CSE 151B Project Final Report

Joshua Wong
jow003@ucsd.edu
https://github.com/jbwong05/CSE151B-kaggle

1 Task Description and Background

1.1 Problem A

The task is to train a develop a motion forecasting model for self-driving vehicles. Given a 2 second observation of a scene, the goal is to predict the motion of a certain object 3 seconds into the future. This task is important as autonomous vehicles and autonomous driving are slowly starting to become more prevalent. In order for autonomous vehicles to be reliable and safe, it is important that they are able to reliably predict the movement of cars and objects around them in order to react accordingly. Thus, this task is important in allowing for autonomous vehicles to have a better understanding of the world around them and thus be able to drive safer and more reliable.

1.2 Problem B

Other methods researched by others:

1. Argoverse: 3D Tracking and Forecasting with Rich Maps:
   In order to accomplish this task, this paper describes first filering the data only selecting "interesting" scenes with agent interaction. All startionary objects are removed and 5 second sequences are used. Coordinates are then normalized and mapped to a different coordinate system where each trajectory starts at the origin and ends on the x-axis. Social features are also constructed such as the minimum distance to other objects, number of neighbors, as well as the lane centerline data. Nearest Neighbor and LSTM Encoder-Decoder models are then tested and used to predict the next feature values over the next time steps. [1]
2. Learning Lane Graph Representations for Motion Forecasting:
   In order to accomplish this task, this paper describes being able to take advantage of both the vehicle or actor features as well as the lane and map features. The past trajectories of the vehicles are inputted into a ActorNet to learn the actor features. From the map data, a lane graph is generated and passed as input to a LaneCGN model to learn the map features. The actor and map features are combined and inputted into a FusionNet model which makes the future predictions for the trajectories of the vehicles. [2]
3. Multiple Futures Prediction:
   In order to accomplish this task, this paper describes the approach of constructing a probabilistic model that is able to learn latent variables in order to predict the future motion of agents in a scene. The model is able to make future predictions without explicit probabilistic labels. This is accomplished with a dynamic attention based encoder. [3]
4. LaneRCNN: Distributed Representations for Graph-Centric Motion Forecasting:
   To accomplish this task, this paper describes a graph based model referred to as LaneRCNN. The graphical input is specially designed to be a local lane graph for each actor. The model also contains an interaction module allowing for communication between local lane graphs with a shared global lane graph. The model is able to learn the actor to actor interactions as well as the actor to lane interactions. [4]

Preprint. Under review.

5. TPCN: Temporal Point Cloud Networks for Motion Forecasting:
   To accomplish this task, this paper describes a Temporal Point Cloud Netowrk that is able to learn spatial and temporal features in order to make the future trajectory predicitons. Many of the proposed approaches are inspired from point cloud learning. The problem is split into two dimensions, spatial and temporal. With the spatial dimension, point cloud learning techniques are used to model the individual agents' locations. The temporal dimension handles being able to model the agents' motion over time. [5]
6. TPNet: Trajectory Proposal Network for Motion Prediction:
   To accomplish this task, this paper uniquely focuses on attempting to account for physical constraints such as traffic rules and movable areas. The two stage model proposed is referred to as Trajectory Proposal Network, or TPNet. It first generates a series of possible future trajectories, and then treats the problem as a classification problem identifying which trajectory best fits the physical constraints for the scene. [6]
7. VectorNet: Encoding HD Maps and Agent Dynamics from Vectorized Representation:
   To accomplish this task, this paper describes a graph neural netowrk that takes advantage of different road componenets encoded as vectors. It is then able to model the iteractions between these components. This approach is claimed to have better performance over convolutional approaches that form birds eye view images of the scene and perform convolution in order to learn the features. [7]

1.3 Problem C

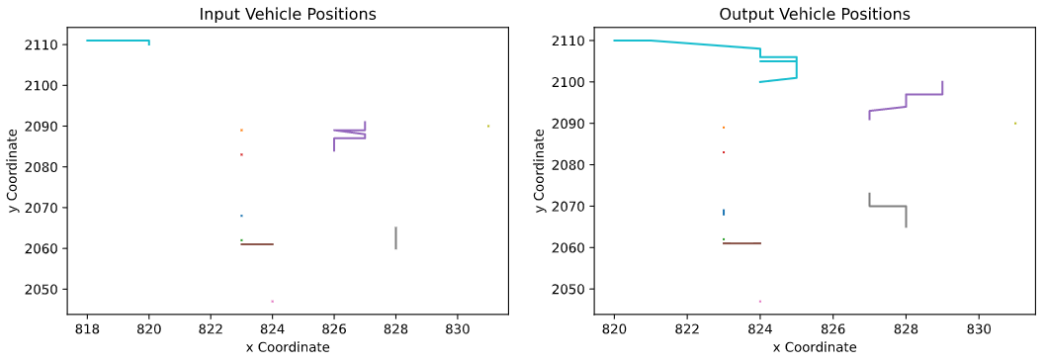
Mathematically, let f be the model we are training. Let the input sequence over 19 time steps be $x_1, x_2, \ldots, x_{19}$. Let the output sequence we wish to predict over 30 time steps be $y_1, y_2, \ldots, y_{30}$. We wish to find a model that learns a function such that $f(x_1, x_2, \ldots, x_{19}) = y_1, y_2, \ldots, y_{30}$.

I believe models trained to solve this problem have the capability of solving other sequential or forecasting problems as long as the input is processed correctly before being fed into the model. In general, models that are able to solve this problem are able to learn a function that is able to take the input features and figure out which are most important in order to output the features for the next time step. With proper tuning and pre-processing of data, I believe models trained for this problem would also be capable of solving other sequence prediction problems. For example, the Encoder-Decoder architechture which can solve this problem is also capable of solving problems in Natural Language Processing such as sequential language translation.

2 Exploratory Data Analysis

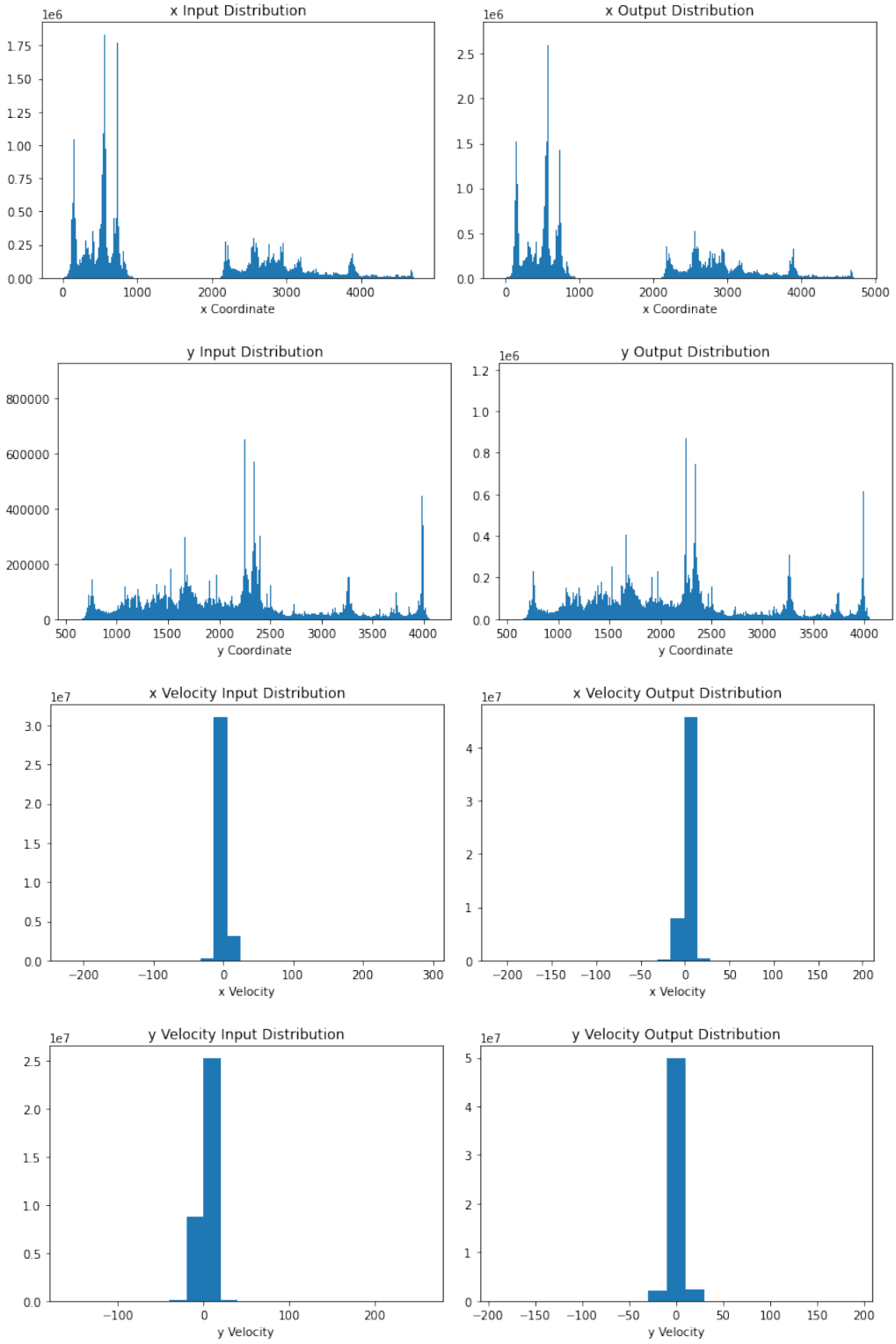
2.1 Problem A

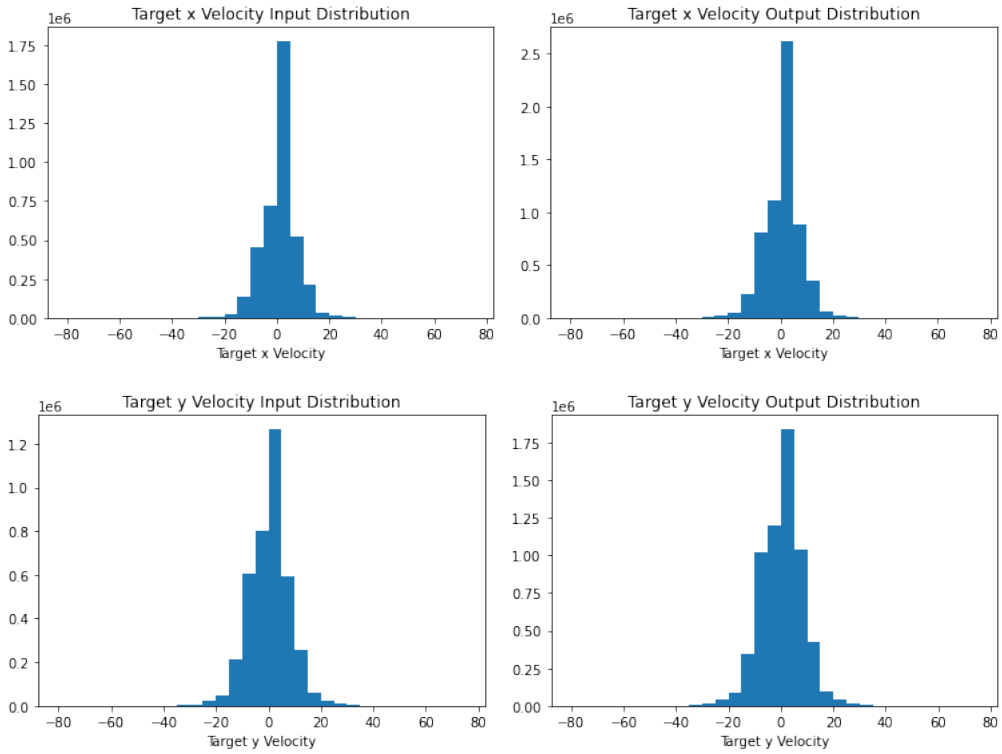
The training dataset has 205942 data points while the test dataset has 3200 data points. Each input is a scene represented by a tensor of shape $60 \times 19 \times 4$. The input contains the input positions (x,y) and input velocities (x,y) for up to 60 different vehicles over 19 different time steps. The output is the positions (x,y) and velocities (x,y) of these vehicles over the next 30 time steps. This is represented by a tensor of shape $60 \times 30 \times 4$ where the only difference from the input is that the output contains the positions and velocities over the next 30 time steps. A sample from the training set is given below:



2.2 Problem B

The distributions for the input positions, output positions, and velocities are given in the following histograms:

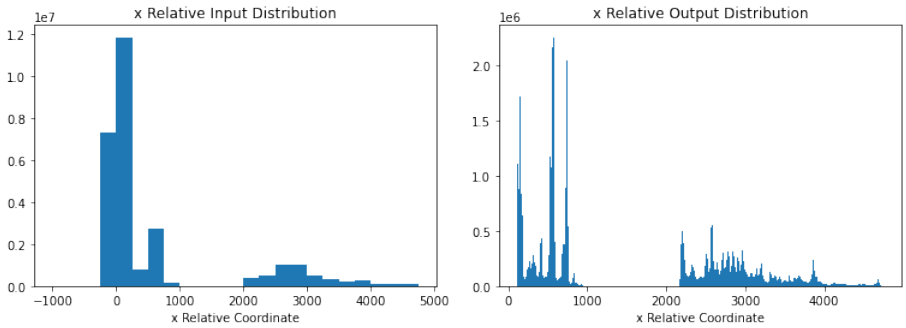


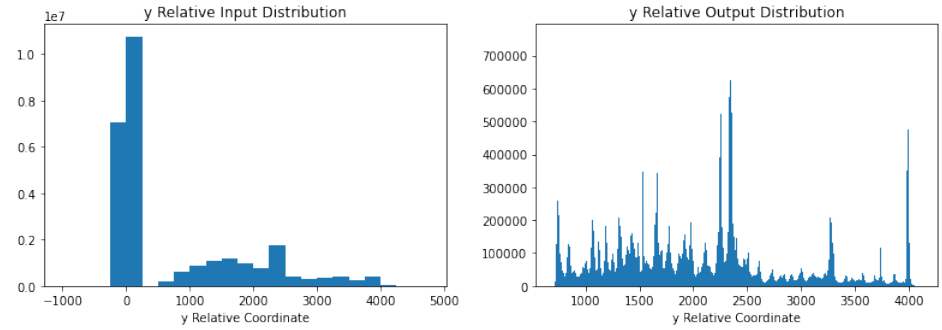


Based on the histograms shown above, there seems to be little overall changes in the distributions of position when comparing the distribution of input positions to the distribution of output positions. The targets' velocity's distribution also experience little change when comparing the input to the output. This however does not imply that there is no movement by all of the vehicles in The scene. Looking at the distribution of all agent velocities, it is clear that there is a clear change in the distribution. while many of the agents do have 0 velocity, there is a noticeable change in number that do not have 0 velocity when comparing input to the output. This implies that many agents are changing velocities from the first 19 time steps in comparison to the next 30 time steps.

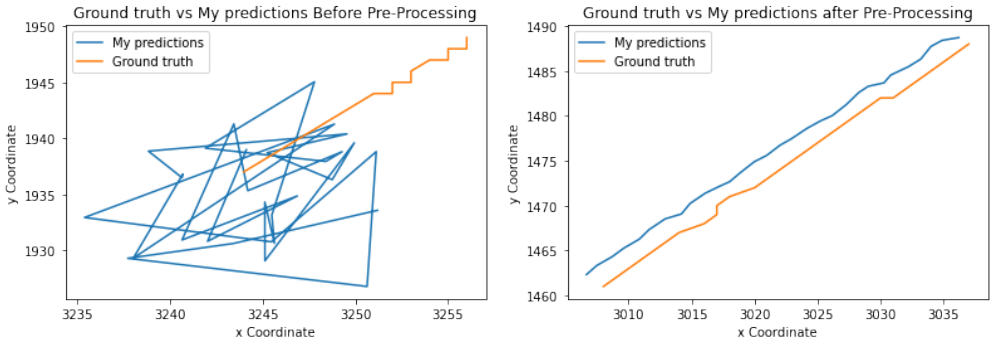
2.3 Problem C

I did not use any feature engineering to accomplish this task. I primarily used the given positional and velocity features that were provided with the initial dataset. Lane information was not used by my models, although I do believe in incorporated properly could lead to better performance. For most of the competition, I did not think to normalize the data before sending it to the model. Only towards the end of the competition did I decide to change the positions so that all of the vehicle positions were relative to the target vehicle. Otherwise the raw absolute positions appeared to be all over the place which I believe was making it difficult for some of the models to fit the data. I also later switched to only feeding the model the 6 closer vehicles as well as the target vehicle to make the predictions more focused around the target vehicle. The new input and output distributions can be seen below:





Pre-processing ended up having a major effect on the performance of my models. The effects can be seen with the graphs below:



3 Deep Learning Model

3.1 Problem A

For most of the competition, the input was simply the input positions and velocities for all of the vehicles and the output was also the output positions and velocities for all of the vehicles. Only towards the end of the competition did I consider pre-processing the data. After pre-processing the input was the input positions relative to the target vehcile as well as the input velocities and the output was either the output positions and velocities or just the output positions depending on the model I was training.

I used both RMSProp and Adam for optimizers depending on the model. For some models RMSProp performed better, and for others Adam seemed to achieve better performance. I tended to stick to optimizers with momentum to increase the speed that minimums could be found. Training loss after a couple of epochs were used to compare optimizers.

I started out with just using a simple linear model because that was what I had the most prior experience with. Towards the end of the quarter, I worked with the encoder-decoder architechture which is supposed to be good at working with sequential data due to the recurrent nature of the RNNs which fits the sequential nature of the problem at hand.

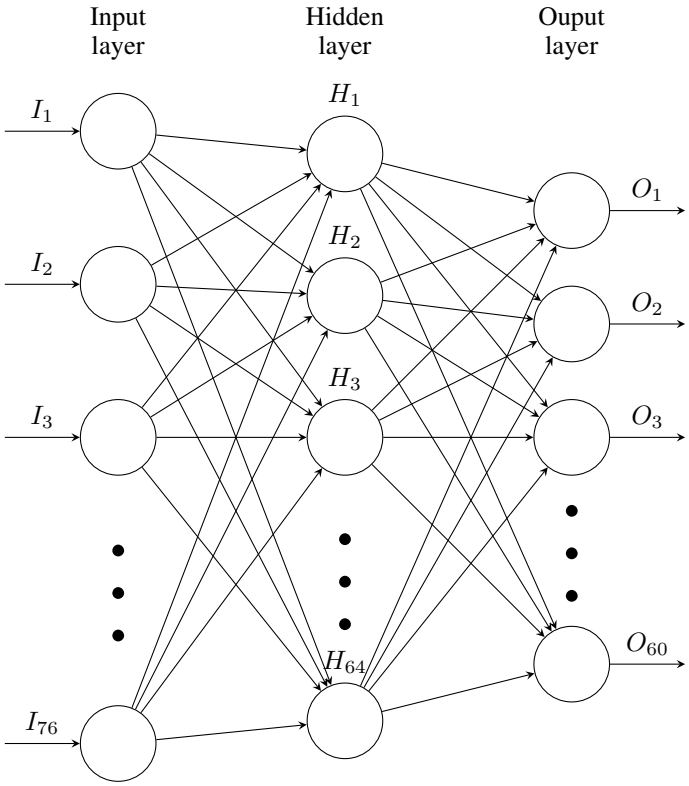
3.2 Problem B

Model Architectures (All model architechtures plus experimental ones can be found at my github repo here)

1. Linear - Simple linear model that with three layers that takes the positions and velocities of the target agent and the 6 closest vehicles and outputs their final positions over the next 30 time steps. A visualization can be found below. The model can be found here .

2. Encoder Decoder - Most of the following architechtures area a version of the Encoder-Decoder architechture with just different implementations of the encoder and the decoder. A visualization for the Encoder Decoder with Attention can be found below.
   (a) RNN-RNN with Attention - Version of the Encoder-Decoder architechture in which both the encoder and the decoder are standard RNNs with hidden state size of 512. This model takes the positions and velocities of the target and the 6 closest vehicles as input and outputs the positions and velocities of these vehicles. The model is autoregressive and uses previous output to predict the next time stamp. An attention mechanism is also used to select which of the hidden states of the encoder to use for the next input state into the decoder. The model can be found here.
   (b) RNN-RNN without Atttention - Exact same as the model above except without the attention module and with a hidden state size of 128. The first hidden state for the decoder is instead the last hidden state from the encoder, and each subsequent hidden state for the decoder is the previous hidden state from decoder. This model can be found here.
   (c) CNN-RNN - Version of the Encoder-Decoder architechture in which the encoder is a convolutional neural network and the decoder is a standard RNN. The encoder consists of a convolutional layer followed by a max-pooling layer. An attention layer is used to select which of the convolved features are passed to the RNN for the next hidden state. The decoder is a standard RNN with a hidden state size of 512. This model takes the positions and velocities of the target and the 6 closest vehicles as input and outputs the positions and velocities of these vehicles. The model is autoregressive and uses previous output to predict the next time stamp. The model can be found here.
   (d) CNN-LSTM - Version of the Encoder-Decoder architechture in which the encoder is a convolutional neural network and the decoder is a LSTM. The encoder consists of two convolutional layers with two max-pooling layers as well. The decoder is a LSTM with a hidden state size of 200. This model takes the positions and velocities of the target and the 6 closest vehicles as input and outputs the positions and velocities of these vehicles for the next 30 time steps. The model can be found here.
   (e) LSTM-LSTM - Version of the Encoder-Decoder architechture in which the encoder is a LSTM with a hidden state size of 512 and the decoder is also a LSTM with a hidden state size of 512. An attention layer is also included to select which of the hidden and cell states from the encoder are passed to the decoder for the next hidden and cell states. This model takes the positions and velocities of the target and the 6 closest vehicles as input and outputs the positions and velocities of these vehicles for the next time step. The model is autoregressive and uses previous output to predict the next time stamp. The model can be found here.
3. RNN - Simple RNN with a hidden state size of 200 followed by 3 linear layers. This model takes takes the positions and velocities of the target agent and the 6 closest vehicles and outputs their final positions over the next 30 time steps. It similar to the encoder portion of the encoder decoder architechture with linear layers following to properly map the RNN hidden state output to the desired output. The model can be found here.

Diagram of the Linear model architechture. The model consists of three linear layers that map an input of batch_size $\times$ 7 $\times$ 19 $\times$ 4 to an output of batch_size $\times$ 7 $\times$ 60 which can be reshaped to an output of batch_size $\times$ 7 $\times$ 30 $\times$ 2:



General diagram of the Encoder-Decoder architechture. Both the encoder and the decoder are not restricted to recurrent neural networks or LSTMs and can be substituted for with any model as long as the model is able to map the input to a sequence of hidden states in the case of the encoder or map the hidden states to the output in the case of the decoder. The attention module is also optional, however some of the models I tested include it. Input to the encoder is the initial input sequence and the initial states. Input to the decoder is either a starting token for the first time step or the previous output. This strategy is known as auto regression and is the one I used as teacher forcing seemed to worse results from my testing:

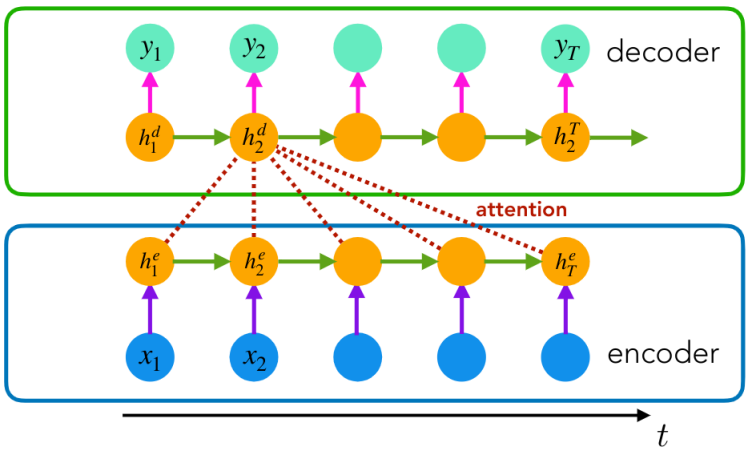


Figure 1: General Encoder-Decoder Architecture with Attention [8]

4 Experiment Design

4.1 Problem A

Because I do not personally own a dedicated GPU, I chose to use UCSD's datahub platform for training and testing models. Google's collab platform was also a consideration, but it had a couple drawbacks including increased difficulty in loading the actual dataset and less GPU memory. UCSD's datahub platform does not suffer from these issues and is the platform that I have been using. It provides students with a Ubuntu 20.04 virtual machine with a Intel Xeon Gold 6130 and a GeForce GTX 1080 Ti.

I have mainly been the RMSprop or ADAM optimizers in order to try to avoid any vanishing gradient problems. Tuning of hyperparameters such as learning has already started with making observations about the training loss values of the model such as the training loss fluctuating and then tweaking corresponding hyperparameters and then seeing how the changes affect the training loss. If the tweaks do not improve the training loss, the changes are reverted and other changes to the hyperparameters are attempted. Making multistep predictions for each target agent has often involved trying to extract features from the given input data and use these features to predict the next 30 time steps either iteratively going step by step or all at once trying to predict the entire 30 step sequence. Often times linear layers have been used in order to transform the data into the proper shape and size. Sets of 3-5 epochs have been used with monitoring being done to observe when the training error converges. The batch-size is currently 64, 128, or 512 and it typically takes around 30-60 minutes for one epoch obviously depending on the size and the architecture of the model.

Many of these design choices have been made based on the little past experience I have with machine learning and the material covered in class. The one parameter that is slightly more constrained is the number of epochs. The time for each epoch is slightly large due to the dataset having a lot of datapoints and I unfortunately do not have infinite time. UCSD's datahub platform is also not always 100% reliable with regards to virtual servers remaining stable after long sessions. As a result, I have used the strategy of training with smaller number of epochs and then saving the model so I can possibly resume training later if needed.

5 Experiment Results

5.1 Problem A

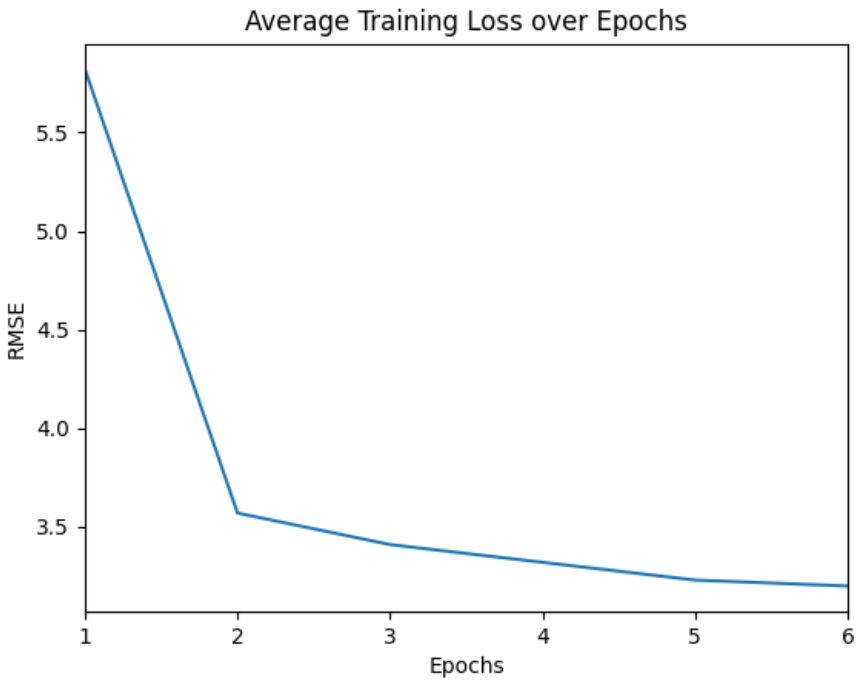
Model Architecture	RMSE	Number of Parameters	Time per Epoch
Linear	3.15843	12988	25-30 min
Enc-Dec (RNN-RNN) with Attention	3.21293	763951	30-40 min
Linear Regression*	12.30562	29128	25-30 min
Enc-Dec (RNN-RNN)*	16.15363	111944	45-60 min
Linear (Incremental Prediction)*	37.40021	4708	25-30 min
Enc-Dec (CNN-RNN)*	49.79448	2360766	45-60 min
RNN*	60.75703	90856600	45-60 min
Enc-Dec (CNN-LSTM)*	284.63522	34907424	45-60 min
Enc-Dec (LSTM-LSTM)*	538.00871	101506	45-60 min

* Before data pre-processing was implemented

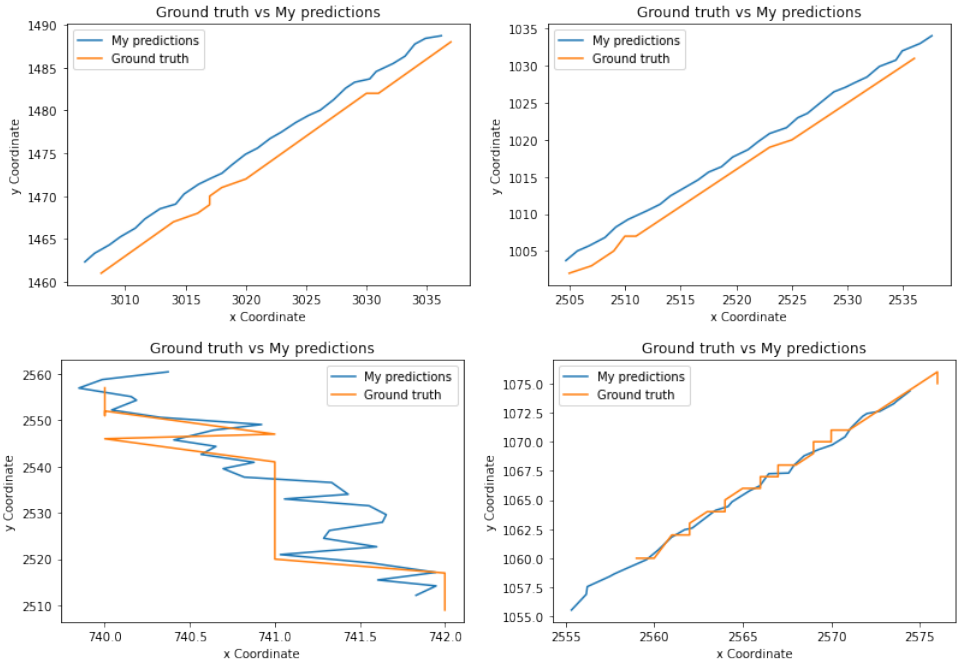
In order to reduce training time, I increased the batch size as well as decreased the number of paramters in the model.

5.2 Problem B

RMSE for best-performing model:



Ground truth vs My predictions:



Current ranking and final RMSE:

6 Discussion and Future Work

6.1 Problem A

Pre-processing the data and performing feature engineering was definitely the most helpful for me in improving my score. For most of the competition, I actually was not pre-processing the data at all and was stuck with a RMSE score of about 12. Choosing to make the positions relative to the targent agent as well as only focusing on the 6 closest vehicles to the target helped me to improve my score. Not pre-processing my data was definitely my biggest bottleneck in combination with datahub not being 100% reliable. I had previously thought that my model was complex enough and did not even consider the thought of processing the data beforehand. If I had considered pre-processing my data sooner, I would have had more time in the end to focus on my tuning my hyperparameters. Only after hearing how important pre-processing was to other groups did I realize my problem. I would advise beginners to start with simple models and to not overlook the pre-processing step. It is important to first understand the given data before writing any code.

In the future, I would like to explore properly using an LSTM Encoder Decoder with Attention and preprocessing instead of an RNN. I would also like to be able to focus more on the tuning of the hyperparameters. I believe also trying to incorporate the lane and map data into the model's input would also be worth exploring to improve the accuracy of cases such as turning.

References

[1] 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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[2] Ming Liang, Bin Yang, Rui Hu, Yun Chen, Renjie Liao, Song Feng, and Raquel Urtasun. Learning lane graph representations for motion forecasting. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 541–556, Cham, 2020. Springer International Publishing.

[3] Yichuan Charlie Tang and Ruslan Salakhutdinov. Multiple futures prediction. *CoRR*, abs/1911.00997, 2019.

[4] Wenyuan Zeng, Ming Liang, Renjie Liao, and Raquel Urtasun. Lanercnn: Distributed representations for graph-centric motion forecasting. *CoRR*, abs/2101.06653, 2021.

[5] Maosheng Ye, Tongyi Cao, and Qifeng Chen. TPCN: temporal point cloud networks for motion forecasting. *CoRR*, abs/2103.03067, 2021.

[6] Liangji Fang, Qinhong Jiang, Jianping Shi, and Bolei Zhou. Tpnet: Trajectory proposal network for motion prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[7] Jiyang Gao, Chen Sun, Hang Zhao, Yi Shen, Dragomir Anguelov, Congcong Li, and Cordelia Schmid. Vectornet: Encoding hd maps and agent dynamics from vectorized representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[8] Rose Yu. Lecture_07-1, 2021.