CSE 151B Project Final Report

Zairan Xiang, zaxiang@ucsd.edu **Daniel Son, dson@ucsd.edu**

1 Task Description and Background

1.1 Problem A

The task is to predict future six seconds trajectories for autonomous vehicles based on the movement in the past five seconds.

Solving this task ensures that this deep-learning technology for self-driving cars adds convenience to people's life. Furthermore, especially with the increase in use for autonomous vehicles, this area of study is essential to ensure people's safety by accurately predicting collision-free paths given the vehicles information and road environment. Since autonomous vehicles are mainly electric, increasing the safety of these features will also lead to more consumers purchasing these clean energy vehicles, which in turn will have environmental benefits through reduced CO_2 emission.

1.2 Problem B

Argoverse is a large dataset for autonomous vehicle motion prediction research. It included 300k vehicle trajectories sensr data and 290k mapped lane information. It is not the only sensor-equipped dataset, TrafficPredict (1) also uses sensor-equipped vehicles to observe driving objects and has also created a benchmark in motion forecasting task. It contains 155 minutes of vehicle motion observations compared to the 320 hours of observations in Argoverse dataset. In the research paper Argoverse: 3D Tracking and Forecasting With Rich Maps, The research group (2) used lane-level information on motion prediction experiments ranging from k-Nearest Neighbors to LSTMs, creating a large-scale forecasting benchmark of trajectories capturing scenarios. This research also contributes largely to 3D tracking as the Argoverse dataset also contains 3D tracking annotations. The research shares some similarities in the models we both evaluate such as LSTM Encoder Decoder. However, the research groups proposed different approach to data pre-processing as they focused on more difficult forecasting task such as managing an intersection and speeding up after a turn, so different from what we did in this project, they removed all stationary vehicles from the dataset and only keep those with "interesting" sequences. They also included "social features" such as number of neighbors in their feature engineering process.

The main difference between the methods proposed before and this project is in the lack of "social features" in our dataset. Social interactions such as the interactions between cars and pedestrians can influence future motion heavily. There are lots of work done in the prediction of such social interactions. Some of them like Deo et al. (3) proposed a convolution social pooling approach to use the maneuver predictions from this methods to predict the vehicle trajectories on that maneuver.

Our data pre-processing process shares some similarities with works done before. Same as how the research group (2) processed Argoverse dataset, we also converted our position data into relative position such that each trajectory start at (0, 0). Different from our approach, they also normalized that data so that the trajectory always ends on the x-axis with $y_{finalTimeStamp} = 0$. Park et al. (4) also adopt the relative coordinate system where its ego vehicle's location is fixed to (0, 0) for their proposed deep learning method for vehicle trajectory prediction.

Preprint. Under review.

1.3 Problem C

This is a regression problem as we have the x and y positions for each of the five seconds in the input and our goal is to predict what position the vehicle should go next. The input is a 50 points sequence $(x_1, x_2, ..., x_{50})$ representing position for the first 5 seconds, output is a 60 points sequence $(y_1, y_2, ..., y_{60})$ representing positions for the next 6 seconds, with each x and y being a 2D point indicating current position of the vehicle.
Mathematically, we would like to train a model that find function f such that f$(x_1, x_2, ..., x_{50})$ = $y_1, y_2, ..., y_{60}$.

Our model is capable of solving any time-series or sequential prediction problem. More specifically, it is able to take a sequence of time series input and get the patterns of the input along time step, and be able to predict future actions along that time series. One similar task might be predicting stock prices due to it having a similar structure of sequential data.

2 Exploratory Data Analysis

2.1 Problem A

Describing the details of this dataset:

There are 50 2D points in each trajectories input representing the movement in 5 seconds, and 60 2D points for each output representing the movement in the later 6 seconds.
total train trajectories: 203816, and total test trajectories: 29843.
There are two dimensions in both inputs/outputs: one for x position and one for y position. so we have 50 x 2 for each input trajectory and 60 x 2 for each output trajectory.

Number of train/test trajectories in each city:
• Austin – train: 43041 test: 6325
• Miami – train: 55029 test:7971
• Pittsburgh – train: 43544 test: 6361
• Dearborn – train: 24465 test: 3671
• Washington D.C. – train: 25744 test: 3829
• Palo Alto – train: 11993 test:1686

Also shown below in the bar chart in Figure 1:

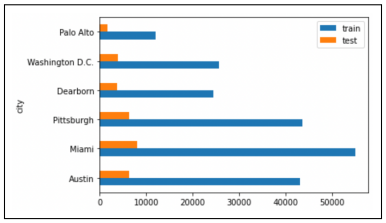


Figure 1: Number of Train and Test Trajectories by City

Visualizing one data sample in Figure2 (input pink, and output green):

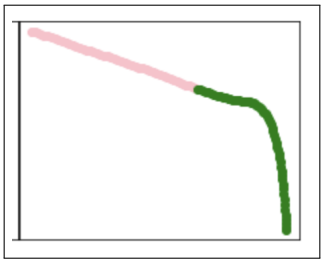


Figure 2: Trajectory Predictions for Sampled Batch

2.2 Problem B

Performing statistical analysis to understand the properties of the data:

The distribution of input and output positions for all agents is shown below in Figure 3:

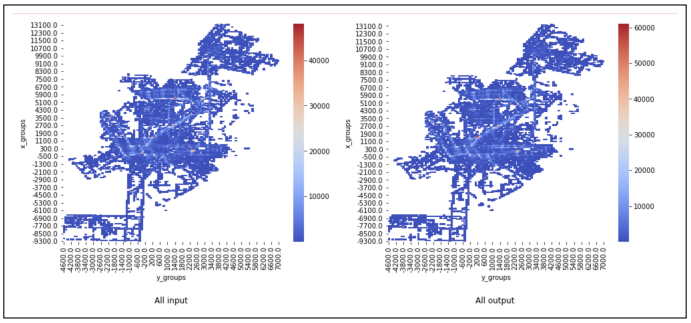
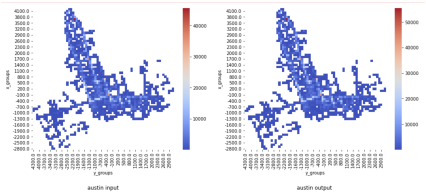


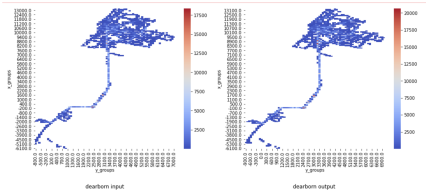
Figure 3: Distribution of Input and Output for All Agents

From Figure 3, We can see that the motions are centered with the high-frequency positions are mostly at the center of heatmaps. We can also see some patterns in the distributions that might indicate potential trajectory with high frequency. Another interesting finding is that the input and output distributions are nearly the same.

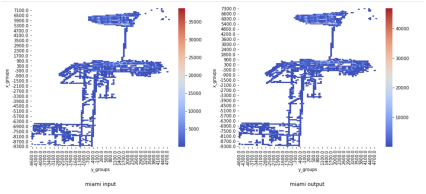
We then get the distributions of input/output positions for each city in Figure 4.
From the distributions in Figure 4, we can see some clear patterns in each city as well as the difference of distributions among cities, and the input / output distributions for each city are also nearly the same as we observed in Figure 3.



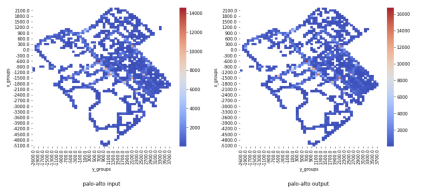
(a) Austin



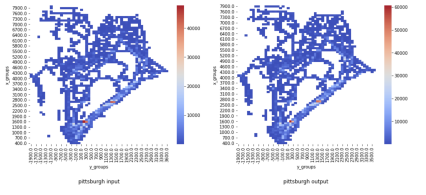
(b) Dearborn



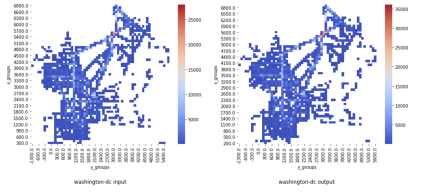
(c) Miami



(d) Palo Alto



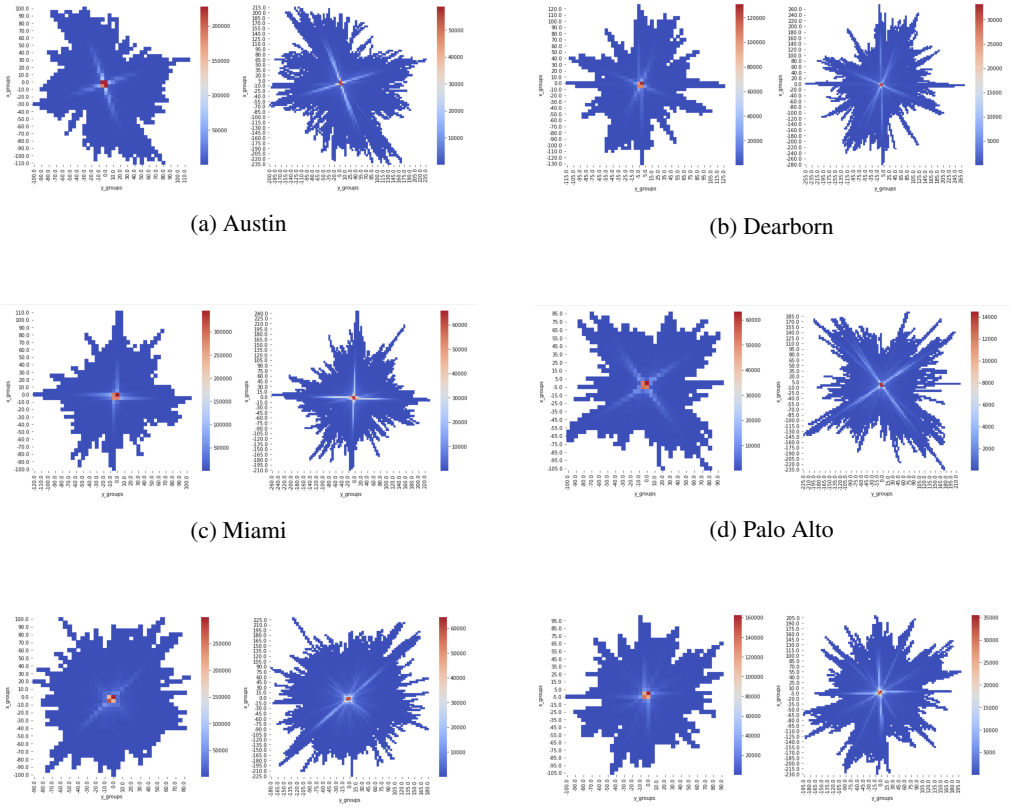
(e) Pittsburgh



(f) Washington D.C.

Figure 4: Distribution of Input and Output for each city

Next, We would like to see the distributions of input/output relative positions (relative to each trajectory's starting point)for each city in Figure 5.



(a) Austin (b) Dearborn

(c) Miami (d) Palo Alto

(e) Pittsburgh (f) Washington D.C.

Figure 5: Distribution of Relative Input and Output for each city

We can see from the relative position distributions in Figure5 that most vehicles move in straight line, and there are certain patterns in each city on the directions of vehicles' movement. For example, vehicles in Miami seem to move along a single axis. Furthermore, if we look closely on the numbers on the axis for input and output distribution, we can see the ranges for x and y position in output are double of the ranges in input, which make sense because we would expect the vehicles to be moving about the same length in 6 seconds as the length they moved in 5 seconds. We would see this further in depth with a velocity histogram in Figure 6.

Velocity histogram for all agents in Figure 6:

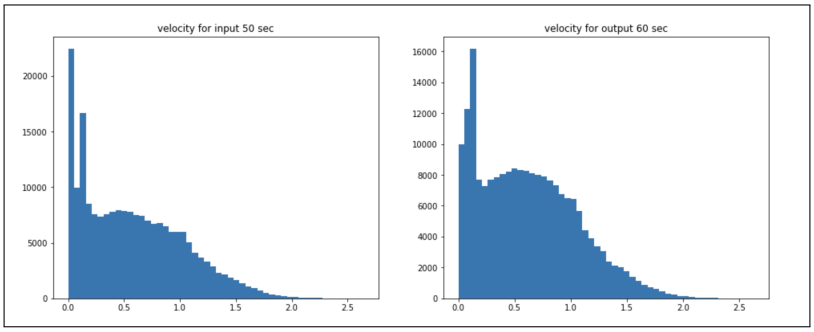


Figure 6: Distribution of Velocity of Input and Output

The average velocity for input data is 0.58 and the average velocity for output is 0.62, which means the vehicles are speeding up instead of slowing down along its path. We can also see a peak around 0 per second in both the distributions for input and output, which indicates vehicles to be static.

2.3 Problem C

Processing the data to prepare for the prediction task:

We split training and validation data by a 8:2 ratio; There are 163051 entries in the training data and 40765 in the validation data. As we see from the above Figure 3 and Figure 5, there is no clear pattern present in the absolute positions of the vehicles, but there are some patterns in their relative position. We believe the changes in position can better reflect how vehicles make their movements, so we converted all data points from its absolute position to relative position (relative to each origin, sample-wise). We did this by subtracting each 50 and 60 data points to the position at first second. Further on the feature engineering, we also added city information to each data point because we have noticed that the movement patterns in each city different a lot. We achieved this by doing one-hot encoding for each city on the input data. For example, the relative data point (-1, 1) in city Austin will be changed into (-1, 1, 1, 0, 0, 0, 0, 0) and the relative data point (3, 5) in city Miami will be changed into (3, 5, 0, 0, 1, 0, 0, 0).
In the later experiments, we also tried training one model for each individual city, in that case, we removed the one-hot encoded city information from the data.

We tried to normalize the data based on the mean of the whole data set, but this ended up to worsening the model's performance slightly, so we decided to not use any normalization. We also tried batch norm which ended up worsening the model performance significantly. Due to the results of these normalization attempts, we chose to exclude normalization from our model.

3 Machine Learning Model

3.1 Problem A

The input feature that was used for the prediction was the 5 seconds sequence trajectory position at each timestamp in and the linear model predicted the 6 seconds sequence future position based off the information given. Mathematically, input feature is $(x_0, x_2, x_3, ..., x_{50})^T$ while output feature is $(y_0, y_1, Y_3, ..., y_{60})$, while each x and y is a 2D data representing current position of the vehicle at that timestamp. In later experiments we included feature engineering and performed data pre-processing.

The model class we chose to use was a single linear Multilayer Perceptron with the loss being the Mean Squared Error. The architecture is shown in Figure 8.

3.2 Problem B

The deep learning pipeline for the prediction task:

The input features used for the deep learning prediction model was the city information one hot encoded and the trajectory paths converted to be relative to the origin of the trajectory. Input data was in form $(x_0, x_1, ..., x_{50}, 0, 1, 0, 0, 0, 0)^T$ as described in Section 2.3. The output for the training step is the output x and y positions for 6 seconds relative to the origin of the trajectory. We convert this relative trajectory to the absolute position in the testing stage by adding the origin to 60 output data points in our predictions. Output data were in form $(y_0, y_1, ..., y_{60})^T$.

We used a Multilayer Perceptron with linear layers in an Encoder and Decoder architecture with the loss being the Mean Squared Error.
The model architecture is shown in Figure 7. The Linear architecture for a single hidden linear layer is shown in Figure 8. We have 4 linear layers in encoder MLP and 3 linear layers in decoder MLP. It takes in data of size (2, 56) and reshape it to (1, 400). The encoder hidden layers take sizes of 240, 96, 48, and 16, and the decoder hidden layers takes in the (1, 16) last output from the encoder and its hidden layers take sizes of 32, 64, and 120. It then reshape the final output of the decoder from (1, 120) to (2, 60) to get the 6 seconds sequence prediction. Among the linear layers in both encoder and decoder, we use ReLU as the activation function.

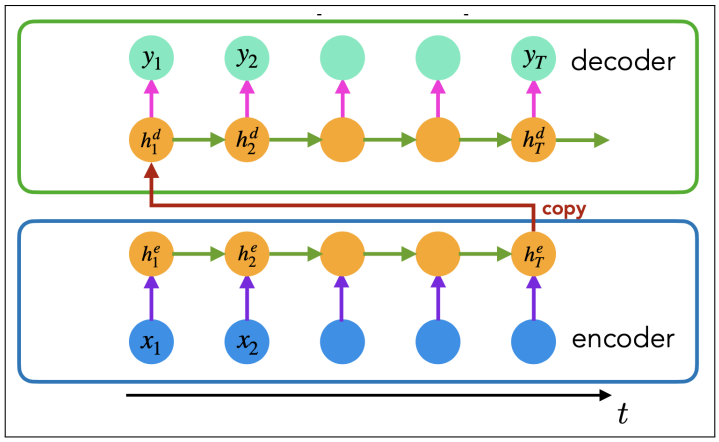


Figure 7: Representation of the Model Architecture(5)

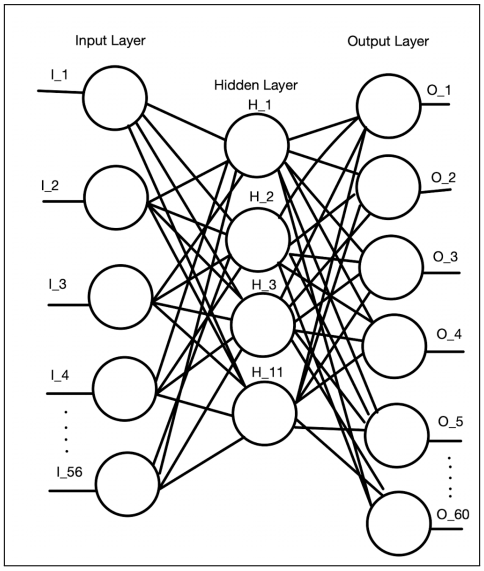


Figure 8: Representation of a Single Layer MLP

3.3 Problem C

All the models we have tried to make predictions:

• Single Linear Layer MLP:
The baseline model we tried is a MLP with a single linear layer. It takes in input data of size (2, 56) and reshape it to (1, 400), the data with new size were then being input into the layer and it will output data of size (1, 200), which will be reshaped into (2, 60). A visualization can be found in Figure 8.

• LSTM with Encoder Decoder
To make the prediction for each target agent, we utilized an encoder and decoder structure. Our first thought of the RNN structure because our task is to predict time-series sequence. We utilized LSTM in our RNN structure to store the information along hidden layers. It consisted of a LSTM layer followed by two linear layers in each the Encoder and Decoder. Each layer was still followed by the ReLU activation function to give it non-linearity. Since the LSTM model is much more complex than linear, we did not want it to overfit to the data, so we took steps to reduce the complexity of the model by removing layers. Additionally, we incorporated a dropout and embedding layer in the model to prevent overfitting. In the LSTM, all the inputs are passed through the hidden layers so it can make predictions utilizing the information stored in its memory cell. Due to being able to store past information, and future information in a bidirectional LSTM, it makes this model a strong

candidate for evaluating sequential data, such as the trajectory information that we are evaluating. A visualization can be found in Figure 9.
The LSTM encoder-decoder model the conditional probability $p(y_1, y_2, ..., y_{60}|x_1, x_2, ...x_{56})$. The encoder provided summary of input sequence data through the LSTM cell state as a single data c_T. The decoder then successively make prediction at timestamp t based on the conditional probability $p(s_t|c'_{t-1}, s_{t-1})$ where c'_{t-1} is the cell state in the decoder state at timestamp t-1.

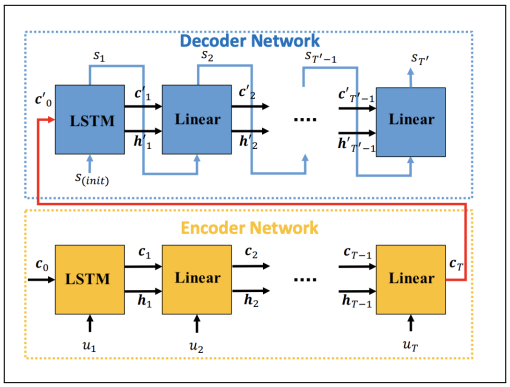


Figure 9: Representation of a Single Layer MLP(4)

• Linear MLP with Encoder Decoder
A similar structure as our RNN-LSTM model, with the same number of inputs and outputs for the encoder and decoder. we also utilized an encoder and decoder structure for the linear model, which was the model that produced the best results. It consists of multiple nn.linear() layers each followed by a ReLU activation function to give each linear layer non-linearity to allow it to predict non-linear relationships within the data. There are 3 layers in the decoder and 4 layers in the encoder. All of these layers and activation functions were run in sequence using nn.Sequential() from PyTorch. As we discussed in Section 3.2, the size of the input sample when it is first passed into the encoder is 400 and when it exits the encoder it is 16. When it is passed into the decoder it is 16 and exists with a size of 120. We added more layers to this model to make up for the simplicity of each individual linear layer. By adding more layers, it will increase the complexity of the model, and therefore increase the trajectory patterns that the model is able to learn. We didn't use any regularization techniques such as batch-norm, dropout, and max-pooling in our final model. We addresed why we chose not to use batch-norm or other kinds of normalization in Section 2.3. We also didn't use max-pooling technique because we are not including feature map in either our MLP or LSTM-RNN model. We chose not to incorporate a dropout layer for the linear model since it did not have the issue of overfitting. When we tested a model with a dropout layer present, it increased the error, so it was promptly removed. Our model did not have an issue with overfitting, due to the test error being lower than the training error, so the dropout error did not contribute much due to its nature of helping to alleviate overfitting. The model architecture visualization can be found in Figure 7.
Similar to LSTM, The MLP encoder-decoder model the conditional probability of output sequence given the input sequence, i.e., $p(y_1, y_2, ..., y_{60}|x_1, x_2, ...x_{56})$. The encoder summaries input sequence into a data h of size 1 X 16 through the 4 fully-connected linear layers. The conditional probability is then $\prod_{t=1}^{60} p(y_t|h, y_1, y_2, ..., y_{t-1})$. During every step in the decoder, it uses the probability of y_t based on the output y_{t-1} in the previous step, and use the tentative decision for the next update in the decoder state.

4 Experiment Design and Results

4.1 Problem A

Setting up the training and testing design for deep learning:

For the training and testing of our model, we utilized the complementary UCSD DataHub to get access to more powerful machines with 1 GPU, 8 CPU, 16G RAM. The training and testing of the model was ran on the GPU due to it being much faster than running it on our own laptop's CPU.

The final optimizer we chose for our model is the Adam algorithm from the torch.optim package. Compared to all the other optimizers we tested, such as Stochastic Gradient Descent, the Adam algorithm consistently found the predictions with the smallest error in the smallest amount of epochs. As for tuning other parameters, such as learning rate, number of epochs, and batch size, it was done through lots of trial and error combined with our past experience from the numerous previous models we tested.

The batch size we chose was 4. We tried various batch sizes and found if we made it larger, the training error was higher. Due to this we kept the model simpler with a batch size of 4. The number of epochs conducted was 50, which took about a hour to run. We found that around 40 epochs, the errors started to converge. When testing it with 100 epochs, the error still continued to minimally decrease, but the testing error was much higher than the training error, meaning the model had overfit the data. For the learning rate, the one that we found worked the best was 1e-3. If it was increased to 1e-2, we found that the training error would bounce around and converge earlier at a higher training error. If the learning rate was reduced to 1e-4 then we found the model learned very slowly and it required many more epochs to get a similar training error to the one we achieved with a rate of 1e-3. Due to these considerations, we went with a learning rate of 1e-3 for most of our testing and training since it was found to be the most balanced option for most models.

Making multistep (60 step) prediction for each target agent:
First, we preprocessed the data by converting the data to the relative position to each of their origins. City information was also added as a feature by one hot encoding that information. The data was also normalized by using the mean before training.
To make the prediction for each target agent, we utilized an encoder and decoder structure, which gives us the freedom to choose what model we want to use in the decoder phase. With this flexibility, it allows for more possibilities to implement the model which was very useful when testing difference models. The model that produced the best results so far was the one based on nn.linear() layers paired with ReLU activation functions. There are 3 layers in the decoder and 4 layers in the encoder. The activation function used after each linear regression to give it non-linearity was ReLU and all of this was run in sequence using nn.Sequential() from PyTorch. The size of the input sample when it is first passed into the encoder is 400 and when it exits the encoder it is 16. When it is passed into the decoder it is 16 and exists with a size of 120.

For the city information, we decided to retain the city information in the data due to different driving behaviors. For example, in more urban cities, there will be more obstacles that have to be taken account of in the prediction due to there being more people and structures being more densely packed. On the other hand, less urban cities will have less obstacles to navigate around and therefore straighter trajectories. We also found through analysis of the data that different cities had different common trajectories. Due to this, there had to be some way in the training to differentiate different city information from each other. The solution we came up with to maintain the different groups of cities was to one hot encoded the city information as an additional feature for the data. Rather than evaluate each city separately, one hot encoding the city allows for us to use one model for all of the data at once instead of training a model for each city. By having all of the city data available in the training dataset, the model will now be able to learn more complex patterns of driving behavior from the now larger training dataset.

In the final model, 50 epochs was used and the batch size was set to be 4. It took around 50 seconds for one epoch.

When testing for the optimal batch size and number of epochs for our final model, we found it worked the best with a smaller batch size (4) and with more epochs (50+). For several other variations of linear models we tested, they took only around 15 epochs for the error to converge and they generally performed worse with higher batch sizes (64+).

4.2 Problem B

A few representative models and report of the their results are shown below in Table 1:

Table 1: Model Designs and Performance

Model Design	MSE Training Loss	MSE Test Loss	Training Time /epoch	Parameters
Encoder Decoder RNN with LSTM	4382047.54824	9993511.61486	5 min	676000
MLP Encoder Decoder run on all cities	24783.68309389	13716.46251	2-3 min	150064
MLP Encoder Decoder run on city separately	821.59873	544.36019	1-2 min	150064
MLP Enc Dec w/ one-hot encoding of city	462.86587	395.99066	2-3 min	150064
MLP Enc Dec w/ one-hot and relative positions	29.87579	26.71983	2-3 min	150064

Table 1 illustrates representative models of our efforts to generate a model that can accurately predict trajectories for autonomous vehicles. The training time mainly depended on the size of the dataset that the model was trained on, as well as the how complex the model itself was. The more complicated models, such as the LSTM with an encoder and decoder took the most amount of time, and depending on what was inside the encoder and decoder it took longer in some cases. When the MLP model was run on all the city data at once, it took longer compared to the model being run only on one city at a time since the data size is larger, there are more parameters the model has to take into account. We tried to improve the training speed for LSTM-RNN model by removing a few layers but the that didn't work well for the model performance. For the MLP model, we tried to train the model and get predictions on each city separately but the amount of time it took to finish training on all six cities didn't vary much compared to training a single model with one-hot encoded city information. As we still hope our model could learn some common patterns across cities, we still used the single MLP model and trained it on all the data. We managed to increase training speed by increasing batch size and adaptive stop after a few epoch when the loss starting to converge. Based on our final model, which was the MLP with a linear encoder and decoder with one hot encoding and relative positions, it seems like what had the most impact to improving the loss was the pre-processing, particularly changing the data to be relative to its original position. Paired with the feature engineering and pre-processing that we conducted, a simpler model provided the lowest test and training error for us.

4.3 Problem C

By visualizing the training/validation loss (MSE) value over training steps in Figure 10, we see an exponential decay in the loss, which is exactly what we expected.

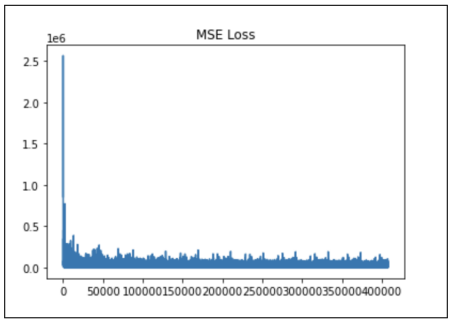


Figure 10: Test MSE and Ranking

Visualizing the ground truth and our predictions by randomly sampling a few training samples after the training has finished, shown in Figure 11.

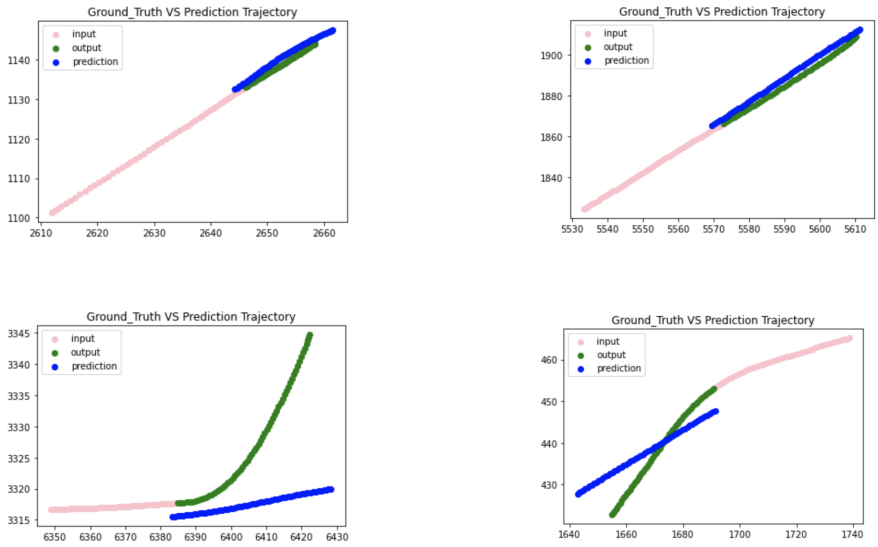


Figure 11: Random Sample of Input, Output, and Prediction

We can see in Figure 11 that our model is better at predicting movement along straight path.

Our final test MSE is 26.71983 and ranking on leaderboard is 17/49.

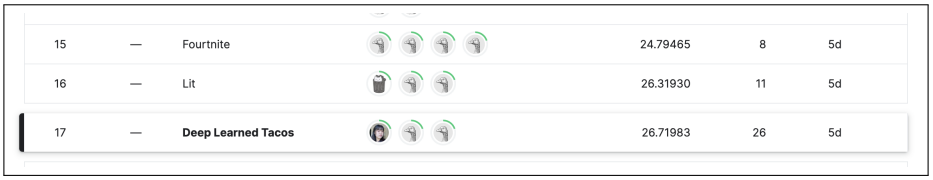


Figure 12: Test MSE and Ranking

5 Discussion and Future Work

5.1 Problem A

Analyzing the results and identifying the lessons/issues that we have learned so far.

• The most effective feature engineering strategy:
The most effective feature engineering strategy that we found was one hot encoding the city information as an additional feature. By adding the city information as a feature into the data, we were then able to run one model for all six cities. This means we can use all the city data for the training set rather than evaluate them separately, which limits the amount the model can learn from the smaller data pool. With more data, it gives the model more opportunities to learn more complex patterns within all the city data. Additionally, having the city information encoded into the data is important due to difference cities having varying common trajectories and driving behaviors. When testing a single model on all the data without having city information, the error was much higher compared to with the city information encoded into the data.

• Techniques we found most helpful in improving the ranking:
When it came to improving our ranking, pre-processing the data to find the relative position to the

original position had the most impact on our overall Kaggle ranking. After doing this processing, it brought our MSE down from approximately 400 to 26, an improvement of 374 for the MSE.
Additionally, tuning the batch size and number of epochs played a significant role in improving our ranking. For certain models we tested, such as our final linear model, it worked best with smaller batch sizes (4) and more epochs (50+). For several other variations of linear models we tested, they took only around 15 epochs for the error to converge. As for the LSTM, the errors tended to converge around 30 epochs and a larger batch size (64+) produced better results. In short, took a lot of testing with each unique model to find the particular best combination of batch size and number of epochs to find the predictions with the lowest error that the model is capable of outputting.

• The biggest bottleneck in this project:
The biggest bottleneck we encountered when developing our model was the time it took to train. For our final model, 50 epochs took about a hour to finish training, so it took a lot of waiting to see if any changes we made resulted in a lower error or where the error would converge.

• How would we advise a deep learning beginner in terms of designing deep learning models for similar prediction tasks:
The best advice that we can give to a beginner at generating deep learning models is to start with focusing on optimizing a simpler model to better understand the process of developing a model from scratch and to get a feel of what the data is like. Once you are more familiar with the data itself, think of what sorts of prepossessing and other models would work with it the best. In this case we were given sequential data, so LSTM theoretically would work well. However that model is harder to properly implement compared to a simple linear model due to the additional intricacies and dependency on the the accompanying pre-processing and in-processing methods to the model.

• If we had more resources, other ideas would we like to explore:
Given more resources, particularly time, computational power, and technical knowledge, some other ideas that we would like to explore are bidirectional LSTM models, optimal normalization methods, and additional preprocessing methods. During our exploration of the data and testing of the model, we touched on these topics and they seemed to have promising results, but in the end the model with the best results differed from what we thought.

6 GitHub Link

`https://github.com/zaxiang/Autonomous-vehicle-motion-forecasting`

References

[1] Yuexin Ma, Xinge Zhu, Sibo Zhang, Ruigang Yang, Wenping Wang, and Dinesh Manocha. Trafficpredict: Trajectory prediction for heterogeneous traffic-agents. In Proceedings of the 33rd National Conference on Artifical Intelligence, AAAI'19. AAAI Press, 2019.

[2] Ming-Fang Chang, John Lambert, Patsorn Sangkloy, Jagjeet Singh, Slawomir Bak, Andrew Hartnett, De Wang, Peter Carr, Simon Lucey, Deva Ramanan, James Hays; Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 8748-8757

[3] Nachiket Deo and Mohan M Trivedi. Convolutional social pooling for vehicle trajectory prediction. arXiv preprint arXiv:1805.06771, 2018.

[4] S. H. Park, B. Kim, C. M. Kang, C. C. Chung and J. W. Choi, "Sequence-to-Sequence Prediction of Vehicle Trajectory via LSTM Encoder-Decoder Architecture," 2018 IEEE Intelligent Vehicles Symposium (IV), 2018, pp. 1672-1678, doi: 10.1109/IVS.2018.8500658.

[5] Rose Yu, Lecture-04-2, CSE151B SP22, 2022.