CSE 151B Project Final Report

Jianming Geng
jgeng@ucsd.edu

Yacun Wang
yaw006@ucsd.edu

Gao Mo
g1mo@ucsd.edu

Github Repository Link

https://github.com/colts661/cse151b_final_report

1 Task Description and Background

1.1 Task Description

Our task is to predict the vehicle's location within the next 6 seconds given the first 5 seconds of location for a specific vehicle in a certain city. This task enables machines to learn the pattern of driving direction according to different traffic conditions, seasons, etc. in different cities. It is essential to the Automated Vehicles (AV) industry so that automatic driving will be made easier when the vehicles adapt to different driving conditions learned from past observations. Companies such as Tesla have been tackling such task. By successfully solving this task, transportation accidents that are caused by human beings, such as impaired driving, distraction, etc, will be significantly decreased. Other causes such as not able to detect malfunction of vehicles will also be decreased.

1.2 Literature & Previous Work

Numerous studies have been done to improve the performance of Autonomous Vehicle system. For example, the approach mentioned in [6] applies two different models when it comes to vehicle control depends on the vehicles movement behavior. A kinematic model which takes in inputs such as linear path reference is employed when the vehicle is moving in a relatively slow speed, whereas a dynamic model which takes in information such as nonlinear control will be in charge of making predictions when the vehicle is moving in a relatively high speed. This is similar to one of our earliest experiments for our feature engineering process, during which we attempted to find the speed of the vehicle by dividing the distance difference with, and use it as one of the input features.

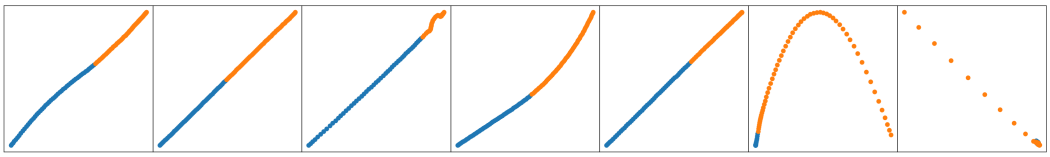
Another approach mentioned in [6] that was introduced to better the AV system's performance was Parallel Autonomy, a term that describes the situation where human beings and the autonomous system operate the vehicle at the same time. Human operates the vehicle until a certain situation where the complexity of the scenarios disallow us human beings to make the safest decision, then the AV systems takes charge and would grant an increased safety level. We found this idea intriguing and would like to corporate such engineering into our model if we were given enough data of human decision information during the vehicle's moving.

1.3 Data Input and Output

A single point is a pair (x, y), where x is the position of the vehicle with respect to the horizontal axis and y is the location of the vehicle with respect to the vertical axis. The input consists of 50 such points, which indicate the movements of a specific vehicle in the first 5 seconds. This further indicates that there are 10 points within each second of movements. Generally, the shape of a single input is 50 x 2. The output consists of 60 such points, which indicate the movements of the same vehicle in the next 6 seconds. And the shape of a single output is 60 x 2. Therefore, we can thus

Preprint. Under review.

Figure 1: One sample batch trajectories from Palo Alto



formulate our supervised regression task to be learning a function

$$f : X \to y \quad \text{where} \quad X \in \mathbb{R}^{50\times 2}\ ,\ y \in \mathbb{R}^{60\times 2}$$

where the model class should be a neural network and the loss function is MSE $= (f(X) - y)^2$

2 Exploratory Data Analysis

2.1 Data

In the given dataset, the paper collected vehicle trajectory data from 6 different cities including Austin, Miami, Pittsburgh, Dearborn, Washington DC, and Palo-Alto. In each city, plenty of agents/vehicles were collected to show their 11 second movement trajectory. From the original paper, the data collected consists of variety in terms of seasons, weather, time of the day, etc. to ensure generalization power in prediction models. Table 1 displays the training and testing data sizes provided for each city. From the numbers, the amount of training data available and therefore training difficulty for each city could be reasonably different.

All data contains input dimension of 50×2 and output dimension of 60×2. In particular, a single point is a pair (x, y), where x is the position of the vehicle with respect to the horizontal axis and y is the location of the vehicle with respect to the vertical axis. The input consists of 50 such points, which represents the movements of a specific vehicle in the first 5 seconds, sampled at a rate of 10Hz. The output consists of 60 such points, representing the movements of the same vehicle in the next 6 seconds. The 110 snapshots form a full trajectory where models are expected to predict the location coordinates by learning its direction, speed, acceleration, etc.

Using the plot function, we plotted a batch of 7 trajectories from the training set of Palo Alto, showed in Figure 1. From the batch, we could see that most agent trajectories follow a linear trend; that is, more vehicles tend to drive straight. However, there are also plenty of variation as some vehicles are speeding up, some are slowing down, and others are making turns then speeding up, etc. The ability to capture such variations will be key to the accuracy of the model.

2.2 Statistical Analysis

2.2.1 Input and Output Distribution

Before finding any pattern, we assemble all (x, y) that any agent passed through in each city and display them in a 2D histogram with continuous color showing the densities of points falling in 2D bins. Figure 2 shows the input and output distributions separately for all 6 cities, resulting in $6 \times 2 = 12$ plots. From all the plots, we see that the points of all agents form clear lines indicating the roads in these cities, which look like the nightlight map and essentially assemble into a map of these cities. More specifically, we have found that:

- Major roads stand out more in the maps, as more agents drive on these roads, it's possible to learn the speed limit, the usual traffic conditions, busy streets, etc.;
- Major roads also tends to be wider as they should be in the real world;
- On major roads there are clear crossroads or stop signs that has the highest density among all grid points as agents have to stop and spend more time steps in these crossroads;
- Minor roads are dimmer and narrower, and in most cases only downtown minor roads are showed in cities such as Dearborn and Washington D.C.; other areas containing more residential areas show less minor roads such as Palo Alto;

Table 1: Train and Test Data Size for Cities

City	Train Size	Test Size
Austin	43041	6325
Miami	55029	7971
Pittsburgh	43544	6361
Dearborn	24465	3671
Washington DC	25744	3829
Palo-Alto	11993	1686

- Finally, it seems that the agents are chosen more or less randomly as in all cities since the input and output graphs show almost the same normalized density

Comparing the distributions across different cities, we could see that the positions and their properties are quite different from each other. Other than the existence of residential districts, some cities like Miami show more condensed road structures (and also two connected components) presumably in downtown areas; others show more structured diagonal roads such as Palo Alto and Washington D.C. In short, these differences should be significant enough so we need to inject some biases to different cities. On the other hand, there are also lots of similarities among cities. For example, the vehicle behaviors must be mostly shared regardless of cities, including the need to slow down in both coordinate directions before turning. These properties should allow us to train a general model before fine tuning on the variations described above to each city.

2.2.2 Starting Positions

In the normalization section below, we recognize the scaling factor of fully-connected layers if starting at different positions and choose to translate all agents to start at the origin $(0, 0)$. However, these inputs lose the mapping information shown above in Figure 2. Therefore we summarize only the starting positions for all agents in Figure 3. From the plots, it does confirms with our observation that the agents are sample rather randomly, as the density plot show almost the same pattern as the input and output plots above. That is, a strong final model can learn the positions on the map closely to get a sense of the context of specific road information in the real world.

2.2.3 Curvature

We also noticed from the sample batch plots that most vehicles tend to drive straight while other agents tend to turn left or right. Consequently, we are interested in how many turns agents tend to turn and explore the balance between straight and curvy trajectories. We compute the curvature parameter from velocity and acceleration, and plot the curvature for all possible time steps for all agents as a histogram.

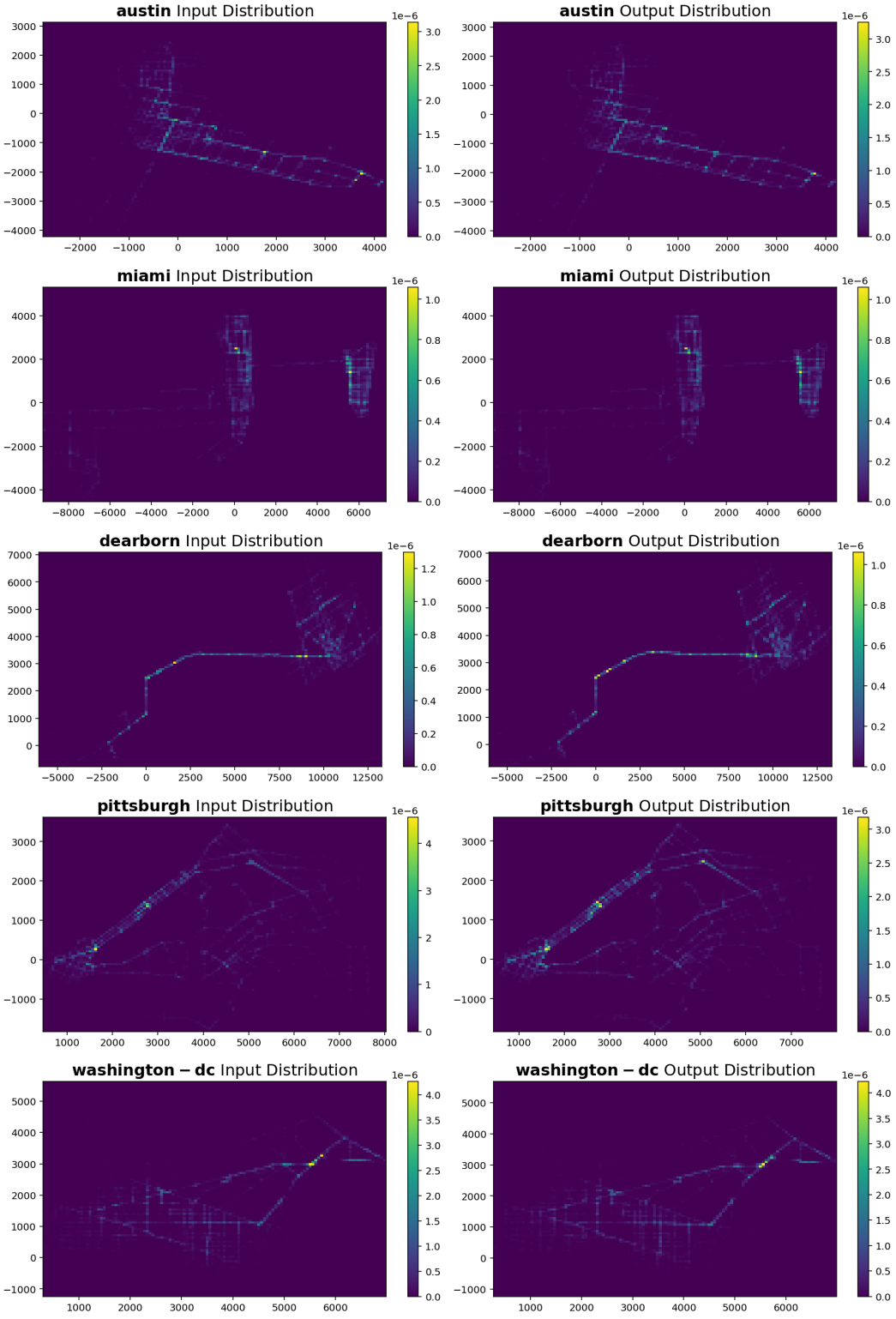
Figure 4 displays the histogram for all cities in a log-scale. The curvature is smaller when the curve is closer to a straight line while it's larger when a sharper turn is needed. From the plots, we see that in all cities the distribution skew to the right, so that in most times the vehicles drive straight and the model should focus more on driving straight and the corresponding velocity. On the other hand, different cities tend to have different distributions, and among all cities Palo Alto seems to have the most straight lines. In contrast, Dearborn has much less straight driving conditions.

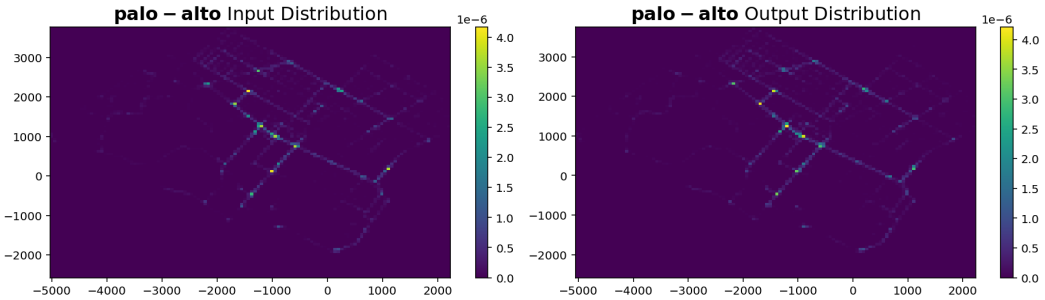
2.3 Preprocessing

2.3.1 Validation Set

To ensure the generalization power of any model trained, we evaluate the model performance on a validation set, which is always 10% of random training examples. In particular, if we trained a city-specific model, then the validation set will contain 10% randomly chosen trajectories from that city; when we trained a generic model on all cities, we shuffle the trajectories and sample 10% of all trajectories from any city to be our validation set.

Figure 2: Input and Output Trajectory Distribution





2.3.2 Feature Engineering

Our approach does not include any feature engineering except normalization in the next section. We only used the normalized (x, y) data as inputs to the model. The main reason behind is that we utilized the power of a deep neural network to learn useful feature representations using a fully-connected trainable linear layer so that the model is capable of expanding the 2-dimensional location data and apply the learned representation to the main parts of the model. The results below showed that the representation is useful in other parts of the prediction. Furthermore, we attempted to add new features such as velocity via careful feature engineering, but they did not outperform the linear layer described above. As a result, our model is not trained based on any feature engineering (except normalization).

2.3.3 Normalization

We recognize that the given trajectories only contain (x, y) coordinates of the agents' location. Exact locations can be drastically different even for the same agent, since it only represents the position of the vehicle. In that sense, it will be problematic to directly feeding the locations into any model since the different scales of the coordinates will affect the convergence of parameters and lead to widely inaccurate predictions. We also realize that the vehicles are very similar in terms of motion, so they should behave in a similar way at the starting position. In addition, similar scales in inputs and parameters allow faster training as well.

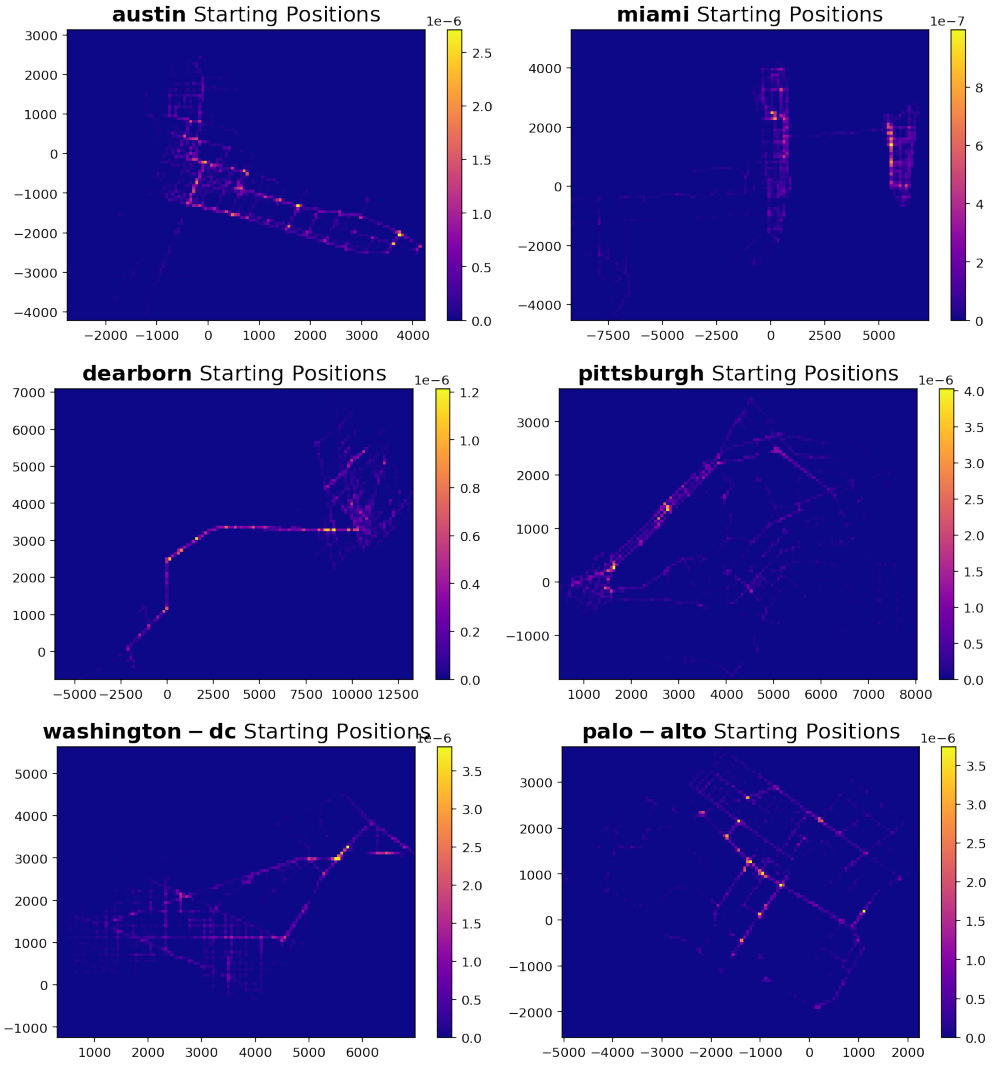
Using the ideas above, we implemented two steps of normalization before feeding the data into the model: translation and rotation. In particular, we record the starting position (x, y) for all agents and move all agents to start at the origin (0, 0). This washes away the effects of exact positions since now the model could focus on movements in a relatively small range (about 0-100). On the other hand, to further alleviate the influences of positive/negative signs when vehicles travel towards various different directions, we implement rotation matrices on each agent so that all agents could start their trajectory by traveling 30 degrees above the x-axis in the first quadrant. Even though agents could still deviate from the first quadrant by making sharp turns after some time, this step still normalizes most agents and allows (mostly) consistent signs of directions.

After normalization, the inputs and outputs of the model are all in the smaller scale, so we also need to revert the normalized predictions to its original scale in order to make predictions. To achieve that, we stored all input positions as well as rotation matrices. Also, all rotation matrices are orthogonal so reverting only requires a multiplication by its transpose; we could then add back the input positions to complete the conversion.

2.3.4 City Information

From our final model described below, the model is firstly trained on all 203,816 agents in all 6 cities, but we also recognize that each city has its own peculiarity and has been explored in the EDA above in sophistication. Given the existence of difference among cities, we utilize information from different cities by taking the generic model and fine tune the model on 6 cities separately. Since the generic model already converged after 100 epochs, the fine-tuning step could only take 3-5 epochs on each of the 6 cities. From the validation MSE observed during the training process, such fine-tuning could further increase the prediction accuracy of the model, and there is also no sign of overfitting.

Figure 3: Agent Starting Positions



3 Machine Learning Model

3.1 Simple Model

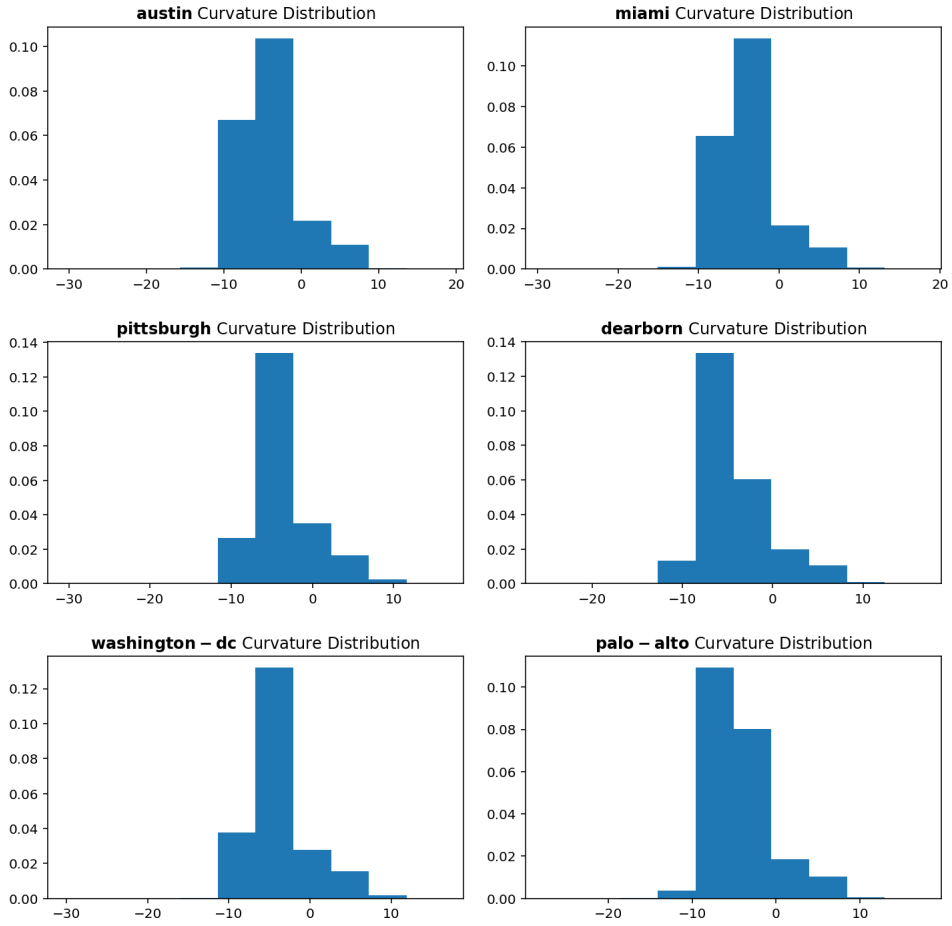
3.1.1 Type of Simple Model

The machine learning model we picked particularly for this task is a single layer perceptron model, which is equivalent of a linear regressor. The reason for picking this model was because of its simplicity in design. With such transparency within the model's structure, we would be able to easily tune and debug the model. More importantly, based on the model's performance, we could conclude if the linear regressor kind of model is capable of capturing the trajectory of each vehicle.

3.1.2 Input/Output Features

The inputs we used are exactly the original features from the data loader: 50 pairs of (x, y) coordinates per sample that describe the first 5 seconds of trajectory for any agents. The Output features are equivalently 60 pairs of (x, y) coordinates per sample that demonstrate the next 6 seconds of trajectory for that agent.

Figure 4: Agent Curvature Distribution



3.1.3 Model Class & Loss Function

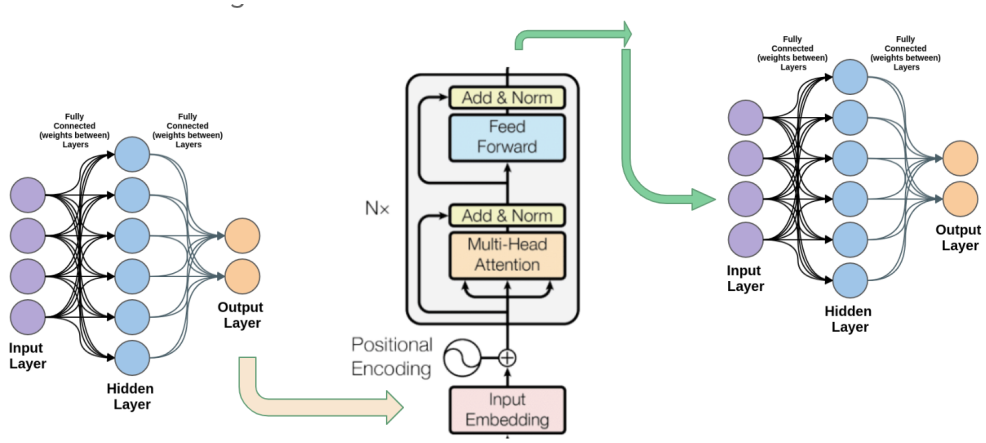
The model class, already identified the the previous section, is a single layer perceptron (SLP) that acts like a linear regressor. The loss function we implemented for our SLP was Stochastic Gradient Descent (SGD). Such implementation of loss function is ascribed to the fact that SGD is easy to understand for the purpose of debugging.

3.2 Deep Learning Model

3.2.1 Deep Learning Pipeline

For our deep learning model, we implemented a three-stage pipeline-embedding stage, encoding stage, and decoding stage-to predict the trajectory. First, the embedding stage takes in the raw input of x-y coordinates and augments the data with its latent features. Our intention is that, through data augmentation, the latent factors (such as velocity, curvature, etc...) can be successfully revealed and thus make it easier for the loss function to converge. The second stage of encoding takes the augmented inputs and add it to a positional encoding vector of the same size. The purpose of such vector is to enforce the presence of position information into the data for the model's learning. As we know, the position is constantly changing as time progresses, hence it is of great importance to incorporate such information into our data. After adding the positional encoding, we fed our inputs into a multi-head attention layer so that the model knows where to focus through this long chain of training. The third and final stage is a multi-layer perceptron decoder that further analyzes the learned matrices and eventually transforms them into the prediction matrices. Starting from the second stage, the model is the standard encoder layer based on the paper [8].

Figure 5: Final Model Architecture



3.2.2 Final Input/Output Features

The final inputs we used are the normalized features from the data loader: 50 pairs of (x, y) coordinates per sample that describe the first 5 seconds of trajectory for any agents after translation and rotation. The Output features are equivalently 50 pairs of (x, y) coordinates per sample that demonstrate the next 6 seconds of trajectory for that agent after translation and rotation.

3.2.3 Model's Architecture & Loss Function

As described in the above parts, our model consists of three stage: embedding stage, encoding stage, and decoding stage. Look at Figure 5 for further reference. In the embedding stage, we essentially performed data augmentations to the original data so that some latent factors of the features could be revealed. Then, we put the the augmented data into a transformer encoder, along with positional encoding as well as multi-head attention layer, to alleviate the disadvantage of a Seq2Seq model. Lastly, we put the encoded data into a MLP decoder layer in order to generalize the encoding part of the data and transform all of them into prediction matrix for the model to learn. In terms of the model's architecture and loss function, this is the only attempt we have tried that produced satisfactory results. We also had some attempts on different architectures that will be briefly discussed in the next subsection.

3.3 Experiments

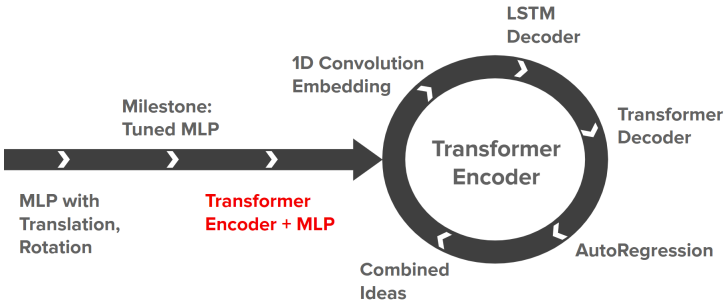
We have experimented with quite a few different deep neural network architectures, with the timeline shown in Figure 6. We briefly summarize the models tried on the timeline and report their performances as well as the prediction steps needed, if the model requires a different training/inference procedure. If not specified below, the model just produces all 60 output time steps simultaneously, and the loss is computed after flattened into a length of 120 tensor. Also, starting from 1D Convolution, the transformer encoders used are all identical as the Transformer Encoder section.

In summary, we discovered the power of transformer encoders in terms of its fast training speed and self-attention mechanism, experimented with various decoders (and a combination of them), and finally returned to the simple MLP decoder as it produces the best predicted trajectories and validation loss.

3.3.1 MLP

We started the exploration of neural networks from the simplest multi-layer Perceptron, using 7 fully-connected layers as the "encoder" followed by 7 fully-connected layers as the "decoder". The dimensions of all hidden layers are set to 64, and the final output is flattened to have 120 units, which corresponds to the dimension of the true labels. This simple model is able to perform well on most linear trajectories and bring the private leaderboard MSE down to 21.37. Due to the high compatibility of most linear trajectories and the linear nature of this model, the performance is relatively promising.

Figure 6: Entire Experiment Timeline



3.3.2 Transformer Encoder

The next big improvement stemmed from the need to encode the 50 input time steps better as well as the need to access all previous time steps in an efficient manner. Under that situation, a transformer encoder could be used, where multi-head attention is computed along with the positional encoding of the time steps, which is the same architecture as the encoder part of the final model. It also fully-utilized parallelism provided by modern GPUs to increase training speed on these matrix multiplications. Using 2 heads, hidden dimension 8 in the transformer layer, and a final fully-connected layer, the model learns better representations and therefore is capable of outperforming simple models like MLP. In particular, this model brings the validation MSE down below 20, at a range of 17-19. Interestingly, even the 120 units of output produced by the linear layer has alternative x, y coordinates, the model could still separate them nicely to fit on reasonable trajectories. Due to the good performance of such an encoder, we keep the encoder in all later experiments.

3.3.3 1D Convolution

Almost keeping all the structures before, we recognize the nature of our sequential data and therefore experimented with a 1D Convolution layer in place of the linear embedding layer. The convolution section consists of two 1D convolution layer and a max pooling layer. Unfortunately, this model is unable to outperform the previous model, so we just mention it here for completeness.

3.3.4 LSTM Decoder / Transformer Decoder

Our next experiment explores the recurrent structure, relying on the fact that we could only rely on previous information when making predictions, but an MLP decoder has to produce all predictions at the same time instead of utilizing predicted information. With the possibility that later predictions might be more accurate if conditioning on the previous predictions, we attempted to use an LSTM or Transformer as the decoder. These models always takes in the previous prediction as the input for the time step (See Figure 7). Note that in this situation, even though a Transformer decoder computes attention scores, it cannot utilize its speed advantage since we must obtain the previous prediction before moving on. The nature of recurrence makes both models extremely inefficient. Compared to the Transformer Encoder + MLP model, this model needs 10 times more time to train. The LSTM model is the recurrent network with long and short term memory based on the paper [1].

Unfortunately, since we have to re-compress the good representations learned from the transformer encoder back to a fixed-length vector for the initial hidden state, the model produces scattered predictions, even it's possible to reach a validation loss of around 50 on the Pittsburgh dataset. Figure 7 also provides a sample prediction from this model that clearly shows when the trajectory is hard to predict, the model is unable to produce satisfying predictions. A side note is that we also attempted to inject teachers' forcing when training; that is, randomly select a few output time steps to feed in the ground truth locations. The sample output also shows a prediction produced when teachers' forcing is injected. In general, the model performs poorly regardless teachers' forcing is present.

Figure 7: Architecture and Predicted Sample from Transformer Encoder + LSTM Decoder

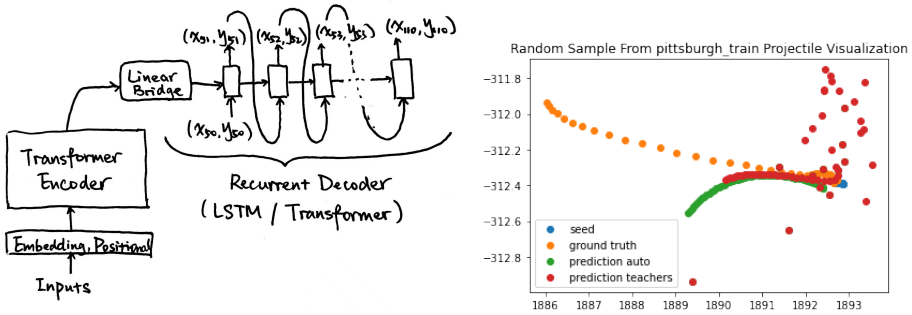
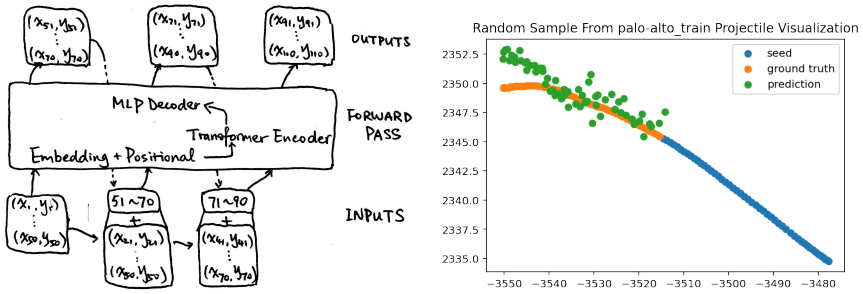


Figure 8: Architecture and Predicted Sample from Autoregressive Transformer Encoder



3.3.5 Auto-regression Predictions

The final experiment was inspired by general time series forecasting: Auto Regression. The idea essentially works by taking input of a fixed length of time and trying to predict the next small chunk of time. In our case, the model always takes in inputs of 50 time steps. We could then let the model only produce 20 time steps (i.e. $t = 51, \ldots, 70$) in one pass, concatenate with the most recent 50 time steps (i.e. $t = 21, \ldots, 70$), predict the next 20 time steps, and repeat the process until all 110 time steps are predicted. The main advantage of such models is that the model now could have less burden by focusing on a smaller time frame and is expected to perform better on the 20 time steps. Then the model could produce accurate outputs recursively by using the same set of parameters and same input sizes. Note that in order to ensure the model scale is unchanged, we translated and rotated all inputs before feeding in the model and reverted to the globally normalized scale after prediction.

Figure 8 visualizes the model architecture used as well as a sample trajectory prediction from Palo Alto. From the prediction, we could observe the main disadvantage of such models: there is no ground truth directions during the iterations. As a result, the predictions will not form a reasonable trajectory, as variations and inaccurate predictions among iterations mess up the parameters. Even if the model reaches a validation MSE of 21.5 on Palo Alto, the scattered predictions prevent us from further developing this experiment into the final model.

3.3.6 Regularization

We utilized a few regularization techniques during model design, mostly relying on the architecture design of the transformer model presented in the paper. In particular, the transformer encoder always contain Layer normalization layers and dropout layers between the multi-head attention section and the feed-forward chunk, so we follow the ideas closely. These layers prevent the model from gradient explosion as well as overfitting. We also utilized a variation of early stopping when training, where we always save the best model with the best validation MSE so the best model might not be the one after 100 epochs. In general, these techniques do help us avoid overfitting to the training data.

3.3.7 Acknowledgements

Upon building the model we experiment with, we send sincere appreciation to Professor Yu and TAs for providing potential directions.

4 Experiment Design and Results

4.1 Training & Testing Design

4.1.1 Computation Platform

All the training and testing are done on the assigned GPU from DSMLP. The intention of training models on GPU is self-explanatory: GPU is faster in terms of computational speed compared with CPU, and thus it will be faster to train on GPU.

4.1.2 Optimizer

Adam is the optimizer that has been implemented for the purpose of training. We did not tune the learning rate decay or momentum. To be specific, the only hyperparameter we tuned is the learning rate itself. The most popular choice of learning rate has been 0.001 across academia, but a larger learning rate is also possible for the model to converge more quickly. However, a constant learning rate will mostly definitely over-fit the model with an increasing number of epochs. Hence a scheduler for the optimizer is also added in order to decrement the learning rate in the middle of training. In other words, the learning rate becomes dynamic and is subject to the overall performance of the model on validation set.

4.1.3 Multistep Prediction

The essence of the task is to make prediction based on time-series progression. Hence, it only makes sense to build a SeqSseq architecture to handle the discrepancy between input and prediction sizes. Yet the confusion arises when it comes to the inner structure of encoder/decoder. One common practice across the industry is to start with multi-layer perceptrons and build up more complex architecture on top of that later on. So this paper also starts with a simple encoder-decoder MLP as the baseline model for the prediction task and later move onto a more complex Transformer-based architecture. The input size for each agent will always be [batch size, 50, 2]. Hence, the input size when we feed the raw data into our model is going to be 2, the last dimension of each batch. To make the multi-step prediction happen, we first performed data augmentations to the original data in the embedding layer so that some latent factors of the features could be revealed. Then, we put the the augmented data into a transformer encoder, along with positional encoding as well as multi-head attention layer, to enforce model's concentration on the more important details of the data. After a redistribution of attention scores, the complicated representation of model's understanding of the path finally get passed into MLP decoder, which generalizes what the model has learned and eventually produces the flattened prediction with size [batch size, 120]. The prediction is simply the next 6 seconds of movement for each target agent. Thus, we finish our multi-step prediction of the vehicle's future trajectory.

4.1.4 Cities Info

Each city has its own peculiarity and has been sophisticatedly explored in the EDA referred above. However, we realized that there is some core similarity behind all cities: most of the roads are linear. Given such astounding discovery, we finally decided to to combine all the train data into a single dataset and feed it entirely into the one model. Then, when the model finishes training on the combined dataset, ewe saved the model and fine tune it on each of the 6 cities. That is, we created another 6 models that are trained individually for each city to confine the performance and further improve the model's understanding of each city. We first trained the single model on combined dataset through 100 epochs of iteration for its completeness of learning. After that, we trained 6 new models on each individual city with 10 epochs to further enhance every model's performance. Given that a scheduler is implemented and thus will tune down the learning rate, there is hardly any concerns for over-fitting.

Table 2: Setup and Results for Different Experimentation

Model	Val/Test Loss	Train Time/Epoch	Parameters
MLP	21.24	1.8	57915
Transformer Encoder (TE) + MLP	17.28	7	78696
TE + LSTM Decoder	50	380	560834
Autoregressive TE	21.5	100	76648
Final Generic TE + MLP	16.5	100	124232

4.1.5 Training Specifics

The batch size for each sub iteration is 32, and the number is particularly decided for a faster convergence rate. It takes 100 seconds to train one epoch of data with a batch size of 32, and it would take relatively shorter period of time as the batch size decreases, for all the models we have trained. Without loss of generality, the time of going through a total training of 100 epochs is about 10000 seconds, or a little bit under 3 hours.

4.2 Experiment Model Performance

We pick a few representative models from our experiments, including milestone, failed models, and final successful model, and report their training time in seconds per epoch with batch size set to 32, validation loss, parameters used, etc. Note that during the process of experimentation, we split our work across members, which lead to results on different parts of the dataset.

From Table 2 for the particular dataset in our situation, we could observe that the models perform very differently under different architectures, and is very closely related to the design choices of these models themselves. In our experiments, we simply discovered that transformer encoders learn good representation of the trajectory, while a simpler decoder tends to produce more accurate predictions.

Runtime rely heavily on the model architecture and the data size inputted into the model. For example, recurrent structures such as LSTM tends to run much slower than a GPU-accelerated linear layers such as Transformers. The latter is much more efficient to train and tune, so the results could be carefully tuned for each model; however, slow models will require much longer to train and could be harder to tune given time restrictions. When the data becomes much larger such as the final model, the runtime also slows down as the training process has to go through much more batches. However, given the fast nature under the final model, it could match autoregression in training time despite having almost 6 times as many data.

Finally we see that all models we implemented contains a huge number of parameters due to the architecture of deep neural networks. However, larger number of parameters don't always lead to better results, which is better determined by the model and its compatibility to the data.

4.3 Generics of the Best Performing Models

4.3.1 Train/Val MSE

Please refer to Figure 9: Training/Validating MSE for the combined cities. We note that there is a peak between 20 and 40 epochs, which is most likely due to the large learning rate, and it's handled by the scheduler so that the remaining curve is smooth. Finally as mentioned above, fine-tuning on different cities only take a few epochs, but the generic performance is also well-represented by this plot.

4.3.2 Random Sample Visualization

Please refer to Figure 10: Random Samples Predictions Visualization. From the plotted dots, we could see that the model has almost perfect performances on linear trajectories and does a reasonably good job if curves and turns exist in the trajectories. However, even though he model achieved top test MSE on the leaderboard, it still has large room of improvement when the shape of the trajectories are more complicated or when they involve complex accelerations.

Figure 9: Training/Validating MSE

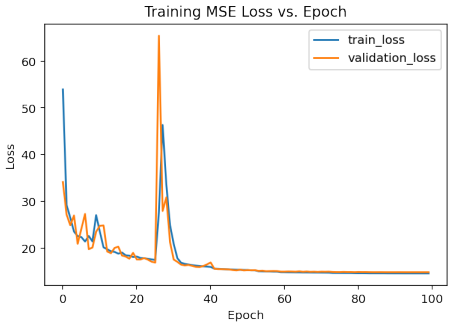
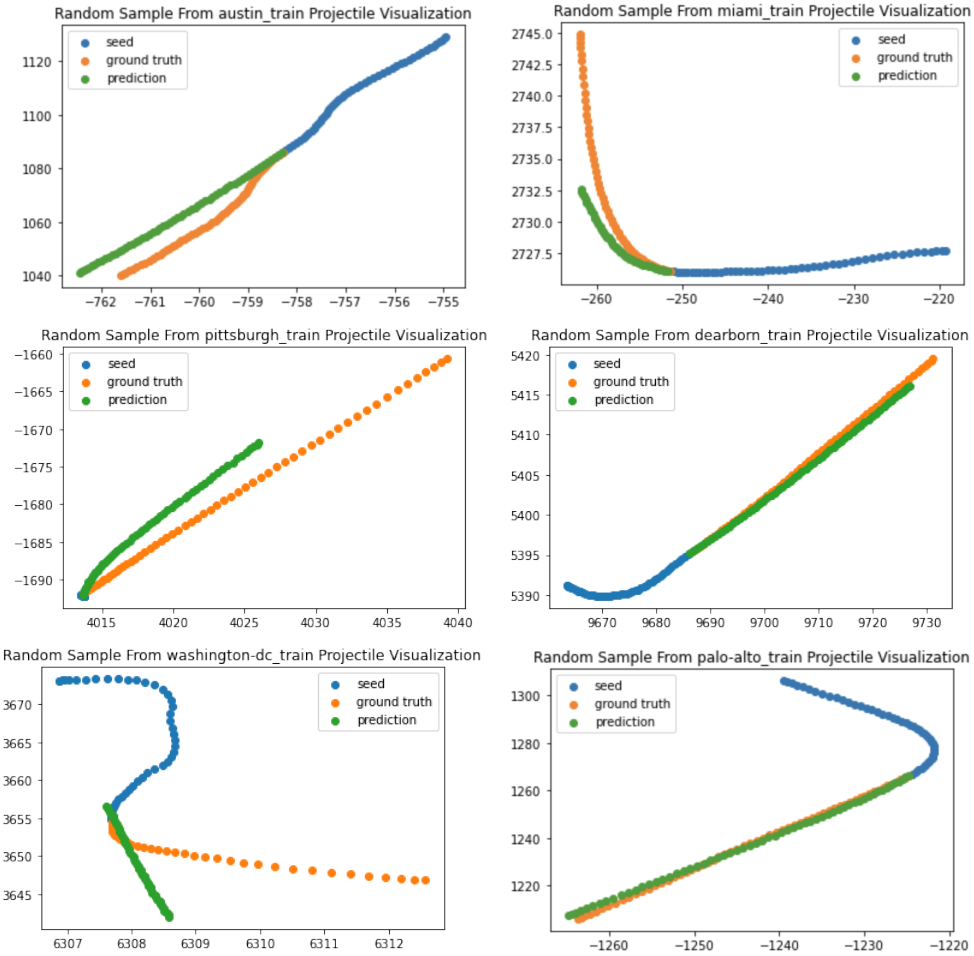


Figure 10: Random Samples Predictions Visualization



4.3.3 Ranking & Test MSE

Our current ranking on the private leaderboard of Kaggle is officially number one, and our corresponding MSE is 15.29349.

4.3.4 Acknowledgements

Upon building the final model we use for the result of the competition, we consulted TensorFlow tutorials [7] and PyTorch tutorials with practical examples [4] and [5]

5 Discussion and Future Work

5.1 Feature Engineering Strategy

One of the most essential strategies we implemented involved translation and rotation, which relates to the scale of the parameters in the deep learning model. This strategy directly brings the trajectories to look similar and the model is capable to learn patterns in the same scale. In general, this strategy falls in normalization, where raw data is converted to be represented under the same scale. Since usually raw data is noisy and could be collected using various different ways, normalization should always be taken into consideration so models could learn more easily.

5.2 Helpful Techniques

We mostly benefited from two techniques in this project: recognizing the power of transformer encoders and its multi-head attention mechanism; recognizing the similarity in cities and the need for more training data.

The first benefit relates to the model design. It's useful to understand the data and the task so that the model design could be closely related to the information. For example, when working with sequential data such as time series, attention could be often useful; even though recurrent architectures don't work out too well, we still consider them as useful experimentation as they fit the task.

The second benefit relates to the training technique. Before the idea evolved, we learned from EDA that cities are very different in terms of their maps so we decided to train 6 different models using almost the same architecture. They worked relatively well after careful hyperparameter tuning and taking the best run among multiple runs. This process is extremely time-consuming. However, when we picked up the idea that all normalized trajectories are similar, we trained the generic model using much more data and it just turned out to be the best model on all cities. Therefore, whenever there is more data available, always use more data to populate the model.

5.3 Bottleneck

The biggest bottle neck of our project is to incorporate the attention mechanism into our model. As we know, Seq2Seq model suffers badly on its performance when the time step of training is too long. And one of the savior is the implementation of attention. The difficulty of implementing the attention for us is specifically that we barely learned anything about its implementation, and we were not sure if it will fit nicely into our MLP model. Hence, the first step we took towards it was to research about the basics about attention. With that being said, we looked up on line and read through different documentations about attention, along with the kind of model that fits nicely with attention mechanism. Eventually, we found out that transformer is a new way to present attention mechanism and to store useful information as training proceeds. Then, we found some online sources about transformer and started to build it from scratch. In the end, our validating MSE decreased from 21 to 17 by simply adding the multi-head attention to one of our encoder layers.

5.4 Advice

For deep learning beginners to start with this project (and generally any project), we find it useful to:

1. Understand the data and the task as well as you can before implementing any model. Use visualizations and exploratory analysis to delve deep into the data and understand the

distributions. Also read the instructions of the task to understand the objective as well as popular approaches attempted to the same kind of task done by people. Specifically in this project, we find distributions of trajectories in each city to be useful, as well as the types of trajectories (linear, curvy, etc.). We also learned about the ideas of autoregression in time series data and how people approached sequential data using either Seq2Seq or sliding window techniques.

2. Also visualize the predictions, especially with time series data, since any loss is just some summary numerical value without context. The model is only working properly if the produced output matches the task proposed, such as trajectories. We understand that even with low loss, the model cannot be considered good if it produced scattered trajectories.

5.5 Future Explorations

When it comes to trajectory predictions, the road information is considered a crucial aspect that may affect the model's prediction. The prediction should always strictly follow the road condition that the vehicle is in. A simple example is that the prediction of a vehicle driving on a straight one-way road should always be a straight line without curves. Our given data did not include such information. Hence, if we were given the location of the starting point of the trajectory and the map information of the city where the vehicle is in, we would definitely attempt to corporate such features into our input to experiment if such features would better the performance of our model. One such example presented in [3] borrows the idea from Word2Vec and attempts to learn a similar embedding for each road in a particular city. We could then identify the road(s) related to a trajectory and use the continuous road embeddings to learn new features about the trajectory.

In addition, the real time of the trajectory as well as the traffic information of the city would be beneficial if they were provided. A certain area of the city may become extremely busy during a certain time of a day, which may lead to heavy traffic. Vehicles that are near the area during the specific time window may move at a steady and slow pace or even change routes, which could affect our trajectory prediction. If we were given such information, we would link such info with the road info that was mentioned previously, and engineer the features together then feed into our model to attempt to get a better performance.

Finally, more realistic autonomous vehicles use sensor information installed at the front of the vehicle to detect the real-time traffic information ahead, including crossing pedestrians, vehicles on other lanes, traffic lights, etc. If we were given such information, we could mimic the approach from [2] to use multiple temporal LSTM layers to predict the sensor situation that's about to happen in the next few time steps. We could also use a Convolutional Neural Network to directly learn from the image (or other signals) captured by the sensor.

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[2] ByeoungDo Kim, Chang Mook Kang, Jaekyum Kim, Seung Hi Lee, Chung Choo Chung, and Jun Won Choi. Probabilistic vehicle trajectory prediction over occupancy grid map via recurrent neural network. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 399–404. IEEE, 2017.

[3] Kang Liu, Song Gao, Peiyuan Qiu, Xiliang Liu, Bo Yan, and Feng Lu. Road2vec: Measuring traffic interactions in urban road system from massive travel routes. *ISPRS International Journal of Geo-Information*, 6(11):321, 2017.

[4] PyTorch. Language modeling with nn.transformer and torchtext. `https://pytorch.org/tutorials/beginner/transformer_tutorial.html`.

[5] PyTorch. Language translation with nn.transformer and torchtext. `https://pytorch.org/tutorials/beginner/translation_transformer.html`.

[6] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. Planning and decision-making for autonomous vehicles. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1):187–210, 2018.

[7] TensorFlow. Transformer model for language understanding. `https://www.tensorflow.org/text/tutorials/transformer`.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.