# CSE 151B Final Project Report

**Brandon Ho**
b1ho@ucsd.edu

**Neelay Joglekar**
njogleka@ucsd.edu

## Abstract

In order to make autonomous vehicles safer and more intelligent, we need to find better solutions to the various subtasks that compose autonomous driving. One important subtask is vehicle trajectory prediction. In the following report, we analyze vehicle trajectory data and propose a deep learning solution to this subtask. We construct and train LSTM-based neural networks that use velocity and local position data, instead of global position data, to solve the trajectory prediction task. We also describe our overall feature engineering and model desgin process, along with the obstacles we faced. Our final model was ranked #10 (team X Æ A-Xii) in the class kaggle competition. Our code base can be found in our GitHub Repo https://github.com/brandonho667/151B-WinningProject.

## 1 Task Description and Background

In the following section, we discuss the trajectory prediction task in detail and explain why it is important for the development of autonomous vehicles.

### 1.1 Problem A

In its most basic form, our task is to take a vehicle trajectory as input and predict the vehicle's future trajectory. There are many factors that influence a vehicle's trajectory, such as traffic lights, pedestrians, speed limit changes, weather, and many others. In addition, each driver has different driving styles, so two drivers in the same situation may still follow significantly different trajectories. Successfully performing trajectory prediction on real-world data can inform an autonomous vehicle about how a vehicle should move when encountering common external conditions. In addition, an autonomous vehicle can use this solution to predict the trajectory of neighboring vehicles, which can help with mitigating collisions and gathering information about a vehicle's surroundings.

### 1.2 Problem B

As a baseline to our project, we researched the common models used to predict trajectories. One such paper compared three different sequential models on predicting position and velocity of vehicles: Long Short Term Memory (LSTM), Gated Recurrent Units (GRU), and Stacked Autoencoders (SAEs) [1]. LSTMs and GRUs are similar in function as a memory based RNN while SAEs are typically used in data compression/reconstruction. Jiang et al. used the NGSIM I-80 dataset which contains trajectory data for 6000 California vehicles on multi-lane freeways. The main difference from our Argoverse dataset is that the NGSIM I-80 dataset also includes velocity data derived from camera data. As such, their velocity data was noisy and required them to apply a Savitzky-Golay filter to account for the noise. In our own data analysis, we saw that velocities calculated from the position of the vehicle in the Argoverse dataset was less erratic across trajectories. We therefore did not apply any smoothing filters to our point-to-point differentials. After training each model for 600 epochs, Jiang et al. concluded that the LSTM model functioned the best out of all methods. During the training of the LSTM, they also note the impact of using dropout as a conventional method to training

Preprint. Under review.

Table 1: Dataset Sizes

| | austin | miami | pittsburgh | dearborn | washington-dc | palo-alto | total |
|---|---|---|---|---|---|---|---|
| raw training | 43041 | 55029 | 43544 | 24465 | 25744 | 11993 | 203816 |
| training split | 34432 | 44023 | 34835 | 19572 | 20595 | 9594 | 163051 |
| validation split | 8609 | 11006 | 8709 | 4893 | 5149 | 2399 | 40765 |
| testing | 6325 | 7971 | 6361 | 3671 | 3829 | 1686 | 29843 |

RNNs. As such, our design of a multi-layered LSTM with dropout as our first iteration baseplate model was formed.

Liu et al. review a body of research on vehicle trajectory prediction with deep learning [2]. In particular for feature encoding, they explain several derived methods from other deep learning problems. From its success in natural language processing, transformers have been applied to feature extraction in trajectory prediction. From object detection, convolutional feature encoders have also been applied. Graph convolutional modules have also been applied to aggregate features across graphs generated from road maps. We have attempted to apply covolutional encoding with our Conv2Seq model and part of multi-head attention found in transformers via our LSTM Encoder Decoder model.

### 1.3 Problem C

In mathematical terms, a trajectory predictor takes an array of 50 2D positions, $x = [x_1, x_2, \ldots, x_{50}]$ where $x_i \in \mathbb{R}^2$, and outputs an array of 60 2D positions, $y = [y_1, y_2, \ldots, y_{60}]$ where $y_i \in \mathbb{R}^2$. The timestamps at which positions are recorded are evenly spaced by a timestep $\Delta t$, so if $x_i$ is recorded at time $t$ then $x_{i+1}$ must be recorded at time $t + \Delta t$ (the same goes for $y$). For our specific task, $\Delta t = 0.1$ seconds. This is a regression task, and the criterion for comparing our model's trajectory prediction $\hat{y}$ with the ground truth $y$ is mean squared error.

Although it isn't specified by the task, we define the following variables that can be derived from $x$ and $y$

- Velocity array $v_x$, where $v_{x_i} = \frac{x_{i+1} - x_i}{\Delta t}$. $v_y$ can be similarly generated.
- Local position arrays $\overline{x}$ and $\overline{y}$, where $\overline{x}_i = x_i - x_0$ and $\overline{y}_i = y_i - x_0$.
- Acceleration array $a_x$, where $a_{x_i} = \frac{v_{x_{i+1}} - v_{x_i}}{\Delta t}$. $a_y$ can be similarly generated.
- Path length $\|x\| = \sum_{i=1}^{49}(x_{i+1} - x_i)$. $\|y\|$ can be similarly generated.

Although this task involves specifically vehicle trajectory data, the mathematical formulation fits the trajectory of any object. As a result, a model that solves this task should be able to solve any other trajectory task. For example, the movement of other bodies, such as humans [3] and animals, can be similarly represented as 2D trajectories. In addition, we can represent movement in 3D space, such as the movement of planes, as 3D trajectories, which can also possibly be solved by such models.

## 2 Exploratory Data Analysis

In the following section, we document how we analyzed the given vehicle trajectory data and what patterns we found.

### 2.1 Problem A

The sizes of the given datasets are displayed in table 1 (both before and after we split training and validation). Each data point consists of a $50 \times 2$ input position vector sequence and a $60 \times 2$ output position vector sequence, i.e. the $x$ and $y$ values as defined above. These data points are sampled from real vehicle trajectories from 6 cities: Austin, Miami, Pittsburgh, Dearborn, Washington D.C., and Palo Alto. Examples of such paths can be found in Figures 4 and 7.

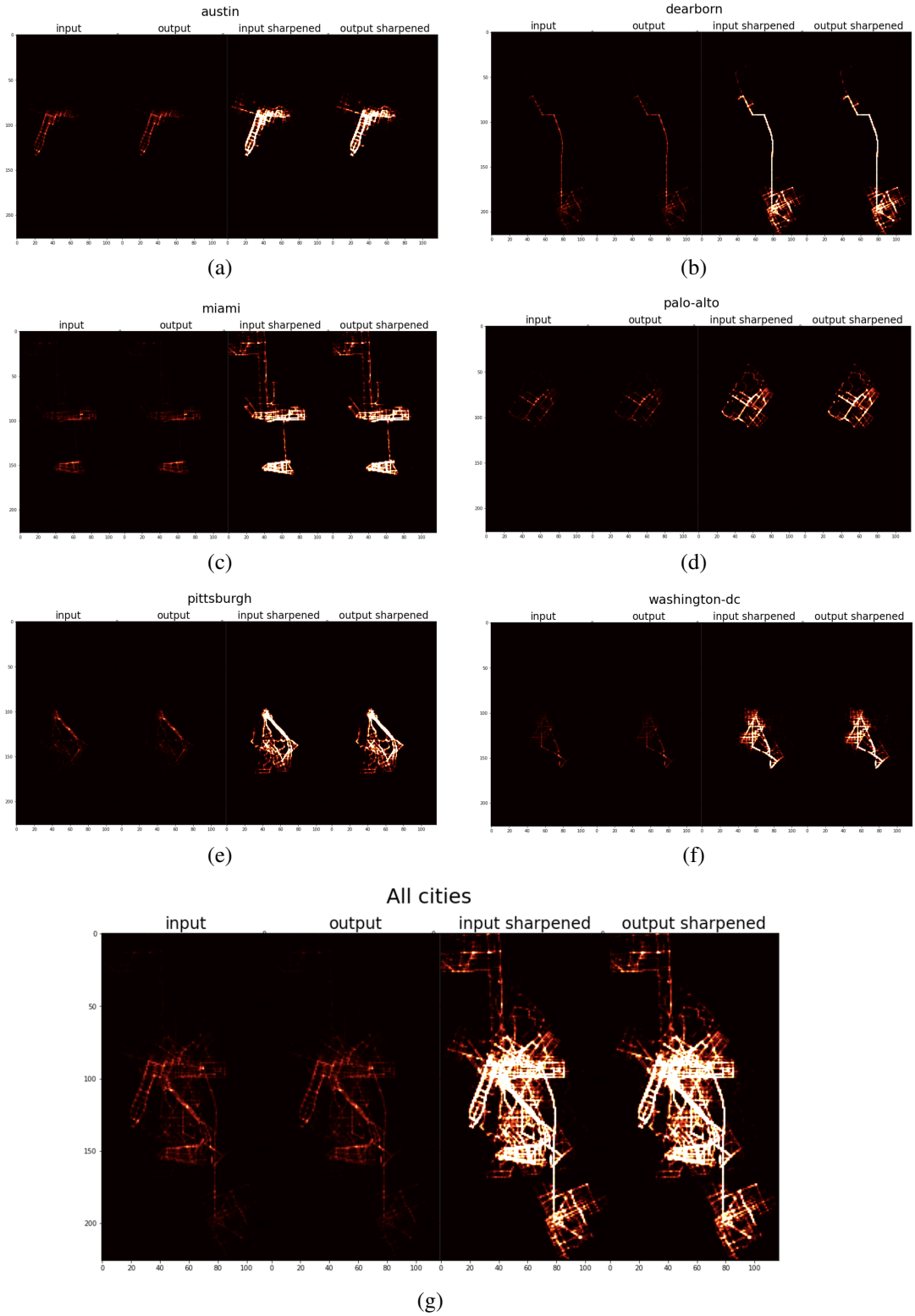



Figure 1: Position heatmaps for (a) Austin, (b) Dearborn, (c) Miami, (d) Palo Alto, (e) Pittsburgh, (f) Washington DC, and (g) all cities combined. Trajectory points are aggregated into chunks to generate brighter heatmaps, and from there are sharpened even further

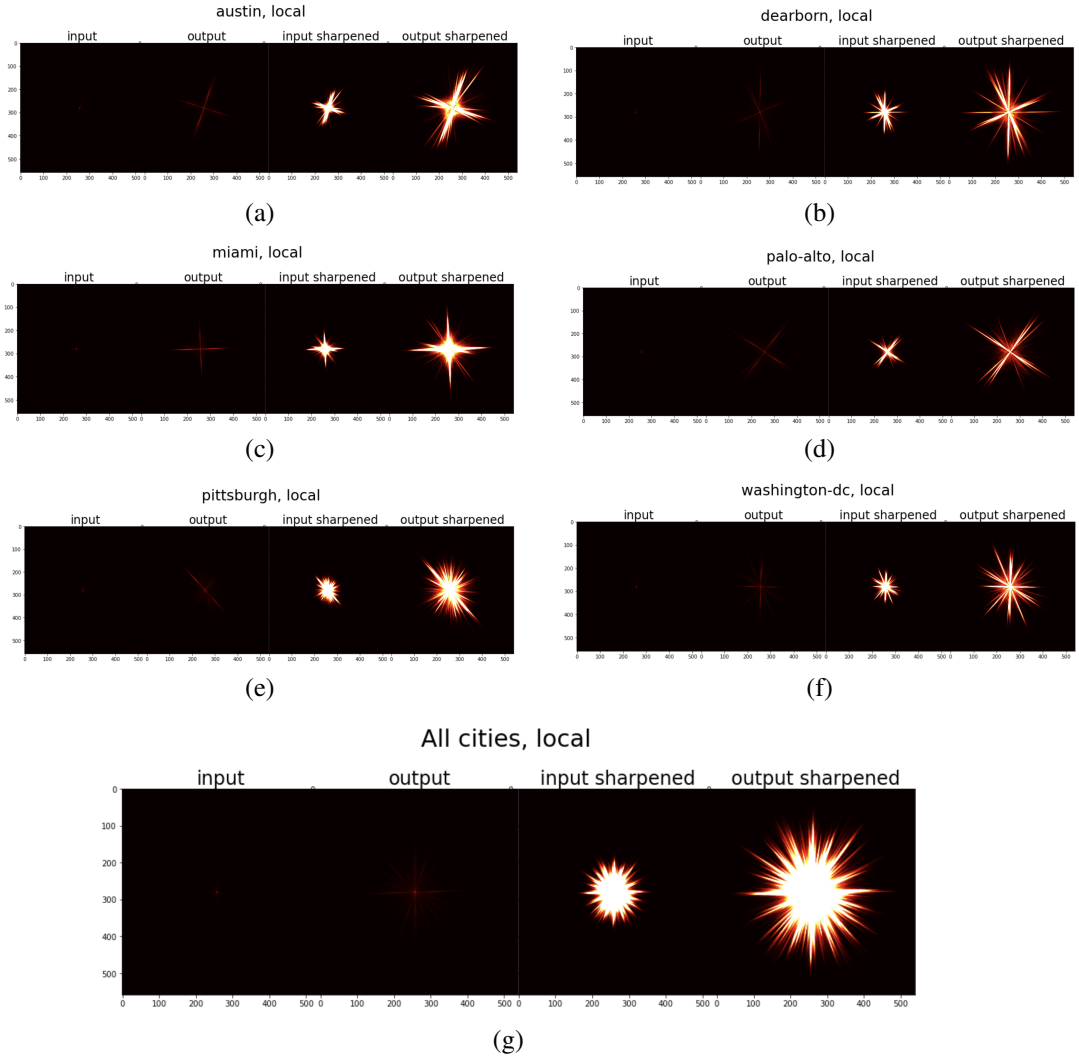



Figure 2: Local position heatmaps for (a) Austin, (b) Dearborn, (c) Miami, (d) Palo Alto, (e) Pittsburgh, (f) Washington DC, and (g) all cities combined. These heatmaps do not aggregate points into chunks.

## 2.2 Problem B

We analyzed 5 aspects of the given data: position distribution, local position distribution, velocity distribution, acceleration patterns, and path-length patterns.

We define position as the unaltered data taken directly from the given datasets (i.e. the unaltered $x$ and $y$ sequences). As shown in Figure 1, we constructed 4 position heatmaps (1 constructed from input data, 1 from output data, and the the sharpened versions of the input and output heatmaps) for each city along with a single combined heatmap incorporating all cities. These heatmaps are constructed such that each pixel represents a $100 \times 100$ chunk, and the brightness of each pixel corresponds to the number of positions ($x_i$ or $y_i$ vectors) that lie in its respective chunk. Despite aggregating positions into chunks, the unaltered heatmaps are relatively dim for every city, suggesting that the given data is very widely distributed throughout each city region. To get a better idea of the position distribution, we "sharpen" each heatmap (i.e. clipping higher pixel values at a set constant to reduce the range of pixel values) such that pixels with lower values can be seen on the heatmap. The sharpened heatmaps clearly outline each city's road layout, as if seen from space. By examining these generated heatmaps, we see that gridded downtown areas shine the brightest. Hence, we can suspect that the data will contain many trajectories that travel along gridded streets and pass through intersections.

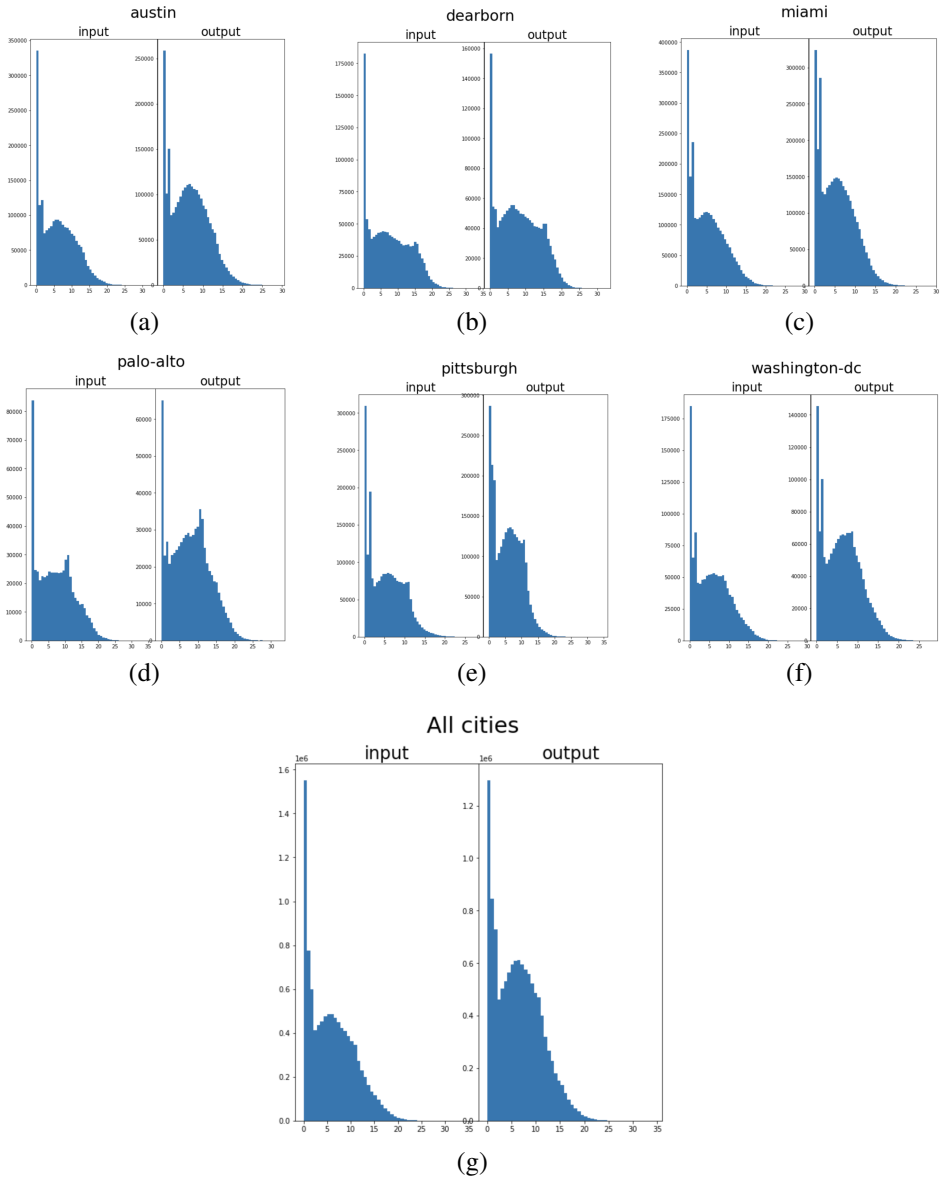



Figure 3: Speed histograms for (a) Austin, (b) Dearborn, (c) Miami, (d) Palo Alto, (e) Pittsburgh, (f) Washington DC, and (g) all cities combined.

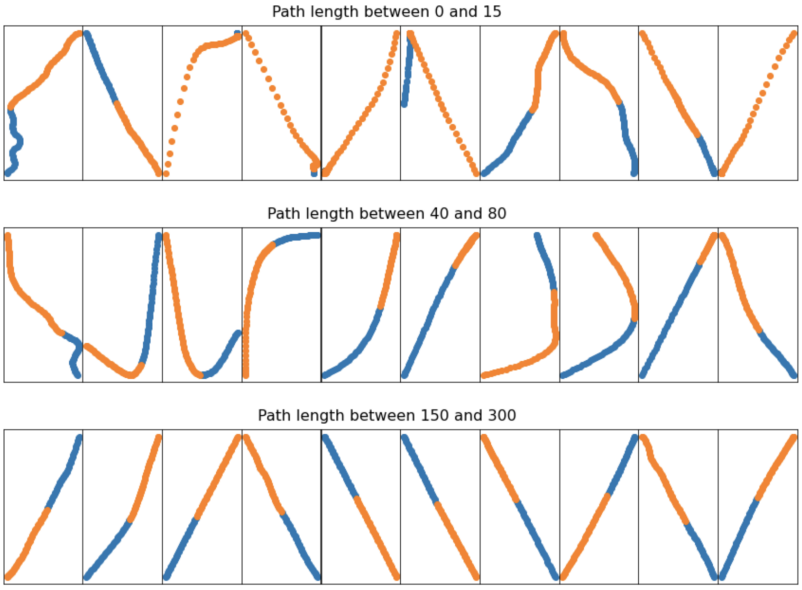



Figure 4: Trajectory samples grouped by path length (0-15 in row 1, 40-80 in row 2, 150-300 in row 3). The input trajectories are in blue while the ground truth output trajectories are in orange. It is clear from the image that the most irregular trajectories have the shortest path lengths.

We define local position as positions relative to the initial point of the corresponding input sequence (i.e. $\overline{x}$ and $\overline{y}$, which can be easily generated from the given data). As shown in Figure 2, We similarly generated 4 heatmaps per city (and the combination of all cities), with the exception that each pixel corresponds to a specific local position (i.e. $\overline{x}_i$ or $\overline{y}_i$) as opposed to a $100 \times 100$ chunk of positions. We made this decision because local positions are much less widely distributed than regular positions. Nevertheless, the unsharpened heat maps are still dim, suggesting that the local position distribution is still somewhat spread out. The sharpened heatmaps split the cities into 2 categories: Austin, Miami, and Palo Alto all display 4-pointed stars, but Pittsburgh, Dearborn, and Washington DC display stars with many more points. This suggests that Austin, Miami, and Palo Alto have a more regular, gridded road layout (since 4-pointed stars have shapes very similar to those of road intersections) while Pittsburgh, Dearborn, and Washington DC have a less regular layout. When compared with the unaltered position heatmaps, this seems to make sense. Another observation we made is that the output stars have a larger "diameter" than the input stars, which suggests that the trajectories tend to proceed in a single direction instead of doubling back on themselves.

We define velocity as the finite difference between adjacent positions, scaled by the timestep between positions (i.e. $v_x$ and $v_y$). We recorded each individual velocity (i.e. $v_{x_i}$ or $v_{y_i}$) and plotted their magnitudes (i.e. speeds) onto histograms, as shown in Figure 3. As before, we have different histograms per city (as well as the combined case), and we categorize velocities based on whether they were calculated from an input position sequence or an output position sequence. The histograms follow a multi-peaked pattern. The first, and largest, peak occurs near 0, suggesting that the vehicles spend a large amount of time stopped. The second peak is a bit past 5, and there is a 3rd peak near 10 or 15 for some cities. These two peaks seem to indicate city driving and freeway/non-city driving, respectively. Austin and Miami don't have a 3rd peak while the other 4 cities do, which further suggests that Austin and Miami have a more regular, city-grid road layout as opposed to other cities (Palo Alto doesn't follow this trend, however). In addition, we noticed that the first peak seems to shrink relative to the second and third peaks in the output histograms, as compared to the input histograms, for all cities. This suggests that the vehicles tend to accelerate instead of decelerating.

As we examined specific trajectory examples, we found that some trajectories were very irregular, composed of many sharp turns and velocity changes. We evaluated 2 metrics for measuring trajectory irregularity: acceleration ($a_x$ and $a_y$) and path length ($\|x\|$ and $\|y\|$). We examined sample trajectories in groups, differentiating by average acceleration magnitude or total path length (i.e. $\|x\| + \|y_1 - x_{50\|+}\|y\|$). Trajectories with shorter path lengths were clearly more irregular than trajectories with

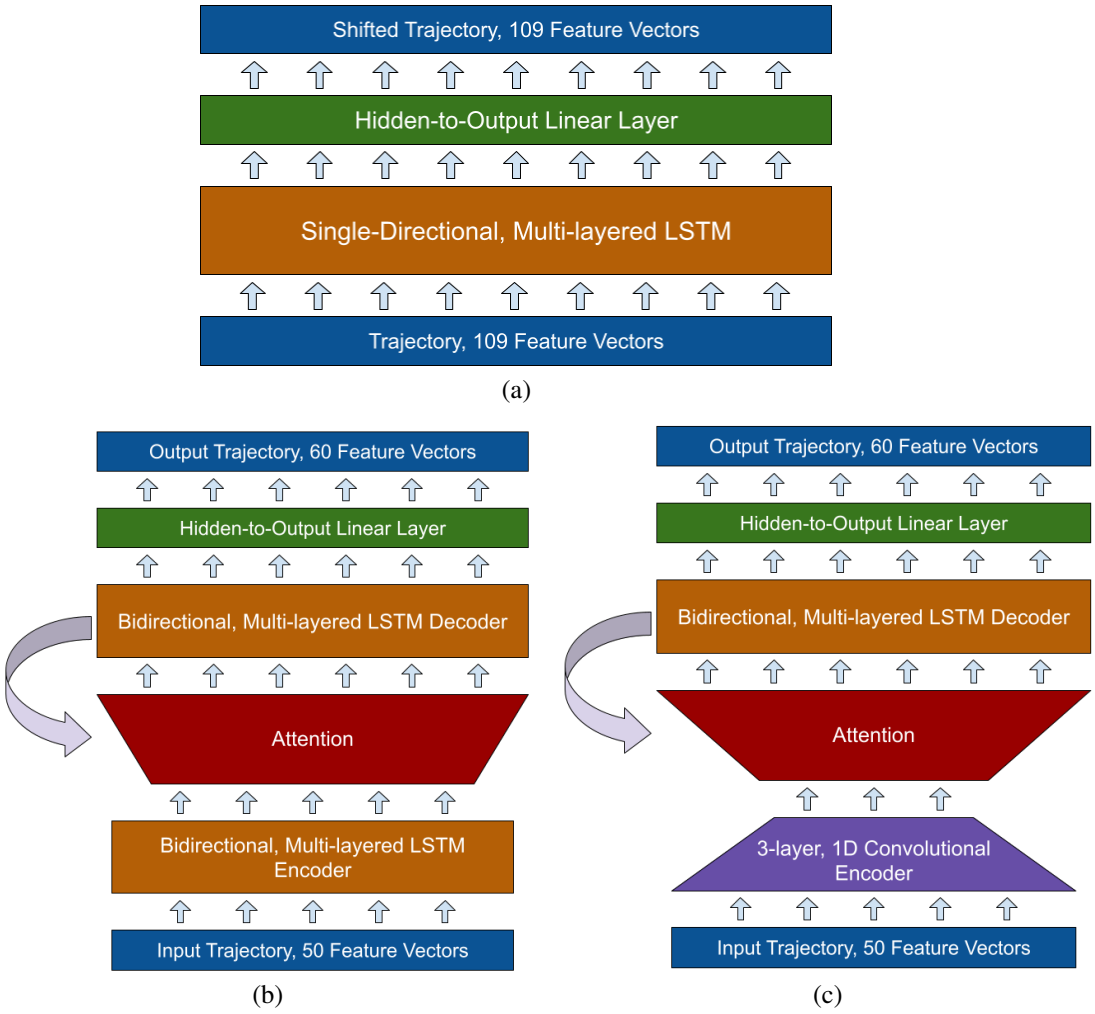



Figure 5: Model architectures for (a) LSTM model, (b) LSTM Encoder Decoder, (c) Conv2Seq

longer path lengths, as is clear from Figure 4. We initially thought that there was a similar relation between irregularity and acceleration, but after the competition we reevaluated the acceleration data and realized this wasn't the case.

### 2.3 Problem C

We used an 80%-20% training and validation split, assigning points randomly. The sizes of each split are displayed in Table 1. We performed feature engineering by replacing the given sequences of position vectors with sequences of processed feature vectors informed by our data analysis. For example, our most successful feature engineering scheme, as explained later, used concatenated local position and velocity vectors, in the form of $x'_i = [\overline{x}_i] + [v_{x_i}]$. As a result our altered input and output vectors $x'$ and $y'$ had dimensions $50 \times 4$ and $60 \times 4$ respectively. We didn't perform any extra normalization, other than the translational normalization provided by using local position. Despite the differences between each city's data, we combined the data from every city into a single dataset and mainly trained on this dataset. Our rationale for this is explained later in this report.

## 3 Machine Learning Model

In the following section, we describe our model architectures.

### 3.1 Problem A

We did not end up trying any normal machine learning models. We instead jumped straight to an LSTM model, as the data was clearly sequential. If we were to try a normal machine learning model, however, we would likely choose to use linear regression, as other teams have had success with this model. We likely would flatten the given position data into vectors (size 100 for input, size 120 for output) and then pass those into the linear regression model. Since the competition criterion is MSELoss, we would use MSELoss for this model as well.

### 3.2 Problem B

For most of our deep learning models, the input and output feature vectors we used were the $x'$ and $y'$ sequences (defined in section 2.3) constructed after feature engineering. We explain our rationale behind this in section 4. In addition, as explained in the previous subsection, we used MSELoss throughout our training process.

As explained earlier, our models were LSTM-based because the data was sequential. Out of curiosity, we did try using 1D convolutional layers as well. We also took inspiration from Attention-Based Encoder Decoder models, as these models are better at finding global patterns throughout a given input sequence than Seq2Seq models. Our experimentation with these models is explained in section 4.

### 3.3 Problem C

We mainly used 3 models, designed as follows

- LSTM (Figure 5(a))
  - Used 64-256 LSTM hidden units
  - Used 2-4 LSTM layers
  - Best performance with 128 hiddens and 3 layers: MSE 91
  - Best performance with 256 hiddens and 4 layers (last-ditch attempt): MSE 75
- LSTM Encoder Decoder (Figure 5(b))
  - Used 128-256 hidden units. Final model used 256 hidden units
  - Used 2-3 layers for both encoder and decoder. Final model used 3 layers
  - Encoder and decoder are both bidirectional.
  - Used weighted dot-product attention scoring function. Context vectors applied to 1st layer of decoder only
  - Best performance: MSE 21, best model overall
- Conv2Seq (Figure 5(c))
  - Decoder and attention mechanism identical to that of LSTM Encoder Decoder
  - 3-layer 1D convolutional encoder, arranged as follows: 32 kernels of size 4, 64 kernels of size 8, and 128 kernels of size 16. No padding, stride of 1
  - Not submitted, as performance did not improve upon LSTM Encoder Decoder.

It's important to note that we designed the simple LSTM to take each element in the entire trajectory ($T' = [x'] + [y']$) and predict its following element. Hence, to make predictions, we needed to use auto-regression to unroll the model to 109 LSTM units, as shown in the diagram. We used teacher forcing to train this model. The training and prediction pipelines for the LSTM Encoder Decoder and Conv2Seq models were more straightforward.

We used dropout with $p = 0.5$ between each layer in all of our models for regularization. Our rationale behind this is explained in section 4.

## 4 Experiment Design and Results

In the following section, we describe our model training setup, experiments, and final results.

Table 2: Data analysis informed experiments, ordered chronologically

| Experiment | Results (MSE change) | Additional Comments |
|---|---|---|
| Local Position Features | $10771797 \rightarrow 726$ | Significant improvement |
| Adding Velocity Features | $176 \rightarrow 99$ | Significant improvement |
| City Specialization | $99 \rightarrow 91$, $25.8 \rightarrow 24.3$ | Mediocre improvement, slow training |
| Adding Accleration Features | $24.3 \rightarrow 23.8$ | Mediocre Improvement |
| Path-Length-Weighted Loss | N/A | No improvement, not submitted |

Table 3: Model architecture and hyperparameter experiments, ordered chronologically

| Experiment | Results (MSE change) | Additional Comments |
|---|---|---|
| Using Dropout | $511 \rightarrow 193$ | Significant improvement |
| LSTM $\rightarrow$ LSTM Encoder Decorder | $75 \rightarrow 26$ | Significant improvement |
| Conv2Seq | N/A | No improvement, not submitted |
| Altering Attention Mechanism | N/A | No improvement, not submitted |
| Doubling Hiddens, $128 \rightarrow 256$ | $23.8 \rightarrow 21.8$ | 1 epoch convergence |

### 4.1 Problem A

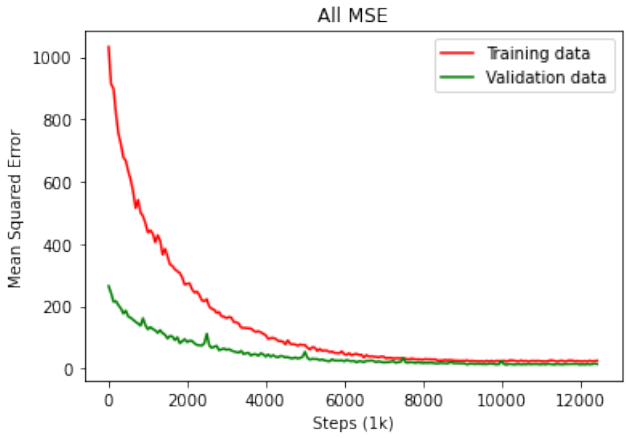
We trained our model on the UCSD DSML Platform to take advantage of its GPU. We used an Adam optimizer, because it is state-of-the-art and widely used, with a learning rate of 1e-4. We initially started with a learning rate of 1e-2 to test the learning capability of the model by overfitting on the train set and gradually reduced it to 1e-4. We also added a ReduceLROnPlateau learning rate scheduler that reduced the learning rate by a gamma factor 0.1 whenever the validation loss plateaued. We used a batch size of 32, tuned to balance quick learning and stochasticity. How each model made 60-step prediction is explained in section 3. We evaluated our model on our validation set every few iterations, such that 1 epoch through the training set was approximately equivalent to 1 epoch through the validation set. We trained most of our models for 5 epochs on the combined city dataset. This was long enough for our models to converge, as shown in Figure 6(a). Epoch times varied based on which model architecture we used, and we mention specific times in the following subsection.

### 4.2 Problem B

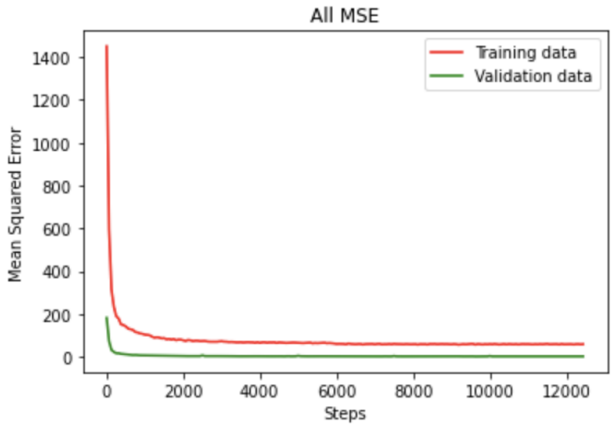
Here, we categorize our experiments into 2 sets: data analysis informed experiments and model architecture/hyperparameter experiments.

The data analysis informed experiments are displayed in table 2. As discussed in our data analysis, we found that local position and velocity features effectively capture patterns in the given data. These features are also less widely distributed than the original position data. As a result, we experimented with passing local position and velocity vectors as inputs instead of global position (i.e. the $x'$ and $y'$ sequences as discussed in section 2.3). As shown in the first 2 rows of the table, these modifications drastically improved our performance.

However, our other data-related experiments were not as promising. Our data analysis indicated that the data varied between cities, so we attempted a 2-step training scheme: we first trained a model on the combined city data and then used that model as a baseplate for 6 specialized models, each of which was trained on data from only 1 city. We attempted this a couple times throughout our experimentation process, but, as shown in row 3 of the table, this did not result in any worthwhile improvement despite more than doubling our training time. In addition, we tried to pass trajectory irregularity information through acceleration and path length, hoping that our model will learn to recognize which trajectories are irregular and predict accordingly. We applied acceleration by concatenating it to the end of our ${x'}_i$ and ${y'}_i$ feature vectors, and we applied path length by weighting the loss function to more heavily penalize samples with shorter path lengths. However, as shown in the last 2 rows of the table, neither of these resulted in a significant improvement. In fact, adding path length weighting resulted in worse validation loss during training, so we didn't even attempt to submit it. As a result, our final model used local position and velocity data, but nothing else.




(a)

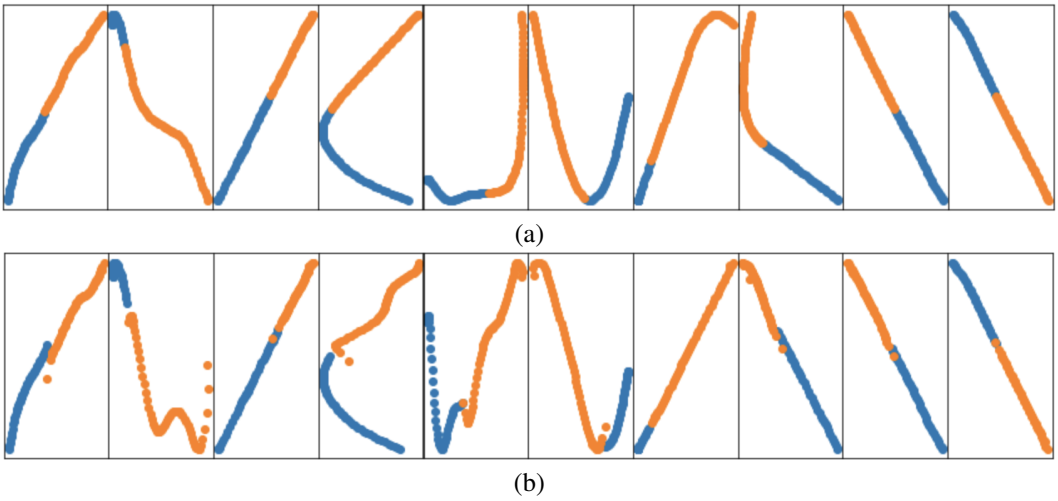



(b)

Figure 6: (a) Typical loss curve during our experimentation vs. (b) the loss curve of our final model, after doubling hidden units. Most of our models converged in 5 epochs, but our final model converged in only 1

The model architecture and hyperparameter experiments are displayed in table 3. Early on, we experimented with using dropout in our LSTM model. As shown in the first row of the table, this resulted in a large performance boost, so we continued to use dropout with $p = 0.5$ throughout all of our models. Our most significant architecture-related improvement occurred when we switched from our LSTM model to our LSTM Encoder Decoder model. This model ran at around 10-15 minutes per epoch, much slower than the 2.5 minutes of the LSTM model, but was much more accurate, as shown in row 2 of the table. For most of our experimentation, we used 128 hidden units in our LSTM Encoder Decoder model. Our of curiosity, we decided to double our hidden units near the end of our experimentation. As shown in the last row of the table, this resulted in only a mediocre improvement in MSE, but allowed our model to converge in 1 epoch as opposed to 5, which was unexpected. The change in convergence is also demonstrated by the loss curves in Figure 6

As with our data-related experiments, some of our model-related experiments were less promising. We attempted to alter our LSTM Encoder Decoder architecture by altering mechanics of the Attention mechanism (i.e. passing context vectors to every layer in the decoder instead of just the first layer, using multiple attention scoring functions, etc.) and even switching the encoder with a 1D Convolutional Layer. These models still performed better than our original LSTM, but they did not improve upon the LSTM Encoder Decoder, as shown in rows 3 and 4 of the table. They also trained at about the same speed as the LSTM Encoder Decoder. As a result, we used the LSTM Encoder Decoder as our final model.



(a)

(b)

Figure 7: (a) Actual paths vs. (b) final model predicted paths. Blue points are shared inputs, orange points are outputs.

### 4.3 Problem C

The loss curves of our final model are displayed in Figure 6(b), and some predictions are displayed in Figure 7. As shown, our model converged within 1 epoch. Our model is quite accurate on straighter, more regular trajectories, but it often failed to correctly predict irregular trajectories. Our final MSE on the private leaderboard was 21.78497, putting us in 10th place overall.

## 5 Discussion and Future Work

### 5.1 Problem A

As discussed in section 4, our most successful feature engineering strategy involved replacing the global position data with local position and velocity vectors. This makes sense because, as shown in our data analysis, there were clear patterns in the local position and velocity data that revealed more about the given trajectory dataset than global position alone. In terms of model design, our best result came from switching to our LSTM Encoder Decoder. Although there were other design choices that helped improve our ranking (such as using dropout), these two choices were the ones that stood out to us most.

One thing we learned is that good data analysis results in good feature engineering. When our analysis of the given global positions revealed that the data was very widely spread out, we decided to investigate local position and were instantly rewarded. Once we found patterns in the velocity data as well, we again incorporated that in our feature engineering and again improved our performance. Not all of our data analysis was useful, such as our path length and acceleration analysis, but we would not have performed so well without our data analysis. If I were to give advice to a deep learning beginner, I would tell them to invest time early in their experimentation to thorough data analysis and then come up with feature engineering skills based on their results.

The largest bottleneck we faced occurred when we were trying to improve upon our LSTM Encoder Decoder model's performance after it hit an MSE of around 25. The main issue was that, although our model was performing well, we didn't know what exactly it was doing. As a result, we had no way to accurately guess what changes would improve our model. This problem was exacerbated by the wide search space of possible experiments we could perform. As a result, many of the experiments we tried near the end of our project timeline were unsuccessful. However, given extra time, we would like to revisit these experiments to make sure we didn't make any mistakes.

Although our feature engineering was successful, our features were still purely sequential. With more resources, we would like to explore how our model could predict trajectories from non-sequential input data, such as an image or heatmap of the trajectory as opposed to a sequence of vectors. We

also want to explore how well a model could perform if we passed two types of data (i.e. both sequential and non-sequential) to a model, such as in [3]. Our rationale is that the model could have two perspectives of the trajectory, like with 2 senses, and then will be able to extract more information from each given trajectory.

## References


[1] Huatao Jiang, Lin Chang, Qing Li, and Dapeng Chen. Trajectory prediction of vehicles based on deep learning. In *2019 4th International Conference on Intelligent Transportation Engineering (ICITE)*, pages 190–195, 2019.

[2] Jianbang Liu, Xinyu Mao, Yuqi Fang, Delong Zhu, and Max Q. H. Meng. A survey on deep-learning approaches for vehicle trajectory prediction in autonomous driving, 2021.

[3] Karttikeya Mangalam, Yang An, Harshayu Girase, and Jitendra Malik. From goals, waypoints paths to long term human trajectory forecasting. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 15213–15222, 2021.