CSE 151B Project Final Report

Aaron Lee
Department of Computer Science
University of California: San Diego
La Jolla, CA 92092
`acl049@ucsd.edu`

Christopher Ly
Department of Data Science
University of California: San Diego
La Jolla, CA 92092
`chl818@ucsd.edu`

GitHub Link

1 Task Description and Background

1.1 Task Description

Autonomous vehicles (AV) are an active research area which can lead to safer navigation. To achieve safe and collision free traffic, there are still many issues related to navigation and trajectory prediction that need to be solved first. Reliable trajectory prediction is necessary in order to make safe decisions when surrounded with various road agents such as cars, cyclists, and pedestrians. Today, we see many auto accidents due to DUI or being distracted while at the wheel. By having an autonomous vehicle that is able to replace most if not all of the driver's functionality, it will reduce the number of collisions and traffic accidents annually. Tesla is an example of pioneering the autonomous vehicle industry.

1.2 Current Methods

We examined several papers from this GitHub repo as well as our own Google searching. We found that many papers focused on both environmental data and actual vehicle data. For example, we looked at the Uber Research paper which focused on a concept called LaneGCN. They developed a network that took in actors as well as the environmental data and fused the two pieces to get output. Another paper we looked at was the SMART paper. It used a convolutional encoder and a ConvLSTM decoder which then fed into a state-pooled ConvLSTM. It looks like they also output $x, v, heading, acceleration, steering$.

1.3 Prediction Task

Given various AV motion data, our goal is to predict the positions of a tracked object 3 seconds into the future. Looking at the inputs, we have the following equation describing the training set:

$$S = \{x_i, y_i\}_{i=1}^{N}$$
$$\texttt{where } x_i = \{x_{ij}\}_{j=1}^{19}, x_{ij} \in \mathbb{R}^{60\times4}$$
$$\texttt{and } y_i = \{y_{ij}\}_{j=1}^{30}, y_{ij} \in \mathbb{R}^{60\times4}$$

The dataset space contained 60 agents each with 4 features (position and velocity on the x and y axis).

On an abstract level, trajectory prediction has many applications beyond autonomous vehicles. Aside from vehicles, there are other technologies being developed for close proximity to humans such as social robots. These robots are designed to interact and communicate with humans and other agents proximally close and would also benefit from the functionality of our model.

2 Exploratory Data Analysis

2.1 Dataset Analysis

Table 1: Dataset size

Dataset	Num. Samples
Training	164754
Validation	41188
Test	3200

The dataset is source from this Kaggle competition. The sizes of the datasets used are shown in Table 1. The original training set of 205942 scenes was split up using an 80/20 split for the training and validation sets for the models. The raw data contains several keys in a dictionary for each sample. We only examined the positions and velocities for each sample. $[60 \times 19 \times 4]$ is the shape of the input positions and velocities for one sample. This means that in one scene, there are 60 agents, 19 timesteps, and 4 data points (x,y for position and velocity). The output shape similarly has 60 agents and 4 data points but covers 30 predicted timesteps instead of the 19 timesteps in the input rusulting in a shape of $[60 \times 30 \times 4]$.

2.2 Agent Feature Distributions

The distributions of the target agents' position and velocity can be found in the Appendix. Figures 5 and 6 show the input positions and figures 7 and 8 show the output positions. The range of the distributions are sensible when observing the plots of the cities. Given the structure of the cites and the orientation of the lanes, the directions of velocity shown in Figures 11 and 12 are components of the overall input velocity of the target agent shown in Figure 9.

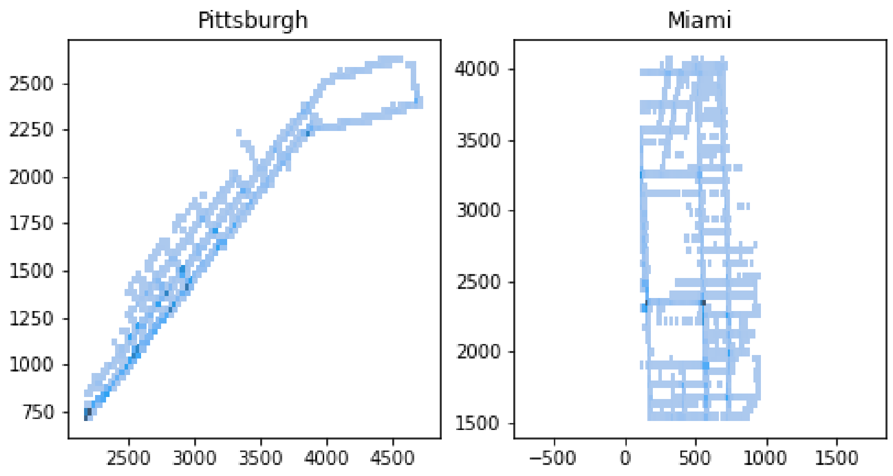

Figure 1: Distribution of the X,Y Position for Target Agent

Structure of the city and the scenes are important to trajectory prediction. Understanding the lane structure of the scene helps alleviate the possible actions taken by the target agents and the others. The sample lane structures shown in Figure 13 provide insight into the flow of traffic for cars in the scene. This structure could prove useful in improving the performance of trajectory prediction.

Another aspect of trajectory prediction is multi-object projection. Most of the scenes observed tend to have fewer cars with positive skew (Figure 14). While the focus of this paper is the trajectory prediction of only one target agent, this is still an important aspect to consider as more agents in the scene could impact the actions of the target agent. [1]

2.3 Feature Engineering

We also looked at two additional features during the preprocessing of the data. One metric is the displacement between the current position and previous position. The distribution of displacement shown in Figure 15 resembles that of the velocity. This makes sense as the agents would have new positions based on what direction they are heading. Another metric is the angle between the velocities of the current position and previous one as shown in Figure 16. As we plotted various predictions to ground truths in Figure 18, we found that our model had trouble adapting to curves and getting accurate positions for each prediction. We thought these two metrics included in the training would help alleviate some of the issues we were seeing. From the distribution of angles, we observe that a majority of agents in the scenes rarely deviate from a straight path.

We used the batch norm to normalize our data in the CNN. The CNN would process spatial and temporal features and normalize it. This normalized data was then used to initialize our hidden states for the LSTM. We tried another method to normalize all data to the target agent's position at t_{19}. We were hoping that our model would become more accurate as everything is relative to that object's position so our CNN doesn't have to focus on memorizing positions in different cities. However, we faced some difficulties in implementing this as our losses were astronomically high. We tried other methods of normalizing our data but weren't able to do much given the amount of time left.

We looked at various methods of incorporating the lane nodes into our model but our attempts were not fruitful. An obstacle to this was the lack of uniformity in the lane node data as each scene had a variable amount of lanes making it difficult to align with the other input features. An attempt to address this was calculating the n nearest lane nodes to the target at each position in order to capture a running view of the target agent's surrounding. Another way was to create a large sparse matrix with a shape based off the largest amount of lanes found in the data but this led to large performance drops due to the conflict of sparseness and our model architecture. In the end, we chose to exclude the lane nodes from consideration by our models.

Inspired by the attempt to capture a moving window around the target agent with the lane nodes, we narrowed the number of agents visible per scene. Taking a look again at Figure 14, the average number of objects in the scene lies around 8. Based on this information, we located the 6 nearest objects to the target agent to capture this proximity window view. Using the 6 closest allows for a social aspect to the models and capturing a dense matrix of input as the scene data is padded with 0 for non-agents. This overall accomplished a speedup in training and a more accurate view into the target agent's surroundings.

3 Deep Learning Model

3.1 Deep Learning Pipeline

We used position, velocity, displacement, and angle between as features to the CNN. Our input shape to the CNN is

$$\begin{aligned}&\mathbb{R}^{B\times C\times T\times N}\\ &\texttt{where } B=\texttt{batch size, } C=\texttt{features,}\\ &T=\texttt{timesteps, } N=\texttt{agents}\end{aligned}$$

The CNN transforms C into a number of hidden units and this output is reshaped for the LSTM.

$$\begin{aligned}&\mathbb{R}^{B*N\times T\times C}\\ &\texttt{where } B=\texttt{batch size, } C=\texttt{hidden units,}\\ &T=\texttt{timesteps, } N=\texttt{agents}\end{aligned}$$

However, we had our LSTM predict everything except position as output. We then used the predicted displacement and added it to previous position to get the current position. We did not do any feature engineering to the outputs.

$$\begin{aligned}&\mathbb{R}^{B*N\times T\times C}\\ &\texttt{where } B=\texttt{batch size, } C=\texttt{output ft,}\\ &T=\texttt{timesteps, } N=\texttt{agents}\end{aligned}$$

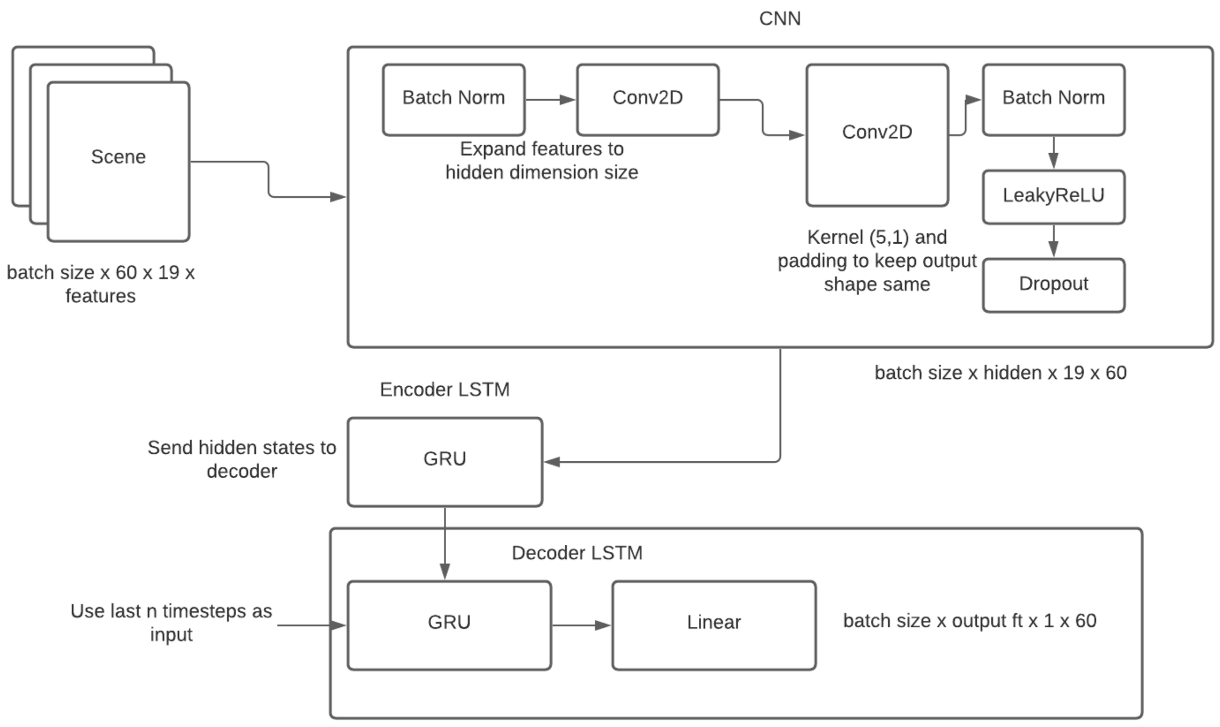
CNN
Scene
batch size x 60 x 19 x features
Batch Norm
Conv2D
Expand features to hidden dimension size
Conv2D
Kernel (5,1) and padding to keep output shape same
Batch Norm
LeakyReLU
Dropout
batch size x hidden x 19 x 60
Encoder LSTM
Send hidden states to decoder
GRU
Decoder LSTM
Use last n timesteps as input
GRU
Linear
batch size x output ft x 1 x 60

Figure 2: CNN LSTM Baseline Model

We compared the resulting position with our target position in our RMSE loss function. We also considered using SmoothL1Loss because it would penalize outliers less. We thought that for points that are within a certain bound of the target position, we would use RMSE but use RMAE for points outside. When plotting various graphs of the ground truths vs predictions, some scenes had scattered ground truths making it hard to find a nice model to fit.

After designing new features, we examined the relationship between using displacement and using velocity in predicting the next position. We came up with this model after reading a research paper [3] and tweaking our input features as well as model design. Since the velocity had noise and didn't seem very reliable compared to actual position, we tried to predict based on displacement which rather would give us the actual change in position. Using displacement ended up getting more accurate predictions than using velocity to predict future positions. This is likely due to velocity also capturing deceleration which would throw off the model as this was not built in the model for consideration.

3.2 Model Variations

We initially started with three different models: LSTM, CNN, ConvLSTM. After finding that the best model was the CNN, we wanted to improve on that and add a LSTM at the end to retain sequences. We tried variations of this new model to improve our accuracy. We established a baseline model of our architecture which can be seen in Figure 2 and it was mostly inspired from GRIP++ [3] with some modifications. All model variations are in this Table 2. The baseline model was run on DataHub, the local models were run on RTX 2080 Ti, and the colab models were run on Google Colab.

While most of the variations had the same input shape, we reduced the number of input agents for LocalV3 because of the fact that we use a CNN. As mentioned in Section 2.3, CNNs don't handle sparse data very well and there are some dummy agents with padded 0s in the dataset. By changing the input to be the nearest 6 agents to the target agent and normalizing position to the t_{19} position of each agent, we were able to achieve a strong model. We also inserted another linear layer in the decoder as well.

Table 2: Model Variations

Name	CNN Units	LSTM Scale	Prev Ts.	Input Ft.	Predict Ft.	Num. Enc/Dec
Baseline	64	30	5	4	Velocity	1
LocalV1	96	30	3	7	Displacement	2
LocalV2	128	30	7	7	Displacement	1
LocalV3	128	64	19	4	Velocity	1
ColabV1	128	30	5	6 (no angle)	Displacement	1
ColabV2	128	64	19	4	Velocity	1

We used dropout and tried to implement early stopping to minimize the impact of overfitting. We used the dropout in the CNN between each kind of block so it would generally be able to find a smooth relationship between the temporal features.

4 Experiment Design

We used three different resources: DataHub, Google Colab, and a RTX 2080 Ti. We did a 80/20 train/validation split and shuffled the indices. We used the Adam optimizer with a learning rate of 0.001. We didn't tune the learning rate because we wanted to find a model that best fit our features first. We experimented slightly with Adam optimizer with weight decay in LocalV3 and ColabV2. For the multistep prediction, we fed a sequence of previous n timesteps to the LSTM and used the last output as our predicted number. We added this number to the previous position to get the current position. In order to fit this model on the resources, we used a batch size of 64 and tried to train our models for 50 epochs. We aimed for our models to take about 30 minutes per epoch on the original input size. The new input size only took about 15 minutes per epoch and freed up memory allowing for more CNN layers. The limiting factor is memory on the GPUs. We would have liked to try slightly wider models but ended up hitting the memory limit on these resources. On Colab, we have a higher memory limit but are limited by compute time as we are restricted to making sure our models train within 24 hours.

5 Experiment Results

The CNN LSTM performed the best out of our various model architectures. Thus, we tried to do hyperparameter tuning to get the best performance possible. To start off, we set a baseline version of the CNN LSTM architecture. From Figure 17, the model continuously improved in the RMSE training loss as it trained for more epochs. Initially starting poorly, the resulting loss on the training set is a large improvement over the first epoch. However, it is important to address concerns of overfitting since the RMSE loss will naturally decrease with more epochs. To see how the CNN LSTM model was performing, the model's predictions were compared against the ground truths in Figure 18. While the model achieved a low training RMSE loss, the model's predictions did not match up well with the true trajectory of the target agent particularly with curves. Perhaps this was due to artificially deflated loss numbers because of the dummy agents so we thought our model was good when seeing lower loss numbers whereas in reality, it was not performing to that standard. Thus, we realized the need to prune our input tensor.

Table 3: Network Results

Name	Num. Parameters	Num. Epochs	Train RMSE	Test RMSE
Baseline	149826	50	0.77	2.87
LocalV1	735848	50	0.702	2.95
LocalV2	715605	30	0.64	2.42
LocalV3	611074	20	1.63	2.12
ColabV1	715605	50	0.67	2.90
ColabV2	611074	20	1.63	2.12

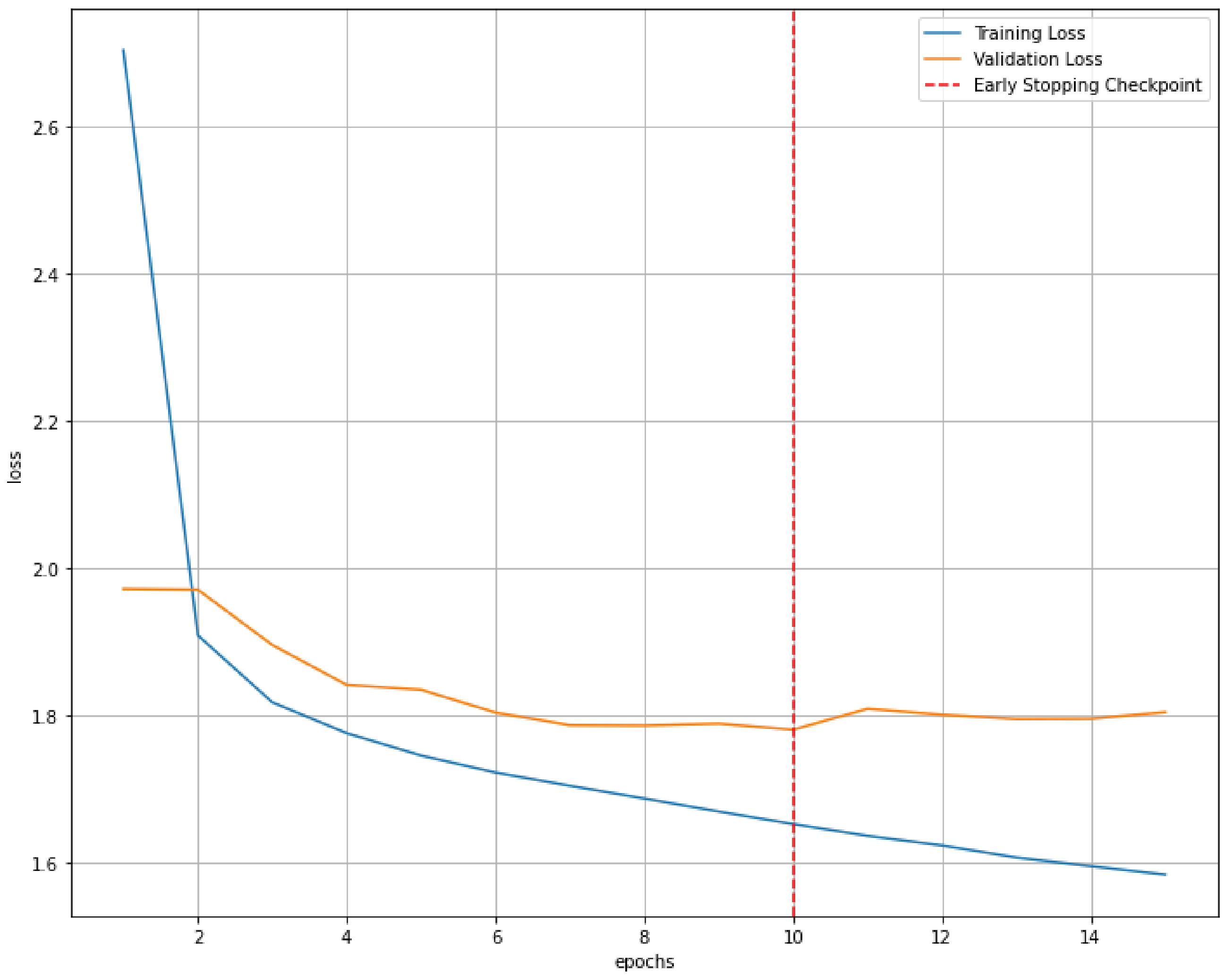

Figure 3: Training vs Validation Loss of Best model

The results of multiple iterations and experimentation of hyperparameters are summarized in Table 3. Our best model similarly continuously improved in the RSME training loss with more epochs iterations as seen in Figure 3. By comparing the ground truth to the predictions of this model in Figure 4, we can visualize qualitatively that the model estimates the output positions better than the baseline model. Narrowing the model's attention to the 6 closest agents to the target agent proved successful in improving both training speed and performance. At the time of writing this paper, our best model achieved a loss of 2.12 placing our group 8th in the Kaggle competition. So despite the discrepancy between the model's predictions and the true trajectory, there are significant areas for improvements as the predictions are still a ways off particularly on curves.

6 Discussion & Future Work

We think that data preprocessing is the most important step because the raw data could be missing at certain places. Furthermore, designing these features reveals a relationship that you can use in your model to help get better results.

From the midterm report, we realized that the high variance in the positions graphs we used was not a good way to predict our output. It was helpful to have various metric graphs to find out what looks

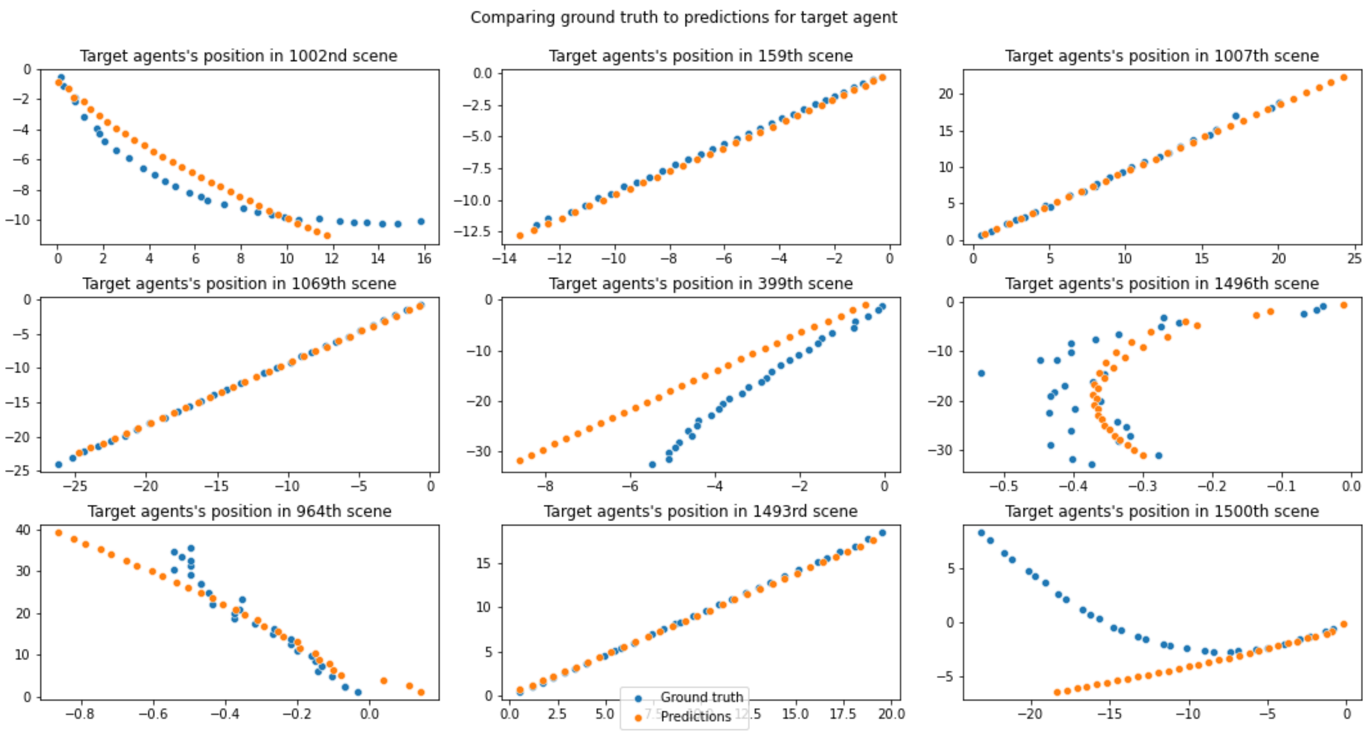

Figure 4: Validation Set Predictions vs Ground Truths for Best model

smooth and what is not. We also tuned our model parameters to try to find out what works best. We found that including more timesteps helped determine a better position and velocity.

A large bottleneck we faced is simply the training time. We trained our initial models for up to 50 epochs but each epoch takes roughly 30 minutes on average. The simpler models have performed even better and didn't take such long compute times. But in addition to time constraints, due to limited memory capacity on the GPUs, we could not construct deeper models utilizing more hidden layers than currently being used. Thankfully, this was somewhat alleviated with our simple social data preprocessing.

We would suggest that you always visualize your data and try to create new features if possible. Exploring and understanding the data in its initial state is very important, but taking the next step of building new and meaningful features off of that was able to improve the performance of our model. We also suggest to start with simpler models and work your way up.

Another idea we looked at is the lane node data. In our earlier visualizations of the lane node data, we can see it looks like a direct graph. If we are able to improve upon the three networks we designed above that focus on the object spatial data and then add a network handling the lane graphs, we think it will be an effective model for vehicle trajectory prediction. We did some initial work and found a research paper that utilizes lane graphs in conjunction with spatial data [4]. Since we have data on the other agents in the scene aside from the target agent, we could also consider additional features to capture the social context between agents [2].

Acknowledgments and Disclosure of Funding

We would like to acknowledge DataHub and Google Colab for providing compute resources to train our models. We also would like to thank Argoverse for providing the dataset for us to train on. This project was not funded in any shape, way, or form.

References

[1] Rohan Chandra, Tianrui Guan, Srujan Panuganti, Trisha Mittal, Uttaran Bhattacharya, Aniket Bera, and Dinesh Manocha. Forecasting trajectory and behavior of road-agents using spectral clustering in graph-lstms. *arXiv preprint arXiv:1912.01118*, 2019.

[2] Ming-Fang Chang, John Lambert, Patsorn Sangkloy, Jagjeet Singh, Slawomir Bak, Andrew Hartnett, De Wang, Peter Carr, Simon Lucey, Deva Ramanan, and James Hays. Argoverse: 3d tracking and forecasting with rich maps. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8740–8749, 2019.

[3] Xin Li, Xiaowen Ying, and Mooi Choo Chuah. Grip++: Enhanced graph-based interaction-aware trajectory prediction for autonomous driving, 2020.

[4] Ming Liang, Bin Yang, Rui Hu, Yun Chen, Renjie Liao, Song Feng, and Raquel Urtasun. Learning lane graph representations for motion forecasting. *CoRR*, abs/2007.13732, 2020.

A Appendix

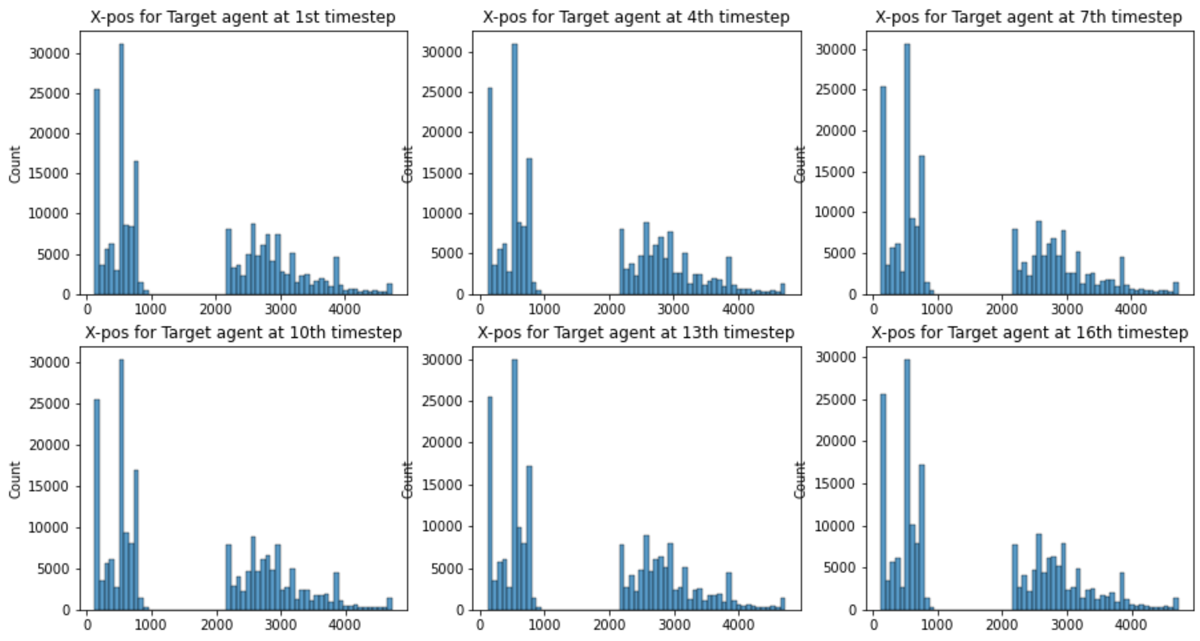

Figure 5: Distribution of the X positional coordinate for the target agent.

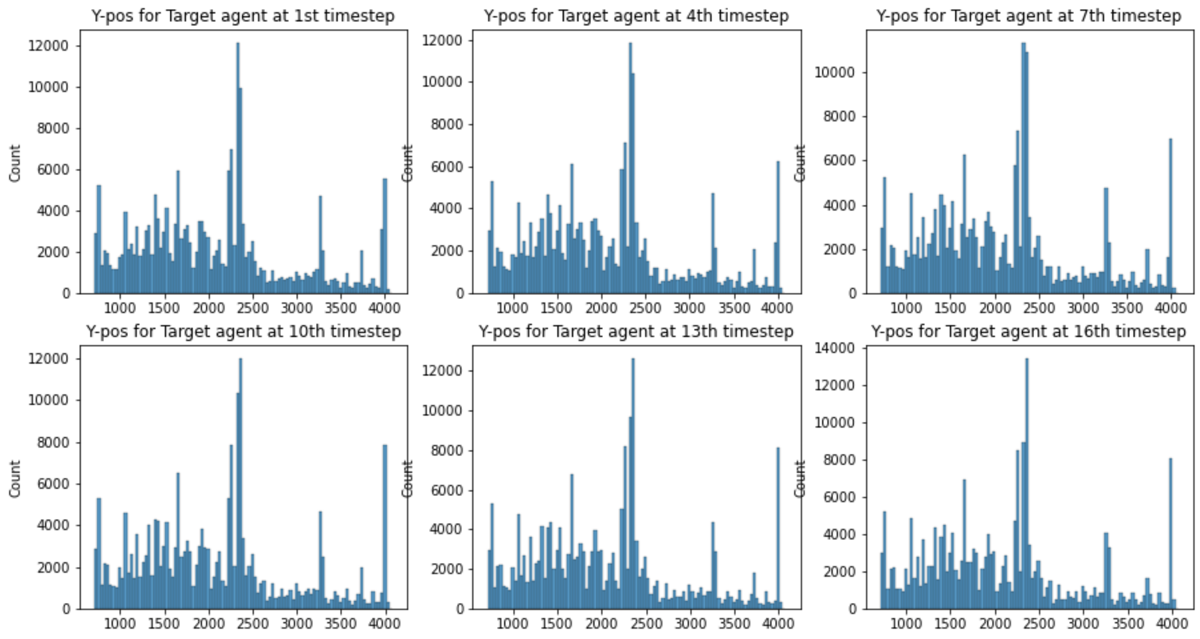

Figure 6: Distribution of the Y positional coordinate for the target agent.

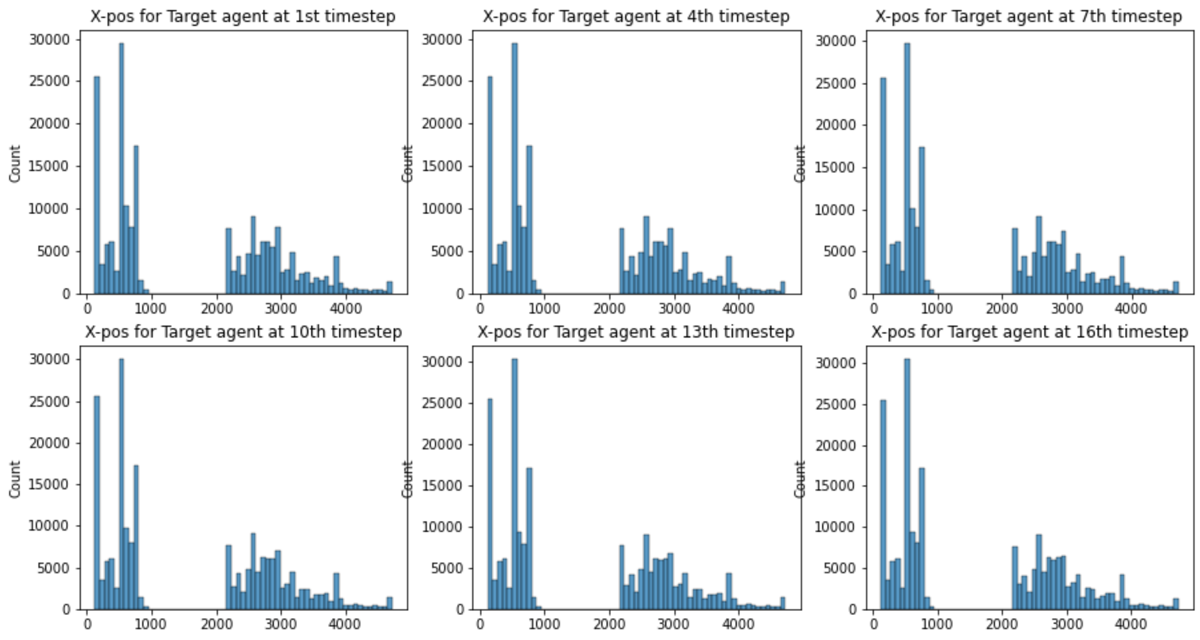

Figure 7: Distribution of the predicted X positional coordinate for the target agent.

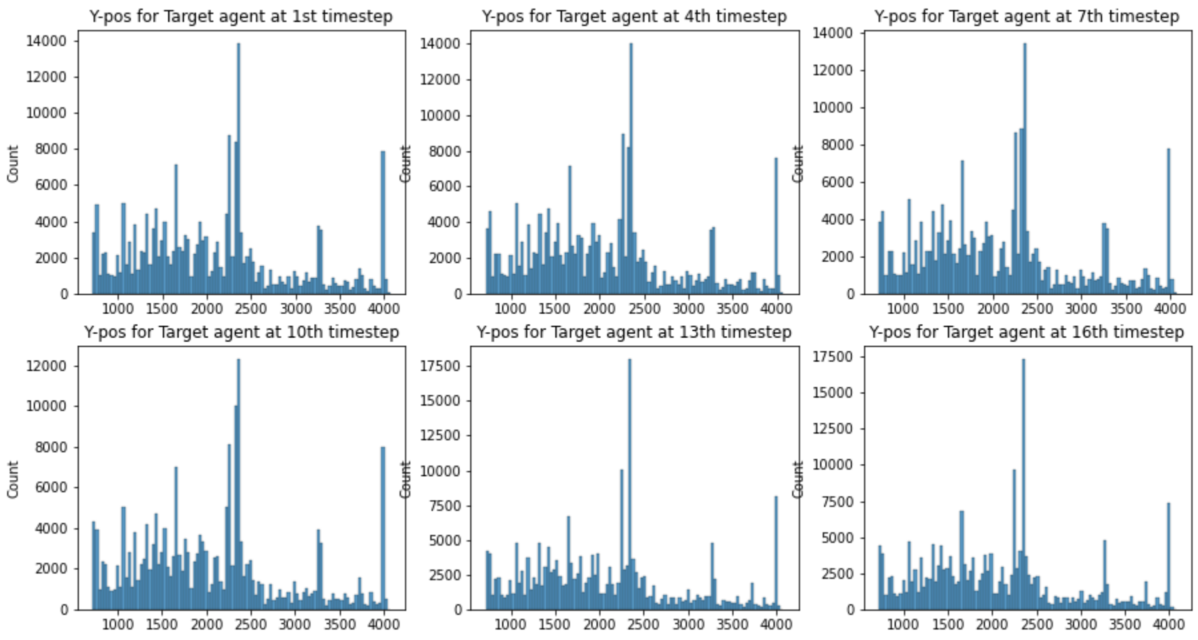

Figure 8: Distribution of the predicted Y positional coordinate for the target agent.

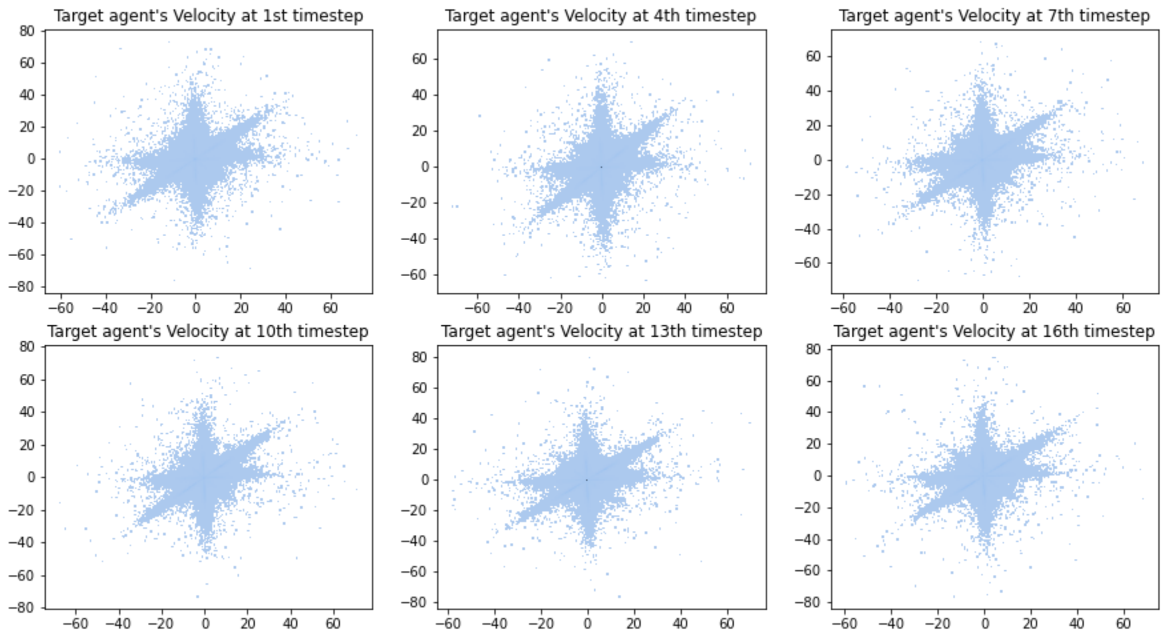

Figure 9: Distribution of the velocity for the target agent.

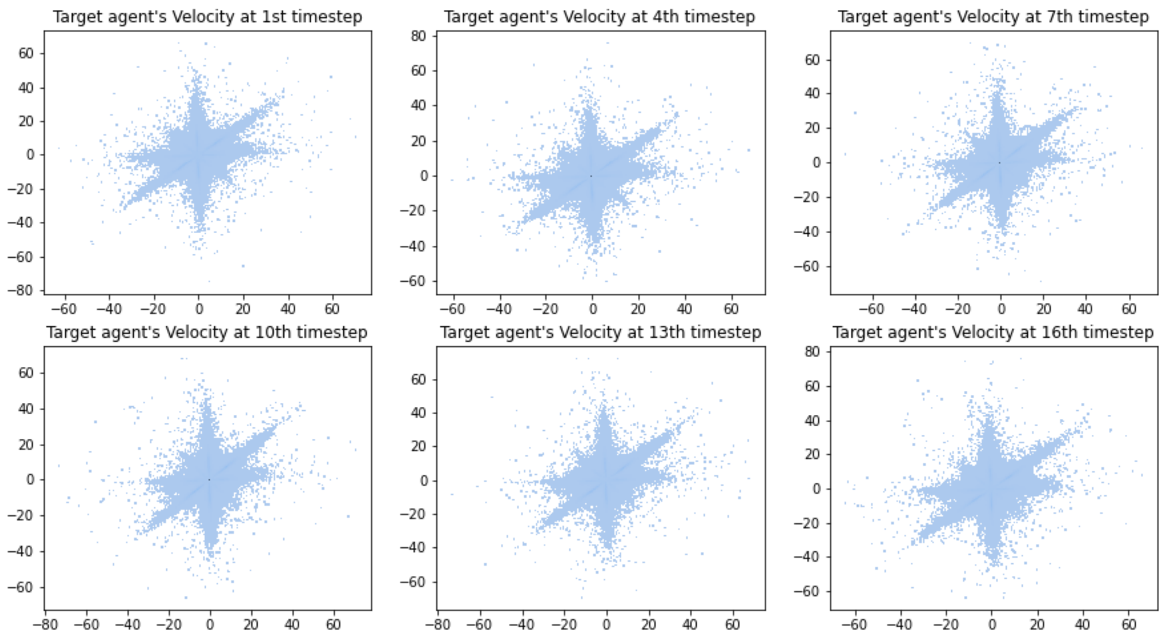

Figure 10: Distribution of the predicted velocity for the target agent.

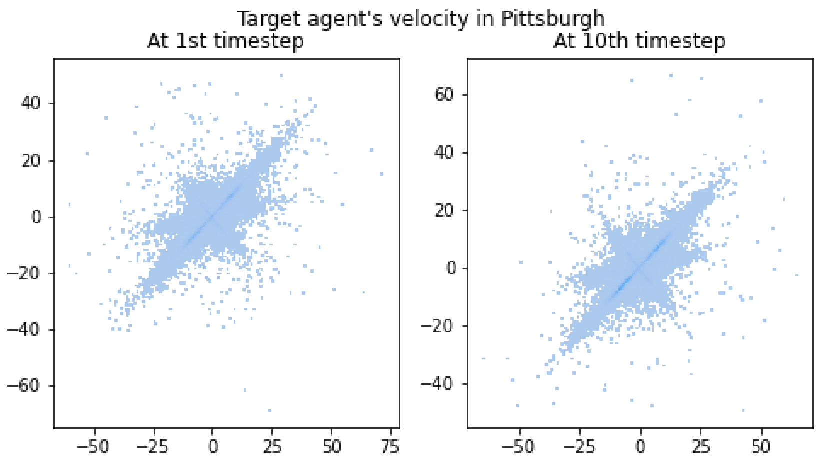

Figure 11: Distribution of the target agent’s velocity in Pittsburgh.

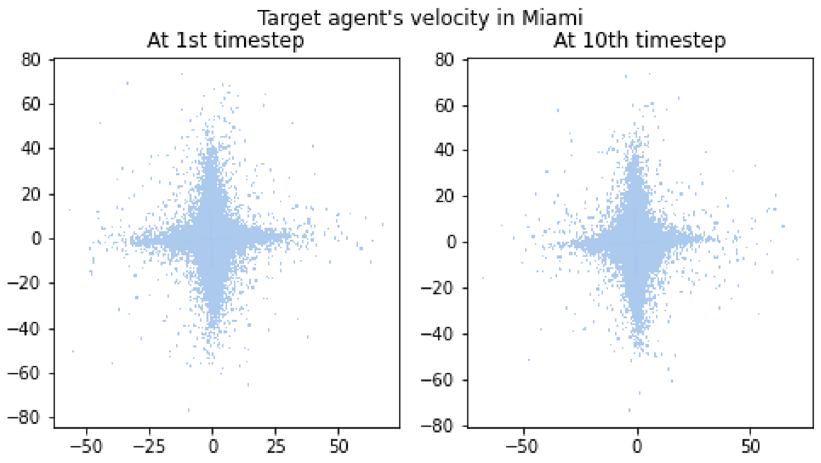

Figure 12: Distribution of the target agent's velocity in Miami.

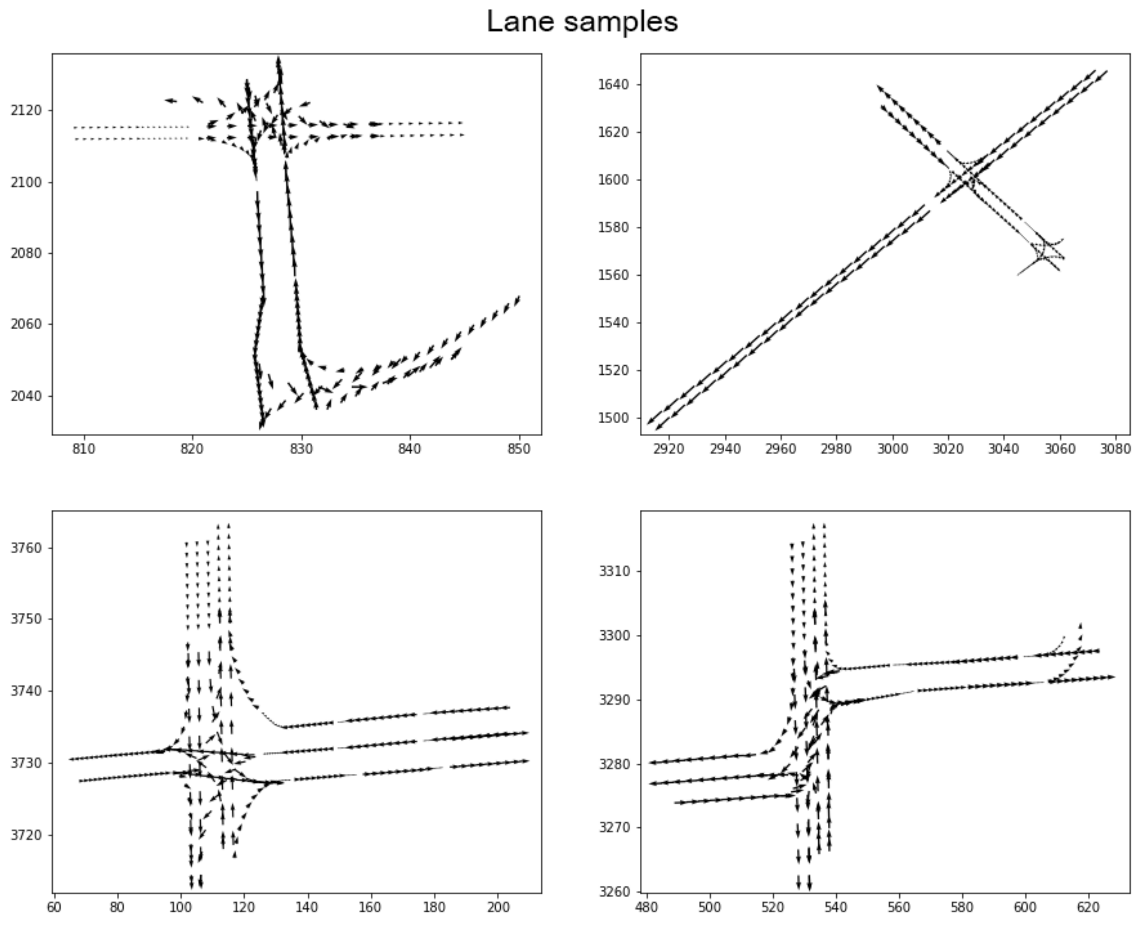

Figure 13: Sample of lanes with direction of traffic flow.

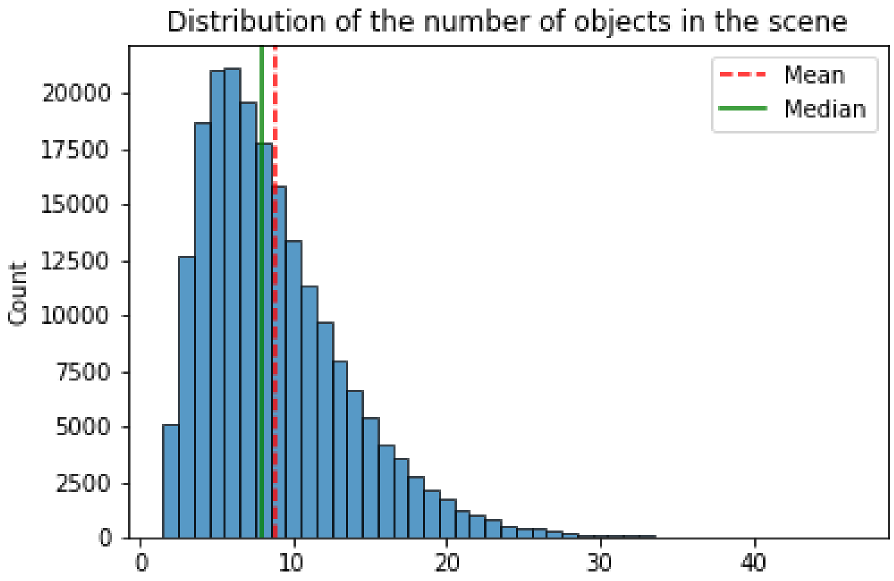

Figure 14: Distribution of the number of cars in the scene.

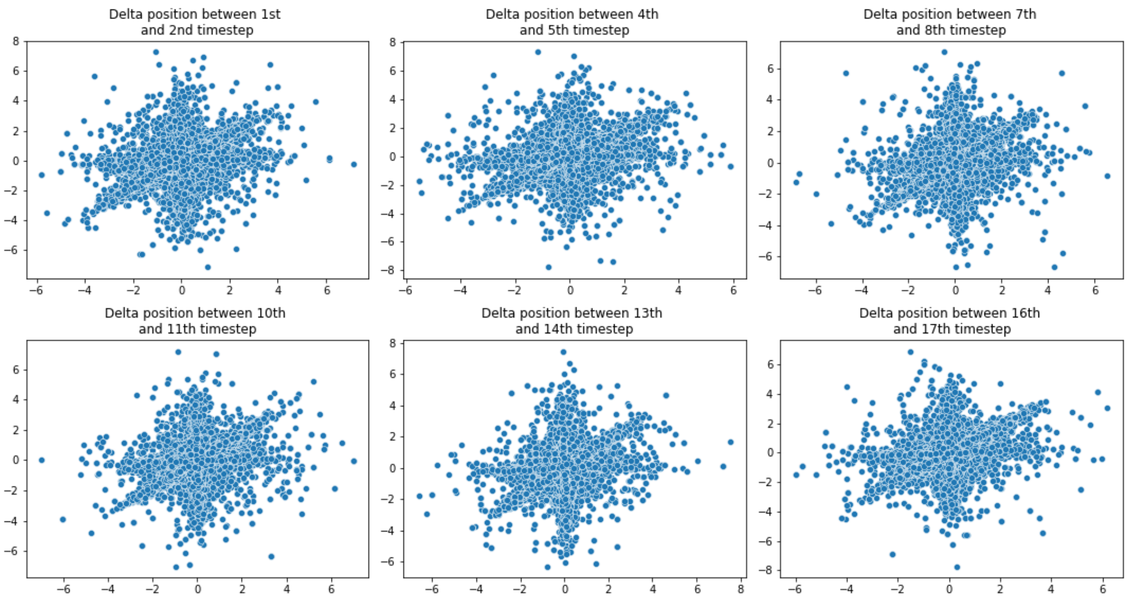

Figure 15: Displacement of agents.

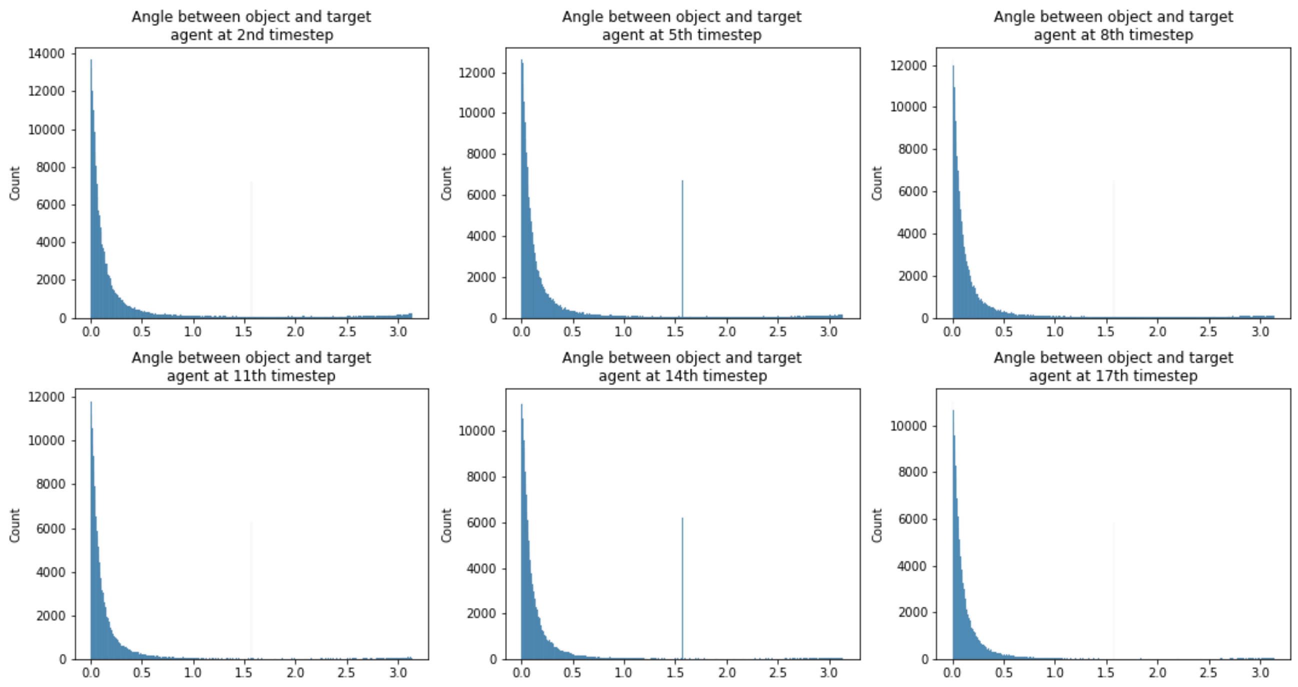

Figure 16: Angle of velocity between current timestep and next one.

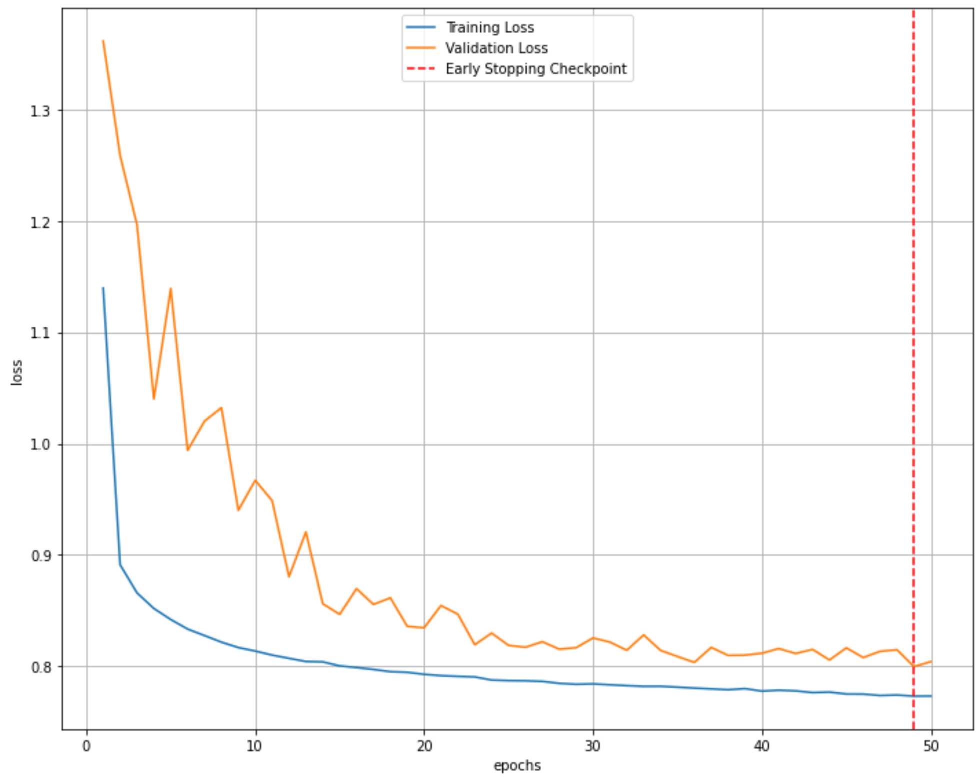

Figure 17: Training vs Validation Loss for Baseline model

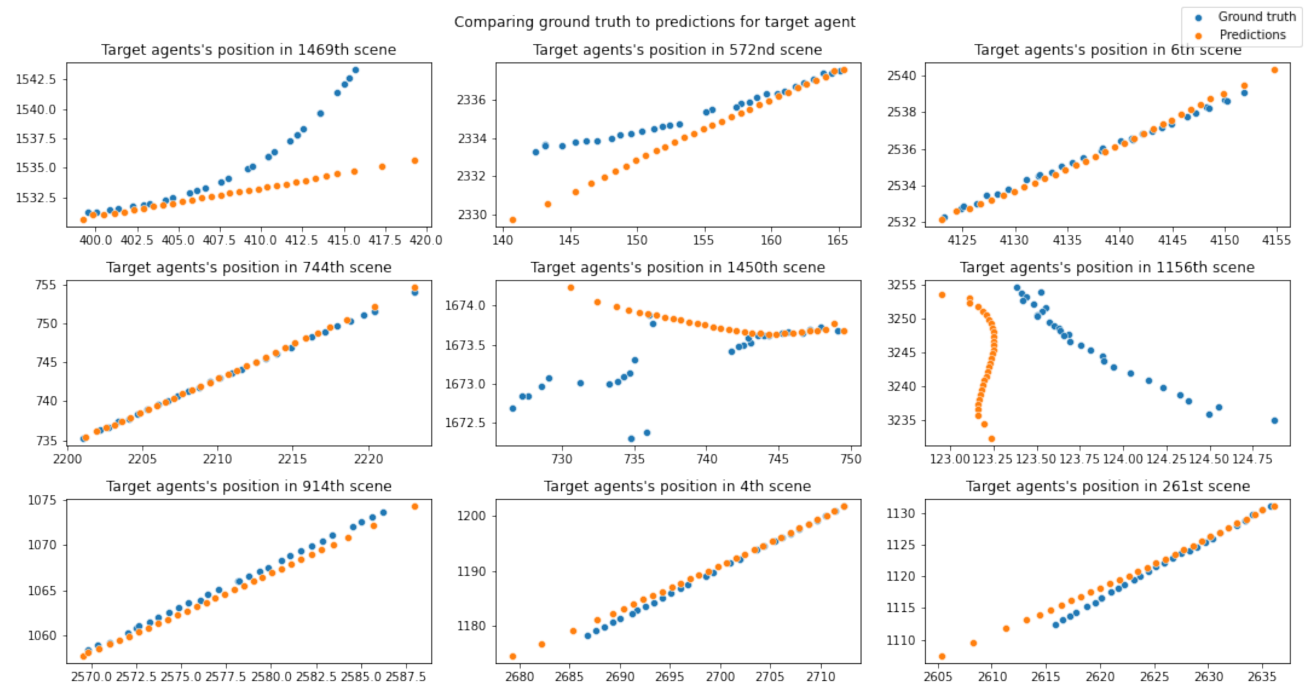

Figure 18: Validation Set Predictions vs Ground Truths for Baseline model