CSE 151B Final Project Report

Samuel Zhou	Jiyoung An	Alan Li	Rama Del Maestro
szhou@ucsd.edu	j6an@ucsd.edu	ahl001@ucsd.edu	rdelmaes@ucsd.edu

GitHub Repository: https://github.com/yuyeon/CSE-151B-Final-Project

1 Task Description and Background

Problem A The task we are addressing is a sequence modeling task, where the input is a sequence of 'scenes' which include positions and velocities of various objects on a street, including a single object which is designated the target agent. In this input 'scene', we aim to predict the trajectory of our target agent at several consecutive intervals of 0.1 seconds into the future.

This task is important because it is a vital step towards developing safe and effective autonomous driving systems. With the growing interest in self driving cars, one task these cars must learn is to accurately predict the future positions of objects around them, on the road. One of the basic skills of any driver is the ability to assess the situation around themselves on the road, a type of 'scene', to determine whether it is safe to switch lanes, for example. Such a decision is based on the motion/future positions of other cars and objects in the 'scene', which is why our task is important.

Problem B Similar trajectory prediction tasks have been attempted before. In a paper by Altché et al., an LSTM network was trained to predict vehicle trajectories on highways, up to 10 seconds into the future, to good results [1]. Another paper by Becker et al. examines trajectory prediction in 5 different models: various RNNs including a Recurrent Encoders, a Seq2Seq model, and a Temporal Convolution Network, and found that a Recurrent Encoder model with a single-layer LSTM performed best [2]. All of these studies involve taking advantage of the sequential nature of the problem and using previous states to predict future outputs. These methods allow for weight-sharing techniques and learning across timesteps.

Problem C This task involves predicting the positions of a specific object within a 'scene', for the next 30 timesteps. This can be formulated as a type of regression task, because we are predicting an (x, y) position for some object, where x and y are both real values.

Formally, we are given 'scenes', where each scene is a data sample. Each scene tracks the positions and velocities of up to 60 objects for 19 timesteps, where each timestep is separated by 0.1 seconds. Given this data, we want to predict the position of a specified object for a next 30 timesteps (separated by 0.1 seconds intervals, so the next 3 seconds).

The input for our task is a scene with multiple features for each agent through the initial 19 timesteps. Initially, we decided to use four features for each agent, representing horizontal and vertical input positions and velocities. Each scene is then a matrix of dimension $(60, 19, 4)$. The 60 rows represent up to 60 objects in the scene; if we have only k tracked objects such that $k < 60$, then the last $60 - k$ rows are dummy values. Then we have 19 columns, to represent each distinct timestep. The final dimension is 4 because we stack the position and velocity of each object on each other. We have a position in the form of (x, y) and velocity in the form of (v_x, v_y), and we concatenate them along the 3rd dimension. When adding more features, they could be concatenated along this third axis.

The output for each scene is a matrix of dimension $(30, 2)$, where the ith row represents the position (x, y) of our tracked object at timestep i.

For a model trained for the above task, we could potentially adapt it to other sequence prediction tasks. If it can effectively learn the association between surrounding objects, position, and velocity, we could try adapting it to other tasks such as music generation, which can be represented similar to

position/velocity on an x, y graph through formats such as a sonogram. Of course, this may require editing the input/output sequence lengths. But we could adapt a model trained on the Argoverse task by adding extra input and output layers in order to translate the sequence shapes as required. This model could also perhaps perform well on tasks such as speech or text generation, which involve completing a certain amount of the data, or filling in a sequence.

2 Exploratory Data Analysis

Problem A Our initial look into the data revealed that the training data set contains 205942 scenes. The test data set we were provided with contains 3200 scenes. Each input scene contains multiple variables, but in general, it seemed that the most relevant inputs are the input positions and input velocities. These inputs have shapes of $60 \times 19 \times 4$. There are up to 60 tracked cars in a scene, and their horizontal and vertical positions/velocities are tracked initially for an initial 2 seconds at 10 Hz, resulting in 19 frames. The output has dimensions of $60 \times 30 \times 2$ and represents just the positions of the same objects for an additional 3 seconds. If there are less than 60 objects, the corresponding positions/velocities of both the inputs and outputs are filled with zeroes. The presence of an actual car can also be determined using the car_mask or track_id variables. Another thing to note is that while the training dataset has output positions for all 60 agents, we are only interested in the output positions of one target agent. This makes our actual output dimensions 30×2.

If we decide to add or remove features from our inputs, the input shape could change. However, the output shape will be consistent as we are always trying to predict ouput positions of a single objects over 30 timesteps.

One way we could choose to process the data is by just taking in the input positions of the target agent. A visualization of the input positions and target output positions of a target agent for a randomly sampled scene is shown below in Figure 1. The paths of vehicles varies, but many of them are fairly easily predictable from a visual standpoint.

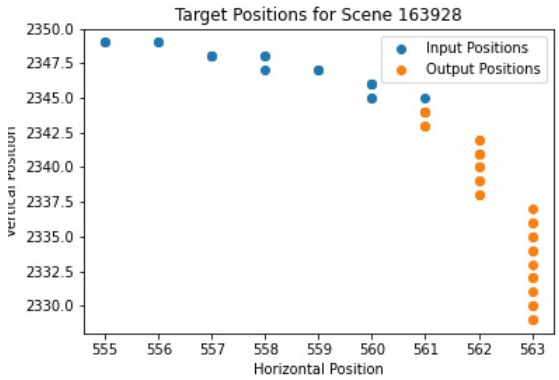

Figure 1: Input and Output Ground Truth Positions for Scene 163928

Problem B Our next investigation involved looking at the distribution of input and output positions for all non-dummy agents, which were extracted using the car_mask variable. The first frame was used for input positions, and the last frame was used for outputs. The resulting histograms of horizontal and vertical input and outputs positions is shown in Figure 2. These histograms showed that the distribution of input and output positions are nearly identical, which suggests that the changes in position over the total 5 second interval are minimal. For this reason, the proceeding exploratory analysis will only concern input positions. The histograms also revealed that there are two clusters of horizontal positions. We suspected that the reason for this was that there are two different cities in the data. To investigate this, we created a bivariate distribution of input positions and colored points by the city they belong to. The result is shown in Figure 3. There are clearly two clusters points which correspond to the two cities. It is also predictable that the 2-dimensional distribution of the data draws out street grid areas where the data were collected.

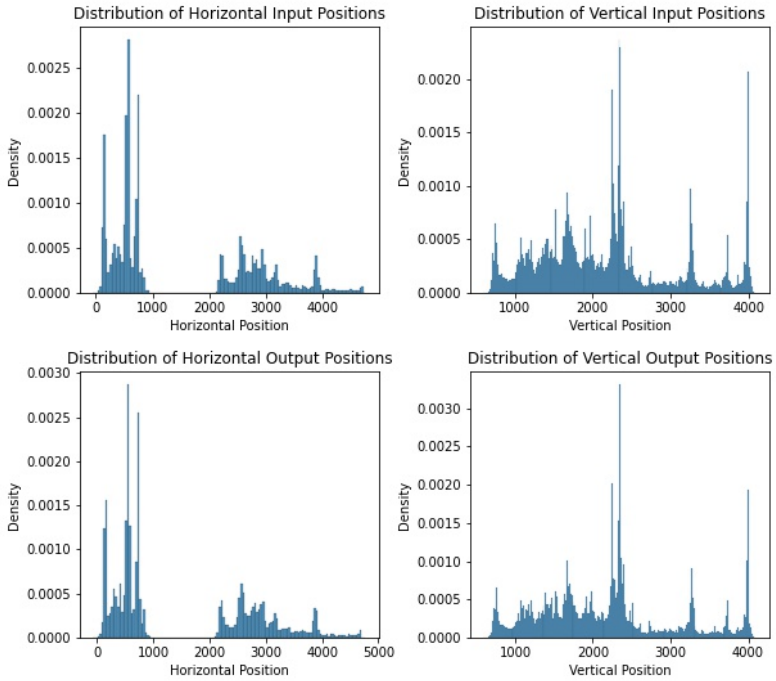

Figure 2: Distributions of Input and Output Positions

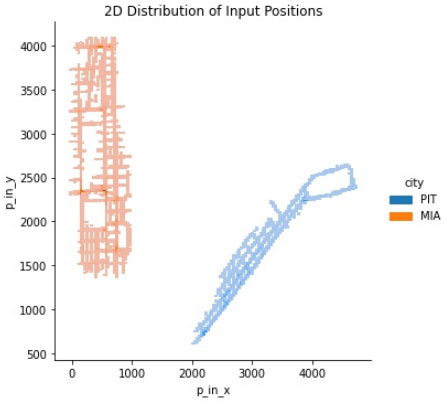

Figure 3: Bivariate Distribution of Input Positions by City

We also investigated the distributions of input positions of just the target agents. The distributions mostly appeared to be identical to the distributions of positions for agents in general. The results are displayed in Figure 4.

Following this, we looked into the distributions of input and output velocities of agents. For both inputs and outputs, we looked the distributions of velocities in general, velocities by city, velocities of target agents, velocities of target agents by city. The results of these analyses are displayed in Figures 5 and 6. One thing to immediately note is that the distributions of velocities differ greatly between the agents in general and the target agents. The general agent velocities are much more heavily weighed toward zero, which means that generally, the task involves tracking the moving objects in a scene rather than the mostly stationary non-target agents. We may wish to limit our inputs to targets at some point then as non-targets are sufficiently different from what we are interested in and may add substantial noise. It is also worth noting that the input and output velocities are slightly different in each of the groups of histograms. This means the objects are changing speed, and we must account

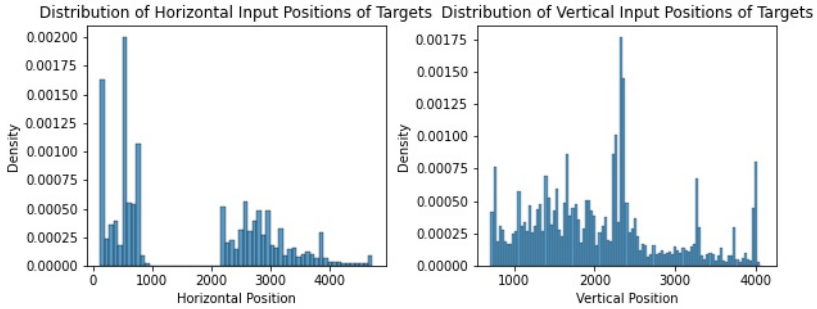

Figure 4: Distributions of Input and Output Positions of Targets

for this. There are also differences in the shapes and locations of input and output velocities between our two cities. The speeds appear to be slightly higher in Pittsburgh than they are in Miami. We may wish to take advantage of this information while building our model.

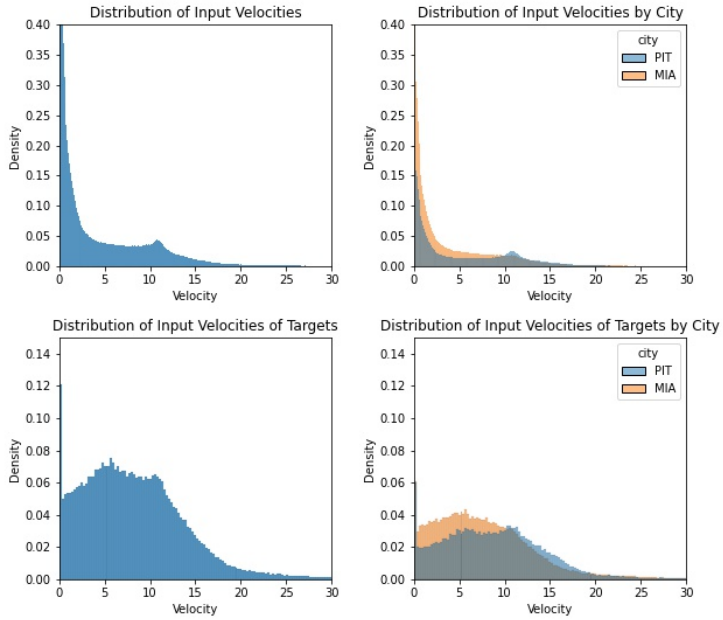

Figure 5: Distributions of Input Velocities of Various Groups

Problem C Initially, we did not really preprocess the data and just inputted the input positions and velocities of all agents at every timestep into our models. For the linear model, we did flatten the input matrix into one dimension and treated every additional agent's positions and velocities as extra features. We did not feel that much preprocessing was necessary and hoped that our models could learn to interpret which agents were more important.

Later, we decided to preprocess the data more. For one model, we took in only the input positions for the target agent instead of all agents. Then, we also engineered a feature to one-hot encode the city variable. We only needed one variable since we have only two cities. We felt this feature was important because our exploratory analysis revealed that the distributions of positions and velocities were quite different between the two cities. Simplifying our inputs was much more successful with our linear model. For our second model, we did not find that the simplified input helped our predictions, so we did not make any changes.

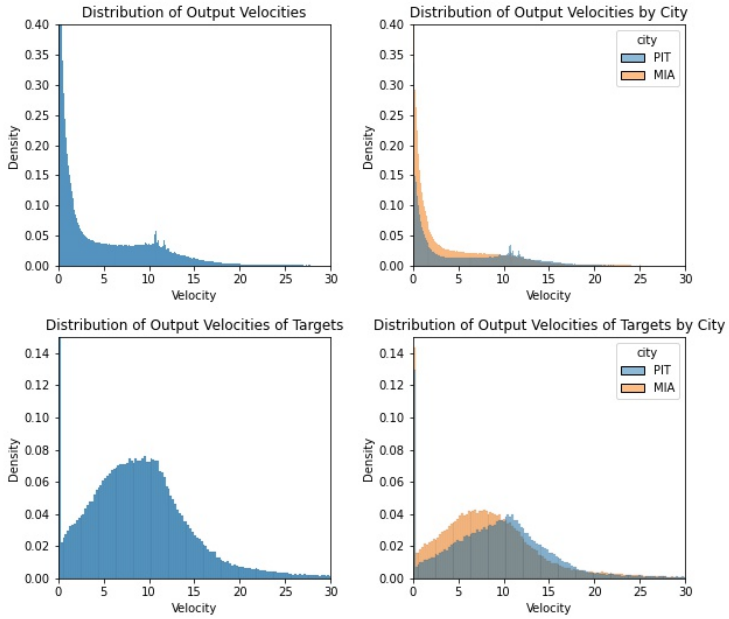

Figure 6: Distributions of Output Velocities of Various Groups

We did not normalize our data directly, but for some of our attempts, we performed batch normalization to inputs inside linear layers before activation functions. This was done for the same reason as one-hot encoding the city. By standardizing values, our models may have been able to better predict output positions because there were substantive differences between some values. Batch normalization also helps keep activation inputs in a reasonable range to make our optimization processes more effective.

We decided to not use lane information. We felt that this data was too complex for our current models. In the future, we may try to engineer some values out of this data, but we did not feel that it was necessary at this current point in time.

3 Deep Learning Model

Problem A For our linear model, the input we decided to use was just the input positions and velocities for the target agent across the 19 timestep, along with an array of one hot encodings of the city for that scene. This made our input shape 19×5. We then squashed this matrix into a one-dimensional vector of size 95. For our Seq2Seq model, we achieved the best loss when we inputted the positions and velocities for all agents. This made our input have size $19 \times 60 \times 4$.

For both models, the output is the positions of the target agent across 30 timesteps. This means our output has shape 30×2. Our output from the linear model is a one-dimensional vector of size 60, so we reshape that to match the desired output shape.

Our loss function is the mean squared error between our predictions and the ground-truth output positions. We chose to use MSE because we are essentially performing a regression task and trying to minimize the difference between our predicted positions and the actual positions. We also chose this loss function because it exaggerates the effect of significantly wrong predictions, and we want our model to be sensitive to these issues. MSE is also differentiable everywhere, which could potentially avoid some errors that functions like absolute loss would have.

We started with a simple linear regression model, because we are performing a regression task, and we wanted to start with a simple model in order to get a good grasp of the data and the task. It was a linear model with no nonlinearities, with not even a single activation function. It was just a single

linear layer, the simplest model we could've attempted. Surprisingly, we attained decent results on our first try.

We also decided to try a more complex model, in addition to the simple linear model, and for that we chose to use a Seq2Seq Encoder-Decoder architecture. We chose to use a one directional model, because we only needed to use past positions to predict future positions. An Encoder-Decoder architecture would work quite well in this case, because we take into account all past positions when predicting the next position in the sequence. In other words, the inductive bias of this architecture suited our task quite well. We also felt that a fixed size hidden vector should be enough, since our input would always be positions across a fixed number of time intervals, so we didn't use attention. This also allowed us to use a weight-sharing scheme to reduce the complexity of our model.

For our Seq2Seq model, we tried using the same input and output as the linear model, but we also tried using the matrix of positions of velocities of all objects in the scene, which would be input shape $60 \times 19 \times 4$. But for each timestep, we can consider the input shape as 60×4, although in practice it would be the same as passing in the entire matrix. The output in this case was the same, a one dimensional vector of size 60 reshaped into a shape 30×2 matrix. Between the two input output approaches, we didn't see much difference in performance.

In both cases, we attempted to use other layers alongside the overall model to varying degrees of success. For the Seq2Seq model, we tried varying numbers of layers, layer sizes (hidden unit size, for example), and types of layers. We chose to add or remove additional layers based on training loss after one or two epochs.

Problem B

- Linear Model - Single linear layer with no activation function
- Seq2Seq Encoder-Decoder Architecture
  - Encoder - Linear layer (output size 512), Batch Normalization Layer (output 512), ReLU activation, LSTM (hidden size 2048)
  - Decoder - Linear layer (output size 256), Batch Normalization Layer (output 256), ReLU activation, LSTM (hidden size 2048), fully connected layer + ReLU, linear layer

We used no references.

The first model we used was just a single fully-connected linear layer. There were no parameters to tune in this model because we must have the input and output shapes fixed. We did have to tune which optimizer we used, which included tuning learning rate and momentum values, however. We attempted to add more hidden layers between the input and output using fully-connected layers with nonlinear activation functions, batch normalization, and dropout layers. However, we could not find a combination of these layers with layer sizes or dropout percentages that resulted in better training losses than just the single fully-connected layer.

Our second model was a Seq2Seq model. We passed in the input data into an encoder LSTM with zeroed hidden states, took the last hidden states, and fed them into the initial hidden states of a decoder LSTM. We then sequentially used this decoder LSTM to predict future positions based on its previous output and hidden states as an input. The initial input was the final input positions of the target agent. With this model, there were many parameters to tune such as the hidden size, which we kept at 2048 for both LSTMs since it produced the best results. We also ended up adding fully-connected layers with batch normalization and activation functions alongside dropout layers to the LSTM inputs and outputs. We attempted many combinations of these layers until we found one that had the lowest training loss after a small number of epochs.

4 Experiment Design

Problem A For our training and testing, we used UCSD DataHub servers and the GPU provided to train and test our models. We saved our code in a GitHub repository, for easy access for all group members. We chose to use DataHub because we all had experience on the platform from homeworks and previous classes, and the environment was easy to setup and use. However, the training speed on DataHub was quite slow. In hindsight, we should have used DataHub to experiment with model

architectures, perhaps to tune hyperparameters or test how it works on a subset of the data, but trained the models locally for faster training speeds, or used Google CoLab.

We did not create a validation set, which was a mistake given how long training often took, because as a group we did not have much experience creating models and fine tuning hyperparameters. We also should've used a validation set to tune parameters. In fact, we could have automated the hyperparameter tuning and increased efficiency drastically through K-folds cross validation, but we thought of this too late. However, we did introduce stochasticity by shuffling batches from the data loader when tuning parameters and the number of layers.

We tried a number of optimizers alongside various parameters to see which gave the lowest training loss. For our linear model, we ended up using the Adam optimizer. For the Seq2Seq model, we used stochastic gradient descent. To tune the parameters of these optimizers, we would train a few sets of batches and look at the decay of the average training loss over these sets. If the loss exploded, we would lower the learning rate and/or momentum. If the loss didn't change much, we'd raise the learning rate. We often had to re-tune these parameters based on the parameters of the model we were using. During training, we also used a method to update learning rate (adaptive learning rate). If the average training loss of a set of batches increased by a certain percentage, we'd lower the training loss so we don't overshoot the local minima.

For multistep prediction, we tried both a linear model and Seq2Seq model. For our final linear model, we input a 1-dimensional size 95 vector to represent the input positions, velocities, and city of the target agent over the initial 19 timestamps. The model then output a length 60 vector, which represents the 30 timesteps multiplied by the 2 coordinates for each timestep for only one target agent. We felt that this model would be sufficient for this task since the model could learn the orderings of features in the input and output vectors. For our Seq2Seq model, we input a length $60 * 4 = 240$ vector, which represents the input positions and velocities of every object over a single timestep. The input into the decoder is a size 2 vector which represents the last position of the target agent. The decoder then outputs a size 2 vector as the next position. We feed this output and the new hidden states back into the decoder to get a new output. We repeat until we have 30 sets of positions. This model intuitively made sense to us since our next predictions should be dependent on our previous predictions.

We used 2 epochs for the linear model, and our batch-size was 64. The reason we used 2 epochs was because over the course of the 2nd epoch, we didn't see much change in the loss. On the other hand, for the Seq2Seq model, we tried higher numbers of epochs (as many as 300), but on DataHub the training was incredibly slow. This forced us to lower the number of epochs, as the server kept terminating. We chose a batch size of 64 because it minimized training time for the whole dataset while also providing as many updating iterations as possible in an epoch. Higher values saw equal training times, but we didn't want to iterate over the whole dataset too quickly and wanted to give our model many opportunities to update in an epoch. For the linear model, an epoch takes around a minute to train. For the Seq2Seq model, an epoch can take as long as 7 minutes.

5 Experiment Results

Problem A

Table 1: Primary Model Comparisons

Model	# of Parameters	Approximate Training MSE Loss	Training Time per Epoch
Simplified Linear	5,760	9.5	1 Minute
Seq2Seq	40,529,410	20,000	7 Minutes

Our linear model performed much better not only in terms of training loss, but it also has far fewer parameters and takes a much shorter time to train. The Seq2Seq model takes about seven times as long to train a single epoch and needed almost 7,000 times as many parameters to get this loss. The Seq2Seq model was also often unstable during training, even after tuning the optimizer parameters.

The loss would sometimes explode randomly, and we we were unsure why. Unless we can find a fixable fatal flaw in our Seq2Seq model or take a much better approach with the inputs, it seems that the simple linear model will be more successful.

To improve training speed, we generally tried to reduce the complexity of our model. We also played with different optimizers to see which gave the best short-term performance. Additionally, we increased the batch size until the training time didn't lower. This helped ensure we were fully using the parallelization capabilities of our platform.

Problem B Our best-performing linear model's training loss and best-performing Seq2Seq model's training loss patterns over a first epochs are shown below. In both cases, we see exponential decay. The linear model ended up with a MSE of around 9.5 The Seq2Seq model unded up with an MSE of around 15,000.

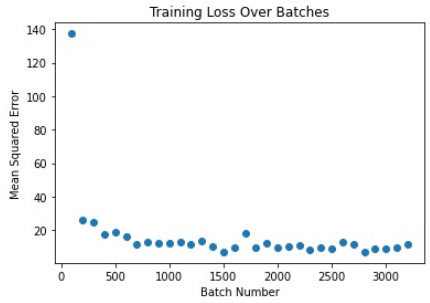

Figure 7: Linear Model Training Loss over One Epoch

Figure 8: Seq2Seq Training Loss over One Epoch (Loss in Scale of Hundreds of Thousands)

Some randomly selected predictions from our linear model are displayed below. The model certainly predicts better for some patterns than others, but predictions were always within the same relative region as the ground truth positions.

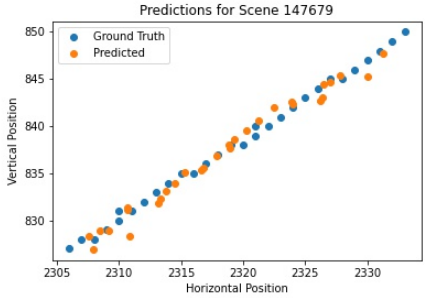

Figure 9: Linear Predictions for Scene 147679

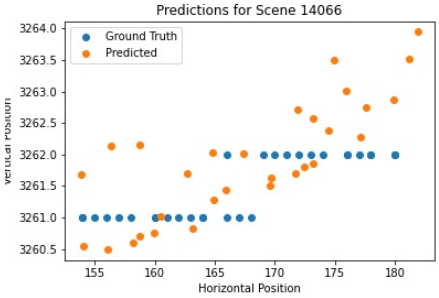

Figure 10: Linear Predictions for Sample 14066

We have also provided random predictions from our Seq2Seq model. It is critical to notice that our model almost always predicts positions in the same location. This indicates that our model has high bias and is too complicated. However, we could not find a way to simplify the model such that it performed better than it does here in terms of MSE. We tried reducing input size in the same way as with the linear model, reducing the hidden size, and stripping the model of all additional layers. However, the model failed to learn and the MSE remained even higher. It appears that our model roughly predicts near the mean of the input/output positions. This is interesting since we used MSE, and the direct minimizer of MSE is the mean. Perhaps in some way our model is learning to just predict the mean of the output positions rather than trying to make more sophisticated predictions with the input data. In any case, our implementation of the model is clearly flawed somewhere or just not suitable for the task.

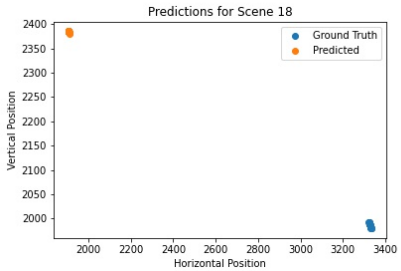

Figure 11: Seq2Seq Predictions for Sample 18

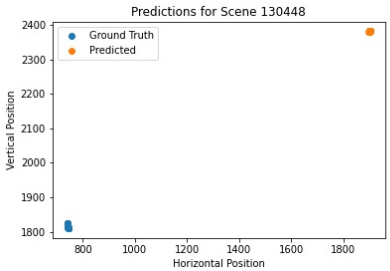

Figure 12: Seq2Seq Predictions for Sample 130448

Our final test RMSE is **2.71**. At the time of writing, this places our team at 30th on the public leaderboard. On the private leaderboard we have a RMSE of **2.82** and placed 29th.

6 Discussion and Future Work

Problem A For us, the most effective feature engineering strategy is to create features that simplify the data and can help create distinctions between inputs. Our one-hot encoding of the city was helpful because the two cities have obviously different position distributions. It was also helpful to get rid of all of the non-target agent features because they don't really help us more than the target agent features, and they add noise that can confuse the models.

Data visualization and hyper-parameter tuning were the most helpful in improving our score. After visualizing the data, we had a better understanding of some of its properties and could engineer better inputs. Tuning our hyper-parameters in our Seq2Seq model also greatly helped reduce its training loss. Choosing a proper optimizer was also an important choice and made our models learn much quicker.

Our biggest bottleneck in this project was the amount of data we had to process and the amount of time it took our resources to process the data. This was especially problematic when trying to build the Seq2Seq model since there were many hyper-parameters to tweak, and it took a significant amount of time to test each combination. This made it difficult to know whether we merely had an issue with our parameters or if our model implementation was just completely wrong. If we had smaller amounts of data and better resources, we could test more combinations in a shorter amount of time and more quickly locate potential issues.

For a beginner, we would advise that they pay special attention toward visualizing the data so they can create effective features. They should try to simplify their inputs to reduce training time and frustration. It's not reasonable to assume that a complicated model could easily filter out unhelpful noise. They should also start with simple models and see how far they can go with just those simple models. They might be surprised by how effective minimalist approaches can be.

We would also advise deep learning beginners to use a combination of grid search and k-fold cross validation in order to tune hyperparameters, rather than tuning them manually. This would save a lot of time, and improve efficiency, as we lost a lot of time training a model and trying to find the best hyperparameters.

If we had more resources, we'd want to explore other LSTM/RNN models, like just a single LSTM, or a graph neural network approach. We were limited in the number and complexity of the models we could try due to how long it took to train a single Seq2Seq mode. There are many other ways we could conceptually approach this problem, and it would be enjoyable to see which methods would perform best on this problem and related ones. We'd also want to see how our approach can perform on another dataset with certain modifications.

References

[1] Florent Altché and Arnaud de La Fortelle. An lstm network for highway trajectory prediction, 2018.

[2] Stefan Becker, Ronny Hug, Wolfgang Hübner, and Michael Arens. An evaluation of trajectory prediction approaches and notes on the trajnet benchmark, 2018.