CSE 151B Project Final Report

Garrett T. Birch
Data Science Undergraduate
University of California - San Diego
La Jolla, 92093
`gbirch@ucsd.edu`

Amogh Patankar
Data Science Undergraduate
University of California - San Diego
La Jolla, 92093
`apatankar@ucsd.edu`

James Lu
Data Science Undergraduate
University of California - San Diego
La Jolla, 92093
`jqlu@ucsd.edu`

Matin Ghaffari
Data Science Undergraduate
University of California - San Diego
La Jolla, 92093
`mghaffari@ucsd.edu`

1 Task Description and Background

1.1 Task

The task for this project is to create a deep learning model that forecasts the positions of self-driving vehicles, known as Autonomous Vehicles (AV). In our project, we're given a five second observation of positions of an AV, and based on this data, our goal is to predict the next six seconds of positions of that particular AV. The task given to us is extremely prevalent given the state of self-driving and autonomous vehicles in the world right now. As tech and car companies attempt to make a fully self-driving future a reality, the task at hand is useful to do so. Specifically, the prediction of the positions of autonomous vehicles is important for two reasons- functionality, i.e. making the AV actually drive properly, as well as safety. Safety is a high priority- creating an environment where AVs can be integrated safely with human-driven cars is the end goal in the domain of self-driving technology. Our dataset is only positional data; while our model is not incredibly accurate, but it is a decent starting point for developing an algorithm.

1.2 Research Conducted

When researching the types of methods that have been already proposed in relevant literature, we found many notable insights that helped us accomplish this deep learning task. Specifically, we explored research papers pertaining to time-series predictions using neural network models and various pattern recognition methods in applicable contexts. At first, the research we conducted was aimed to understand how various models perform and their advantages and disadvantages in similar contexts. We knew from prior knowledge that a recurrent neural network architecture such as an LSTM would be well suited for predicting future sequences from input sequences, as it provides long and short-term memory states. The LSTM we implemented closely corresponded with a research paper we read regarding oil production forecasting. The literature from the Journal of Petroleum Science and Engineering believes that the "output of hidden layer with the present information is transferred to the hidden layer of the next time step as part of the input... this kind of loop can preserve the information of the previous step to keep the data dependency, which improves the ability of learning and abstracting from the sequential data."[2]. Although we knew this advantage about the LSTM, we gained insight about the potential drawbacks in a different research paper, noting the "shortcomings of naive LSTM implementations in capturing dependencies between multiple correlated sequences" [1]. As a result, we learned that we would likely not achieve adequate results using a simple LSTM and would need additional components. A component that we learned that could be implemented within an LSTM to address the issue of capturing dependencies between correlated sequences is a social pooling layer that allows LSTMs to share their hidden states with one another [1]. The idea is to have an encoder and decoder LSTM and allow them to share their hidden state with one another via "max-pooling over state vectors [hidden states] of nearby agents within a predefined distance range, but does not model social interaction with far-away agents" [4]. While we didn't have time to implement it, it was an idea that we would've liked to pursue further. Furthermore, our research introduced us to the idea of using a Gated Recurrent Unit (GRU), which we learned could outperform an LSTM. We knew that our data and sequences weren't particularly large, and we believed the lower number of parameters of the GRU model might provide faster and less volatile results [3].

1.3 Mathematical Modeling

In a mathematical sense, we can think of our model that we are developing to be written as a function f. The frequency at which we are given data is 10 Hz, meaning one second of position data is broken down further into data at ten time steps. As such, our input, which contains the positions at 50 time steps, can be defined as $x_1 \ldots x_{50}$, and $y_1 \ldots y_{50}$ as x and y positions. Using our model, we generate positional data as output at 60 different time steps, which we define as $x_1 \ldots x_{60}$, and $y_1 \ldots y_{60}$. Hence, our goal is to create a model f such that the ground truth predictions of the positions are equivalent or nearly equivalent to our model output. That is, $f(x_1 \ldots x_{50}, y_1 \ldots y_{50})$, should be nearly equivalent to $x_1 \ldots x_{60}$, and $y_1 \ldots y_{60}$.

From this mathematical abstraction, we believe that the model defined *does* have capabilities to solve other tasks beyond this project. We believe there are opportunities for our model to solve problems involving time-series data, sequential data, and/or forecasting data. In fact, we also believe that our final model (a MLP) also has the ability to solve other classification, prediction, or recognition tasks.

2 Exploratory Data Analysis

2.1 Data Descriptions

The dataset we worked with had trajectory data for six cities: Austin, Miami, Pittsburgh, Dearborn, Washington DC, and Palo Alto. Since the trajectories were sampled at a frequency of 10 Hz, our inputs were the initial five seconds and outputs were the remaining six seconds of an 11-second sequence. The training set has input and output data, while the test set has only input data for the six cities. Austin has a training size of 43,041 and a test size of 6,325 samples. Miami has a training size of 55,029 and a test size of 7,971 samples. Pittsburgh has a training size of 43,544 and a test size of 6,361 samples. Dearborn has a training size of 24,465 and a test size of 3,671 samples. Washington DC has a training size of 25,744 and a test size of 3,829 samples. Lastly, Palo Alto has a training size of 11,993 and a test size of 1,686 samples.

The inputs and outputs for each city are three-dimensional with the shape of $(N, 50, 2)$ for inputs and $(N, 60, 2)$ for outputs. Here, N represents the number of scenes, the 50 and 60 represent the 50 and 60 time steps for the five second input and the following six second output, and 2 is the dimension for the x and y values.

2.2 Exploratory Data Analysis

1) Random Batch EDA

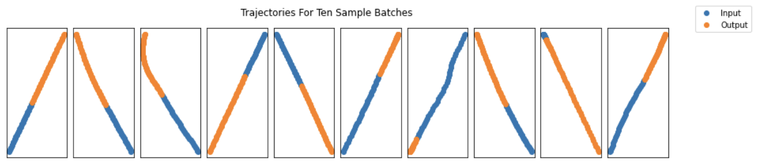


Figure 1: This data sample shows the trajectories for ten sample batches of our data. In these figures, the blue represents the input trajectories of those agents, and the orange represents the output trajectories. We visualized the batch trajectories in order to give us an idea how various portions of our data looked, and not just one random agent who may be an outlier. We see that a lot of these trajectories seem to follow a *fairly* linear path, and we leverage this knowledge later in our project.

2) Input and Output Position Distributions for All Cities

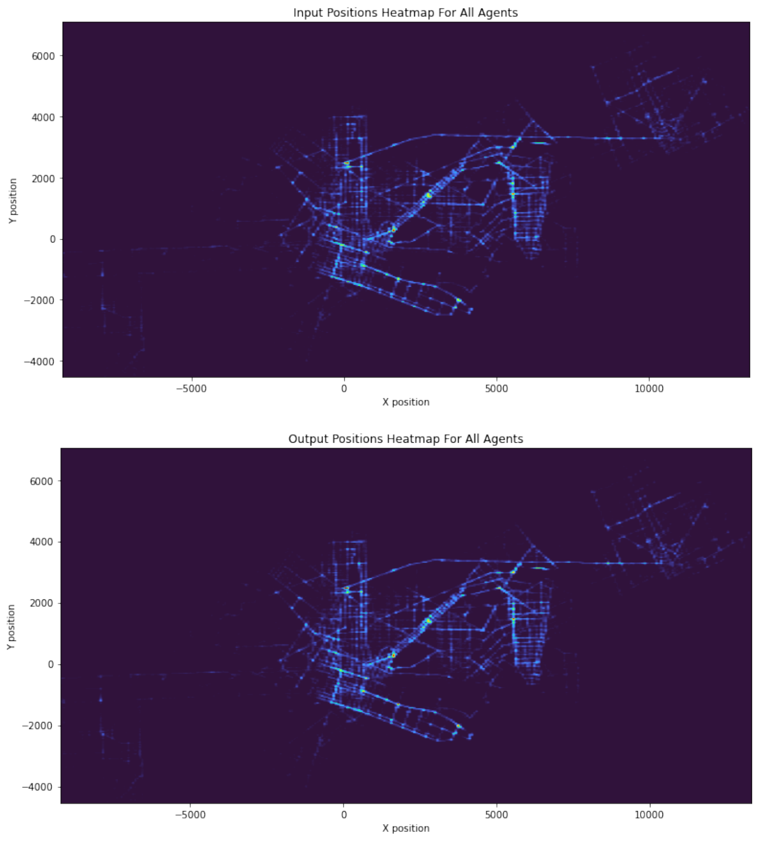


Figure 2: The top heatmap illustrates the distribution of input positions while the bottom heatmap illustrates the distribution of output positions for all agents. Overall we can see that both the distributions are very similar and resemble city streets since city streets have a similar grid-like view. So, we can infer that a good portion of our data follows a linear trend. The similarity of the input and output distribution can be supported through the rest of our exploratory data analysis, as we know that the agents' velocities and by extension displacements aren't extremely high. Therefore, we know that there are only minor "shifts" between the superimposed input and output heatmaps, not major differences. In addition, we can see that the most frequent positions as shown by the green, yellow, and red areas appear to be at intersections.

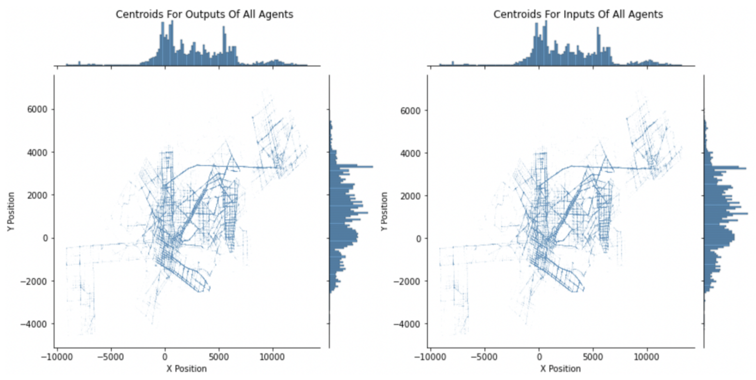
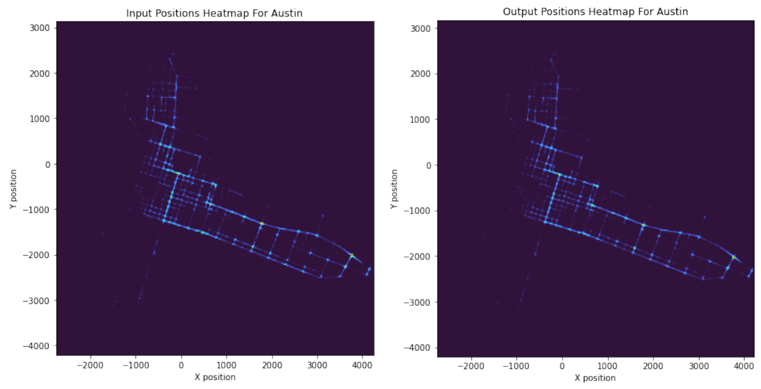
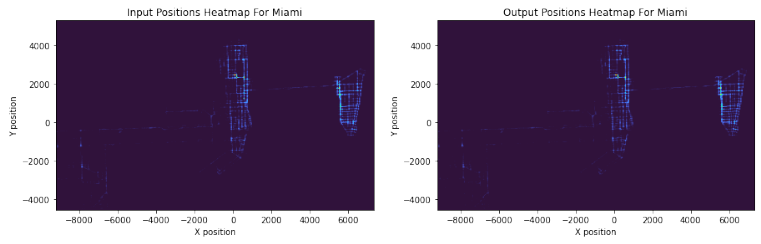


Figure 3: The plot on the left illustrates the centroids for inputs of all agents and the plot on the right illustrates the centroids for outputs of all agents. Due to the analysis of agent speed (Figure 8), we know that the speeds aren't fluctuating a lot. This explains our analysis of the centroids, as averaging the vertices across each the inputs and outputs helps us better understand the distribution at hand. These plots ultimately support the trends noted in the heatmaps in Figure 2, as we see the grid like patterns which resemble city streets. The histograms on the x and y axis also provide additional insight into how the distribution of x and y positions are multi-modal and does not fit a normal/Gaussian distribution. This lack of a Gaussian distribution indicates that it would be better to scale our data using a method such as min-max scaling rather than standard normalization.

3) Input and Output Position Distributions for Individual Cities





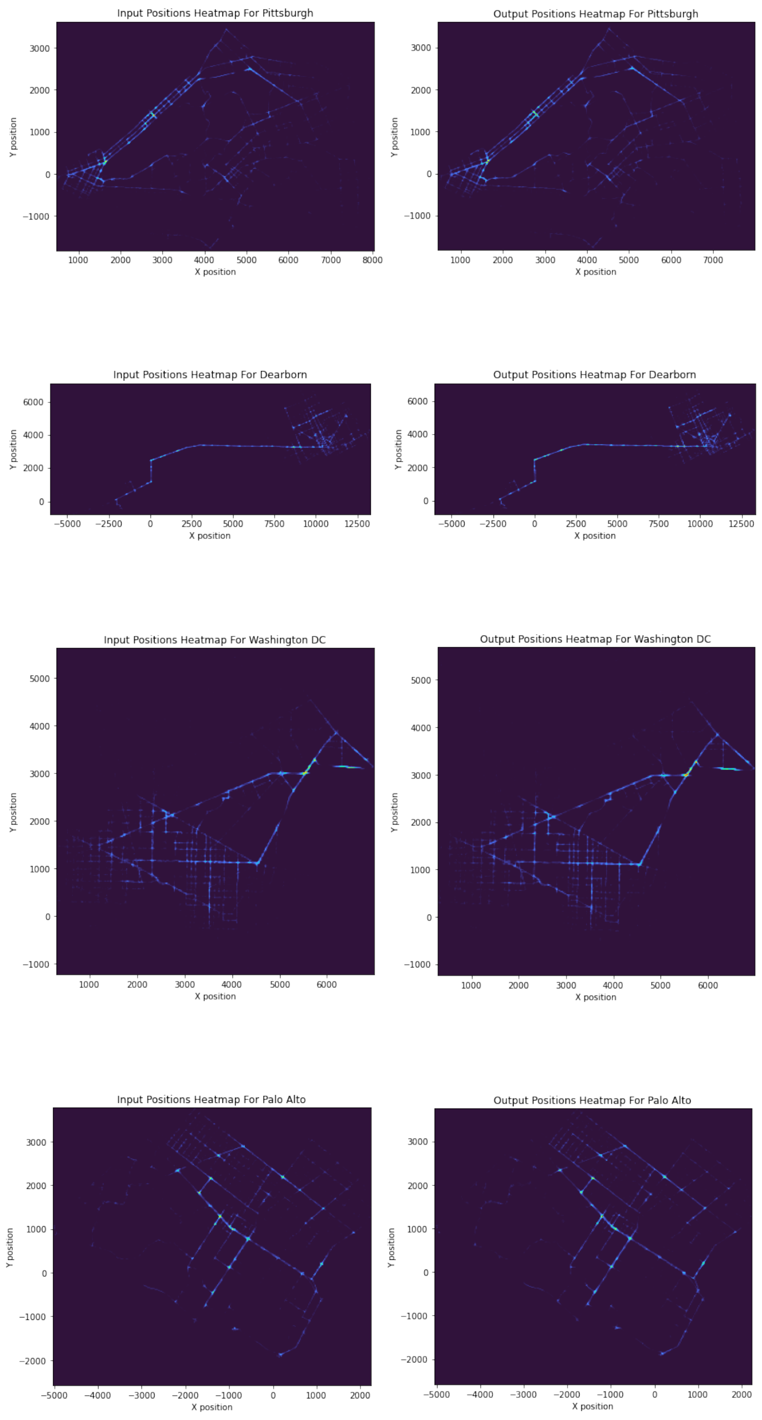


Figure 4: The heatmaps above illustrates the distribution of input and output positions for each city. The left column of heatmaps represents the input positions, while the right column represents the output positions of cities. These heatmaps reinforce the fact that the input and output distributions for each city are similar, also indicating that there seems to be a semi-linear trend in our data.

4) QQ Plots for X and Y coordinates

We decided to explore our data distribution via Quantile-Quantile plots to better understand the distribution of the data. More specifically we were interested in determining if the data follows a Gaussian distribution. We ultimately found that the distribution of the x and y positions of both the input and output data does not seem to follow a normal/Gaussian distribution. This can be seen by the blue curve, representing our distribution, deviating from the red line which represents the normal theoretical quantiles.

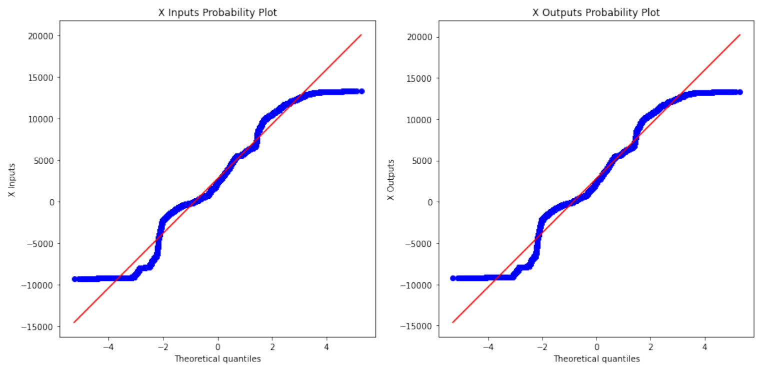


Figure 5: The QQ-plots illustrate the x-coordinate inputs on the left and output values on the right, and are superimposed upon a line is indicative of a normal distribution within the data. The QQ-plots allow us to understand if our two sets of data, x inputs and x outputs, come from the same distribution. Because the two graphs are very similar, we confirm that the distribution of our outputs are reasonably close (overall) to our inputs' distribution. Moreover, because our line (in blue) doesn't fall perfectly on the red line, we know that our x-coordinate data isn't normally distributed.

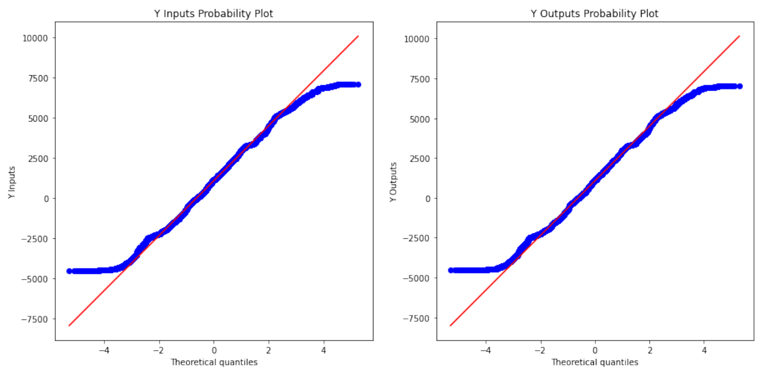


Figure 6: Furthermore, these QQ-plots illustrate the y-coordinate values superimposed upon a line y = x. The QQ-plots allow us to understand if our two sets of data, y inputs and y outputs, come from the same distribution. Because the two graphs are very similar, we can also confirm that the distribution of y-coordinate outputs are reasonably close (overall) to the y-coordinate inputs' distribution. Moreover, because our line (in blue) doesn't fall perfectly on the red line, we know that our y-coordinate data isn't normally distributed.

5) Distributions of Distance, Speed, and Velocity

These next figures work cohesively and indicate an idea that we suspect to be true due to the limitation of our data (having only five seconds of data). We don't expect high fluctuations in the speed and the distances traveled by the agents. That is, it is rare for agents to rapidly accelerate or decelerate in such a short span of time, which explains the lack of fluctuations in the speeds and velocities. The distances traveled by agents are restricted due to the restricted speed of the agents.

So, it makes sense that the following figures for distance and speed of our data both have similar shapes and distributions, which supports the velocity distribution (Figure 9) appearing as going in all directions at approximately the same frequency.

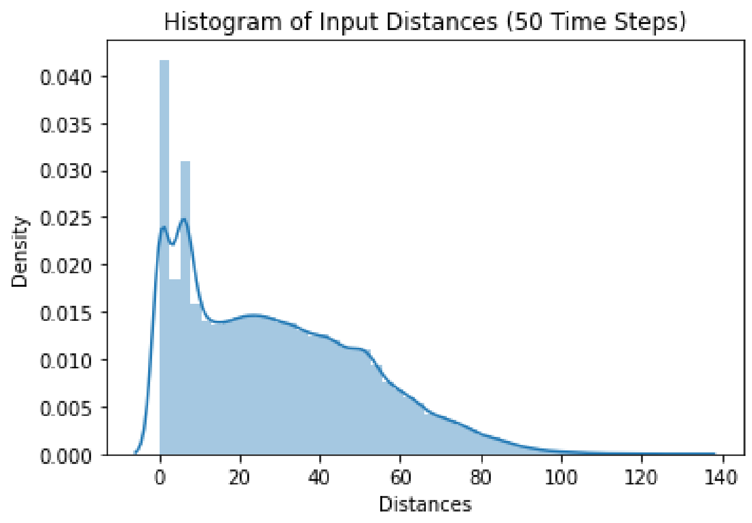


Figure 7: The figure above shows a histogram of the distances traveled by each agent, with the x-axis being the distance and y-axis being the density. We expected a histogram similar to this, because our data is so limited with respect to time. Because we only have five seconds (50-time steps) worth of position data, the distance traveled cannot exceed a particular limit. As shown in the graph, the majority of the agents do not travel relatively far from their starting position.

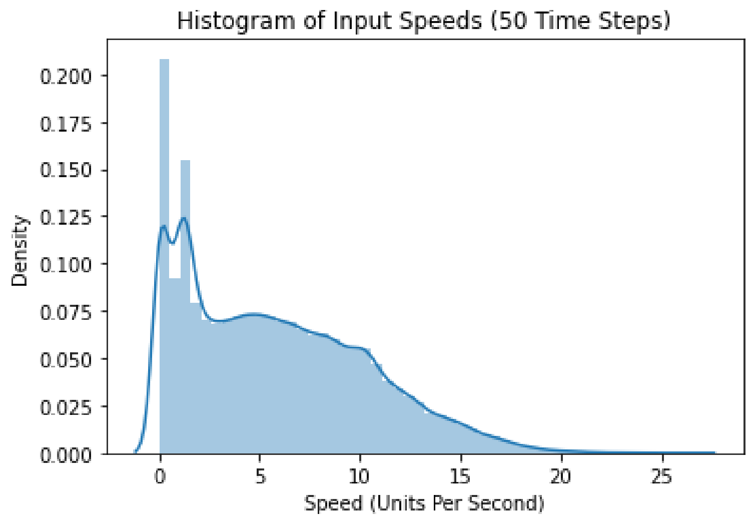


Figure 8: The figure above shows a histogram of the speed traveled by each agent, with the x-axis being the speed and y-axis being the density. We expected a histogram similar to this, because our data is so limited with respect to time and distance. Because we only have five seconds (50-time steps) worth of position data, the distance traveled cannot exceed a particular limit, and so, the speed also cannot be past a certain limit. As shown in the graph, the majority of the agents do not travel at a very high speed.

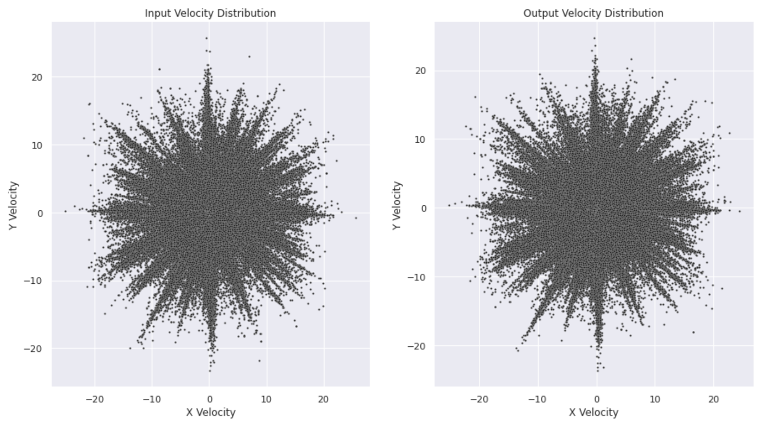


Figure 9: This plot shows the velocity distribution of inputs on the left hand side and velocity distribution of outputs on the right hand side. We can see that the distribution of velocity is at approximately the same frequency, indicating that using velocity as a model feature would have minimal impact on our model.

2.3 Data Preprocessing

We split our data in the following proportions:

City	Train (80%)	Validation (20%)	Test
Austin	34433	8608	6325
Miami	44023	11006	7971
Pittsburgh	34835	8709	6361
Dearborn	19572	4893	3671
Washington D.C.	20595	5149	3829
Palo Alto	9594	2399	1686

We didn't actually implement feature engineering, but we had two potential ideas in mind. The first idea was to create a velocity feature, which is simply the displacement between two time steps divided by one, as that is the unit of time. However, we decided against using that idea because we realized that velocity is derivative from displacement (in a mathematical and literal sense), and hence, it may be too similar to our feature. The other idea we had was data augmentation, in the sense that we insert new data in between two time steps. For instance, if at time step 1 we had x = 10, y = 300, and at time step 2 we had x = 30, y = 500, we create a new point at time step "1.5", with values x = 20, y = 400. We decided against this because we believed that it would make the training process much longer. Moreover, we were afraid to insert data points calculated as the average of the two time steps, as we didn't know how accurate that would be in certain cases.

With respect to normalization, we attempted to use min-max scaling to normalize our data, but it ended up producing worse results than a non-normalized model. We realized that normalization techniques do not eliminate outliers, and because of this property, our models didn't perform optimally. We tried using batch normalization and layer normalization, yet both gave worse results when implemented. However, we did transform our training data in an attempt to reduce learning time and improve effectiveness. In order to do this, we subtracted the initial coordinate from the other coordinates for every single agent in the dataset. This data transformation gave us a much smaller range of values, and gave us a measurement of distance that was relative to the origin. When predicting the output for each agent, we [obviously] needed to reverse this transformation. To do so, we simply added the first point of the sequence for that particular agent to all the other coordinates for that particular agent. In doing so, this data transformation greatly increased our model accuracy and training speed.

When looking at the city information provided in the dataset, we were able to exploit some of the information we gleaned. While doing exploratory data analysis, we noticed a trend when looking at input trajectories and the ground truth trajectories across various batches. We noticed that a significant number of the trajectories in our dataset contained a linear pattern, or some form of linearity. We leveraged this information to improve our architecture and model (through hyperparameter tuning), which will be discussed later. Using the data was key for us to improve our MSE score, which was an indicator of how well our model performed.

3 Machine Learning Model

3.1 Feature and Model Selection

The features we chose for our model were [unsurprisingly] the x and y positions, as these were the only features we were given in our data. We chose to use Linear Regression as our base model using scikit-learn. The loss function we used was mean squared error, otherwise known as squared L2 norm.

3.2 Deep Learning Pipeline

Similar to our machine learning model, the features we chose for our model were the x and y positions, as these were the only features we were given in our data. Surprisingly, the initial model architecture we chose was an LSTM based RNN. The intent behind this decision was rooted in the strengths of a LSTM architecture. We knew that LSTM architectures are useful specifically for time-series and sequential data, and we believed we could leverage this capability and apply it to our data. We also experimented with an architecture known as gated recurrent unit (GRU), due to its similarity with an LSTM, but smaller complexity. The GRU architecture doesn't contain a forget or an output gate, thus containing fewer parameters. We also experimented and saw the best results with a Multi Layer Perceptron (MLP). The loss function we used was the mean squared error, otherwise known as squared L2 norm.

3.3 Complete Model Summary

1) Two Layer RNN with LSTM

- Performance Summary:
  This model did not perform well at all; however some of our issues with this model originated from an incorrect implementation of the architecture. After fixing the architecture, we found that the training was taking extremely long and the losses were very large.
- Architecture:
  This model can be defined as an RNN with two LSTM layers. Each LSTM layer had 128 hidden nodes and had a dropout percentage of 0.2. The model passes in an input sequence of 50 to the LSTM and takes the last hidden state to make the first prediction. It then uses input positions and previous predictions' hidden states to generate a sequence of 60 output predictions. We passed in an input shape of (batch size, 50, 2) and outputted a size of (batch size, 60, 2).
- Parameters:
  This model had two parameters that we experimented with: hidden state size and dropout. We struggled with tuning these parameters because of the long training times and high loss, and after numerous experiments we decided that this model may not be the best choice for this problem.

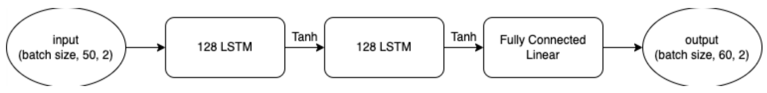


Figure 10: LSTM Architecture

2) Two Layer RNN with GRU

- Performance Summary:
  This model performed slightly better than the LSTM model, however the losses were still extremely high compared to the other submissions on the leaderboard.
- Architecture:
  This model had very similar architecture to the previous LSTM model, and the only difference was that we used GRU layers instead of LSTM layers. We passed in an input tensor with the shape (batch size, 50, 2) and output a tensor with the shape (batch size, 60, 2).
- Parameters:
  This model had two parameters that we experimented with: hidden layer size and dropout. We had the same issues that we had with the LSTM model, long training times and high loss. After numerous experiments we decided that this model also not the best choice for our problem, as it seemed that it was performing similarly to the LSTM model.

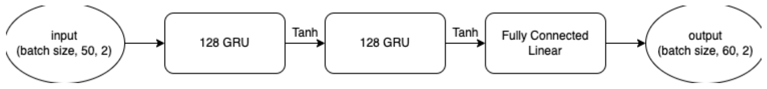


Figure 11: GRU Architecture

3) AutoEncoder MLP

- Performance Summary:
  This model was a breakthrough model for our group, it performed far better than our previous models. The training times were much faster and the losses were significantly lower as well. This model had an MSE score of around ~ 184, which was a massive improvement from the $\sim 80,000$ loss in our previous models.
- Architecture: This model followed an autoencoder design structure where we compress the input then decompress it to generate final predictions. We first flattened our input layer from a shape of (batch size, 50, 2) to (batch size, 100). We then used two linear layers with ReLU activation and hidden sizes 64 and 32 to compress the input data, and two linear layers with ReLU activation and hidden sizes 32 and 64 to decompress the data. We then transformed the flattened prediction sequence of shape (batch size, 120) to the correct prediction shape of (batch size, 60, 2).
- Parameters:
  This model only had one tunable parameter- the hidden size for each linear layer. We used linear layers of 32 and 64 to compress and decompress the data.

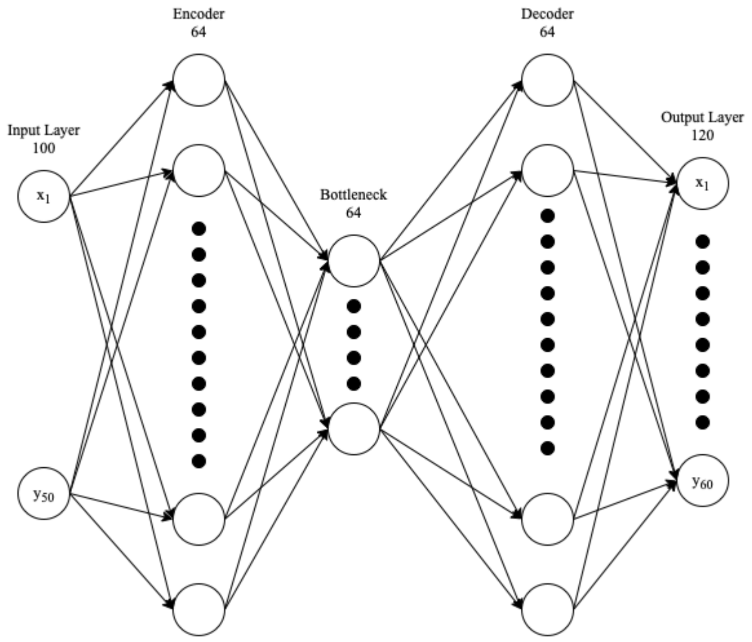


Figure 12: AutoEncoder MLP Architecture

4) Scikit-learn Linear Regression

- Performance Summary:
  This model was yet another breakthrough model for our group, it performed even better than the autoencoder in terms of train time and MSE loss.
- Architecture:
  This model used the linear regression model provided by sklearn.
- Parameters:
  There weren't any tunable parameters for this model.

5) PyTorch Linear Regression

- Performance Summary:
  This model was attempt to re-make the sklearn linear regression so that we may tune some parameters. We found that this model performed worse than the sklearn linear regression, and was closer to the previous autoencoder model.
- Architecture:
  We simply used a single linear layer with no activation function to recreate linear regression.
- Parameters:
  There were no tunable parameters for this model.

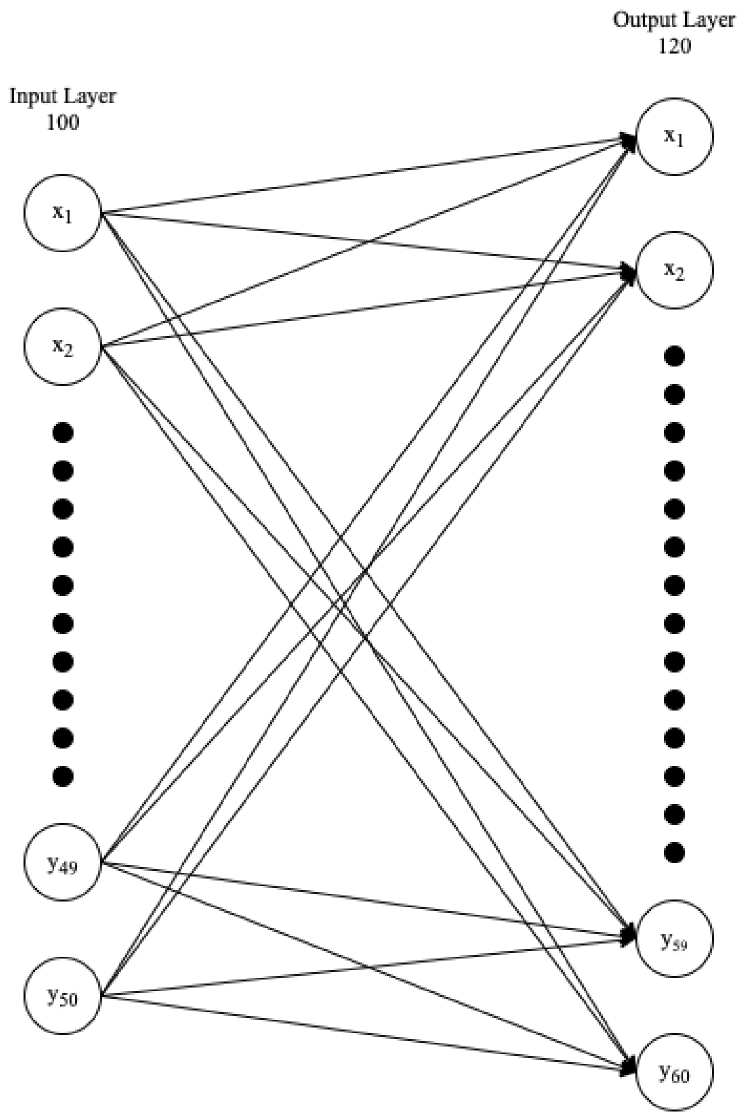


Figure 13: PyTorch Linear Regression Architecture

6) Dense MLP

- Performance Summary:
  This model was our best performing and final model. We wanted a model that matched the performance of linear regression while also having some sort of non-linearity for sequences the curves in our data. This model scored around ~ 18 MSE after hyperparameter tuning and greatly outperformed our previous models.
- Architecture:
  This model was composed of one linear input and output layer, and four hidden linear layers with ELU activation functions. Our hidden sizes were 1024 and 512 respectively.
- Parameters:
  The only parameter that we really had to tune was our hidden sizes. We experimented with a number of hidden sizes ranging from 32 to 1024, however we found that the higher hidden sizes had lower training and validation loss.

Throughout the course of our project, we experimented with a few different regularization techniques. Namely, we used early stopping, and dropout regularization while implementing our LSTM and GRU-based models/architectures. The reason for using early stopping was because our validation curve would begin to curve upwards, or rise, as we progressed through our epochs. Hence, we used early stopping to stop our model to stop overfitting. We used dropout regularization to attempt to improve our model performance by excluding a particular percentage of connections. But, neither regularization technique that we implemented worked well with those particular architectures. When we moved on to our linear regression and MLP models, we used L2/weight decay, as well as dropout regularization. We experimented with dropout however it didn't provide any benefits in doing so. Weight decay (i.e. L2) wasn't particularly helpful either, as our weights weren't large enough to begin with.

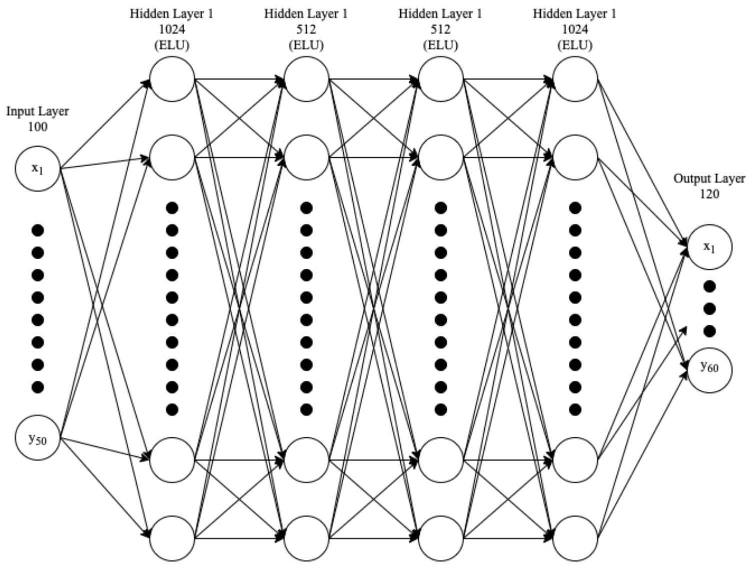


Figure 14: Dense MLP Architecture

4 Experiment Design and Results

4.1 Experiment Design

For our training and testing of our models, we used a combination of a few different computational platforms. The majority of our group used the UCSD datahub server (DSMLP), and had access to GPUs there (1080ti). One of our team member's, Garrett Birch, had access to and utilized a NVIDIA 3080ti for computational resources. If datahub wasn't accessible, the other group members utilized their local machines.

4.2 Hyperparameter Tuning

Optimizer: The optimizer that we started our project with was stochastic gradient descent (SGD), but quickly realized that it was a little unstable for our dataset and model architecture. We experimented with AdaGrad, but eventually finalized the Adam optimizer in PyTorch.

Learning Rate: With respect to learning rate, we used log scale for hyperparameter tuning, as it searches a larger space in a smaller period of time. We began with a learning rate of 0.001, but eventually settled at 0.0001 for our final model. We did this because we noticed that a large learning rate caused our model to not converge, or kept "bouncing around" as seen in the loss plots.

Learning Rate Decay: To combat the problem mentioned above, we also utilized learning rate decay, specifically, the StepLR learning rate decay. We did a similar tuning process for the gamma value, and the step size, ending with 0.9 and 25, respectively. We used learning rate decay to assist our model to converge to a minimum, and avoid the fluctuations that could occur in doing so.

Activation Functions: We tried quite a few different activation functions- tanh, sigmoid, ReLU, and other variants. We quickly realized that while tanh and sigmoid worked for LSTM, they were poor for our final MLP model. This is due to the vanishing gradients problem, which steered us to use ReLU. While ReLU does have an exploding gradient problem in some cases, our weights in this challenge weren't nearly large enough to encounter that which was also the reason why we didn't implement L2 regularization. We experimented with Leaky ReLU and its corresponding hyperparameter (negative slope), and found it worked better than ReLU. Finally, we actually tried ELU, an element-wise function, and it further lowered our loss.

4.3 Multistep Prediction Task

We went through multiple methods of generating our multistep predictions. In the beginning with our more complex RNNs we would train on the input sequence (50 coordinates), and get the last hidden state from the RNN layer (LSTM / GRU) which in essence was used for our prediction. We would then append our new predictions onto the input and pass in the last 50 coordinates from the new appended list i.e. 50 input, 0 predictions $\rightarrow$ 49 input, 1 prediction $\rightarrow$ 48 input, 2 predictions, etc. until we had 60 predictions. For our later MLP and regression models we first flattened our input layer from a shape of (batch size, 50, 2) to (batch size, 100). We then passed this new sequence into our model and finally transformed the flattened prediction sequence of shape (batch size, 120) to the correct prediction shape of (batch size, 60, 2).

As mentioned in a prior section, we noticed that when analyzing the batch visualizations, we noticed that a majority of them were linear, or had linearity. In building our model, we utilized this information and naturally, our best scoring models were a linear regression model and a multi-layer perceptron with multiple linear layers within.

We initially trained our model for 150 epochs, and noticed that we weren't getting conclusive results until we increased our epochs. Hence, we tuned the number of epochs between 150 and 500, but noticed that 200 was the optimal number. With regards to batch size, we once again tuned that on a log scale with base 2 numbers. We used batch sizes of 32, 64, 128, and 256, and finalized a batch size of 128. With our final tuned model, each epoch took around 5 seconds, and the entire model took 45-60 minutes overall, as it was a fairly trivial, non-complex model.

4.4 Comparison of Models

Architecture	MSE	Num of Params	Time Per Epoch	Overall Training Time
LSTM	$\approx 81,000$	1057794	10-15 seconds	100-110 minutes
GRU	$\approx 80,000$	921256	10-15 seconds	100-110 minutes
Sklearn LR	≈ 21	N/A	N/A	3-5 minutes
Autoencoder	≈ 184	480128	5-8 seconds	60-75 minutes
Pytorch LR	≈ 102	12120	3-7 seconds	55-70 minutes
Dense MLP	≈ 18	1553712	2-5 seconds	45-60 minutes

4.5 Final Model Loss Graphs

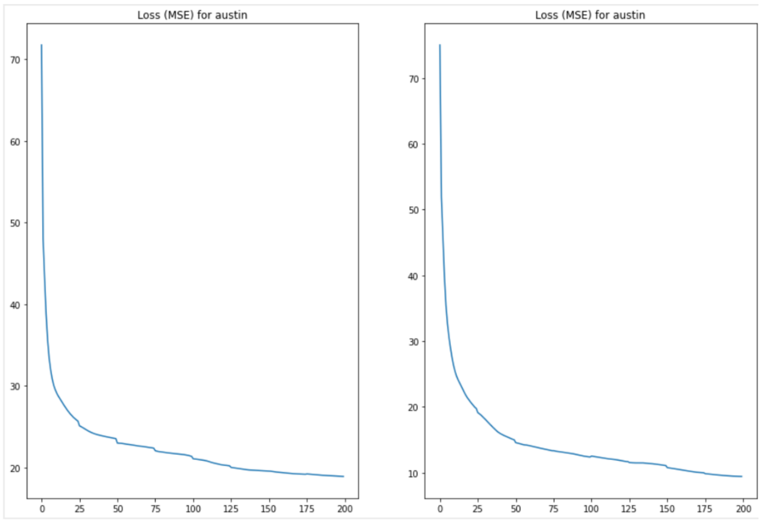


Figure 15: Training and Validation Loss for Austin

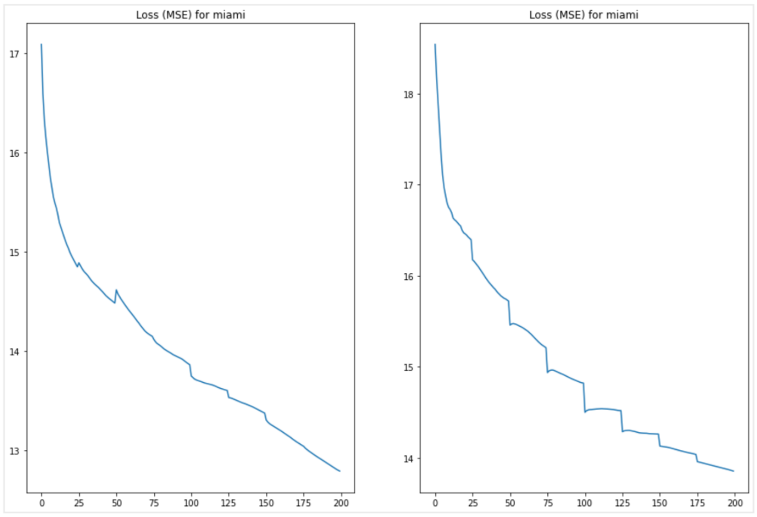


Figure 16: Training and Validation Loss for Miami

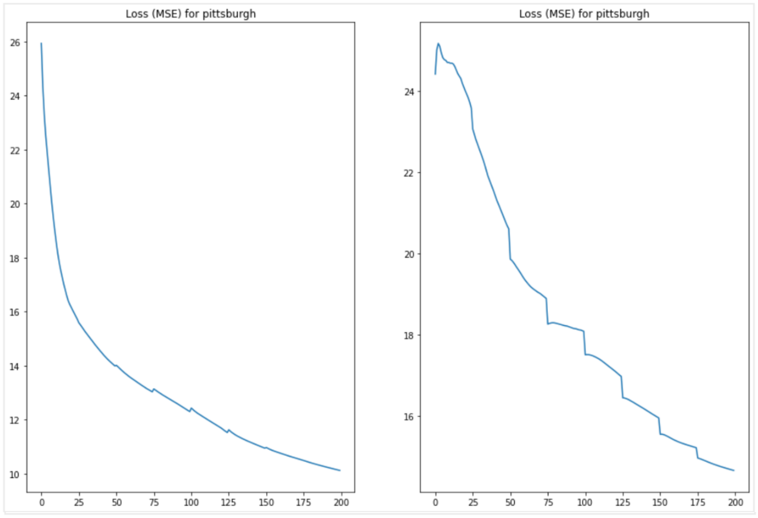


Figure 17: Training and Validation Loss for Pittsburgh

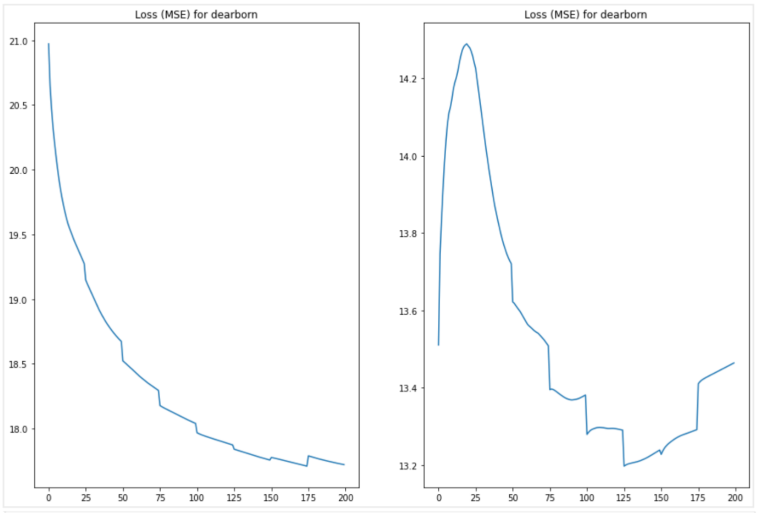


Figure 18: Training and Validation Loss for Dearborn

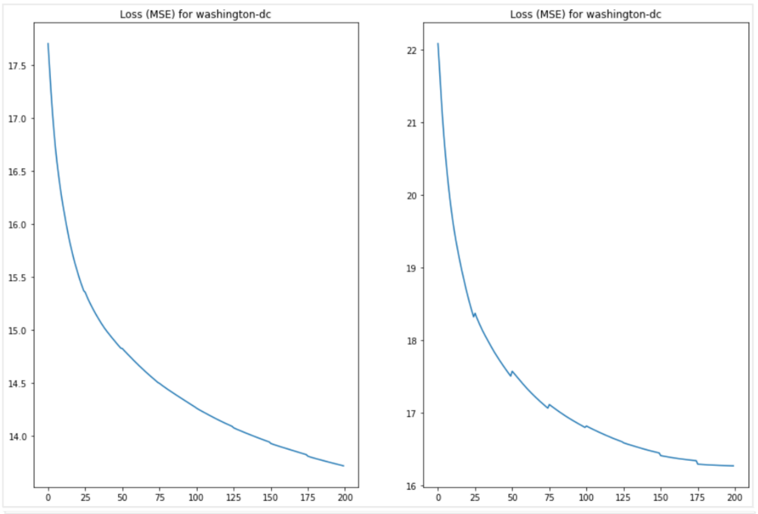


Figure 19: Training and Validation Loss for Washington D.C.

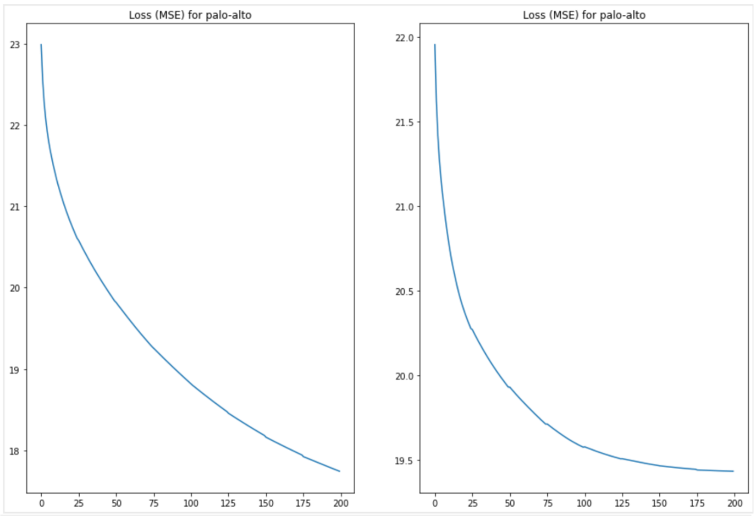


Figure 20: Training and Validation Loss for Palo Alto

4.6 Final Model Predictions vs. Ground Truth

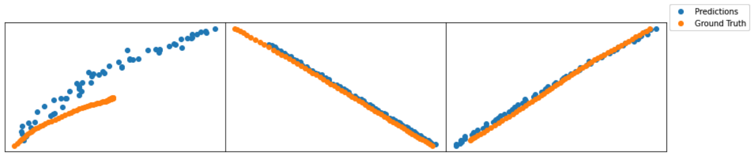
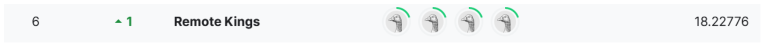


Figure 21: This figure visualizes our model's performance by plotting the predicted values (blue) along with the ground truth values (orange) for a random training sample. It is shown how our model does a relatively good job capturing linear trends but struggles to accurately capture when agents are making turns or slowing down rapidly.

4.7 Final Ranking



6 ▲1 **Remote Kings** 18.22776

Figure 22: Our team (Remote Kings) ultimately finished in 5th place on Kaggle (appears as 6th place on leaderboard due to erroneous first place submission), and achieved a private MSE score of 18.22102 and a public MSE score of 18.12318.

5 Discussion and Future Work

We didn't use a feature engineering strategy, but we believe the data augmentation strategy, if implemented correctly, could have positive results. However, that may also overfit the model, hence we cannot make a concrete statement one way or another. However, we will say that our data transformation strategy (subtracting the initial position of each agent from each point) was highly effective, as it sped up our model and improved its performance too.

Our data visualization, and parameter and hyperparameter tuning was essential to our success on the leaderboard rankings. Because of our expertise in data science (all of us are data science majors), we knew that analyzing our data and understanding it thoroughly would help us. In the end, the realization that nearly all our data was linear or followed a semi-linear trajectory allowed us to change our model, and score well on the rankings. Similarly, tuning our hyperparameters and parameters with intent, and not in a random manner allowed us to debug easily, and also make changes that were appropriate for situations at hand. Hence, using a combination of those two things, we were able to improve our ranking.

Our biggest bottleneck was definitely time and lack of experience, specifically with regards to our architecture and feature engineering. We spent an excessive amount of time implementing and unsuccessfully debugging our LSTM and GRU based models, which didn't allow us to experiment with other models as freely. Moreover, our time constraint in general didn't allow us to go back and attempt to resolve our LSTM issues; had we done that, I believe with our knowledge of the data and intentional hyperparameter tuning we could've gotten a score of below 15, which would've placed us first on the leaderboard. Only one of us (Amogh) had done a complete machine learning or deep learning project, and even that was developed in Tensorflow. Hence, our lack of experience with the different tools and architectures definitely showed.

If we had to advise a deep learning beginner in designing deep learning models for a similar task, we would give a few pieces of advice. We think that understanding the data you're given and finding patterns or trends can be valuable to creating a fairly optimal deep learning model. In addition, definitely start with a simple model first (unlike what we did), and build on top of that. In doing so, you will definitely learn lessons along the way that you can apply to your more complex model.

If we had more resources, we would've liked to implement the LSTM and/or GRU architecture properly. Another thing that would've been fun is if we had more features in our data, and could perform unique feature engineering. Finally, I think autoregressive models is definitely something that we overlooked, and it probably would have been fun to attempt to use.

References

[1] Kartik Patath et al. "Motion Forecasting for Autonomous Vehicles using Argoverse Dataset". In: (2022). URL: `https://sapan-ostic.github.io/Sapan_Agrawal_files/Motion_Forecasting_for_Autonomous_Vehicles_using_Argoverse_Dataset.pdf`.

[2] Xuanyi Song et al. "Time-series well performance prediction based on Long Short-Term Memory (LSTM) neural network model". In: *Journal of Petroleum Science and Engineering* 186 (2020), p. 106682. ISSN: 0920-4105. DOI: `https://doi.org/10.1016/j.petrol.2019.106682`. URL: `https://www.sciencedirect.com/science/article/pii/S0920410519311039`.

[3] Peter T Yamak, Li Yujian, and Pius K Gadosey. "A comparison between ARIMA, LSTM, and GRU for time series forecasting". In: (2019), pp. 49–55. URL: `https://dl.acm.org/doi/pdf/10.1145/3377713.3377722`.

[4] Tianyang Zhao et al. "Multi-agent tensor fusion for contextual trajectory prediction". In: (2019), pp. 12126–12134. URL: `https://arxiv.org/pdf/1904.04776.pdf`.