CSE 151B Project Final Report

Yujie Zhang
University of California, San Diego
yuz035@ucsd.edu

Muchan Li
University of California, San Diego
mul005@ucsd.edu

Zhendong Wang
University of California, San Diego
zhw005@ucsd.edu

Sijie Liu
University of California, San Diego
sjliu@ucsd.edu

Github Link:
https://github.com/Lord-of-Bugs/UCSD-autonomous-vehicle-motion-forecasting-challenge

1 Task Description and Background

In this section, we will introduce the prediction task in detail and introduce the background research about this task.

1.1 Autonomous Vehicle Motion Forecasting Task

In this competition, the task that my team is trying to achieve is to predict a vehicle's motion in a given city at a particular scene, as defined by it's XY position in the 2D Cartesian coordinate plane, 6 seconds into the future (broken down into 60 timestamps), given the object's current 5 seconds (or 50 timestamps) sequence of motion.
This task is especially relevant under the current surge in autonomous driving breakthroughs, where people on one hand are beginning to realize the convenience and sustainability of this field of technologies, but on the other hand still remain doubtful about the its reliability and precision. While our deep learning model may just be a start, the larger suite of more advanced and rigorous deep learning models can bring important contributions to the field and aid the AV engineers in understanding more reliable algorithms and hence constructing more secure and more convenient autonomous vehicles.

1.2 Background Research

In the "Motion Prediction for Autonomous Vehicles from Lyft Dataset using Deep Learning" article, the authors examine the performance of different EfficientNet and ResNet architectures in making motion predictions for autonomous vehicles. The displacement error, which is the euclidean distance between the ground truth and model predicted positions of the agent trajectories, is used as the optimization problem. [1]

In the "Learning to Predict Vehicle Trajectories with Model-based Planning" article, the authors introduced a new prediction framework called the PRIME, which is Prediction with Model-based Planning. The PRIME utilizes the model-based generator to predict future vehicle trajectories under explicit constraints and uses a learning-based evaluator to select future vehicle trajectories. [2]

In the article written by Alahi and Goel et al., they introduced the Social-LSTM, which conducts social pooling within an encoder-decoder network, providing a solution to address permutation problems caused by various social inputs. The authors utilized the social features when making

Preprint. Under review.

human trajectory predictions. In our project, we did not utilize the social features, however, this can be a future work to do. [3]

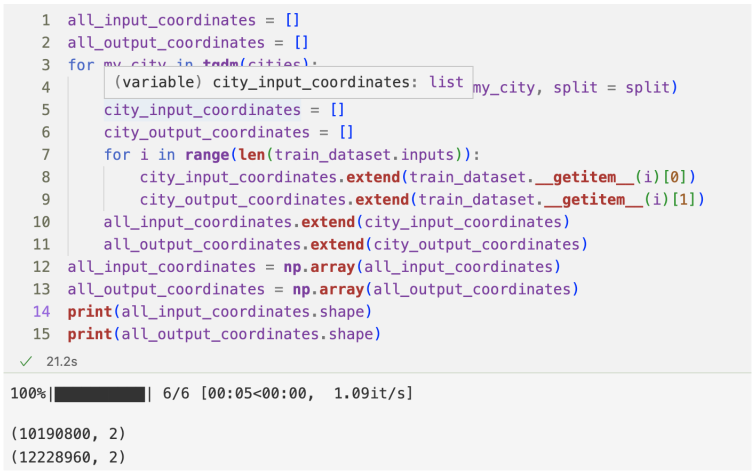
In the “Argoverse: 3D Tracking and Forecasting With Rich Maps” article, the authors introduced several state-of-the-art methods for making autonomous vehicle motion predictions. They utilize the social context (for example, the number of neighbors to an agent) and the spatial context to conduct feature engineering. The deep learning model they used is the LSTM Encoder-Decoder model with different features. In our project, we used the LSTM model as well but didn’t consider the social context.[4]

1.3 Define the task mathematically

To quantify our problem’s input and output space mathematically, our model will train on input matrices of dimension $N \times 50 \times 2$, where N denotes the number of geographical scenes where our 50-timestamp vehicle motion data are acquired from (for example, if we sample 300 scenes worth of data from San Diego, then our training dataset for San Diego would have dimension $300 \times 50 \times 2$), and attempt to predict output matrices that are of dimension $N \times 60 \times 2$.
We believe that the models trained for this task might be useful in other sequence or trajectory prediction tasks as well. In particular, if we can re-frame some of the times series data analysis and forecast task into purely trajectory analysis, such as tracing the trajectory of stock market, we believe our model, with minor revision in its architecture, can also make effective predictions.

2 Exploratory Data Analysis

2.1 General View



```
all_input_coordinates = []
all_output_coordinates = []
for my_city in tqdm(cities):
        (variable) city_input_coordinates: list my_city, split = split)
    city_input_coordinates = []
    city_output_coordinates = []
    for i in range(len(train_dataset.inputs)):
        city_input_coordinates.extend(train_dataset.__getitem__(i)[0])
        city_output_coordinates.extend(train_dataset.__getitem__(i)[1])
    all_input_coordinates.extend(city_input_coordinates)
    all_output_coordinates.extend(city_output_coordinates)
all_input_coordinates = np.array(all_input_coordinates)
all_output_coordinates = np.array(all_output_coordinates)
print(all_input_coordinates.shape)
print(all_output_coordinates.shape)
```

```
✓ 21.2s

100%|██████████| 6/6 [00:05<00:00,  1.09it/s]

(10190800, 2)
(12228960, 2)
```

Figure 1. Dimensions

The input shape: [50, 2]; output shape: [60, 2]. The input training dataset, combining all six different cities’ inputs into one, is of dimension 10190800×2. The output training dataset, combining all six different cities’ outputs into one, is of dimension 12228960×2.
The dimensionalities agree since we can confirm that $10190800/50 = 12228960/60 = 203816$, meaning our input and output training set contain equal number of 203816 scenes from all 6 cities.

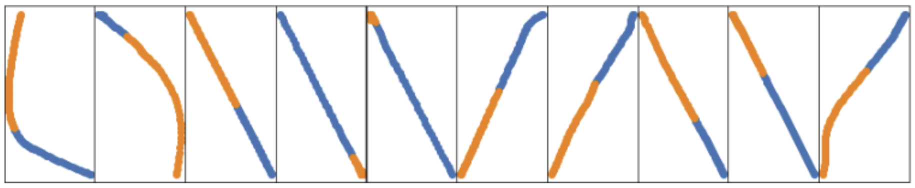


Figure 2. Data Samples for Austin

As we can see, data samples for Austin are represented in Figure 2. Each data point is a data sample. Blue lines represent first 50 time step trajectories, and orange lines represent next 60 time step trajectories (the truth we want our predictions to be as close to as possible).

2.2 Distributions

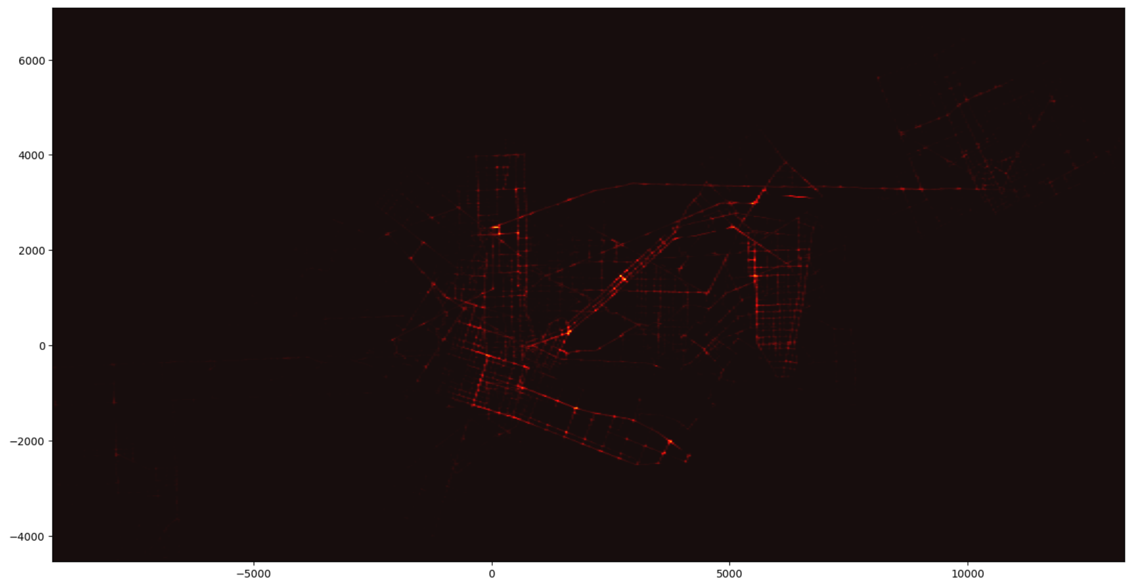


Figure 3. Distribution of input positions for all agents

The distribution of input positions for all agents shows in Figure 3 above.

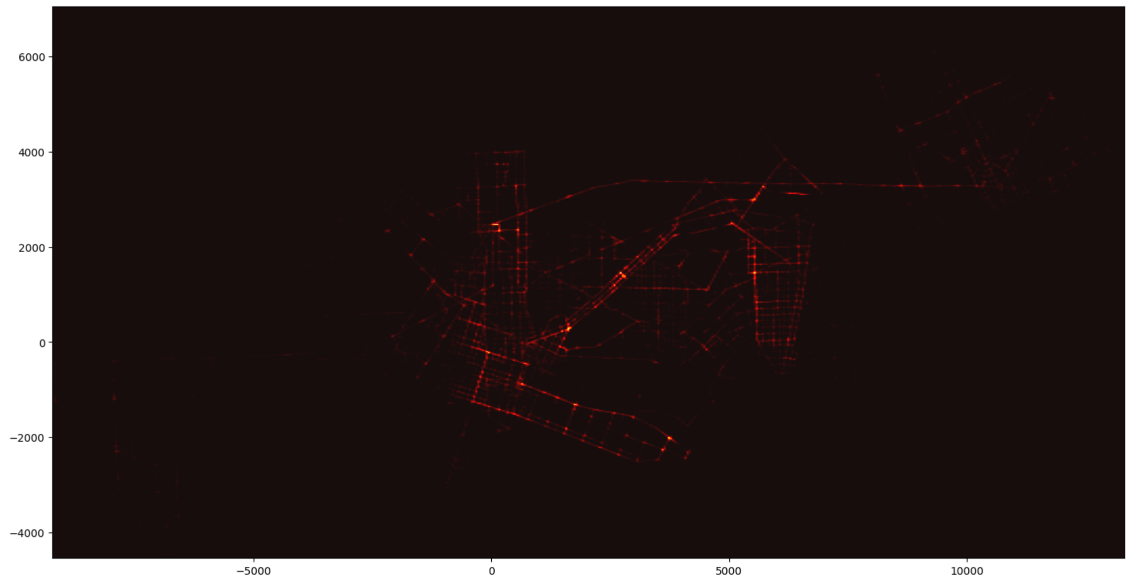


Figure 4. Distribution of output positions for all agents

The distribution of output positions for all agents shows in Figure 4 above.

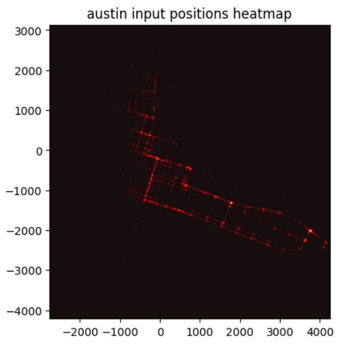


Figure 5. Austin Input

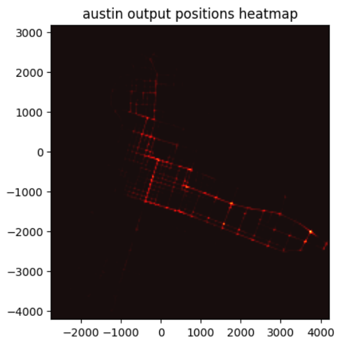


Figure 6. Austin Output

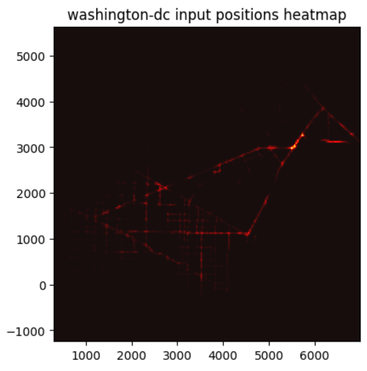


Figure 7. Washington D.C. Input

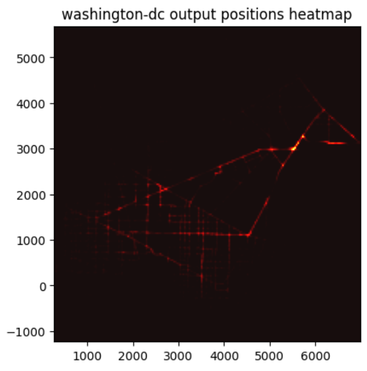


Figure 8. Washington D.C. Output

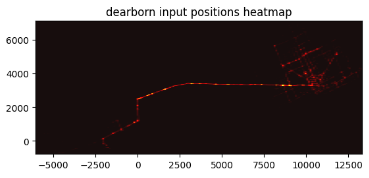


Figure 9. Dearborn Input

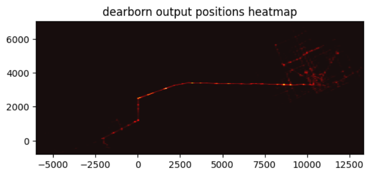


Figure 10. Dearborn Output

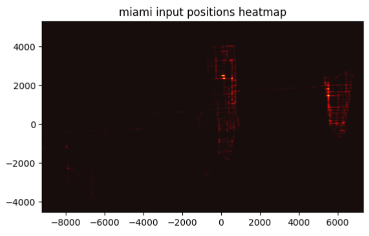


Figure 11. Miami Input

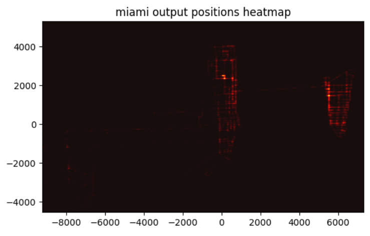


Figure 12. Miami Output

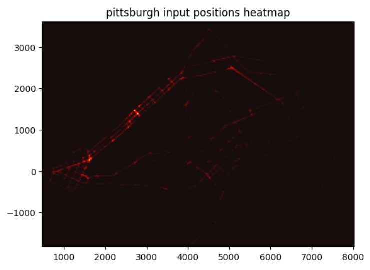


Figure 13. Pittsburgh Input

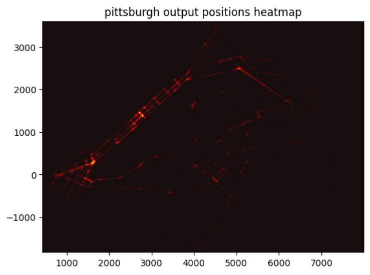


Figure 14. Pittsburgh Output

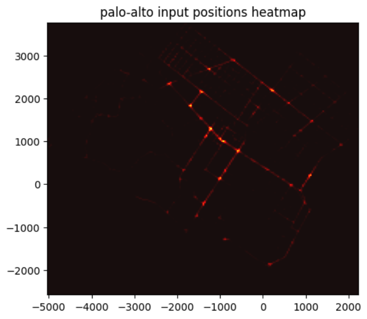


Figure 15. Palo Alto Input

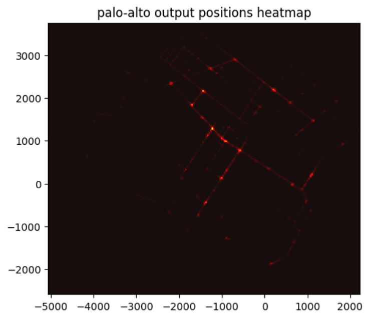


Figure 16. Palo Alto Output

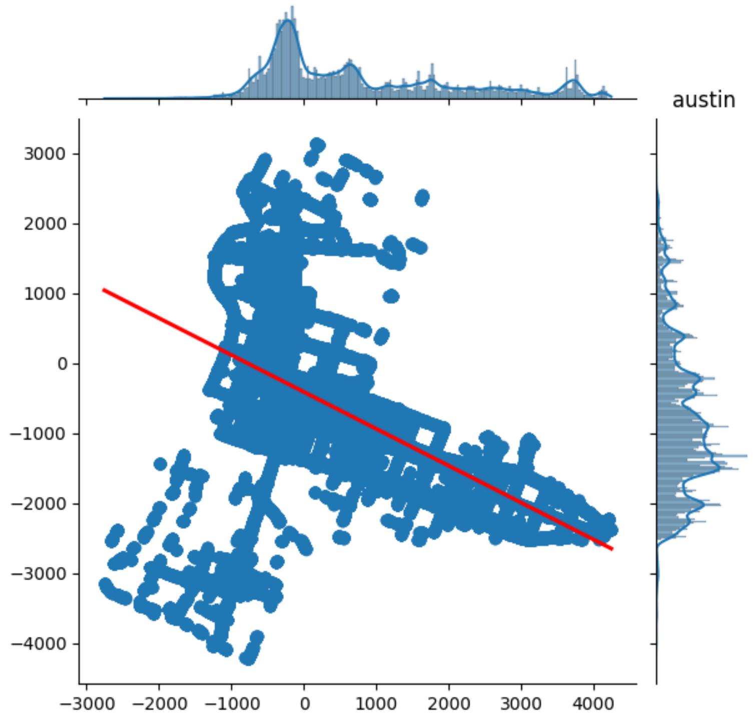


Figure 17. Joint plot of vehicle XY positions in Austin

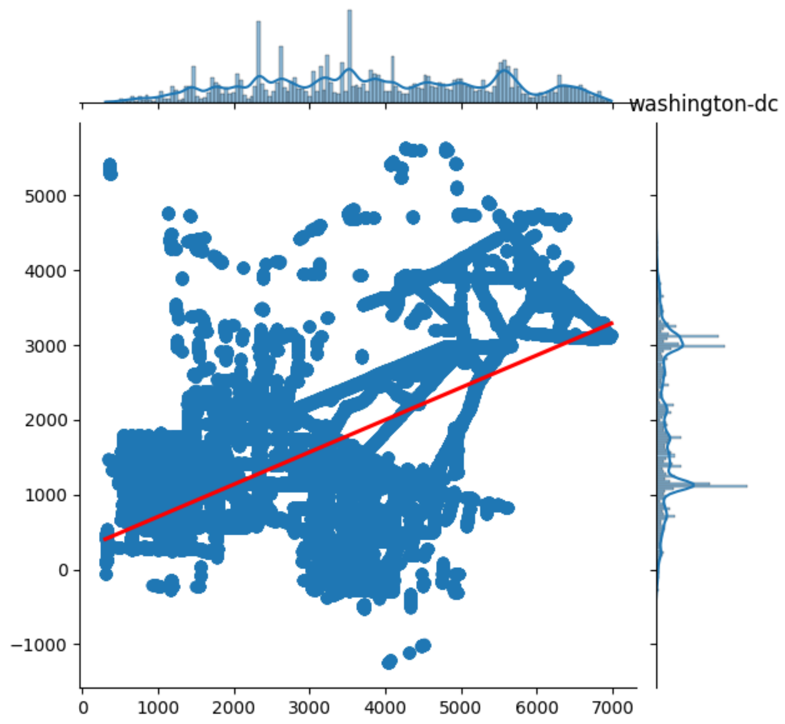


Figure 18. Joint plot of vehicle XY positions in Washington D.C

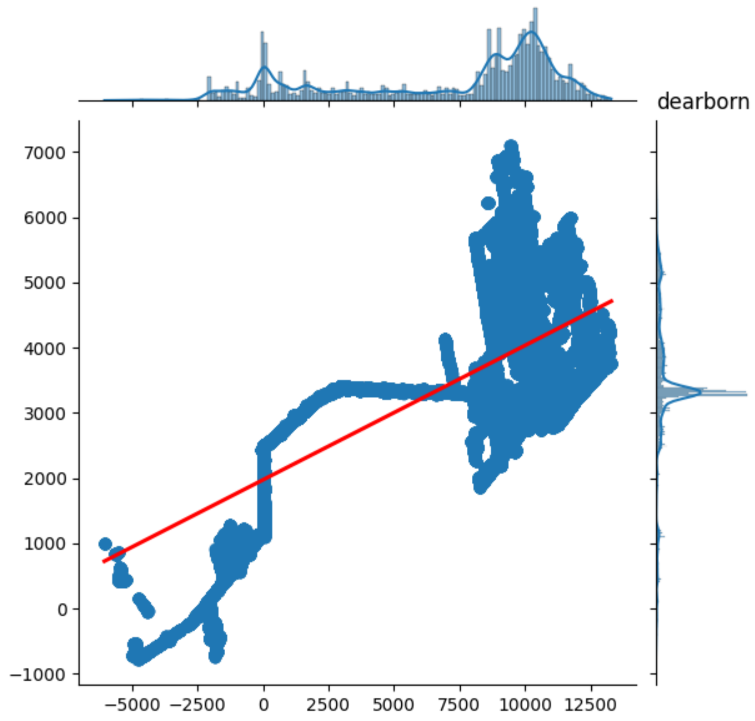


Figure 19. Joint plot of vehicle XY positions in Dearborn

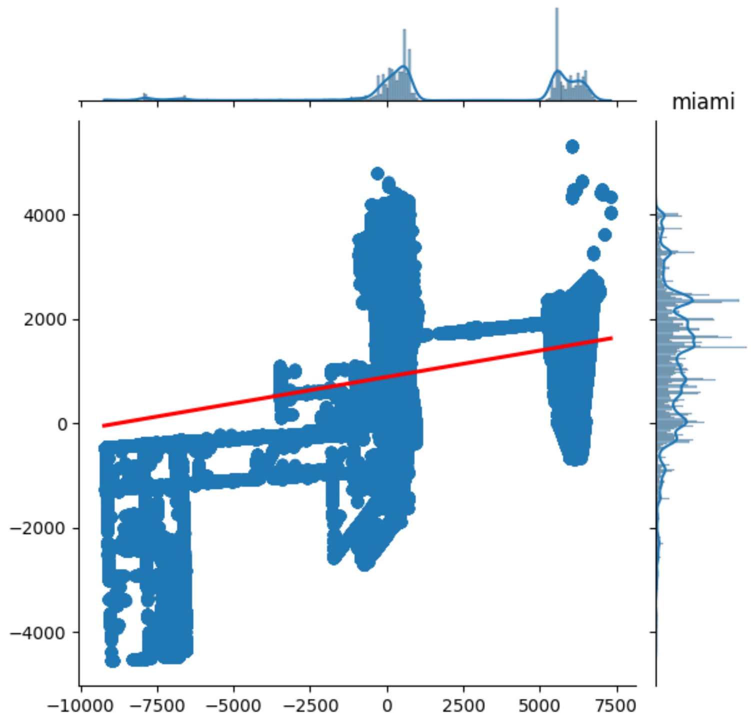


Figure 20. Joint plot of vehicle XY positions in Miami

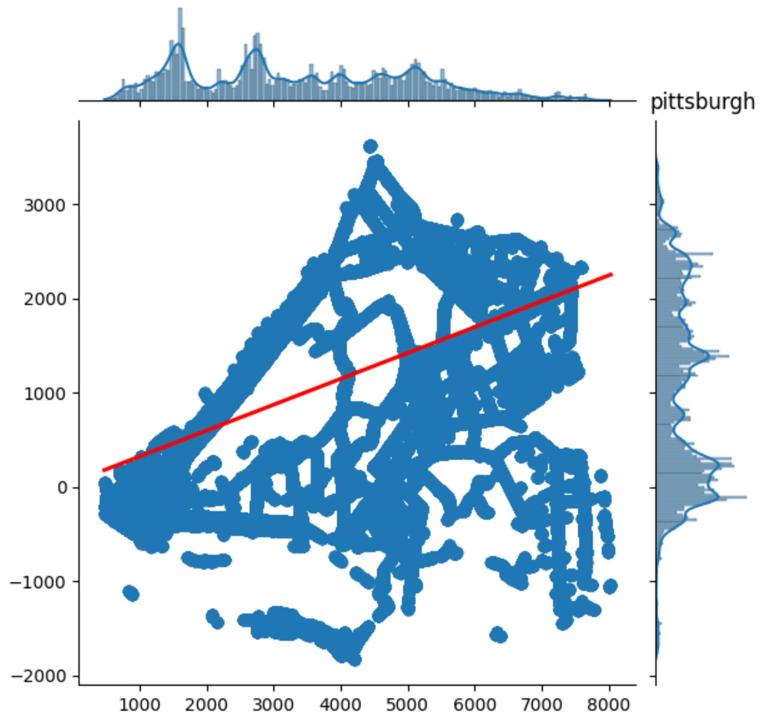


Figure 21. Joint plot of vehicle XY positions in Pittsburgh

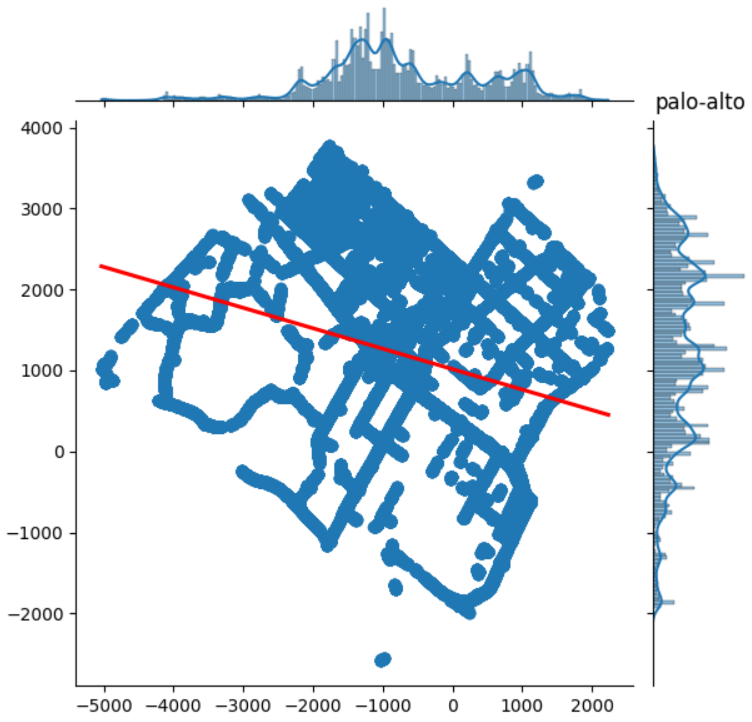


Figure 22. Joint plot of vehicle XY positions in Palo Alto

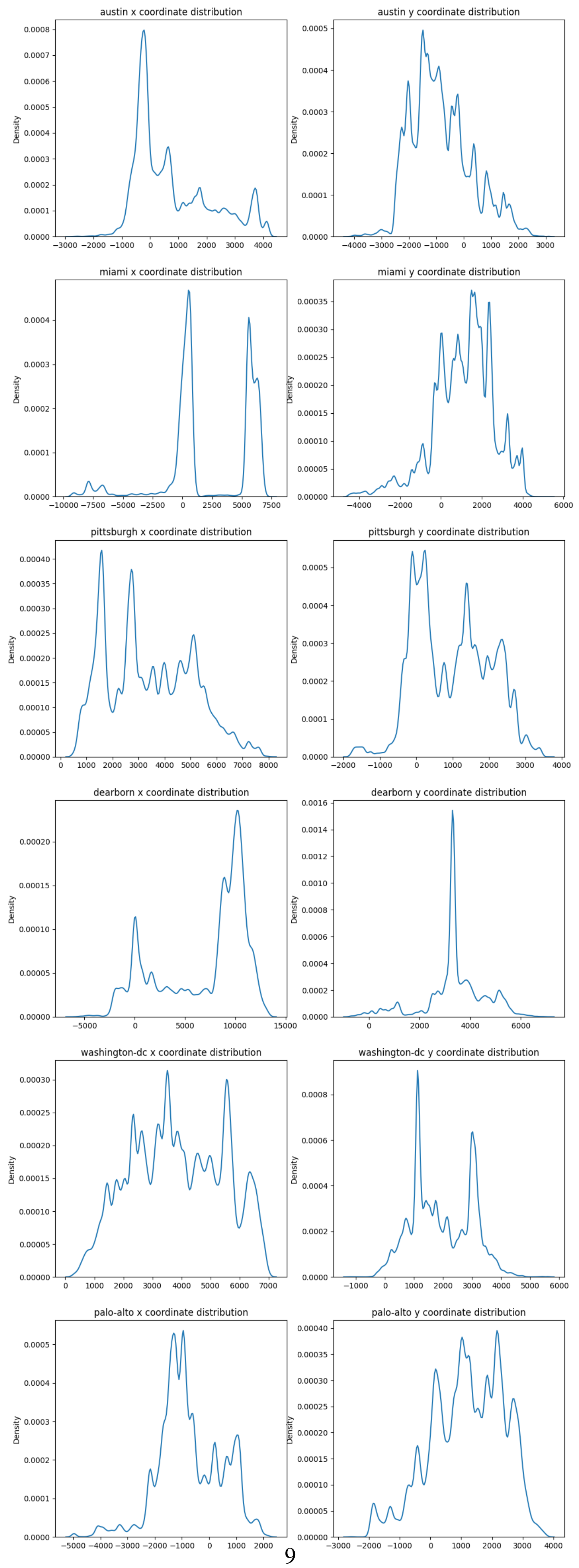

austin x coordinate distribution
austin y coordinate distribution
miami x coordinate distribution
miami y coordinate distribution
pittsburgh x coordinate distribution
pittsburgh y coordinate distribution
dearborn x coordinate distribution
dearborn y coordinate distribution
washington-dc x coordinate distribution
washington-dc y coordinate distribution
palo-alto x coordinate distribution
palo-alto y coordinate distribution
Density

Figure 23. x and y coordinate distributions for 6 cities

2.3 Processing

We split the dataset in 8/2 to create training/validation data. The first 80 percent of data goes to training and the rest 20 percent goes to validation. In terms of feature engineering, we let our model directly learn from the built-in X, Y coordinate features since we presumed that these two features already encode all the information needed to learn a trajectory. However, in a more careful afterthought, we believe if we were to engineer another one or two features such as acceleration, derived from XY coordinate displacements, they would perhaps encode more insightful information about the changes in a vehicle's trajectory, allowing our model to learn more subtle behaviors in the data, making better predictions. Given more time, this is certainly something we would like to revisit upon. Moreover, for normalization, we used min-max normalization for our task, which is defined as $x' = \frac{x - min(x)}{max(x) - min(x)}$. This min-max scaler makes sure that our input will be within 0 and 1, mitigating outliers of significant magnitude, and helps us to better understand our data. As to the city information, we plotted and examined the varying XY trajectory positions for different cities to helps us better see the general trends among different cities. The EDA ultimately helped us decide that we would train 6 separate models, each specializing in making predictions for one of the six cities, so that we could push each model to maximize its predictive strength under each city's locality constraint.

3 Machine Learning Model

In this section, we will introduce the machine learning models we used to make the prediction, as well as the project pipeline.

3.1 Linear Regression Model

Take linear regression model as an example, it regards the input data points as x axis, and calculate a y value based on the optimal k, b value we trained. The math formula is y = kx + b.
The model class that linear regression models belong to is the "linear model". This is consistent with how linear regression models are instantiated from the Scikit-learn library, as "sklearn.linear_model.LinearRegression".
This simple regression model did not perform well at all since the relationship between X positions and Y positions cannot be explained by a linear line, as already shown by each city's joint plots we presented in section 1.3. The next step for us is to come up with a model that would take the geographical representation of XY positions, as revealed in each city's scatter plots above, into account and recurrently pass it on so the XY position from one point in time is indicative of the possible XY position in the moment following.

3.2 Project Deep Learning Pipeline

The general pipeline of this project goes as follows:

- Perform train-validation split on the training dataset, then load the splitted training data into data loader to prepare for training.
- Build corresponding model class, define its constructors, forward, predict, and such primary model class methods.
- Feed in the data loader and train the model using its default parameters and hyperparameters; debug any mistakes and inconsistency that halts the training and program execution along the way.
- Inspect the loss curve and validation score to briefly examine model's training and validation performance.
- If the primary inspection is satisfactory–for example, the loss curve indicates that the model converged quickly with very little mistakes–feed the model with the testing set and obtain

the bench-marking predictions; make a bench-marking submission to the competition and record the baseline results.

- Following the step above, if the baseline model training and prediction result are satisfactory, we then repeatedly recycle the procedures from all above to tune model's hyperparameters.
- Make our final submission with the current model class using the one model instance with the optimal hyperparameters. If this submission does not yield satisfactory competition performance ranking relative to other group's model predictions, we will reorient ourselves to building a new model class, and re-run the whole pipeline procedures as described so far.

For deep learning models, since we did not introduce any additional features to our data through feature engineering, the input features that we selected are the X and Y coordinates, which in turn output X, Y features. The only thing about the features that is different between the input stage and the output stage is that the XY 2D feature vector outputs to greater length (from 50 to 60) than its input counterpart.
We eventually decided on deploying a Multi-Layer Perceptron model to solve our task since we observed that among all our alternative models, the MLP model is best at capturing linear or nearly-linear trajectories, which appear to be the predominant type of trajectories throughout all 6 cities. For our MLP model, the model architecture can be summarized as a 5-layer perceptron, where the first hidden layer takes the input and transforms it into data compatible with the hidden layer dimension and the last layer takes the hidden information from the layer before and transforms it into output data of format that's compatible with the 60 by 2 output feature vector. During the intermediate hidden layers, the trajectory information flows through and the corresponding weights at each layer can be optimized for with respect to our loss function, for which we decided to be MSE loss again since it makes most sense with quantitative data. Lastly, after many trials, we selected LeakyReLU function as our non-linear activation since effectively prevents the vanishing gradient problem when our MLP model becomes deep, as well as controlling for exploding gradient problem by scaling the gradient with a slope hyperparameter.

3.3 Trials of Different Models

We tried overall 3 different classes of model (excluding the simple linear regression we used to gain intuition about the data and project pipeline):

1. Multi-layer Perceptron

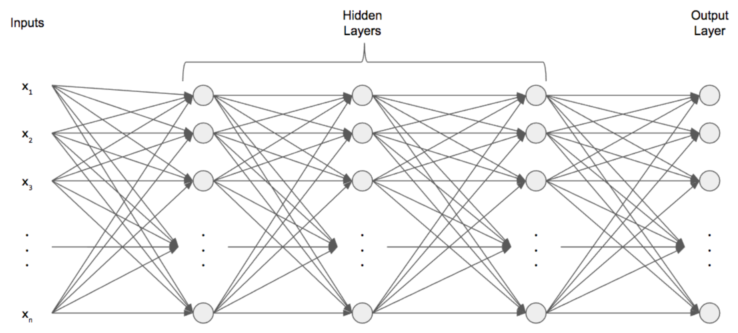


Figure 24. MLP

   (a) Data preprocessing: min-max normalization
   (b) Architecture: 5-layer perceptron model
   (c) Parameters: input dimension=100, output dimension=2, and hidden dimension uniformly=128.
   (d) Activation function: LeakyReLU
   (e) Loss function: MSE loss
   (f) Regularization technique: none

2. Encoder Decoder

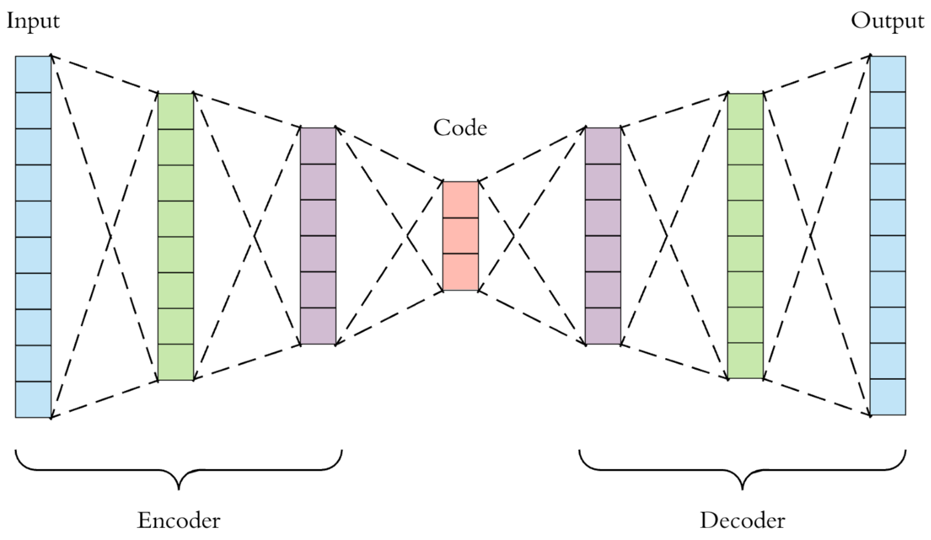
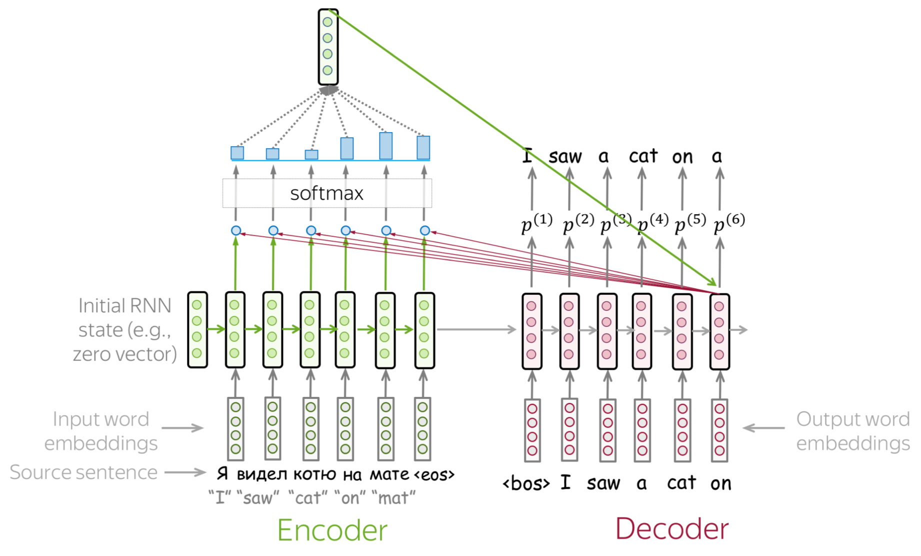


Figure 25. Encoder-Decoder

(a) Data preprocessing: standardization
(b) Architecture: consists of an encoder and a decoder component, where each is made up of a 3-layer perceptron network.
(c) Parameters:
    i. Encoder: input dimension=100, hidden dimensions=64,32, output dimension=32
    ii. Decoder: input dimension=32, hidden dimension=64, 120, output dimension=120
(d) Activation function: LogSigmoid
(e) Loss function: MSE loss
(f) Regularization technique: none

3. Encoder Decoder with Attention Mechanism



softmax
I saw a cat on a
$p^{(1)}$ $p^{(2)}$ $p^{(3)}$ $p^{(4)}$ $p^{(5)}$ $p^{(6)}$
Initial RNN state (e.g., zero vector)
Input word embeddings
Source sentence
Я видел котю на мате <eos>
"I" "saw" "cat" "on" "mat"
Output word embeddings
<bos> I saw a cat on
Encoder
Decoder

Figure 26. Encoder-Decoder with Attention Mechanism

(a) Data preprocessing: standardization
(b) Architecture: consists of an encoder and a decoder with attention component, each component relies on LSTM units
(c) Parameters: input dimension=2, output dimension=2, 128 hidden LSTM units per encoder and per decoder, 1 LSTM layer, 0 dropout probability

i. Encoder: input dimension=2, 128 hidden LSTM units, 1 LSTM layer, 0 dropout probability

ii. Decoder: output dimension=2, 128 hidden LSTM units, 1 LSTM layer, 0 dropout probability

A. Attention layer: linear layer with input dimension=130 and output dimension=50

B. Attention combine layer: linear layer with input dimension=130 and output dimension=128

C. LSTM: input size=128, hidden units=128, 1 layer, 0 dropout probability

D. Output layer: linear layer with input dimension=128 and output dimension=2

(d) : Activation function: Sigmoid (default)

(e) Loss function: MSE loss

(f) : Regularization technique: none

We tried MLP models because it is the most intuitive and also good at capturing and predicting linear and mostly-linear trajectories.
We then tried Encoder Decoder RNN models since it recurrently takes in information and pass them down to the following stages of model training. The information that happen earlier in time would influence what our model gain from training with the information passed in later down the stream–this is exactly what happens with driving!
We tried to improve our Encoder Decoder's performance by introducing the attention mechanism since it would help solve the challenge of preserving long term dependency, which is again what we presume to be important for predicting the trajectory of a vehicle since where a vehicle re-positions to in the next second has to depend on where it is currently and where it was one second ago.

4 Experiment Design and Results

4.1 Training and Testing Design

For the training and testing, we used UCSD datahub platform GPU as the computational platform.

The optimizer we used for the models is Adam (adaptive moment estimation) optimizer. Different with the stochastic gradient descent, which gives the same learning rate for all weight updates and the learning rate does not change, Adam optimizer derives different learning rates for different parameters from estimates of first and second moments of the gradients. In this task, the learning rate that we used is 0.001. The exponential decay rate for the first moment estimate is 0.9. The exponential decay rate for the second-moment estimates is 0.999. And epsilon is 1e-8. Because of the self-tuning feature of Adam optimizer, we got good results without tuning the parameters.

We used the encoder decoder, Seq2Seq with attention, and MLP to make multi-step predictions for each target agents. The input is the trajectory data of past 50 timestamps. The output of these models is the predicted next 60 timestamps' trajectory.

To utilize the city information, we trained one model per city using different city training data, and that's 6 models in total. For each city, we tuned the model parameters and made predictions for the corresponding city's testing data. After making predictions for all six cities, we combined the predictions based on the required format.

For each city, we trained 10 epochs. The batch size is 64. We only trained 10 epochs because training one epoch using the entire training takes a long time: It takes about 158s to train the Multi-Layer Perceptron model for one epoch.

4.2 Experiment Results

Below are prediction performances of model designs:

Models	Public Test MSE
Encoder Decoder	1180.30894
Seq2Seq with Attention	705.00347
MLP	62.38054

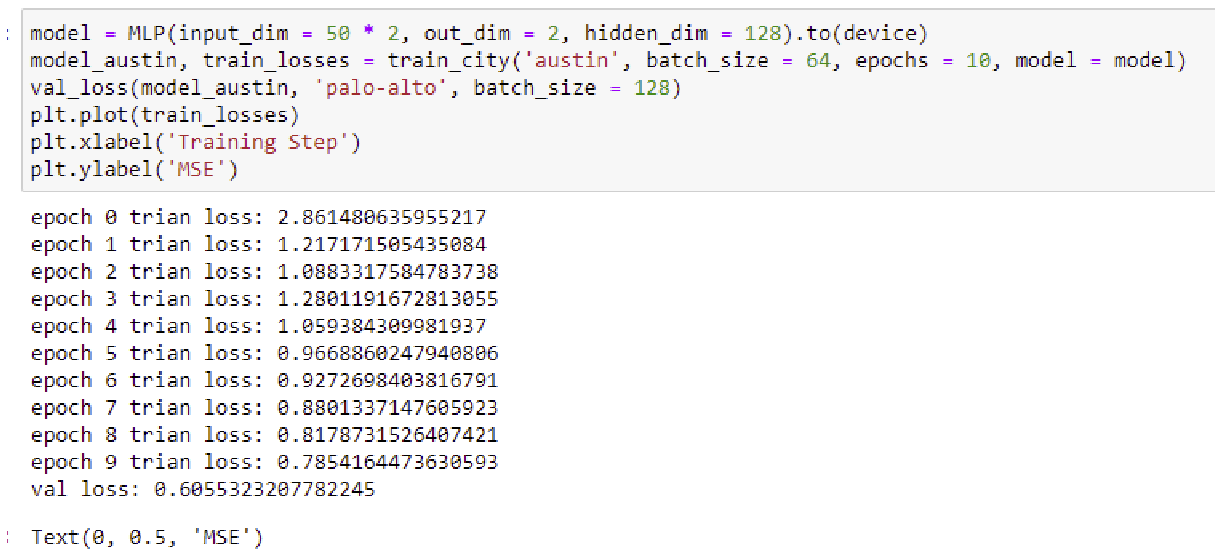
Table 1. Prediction Performances of Model Designs

From Table 1, we can see that the MLP is the best performing model, so the more complex models like the Seq2Seq model do not necessarily have better performance. The training time for the Encoder Decoder model is the fastest among three models, and it took about 2 minutes to train on the entire dataset for 10 epochs. The training time for the MLP is much slower compared to the Encoder Decoder model, and it took about 30 minutes to train on the entire dataset for 10 epochs. There are 10 parameters in the Encoder Decoder model, 5 parameters in the Seq2Seq model, and 9 parameters in the MLP model.

4.3 Best-performing Model's Performance

In this section, we will present the performance of our best-performing model - MLP.

To visualize the experiment results, we will use the MLP model trained on "austin" dataset as an example. The training loss (MSE) over 10 epochs is visualized below, and the training losses of other 5 models trained on other cities' dataset have the similar trend.



```
model = MLP(input_dim = 50 * 2, out_dim = 2, hidden_dim = 128).to(device)
model_austin, train_losses = train_city('austin', batch_size = 64, epochs = 10, model = model)
val_loss(model_austin, 'palo-alto', batch_size = 128)
plt.plot(train_losses)
plt.xlabel('Training Step')
plt.ylabel('MSE')
```

```
epoch 0 trian loss: 2.861480635955217
epoch 1 trian loss: 1.217171505435084
epoch 2 trian loss: 1.0883317584783738
epoch 3 trian loss: 1.2801191672813055
epoch 4 trian loss: 1.059384309981937
epoch 5 trian loss: 0.9668860247940806
epoch 6 trian loss: 0.9272698403816791
epoch 7 trian loss: 0.8801337147605923
epoch 8 trian loss: 0.8178731526407421
epoch 9 trian loss: 0.7854164473630593
val loss: 0.6055323207782245
```

```
Text(0, 0.5, 'MSE')
```

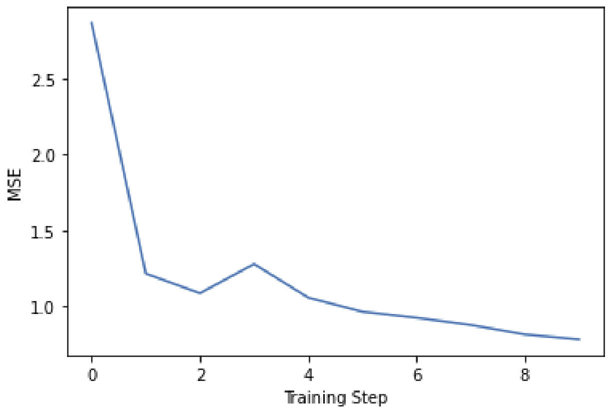


Figure 27. Training Loss

We randomly sampled 10 training samples from the "austin" dataset and compared their ground truths and predictions.

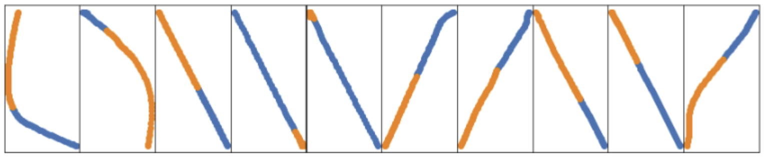


Figure 28. Ground Truths

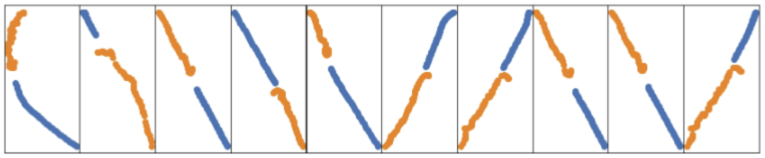


Figure 29. Predictions

Our current ranking on the leaderboard is 27, and the final test MSE is 61.78307 (private score).

In summary, the training loss of all six models followed the shape of the exponential decay. The models can accurately predict the direction of the car if the the car is moving in a line, and the model can also roughly predict the trajectory if the car makes some small turns. In addition, as we can see from the training samples, there is always a small deviation between the start of the prediction and the end of the input.

5 Discussion and Future Work

We conducted normalization in the data pre-processing section. The reason we need normalization is that large target values can lead to large error gradient values, which causes weight values to change dramatically, making the learning process unstable. After normalizing the input values, the training process is more stable and faster.

We think the biggest bottleneck in this project is choosing a suitable model. We started with the encoder-decoder model, then tried Seq2Seq with attention, finally it turned out that the MLP is the one that performs the best.

We found that using the right model is the most helpful in improving our ranking. Also, increasing the number of epochs is helpful in improving the ranking too.

For a deep learning beginner, we would advise him/her to start with simple models, do lots of research online to see what other people did when they were dealing with similar tasks, and see if we can use similar approaches. Also, it is important to start early so that you will have enough time for the training process.

For future works, we can do more feature engineering such as utilizing the social context by counting the number of neighbors for each agent. We can also try other models such as the stacked-LSTM model and transformers. Also, if we had more time, we'd definitely want to train the model using training data in more epochs, perhaps 100 epochs, while imposing regularization techniques at the same time to prevent overfitting, to see if the performance can be better. Also, we can incorporate learning rate scheduler to help speed up training and model convergence.

References

[1] Sampurna Mandal, Swagatam Biswas,Valentina E. Balas, Rabindra Nath Shaw and Ankush Ghosh. (2020) *Motion Prediction for Autonomous Vehicles from Lyft Dataset using Deep Learning*. Research Gate.

[2] Song, Haoran and Luan, Di and Ding, Wenchao and Wang, Michael Y and Chen, Qifeng. (2022) *Learning to Predict Vehicle Trajectories with Model-based Planning*. PMLR.

[3] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, Silvio Savarese. (2016) *Social LSTM: Human Trajectory Prediction in Crowded Spaces*. IEEE.

[4] Chang, Ming-Fang and Lambert, John and Sangkloy, Patsorn and Singh, Jagjeet and Bak, Slawomir and Hartnett, Andrew and Wang, De and Carr, Peter and Lucey, Simon and Ramanan, Deva and Hays, James. (2019) *Argoverse: 3D Tracking and Forecasting With Rich Maps*. IEEE.