CSE 151B Project Final Report

Wesley Chen
wec137@ucsd.edu

Brent Min
bjmin@ucsd.edu

Trinity Pham
ttpham@ucsd.edu

Brandon Phan
bnp009@ucsd.edu

The GitHub repository for our solution is available on GitHub at:

`https://github.com/wes-chen/fireteamassemble`

1 Task Description and Background

1.1 Problem A

Our task is to perform motion prediction given a video sequence.

In this competition, we are tasked to predict an object's location three seconds into the future given the positions and velocities of the object over a two second time span. We are also given the positions and velocities of other objects in the frame of the camera, likely to help in our predictions.

The implications of being able to perform this task successfully are massive - if we are able to consistently predict an object's motion in the future, we can envision a future automating tasks that would otherwise require us humans to do this computation ourselves. These tasks are endless: they could include self driving cars, machines that perfectly shoot basketballs, self navigating robots to explore other planets, and more. Our task specifically focuses on the self driving car aspect. In an age where self-driving cars are becoming more prevalent, robust models can help prevent disasters as well as create a more efficient society. Self-driving cars can also help the elderly and disabled become more independent if they are not able to drive due to difficulties such as motor impairment or worsening eyesight.

1.2 Problem B

To research this task, we first looked into the Argoverse [4] paper: the official research paper accompanying the dataset. Within this paper, it compares several baseline models that they implemented such as a constant velocity model, a neural network model, a LSTM model, etc. In one of the comparison metrics, the minimum average displacement error for each model, LSTM performed the best. We would base our model from their baseline.

We also looked into Giuliari et al.'s trajectory forecasting research [1] using transformer networks. They take advantage of attention based memory in original and bidirectional transformers to predict trajectories of agents in scenes similar to our task. Additionally, Giuliari et al. were able to predict multiple trajectories for an agent and account for missing data as sequential data is not guaranteed in real life.

After implementing our baseline encoder-decoder LSTM model, we looked for ways to improve our model. From Schmidt et al's paper "Descending through a Crowded Valley — Benchmarking Deep Learning Optimizers" [5], they found that optimizer performance varies across tasks and model performance is similar whether it uses the default optimizer hyperparameters or tunes the optimizer hyperparameters. Additionally, on their tasks, they found ADAM to perform generally well. We would reference this paper as we experimented with different optimizers.

From Luo et al.'s paper "Adaptive Gradient Methods With Dynamic Bound of Learning Rate" [2], we found that using a scheduled learning rate to manually tune the learning rate for each epoch resulted in better loss convergence because the optimizers we used gave us learning rates that were too high or too low. They discovered this engineering trick after analyzing poor generalization of adaptive optimization methods such as Adam and proposing variants of these optimization methods that implement this to obtain better results.

In search of more ways to tune the model's learning rate, we looked into Smith's research on cyclical learning rates for training neural networks [3]. He finds that cycling between learning rates for each epoch more beneficial than monotonically decreasing the learning rate since this allowed the learning rates to be bound to reasonable learning rates. We would implement cyclic learning rates in our experiments.

1.3 Problem C

Mathematically, we are given a scene, which is a sequence of frames containing spanning time t (in seconds) of features $p(t_{train})_{in}^{(i)}$ where $p(t)_{in} \in \mathbb{R}^2$ represents the x, y positions for each object i, and $v(t_{train})_{in}^{(i)}$ where $v(t)_{in} \in \mathbb{R}^2$ represents the x, y velocities for each object i and a specific $agent$ to track the position for over $t_{train} = \{0, 0.1, 0.2, \ldots, 1.8, 1.9\}$. From these frames, we want to predict $p(t_{predict})_{out}^{(agent)}$ where $p(t)_{out} \in \mathbb{R}^2$, which is the x, y position for the specified agent over times $t_{predict} = \{2.0, 2.1, 2.2, \ldots, 4.8, 4.9\}$ that we want to track in each respective scene.

From the abstraction, our model could predict tasks like ball trajectory for sports (basketball, tennis, soccer, etc) as well as potentially player movement for the same sports. Both the proposed tasks and current car trajectory task require historical time-series data of x, y positions and velocities to output the predicted position for the future time steps. We could also adapt our model to predict other sequence prediction tasks like stock prediction and weather prediction by decreasing or increasing the number of features. For stock prediction, we would not need y positions and just use historical price points to predict future price points. For weather prediction, we could use our model to predict the trajectory of clouds and add features such as wind speed.

2 Exploratory Data Analysis

2.1 Problem A

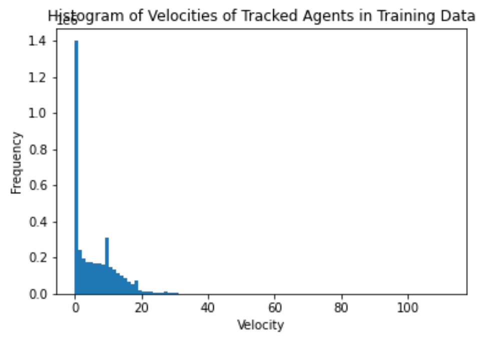
There are 205,942 training set pkl files and 3,200 testing set pkl files. Values of each pkl file include:

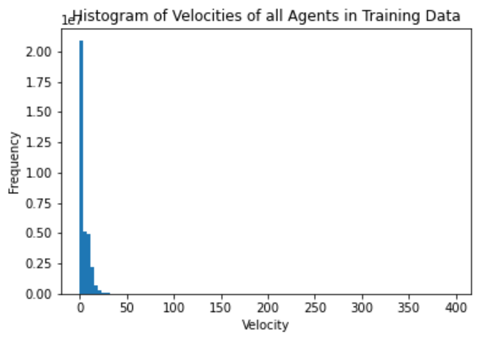
- 'city': City where the scene took place (Either 'PIT' or 'MIA').
- 'scene_idx': Unique identifier of the scene.
- 'agent_id': ID of the object being tracked in the scene.
- 'car_mask': Array of length 60 that only has 1s and 0s. There is a 1 for each object in the scene.
- 'track_id': IDs of all the objects in the scene.
- 'p_in': The x and y coordinates of all objects in the scene. The positions are sampled every .1 second over the course of 2 seconds. The shape of this is 60x19x2; 60 objects, 19 time samples, x and y position.
- 'v_in': The x and y velocities of all objects in the scene. The velocities are sampled every .1 second over the course of 2 seconds. The shape of this is 60x19x2; 60 objects, 19 time samples, x and y velocity.
- 'p_out': Only in the training dataset; the target positions that are compared against the predictions. The shape of this is 60x30x2; 60 objects, 30 time samples, x and y position.
- 'v_out': Only in the training dataset; the target velocities that are compared against the predictions. The shape of this is 60x30x2; 60 objects, 30 time samples, x and y velocity.
- 'lane': Array of x and y coordinates of the number of L lanes in the scene.
- 'lane_norm': Array of vector directions (x,y) of the number of L lanes in the scene

We extrapolate only the positions (p_in) and velocities (v_in) of all of the 60 objects in a scene to use as our inputs. As such, each input is a 60x19x4 input, where there are the x,y positions and x,y velocities over 2 seconds sampled every .1 second starting from zero (19 samples total) for up to 60 objects. The output has a similar dimension of 60x30x4, where instead of 19 samples, there are 30 samples representing the 3 seconds into the future sampled at .1 seconds.

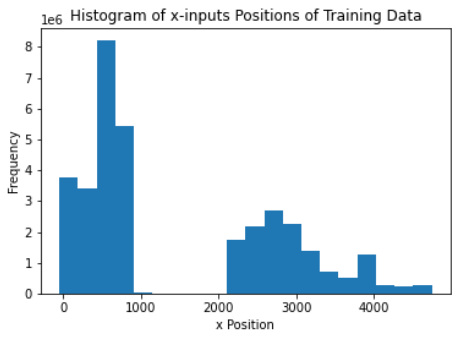
2.2 Problem B

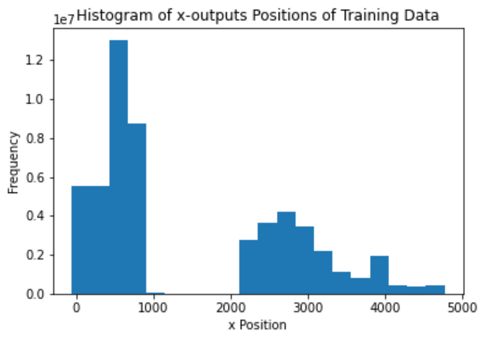
To get a better grasp of the properties of the dataset, we plotted the velocities and positions.



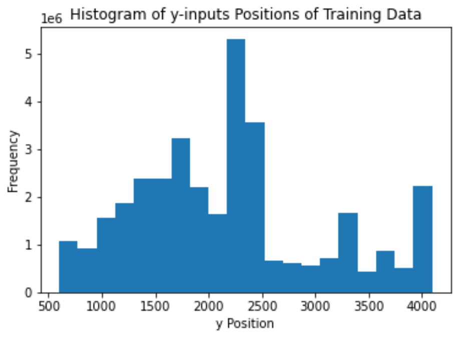


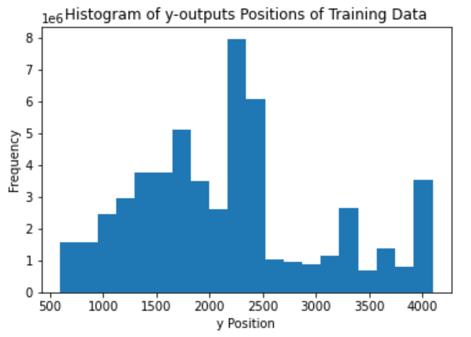
For velocities of the agents, there is a heavy right skew in the histogram which represents that a majority of cars are going relatively slowly. There are a few velocities that are very high, which may be from ongoing traffic on the highway.





With the x-positions of the training data, there is a bimodal distribution. There seems to be a very sparse number of objects that have the x-positions between the 2000 and 3000 x-positions. This may indicate a blind spot or another defect with the camera.





With the y-positions of the training data, there is a unimodal distribution with a slight skew to the right. There does not seem to be as many objects with higher y-values.

2.3 Problem C

The only preprocessing we deemed necessary was reformatting and normalizing our data. We wanted our data to be in the tensor format of batch_size x sequence_length x feature_length, so creating a custom collate function was crucial. Then to normalize our data, the first (x,y) position is subtracted from every (x,y) position for each object, resulting in relative positions and starting at position (0,0). This was to prevent the model from having skewed predictions from the large numbers being used as inputs. After the model produced its predictions, the first (x,y) position is added to the predictions, essentially creating the absolute position predictions.

Additionally, because the model does not use social variables with the other objects, we decided to use all the objects in every scene to train our model. We believed this would be beneficial as it would give more training data to our model, which would help the model overfit less and generalize better.

We did not find feature engineering necessary, as our model only used positions and velocities as inputs. We also did not utilize the lane information provided in each pkl file because we found it difficult to incorporate.

3 Deep Learning Model

3.1 Problem A

We believed that using an LSTM Encoder-Decoder model would score well given the time series nature of data. After reviewing the official Argoverse paper[4], it confirmed our beliefs as their baseline LSTM model performed best and decided to implement the model. We decided improve upon the model because we were getting consistently good results with each iteration. The final model is an LSTM Encoder-Decoder model. The encoder has two layers, one fully connected layer with a ReLU activation which then leads to an LSTM cell. The decoder has three layers, a full connected layer with a ReLU activation into an LSTM cell into another full connected layer.

After processing our data, the data that would serve as the input to the model was in the tensor format of batch_size x sequence_length x feature_length. A batch consisted of one or more objects. For each object, the input has the sequence_length of 19, representing 2 seconds. For each sequence/timestep, there are 4 features, x,y positions and x,y velocities.

The output data would be in a similar format of batch_size x sequence_length x feature_length, where the only difference in tensor format size would be the sequence length. The sequence length for the output would be 30, representing the 3 seconds of predictions.

Because the competition used RMSE to judge our models, we subsequently used RMSE as our loss function. The idea behind it was that minimizing the loss while training would in turn minimize the loss on the test data because the loss function is the same.

3.2 Problem B

Overall, the road to our model was very short, as we had only tried one other model architecture before our final one. The first neural network was a simple RNN/LSTM because the format of the data was sequential and the desired output is also a sequence. Recurrent neural networks made the most sense to fit to this type of data.

We first tried a simple LSTM model where we feed the first two seconds of agent positions into a many-to-one LSTM and use that output as input into a many-to-many LSTM. The many-to-many LSTM uses the output of the previous hidden state as input to the next one to predict agent positions for the next three seconds. However, we encountered a very poor test score not because our model was actually ineffective, but because we failed to normalize the trajectories from a (0, 0) and relative positions. So what would happen is that our model would be fed absolute positions in the scene, and have to try to predict sequences off of that. For our models attempted without normalization, the sequences would all converge into the same position, which is not much better than the baseline submission of all zeros. However, we did not realize this until the implementation of our second model was almost complete.

Upon further research, we came across the Argoverse paper that also contained some baseline metrics off of different types of architectures. One of them was a simple encoder decoder architecture using LSTM based solely off the X and Y positions. We were inspired by this, and adopted an encoder decoder model and adding our own changes. We decided to use velocities as well as positions, as we believed it would help the model determine direction and acceleration. Because we achieved a fairly competitive score with this model early on, we decided to improve hyperparamaters and make small adjustments to optimize our model instead of creating a new model.

We tested implementing regularization techniques such as dropout and batch-norm with both the default hyperparameters and tuned hyperparameters. However, our model performed worse with these regularization techniques, so we decided to not implement them in our final model.

4 Experiment Design

4.1 Problem A

We used an EVGA GeForce RTX 2080 XC ULTRA GAMING, 8GB GDDR6, Dual HDB Fans & RGB LED Graphics Card 08G-P4-2183-KR for training and testing. We chose to use this over the current environment in DataHub because when running our current model, it took longer for the DataHub environment to complete one epoch (about 3 hours) in comparison to our local environment with the RTX 2080 (about 1 hour 30 minutes).

We did not split our data into training and validation sets. We instead used Kaggle's evaluation metrics to compare model performance since we started early and had time to run and submit all of our experiments.

We used an Adam optimizer and are currently using the default learning rate, learning rate decay, momentum, and other parameters. Adam has an adaptive learning rate that automatically adjusts for us. In addition, we tuned Adam's learning rate by implementing engineering tricks such as scheduled and cyclical learning rate, with a 0.5 scheduled learning rate multiplier producing the best results.

Our current model is an encoder-decoder model. The decoder outputs a sequence of predictions for each timestep; the multistep (30 step) predictions. The decoder outputs the position and velocity at a given timestep. This output is then fed back into the decoder to give the position and velocity of the next timestep. To get the final prediction, the original (x,y) position at timestep 0 gets added back to all positions.

After testing our model with 4-12 epochs and 2-8 batch size, we concluded that eight epochs with a batch-size of four gave the best performance, taking about 1 hour and 30 minutes on our local environment to train our model for one epoch.

While we were modifying our model, we used two epochs to check and use the loss to determine our future modifications to the model. In addition, we chose 8 epochs to not overfit the model.

epochs	4	5	6	7	8	9	10	11	12
RMSE	2.04862	2.03341	2.02725	2.02881	2.02689	2.02851	2.02928	2.02956	2.02919

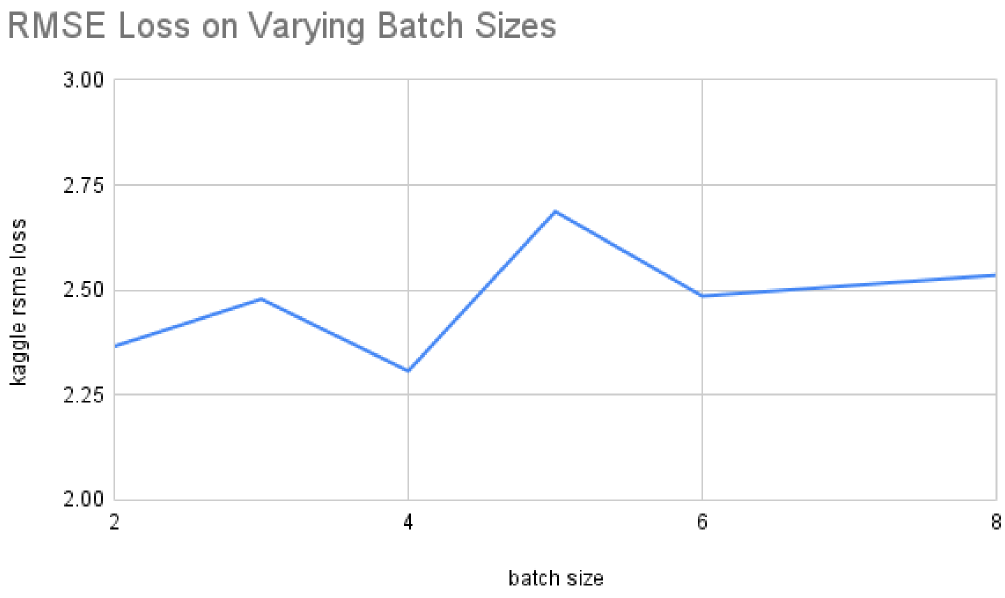
5 Experiment Results

5.1 Problem A

Our model is an encoder-decoder with LSTM units. Our baseline model did relatively well with an RMSE loss of 3. To improve our baseline model, we experimented with many hyperparameters including encoder-decoder embedded size, hidden size, batch size, and learning rates. We also experimented with different optimizers and experimented with scheduled and cyclic learning rates.

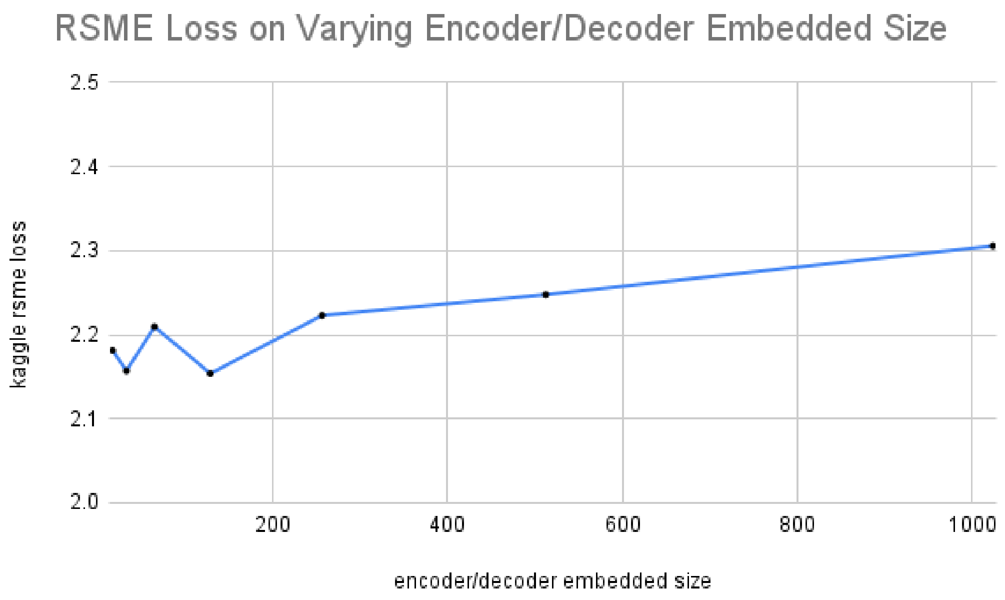
Batch Size:
We experimented on batch sizes of 2, 3, 4, 5, 6, and 8. We experimented with smaller numbers for better generalization. A batch size of 4 had the best results.

batch size	RMSE loss
2	2.36618
3	2.4785
4	2.30747
5	2.68707
6	2.48598
8	2.53557



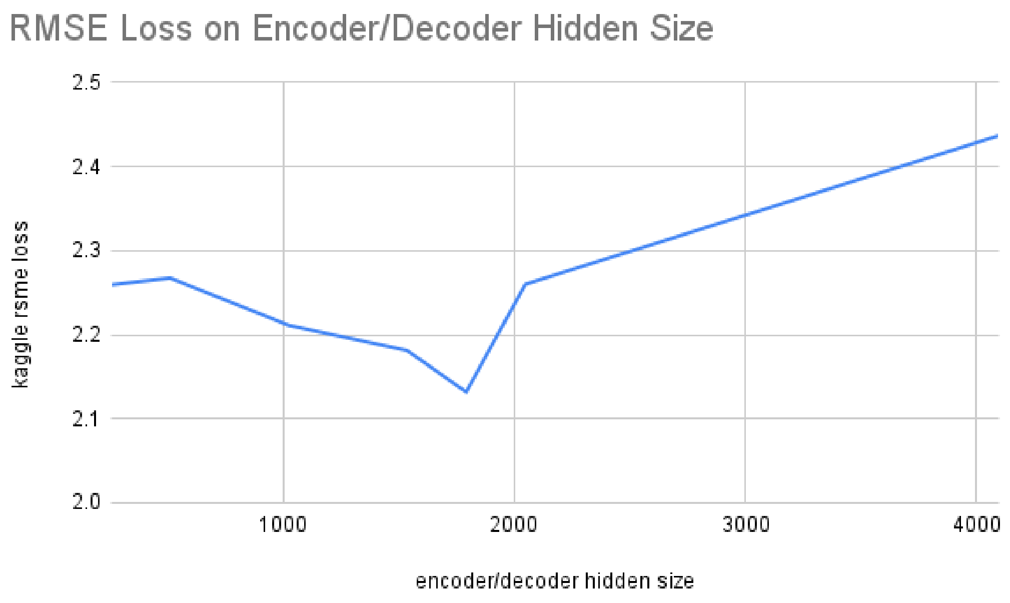
Embedded Size:
First, we wanted to see how changing only the decoder embedded size or encoder embedded size affected the performance. However, we soon realized that they give similar results, so we decided to modify both in our experiment. We experimented on embedded sizes of 16, 32, 64, 128, 256, 512, 1024. Embedded size 32 and 128 had comparable results. We used 32 because of speed.

embedded size	RMSE loss
16	2.18121
32	2.15698
64	2.20949
128	2.15379
256	2.22279
512	2.24767
1024	2.30562



Hidden Size:
Encoder and decoder hidden sizes has to be kept the same. We experimented on hidden sizes of 256, 512, 1024, 1536, 1792, 2048, and 4096. We designed these experiments with binary search. 1792 had the best results.

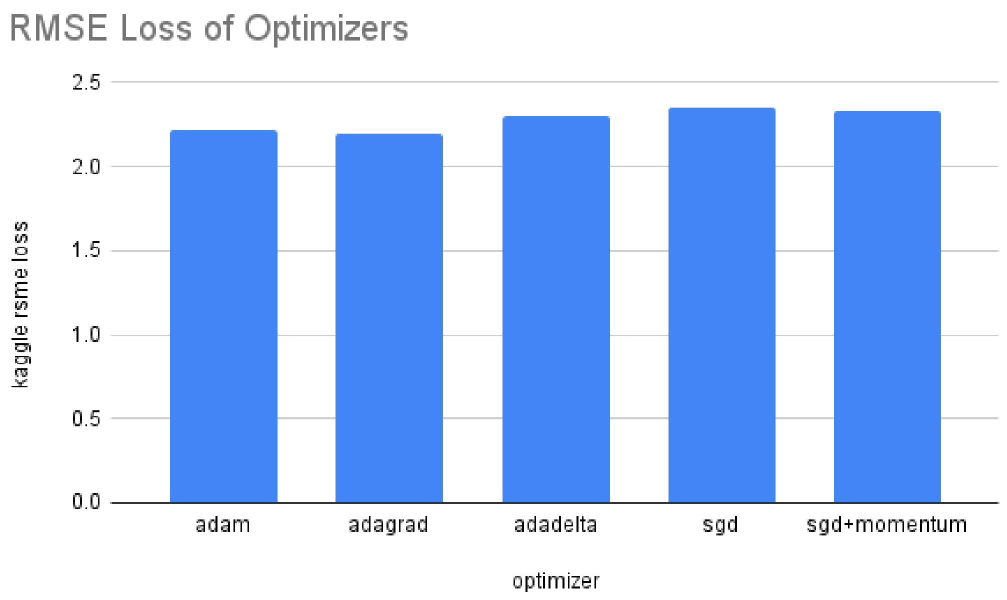
hidden size	RMSE loss
256	2.25942
512	2.26733
1024	2.21117
1536	2.18121
1792	2.13181
2048	2.26001
4096	2.43677



Optimizer:
We experimented with RMSProp, Adam, AdaGrad, Adadelta, SGD, and SGD+momentum. Most optimizers compared similarly, other than RMSProp (not pictured). Over the course of experimentation, we decided that Adam was consistently performing the best. We did not tune the optimizer hyperparameters since from Schmidt et al's paper [5], they found that model performance is similar whether the model uses the default optimizer hyperparameters or tunes the optimizer hyperparameters.

Note: RMSProp is not pictured since we experimented with RMSProp early on with different hyperparameters in this experiment. However, when we tested it against Adam and AdaGrad, it had comparatively higher RMSE loss.

optimizer	RMSE loss
Adam	2.21117
AdaGrad	2.19748
Adadelta	2.29839
SGD	2.34922
SGD+momentum	2.33313



Learning Rate:
From Luo et al.'s paper [2], they found that using a scheduled learning rate to manually tune the learning rate for each epoch resulted in better loss convergence. We also experimented with this, multiplying the learning rate by a gamma after each epoch. We experimented with a gamma of 0.5 every epoch, 0.75 every epoch, and 0.1 every other epoch. From Smith's research on cyclical learning rates for training neural networks [3], we learn that cycling between learning rates for each epoch may be beneficial than monotonically decreasing the learning rate. This allows the learning rates to be bound to reasonable learning rates. We extended our experiment to alternate 0.1 and 5 every epoch. Lastly, we experimented with manual single learning rate adjustments where the learning rate was only multiplied by gamma at a particular epoch. These experiments include gamma = 0.1 after 1 epoch, gamma = 0.01 after 1 epoch, and gamma = 0.1 after 4 epochs. We found that 0.5 every epoch gave us the best result.

Note: We performed the following experiments on many epochs (up to 12) and the table records the best RMSE loss.

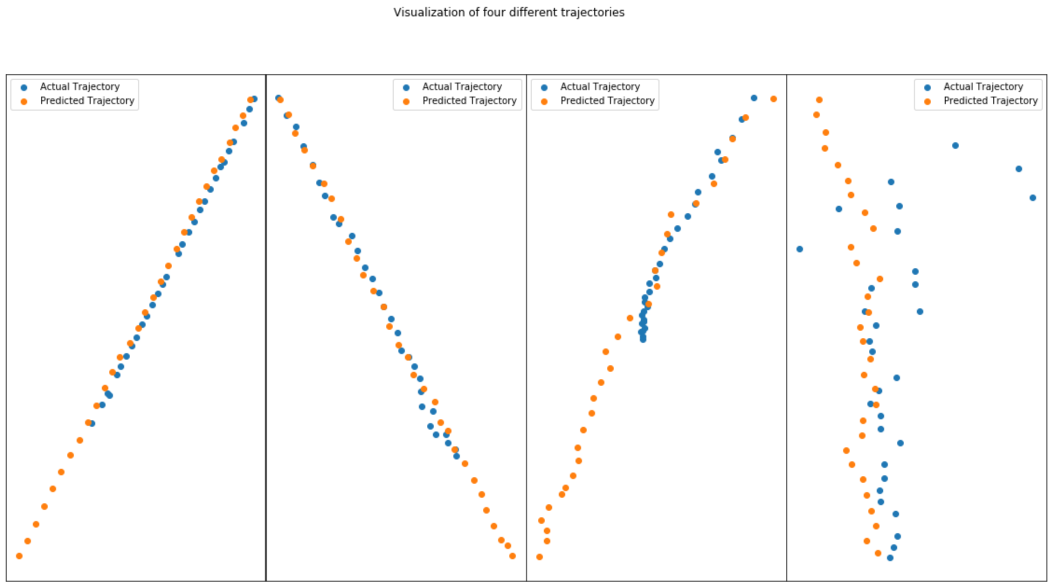
learning rate	RMSE loss
0.5 every epoch	2.02689
0.75 every epoch	2.14439
0.1 every other epoch	2.15933
(cyclic) 0.1 and 5	2.03496
0.1 after 1 epoch	2.09195
0.01 after 1 epoch	2.11562
0.1 after 4 epoch	2.14555

All of our models ran at roughly the same amount of time, 1 hour 20 minutes to 1 hour 40 minutes locally, in exception to the hidden size experiment where increasing the hidden size increased the time it took to complete an epoch. To increase the training speed, we increased the batch size. This was mainly done to see if our code was working before working with smaller batch sizes. A batch size of 100 ran in about 20 minutes. Since the training the model took a long time, we saved the model after each epoch.

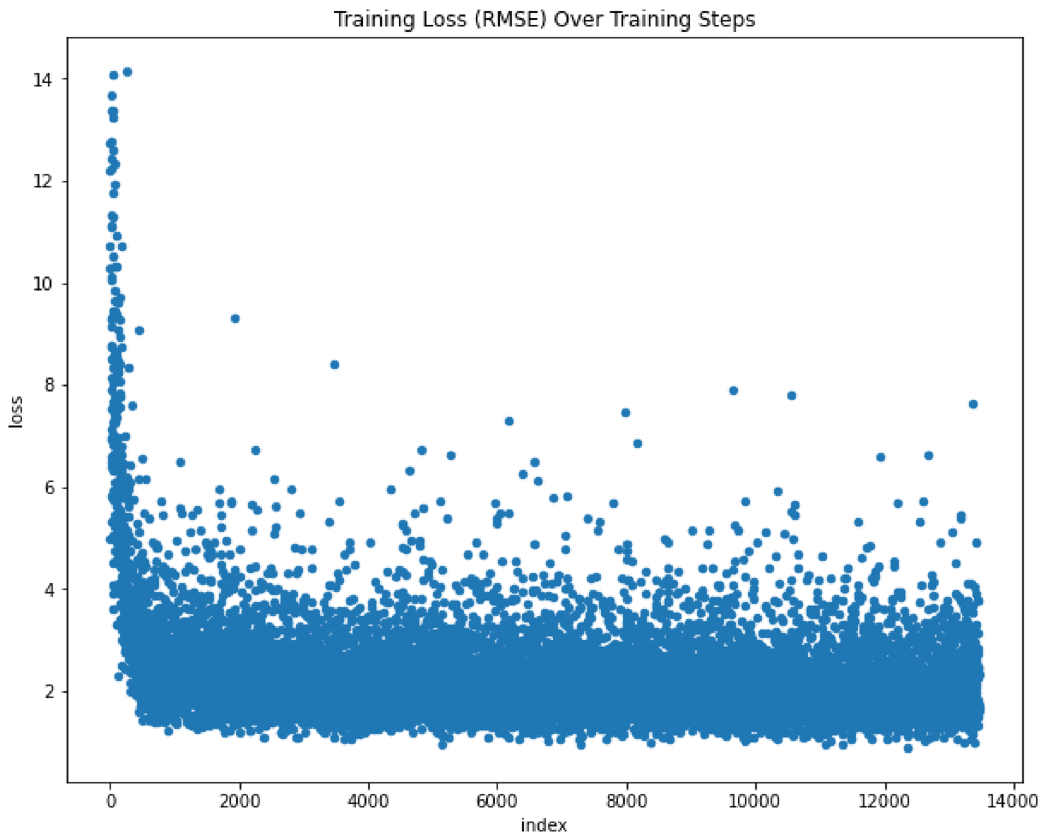
There are 9,646,240 parameters in the encoder model and 9,652,388 parameters in the decoder model.

5.2 Problem B

We finished the competition rank 3 on the public leaderboard with a test RMSE of 2.02689 and rank 4 on the private leaderboard with a test RMSE of 2.09605.



We noticed that when the actual trajectory is linear or in a straight line, our model better predicts the trajectory. However, for more non-linear curved lines, especially with abnormal driving paths with both left and right steering, our model does not predict as accurately.



From visualizing the training loss (RMSE) values over training steps, we see that the loss converges to ~2. Additionally, we see exponential decay, this indicates that the model is learning.

6 Discussion and Future Work

6.1 Problem A

We did not do feature engineering, but we normalized the x, y positions. Normalization is very important when it comes to deep learning as it affects model results severely when done incorrectly. Without normalization, our prediction results were skewed, giving us high test losses.

We found hyper-parameter tuning to be the most helpful in improving our score. Tuning epochs, batch sizes, optimizers, hidden and embedded sizes for the encoder and decoder, and learning rates improved our public score from around 3 to just over 2.

Our biggest bottleneck in this project would definitely have to be the time constraint and lack of resources. Our models were time consuming to run and took up a lot of resources which we didn't have as most of us were using DataHub to run our models. This in addition to the ten weeks that we had to work on the project limited us in the amount of experiments we could run to optimize hyperparameters and try new models.

Our advice for a deep learning beginner is to fully understand the data that models are built on and not underestimate the time commitment needed to train these models as it is time consuming to train models when there is a lot of data. Preprocessing is also very important when working with data. Lastly, start early, start often.

If we had more resources, we would like to explore other engineering tricks that others have found success using to bring down our loss even further and experiment with the transformer model which uses the attention mechanism. Also, we would like to implement other features of the data into our model such as lanes.

References

[1] Francesco Giuliari, Irtiza Hasan, Marco Cristani, & Fabio Galasso. (2020). Transformer Networks for Trajectory Forecasting.

[2] Liangchen Luo, Yuanhao Xiong, Yan Liu, & Xu Sun. (2019). Adaptive Gradient Methods with Dynamic Bound of Learning Rate.

[3] Leslie N. Smith. (2017). Cyclical Learning Rates for Training Neural Networks.

[4] Ming-Fang Chang, John Lambert, Patsorn Sangkloy, Jagjeet Singh, Slawomir Bak, Andrew Hartnett, De Wang, Peter Carr, Simon Lucey, Deva Ramanan, & James Hays. (2019). Argoverse: 3D Tracking and Forecasting with Rich Maps.

[5] Robin M. Schmidt, Frank Schneider, & Philipp Hennig. (2021). Descending through a Crowded Valley – Benchmarking Deep Learning Optimizers.