Loss for AR Decoded

Figure A.1: Professor Training: Legend represents the step at which pre-training stopped

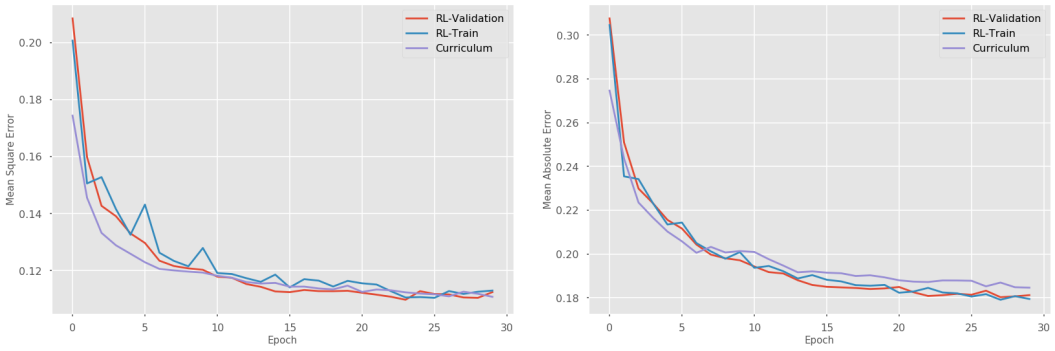
A.2.1.1 RL-Validation

The professor training requires an extra discriminator and parallel training in both configurations for adaptive training. To simplify the process, we implemented a Q learning algorithm to control the value of ϵ, with the state space, $S \in [0, 1]$, possible actions as $\{+0.01, -0.01\}$ increasing or decreasing the value of ϵ, and the reward is the average of the relative difference between the training and validation loss. The update equation for Q learning is as follows (Eq. A.2.1.1),

$$
\begin{aligned}
a &= argmax(Q[\epsilon]) \\
\epsilon' &= \begin{cases} \epsilon + 0.01 & if a = 1, \\ \epsilon - 0.01 & otherwise \end{cases} \\
r &= \frac{|mean(L_{train}) - L_{validation}|}{mean(L_{train})} \\
Q(\epsilon, a) &= Q(\epsilon, a) + \alpha(r + \gamma max(Q(\epsilon'))) \qquad \text{(A.10)}
\end{aligned}
$$

where Q is a matrix, α is the learning rate, and γ is the discount factor.

The training from RL-Validation does not just improve the overall performance of the model, but the learned ϵ threshold is adaptive, and the resultant model has better MAE as compared to curriculum training, with similar mean square error, as shown in Fig. A.2.



(a) MSE Validation (b) MAE Validation

Figure A.2: RL-Validation Performance

RL-Validation introduces two issues to the training process. The optimization of the Q learning parameter only happens during validation(Fig. A.3). Thus, the adaption is slow and requires more epochs for finetuning of the Q learning agent. Also, the model's performance on the validation set impacts the threshold behavior, introducing the posterior effect of performance on Validation data on the model, which is not correct.

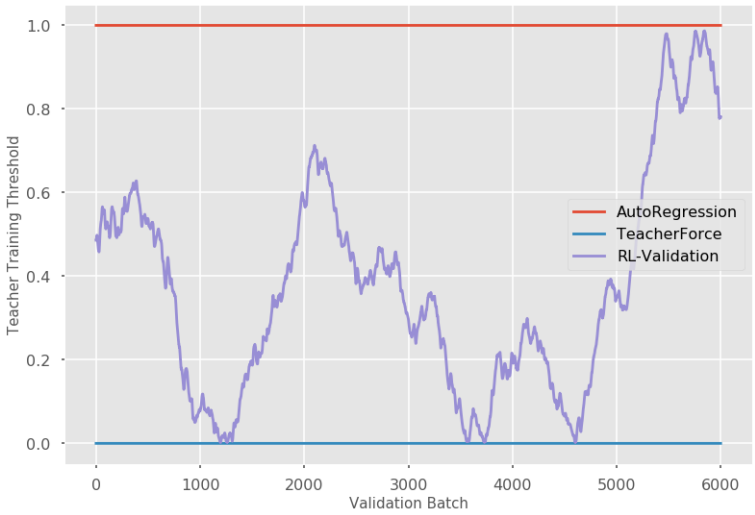


Figure A.3: ϵ value for different models

A.2.2 RL-Training

We implemented another Q learning that updates its parameter at every batch and only utilizes the training data to tackle both issues. We divide each batch into mini-batches such that each batch consists of 3 training samples; we train the first sample in TTF fashion, second in an AR fashion and third wrt to the learned TTR threshold.

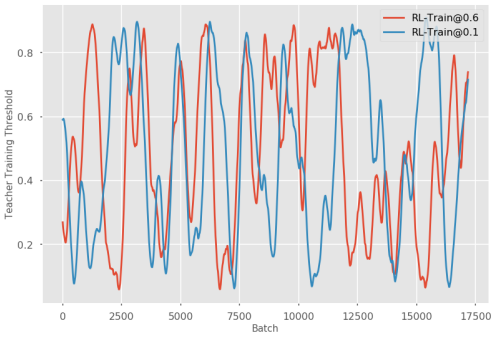
Updations of the Q learning happens after each mini batch, with reward as

$$r' = \begin{cases} 1 & if\ |\dfrac{L_{TTF} - L_\epsilon}{L_{AR} - L_\epsilon}| < 1.25\ \&|L_{TTF} - L_\epsilon| > |L_{AR} - L_\epsilon| \\ 1 & if\ |\dfrac{L_{AR} - L_\epsilon}{L_{TTF} - L_\epsilon}| < 1.25\ \&|L_{AR} - L_\epsilon| > |L_{TTF} - L_\epsilon| \\ 0 & otherwise \end{cases} \tag{A.11}$$

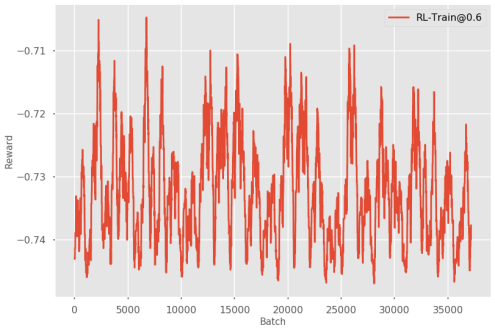
Reward, r' ensures that loss incurred through the threshold is the approximate middle of loss due to AutoRegression and Teacher Force. It is done because an optimal AutoRegression agent would have less loss when compared to noisy outputs and more loss than ground truth during training.

We observed RL-Train (Fig. A.4) behaved in a sigmoidal fashion, i.e., during certain phases, it switched to TTF. During certain stages, it switched to AR, regularizing the learning process.

But upon further analysis, we observed it to have minimal impact on the overall



(a) Threshold for RL-Train using 0.6 and 0.1 learning rate



(b) Reward obtained by the Q learning

Figure A.4: TTR Decay of a different algorithm, value of 1 implies autoregression, 0 implies teacher force

learning of the algorithm as reward obtained during multiple trials never increased. It acted as a linear control algorithm, i.e., as soon as the error between AR and TTF setting increased, i.e., regularized the model by increasing the teacher training force, which increased the error between TTF and ϵ strategy prompting it to decrease the threshold.

	Training Error			Validation Error			Training Time
	MSE	MAE	RMSE	MSE	MAE	RMSE	Epoch/sec
Teacher Force	**0.0404**	**0.1291**	**0.2007**	0.1277	0.1906	0.3541	102.21s
Auto Regression	0.0615	0.1502	0.2473	0.114	0.1898	0.3351	99.55s
RL-Validation	0.0843	0.175	0.2891	0.1098	0.1803	0.3287	157.37s
RL-Train@0.1	0.0741	0.1652	0.2711	**0.1075**	0.1755	**0.3251**	160.97s
RL-Train@0.6	0.0739	0.1656	0.2708	0.1093	**0.1753**	0.328	164.41s
Professor Training	-	-	-	-	-	-	-
Curriculum	0.0762	0.1672	0.2749	0.1099	0.1808	0.3289	163.4s

Table A.1: Results for Experiment : AutoRegression