CSE 151B Project Final Report

Annie Fan, Yunyi Huang, Zixin Ma, Zhenjian Wang
c7fan@ucsd.edu
yuh022@ucsd.edu
zima@ucsd.edu
zhw032@ucsd.edu
github repo: https://github.com/ZixinMa27/DL-project

1 Task Description and Background

1.1 Problem A

In this competition, our task is to build a predictive model using deep learning for forecasting autonomous vehicles motions. Our model takes in the observation data from the previous 5 seconds and predicts the positions of a tracked AV in the next 6 seconds. The task is important because autonomous vehicles are shaping the future of transportation but they are still in constant development and improvement. It is critical for autonomous vehicles to correctly access the states of themselves and the environment around them, including other cars or pedestrians. Inaccurate detection of these states can cause accidents, resulting in damages, injuries, and even casualties. In practice, correct functioning and detection of AVs' environment can significantly improve overall driving safety and traffic efficiency. For example, when people travel to long distance place, we will not worry about the drowsy driving problem. Another example is that we will not worry about drunk driving problem for the young teenagers. Therefore, our task is important because accurate forecasting of AVs motion can improve their awareness of themselves and their environment's future trajectories.

1.2 Problem B

There are many previous experiments of using deep learning models for motion prediction and forecasting. For example, Mandal et al. [1] experimented with multiple architetures EfficientNet and ResNet for the task of autonomou vehicles motion prediction. They used the Lyft dataset, which contained high-definition scene data and high-definition scene data collected from more than 1000 hours of data along a single route. They compared their implemented models by plotting and comparing average training and validation loss. The task of Mandal et al. is very similar to our task in this competition but they used a very different dataset for motion prediction. Therefore, their approach might not be directly applicable to our task, but their paper provides many insights of how to compare different architectures of the same model as well as between different models. In another paper, Martinez et al. [3] used recurrent neural networks to predict short-term human motions. Their dataset was motion capture data from humans performing multiple actions and they pre-processed the dataset using global translation and rotation. To address the time dependent sequentially, they employed the Seq2Seq architectures with a encoder-decoder structure. Although their task was about human instead of autonomous vehicles, the nature of their data and task is very relatable to our task and therefore their approach provides helpful directions and insights about how we can approach our task.

1.3 Problem C

In mathematical language, trajectories sampled at 10HZ rate, which captures the coordinate every 0.1s. The input in this task is positions from the previous 5 seconds (50 time steps) from the observation data plus our generated features represented in a multidimensional tensor. The output is the predicted trajectories for the next 6 seconds (60 time steps) in a multidimensional tensor. We obtained our loss by calculating the mean squared error between our predictions and the ground truth:

$$Loss = \frac{1}{n}\sum(Y_{pred} - Y)^2$$

Our learning task was to optimize over all input and time steps to find a set of parameters θ that minimize the training loss:

$$\theta^* = \arg\min_{\theta} Loss = \arg\min_{\theta} \frac{1}{n}\sum(Y_{pred} - Y)^2$$

We think our models can potentially be applied to the general time series forecasting tasks because our data and time-series data share the same nature and our prediction task is very similar to time-series prediction as well.

2 Exploratory Data Analysis

2.1 Problem A

2.1.1 Training and test data size

Combined all cities:

- The training set length is 203816
- The training data input shape:(203816, 50, 2)
- The train data output shape:(203816, 60, 2)
- the testing set length is 29843
- The test data input shape:(29843, 50, 2)

Austin Dataset:

- Austin training set length is 43041. The train data input shape is (43041, 50, 2) and the train data output shape is (43041, 60, 2).
- Austin testing set length is 6325 and the test data input shape is (6325, 50, 2).

Miami Dataset:

- Miami training set length is 55029. The train data input shape is (55029, 50, 2) and the train data output shape is (55029, 60, 2).
- Miami testing set length is 7971 and the test data input shape is (7971, 50, 2).

Pittsburgh Dataset:

- Pittsburgh training set length is 43544. The train data input shape is (43544, 50, 2) and the train data output shape is (43544, 60, 2).
- Pittsburgh testing set length is 6361 and the test data input shape is (6361, 50, 2).

Dearborn Dataset:

- Dearborn training set length is 24465. The train data input shape is (24465, 50, 2) and the train data output shape is (24465, 60, 2).
- Dearborn testing set length is 3671 and the test data input shape is (3671, 50, 2).

Washington DC Dataset:

- Washington DC training set length is 25744. The train data input shape is (25744, 50, 2) and the train data output shape is (25744, 60, 2).
- Washington DC testing set length is 3829 and the test data input shape is (3829, 50, 2).

Palo Alto Dataset:

- Palo Alto training set length is 11993. The train data input shape is (11993, 50, 2) and the train data output shape is (11993, 60, 2).
- Palo Alto testing set length is 1686 and the test data input shape is (1686, 50, 2).

2.1.2 Dimension and meaning of input/output data

Both input and output are three dimensional. The first dimension is the number of observations. The second dimension is the number of timestep: input data has 50 time steps and output data has 60 time steps. The third dimension is 2 which represents x and y coordinates.

2.1.3 Visualization of one data sample

Below is the visualization of one data sample. The blue points are the input data and the orange points are the output data. In this data sample, we see a linear trajectory.

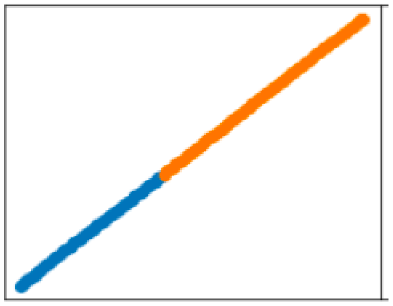
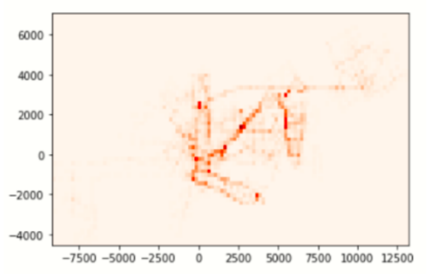


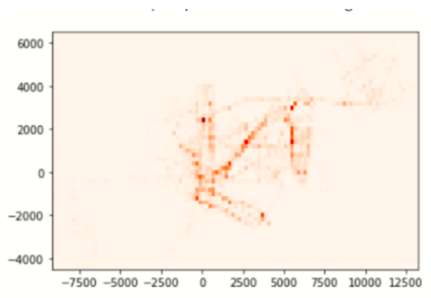
Figure 1: Visualization of One Data Sample

2.2 Problem B

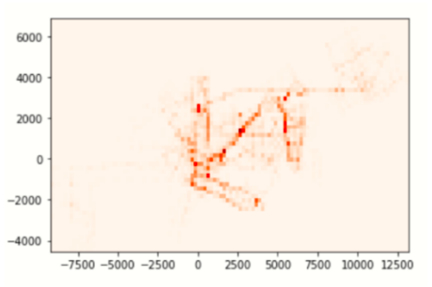
We performed some exploratory data analysis to explore the distribution of our dataset. First, we used heatmaps to visualize the distribution of input and output positions for all agents. The distribution of the input positions in training data and test data looks similar. In the following plots, the x-axis represents x coordinates of the cars' position and the y-axis represents y coordinates of the position of the cars. The distribution shows the trajectory of the carswhere deeper color represents more cars are on this trajectory.



(a) Distribution of input positions in training data



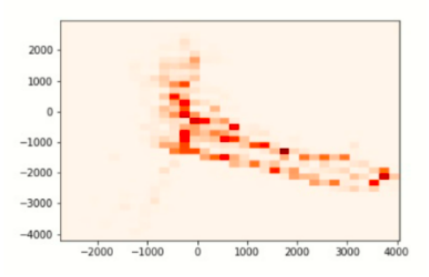
(b) Distribution of input positions in testing data



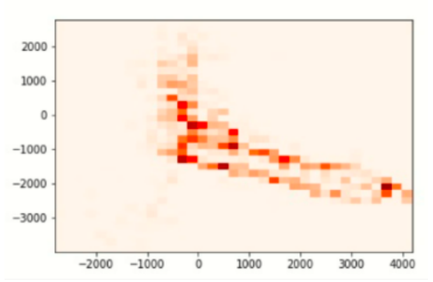
(c) Distribution of output positions in training data

Figure 2: Distribution of input and output positions all agents

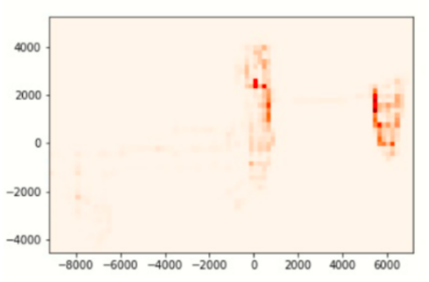
Next, we used heatmaps to visualize the input and output positions in each city. The following plots showed that different cities had very different trajectory patterns and therefore they may had different behaviors.



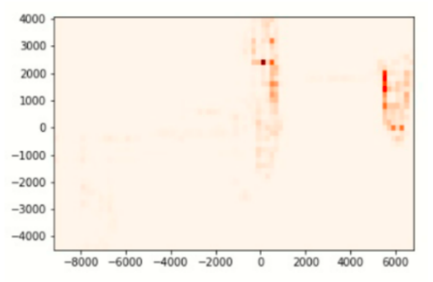
(a) Distribution of input positions Austin training data



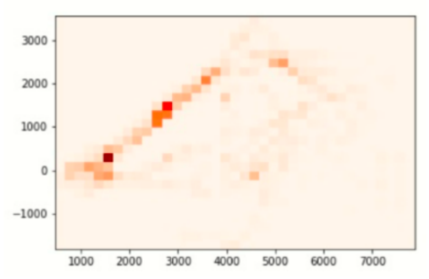
(b) Distribution of input positions Austin testing data



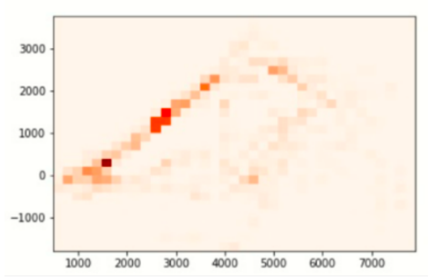
(c) Distribution of input positions Miami training data



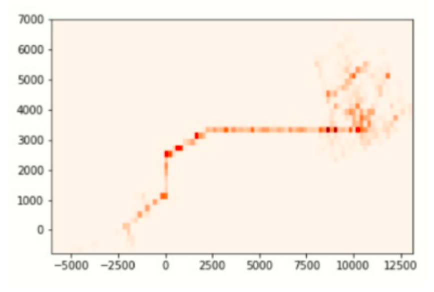
(d) Distribution of input positions Miami testing data

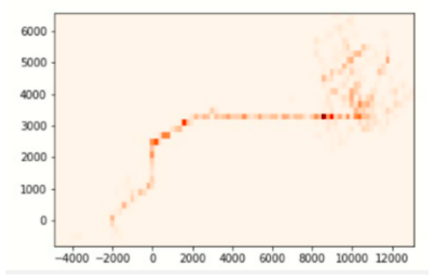


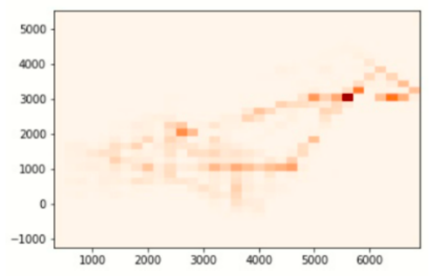
(e) Distribution of input positions Pittsburgh training data

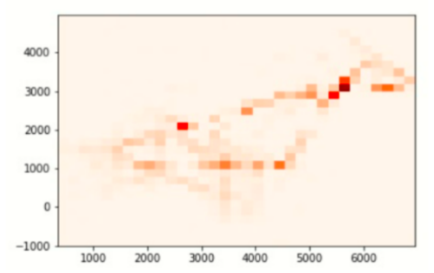


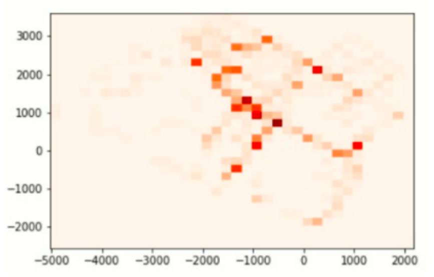
(f) Distribution of input positions Pittsburgh testing data


(g) Distribution of input positions Dearborn training data


(h) Distribution of input positions Dearborn testing data


(i) Distribution of input positions Washington DC training data


(j) Distribution of input positions Washington DC testing data


(k) Distribution of input positions Palo Alto training data

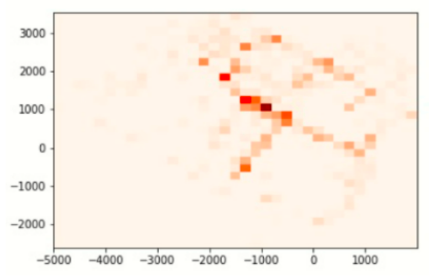
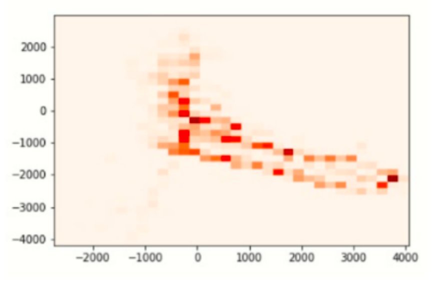
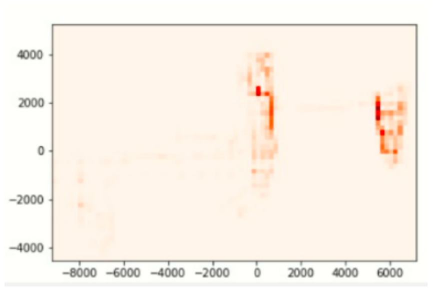

(l) Distribution of input positions Palo Alto testing data

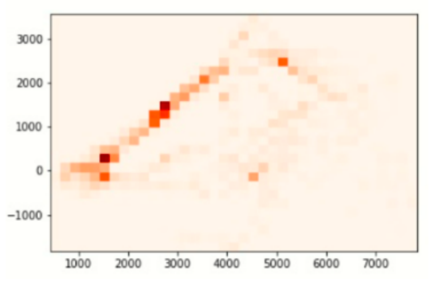
Figure 3: Distribution of input positions in each city



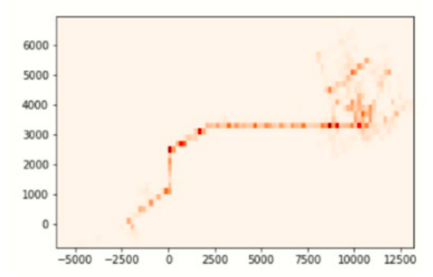
(a) Distribution of output positions Austin training data



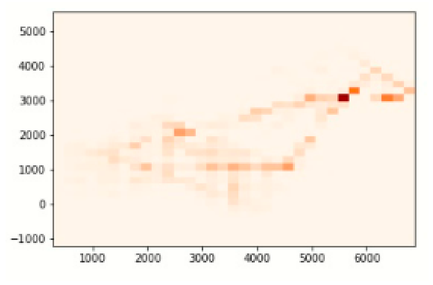
(b) Distribution of output positions Miami training data



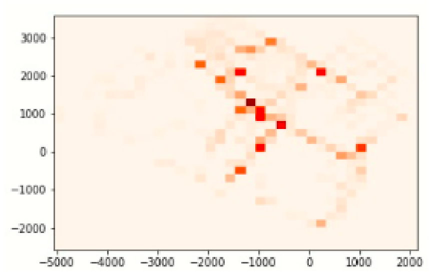
(c) Distribution of output positions Pittsburgh training data



(d) Distribution of output positions Dearborn training data



(e) Distribution of output positions Washington DC training data



(f) Distribution of output positions Palo Alto training data

Figure 4: Distribution of output positions of in all cities

2.3 Problem C

2.3.1 Train/split Data

We combined the dataset of all cities into one big dataset and splitted the combined dataset into to 80% training data and 20% validation data. The train dataset shape is 163051×256 and the validation dataset shape is 40765×256.

2.3.2 feature engineering

In feature engineering, we generated three new features: city, distance, and velocity. We decided to calculate distance and velocity because we believed these features could provide us extra information the motions of the vehicles in addition to just the positions.
We calculated the euclidean distance between two points (a_0, a_1), (b_0,b_1) using Euclidean distance with the formula:

$$distance = \sqrt{(b_0 - a_0)^2 + (b_1 - a_1)^2}$$

We calculated the velocity between two points (a_0, a_1), (b_0,b_1) by calculating the displacement between two points divided by one time step using the formula:

$$velocity = [(b_0 - a_0)/0.1, (b_1 - a_1/0.1)]$$

To capture the different behavior in each city, we included the city information by one-hot encoded the cities. For example, the one-hot encoded representation of Austin is [1,0,0,0,0,0] and the one-hot encoded representation of Miami is [0,1,0,0,0,0].

2.3.3 Data normalization

In addition, we normalized the data by applying data translation and using local positions instead of global positions. We first chose the starting and ending xy-coordinates of each car as our origins and then minused the xy-coordinates of each point by their corresponding starting and ending xy-coordinates respectively. We chose this normalization in order to reduce the range of values in our training data. We found that subtracting ending positions had better performance than subtracting starting positions and thus we employed the local positions that subtracted ending positions.

3 Machine Learning Model

3.1 Problem A

We started with the linear regression model. We tried different features such as xy coordinate position, 6 cities, velocity between two points, and distance between two points as input features. After experimenting and comparing the loss, we finally picked 6 cities, position, velocity, and distance as input features and its size is 256. Inspired by the autoencoder, we picked size 2048 as the hidden dimension and size 4096 as the latent dimension and reverse architecture for the decoder. Such an expanding scheme on input dimension is our attempt to capture high dimensional latent space feature and variation. The output feature is the next 6 seconds predicted xy position, and its size is 120.

The model architecture looks like:

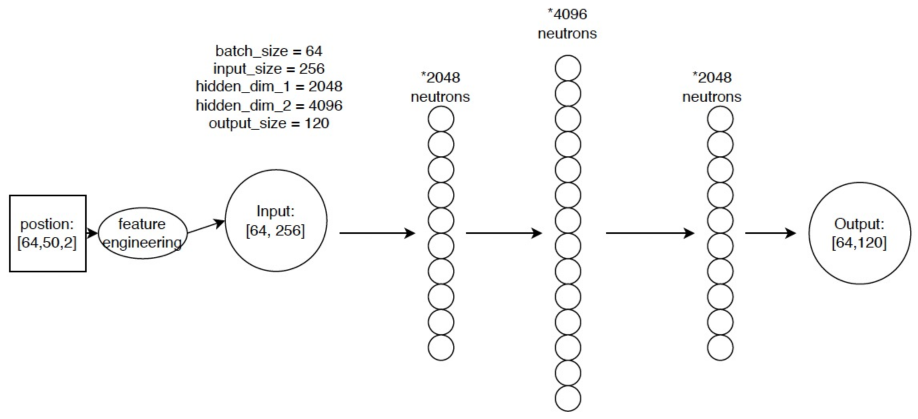


Figure 5: Linear Model Architecture

The model class is torch.nn.Linear. The loss function is torch.nn.MSELoss(), which measures the mean squared error between each element in the estimated values (x_0, y_0, x_1, y_1..., x_{50}, y_{50}) and the actual value (x_0,y_0,x_1,y_1,..., x_{50},y_{50}).

3.2 Problem B

Our deep learning model is LSTM + MLP. The model architecture looks like:

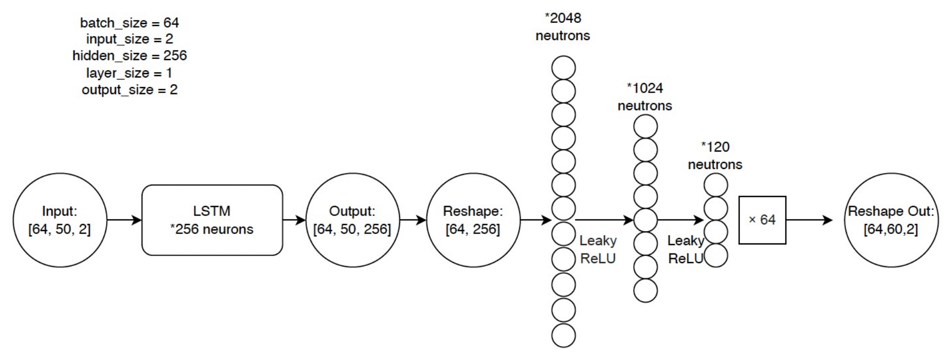


Figure 6: LSTM+MLP Architecture

We picked the initial 5-second xy position as the input feature, and its size is 50*2. LSTM has 1 recurrent layer and 256 hidden units. Then, we applied the MLP algorithm with activation function Leaky ReLu . We expanded the dimension from size 256 to size 2048. Next, the 2048 dimensional input was passed through the first hidden layer to transform it into 1024 dimension, then, another hidden layer, which would transform it to 120 dimension. Finally, we transformed it into a 60 * 2 dimensional vector, which was our output features: the next 6-second predicted xy position. The transformations between 256 to 2048, 2048 to 1024, and 1024 to 120 dimensions were done by Linear layers. The loss function is torch.nn.MSELoss(), it measures the mean squared error between each element in the estimated values (x_0, y_0, x_1, y_1,..., x_{50}, y_{50}) and the actual value (x_0,y_0,x_1,y_1,x_{50},y_{50}).

3.3 Problem C

Below are the models, model architectures, and parameters that we have experimented with. For all models we have tried, we picked Adam as optimizer and mean squared error (MSE) as loss function.

3.3.1 Linear Regression Model/ MLP

Feature Selections	Linear Layers	Hidden dimension size	Activation Function
• Position • City + Position • City + Position + Velocity • City + Position + Velocity + distance	• 2 • 3 • 4	• 512 • 2048 • 4096 • 8192	• No Activation • Sigmoid • ReLU • Leaky ReLU

Linear Regression model is our model for milestone, and its model architectures is described in problem A. After tuning the parameters, we found out that adding 6 cities, 50 time steps position, and velocity, and distance to input features, 3 hidden layers with size 2048, 4096, 2048, and no activation function produced he best results.

3.3.2 LSTM + MLP Model

Model	Linear Layers	Learning rate scheduler	Activation Function	Regularization
• LSTM Model • LSTM+ MLP Model	• 2 • 3	• ReduceLROnPlateau • MultiStepLR	• tanh • Sigmoid • ReLU • Leaky ReLU	• Batch Normalization • dropout

We wanted to explored the temporal nature of our dataset. Therefore, we implemented LSTM that utilized the temporal features. However, we discover that building LSTM alone did not produce very good results. Therefore, we incorporated our previous model with LSTM. Our final model architecture is described in problem B, which is LSTM + MLP model. After tuning the parameters, we found out that after applying LSTM, transforming through 3 linear layers, using MultiStepLR as learning rate scheduler, Leaky ReLu as activation function, and no regularization produced the best results.

4 Experiment Design and Results

4.1 Problem A

4.1.1 Computational Platform/GPU

In order to speed up training, we use datahub and utilize GPU for training and testing. The GPU type is GeForce GTX 1080 Ti.

4.1.2 Optimizer/ Learning rate decay

Our optimizer is Adam. Reading from the article "A Comprehensive Guide on Deep Learning Optimizers" [2], we have learnt that Adam optimizers have faster computation time, require fewer parameters for tuning when compared to other algorithms such as RMSProp, SGD, and AdaGrad. Therefore, we picked Adam as our optimizer.
We learnt from the lecture that a fixed learning rate may not be able to converge to global optimum, instead, we want to vary the learning rate over the training process. Therefore, we set up two learning rate schedulers.
The first scheduler was ReduceLROnPlateau. We tried different initial learning rates in the range of 1e-2 to 1e-5, factors in the range of 0.1 to 0.9, and patience in the range of 0 to 5. After testing and comparing, we reached our decision: the learning rate starts with 1e-4 and the learning rate will decrease by 0.1 (factor is 0.1) if the training loss is continuing greater than the last 3 (patience is 2). If the learning rate reaches 1e-7, the learning rate will stop updating (eps is 1e-7). For better observation, we set verbose as True so that we could know at which epoch the learning rate decreases. This scheduler allows large weight changes at the beginning of the learning process and decreases over time to guarantee convergence to a global optimum. We use this scheduler for linear regression models.
The second scheduler was MultiStepLR; the learning rate started with 1e-4, and decays the learning rate by each parameter group by 0.1(gamma = 0.1) once the number of epochs reaches one of the milestones (milestones = [30,60,80]). We use this learning rate scheduler for our LSTM + MLP model, because we observed that using ReduceLROnPlateau, the training loss continued decreasing and so our learning rate will never update, but the problem is that the validation loss did not decrease. Therefore, using multistep LR can ensure learning rate updates over the training process.

4.1.3 60 step prediction for each target agent

After training, we can now use the trained model to make predictions from the test dataset. We loop through every city. Every city dataset converts to an input tensor, and then makes a prediction using the trained model. For the linear regression model, the last linear layer transformed the hidden dim into output size 120, corresponding to (x0, y0, x1, y1, x2,y2,. . . ., x119, y119) for each target agent. For the LSTM+MLP model, the last linear layer transformed the hidden dim into output size 120, then we reshape the size into [60,2], corresponding to [[x0, y0][x1,y1]...,[x50,y50]] for each target agent.

4.1.4 City information

In our feature engineering process, we use one hot encoding for the city information. For linear regression model, we added to the input features. But for our LSTM+MLP model, we did not utilize the city information.

4.1.5 Epoch/Batch Size/Train time

We used a linear regression model with Position as input features for tuning the epoch length and batch size. We have tried different epoch sizes such as 32, 64, and 128. We found out that batch size 32 has similar performance as 64, but it will take longer to train than batch size 64, so we chose batch size 64. We also tried different epoch sizes such as 50, 100, 150, 200, and found out that the loss continued decreasing until 150, so we chose 150. Therefore, for all experiments, we decided that the epoch size is 150 and batch size is 64 when comparing different models. Every model have different train time for one epoch, for linear regression, it took average 17s to run one epoch.

4.2 Problem B

4.2.1 Few representative Experiments for Linear Regression

Model	Features	Train Loss	Validation loss	One epoch Train time
Linear(batchSize 32)	Position	81.2334	81.9802	30.1s
Linear(batchSize 64)	Position	81.3064	82.0738	16.42s
Linear	Using local position	25.4018	25.6513	16.2s
Linear	Using local position + City specialization + velocity + distance	24.2054	24.3439	16.2s

We observed that different inputs features with the same model can produce different results. The data prepossessing is extremely important as we can see that using local position, the train loss dropped from 29.24 to 25.401, and adding more features, slightly decreased the train loss a little bit. When batch size is 32, the train time is about double the train time with batch size 64, but their train loss is not much difference. Therefore, considering time efficiency, we set batch size 64 as our optimal batch size.

4.2.2 Few representative Experiments for Deep Learning model

Model	Features	Train Loss	Validation loss	One epoch Train time
MLP with Leaky ReLu	Using local Position + city Specialization	24.3027	24.9364	16.26s
MLP with Sigmoid	Using local Position + city Specialization	25.1025	25.7831	17.53s
LSTM	Using local position	19.9441	20.2085	16.1s
LSTM+MLP (ReduceLROnPlateau)	Using local position	13.7089	18.8053	16.4 s
LSTM+MLP (MultiStepLR)	Using local position	15.8924	17.5608	16.5 s

We observed that different activation function with same model can produce different results. After comparing, activation function Leaky ReLu performs better than other activation functions such as ReLU, Sigmoid, and Tanh. LSTM performs better than MLP, then when we incoporate LSTM with MLP, the train loss decreases again. This is our final model architecture. Then, we go into the parameter tuning process. We observed that the training loss of this model continuing decrease, but the validation loss will plateau when it reaches around 19. Our original learning rate scheduler ReduceLROnPlatea will never update the learning rate since the training loss will not go up, so we create another learning rate scheduler MultiStepLR, which will update the learning rate at whatever epoch we want. And the experiment results shows that this new scheduler works well and improve our score on Kaggle.

4.3 Problem C

4.3.1 Visualization of training/validation loss value over training step

Our best-performing model design was LSTM + MLP. The epoch length was 100 and the batch size was 64. The learning rate started with 1e-4 and decreased by 0.1 when epoched reached 30, 60, and 80. The loss function is a mean squared error, it measures the average squared difference between each element in the estimated values (x0, y0, x1, y1. . . , x50, y50) and the actual value (x0,y0,x1,y1,..., x50,y50). From the graph, we observed that both train loss and validation loss were exponential decay, but validation loss was a little unstable at the first few epochs.

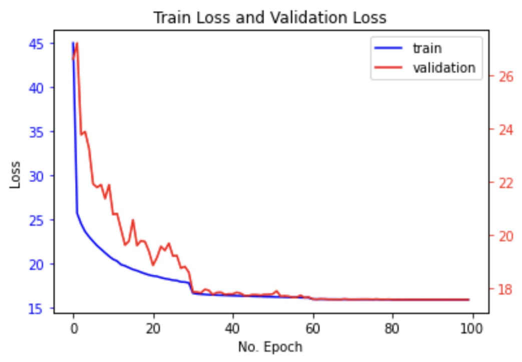


Figure 7: Model LSTM+MLP train/valid loss

4.3.2 Visualization of the ground truth and prediction for some training sample

The blue color is the input, representing the first 5 seconds of the car movement position. The orange color is the output, representing the next 6 seconds of the car movement position. The green color is the prediction, representing the predicted next 6 seconds of the car movement position. From the graph, we observed that our model can produce very good results for straight line output but not very good for curves, which means that our model can predict for the car that go in a straight line movement, but can not know if the car will go left, right or turn into another direction.

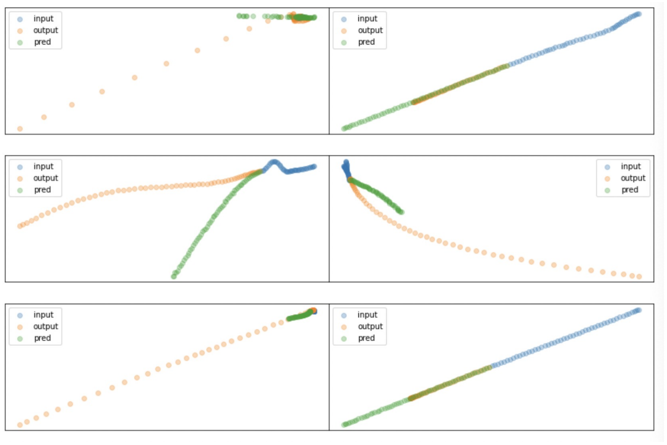


Figure 8: first two samples for Austin, Miami, Pittsburgh city

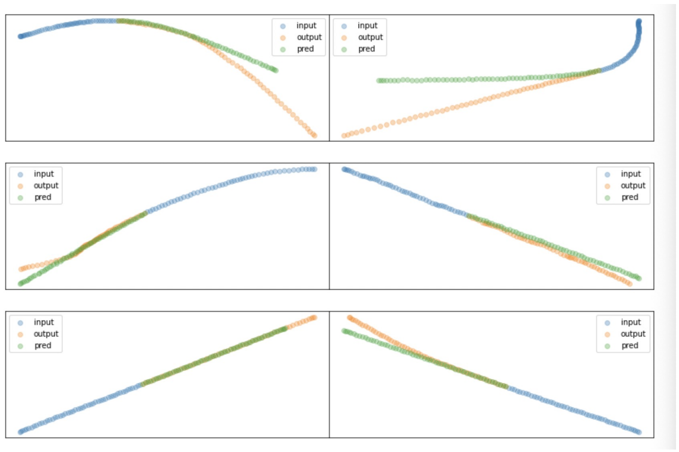


Figure 9: first two samples for Dearborn, Washington-DC, Palo-Alto city

4.3.3 Ranking

Rank No. 4
Public Score: 17.32344 Private Score: 17.33729

5 Discussion and Future Work

5.1 Problem A

5.1.1 Effective feature engineering strategy

The most effective feature engineering strategy is data translation, which normalized the data and decrease the model train loss from more than 80 to approximately 25. Additionally, we have derived new features from existing data by applying mathematical operations in the linear model. For example, we have calculated the Euclidean distance and velocity between data points to augment existing features, which also improved the model performance.

5.1.2 Helpful technique in improving ranking and bottleneck

The most helpful technique in improving our ranking is hyperparameter tuning such as finding a optimal epoch length, batch size, and learning rate, which is also the biggest bottleneck in this project. When we first have our best model LSTM+ MLP, we discovered that our training loss continues decreasing when epoch reach 100. So we increased the epoch length to 200, however, we discovered that the model's training loss continues to decrease but validation loss seems to reach a plateau. Then, we realized maybe it's not the problem of epoch length. Later, we created a new learning rate scheduler for tuning the learning rate and this technique helped us improve our ranking.

5.1.3 Advice for beginner

For a deep learning beginner to design models for similar prediction tasks, we would advise that: first, it is important to understand the data, such as the dimensions and distributions of data, which can give you a better idea to pre-process the data; second, the more complex models do not always have a better performance, and sometimes a simpler model can achieve better results. Therefore, start with simple model, and then continue increase the complexity of the model.

5.1.4 Future work

If we had more resources, we would like to explore more data preprocessing including data rotation, which can enable us to rotate the data to a desired angle, providing extra information in the data. We would also like to apply the ensemble learning methods to improve our neural network and explore the Seq2Seq model, as well as Transformers.

References

[1] Nemanja Djuric and Thi Nguyen Fang-Chieh Chou Tsung-Han Lin Jeff Schneider Vladan Radosavljevic, Henggang Cui. Short-term motion prediction of traffic actors for autonomous driving using deep convolutional networks. *arXiv*, 2018.

[2] Ayush Gupta. A comprehensive guide on deep learning optimizers, May 2022.

[3] Julieta Martinez, Michael J. Black, and Javier Romero. On human motion prediction using recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.