Project Report

50.039 Theory and Practice of Deep Learning

Full Name	**Student ID**
Darren Chan Yu Hao	1006340
Isaac Koh Jun Wei	1005998
Teng Shin Shoon, Nicholas	1003416

Table of Contents

1. Abstract

In urban environments like Singapore, timely access to taxi services plays a huge part in ensuring commuter satisfaction and travel efficiency. This project seeks to develop a predictive model that forecasts taxi availability in a specific location and time, using a variety of curated and compiled data sources ranging from historical taxi movement data, weather conditions, and additional factors such as public holidays and weekends.

We approached this problem by exploring and training a wide range of models from simple vanilla LSTMs to more complex Bi-directional Encoder Decoder LSTMs so as to better understand not only the complexity of the problem but also to ensure that the problem is not tackled with an overly complex and complicated model.

2. Introduction

Many passengers struggle with finding taxis during off-peak hours or in certain locations, especially late at night or early in the morning, leading to uncertainty and frustration when they are unable to arrive at their destination on time. Without reliable insights into when and where taxis are most available, commuters are left guessing when the best time is to call for taxis, resulting in long waiting times, customer dissatisfaction and overall inconvenience.

Our project aims to combat this by using data-driven predictions to develop a model that predicts ahead of time the number of taxis within a certain location while factoring in major variables that can affect the availability like weather, time of day, day of the week, holidays and date. This proactive approach will allow passengers the confidence to plan their day around when taxi is the most available. This prediction can even further allow users to plan and prevent potential surge charges due to booking when there is high demand and low taxi availability, improving overall commuting convenience.

To accomplish the goals, we explored and trained a variety of known deep learning models to find which is the best for our use case. More specifically, we have trained a vanilla Long Short-Term Memory (LSTM) model, a Bi-directional LSTM, an Encoder-Decoder LSTM, a Encoder-Decoder Bi-directional LSTM and finally a Transformer model

3.0 Data Curation, Compilation and Preparation

Data preparation is a crucial step in any machine learning project, and this project was no exception. For our work, we successfully gathered data from two different sources and merged them into a single cohesive dataset, enabling richer feature representation and more meaningful training inputs for the model.

3.1 Understanding and Retrieving the Data

3.1.1 Taxi Availability Dataset

To accomplish our goal, the team first looked at available datasets and what information can be obtained. We narrowed down on the Taxi Availability API available on data.gov.sg website and were able to query good information such as Taxi Availability per 5 minutes as well as every single taxi and their coordinates. To ensure a focused analysis, and obtain a good proof of concept, we narrowed our geographical area within a specific bounding box around Singapore University of Technology and Design (SUTD) which is as shown below:

Fig 3.1.1a, Specific bounding box
This can be obtained online using OpenStreetMaps.
Coordinates are: north = 1.35106, south = 1.32206, east = 103.97839, west = 103.92805

A simple python script was written to retrieve hourly taxi availability data from this area for the past three years.

After several observations of the dataset, we notice that there is actually significant noise in the dataset. We were concerned about the sampling rate error when we are only taking data every hour. As such, we decided to check the dataset and compare trends between data taken every 5 minutes and data taken every hour.

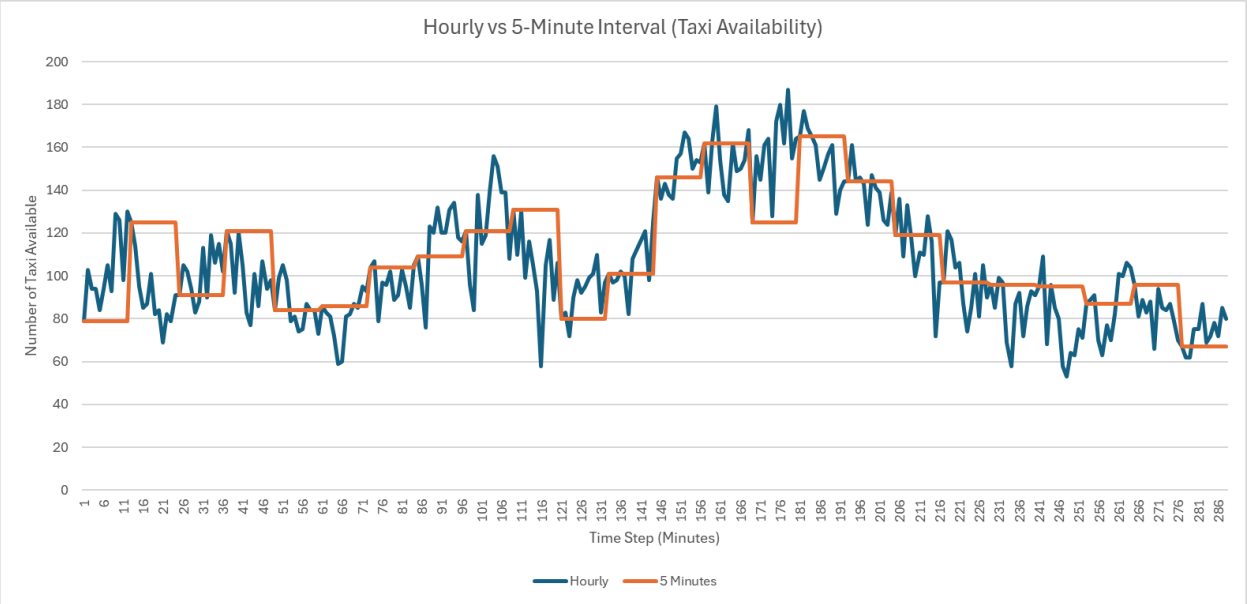


Fig 3.1.1b. Hourly vs 5-Minute Interval (Taxi Availability).
This can be found in the taxi_availability_5_minutes_3_years.csv under data_retrieval_and_cleaning folder

We can see that actually, when the hourly data showed that there are changes, some of these changes are not present in the 5-minute interval data. This might result in the model learning trends that are actually not really there. As such, we decided to denoise the signal but choose to use average taxi availability instead. This also made more sense for prediction as we are not trying to predict the number of taxi available at each hour start but rather the number of taxi available on average within that hour. Below shows the difference between the average data (done by taking the 12 5-minute data points per hour and averaging it) vs data taken at hourly point time:

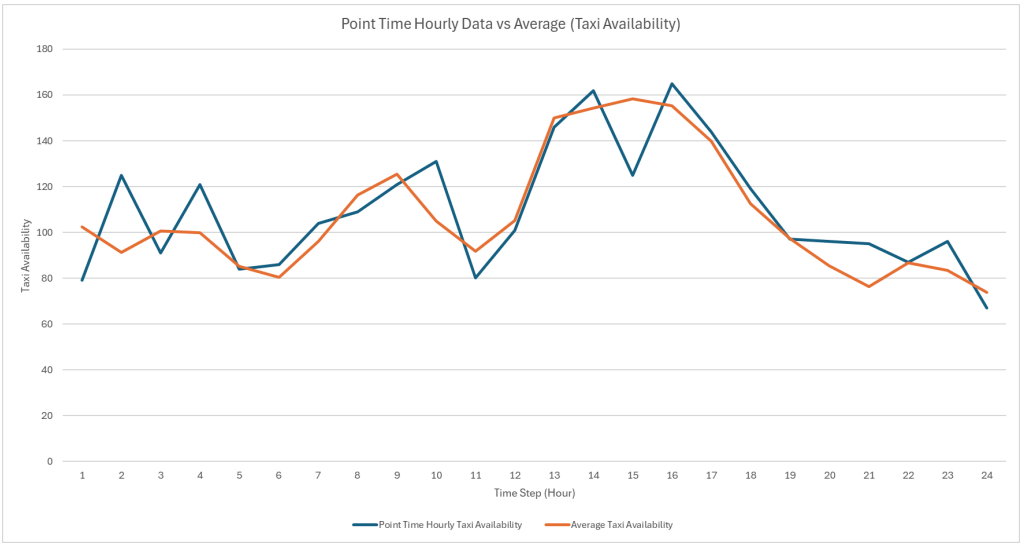


Fig 3.1.1c. Point Time Hourly Data vs Average (Taxi Availability).
This can be found in the merged_file_with_mean.csv inthe root folder

We can see that of course the data is much smoother, which we hope would allow the model to learn more general trends instead of noise.

3.1.2 Weather Dataset

In addition to the Taxi Availability dataset, we looked to enhance the accuracy of our prediction by integrating weather information as we believe it can significantly impact transportation patterns. Since the taxi API allows for bounding-box precision but the weather API does not, we selected data from the East Coast Parkway (ECP) weather station, which is geographically the closest to our area of interest. In the end, we retrieve the rainfall, air temperature and relative humidity data for the past 3 years.

3.1.3 Combining Dataset

To finally use our dataset, we join the two dataset into a unified dataframe by the date and time and drop unnecessary columns such as weather station location and coordinates.
The datetime was split into “IsWeekend” instead of using the raw data, and also the time was converted into 0 to 23 as well so as to make these parameters usable for the model.

Our final dataset contains the following columns: Taxi Available throughout SG, Average Taxi Availability, temp_value (temperature), humidity_value (humidity), rainfall_value (rainfall), IsWeekend, Hour (represents the time of the day) and peak_period.

```
<bound method NDFrame.head of        Taxi Available throughout SG  temp_value  humidity_value  \
0                              1924        27.1            84.1
1                              2259        27.3            82.5
2                              2400        27.4            81.2
3                              2677        27.5            81.9
4                              2437        27.7            78.0
...                             ...         ...             ...
25586                          1962        27.0            82.9
25587                          2025        27.3            81.7
25588                          2144        27.4            82.2
25589                          2447        27.3            82.9
25590                          2615        27.6            82.3

       rainfall_value  peak_period  Average Taxi Availability  IsWeekend  Hour
0                 0.0            1                 102.416667          0    24
1                 0.0            1                  91.166667          0    23
2                 0.0            1                 100.666667          0    22
3                 0.0            1                  99.916667          0    21
4                 0.0            1                  85.250000          0    20
...               ...          ...                        ...        ...   ...
25586             0.0            0                 101.000000          0     5
25587             0.0            0                 109.500000          0     4
25588             0.0            0                 115.583333          0     3
25589             0.0            0                 121.833333          0     2
25590             0.0            0                 107.583333          0     1

[25591 rows x 8 columns]>
```

Fig 3.1.3a, Preview of the resulting dataframe
This can be seen when the python notebook for training the model is run

3.2 Data Cleaning

3.2.1 Data Normalisation

After a few initial training and tests with our model, we realised that all the inputs and outputs of our data had to be normalized. Without it, the training process became unstable, leading to poor model performance and was not training and converging to a satisfactory level.

3.2.2 Sequence Creation

We also organised the data into daily sequences instead of the conventional sliding window technique. Each sequence represents a full day, with 24 timesteps (one for each hour in the day) and the next sequence begins at the start of the next day, ensuring clear segmentation between each sequence via calendar days.

While the sliding window method creates many overlapping sequences and a larger number of training samples, our method creates fewer, non-overlapping sequences. However, this comes with important benefits such as the model becoming more stable and giving more consistent results as well as helping the model to learn daily trends in taxi availability which changes significantly from day to day.

The corresponding target value (“y”) for each sequence is defined as the taxi availability for the next three hours following the input sequence. This allows the model to learn how to forecast short-term availability based on daily trends, aligning with real-world use cases where short-term planning (e.g., for dispatch or navigation) is critical.

```python
seq_length = 24
pred_horizon = 3  # Number of future time steps to predict

def create_sequences(data, labels, seq_length, pred_horizon):
    xs, ys = [], []
    for i in range(0, len(data), seq_length):  # Start from 0 and increment by seq_length
        if i + seq_length + pred_horizon <= len(data):  # Ensure enough data for prediction horizon
            xs.append(data[i:i + seq_length])  # Input sequence (continuous)
            ys.append(labels[i + seq_length : i + seq_length + pred_horizon])  # Next 3 values
    return np.array(xs), np.array(ys)
```

Fig 3.2.2a, Code for creating the sequence.

3.2.3 DataLoader

After creating the sequences, we split the dataset via a typical train-validation-test split of 80/10/10.

```
Total samples: 1066
Training samples: 852, Batches: 50
Validation samples: 106, Batches: 6
Testing samples: 108, Batches: 6
```

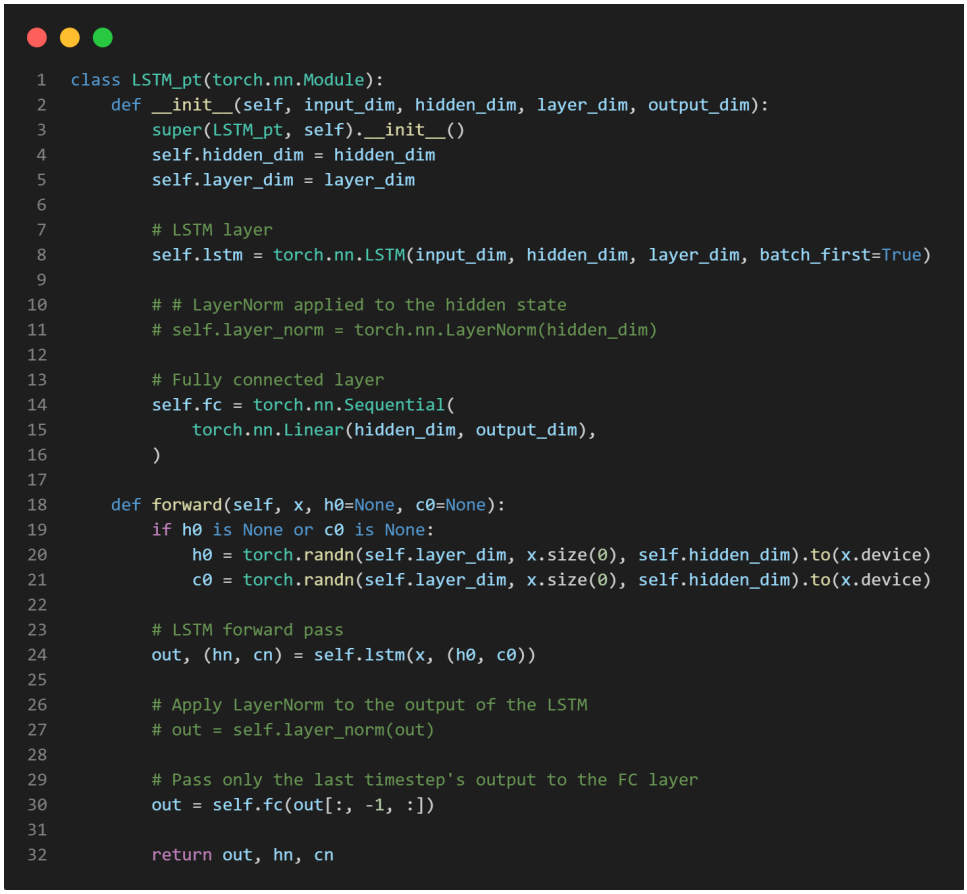
Fig 3.2.3a, Training, validation and testing samples split.

4.0 Long Short Term Model (LSTM)

4.1 Implementation

Our first implementation of a predictive model is based on a vanilla Long Short-Term Memory (LSTM) network. LSTMs are particularly well-suited for time series forecasting tasks due to their ability to capture both short-term fluctuations and long-term temporal dependencies. Given that taxi availability is influenced by both immediate trends (e.g., time of day, weather) and historical patterns (e.g., weekday vs. weekend behavior), the LSTM was a natural choice for our problem.

This LSTM setup served as a strong baseline and offered valuable insights into the temporal structure of taxi availability in our chosen region.



```python
class LSTM_pt(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(LSTM_pt, self).__init__()
        self.hidden_dim = hidden_dim
        self.layer_dim = layer_dim

        # LSTM layer
        self.lstm = torch.nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

        # # LayerNorm applied to the hidden state
        # self.layer_norm = torch.nn.LayerNorm(hidden_dim)

        # Fully connected layer
        self.fc = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, output_dim),
        )

    def forward(self, x, h0=None, c0=None):
        if h0 is None or c0 is None:
            h0 = torch.randn(self.layer_dim, x.size(0), self.hidden_dim).to(x.device)
            c0 = torch.randn(self.layer_dim, x.size(0), self.hidden_dim).to(x.device)

        # LSTM forward pass
        out, (hn, cn) = self.lstm(x, (h0, c0))

        # Apply LayerNorm to the output of the LSTM
        # out = self.layer_norm(out)

        # Pass only the last timestep's output to the FC layer
        out = self.fc(out[:, -1, :])

        return out, hn, cn
```

Fig 4.1a, LSTM class

```python
def train(model, dataloader, val_loader, num_epochs, learning_rate, device):
    # Set the loss function and optimizer
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    model.train()  # Set the model to training mode
    loss_graph = []
    val_graph = []

    for epoch in range(num_epochs):
        epoch_loss = 0.0

        hidden_state, cell_state = None, None  # Reset for each epoch

        for batch_idx, (inputs, targets) in enumerate(dataloader):
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()

            output, hidden_state, cell_state = model(inputs, hidden_state, cell_state)
            output = output.unsqueeze(-1)

            # Compute loss
            loss = criterion(output, targets)
            loss.backward()
            optimizer.step()

            hidden_state = hidden_state.detach()
            cell_state = cell_state.detach()

            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(dataloader)
        loss_graph.append(avg_loss)

        model.eval()
        val_loss = 0.0

        with torch.no_grad():
            for val_inputs, val_targets in val_loader:
                val_inputs, val_targets = val_inputs.to(device), val_targets.to(device)

                val_output, _, _ = model(val_inputs, hidden_state, cell_state)
                val_output = val_output.unsqueeze(-1)

                # Compute validation loss
                v_loss = criterion(val_output, val_targets)
                val_loss += v_loss.item()

            model.train()
            avg_val_loss = val_loss / len(val_loader)
            val_graph.append(avg_val_loss)


        if epoch % 50 == 0 or epoch == num_epochs - 1:
            print(f'Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.6f}, Val Loss: {avg_val_loss:.6f}')

    return loss_graph, val_graph
```

Fig 4.1b, LSTM training function

4.2 Hyper Parameters

For this project, we will be tuning our hyper parameters using Weight and Biases sweeps (https://docs.wandb.ai/guides/sweeps/). This will allow us to have a comprehensive sweeping of hyper parameters. There are various methods to search for the best hyper parameters. Below are three and their comparisons:

	Bayesian	Grid Search	Random
Search can be done in parallel	No	Yes	Yes
Informed search	Yes	No	No
Iterations	Least	Most	Less
Run Time (due to different iterations required)	Shortest	Longest	Middle

In the end, we went with Bayesian search due to our limited hardware (so we want to run as few iterations as possible) and its shortest total run time allowing us to do more experiments.

For all of our models, we start off with arbitrary hyper parameters, such as implementing the hidden_size as 50 and using only 1 LSTM layer. This is to first ensure that the model can work and can be trained/learning properly.

Thereafter, we set up various hyper parameters as variables for us to train. For the LSTM, we decided to tune the hidden_size, learning_rate and number of epoch.

Below is our tuning configuration code:

```
sweep_config = {
    "method": "bayes",
    "metric": {
        "name": "val_loss",
        "goal": "minimize"
    },
    "parameters": {
        "learning_rate": {
            "min": 1e-3,
            "max": 0.01
        },
        "hidden_size": {
            "values": [64, 128, 256]
        },
        "num_epochs": {
            "values": [300, 500, 1000]
        }
    }
}
```

Fig 4.2a, Sweep Configurations

We then modify the training to use the sweep. This is done for every model:

```python
def sweep_train():
    wandb.init()
    config = wandb.config

    model = LSTM_pt(8, config.hidden_size, 1, 3).to(device)

    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate, weight_decay=1e-5)

    loss_graph = []
    val_graph = []

    model.train()
    for epoch in tqdm(range(config.num_epochs)):
        epoch_loss = 0.0

        hidden_state, cell_state = None, None  # Reset for each epoch

        for batch_idx, (inputs, targets) in enumerate(dataloader):
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()

            output, hidden_state, cell_state = model(inputs, hidden_state, cell_state)
            output = output.unsqueeze(-1)

            # Compute loss
            loss = criterion(output, targets)
            loss.backward()
            optimizer.step()

            hidden_state = hidden_state.detach()
            cell_state = cell_state.detach()

            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(dataloader)
        loss_graph.append(avg_loss)

        model.eval()
        val_loss = 0.0

        with torch.no_grad():
            for val_inputs, val_targets in val_loader:
                val_inputs, val_targets = val_inputs.to(device), val_targets.to(device)

                val_output, _, _ = model(val_inputs, hidden_state, cell_state)
                val_output = val_output.unsqueeze(-1)

                # Compute validation loss
                v_loss = criterion(val_output, val_targets)
                val_loss += v_loss.item()

            model.train()
            avg_val_loss = val_loss / len(val_loader)
            val_graph.append(avg_val_loss)

        # Log to W&B
        wandb.log({
            "epoch": epoch,
            "train_loss": avg_loss,
            "val_loss": avg_val_loss
        })

        if epoch % 25 == 0:
            print(f"Epoch {epoch+1}/{config.num_epochs}, Train Loss: {avg_loss}, Val Loss: {avg_val_loss}")

    return loss_graph, val_graph
```

Fig 4.2b, Modified Training Function for Sweep

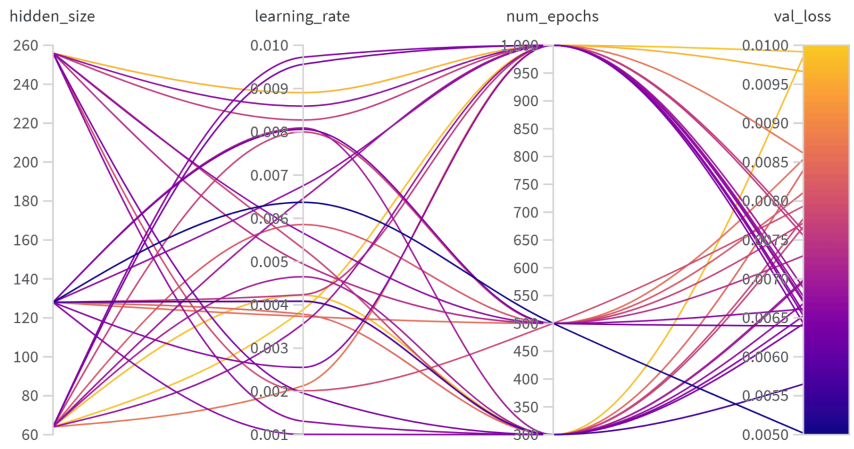
Finally, we initialize our sweep as is done for all model:

```python
sweep_id = wandb.sweep(sweep_config, project="DeepLearning Project")
wandb.agent(sweep_id, function=sweep_train, count=30)
```

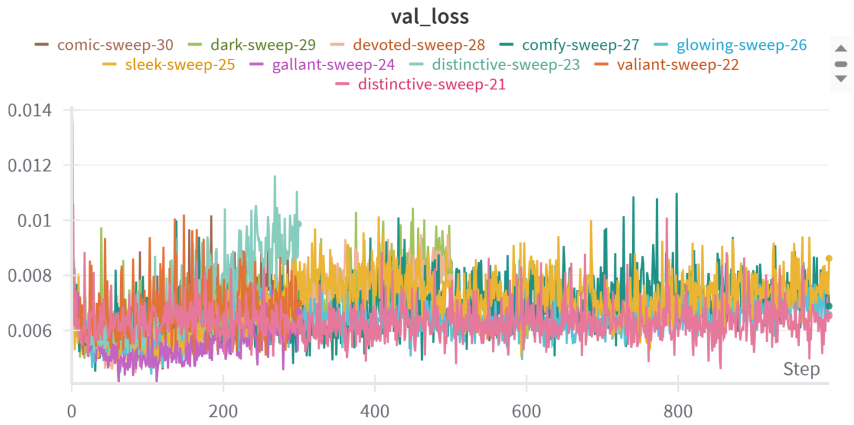
Fig 4.2c, Initialise and running sweep

From this, we ran 30 runs of the model to see which parameter will result in the best performance. We can see that in general, the higher the learning rate, the better the

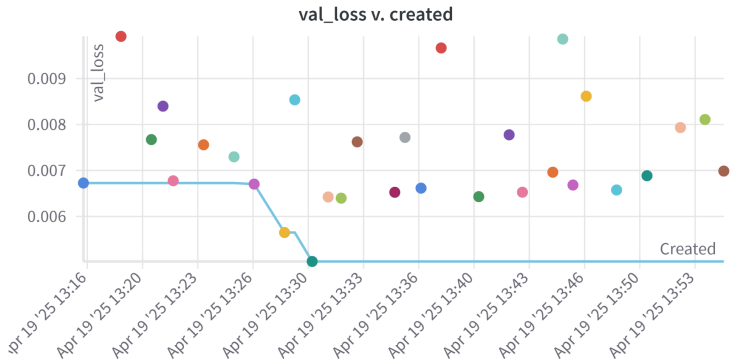
performance but we also understand that it seems regardless of the hyperparameters, the majority of the val_loss will end up at around 0.006.



4.2b, Hyperparameters chosen and the resulting validation loss



4.2 c, Validation Loss curve for each run



4.2d, Validation loss over time across different runs

Additionally, we also see that due to overfitting, the validation loss starts going back up from around 50 epoch onwards. Regardless, the best parameters are based on the 30 runs: hidden_size = 128, num_epochs = 500, learning_rate = 0.00637. From here, we will further reduce the epoch to only around 50 and it should yield better results without overfitting as much.

4.3 Results

Overall, the results indicate that the model is able to learn, as shown by the decreasing trend in training loss. However, there remains a noticeable gap between the training and validation losses, suggesting potential instability and limited generalization. Despite this, the model performs relatively well compared to others introduced later, achieving a test set MAE of around 16. This indicates a strong result that highlights its effectiveness in capturing key patterns in the data.

```
Normalized output[0]: [[0.3320006728172302], [0.3224940896034241], [0.31204646825790405]]
Normalized target[0]: [[0.2721712589263916], [0.2453279048204422], [0.22799864411354065]]
Denormalized output[0]: [[81.42316436767578], [79.0916748046875], [76.5293960571289]]
Denormalized target[0]: [[66.75], [60.16666793823242], [55.91666793823242]]
--------------------------------------------------
Normalized output[0]: [[0.34856271743774414], [0.35584086179733276], [0.3918035328388214]]
Normalized target[0]: [[0.3530411124229431], [0.4172613024711609], [0.4108053147792816]]
Denormalized output[0]: [[85.4850082397461], [87.26997375488281], [96.08981323242188]]
Denormalized target[0]: [[86.58333587646484], [102.33333587646484], [100.75]]
--------------------------------------------------
Normalized output[0]: [[0.4242748022079468], [0.4503340125083923], [0.4419099688529968]]
Normalized target[0]: [[0.36153584718704224], [0.3611960709095001], [0.3459055423736572]]
Denormalized output[0]: [[104.05339813232422], [110.44441986083984], [108.37841796875]]
Denormalized target[0]: [[88.66666412353516], [88.58333587646484], [84.83333587646484]]
--------------------------------------------------
Normalized output[0]: [[0.4344785213470459], [0.4459074139595032], [0.45592787861824036]]
Normalized target[0]: [[0.3785253167152405], [0.418280690908432], [0.383961945772171]]
Denormalized output[0]: [[106.55585479736328], [109.35879516601562], [111.81631469726562]]
Denormalized target[0]: [[92.83333587646484], [102.58333587646484], [94.16666412353516]]
--------------------------------------------------
Normalized output[0]: [[0.32726365327835083], [0.34130859375], [0.3686755895614624]]
Normalized target[0]: [[0.33129459619522095], [0.2643561065196991], [0.4012911915779114]]
Denormalized output[0]: [[80.26141357421875], [83.7059326171875], [90.41768646240234]]
Denormalized target[0]: [[81.25], [64.83333587646484], [98.41666412353516]]
--------------------------------------------------
...
--------------------------------------------------
Average Validation Loss: 0.0087
Mean Absolute Error: 16.3315
Average Validation Loss: 0.0087
```

Fig 4.3a, LSTM test set results, showing MAE.

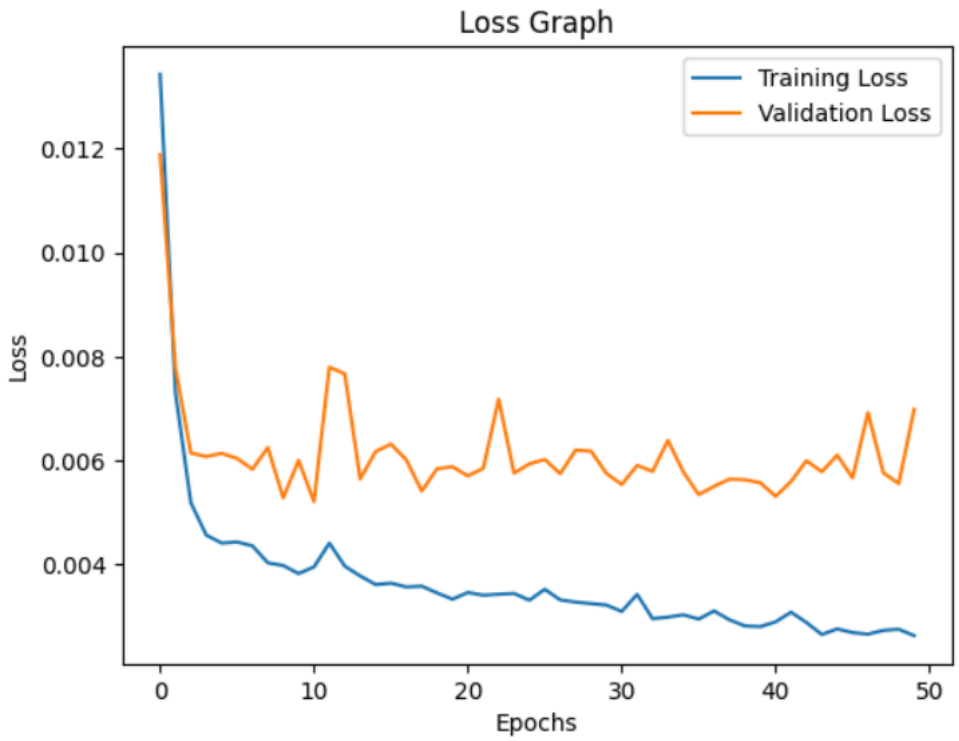


Fig 4.3b, LSTM Loss graph

Although the MAE results were promising, we decided to change the model into a Bi-LSTM to see if it would be able to train with the dataset with more stability and generalize better. Additionally, it can be seen that despite the validation loss flat lining, the training loss is still going down which means that we might still be able to obtain a better MAE. Below shows some of the better predictions against the target. We can see that it is not too bad.

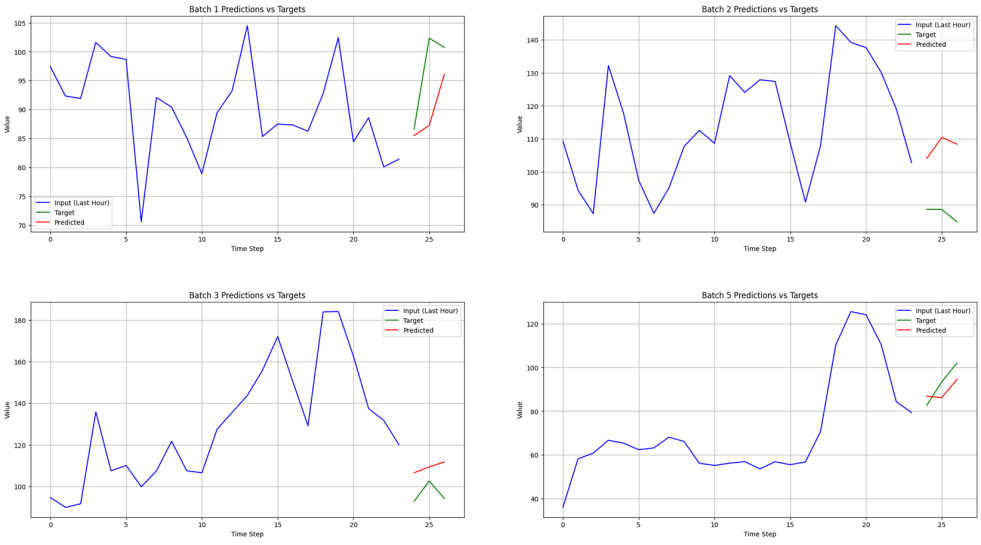


Fig 4.3c, LSTM Predictions vs Targets Graphs

5.0 Bidirectional-LSTM (BiLSTM)

A variant of the LSTM architecture, known as the BiLSTM (Bidirectional LSTM), is commonly used for sequence modeling tasks such as time series forecasting and sequence classification. In this model, information is processed in both forward and backward directions, allowing it to capture patterns and trends from the entire sequence. The BiLSTM model is built using two key components: a BiLSTM layer and a fully connected (FC) layer.

The BiLSTM works by maintaining two hidden states and two cell states, one for processing the sequence in the forward direction (from past to future) and the other for processing the sequence in the backward direction (from future to past). This dual-state mechanism enables the model to retain both short-term and long-term dependencies from the entire sequence, rather than just from one direction.

In the context of our project, we hypothesize that there may be trends in taxi availability that can be more effectively captured by considering not only the past data (forward sequence) but also information from the future (backward sequence). By capturing the dependencies in both directions, the BiLSTM is particularly well-suited to identify complex patterns and improve the accuracy of predictions in time series data.

5.1 Implementation

The BiLSTM layer processes the input sequence in both forward and backward directions, enabling the model to capture dependencies from both past and future time steps simultaneously. As a result, the output dimension per time step is doubled, becoming “2 * hidden_dim” to account for the information learned from both directions.

This increase in output dimension is reflected in the fully connected layer, where the input dimension must match “2 * hidden_dim” to accommodate the combined output from both the forward and backward passes.

Additionally, when initializing the hidden and cell states in the forward function, they must also be scaled by a factor of 2 to align with the bidirectional structure, ensuring that both forward and backward states are considered during computation.

```python
class BiLSTM_pt(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(BiLSTM_pt, self).__init__()
        self.hidden_dim = hidden_dim
        self.layer_dim = layer_dim
        self.num_directions = 2  # Since it's bidirectional

        # LSTM layer
        self.lstm = torch.nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True, bidirectional=True)

        # Fully connected layer
        self.fc = torch.nn.Linear(hidden_dim * 2, output_dim)  # Fix here

    def forward(self, x, h0=None, c0=None):
        if h0 is None or c0 is None:
            h0 = torch.randn(self.layer_dim * self.num_directions, x.size(0), self.hidden_dim).to(x.device)
            c0 = torch.randn(self.layer_dim * self.num_directions, x.size(0), self.hidden_dim).to(x.device)

        # LSTM forward pass
        out, (hn, cn) = self.lstm(x, (h0, c0))

        # Pass only the last timestep's output to the FC layer
        out = self.fc(out[:, -1, :])

        return out, hn, cn
```

Fig5.1a, BiLSTM class

```python
def train(model, dataloader, val_loader, num_epochs, learning_rate):
    # Set the loss function and optimizer
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    model.train()  # Set the model to training mode
    loss_graph = []
    val_graph = []

    for epoch in range(num_epochs):
        epoch_loss = 0.0

        hidden_state, cell_state = None, None  # Reset for each epoch

        for batch_idx, (inputs, targets) in enumerate(dataloader):
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()

            output, hidden_state, cell_state = model(inputs, hidden_state, cell_state)
            output = output.unsqueeze(-1)
            output = output.permute(0, 2, 1)

            # Compute loss
            loss = criterion(output, targets)
            loss.backward()
            optimizer.step()

            hidden_state = hidden_state.detach()
            cell_state = cell_state.detach()

            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(dataloader)
        loss_graph.append(avg_loss)

        # Validation step (optional)
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for val_inputs, val_targets in val_loader:
                val_inputs, val_targets = val_inputs.to(device), val_targets.to(device)

                val_output, _, _ = model(val_inputs, hidden_state, cell_state)
                val_output = val_output.unsqueeze(-1).to(device)
                val_output = val_output.permute(0, 2, 1).to(device)

                # Compute validation loss
                v_loss = criterion(val_output, val_targets)
                val_loss += v_loss.item()

        avg_val_loss = val_loss / len(dataloader)
        val_graph.append(avg_val_loss)
        model.train()

        if epoch % 50 == 0 or epoch == num_epochs - 1:
            print(f'Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.6f}, Validation Loss: {avg_val_loss:.6f}')
            # Save model after every epoch
            save_path = os.path.join(f'./bi_LSTM/bi_LSTM{epoch+1}_loss_{avg_loss:.6f}.pth')
            torch.save(model.state_dict(), save_path)

    return loss_graph, val_graph
```

Fig 5.1b, BiLSTM training function

5.2 Hyper Parameters

Similar to above, we have done hyper parameter tuning with wandb as well. Below is the setup code:

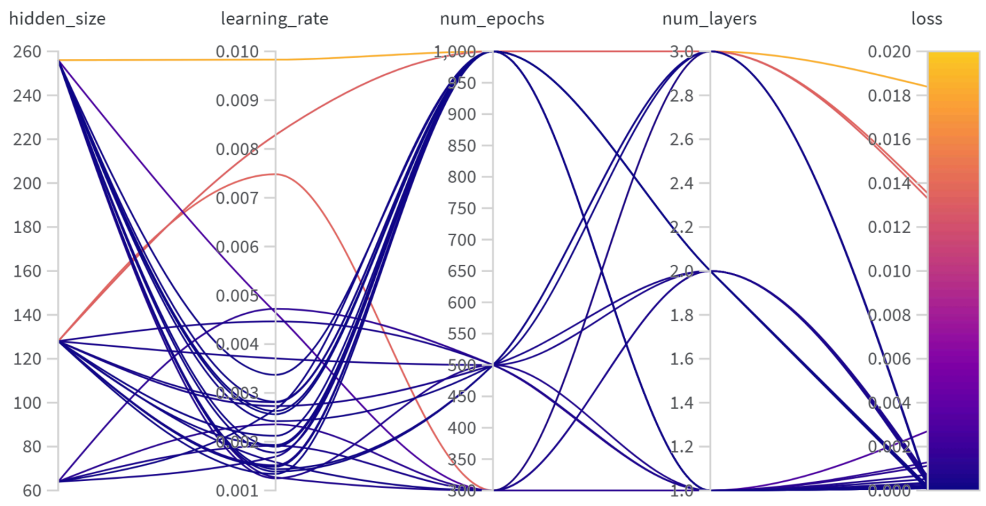
```
sweep_config = {
    "method": "bayes",
    "metric": {
        "name": "loss",
        "goal": "minimize"
    },
    "parameters": {
        "learning_rate": {
            "min": 1e-3,
            "max": 0.01
        },
        "hidden_size": {
            "values": [64, 128, 256]
        },
        "num_layers": {
            "values": [1, 2, 3]
        },
        "num_epochs": {
            "values": [300, 500, 1000]
        }
    }
}
```

Fig 5.2a, Sweep Configurations

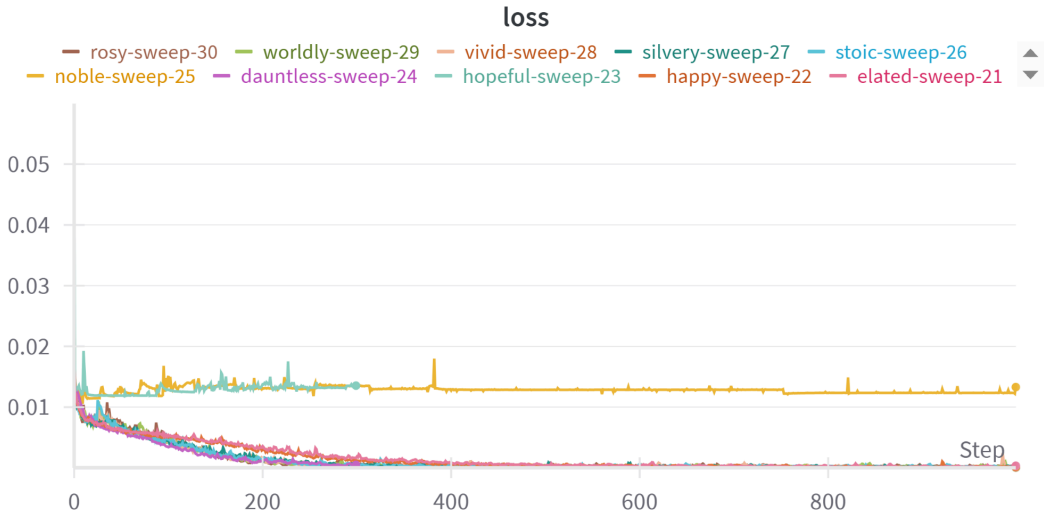
Most importantly, we are adjusting our hidden size to see if a larger hidden size will allow it to learn longer patterns (or if it is even necessary) and the number of LSTM layers will allow us to tune the complexity of the model. Finally we will be tuning the number of epoch as well to be able to observe any overfitting.

Similarly to the previous model, we decided to do 30 runs because we have 3 x 3 x 3 = 27 discrete options and a continuous learning rate. With Bayesian search, we do not need to do ALL the combinations. As such, we decided 30 which is more than discrete options and should be able to adequately tune the learning rate as well.

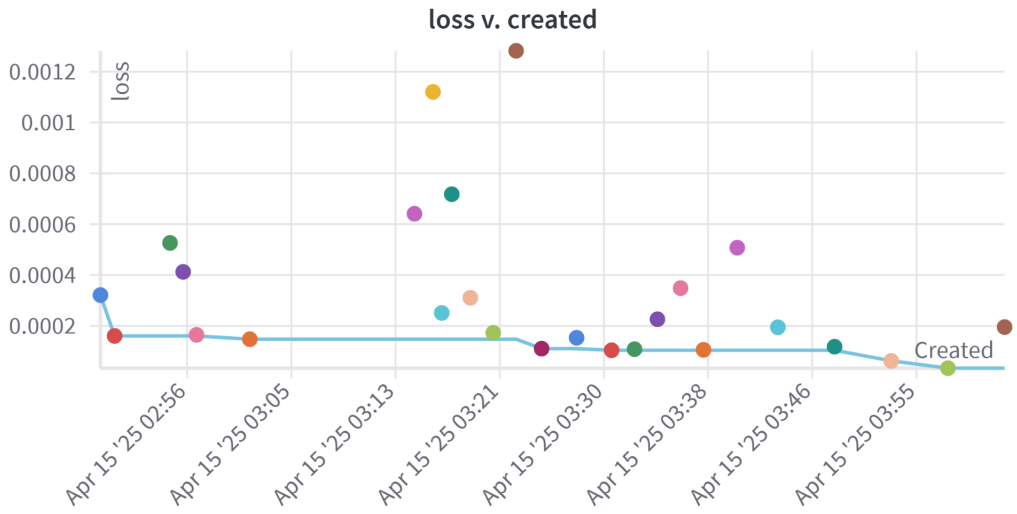
The results from the hyper parameter tuning are as shown:



5.2 a, Hyperparameters chosen and the resulting validation loss



5.2 b, Validation Loss curve for each run



5.2 c, Validation loss over time across different runs

Interestingly, what we notice is that while the validation loss is close to the training loss, and the loss value is actually lower than training and the base LSTM, in reality, it is actually unable to produce a better prediction than the base LSTM. Regardless, from this, we understood that the biggest factor was the learning rate and through the sweep we identified that run 29 with the following parameters provided the lowest loss:

hidden_size: 256
learning_rate: 0.003370260102354819
num_epochs: 1000
num_layers: 2

5.3 Results

With these parameters, we finalize the Bi-LSTM model. The resulting loss graph is shown below. To evaluate performance, we compute the final MAE and validation loss as follows:

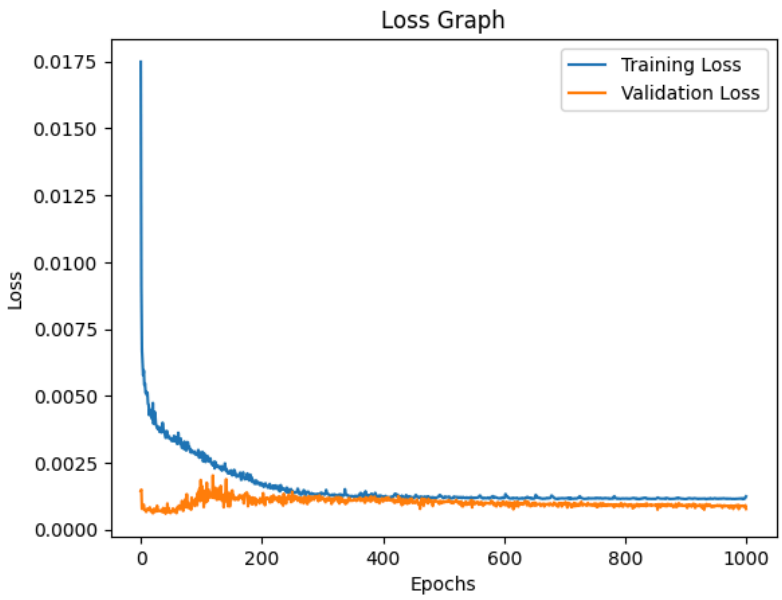


Fig 5.3a, BiLSTM Loss Graph

```
Normalized output[0]: [[0.3401377201080322, 0.3402456045150757, 0.3402026891708374]]
Normalized target[0]: [[0.4254162609577179], [0.4447842538356781], [0.5280326008796692]]
Denormalized output[0]: [[83.41877746582031], [83.44523620605469], [83.43470764160156]]
Denormalized target[0]: [[104.33333587646484], [109.08333587646484], [129.5]]
--------------------------------------------------
Normalized output[0]: [[0.5000786185264587, 0.5000811219215393, 0.5000476241111755]]
Normalized target[0]: [[0.28950050473213196], [0.253482848405838], [0.2881413400173187]]
Denormalized output[0]: [[122.64427947998047], [122.6448974609375], [122.63668060302734]]
Denormalized target[0]: [[71.0], [62.16666793823242], [70.66666412353516]]
--------------------------------------------------
Normalized output[0]: [[0.33464109897613525, 0.3346370756626129, 0.33469393849372864]]
Normalized target[0]: [[0.3479442894458771], [0.3217804729938507], [0.49643221497535706]]
Denormalized output[0]: [[82.07073211669922], [82.06974029541016], [82.08368682861328]]
Denormalized target[0]: [[85.33333587646484], [78.91666412353516], [121.75]]
--------------------------------------------------
Normalized output[0]: [[0.2134014368057251, 0.2133832573890686, 0.2133883833885193]]
Normalized target[0]: [[0.1994563490152359], [0.20931021869182587], [0.20965002477169037]]
Denormalized output[0]: [[52.33670425415039], [52.332244873046875], [52.333499908447266]]
Denormalized target[0]: [[48.91666793823242], [51.33333206176758], [51.41666793823242]]
--------------------------------------------------
Normalized output[0]: [[0.2796518802642822, 0.27961504459381104, 0.2797371447086334]]
Normalized target[0]: [[0.3367312252521515], [0.34386682510375977], [0.3710499405860901]]
Denormalized output[0]: [[68.58462524414062], [68.57559204101562], [68.60553741455078]]
Denormalized target[0]: [[82.58333587646484], [84.33333587646484], [91.0]]
--------------------------------------------------
...
Predicted output shape: torch.Size([17, 1, 3])
True output shape: torch.Size([17, 3, 1])
Average Validation Loss: 0.0023
Mean Absolute Error: 19.0398
```

Fig 5.3b, BiLSTM test set results, showing MAE

This is interesting as even though the loss is similar or less than base LSTM, the result is not better. Seeing this, we decided to take a look at what it is actually predicting to achieve this “good loss” value.

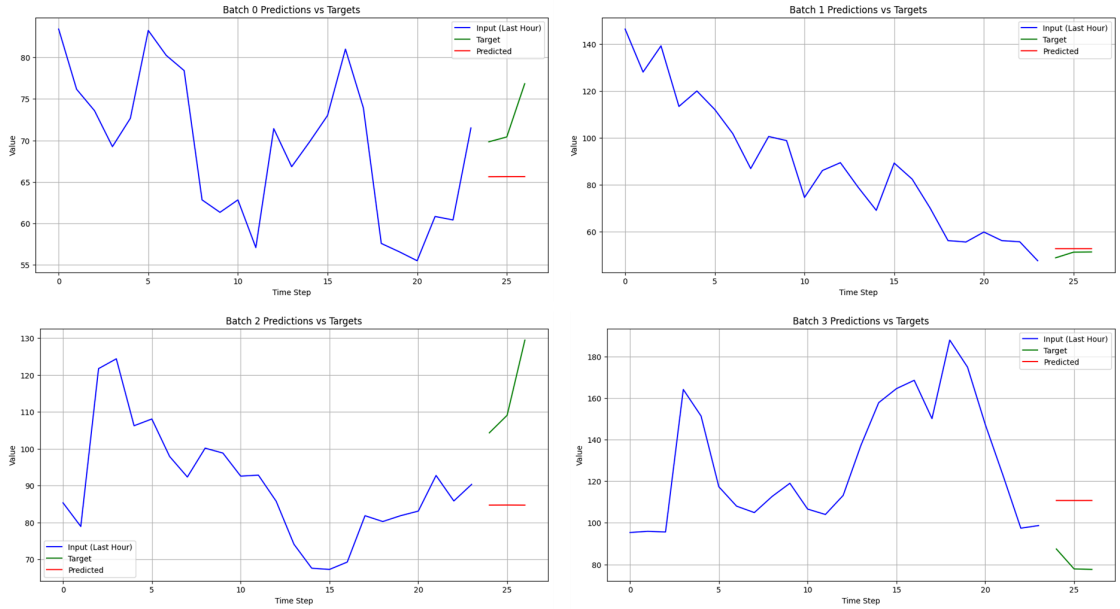


Fig 5.3c, Bi-LSTM Predictions vs Targets Graphs

From the graphs above, the model appears to default to predicting a flat or constant sequence, where all three predicted time steps are nearly equal. This behavior suggests that the model may have learned to minimize error by averaging out the signal, rather than capturing meaningful temporal dynamics.

Ultimately, while the model may achieve a lower validation loss, it fails to learn meaningful sequential patterns. This indicates that the current BiLSTM configuration may not well-suited for capturing the temporal dependencies needed for accurate sequence prediction.

6.0 ED-LSTM

6.1 Implementation

Another hypothesis that we have is that the taxi availability dataset is likely influenced by previous time steps. For example, the availability of taxis at 14:00 is likely influenced by the availability at 13:00, since they are only an hour apart and likely to follow similar trends. To capture this short-term dependency, we decided to implement an ED-LSTM.

However, unlike traditional ED-LSTMs, where only the time steps and output are considered, our implementation uses all 8 parameters (such as weather, time, peak period and other features) as inputs and outputs the taxi availability for the next 3 hours. To accommodate this, we modified the standard ED-LSTM architecture: while the encoder remains the same, we modified the decoder to use linear layers to adjust the input dimensions appropriately.

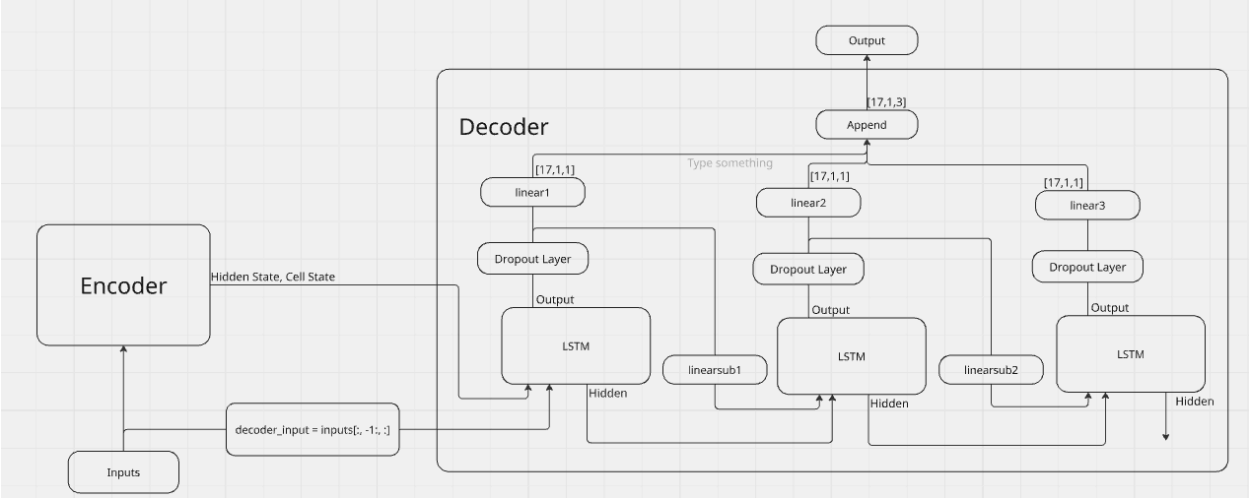


Fig 6.1a, Overview of decoder

In this setup, the encoder takes a batch of inputs with dimensions [17, 24, 8], where 17 represents the batch size, 24 is the number of time steps, and 8 refers to the number of features. For this, we set the hidden layer size of the LSTM to 128, resulting in memory vectors with the shape [1, 17, 128]. These vectors represent the hidden and cell states of the encoder.

The decoder LSTM needs to accommodate both the hidden state vector from the encoder and the input size of 8, since the first input to the decoder will be the last dataset input.

After this, we ignore the hidden and cell state vectors produced by the decoder LSTM. Instead, we focus on the output, which has the shape [17,24, 3]. The output is then passed through two different linear layers:

- One linear layer is used to generate the final predicted taxi availability.
- The second linear layer resizes the output to match the input size, which is then re-fed into the decoder LSTM for the next time step.

This process is repeated for a total of three steps, producing three separate outputs. The outputs are then stacked together into a final tensor with the shape [17, 3, 1], which contains the predicted taxi availability for the next three time steps.

```python
class EncoderLSTM(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderLSTM, self).__init__()
        self.hidden_size = hidden_size

        # print("Encoder's input size: ", input_size)
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)

    def forward(self, inputs):
        # Passing the input sequence through the LSTM

        output, (hidden, cell) = self.lstm(inputs)
        # print("encoder output size: ", output.shape)

        return hidden, cell
```

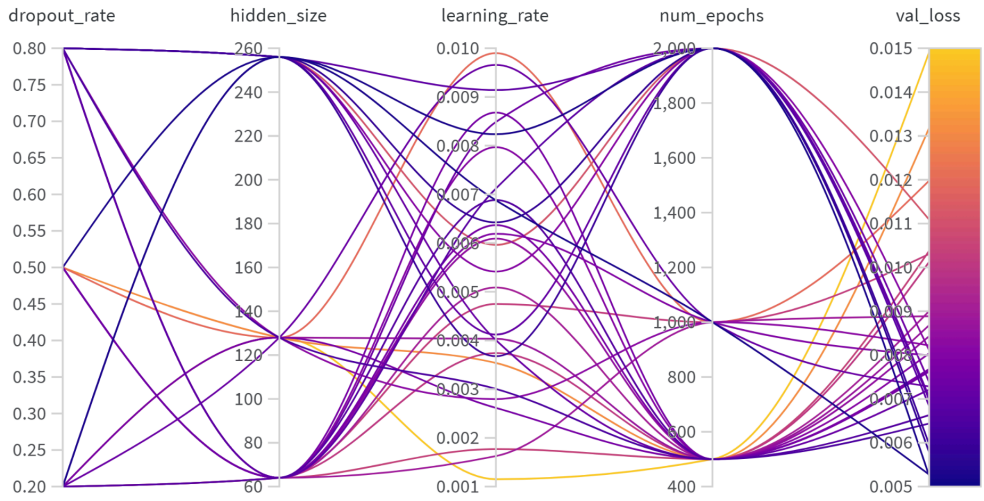
Fig 6.1b, ED-LSTM Encoder

```python
class DecoderLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, dropout_rate=0.3):
        super(DecoderLSTM, self).__init__()


        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)

        # Implementing Prof Idea
        self.linear1 = nn.Linear(hidden_size, 1)
        self.linearsub1 = nn.Linear(hidden_size, input_size)

        self.linear2 = nn.Linear(hidden_size, 1)
        self.linearsub2 = nn.Linear(hidden_size, input_size)

        self.linear3 = nn.Linear(hidden_size, 1)

        self.dropout = nn.Dropout(p=dropout_rate)

    def forward(self, x, hidden,target=None):
        outputs = []
        decoder_input = x

        # First LSTM layer
        y1, hidden1 = self.lstm(decoder_input, hidden)
        y1 = self.dropout(y1)
        linear_y1 = self.linear1(y1)
        linear_y1 = linear_y1.mean(dim=1, keepdim=True)
        linear_suby1 = self.linearsub1(y1)

        # Second LSTM layer
        y2, hidden2 = self.lstm(linear_suby1, hidden1)
        y2 = self.dropout(y2)
        linear_y2 = self.linear2(y2)
        linear_suby2 = self.linearsub1(y2)
        linear_y2 = linear_y2.mean(dim=1, keepdim=True)

        y3, hidden3 = self.lstm(linear_suby2, hidden2)
        y3 = self.dropout(y3)
        linear_y3 = self.linear3(y3)
        linear_y3 = linear_y3.mean(dim=1, keepdim=True)

        outputs.append(linear_y1.squeeze(1))
        outputs.append(linear_y2.squeeze(1))
        outputs.append(linear_y3.squeeze(1))

        final_output = torch.stack(outputs, dim=1)
        return final_output
```

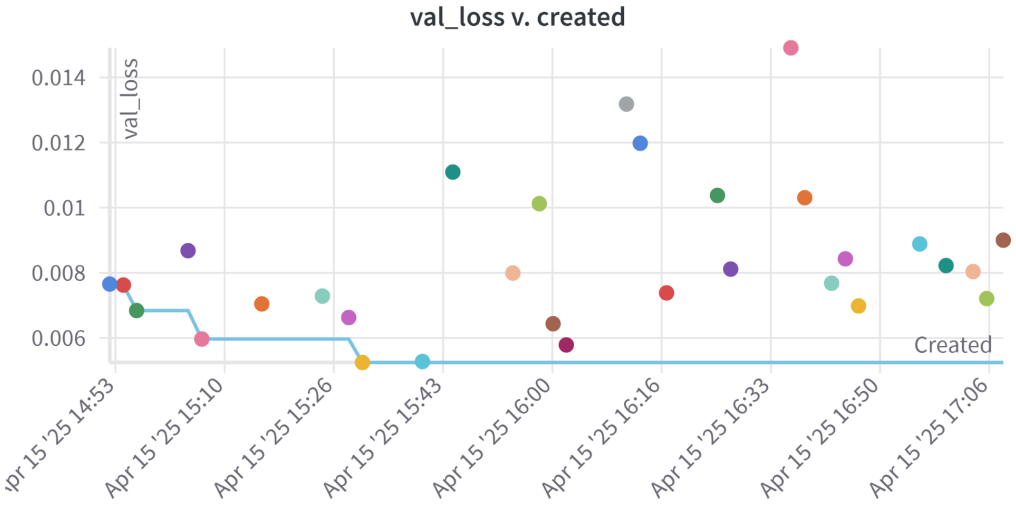
Fig 6.1c, ED-LSTM Decoder class

6.2 HyperParameters

Once again we run sweeps for hyper parameter tuning for the ED LSTM. The hyper parameters we tuned this time are similar to previous but as we learnt from the previous model, our model is likely overfitting so a drop out rate has been added. Additionally, weight decay (L2 regularization) was also added to help the model generalise better. Our purpose is to see if this model will be able to generalise..



7.2 a, Hyperparameters chosen and the resulting validation loss



7.2 b, Validation loss over time across different runs

Using Bayesian search for hyperparameter tuning, we achieve the above results. This is concerning as Bayesian search should result in a decrease in loss as more runs are conducted but we can see that our graph does not indicate this at all.

This meant that the algorithm was actually unable to find a direction towards reducing val loss which is concerning as it can mean that the hyper parameter we are tuning does not affect the model (which is unlikely the case as we are tuning key parameters like epoch) or the model is memorising the training data and not generalising at all even with regularisation added.

6.3 Results

During training and validation, the ED-LSTM model successfully reduced training loss over each epoch, indicating that it was able to learn patterns from the training data. However, its performance on the validation set did not reflect the same trend. The validation loss remained higher and fluctuated throughout training. While there was some reduction in validation loss, the gap between training and validation losses suggests that the model still struggles to generalize and overfitting.

One possible reason for this observation could be the increased complexity introduced by the final linear layer, where it resizes the outputs so that it can be either fed back into the LSTM or be used as the predicted output. This could make it harder for the network to converge effectively, leading to suboptimal validation performance. The MAE from the test set results is 49.0300.

```
Normalized output[0]: [[0.4695374667644501], [0.24443894624710083], [0.23713462054729462]]
Normalized target[0]: [[0.33129459619522095], [0.2643561065196991], [0.4012911915779114]]
Denormalized output[0]: [[115.15406036376953], [59.94865036010742], [58.157264709472656]]
Denormalized target[0]: [[81.25], [64.83333587646484], [98.41666412353516]]
--------------------------------------------------
Normalized output[0]: [[0.6830933094024658], [-0.0012765576830133796], [0.3442654013633728]]
Normalized target[0]: [[0.44648319482803345], [0.3978933095932007], [0.3560992181301117]]
Denormalized output[0]: [[167.5286407470703], [-0.3130757808685303], [84.43109130859375]]
Denormalized target[0]: [[109.5], [97.58333587646484], [87.33333587646484]]
--------------------------------------------------
Normalized output[0]: [[0.41228771209716797], [0.27134978771209717], [0.23640379309654236]]
Normalized target[0]: [[0.1607203632593155], [0.11688752472400665], [0.14237172901630402]]
Denormalized output[0]: [[101.11356353759766], [66.54853820800781], [57.978031158447266]]
Denormalized target[0]: [[39.41666793823242], [28.66666603088379], [34.91666793823242]]
--------------------------------------------------
Normalized output[0]: [[0.6676990985870361], [0.01384001411497593], [0.33371293544769287]]
Normalized target[0]: [[0.37410804629325867], [0.40366971492767334], [0.3948352038860321]]
Denormalized output[0]: [[163.75320434570312], [3.394263505935669], [81.84309387207031]]
Denormalized target[0]: [[91.75], [99.0], [96.83333587646484]]
--------------------------------------------------
Normalized output[0]: [[0.3895131051540375], [0.31920260190963745], [0.20602774620056152]]
Normalized target[0]: [[0.27387019991874695], [0.3676520586013794], [0.5861365795135498]]
Denormalized output[0]: [[95.52809143066406], [78.28443908691406], [50.52830505371094]]
Denormalized target[0]: [[67.16666412353516], [90.16666412353516], [143.75]]
--------------------------------------------------
Predicted output shape: torch.Size([17, 3, 1])
True output shape: torch.Size([17, 3, 1])
Average Validation Loss: 0.0624
Mean Absolute Error: 49.0300
```

Figure 7.3a, ED-LSTM t*est set results, showing MAE.*

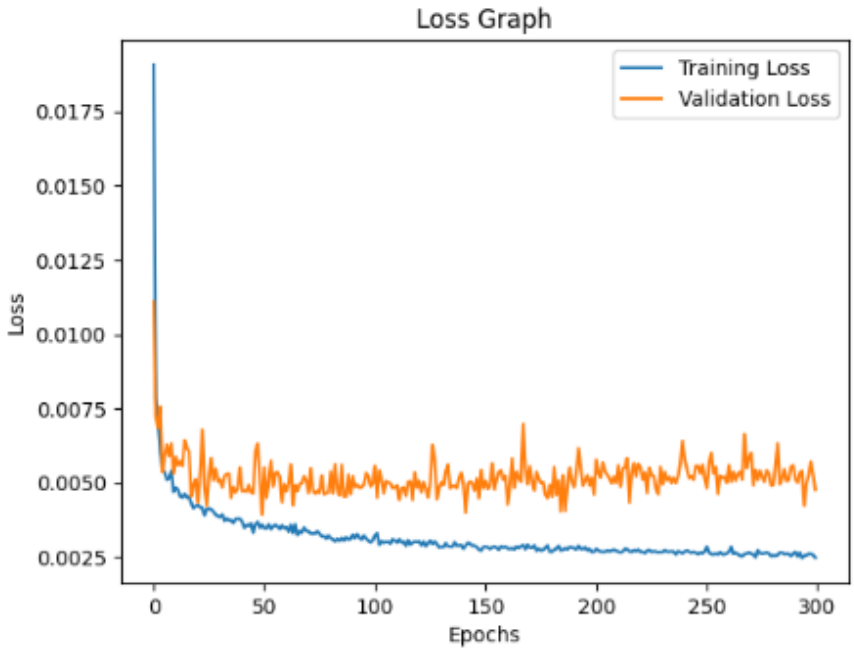


Figure 7.3a, Loss Graph of ED-LSTM

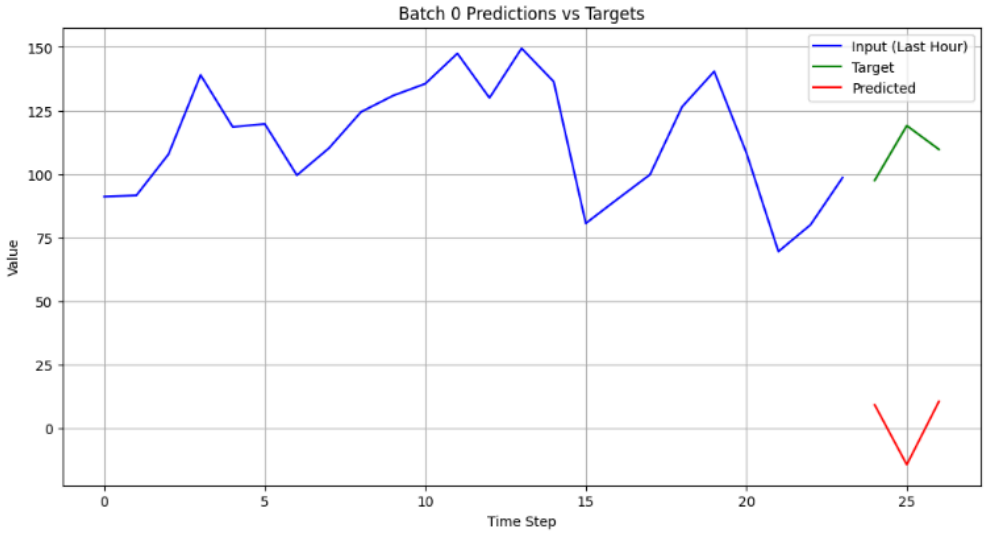


Fig 7.3b, ED-LSTM Predictions vs Targets graph

7.0 ED-BiLSTM

7.1 Implementation

Given that the ED-LSTM model was unable to actually predict well, we decided to explore an ED-BiLSTM architecture. This adjustment was motivated by the need to improve the model's ability to capture richer temporal patterns from the input sequence.

Unlike standard LSTMs, Bidirectional LSTMs (Bi-LSTMs) process data in both forward and backward directions, allowing the encoder to gather context from the entire sequence more effectively. By integrating a Bi-LSTM as the encoder, the model is better equipped to understand the full temporal structure of the input, which can lead to improved performance in downstream prediction tasks like taxi availability forecasting.

It closely follows the architecture of both the BiLSTM and ED-LSTM models, with the key difference being in how the hidden and cell states are handled. Specifically, the forward and backward hidden and cell states are combined by summing them, as shown in Lines 14–15. This fusion allows the model to incorporate information from both past and future time steps into a unified state representation, potentially enhancing its ability to capture temporal dependencies.

```
1  class BiEncoderLSTM(torch.nn.Module):
2      def __init__(self, input_dim, hidden_dim):
3          super(BiEncoderLSTM, self).__init__()
4          self.hidden_dim = hidden_dim
5          self.num_directions = 2  # Since it's bidirectional
6
7          # LSTM layer
8          self.lstm = torch.nn.LSTM(input_dim, hidden_dim, batch_first=True, bidirectional=True)
9      def forward(self, inputs):
10
11
12         inputs = inputs.to(device)
13         out, (hn, cn) = self.lstm(inputs)
14         hn_dec = hn[0] + hn[1]
15         cn_dec = cn[0] + cn[1]
16
17         hn_dec = hn_dec.unsqueeze(0)
18         cn_dec = cn_dec.unsqueeze(0)
19
20         return hn_dec, cn_dec
21
```

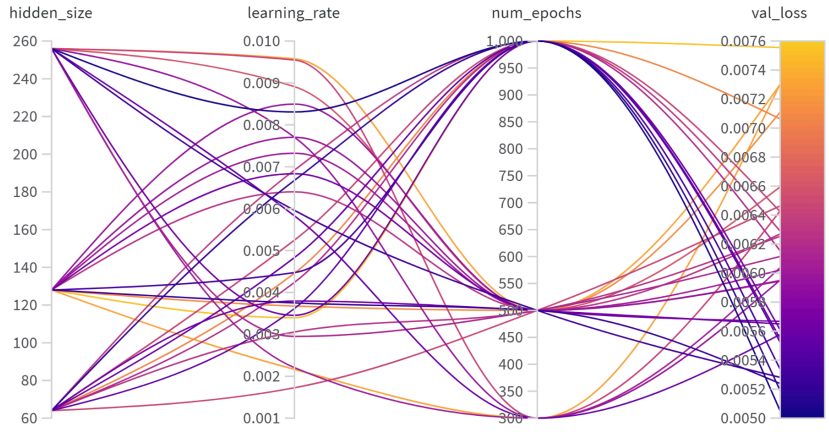
Fig 7.1a, BiEncoderLSTM class

```python
class DecoderLSTM(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(DecoderLSTM, self).__init__()


        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)

        # Implementing Prof Idea
        self.linear1 = nn.Linear(128, 1)
        self.linearsub1 = nn.Linear(128, input_size)

        self.linear2 = nn.Linear(128, 1)
        self.linearsub2 = nn.Linear(128, input_size)

        self.linear3 = nn.Linear(128, 1)

    def forward(self, x, hidden,):
        outputs = []
        decoder_input = x

        # First LSTM layer
        # print("Entering First Layer of Decoder")
        decoder_input = decoder_input.to(device)
        y1, hidden1 = self.lstm(decoder_input, hidden)
        linear_y1 = self.linear1(y1)
        linear_y1 = linear_y1.mean(dim=1, keepdim=True)
        linear_suby1 = self.linearsub1(y1)

        # print("Entering Second Layer of Decoder")
        # Second LSTM layer
        y2, hidden2 = self.lstm(linear_suby1, hidden)
        linear_y2 = self.linear2(y2)
        linear_suby2 = self.linearsub2(y2)
        linear_y2 = linear_y2.mean(dim=1, keepdim=True)

        # print("Entering Third Layer of Decoder")
        y3, hidden3 = self.lstm(linear_suby2, hidden)
        linear_y3 = self.linear3(y3)
        linear_y3 = linear_y3.mean(dim=1, keepdim=True)

        outputs.append(linear_y1.squeeze(1))  # shape: [17, 1]
        outputs.append(linear_y2.squeeze(1))
        outputs.append(linear_y3.squeeze(1))

        final_output = torch.stack(outputs, dim=1)  # [17, 3, 1]
        # print("linear_y1", linear_y1)

        # print("Decoder input shape: ", decoder_input.shape)
        # print("linear_y1: ", linear_y1.shape)
        # print("final_output: ", final_output.shape)
        return final_output

```

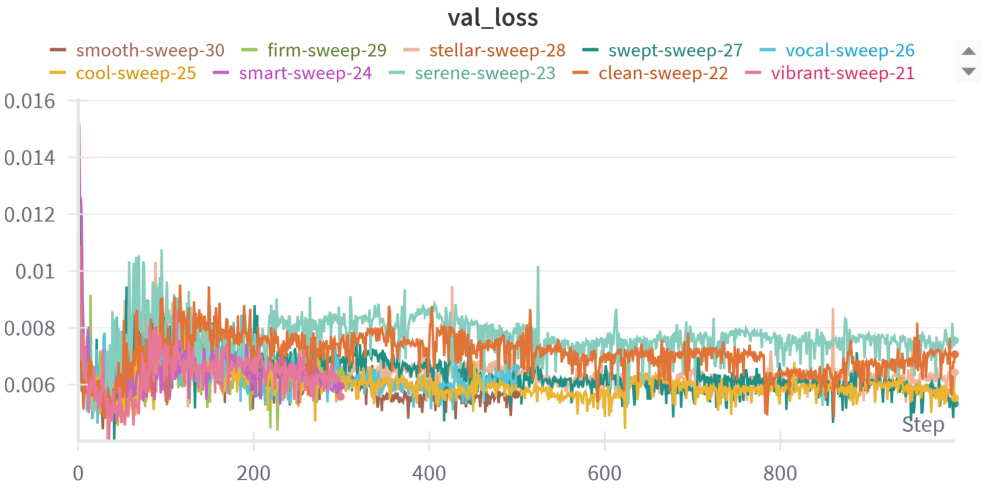
Fig 7.1b, BiDecoderLSTM class

7.2 Hyper parameters

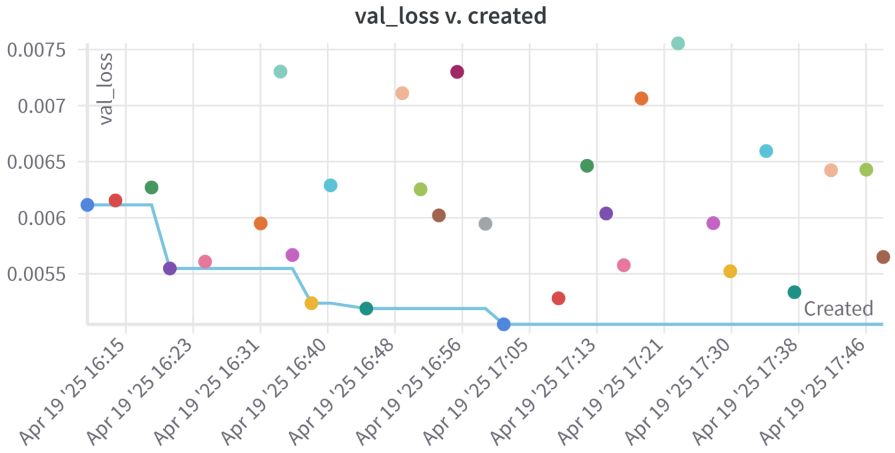
For the hyperparameters, we once again tuned the hidden_size, learning_rate and number of epoch to find the best hyper parameters. Similarly we ran it for 30 iterations.



7.2 a, Hyperparameters chosen and the resulting validation loss



7.2 b, Validation Loss curve for each run



7.2 c, Validation loss over time across different runs

If we look at the val_loss over subsequent runs, we can see that it is still very scattered, meaning our hyper parameters are having little effect on the model. But if we look at the val_loss per iteration, we notice a very important trend. We can see that the validation loss actually has a bit of U-turn to be worse consistently after around 100 epoch which suggests to us that anything after that it is overfitting. We shall therefore train our final model on around 40-50 epoch for the best results.

7.3 Results

Despite leveraging the temporal structure of both forward and backward sequences, a key advantage of the Bi-LSTM architecture, the model did not show substantial improvements over the base LSTM in validation performance. After training and validation, the ED-BiLSTM model demonstrated a consistent reduction in training loss, indicating that it effectively learned from the training data. However, the validation loss remained unstable and higher than the training loss. The model still struggles with generalization and is still overfitting. Despite this, The MAE from the test set results is 17.8007. Additionally, we should note that this only took 40-50 epoch to reach this MAE which is better and faster than the previous models.

This indicates that while the bidirectional encoding may help capture richer context, it alone may not be sufficient to address the generalization issue in this prediction task.

```
tensor(18.0119, device='cuda:0')
Normalized output[0]: [[0.2378053516149521], [0.2418629378080368], [0.25599589943885803]]
Normalized target[0]: [[0.15392456948757172], [0.10669384896755219], [0.156642884016037]]
Denormalized output[0]: [[58.32176208496094], [59.31688690185547], [62.78299331665039]]
Denormalized target[0]: [[37.75], [26.16666603088379], [38.41666793823242]]
--------------------------------------------------
tensor(16.9270, device='cuda:0')
Normalized output[0]: [[0.3157176375389099], [0.32431888580322266], [0.356317400932312]]
Normalized target[0]: [[0.2721712589263916], [0.2453279048204422], [0.22799864411354065]]
Denormalized output[0]: [[77.42974853515625], [79.5392074584961], [87.3868408203125]]
Denormalized target[0]: [[66.75], [60.16666793823242], [55.91666793823242]]
--------------------------------------------------
tensor(15.4544, device='cuda:0')
Normalized output[0]: [[0.465930700302124], [0.4487111568450928], [0.44039201736450195]]
Normalized target[0]: [[0.3615358471870422 4], [0.3611960709095001], [0.34590554237365 72]]
Denormalized output[0]: [[114.26950073242188], [110.0464096069336], [108.00614166259766]]
Denormalized target[0]: [[88.66666412353516], [88.58333587646484], [84.83333587646484]]
--------------------------------------------------
tensor(21.8281, device='cuda:0')
Normalized output[0]: [[0.5300065875053406], [0.5151795744895935], [0.491495460271 8353]]
Normalized target[0]: [[0.6744818687438965], [0.7138973474502563], [0.6965681314468384]]
Denormalized output[0]: [[129.98411560058594], [126.34779357910156], [120.53926086425781]]
Denormalized target[0]: [[165.4166717529297], [175.0833282470703], [170.8333282470703]]
--------------------------------------------------
tensor(20.9864, device='cuda:0')
...
Predicted output shape: torch.Size([17, 3, 1])
True output shape: torch.Size([17, 3, 1])
Average Validation Loss: 0.0096
Mean Absolute Error: 17.8007
```

Fig 7.3a, ED-*BiLSTM test set results, showing MAE.*

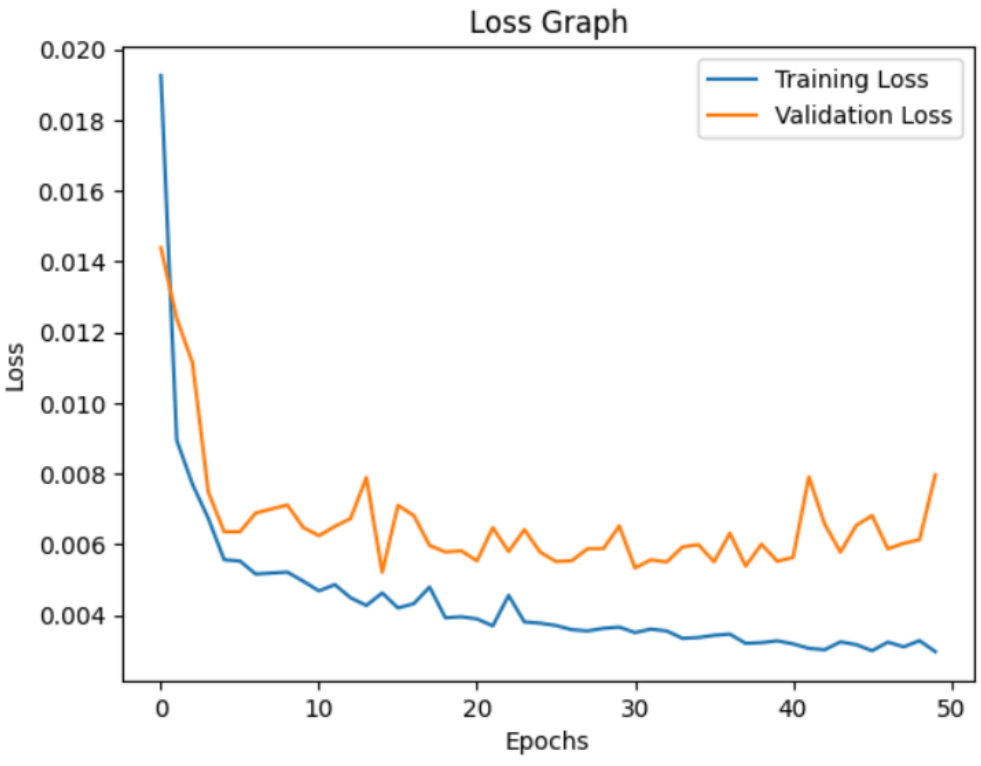


Fig 7.3a, ED-BiLSTM Loss graph

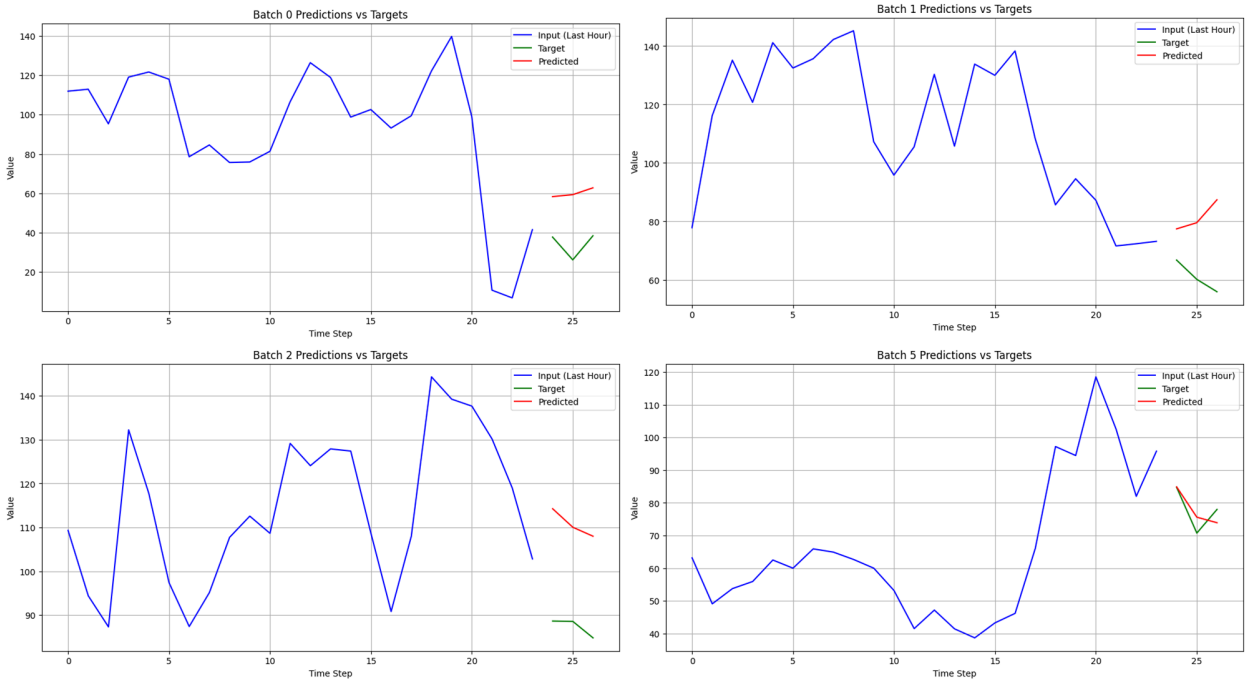


Fig 7.3b, ED-BiLSTM Predictions vs Loss Graph

Based on the graph above, it appears that the ED-BiLSTM model effectively combines the strengths of both the BiLSTM and the ED-LSTM architectures. While the BiLSTM tends to predict flat or linear trends, capturing the general level of taxi availability, it often lacks the flexibility to model non-linear patterns. On the other hand, the ED-LSTM introduces non-linearity into its predictions but may struggle with overall accuracy or stability.

The ED-BiLSTM benefits from the bidirectional structure of the BiLSTM, which allows it to understand context from both past and future time steps, while also leveraging the encoder-decoder architecture to better capture complex, dynamic patterns in the data. As a result, the ED-BiLSTM not only maintains accurate estimates of taxi availability but also adapts to non-linear trends, making it a more robust and expressive model than either the standalone BiLSTM or ED-LSTM.

8.0 Transformer

8.1 Implementation

As the LSTM models did not perform as well as we hoped, we moved on to explore transformer models. Transformers offer several advantages over LSTMs and ED-LSTMs when predicting taxi availability from time-series data. Unlike LSTMs, which process sequences step-by-step and struggle with long-term dependencies due to vanishing gradients, transformers use self-attention to learn relationships across all time steps simultaneously. This enables them to better capture how earlier events influence future taxi availability.

Moreover, transformers excel at learning inter-feature relationships such as how different TCP parameters or external features (like weather or time of day) interact over time. This makes them particularly powerful for modeling complex, multivariate time-series data like taxi availability, where both temporal trends and feature interactions matter.

Hence, we investigated to see if we could use transformers to predict the taxi availability.

```python
class TransformerModel(nn.Module):
    def __init__(self, input_size, output_size, num_heads, num_layers, hidden_dim, dropout=0.1):
        super(TransformerModel, self).__init__()

        self.embedding = nn.Linear(input_size, hidden_dim)         # input_size = 3
        self.decoder_embedding = nn.Linear(1, hidden_dim)
        self.encoder_positional_encoding = nn.Parameter(torch.rand(1, 100, hidden_dim))
        self.decoder_positional_encoding = nn.Parameter(torch.rand(1, 100, hidden_dim))  # max_seq_len=100

        self.transformer = nn.Transformer(
            d_model=hidden_dim,
            nhead=num_heads,
            num_encoder_layers=num_layers,
            num_decoder_layers=num_layers,
            dim_feedforward=hidden_dim * 2,
            dropout=dropout,
            batch_first=True
        )

        self.fc_out = nn.Linear(hidden_dim, output_size)

    def forward(self, src, tgt=None):
        src = self.embedding(src)
        src += self.encoder_positional_encoding[:, :src.size(1), :]


        if tgt is not None:
            tgt_input = self.decoder_embedding(tgt)
            tgt_input += self.decoder_positional_encoding[:, :tgt_input.size(1), :]
        else:
            print("target is using source")
            tgt_input = src

        output = self.transformer(src, tgt_input)
        output = self.fc_out(output)
        return output
```

Fig 8.1a, Transformer model class

The first step to the transformer model initialization is to increase the number of dimensional size via a Linear layer to the hidden_size (Line 5-6). This is because attention layers usually work better in richer, more expressive spaces.

The second step is to add additional positional information (Line 7-8). Unlike RNNs or LSTMs, which process sequences one step at a time and inherently know which time step comes next, transformers process all time steps in parallel. Unfortunately, this means that transformers don't know the position of each time step unless we explicitly tell them. It is necessary for the transformer to know the position because it is being fed a sequence of feature vectors , one for each time step. Without it, the transformer would treat the time steps like a bag of features, with no sense of sequence. This would completely break the model's understanding of temporal patterns.

Next, we initialized the transformer mode (Line 10-18). To model sequential dependencies in the data, we initialize a Transformer using PyTorch's **"nn.Transformer "**module. The key hyperparameters are carefully chosen to balance model capacity and computational efficiency:

- **"D_model"**: This defines the dimensionality of the embeddings processed inside the Transformer. Since our embedding will have the dimensions of the "hidden_dim", then d_model will be the same value as well.

- **"Nhead"**: It is the multi-head attention that is used to let the model focus on different parts of the input sequence simultaneously. The number of attention heads divides "d_model" into subspaces, enabling diverse attention perspectives.

- **"num_encoder_layers / num_decoder_layers"**: These parameters define the depth of the Transformer architecture. Each layer contains a self-attention mechanism and a feedforward network. Stacking multiple layers increases the model's ability to learn complex temporal patterns in the data. For simplicity, they are both set to 1.

- **dim_feedforward**: This sets the hidden layer size in the position-wise feedforward subnetwork within each Transformer layer. It is typically a multiple of "d_model". Doing so allows for nonlinear feature transformations between attention operations.

Together, these parameters configure a Transformer architecture capable of learning temporal dependencies from our time series data.

8.1.1 Forward Method

The forward method first does the embedding and positional encoding for the source, which is the input for the encoder (Line 23-24). We do these by first projecting the input features to the model's “d_model” dimension. Then we add time-step information so the model knows the order of the sequence (because transformers have no recurrence). “src.size(1)” refers to the time step, which is 24.

Next, we prepare the target value, which is the input for the decoder (Line 27-32). Typically, this input is actually the output of the transformer model. If the target (“tgt”) is provided, it is first passed through the decoder embedding layer, followed by the addition of positional encoding to capture time-step information, similar to how the source sequence is processed in the encoder. However, if the target is not provided, the source sequence (“src) is used as the target input instead. This approach ensures that the model can function properly even when a target sequence is unavailable.

Finally, we use the transformer model (Line 34-36). The Transformer model takes two inputs: the source sequence (“src”) and the target sequence (“tgt_input”). These inputs are processed by the Transformer layer, which leverages self-attention mechanisms to capture dependencies across the entire sequence. After passing through the Transformer, the output undergoes a final transformation using a fully connected layer, specifically a linear layer (“fc_out”). This layer projects the model’s output into the desired output shape, which is then returned as the final model output.

8.1.2 Training the transformer model

```python
def train_transformer(model, dataloader, val_loader, num_epochs, learning_rate, device):
    # Set the loss function and optimizer
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    model.train()
    loss_graph = []
    val_graph = []

    for epoch in range(num_epochs):
        epoch_loss = 0
        for batch_idx, (inputs, targets) in enumerate(dataloader):
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward pass
            output = model(inputs, tgt=targets)
            loss = criterion(output, targets)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            epoch_loss += loss.item()

        loss_graph.append(epoch_loss / len(dataloader))

        # Validation step
        model.eval()
        with torch.no_grad():
            val_loss = 0
            for val_inputs, val_targets in val_loader:
                val_inputs, val_targets = val_inputs.to(device), val_targets.to(device)
                val_output = model(val_inputs, tgt=val_targets)
                val_loss += criterion(val_output, val_targets).item()
            val_loss /= len(val_loader)
            val_graph.append(val_loss)
            model.train()

        # Print the loss every 50 epochs
        if epoch % 50 == 0:
            print(f"Epoch {epoch + 1}/{num_epochs}, Training Loss: {epoch_loss / len(dataloader)}, Validation Loss: {val_loss}")

    print(f"Final Epoch {epoch + 1}/{num_epochs}, Training Loss: {epoch_loss / len(dataloader)}, Validation Loss: {val_loss}")
    return loss_graph, val_graph
```

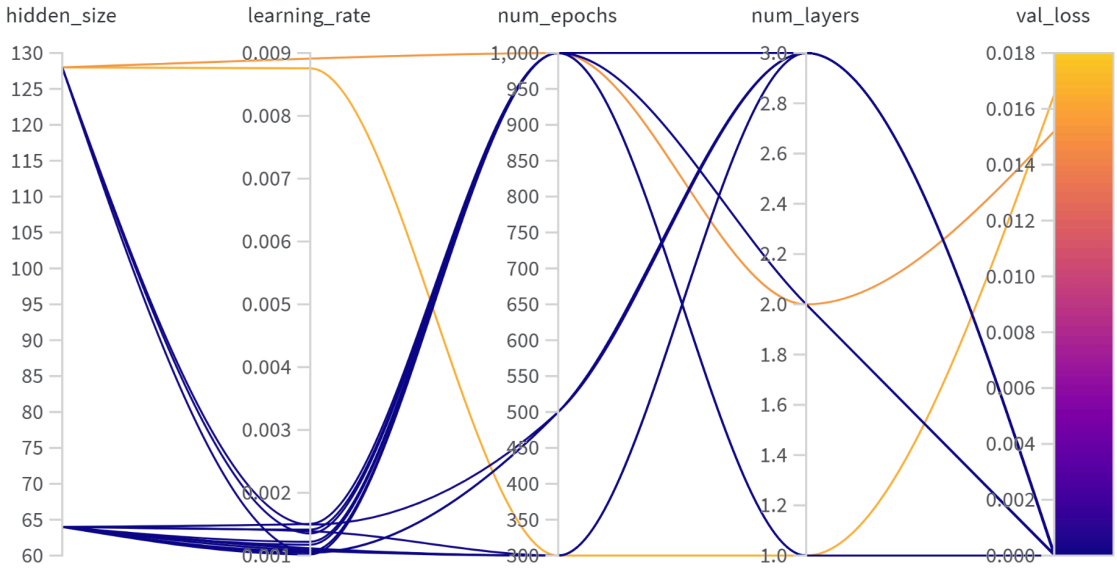
8.1.2a, Transformer training function

The model is trained by using the target sequence (“tgt”) as the input to the decoder, which is a key part of the teacher-forcing technique. This technique helps the transformer learn more effectively by guiding the decoder with true values at each time step, rather than relying on its own predictions. It’s particularly useful in time series forecasting tasks, as it accelerates convergence and reduces error propagation over long sequences.

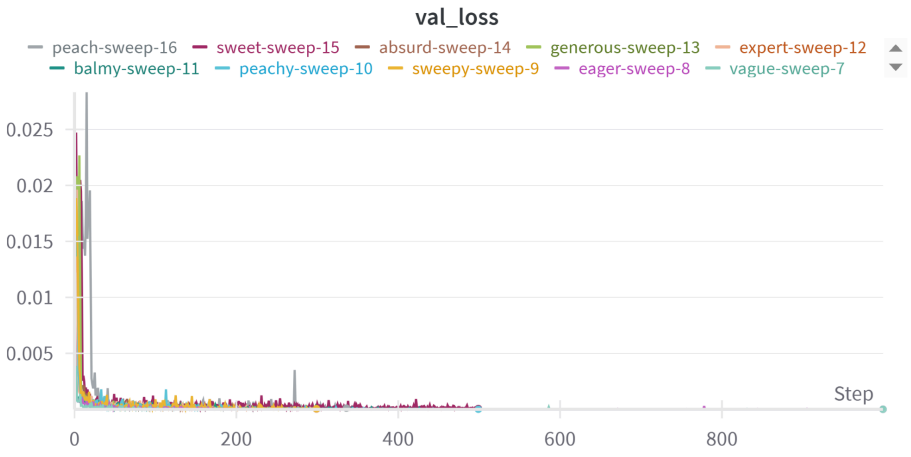
8.2 HyperParameters

We did the same hyper parameter tuning for the transformer as well by adjusting the number of layers, hidden_size, learning rate and the number of epochs.

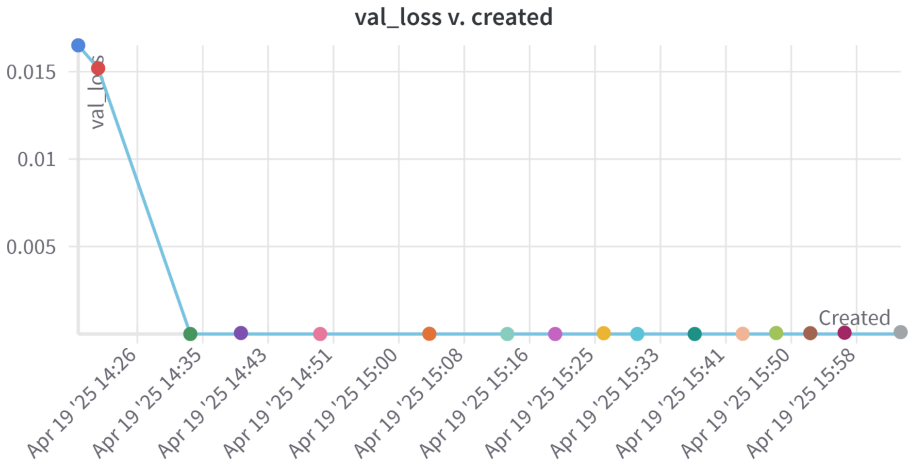
What we noticed is that there are no significant changes between different hyper parameters except for the hidden size. If the hidden size is above 64, then the model will interestingly perform worse. We hypothesize that this means that the model is probably too complex if the hidden size goes up. The results of the tuning are as shown below:



8.2 a, Hyperparameters chosen and the resulting validation loss



8.2 b, Validation Loss curve for each run



8.2c, Validation loss over time across different runs

An additional perk that we noticed when training the transformer model is that the validation_loss closely follows the training_loss while at the same time having a lower loss than all the LSTM-based models. This is great! This means that the model is likely not overfitting too much and is performing well. This likely means that this will result in a model that will give us our best results.

8.3 Results

```python
import collections
model.eval()

loss_value = 0
num_batches = 0

criterion = torch.nn.MSELoss()
mae_list = []
visualise_dataset = collections.defaultdict(list)

with torch.no_grad():
    for batch_idx, (inputs, targets) in enumerate(test_loader):


        inputs, targets = inputs.to(device), targets.to(device)  # inputs: [B, 24, 7], targets: [B, 3, 1]

        batch_size = inputs.size(0)
        tgt_input = inputs[:, -1:, 5:6]  # Feed in last input as first output

        # Autoregressive prediction
        for step in range(targets.size(1)):  # Predict 3 steps
            output = model(inputs, tgt=tgt_input)  # output: [B, current_step+1, 1]
            next_token = output[:, -1:, :]  # [B, 1, 1]
            tgt_input = torch.cat([tgt_input, next_token], dim=1)  # Grow decoder input

        predicted = tgt_input[:, 1:, :]  # [B, 3, 1]

        inputs_denorm = inputs * (data_max["Average Taxi Availability"] - data_min["Average Taxi Availability"]) + data_min["Average Taxi Availability"]
        output_denorm = predicted * (data_max["Average Taxi Availability"] - data_min["Average Taxi Availability"]) + data_min["Average Taxi Availability"]
        targets_denorm = targets * (data_max["Average Taxi Availability"] - data_min["Average Taxi Availability"]) + data_min["Average Taxi Availability"]

        visualise_dataset[batch_idx] = inputs_denorm[:,:, 5:6].cpu().numpy()
        # Add targets to visualise_dataset
        visualise_dataset[batch_idx] = np.concatenate((visualise_dataset[batch_idx], targets_denorm.cpu().numpy()), axis=1)
        visualise_dataset[batch_idx] = np.concatenate((visualise_dataset[batch_idx], output_denorm.cpu().numpy()), axis=1)

        # Compute loss on normalized data
        loss_value += criterion(output, targets)
        mae = torch.mean(torch.abs(output_denorm - targets_denorm))
        mae_list.append(mae)

        print("Normalized output[0]:", output[0].tolist())
        print("Normalized target[0]:", targets[0].tolist())
        print("Denormalized output[0]:", output_denorm[0].tolist())
        print("Denormalized target[0]:", targets_denorm[0].tolist())
        print("-" * 50)

loss_value = loss_value / (len(test_loader) - 1)
print(f'Average Validation Loss: {loss_value:.4f}')

mae = torch.mean(torch.tensor(mae_list))
print(f'Mean Absolute Error: {mae:.4f}')

for batch_idx, data in visualise_dataset.items():
    inputs = data[0, :24]
    targets = data[0, 24:27]    # 3 timesteps: ground truth
    predictions = data[0, 27:]  # 3 timesteps: model predictions
    plt.figure(figsize=(12, 6))
    # Input: plotted from timestep 0 to 24
    plt.plot(range(24), inputs[:, -1], label='Input (Last Hour)', color='blue')
    # Targets: plotted starting from timestep 24 to 26
    plt.plot(range(24, 27), targets[:, -1], label='Target', color='green')
    # Predictions: plotted from timestep 27 to 29
    plt.plot(range(24, 27), predictions[:, -1], label='Predicted', color='red')
    plt.title(f'Batch {batch_idx} Predictions vs Targets')
    plt.xlabel('Time Step')
    plt.ylabel('Value')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Fig 8.3a, Evaluation code of the transformer via the test set

During evaluation, the model is set to evaluation mode, disabling training-specific behaviors like dropout and batch normalization. The evaluation follows an autoregressive prediction strategy, where the model predicts one future step at a time and feeds each predicted output back into itself as input for the next step. The decoder is initialized using the last known value from the input sequence, which represents the most recent taxi availability reading before the prediction window. This grounding helps the model begin predictions from a realistic reference point. The process is repeated for the full prediction horizon (three steps in this case), allowing the model

to generate a sequence of future values based on its own prior predictions. The MAE of this model when used to predict the test data samples is 15.3445, the lowest amongst all the models.

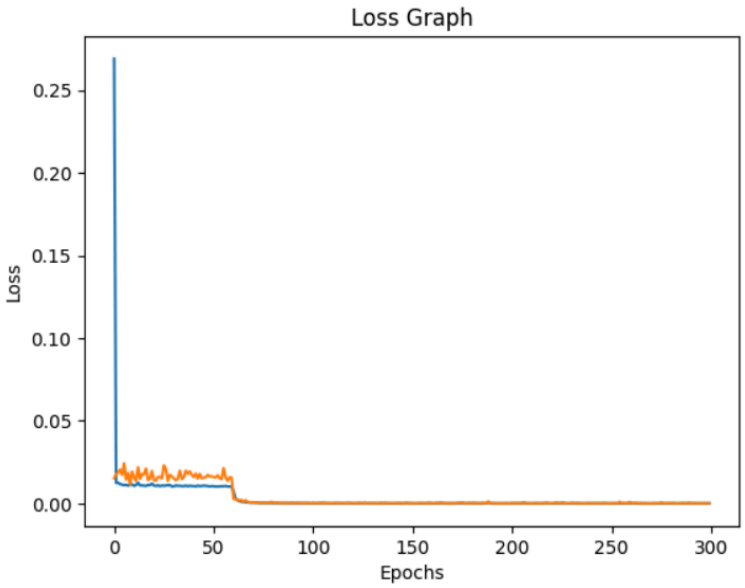


Figure 8.3b, Transformer Loss graph

```
Normalized output[0]: [[0.40957438945770264], [0.412779867649078 37], [0.4178571403026581]]
Normalized target[0]: [[0.3798844516277313], [0.33978933095932007], [0.35032278299331665]]
Denormalized output[0]: [[99.8270263671875], [101.21166229248047], [102.47946166992188]]
Denormalized target[0]: [[93.16666412353516], [83.33333587646484], [85.91666412353516]]
--------------------------------------------------
Normalized output[0]: [[0.5487358570098877], [0.5530897378921509], [0.5509904623031616]]
Normalized target[0]: [[0.4216785430908203], [0.3261977434158325], [0.3343527019023895]]
Denormalized output[0]: [[135.8440399169922], [135.48583984375], [135.1304168701172]]
Denormalized target[0]: [[103.41666412353516], [80.0], [82.0]]
--------------------------------------------------
Normalized output[0]: [[0.42268258333206177], [0.4248197674751282], [0.4285907447338104]]
Normalized target[0]: [[0.49167516827583313], [0.4566768705844879], [0.3720693290233612]]
Denormalized output[0]: [[103.1781997680664], [104.12602996826172], [105.11187744140625]]
Denormalized target[0]: [[120.58333587646484], [112.0], [91.25]]
--------------------------------------------------
Normalized output[0]: [[0.47260674834251404], [0.47448796033859253], [0.4744149446487427]]
Normalized target[0]: [[0.4243968725204468], [0.4362894892692566], [0.5878355503082275]]
Denormalized output[0]: [[116.24095916748047], [116.25716400146484], [116.35026550292969]]
Denormalized target[0]: [[104.08333587646484], [107.0], [144.1666717529297]]
--------------------------------------------------
Normalized output[0]: [[0.3881867527961731], [0.3914094567298889], [0.3993828296661377]]
Normalized target[0]: [[0.3639143705368042], [0.3564389944076538], [0.3605164587497711]]
Denormalized output[0]: [[94.12964630126953], [95.98518371582031], [97.94863891601562]]
Denormalized target[0]: [[89.25], [87.41666412353516], [88.41666412353516]]
--------------------------------------------------
Predicted output shape: torch.Size([17, 3, 1])
True output shape: torch.Size([17, 3, 1])
Average Validation Loss: 0.0063
Mean Absolute Error: 15.3445
```

Figure 8.3c, Transformer test set results, showing MAE.

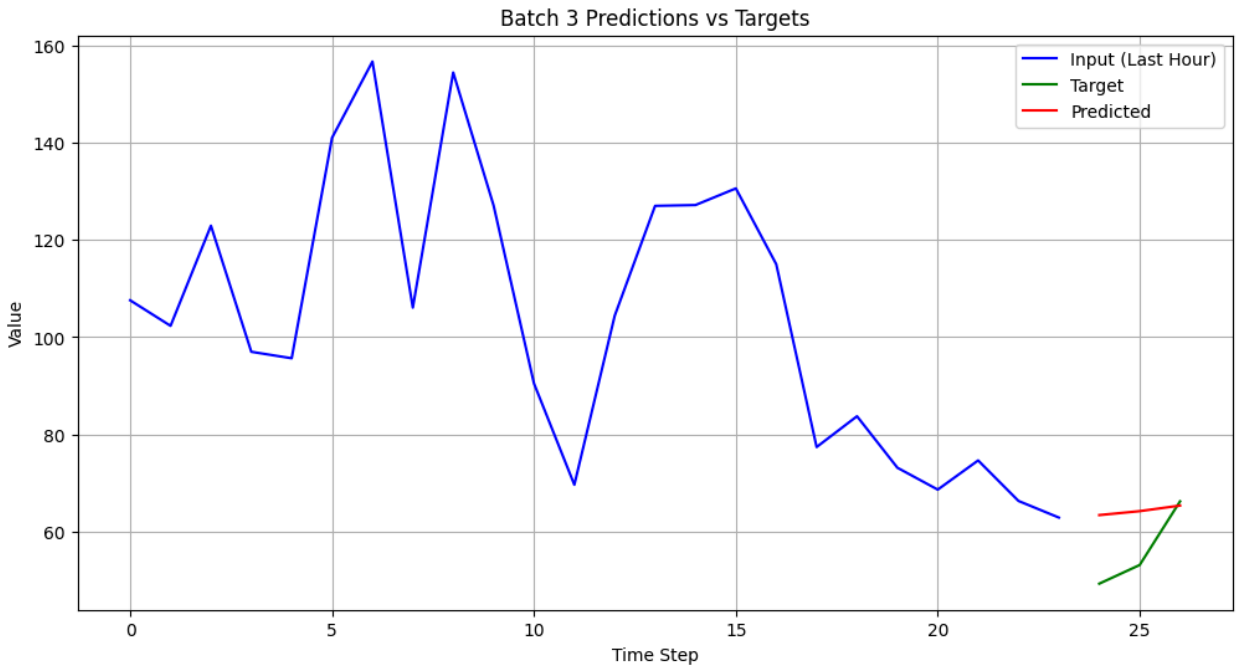


Fig 8.3d, Predictions vs Targets graph

The model's predictions closely align with the target values, indicating that it performs better than the other models. This is further supported by the loss graph, MAE results on the test set, and the Predictions vs Targets visualization, all of which are consistent and reinforce the model's effectiveness.

9.0 Model Malfunctioning

9.1 Not normalizing the data

Initially, normalization was not applied to the data. As a result, the model consistently outputs the same predicted values across different inputs, indicating a failure to learn meaningful patterns. Upon closer inspection, it was observed that the input features varied drastically in scale, and their magnitudes differed significantly from the target output range.

A mismatch in feature scales can cause several issues during training. Large or unbalanced input values may lead to activation saturation especially in functions like “tanh” (found in LSTMs) or Sigmoid. This could result in vanishing gradients that prevent effective weight updates. Additionally, inconsistent feature scales make it difficult for the model to assign appropriate importance to each input, hindering the optimizer’s ability to learn meaningful patterns. As a result, the model often converges to a trivial solution, such as consistently predicting a constant value.

Despite the model suffering from vanishing gradients due to unnormalized input data, the loss function may still appear to decrease slightly during training. This is typically because the model converges to a trivial solution—such as outputting a constant value close to the mean of the target—which minimizes loss to some extent. However, in reality, the model is not capturing any meaningful input-output relationships.

Normalizing the data before training resolves this issue, allowing the model to learn meaningful patterns and produce varied predictions.

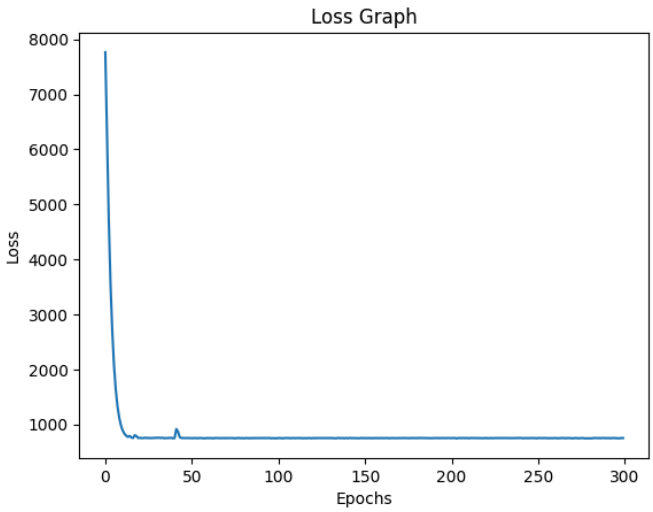


Fig 9.1a, The model is still able to reduce the loss significantly before plateauing, as it converges to a trivial solution.

```
Normalized output[0]: [[90.34992980957031], [85.23204040527344], [86.01207733154297]]
Normalized target[0]: [[126.0], [143.0], [176.0]]
--------------------------------------------------
Normalized output[0]: [[90.34992980957031], [85.23204040527344], [86.01207733154297]]
Normalized target[0]: [[96.0], [108.0], [97.0]]
--------------------------------------------------
Normalized output[0]: [[90.34992980957031], [85.23204040527344], [86.01207733154297]]
Normalized target[0]: [[92.0], [103.0], [107.0]]
--------------------------------------------------
Normalized output[0]: [[90.34992980957031], [85.23204040527344], [86.01207733154297]]
Normalized target[0]: [[104.0], [127.0], [80.0]]
--------------------------------------------------
Normalized output[0]: [[90.34992980957031], [85.23204040527344], [86.01207733154297]]
Normalized target[0]: [[47.0], [34.0], [46.0]]
--------------------------------------------------
Normalized output[0]: [[90.34992980957031], [85.23204040527344], [86.01207733154297]]
Normalized target[0]: [[95.0], [68.0], [75.0]]
--------------------------------------------------
Predicted output shape: torch.Size([17, 3, 1])
True output shape: torch.Size([17, 3, 1])
Average Validation Loss: 1077.6472
```

Fig 9.1b, During evaluation, the model consistently outputs the same value regardless of the input, indicating it has converged to a trivial solution.

9.2 Incorrect Gradient Reset and Missing Tensor Reshape

In the early stages of the project, “optimizer.zero_grad()”was mistakenly placed outside the training loop over batches, causing gradients to accumulate across batches instead of being reset after each one. This led to incorrect weight updates and unstable training behavior. The issue was resolved by correctly placing “optimizer.zero_grad()” inside the inner batch loop, ensuring gradients are cleared before each backward pass.

```python
def train(model, dataloader, num_epochs, learning_rate, device):
    # Set the loss function and optimizer
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    model.train()  # Set the model to training mode
    loss_graph = []

    for epoch in range(num_epochs):
        epoch_loss = 0.0
        hidden_state, cell_state = None, None  # Reset for each epoch
        optimizer.zero_grad()

        for batch_idx, (inputs, targets) in enumerate(dataloader):
            inputs, targets = inputs.to(device), targets.to(device)
            output, hidden_state, cell_state = model(inputs, hidden_state, cell_state)
            output = output.unsqueeze(-1)

            # Compute loss
            loss = criterion(output, targets)
            loss.backward()
            optimizer.step()

            hidden_state = hidden_state.detach()
            cell_state = cell_state.detach()

            epoch_loss += loss.item()
        avg_loss = epoch_loss / len(dataloader)
        loss_graph.append(avg_loss)

        if epoch % 50 == 0 or epoch == num_epochs - 1:
            print(f'Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.6f}')

    return loss_graph
```

Fig 9.2a, Optimizer.zero_grad() placed outside of the dataloader for loop instead, causing incorrect weight updates.

```
Epoch 1/300, Loss: 0.050115
Epoch 51/300, Loss: 0.014688
Epoch 101/300, Loss: 0.013347
Epoch 151/300, Loss: 0.015498
Epoch 201/300, Loss: 0.051929
Epoch 251/300, Loss: 0.026197
Epoch 300/300, Loss: 0.029949
```

Fig 9.2b, Although the loss outputs every 50 epochs suggest that the loss may be decreasing, it is misleading.

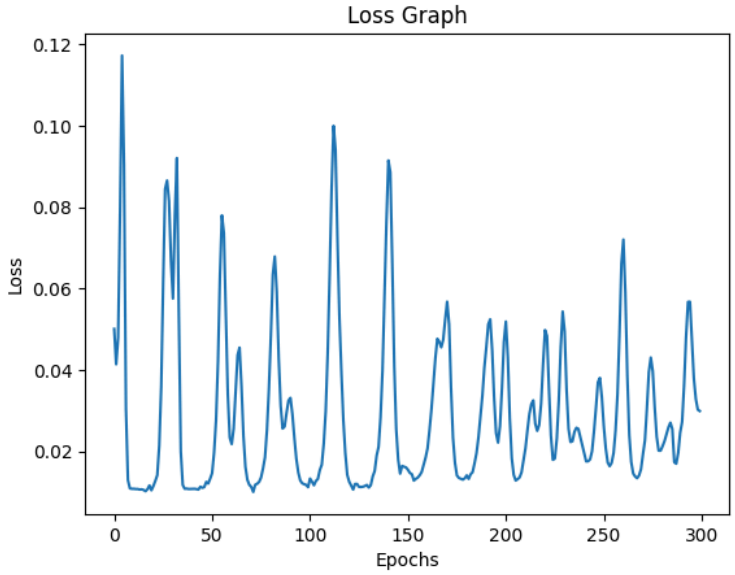


Fig 9.2c, The loss graph appears unstable, indicating that the model did not train properly or failed to converge.

9.3 Wrong output data shape

We realised later on that there was an error in the output shape for our Bi LSTM model as shown below:

```
Normalized output[0]: [[0.4705219268798828, 0.47051459550857544, 0.47052156925201416]]
Normalized target[0]: [[0.4243968725204468], [0.4362894892692566], [0.5878355503082275]]
Denormalized output[0]: [[115.39550018310547], [115.39370727539062], [115.39541625976562]]
Denormalized target[0]: [[104.08333587646484], [107.0], [144.1666717529297]]
```

Fig 9.3a, Error in output shape for BiLSTM

Due to the way we were unsqueezing our output data, it was resulting in a single [[x, x, x]] tensor like this instead of [[x],[x],[x]]. This resulted in wrong calculation for the loss as the MSE would be wrongly calculated, the same and similarly with the MAE as well. We managed to identify this and it is fixed in the final version of the Bi LSTM model.

10.0 Exploration of Alternative Models

In exploring other works for alternatives to our predictive models for short-term taxi availability, we drew methodological inspiration from *Vanichrujee et al. [1]*, who proposed an ensemble approach for taxi demand forecasting by combining Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), and eXtreme Gradient Boosting (XGBoost) models. Their framework dynamically adjusted the weight of each model in the ensemble based on its recent performance over a sliding window of past time steps, using Symmetric Mean Absolute Percentage Error (sMAPE) as the evaluation metric.

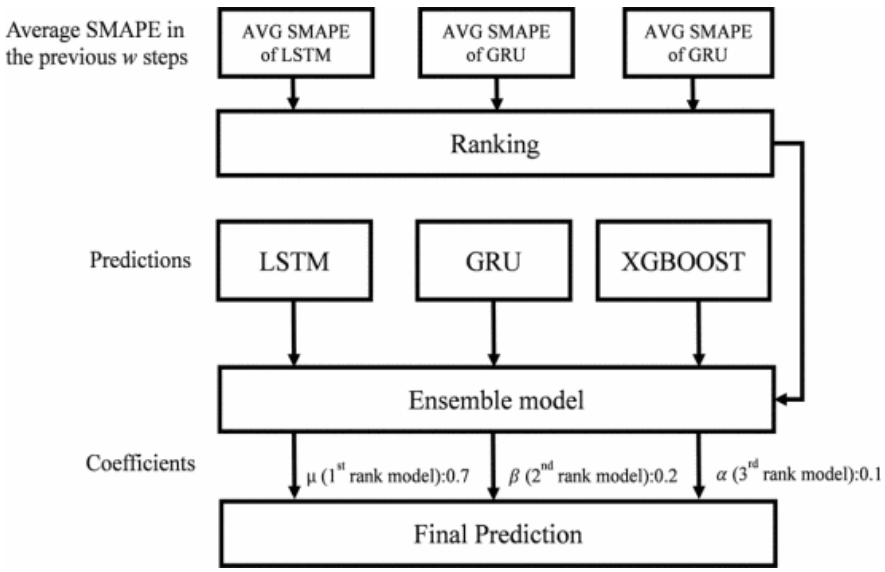


Fig 10.1a, Architecture of reference paper ensemble model

Building upon this ensemble concept, we designed a stacked ensemble model comprising two base learners: Random Forest (RandomForestRegressor from Scikit-Learn) and XGBoost

(XGBRegressor) which were selected for their complementary strengths in handling nonlinear relationships, robustness, and feature importance interpretability. A third model, also based on XGBoost, was employed as a meta-learner to combine the predictions of the base models through supervised stacking.

The model was trained on the same engineered feature set as our LSTM models: temporal variables (hour of day, day of week, weekend flag), environmental conditions (temperature, humidity, rainfall) while including a multiple lagged versions (1 to 3 hours) of key input features such as taxi availability and weather. Multi-step prediction targets were constructed to forecast taxi availability for the next 1, 2, and 3 hours (denoted as t+1, t+2, and t+3). These targets were created by shifting the time series using negative offsets to reflect future values. Input features were normalized using MinMaxScaler, and the dataset was split into 80% training, 10% validation, and 10% testing sets.

```
# Load data
df = pd.read_csv("merged_file_with_mean.csv")

# Time-based features
df["DateTime"] = pd.to_datetime(df["DateTime"])
df["IsWeekend"] = (df["DateTime"].dt.weekday >= 5).astype(int)
df["Hour"] = df["DateTime"].dt.hour + 1  # 1 to 24
df["DayOfWeek"] = df["DateTime"].dt.dayofweek

# Drop unnecessary columns
df = df.drop(columns=["DateTime", "stationId", "Coordinates[]", "Group"])

# Add lag features for past 3 hours
lag_features = ["Average Taxi Availability", "Taxi Available throughout SG", "temp_value"
,"humidity_value"]
for lag in range(1, 4):
    for col in lag_features:
        df[f"{col}_lag{lag}"] = df[col].shift(lag)

# Create multi-step ahead targets
df["target_t1"] = df["Average Taxi Availability"].shift(-1)
df["target_t2"] = df["Average Taxi Availability"].shift(-2)
df["target_t3"] = df["Average Taxi Availability"].shift(-3)

# Drop NaNs caused by shift (front and back)
df = df.dropna().reset_index(drop=True)

# Split features and targets
target_cols = ["target_t1", "target_t2", "target_t3"]
y = df[target_cols].values
X = df.drop(columns=target_cols).values

# Normalize features
scaler_X = MinMaxScaler()
X_scaled = scaler_X.fit_transform(X)

# Train/Validation/Test split (80/10/10)
random_state = 23
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.2,
random_state=random_state)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
random_state=random_state)

# Print diagnostics
print("Feature matrix shape:", X.shape)
print("Target matrix shape:", y.shape)
print("Train size:", X_train.shape[0])
print("Validation size:", X_val.shape[0])
print("Test size:", X_test.shape[0])
```

Figure 10.1b, Implementation of Data preprocessing for ensemble model

To generate robust training inputs for the meta-learner, out-of-fold (OOF) predictions from each base model were collected using 5-fold cross-validation. The meta-learner was then trained on these OOF predictions to learn an optimal combination of the base models' outputs. On the held out test set, the final ensemble model achieved a Mean Absolute Error (MAE) of approximately **10.5**, outperforming our previously implemented LSTM-based models which had MAEs in the range of **20**. This result suggests that a tree-based ensemble approach combined with structured feature engineering and multi-output regression could possibly yield superior accuracy and faster training for short-term taxi availability forecasting.

11.0 Final Comparison

Below we present the final comparison between our 5 models against the state of the art ensemble model.

	Training Loss*	**Validation Loss***	**MAE***
Base LSTM	~0.006982	~0.0065	~16.5039
ED_LSTM	~0.002622	~0.1654	~43.7470
BI_LSTM	~0.001146	~0.0023	~19.5155
BI_ED_LSTM	~0.000174	~0.0078	~17.5449
Transformer	~0.000087	~0.000071	~15.4335
Ensemble (State of the Art)	~0.0035	~0.0036	~10.4479

* Values are an approximation and average of multiple runs

It is no surprise that the Transformer model outperformed all the recurrent neural network models we have built, with an MAE of 15.4335. Its attention mechanism allows the model to focus on the most relevant parts of the input sequence, capturing long-range dependencies more effectively than our traditional recurrent models. In addition, the use of teacher forcing during training helped the Transformer converge faster and learn more effectively. By providing the correct previous output during training (rather than relying on its own predictions), the model was able to stay closer to the ground truth distribution and avoid compounding errors across the predicted sequence.

However, this does not mean the other models performed poorly. The base LSTM achieved a MAE of 16.5039, which shows that even a relatively simple architecture can produce reasonable predictions.

Interestingly, the ED-BiLSTM—designed to leverage the strengths of both the encoder-decoder architecture and bidirectional LSTM—also performed well, achieving an MAE of 17.5449. Notably, it outperformed both the standalone BiLSTM and ED-LSTM models, suggesting that hybrid architectures can effectively balance sequence modeling with contextual understanding from both directions. However, its performance still fell slightly behind the simpler base LSTM, which may indicate that the added complexity of the ED-BiLSTM was not fully beneficial given the characteristics of this dataset.

When comparing the top three models, Transformer, LSTM, and ED-BiLSTM, with the ensemble state-of-the-art model, the difference in MAE is relatively small, ranging only between 5 to 7 points. This indicates that these models demonstrated strong predictive performance and came reasonably close to the benchmark set by the ensemble. Despite their varying levels of complexity, all three models were able to capture meaningful patterns in the data, making them viable alternatives to the more complex state-of-the-art approach.

11.1 Possible explanation for poor performance

Here we discuss some possible reasons for why our model did not perform as well as the state of the art models or why the models were not able to achieve higher accuracy.

11.1.1 More than just temporal features:

We focused mainly on LSTM which focuses more on temporal features. However, this is actually not entirely factual. Factors like weather are actually not technically related to time but rather whether there is rain or not. This is actually better taken into account by the state of the art model which might be why it is resulting in much better results.

11.1.2 Area Chosen for Prediction:

We only realised after the fact that part of the area we chose is actually the airport. This is significant and will result in increased complexity as airports being included would mean that airport arrivals and departures then becomes an additional significant factor in the taxi availability. Additionally, we might even expect taxis to be highly available in our area throughout the day which might not be true for instance for heartlands. This is an issue as in order to generalise our model for more area, we chose not to have airport specific features be included. This would therefore result in our model performing potentially worse.

Another issue with our area chosen being an issue is that it was huge. It actually encompasses both a heartland area (Bedok) and also the airport. This is an issue as those two areas actually have significantly different taxi availability values. There is also a possibility that 90% of the taxis are actually located at the airport which skews the prediction.

However, if we decide to narrow down to just one single area (E.g SUTD) then the size is too small and the results are too erratic as shown below:

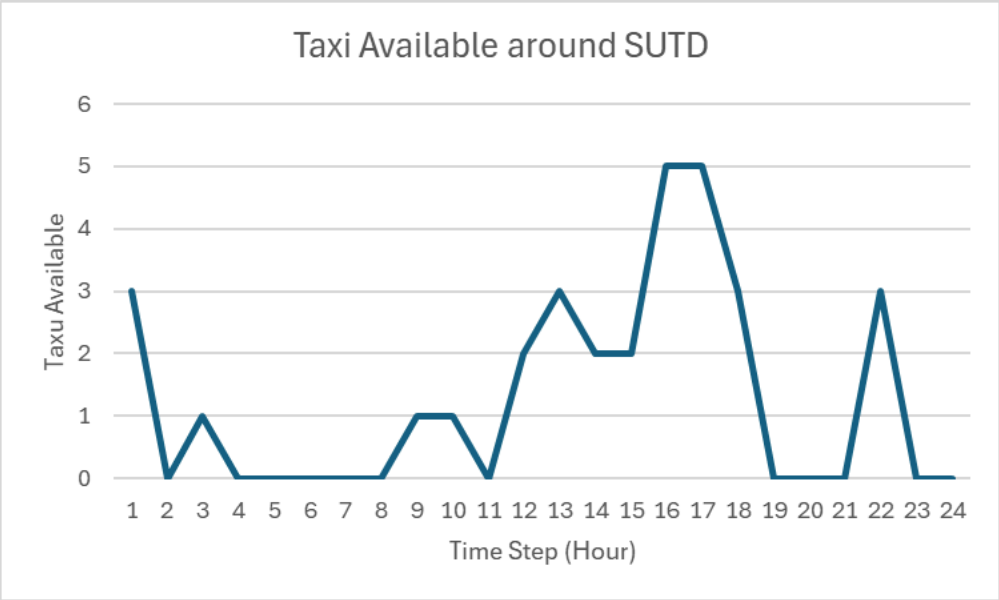


Fig 11.1.2a, Taxi available around SUTD

Ultimately there should be an ideal size that will be a perfect balance that is able to capture trends while still small enough that it is able to give a good prediction on the location of the taxis.

12.0 Reference

[1] U. Vanichrujee, T. Horanont, W. Pattara-atikom, T. Theeramunkong and T. Shinozaki, "Taxi Demand Prediction using Ensemble Model Based on RNNs and XGBoost," *2018 International Conference on Embedded Systems and Intelligent Technology & International Conference on Information and Communication Technology for Embedded Systems (ICESIT-ICICTES)*, Khon Kaen, Thailand, 2018, pp. 1–6, doi: 10.1109/ICESIT-ICICTES.2018.8442063.