CSE 151B Project Final Report

William Sun, wis003@ucsd.edu, https://github.com/wis003/CSE151B

1 Task Description and Background

1.1 Problem A

The task is to predict and forecast the motion of agents tracked by an autonomous vehicle. It is important to develop models to accurately predict the future positions of these agents for autonomous vehicles to work properly and safely. In the real world, autonomous vehicles need to quickly determine future positions of important agents based on visual/sensor data, and thus a successful deep learning model could be the key to solving this problem.

1.2 Problem B

According to the original Argoverse 2 paper [1], they describe various methods tested on the motion forecasting problem, where they use a constant velocity method, a feed forward neural network method, an LSTM, and WIMP. They emphasize that even though the basic constant velocity prediction method provides a relatively accurate baseline (average L2 distance = 7.75), WIMP ultimately performs significantly better than all the other approaches (average L2 distance = 3.09). The WIMP paper [2] describes the architecture as "a recurrent graph-based attentional approach with interpretable geometric (actor-lane) and social (actor-actor) relationships that supports the injection of counterfactual geometric goals and social contexts." Thus, whereas this method seems to be the most powerful and promising model to utilize, it requires information and context about each tracked agent, but our dataset only gives positional data over time. Thus, intuitively the ideas that seemed most promising for our dataset would be to start with the constant velocity method, or use a basic feed forward neural network.

1.3 Problem C

Where the data is sampled at 10 timestamps per second, the input to the prediction task is an input array of 50 timestamps, with each time stamp having an (x, y) coordinate. Thus, the input array has shape $(50, 2)$. After processing through the model, output needs to predict the position of the agent for the next 60 timestamps, so the output array has shape $(60, 2)$.

Thus, the prediction task will consist of taking in 50 positional coordinates which are ordered in time (5 seconds), and output the next 60 predicted positional coordinates in time (6 seconds).

Using this abstraction, a deep learning model that can successfully solve a problem with this type of input/output, it could also solve any other problem that can be described with time relevant 2D data that needs to predict future 2D data. This is possible because the deep learning model can fit any problem that can be abstracted to the same description.

Preprint. Under review.

2 Exploratory Data Analysis

2.1 Problem A

The training data set has input shape $(203816, 50, 2)$ and output shape $(203816, 60, 2)$. The test data set has input shape $(29843, 50, 2)$ and output shape $(29843, 60, 2)$. For each trajectory, the model input dimension is $(50, 2)$ and its output dimension is $(60, 2)$.

The meaning of these input and output dimensions can be interpreted as the following. Where the data is sampled at 10 timestamps per second, the input to the prediction task is an input array of 50 timestamps, with each time stamp having an (x, y) coordinate. Thus, the input array has shape $(50, 2)$. After processing through the model, output needs to predict the position of the agent for the next 60 timestamps, so the output array has shape $(60, 2)$.

Figure 1 below shows a batch of 4 data samples, where the blue dots are 50 input positions, and the orange dots are 60 output positions.

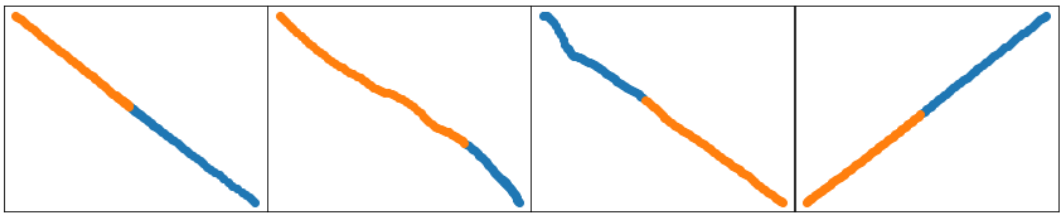

Figure 1: Batch 3000 sample for Austin

2.2 Problem B

The distribution of input positions for all agents is shown as a heatmap in Figure 2. The distribution of output positions for all agents is shown as a heatmap in Figure 3. The distributions of positions for each city is shown in the following heatmaps (Figure 4-9).

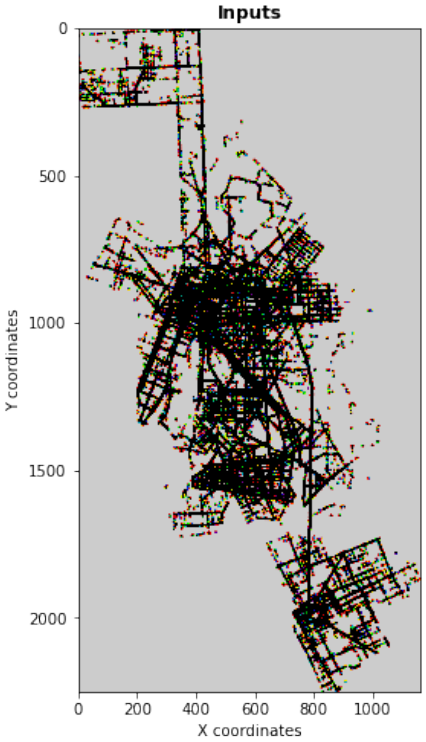

Figure 2: Input 2D heat map

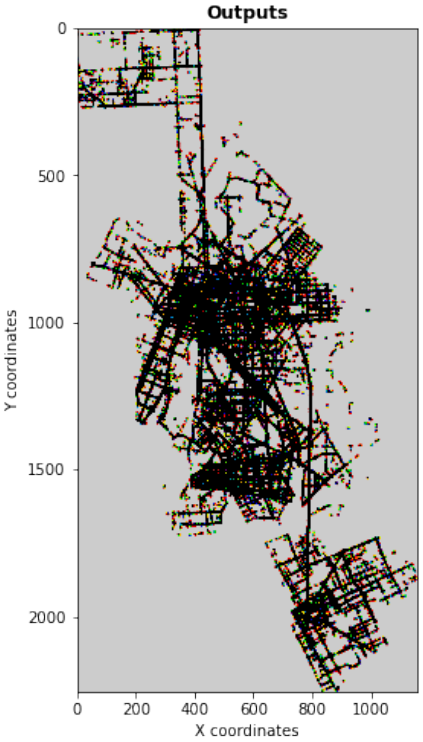

Figure 3: Output 2D heat map

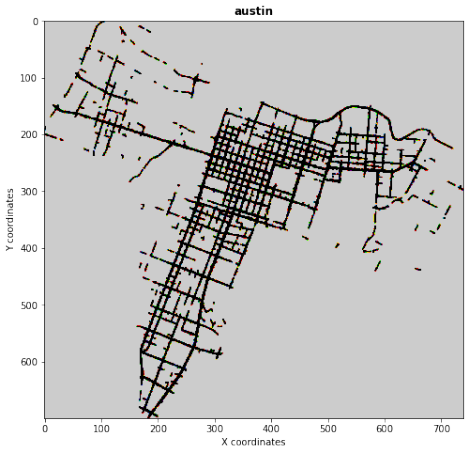

Figure 4: Austin 2D heat map

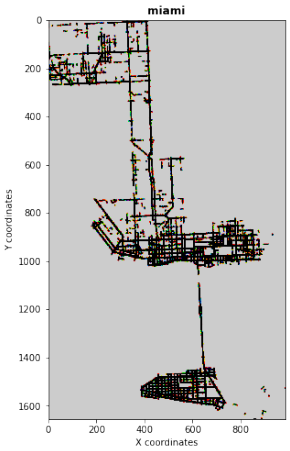

Figure 5: Miami 2D heat map

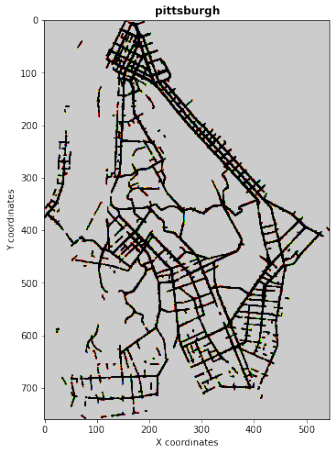

Figure 6: Pittsburgh 2D heat map

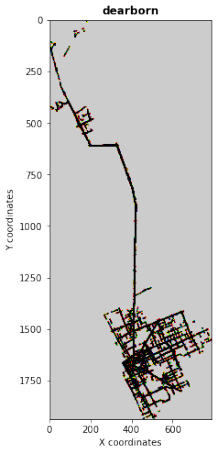

Figure 7: Dearborn 2D heat map

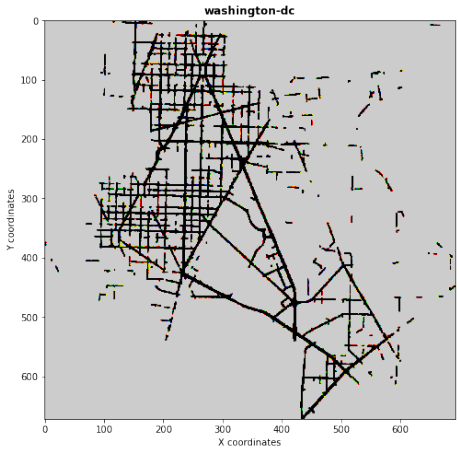

Figure 8: D.C. 2D heat map

Figure 9: Palo Alto 2D heat map

The positions shown in these heatmaps illustrate that the motion/direction of agents are generally very linear and straight. Thus, in the bonus exploratory analysis, I will dive into the information provided by the motion of the agents.

2.3 Bonus exploratory analysis

We visualize the frequency of average velocity in each input set.

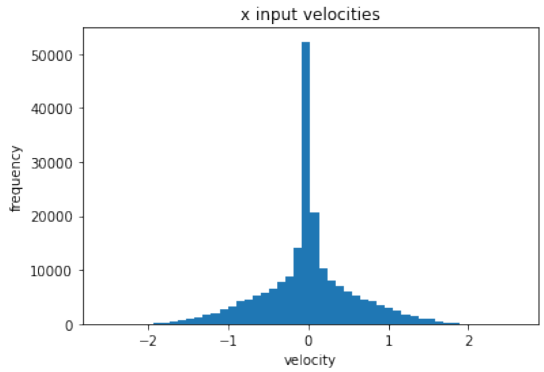

Figure 10: X input velocity

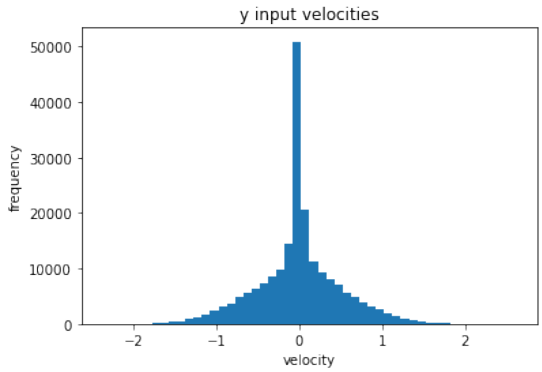

Figure 11: Y input velocity

Next, we visualize the frequency of average velocity in each output set.

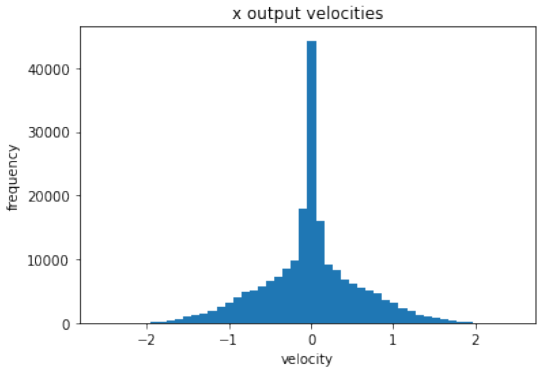

Figure 12: X output velocity

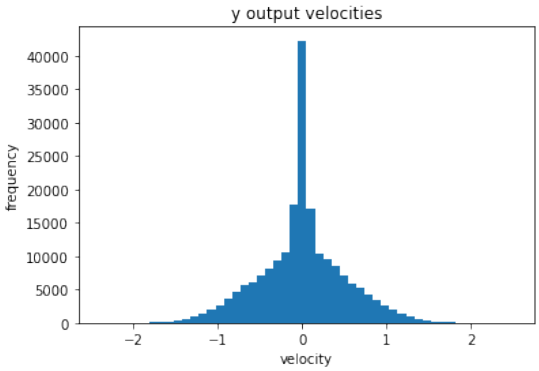

Figure 13: Y output velocity

Next, we visualize the frequency of average acceleration in each input set.

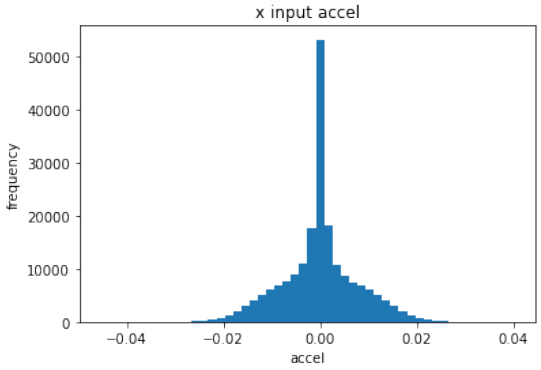

Figure 14: X input acceleration

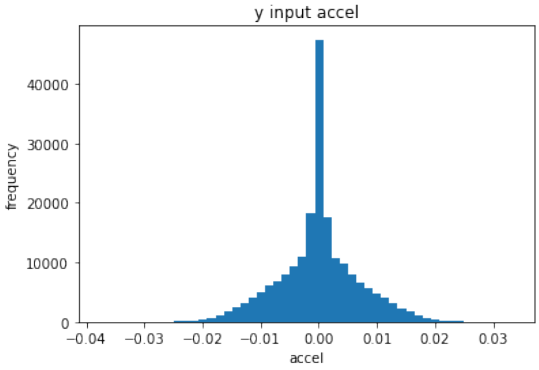

Figure 15: Y input acceleration

Next, we visualize the frequency of average acceleration in each output set.

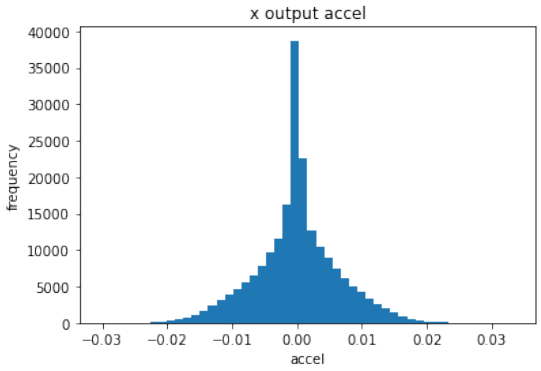

Figure 16: X output acceleration

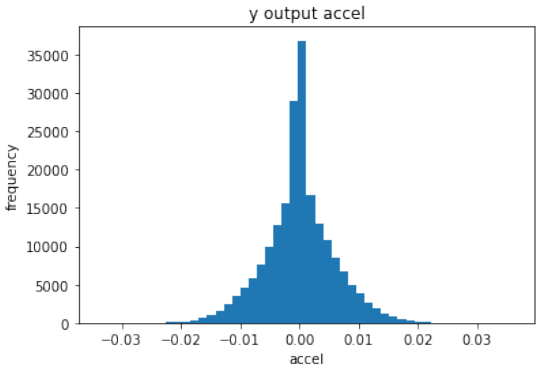

Figure 17: Y output acceleration

Next, we take the the previous data and visualize them in 2D.

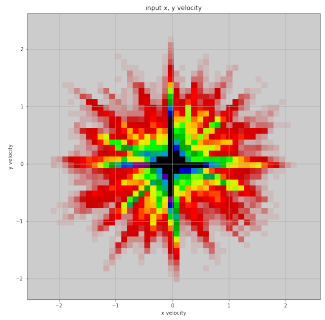

Figure 18: 2D input velocity

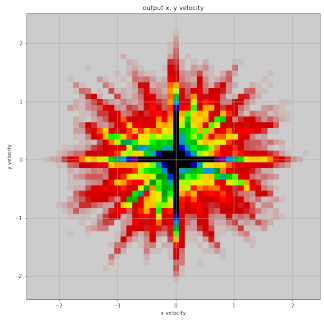

Figure 19: 2D output velocity

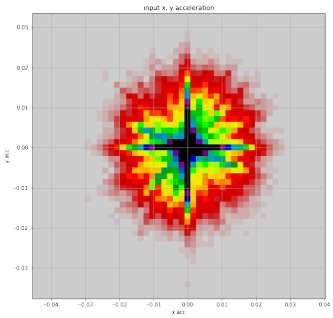

Figure 20: 2D input acceleration

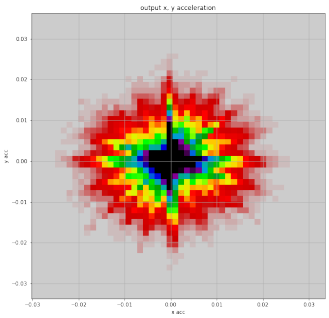

Figure 21: 2D output acceleration

Next, we show the frequency of differences between input and output data.

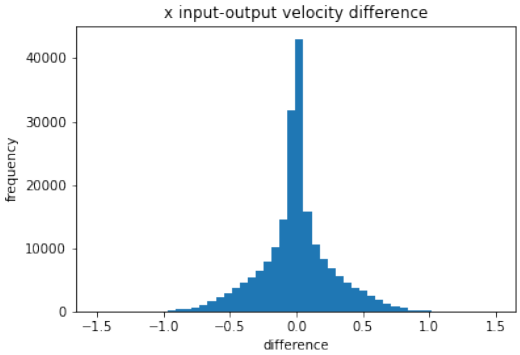

Figure 22: X velocity difference

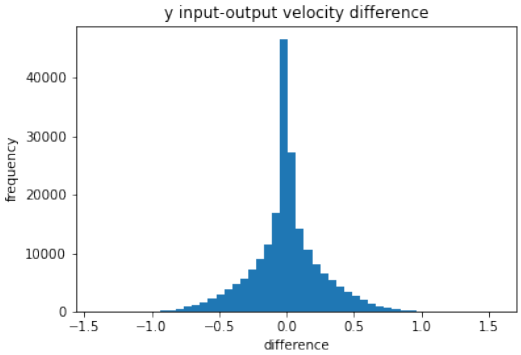

Figure 23: Y velocity difference

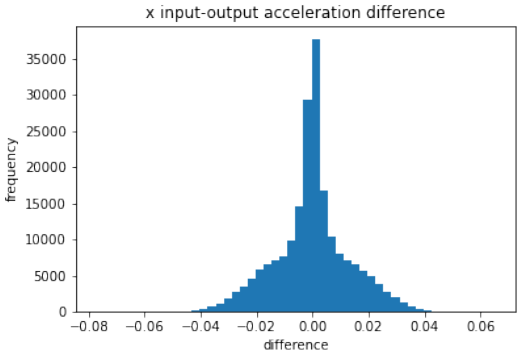

Figure 24: X acceleration difference

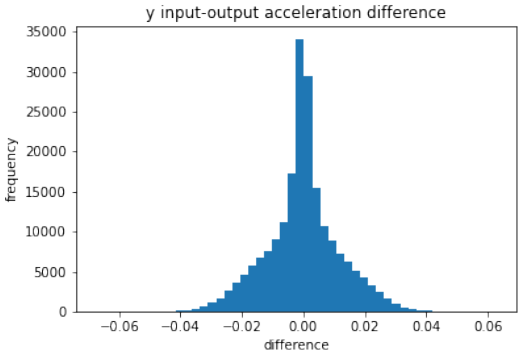

Figure 25: Y acceleration difference

The further exploratory data analysis reveals that a linear prediction model will be very accurate for the data set, because we can see the density of velocity/acceleration is greatly concentrated near zero, and also the difference between the input and output data is typically zero. Thus, in general a straight linear prediction based on input data should accurately predict the output positions.

2.4 Problem C

Initially for trying out different models, an 80/20 training/validation split was used. This was to determine if a model was working as intended (loss decreasing over iterations). In this split, the training set was 80 percent of the original 203816 size dataset, and the validation set was the remaining data.

No feature engineering was used, because only (x, y) coordinates were given, so nothing needed to be done. Since this data and number of features are already limited, there was no attempt to limit this further.

No normalization was done because the performance of the linear regression model was already at the top of the leaderboard. Given more time however, I would have tried to implement a min-max normalization scheme.

The city information was not specifically used to modify prediction. In general, data for all cities were used in each epoch to train the model.

3 Machine Learning Model

3.1 Problem A

The input and output features used for a simple machine learning model (linear regression) is as follows. I wish to use the most data possible to make a prediction, so the input feature will be all coordinates from timestamps before the current timestamp, and the output is the current timestamp prediction. For example, for the first prediction, there are 50 input timestamps, and the output will be the 51st timestamp. Then, the final prediction will have 109 input timestamps, and the output will be the 110th timestamp. Thus, there are 60 separate models total in the linear regression implementation, where each model is prediction the 51-110th timestamps respectively.

The model class picked here is sklearn's default linear regression, which uses sum of squares as the loss function.

3.2 Problem B

Multiple types of deep learning pipelines were tested for the prediction task, from feed forward neural networks to LSTMs to transformers. However, they all shared the same input output structure, where the input layer took in a matrix of shape $(100, 1)$, which is just the flattened version of the input series data $(50, 2)$. Then, it is fed through a series of hidden layers depending on the model, and the output was the same between all models, having an output shape of $(120, 1)$, which is just the flattened version of the output series data $(60, 2)$.

As stated previously, feed forward neural networks, LSTMs, and transformers were architectures that were tested. Similar to the linear regression model, sum of squares was the loss function used. My ideas and observations were distinctly disparate during experimentation. I initially thought the more powerful methods would be better (LSTM and transformer), but clearly the scores show that the simpler feed forward neural network worked better. This makes sense based on the data analysis because the linear trends, and simple patterns in the data are more geared towards the simpler models.

3.3 Problem C

1. Encoder/Decoder (Discussion 7 Architecture)
    - Encoder - 3 hidden layers, Decoder - 3 hidden layers, Batch-size - 4, lr - 1e-3
2. Deeper Architecture
    - Encoder - 4 hidden layers, Decoder - 4 hidden layers, Batch-size - 4, lr - 1e-3
3. Transformer Model (torch.nn.Transformer)
    - Encoder - 4, Decoder - 4, Batch-size - 4, lr - 0.1, dropout - 0.2, dmodel - 2, nhead - 2
4. Encoder/Decoder with dropout
    - Encoder - 3 hidden layers, Decoder - 3 hidden layers, Dropout - 0.2, Batch-size - 6, lr - 1e-2
5. Mean Velocity with Acceleration
6. Median Velocity with Acceleration
7. Linear Regression (sklearn.linearmodel.LinearRegression), Default Parameters
8. Adaboost Regression (sklearn.ensemble.AdaBoostRegressor), Default Parameters

4 Experiment Design and Results

4.1 Problem A

The computational platform used for both training and testing was DSMLP, where the environment was specifically "ucsdets/scipy-ml-notebook:2022.2-stable, Python 3, nbgrader (1 GPU, 8 CPU, 16G RAM)"

For deep learning models, the optimizer used was PyTorch's Adam. Learning rate was tested multiple times, where 0.01 and 0.001 both worked equally well. Learning rate decay and momentum were kept using PyTorch's default parameters and were not touched.

In our best deep learning implementation, multistep prediction for each target agent was made through an encoder-decoder architecture, where the decoder outputs a 120 element array, reshaped into the expected prediction output shape of $(60, 2)$.

The city information was not specifically used to modify prediction. In general, data for all cities were used in each epoch to train the model.

The MLP model was trained in 40 epochs. The batch size was 4. It took on average 2.5 minutes to train the model through one epoch, in which one epoch went through the entire training data set.

Table 1: Summary of experiment results

Design	Description	Score	Training Time (min per iteration)
(1)	Encoder-Decoder	1733.08	5.2
(2)	Transformer	13579726.63	7.5
(3)	Linear Regression	21.58	0.12
(4)	Average Velocity w/ Acceleration	71.73	N/A
(5)	Median Velocity w/ Acceleration	54.86	N/A

The above design choices where made due to the inspiration of the discussion code. I started developing and testing the model beginning with the sample discussion code, and began to tweak hyperparameters and tried different architectures to see what generally works the best. In the end, computational limitation did play a factor in the design choice. For example, I tried the LSTM and transformer implementations from PyTorch, but they both take too long to train (>10 min per epoch), while the training loss was unreasonably high (in the millions). Thus, simpler models for prediction that were more lightweight were chosen.

4.2 Problem B

Table 1 compares a collection of varying models that were tested for this prediction task. It is clear that the simpler models performed much better than complex models in both speed and accuracy. For example, the simplest models (velocity w/ acceleration) didn't even need training time, and they performed relatively well with low loss scores. Additionally, it is interesting that the median performs better than the average metric, and this reveals that the test data is sensitive to outliers. Overall, we see that the simple linear regression model performs the best, and this makes sense due to the linear nature of motion over time in the data. Linear regression trained relatively quickly, on the scale of seconds, whereas the complex models took minutes per iteration. We see that the complex models performed quite poorly, particularly the transformer, which indicates that the model complexity was not able to fully learn the data patterns under a short amount of training time. Due to the constraints of the training platform as well as the amount of project time, it was unreasonable to further tune these architectures compared to the success of the simpler models.

In terms of parameters for the machine learning and deep learning models, linear regression has the least. For training and testing, there were 60 separate models in total, one per output. Each model has the number of weights based on the input plus one extra parameter from the bias. Summing the 60 models up, there are 4830 parameters total. Next, the Encoder-Decoder model had 8 hidden layers total. This calculation for the number of parameters is as follows: $101 * 64 + 65 * 64 + 65 * 32 + 33 * 16 + 17 * 32 + 33 * 64 + 65 * 64 + 65 * 120 = 27848$.

4.3 Problem C

The training loss (RMSE) value over training steps for my best performing deep learning model is shown below.

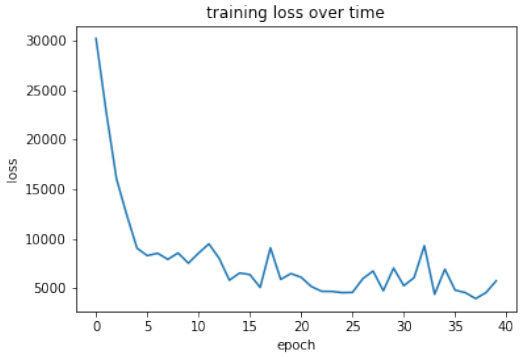

Figure 26: Loss over time

Figures 27-30 visualize the ground truth and the linear regression model's predictions on a 2D plane. Clearly the linear regression model very accurately models linear data, but struggles more when there are nonlinear outliers. Luckily, the data analysis shows that most data is very linear.

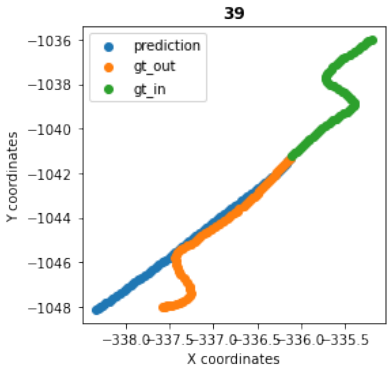

Figure 27: Austin sample 39

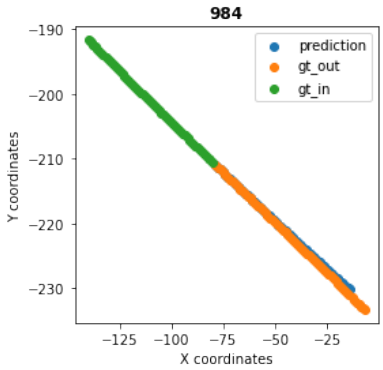

Figure 28: Austin sample 984

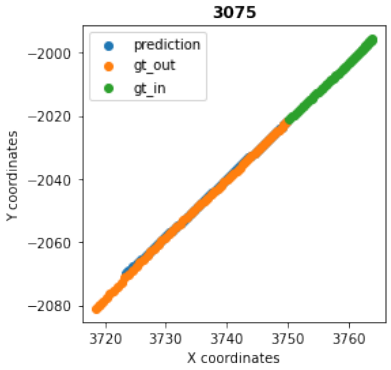

Figure 29: Austin sample 3075

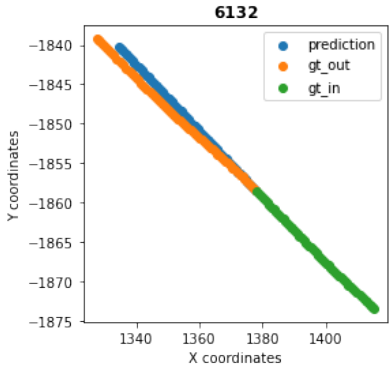

Figure 30: Austin sample 6132

My current ranking on the leaderboard is 12, with a score of 21.92163.

5 Discussion and Future Work

5.1 Problem A

The most effective feature engineering strategy depends on the application. For this prediction task, we see through data visualization and experiments on different models that generally a linear model suits the data. Thus, handling outliers is extremely effective and important for an accurate regression

model. We see that simply changing from average velocity to median velocity improves the model significantly, which highlights the effectiveness of certain feature engineering techniques in this application.

Data visualization was most helpful in the early stages of the project. Understanding that most agents moved in a straight line indicated that a linear model and a simple model would work well. Accordingly, the model design was very helpful in decreasing the loss. As shown in section 4.1, linear regression performed much better than any other model tested. Finally, hyper parameter tuning would be necessary to beat out the top scores which were all very close.

The biggest bottleneck in this project was setting up the training environment and training itself. With DSMLP, I was having issues with running out of RAM while training, especially with the more complex deep learning models, as well as ensemble training for regression. Next, I tried Google Colaboratory, but it was very unreliable, where is crashes randomly overnight. Thus, any model that needed a long time to train was really time consuming and considered a huge bottleneck.

Due to these training bottlenecks, I would highly recommend deep learning beginners to start first with data visualization to understand the task, and then begin experimentation with the simplest possible models first.

If I had more computation resources and more time to test models, I would revisit the LSTM and tranformer models, as I would like to learn to tune these models to give accurate results. Additionally, there was more pre/post processing ideas I had that I didn't have time to test. For example, I wanted to try a nearest neighbor approach applied additionally onto the linear regression model. I also wanted to try filtering methods for post processing, such as using a Kalman filter.

References

[1] Argoverse 2: Next Generation Datasets for Self-Driving Perception and Forecasting, Wilson et al. *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[2] What-If Motion Prediction for Autonomous Driving, Khandelwal et al. *arXiv preprint arXiv:2008.10587*, 2020.