CSE 151B Project Final Report

Bright Lian blian@ucsd.edu

Michael Askndafi maskndaf@ucsd.edu

Anton Panis aopanis@ucsd.edu

Gaopo Huang ghuang@ucsd.edu

Github Link https://github.com/MikeDafi/CSE151B-Kaggle-Competition

1 Task Description and Exploratory Analysis

1.1 Problem A

The task in this assignment is to predict the motion of vehicles in a given scene. Using the past positions and velocities of all the vehicles as well as the positions and directions of lanes in a scene, we want to predict the motion of a particular vehicle. Our model for prediction needs to incorporate and consider not only the position and velocity of the vehicle in question, but also environmental factors such as traffic caused by other vehicles, changing directions of lanes as well as the city in which the scene is taking place. Studying problems of this type is extremely useful in the real world as they heavily relate to the optimization of autonomous vehicles which are increasingly popular. If we can predict the safe and correct motion of a vehicle in a given scene, then this could help create safer and more reliable self-driving vehicles.

1.2 Problem B

When forecasting data given a set a number of inputs, sliding window protocols have become the standard in industry to process data. In the article from Google Scholar, we discovered that Recurrent Neural Networks allow for these variable-sized inputs in which we can rely on the previous time-steps to provide more information for future predictions. The article also refers to LSTMs as a RNN that will have less redundant information and can absorb larger time-steps which benefits the case where we want to include categorical and multiple numerical columns. Another article,this employs convolutional neural networks with LSTM's for stock prediction, where we can keep the input constant, by concatenating the input and output. This usually has good performance on images but can still provide predictions using local perception and weight sharing. First, having the CNN layer then the LSTM layer calculations followed by back-propagation. The article also recommended to use root mean square error, which aligned with the competition's grading format.One other article that tried a different method for time-step forecasting, introducing Stacked Gate Recurrent Units. Giving a step-by-step example, they detail how to include different types of data for the environment. They acknowledged the over-fitting issue which is why they use dropout and multiple LSTMs. They also use an optimizer we hadn't heard of, Adam, and the linear activation function. This proved to perform the best on their project as well as ours.

Preprint. Under review.

1.3 Problem C

For each scene, we are provided a dictionary consisting of 11 items- lane, lane_norm, city, scene_idx, agent_id, car_mask, track_id, p_in ,p_out, v_in, v_out

$lane$ - (k,3) numpy array of floats where k = number of lanes

$lane\_norm$ - (k,3) numpy array of floats where k = number of lanes

$city$ - (k,3) numpy array of floats where k = number of lanes

$scene\_idx$ - unique non-zero integer

$agent\_id$ - a string of the form "00000000-0000-0000-0000-0000XXXXXXXX" where X is 0 to 9

car_mask - (60, 1) numpy array of float 1's and 0's

$track\_id$ - (60,30, 1) numpy array of strings where string format is "00000000-0000-0000-0000-0000XXXXXXXX" and X is 0 to 9

p_in - (60, 19, 2) numpy array of floats

v_in - (60, 19, 2) numpy array of floats

p_out - (60, 30, 2) numpy array of floats

v_out - (60, 30, 2) numpy array of floats

2 Exploratory Data Analysis

2.1 Problem A

The training data size is 205942, the validation/test data size 3200. The training and test data are a dictionary with 11 elements, some of these elements such as 'city' are just a single value, while other elements such as $'p\_in'$ are a $60 \times 19 \times 2$ 3D array. Generally speaking, the raw input data can be encompassed within a four dimensional array. A detailed description of each of the raw input fields can be found in Question 1 Problem C. The raw output is just p_out and v_out which are each $60 \times 30 \times 2$ dimensions since there are up to 60 vehicles for which we predict the positions and velocities x and y for 30 time steps.

For one sample data of the training data set, it includes both the raw input and the raw output. The data contains 11 fields including: city, lane, lane_norm, scene_idx, agent_id, car_mask, track_id, p_in ,p_out, v_in, v_out. The p_out and v_out fields are used in the output and the remaining fields are part of the input.

2.2 Problem B

Here are some histograms for input and output data for all agents for all the data from the training set:

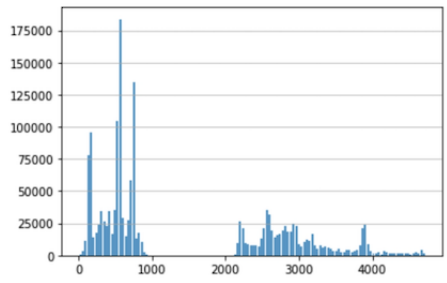

Figure 1: Input position x values all agents

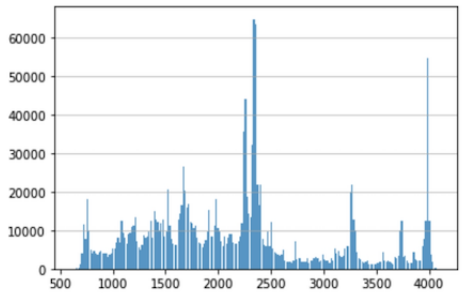

Figure 2: Input position y values all agents

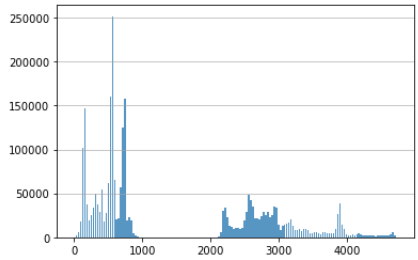

Figure 3: Output position x values all agents

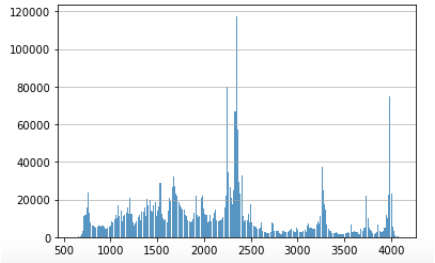

Figure 4: Output position y values all agents

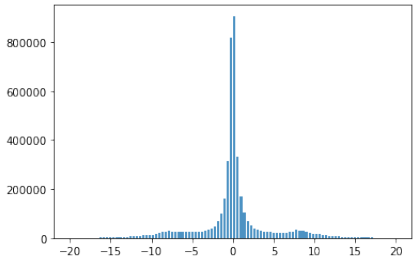

Figure 5: Input velocity x values all agents

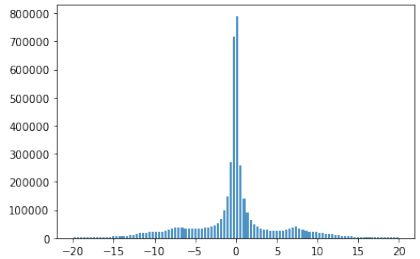

Figure 6: Input velocity y values all agents

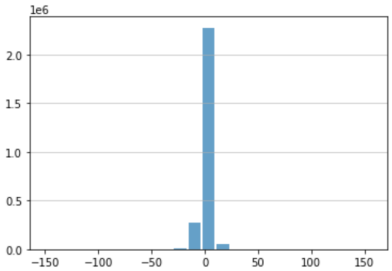

Figure 7: Output velocity x values all agents

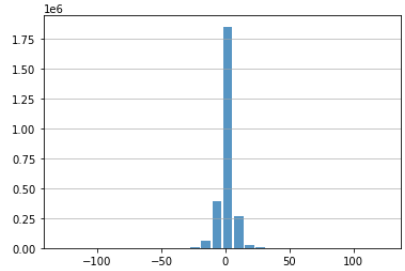

Figure 8: Output velocity y values all agents

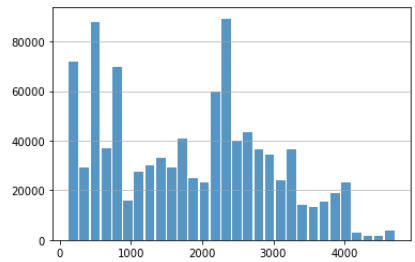

Figure 9: Position values target agent

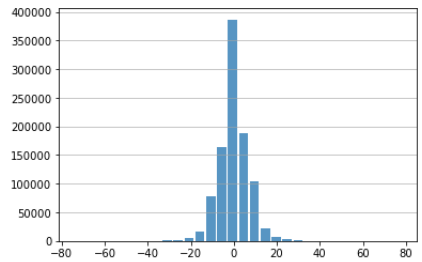

Figure 10: Velocity values target agent

2.3 Problem C

For data processing, we tried many approaches. First we just tried to pass in the positions and velocities of all 60 vehicles into the model and that produced some results, but was far from optimal. Next we found a way to add lanes and lane norms to the input by making the input 4 dimensional and an extremely large matrix since some scenes had over a thousand lanes. That sort of worked, but it was highly inefficient and the model ran extremely slowly as to be expected and the input matrix was very sparse. Next we tried to normalize values in order to standardize them and we did this by finding the min and max in the entire inputs and then normalizing them all by the same scale to make all values between -1 and 1. This seemed to make things better, but we still had the same issue of a massive 4 dimensional matrix as the input and normalizing it took a bit of time, also our error was deceptively low since we also normalized the ground truth so our error seemed much lower than it actually was. Finally, we decided a better way to incorporate lane information is to append the lanes and lane norm information together and then reshape that matrix to fit the shape of the positions and velocities matrix and then torch.cat them together. This made the run time a lot better since the input matrix was a lot smaller, but we simply had to hope that the model could learn how to interpret the lane information. In the end, we also removed normalization of the input since it took a long time and the results still seemed to be fine without normalization. We also included the city information as part of the input.

3 Deep Learning Model

3.1 Problem A

The input includes one-hot encoded vector for cities, if in Miami encode 0 otherwise encode 1. For the other matrices, we will be using a dstacks. One dstack entailing all the lanes in the scene, which includes the lane norms and lanes. Another dstack is just the input's position and velocity for the first 19 time-steps. The third dstack is the output's position and velocity for the last 30 time-steps. When concatenating these 4 variables, the dimensions match because we shape the lane information to 60 * x * 4, where x = $\lceil \text{lanes in scene}/240 \rceil$. As for the other pieces of information relating to agent id and the ids of each car, we use that data in a separate collate function when validation testing because we are looking at a single car per scene. The loss function is MSE(Mean Squared Error) loss. There were no alternative options chosen because Kaggle relies on RMSE(Root Mean Squared Error). Therefore, this loss function gave an accurate depiction of our model's expected performance. Cross-Entropy Loss wouldn't work here due to the fact there aren't classes we are clustering the data into. We had initially chosen to have a LSTM with a Conv1d layer. The idea was to have a sliding window that incorporated accumulated gradient,dropout, and 16-bit float comparisons to increase efficiency. An LSTM looks at the step dimensions so having a 60 * 8 time-step dimension, we can concatenate the input discussed above without breaking the dimension boundary. However, this model plateaued at mse loss of 2000. Then, simplifying the problem we realized we can have a fixed sized input of 19 time-steps + one hot encoded vector. So, we tried a simple sequential neural network of 2 linear layers without dropout and relu activation, which proved to have an mse of 2. We made the output based on the position and velocity, despite including more in the previous model.

3.2 Problem B

- Sliding-Window LSTM Initially, we had chosen to have an LSTM layer with a Conv1d layer. Having 2 LSTM layers with 3000 hidden layers each, the conv1d layer was expected to output 60 * 4 which is supposed to be the prediction. From the training method, we append this prediction to a list of predictions but use the ground-truth to append to the input vector of 19 time-steps. To be more effficent, we concatenated the input's 19 time-steps with the output's 30 time-steps. Then, when predicting, we took the last 30 time-steps, excluding the last prediction. This proved to be more efficient in runtime.
- Fully Connected Neural Network Due to the runtime still taking roughly a day to process 30 epochs, we resorted to simplifying the design. Still concatenating, we now include the city and lane information with the input's 19 time-steps. Having two linear layers, the input dimension is 240 * 39(19-timesteps, 19 columns of lanes, 1 one-hot encoded city). We chose 19 columns for lanes because on average there weren't more than 60 * 19 lanes. The hidden dimension was 240 * 32 into the second linear layer that returned 240 * 30,which

will be reshaped to 60 * 30 * 4 for the next 30 time-steps of each car. The forward function takes the flattened down concatenated input and returns the predicted 30 time-steps

Some regularization/standardization techniques attempted were dropout. However, the model only proved to have longer runtime without any significant improvement to the rmse error. Using an ensemble method learning, we tried 5 variations of dropout [0.1,0.2,0.3,0.4,0.5] then ran the LSTM model and linear layer model for 30 epochs. Surprisingly no dropout proved to have the best effect on the kaggle's prediction, while the training error eventually plateaued to the same error for each dropout value.We had experimented with mean normalization, where we would would subtract each numerical matrix by the mean of each column in the matrix. However, that proved to skew the data if it isn't the datasets mean, not the scene's mean. While that was a normalization technique,We wanted to discuss what we tried.Another normalization technique was finding the global min and max of each column and relying on a dataset-based min-max scaler. Since we have Conv1d layer, we adjusted the max-pooling technique using [2,3,4,5] but yet again we didn't rely on max-pooling because the training error increased to a 5000 mse error under the LSTM model. There was no conv1d layer in the fully connected neural network model. The biase-variance trade-off had a low training error but high validation error so that implies we have high variance so we should be using regularization and early stopping. Therefore, we tried saving the model every 5 epochs and found the best performing model was at 30 epochs.

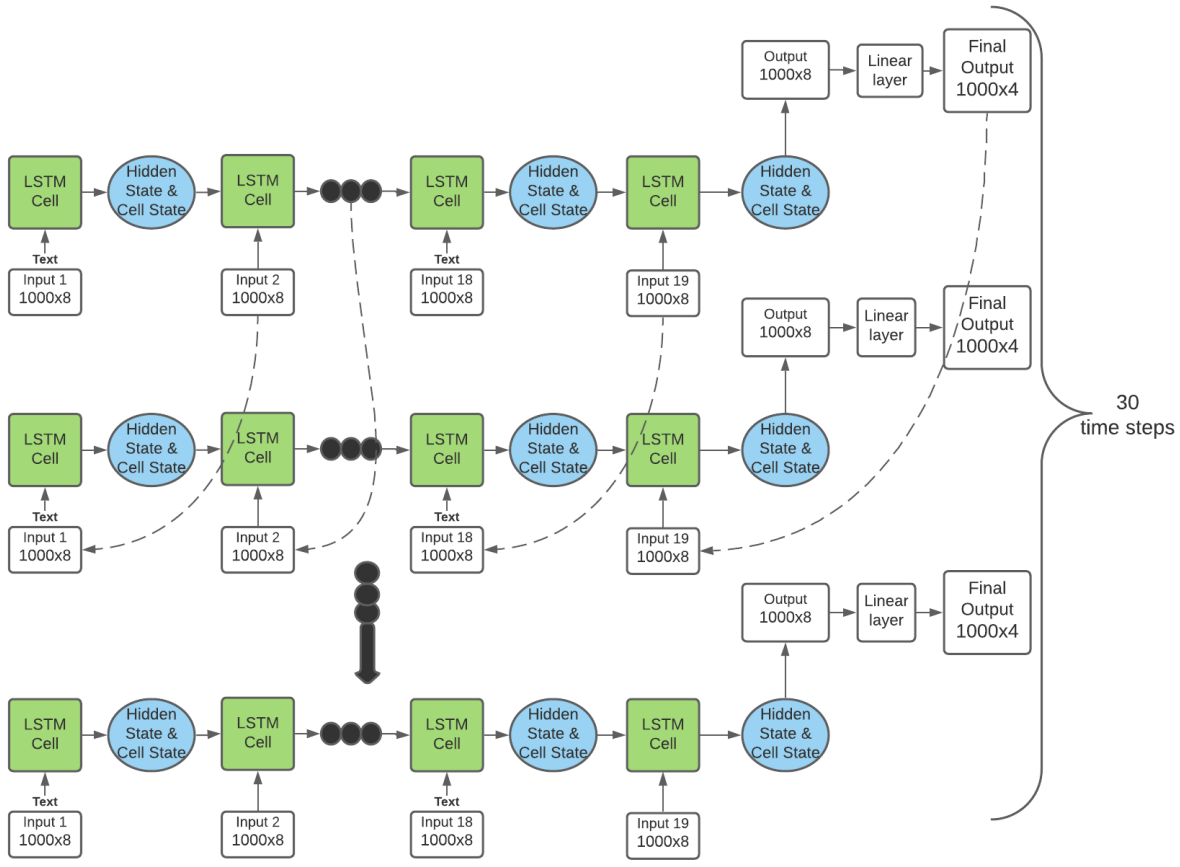

Figure 11: Sliding window approach with LSTM cells (each step should have 2 LSTM layers)

4 Experiment Design

4.1 Problem A

We used Datahub to run the model and we used the GPU version to use cuda so that the model would run faster.

We trained over the entire training data set and then after training we would run the model on the validation data set provided for us to generate the .csv file that we would then submit on Kaggle to find our test error.

We used Adam as our optimizer. We adjusted the learning rate as we went, at the start we used a larger learning rate just to see if the model was even learning, but after that we use a smaller and smaller learning rate to lower our error more reliably and we settled on $1e-5$ as our final learning rate that we used. We did not use learning rate decay or momentum, but we did use a weight decay of $1e-5$. We tried to use a learning rate decay and momentum, but we found more success without using them.

Our model predicts the positions and velocities of all agents for the next 30 time steps all at once. A previous model employed a sliding window where the model would predict the positions and velocities one time step at a time and then feeding the new guess into the next input, but we found that the model ran incredibly slowly and also one really bad guess would mess up future guesses. Our model is able to predict all 30 time steps at once because we specify the output dimension to be 4*60*30 and we use a fully connected linear model.

We trained for 30 epochs because more epochs did not seem to make the model much better. Our batch size was 1 so we just looked at one scene at a time. It takes us x amount of time to run one epoch.

We chose the design for a multitude of reasons, initially it was because these parameters were the easiest to implement. We had a lot of dimension errors at the start while we learned how to use Pytorch and that is why we ended up using a batch size of 1 for example because that is what we started with. For the optimizer and its parameters, we tried many different values and ultimately found the best results with the ones we chose. We increased the number of epochs we ran for as we built more efficient models. When we first built our sliding window model that guessed one time step at a time, that took so long to run that we only trained for 5 epochs and that would still take over a day to run. As our models have gotten faster, we have been able to run for more epochs. Using Datahub and the GPU provided for us a was a game changer because we could leave models running for hours without interfering with our own devices and we did not need to keep laptops running to keep the model running.

5 Experiment Result

5.1 Problem A

The results with different design is shown in the table below. This table also includes the estimated training time per epoch and the number of parameters for each model.

Table 1: Experiment design comparisons

Approach	Epochs	Final test RMSE	Estimated time per epoch (min)	Number of parameters parameters
Encoder-decoder attention model	10	734.37	6	3636120
Sliding window with LSTM cells	10	1091.98	20	186552240
Fully connected neural network	30	11.04006	18	127195680

The current result shows that the fully connected model has the best prediction. We used the most number of epochs here because it is still improving after training for 10 epochs, but the other two models doesn't have significant improvement afterwards. This result shows that simple fully connected neural network can be useful and accurate when implemented correctly with good input

data and hyperparameters. It also shows that our approach with the RNN needs to be further modified to obtain a better result.

To speed up the training for these models, we tried different values of parameters like the number of hidden dimensions and number of layers. We also tried reducing the dimension of inputs and thus lowering the need of neurons. In the RNN model with multiple layers, we also added Dropout layer to speed up training. However, the most effective speedup for us is when we increase the batch size and utilize multiple number of workers in the data loader. Setting num_of_workers to 8 significantly lowers the training time.

5.2 Problem B

The model with the best performance is the fully connected neural network. For this model, the visualization of our training loss value over training steps is shown below. We also added the visualization with the loss in the last 20 epochs considering the training loss drastically decreases over the first few batches and it's hard to see the change in the later epochs.

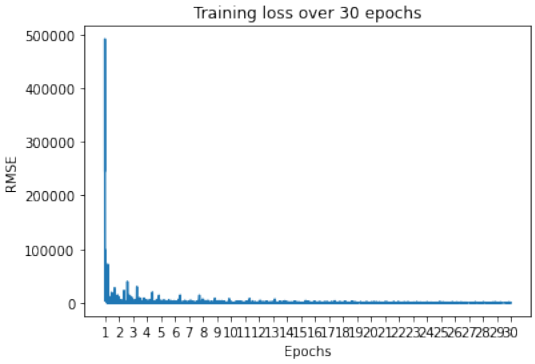

Figure 12: Training loss (RMSE) over steps

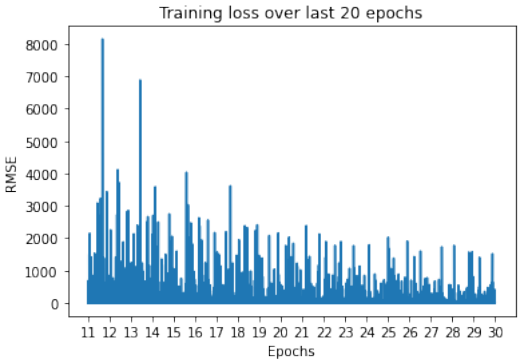

Figure 13: Training loss (RMSE) over steps for the last 20 epoches

We can see that the training loss decreases exponentially in the first few epochs from 500000 to 8000, then stabilizes and fluctuates around 1000 after that (for a batch size of 10).

Four training samples with both ground truth values and predictions are shown below (Red circles are input, red X-marks are ground truth output, and blue X-marks are predictions).

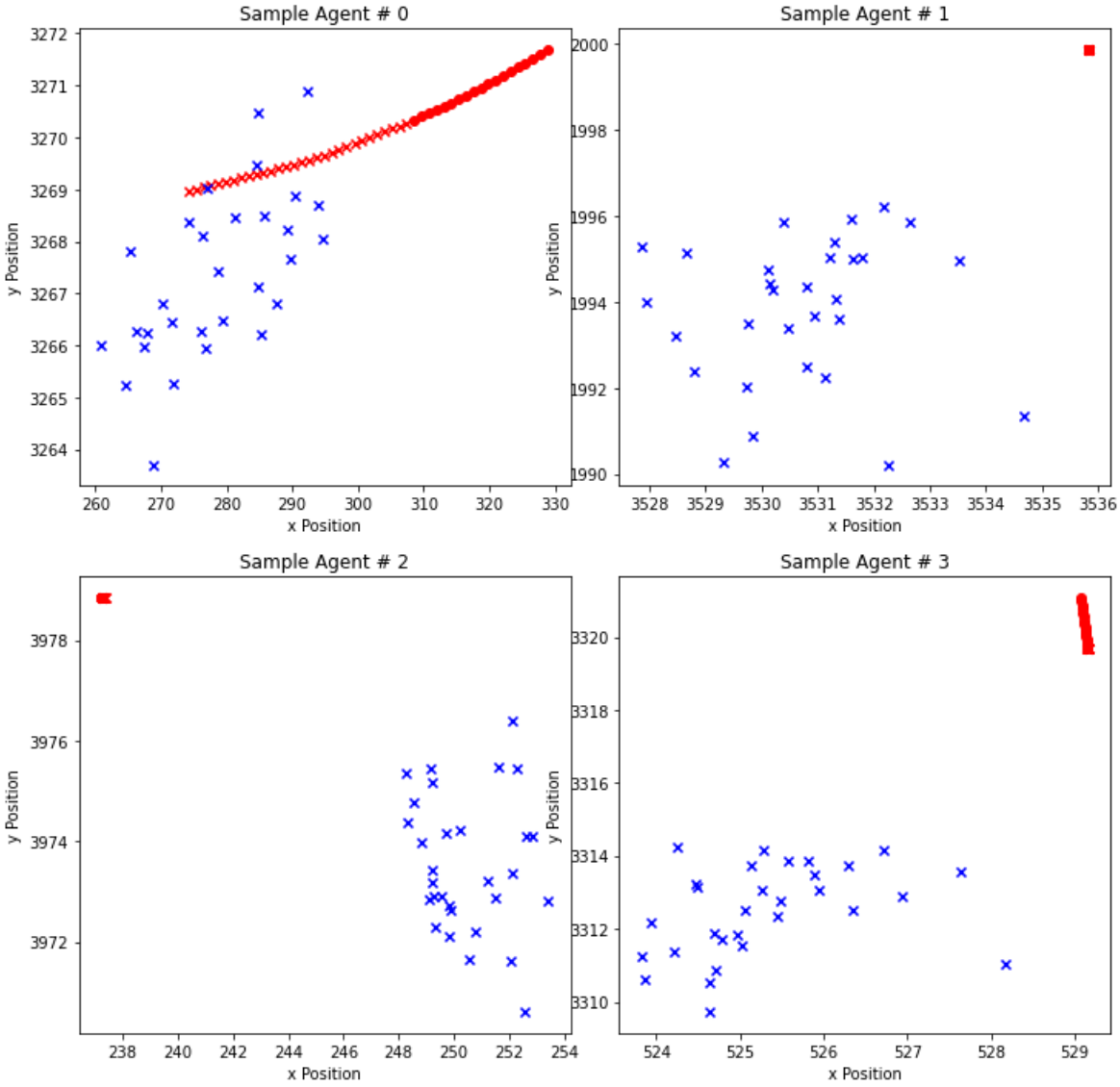

Figure 14: Four training samples with ground truth and predictions

These four sample shows that our prediction is relatively far from the ground truth output positions, usually deviates from the input for around 5 to 10 units, while the deviation is consistent in all output time stamps.

Our current ranking on the leaderboard is 22 with test RMSE of 2.68. By using the exponential moving average for the x and y velocities, we found the next 30 timesteps by relying on the equation velocity * change of timestep + previous position. [1]

[1]This best result was in fact obtained from physics simulation by computing the exponential moving average of the positions and velocities from previous timesteps to yield the predictions. It is not included because it isn't using a neural network model.

6 Discussion and Future Work

6.1 Problem A

The most effective feature engineering strategy is to incorporate lane information with some sort of attention mechanism so that more importance is placed on the agents and environmental factors closest to the target agent. Our current model regards all vehicles and lanes with equal importance and that makes the model take longer to run and also there is more noise/irrelevant data.

We found that using a fully connected neural network was the most helpful in improving our performance. Data visualization and understanding the input data was really important at the start of the project to help us understand how to process the data. Tuning hyperparameters was also helpful in improving our score but the most important change was the implementation of the fully connected model.

We had many bottlenecks for this project, at the beginning it was definitely our lack of comfort/experience with Pytorch and building neural networks as we ran into a plethora of runtime issues that bogged us down. Next, when we started building models that were learning to some extent, our models were super inefficient and it took us many hours if not days to train and that made it really difficult to try new changes or new ideas. Finally, towards the end of the project, Datahub was being really slow and crashing/having issues for us which also made it difficult for us to try new ideas or new designs.

Our advice for a beginner would be to start with a simple model that is easier to implement and build and hopefully will run into fewer issues. From there, they can try to add more feature engineering or build more complex models with a simple model to fall back upon. We made a mistake of trying to implement a more complex encoder/decoder with GRU and a LSTM with sliding window at the start where a simple connected model could have been much easier.

If we had more resources, we would try to run the model and train it with a variety of different hyperparameters over more epochs. If we just had more time, we would also try to implement more complicated models such as encoder/decoder with attention mechanism which is something we wanted to try or incorporate lane information in a more impactful and meaningful way instead of just throwing in all the lane info like we did.

References

[1] Gao, Shuai, et al. “Short-Term Runoff Prediction with GRU and LSTM Networks without Requiring Time Step Optimization during Sample Generation.” Journal of Hydrology, vol. 589, no. 125188, 2020. SciencDirect, www.sciencedirect.com/science/article/abs/pii/S002216942030648X.

[2] Wenjie Lu, Jiazheng Li, Yifan Li, Aijun Sun, Jingyang Wang, "A CNN-LSTM-Based Model to Forecast Stock Prices", Complexity, vol. 2020, Article ID 6622927, 10 pages, 2020. https://doi.org/10.1155/2020/6622927

[3] Maitra, Sarit. “Technical Indicators and GRU/LSTM to Predict Stock Price: Time Series Analysis with Python Code.” Medium, Towards Data Science, 12 Apr. 2021, towardsdatascience.com/forecasting-with-technical-indicators-and-gru-lstm-rnn-multivariate-time-series-a3244dcbc38b.