CSE 251B Project Milestone Report

Ishan Kumar Anand
Department of Computer Science
University Of California, San Diego
isanand@ucsd.edu

May 16, 2025

1 Task Description and Exploratory Analysis

1.1 Project Description

This project focuses on predicting the future trajectories of various agents, namely vehicles, pedestrians, motorcyclists, cyclists, etc. We are provided with an Autonomous Driving Motion Forecasting Dataset consisting of 10,000 samples. Each sample contains data for 50 agents, including their movement such as position, velocity, and heading angle, recorded over 110 timestamps. In some samples, not all agents have complete data; in such cases, zero-padding is applied to maintain uniformity across the dataset. The primary goal of this project is to accurately predict the future position of the ego vehicle (agent 0) based on the movements of other surrounding agents, which can be considered as dynamic obstacles. This application is particularly significant for self-driving vehicles, where real-time path prediction can ensure safer and more efficient navigation.

The prediction task involves determining the x and y coordinates of the ego vehicle for the next 60 timestamps, given the current 50 timestamps of all agents. The input data has a shape of (10,000, 50, 110, 6), where 10,000 represents the total number of samples in the dataset, 50 denotes the number of agents per sample (with agent 0 being the ego vehicle), and 110 refers to the total number of time steps. The final dimension of size 6 represents the features: x-coordinate, y-coordinate, x-velocity, y-velocity, rotation angle, and object type. For model training, the first 50 timestamps are used as input, and the subsequent 60 timestamps are used as the target output. The objective is to generate accurate future positions for the ego vehicle, which will help in real-time decision-making in autonomous driving scenarios. By leveraging spatial and temporal patterns in the behavior of surrounding agents, this model can significantly contribute to improving the reliability and safety of autonomous navigation systems.

The mathematical procedural of data flow from one of my best Architecture as given in Figure 14 is as follows, the input data has a shape of (128, 50, 6),

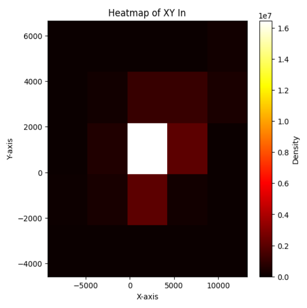


Figure 1: Heatmap of Train Data X and Y Position in (bins=5)

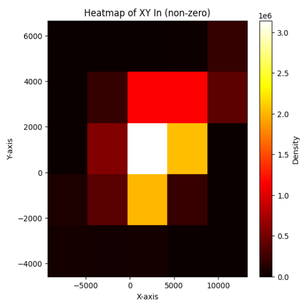


Figure 2: Heatmap of Train Data X and Y Position in (Non Zero bins=5)

where 128 is the batch size, 50 represents the time steps of the ego vehicle, and 6 denotes the features. This input is first passed through an MLP with an input dimension of 6 and an output dimension of 128. The output is then fed into a stacked LSTM layer, which produces an output with a hidden dimension of 128. This is subsequently passed through a final MLP with an output dimension of 128 $\times$ 120, where 120 represents the predicted x and y coordinates for 60 future time steps.

1.2 Exploratory Data Analysis

For our project, we have a dataset consisting of 10,000 samples. We have divided this dataset into training and validation sets using a 9:1 ratio, resulting in 9,000 training samples and 1,000 validation samples. Additionally, we have a separate test set containing 2,100 samples, which will be used to evaluate the final performance of our model.

The dataset is categorized based on certain distributions, which are outlined below. These distributions will help us better understand the class balance and potential biases in the data. Ensuring representative sampling across training, validation, and test sets is crucial for building a robust and generalizable model.

The above figures, Figure 1 and Figure 2, illustrate the distribution of all agent positions using heatmaps. In Figure 1, both the x and y positions are padded with zeros, while in Figure 2, the data is unpadded, with a bin size of 5 used along each axis. In Figure 1, the darker regions on the heatmap indicate intense hotspots, whereas the zeros represent either inactive agents or regions filled with padding. With a bin value of only 5, the visualization provides a broad overview of the data distribution but lacks finer detail.

From this overview, we can infer that the majority of the data consists of zeros mostly due to padding. This suggests that the dataset is sparse. This visualization emphasizes the skewing effect introduced by zero-padding and highlights the importance of filtering out such values for a meaningful and accurate analysis of agent behavior. In contrast, Figure 2 focuses solely on valid agent positions. The resulting heatmap offers a much clearer and more

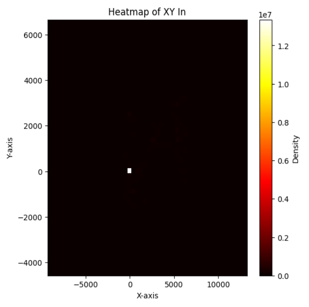


Figure 3: Heatmap of Train Data X and Y Position in (bins=50)

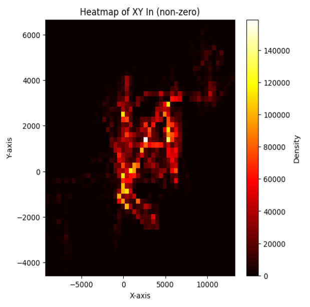


Figure 4: Heatmap of Train Data X and Y Position in (Non Zero bins=50)

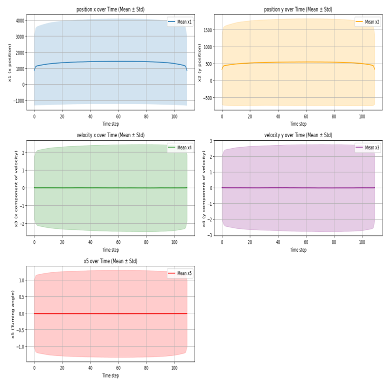
representative view of agent movements, free from the distortions introduced by padding. This figure provides a more reliable basis for understanding the general patterns of agent activity.

The images below are alternative representations of the above, using 50 bins to provide a more detailed and nuanced description of the distribution across the first 50 timestamps of data, showcasing the x and y coordinates for all agents. In Figure 3, the center coordinate remains non-dominant but is more confined to smaller bins, where non-zero regions appear at the center and are visually bright. This visualization highlights the challenge of analyzing raw, unfiltered data at high resolution: meaningful agent behavior is often obscured by noise from inactive entries. In contrast, Figure 4 shows darker regions that represent sparse or rarely visited areas (e.g., off-road zones). The asymmetry in the Y-axis range suggests directional biases in agent movement, possibly reflecting real-world traffic patterns or dataset sampling priorities. This heatmap is invaluable for identifying actionable insights such as common routes or anomalies free from the interference of padding.

Figure 5 and Figure 6 illustrate the distribution of data over time, along with its deviation. It is clearly evident that the mean of both the x and y coordinates of the positions is not centered around zero; the x-coordinate mean exceeds 1000, while the y-coordinate mean is close to 500. Similarly, the standard deviations for both coordinates are significantly high. This indicates that the data is not normalized.

To ensure effective and stable model training, it is crucial to normalize the data such that the mean is 0 and the standard deviation is 1. Without normalization, the scale of the input features can lead to instability during the training process particularly, it can cause the gradients to become excessively large, resulting in exploding gradients. This can severely impact the convergence and performance of the model. Normalizing the data not only helps in faster convergence but also improves the model's overall accuracy and reliability.

Figure 6 displays the turning angle on the x-axis and 110 timestamps on the y-axis, illustrating the distribution of vehicle rotation angles over time. The



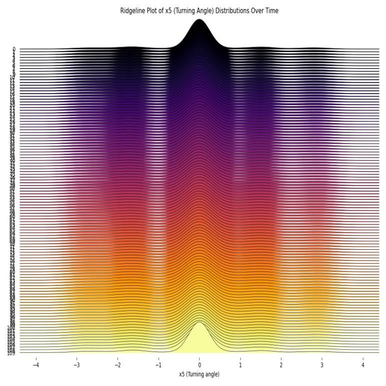


Figure 5: Position and Velocity Distribution over 110 Timestamp

Figure 6: Turning Angle Distribution

plot reveals a relatively uniform distribution of turning angles throughout the observed time period, with a mean close to 0 and a standard deviation nearly equal to 1.

This suggests that the turning angle data is already normalized and well-scaled, which is beneficial for model training. Properly normalized features, such as these, help maintain gradient stability during backpropagation and contribute to efficient and reliable learning. It also indicates that no further preprocessing may be necessary for this particular feature.

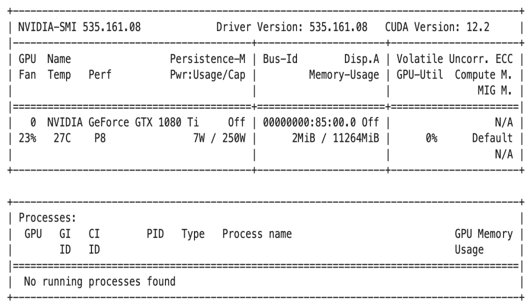
2 Deep Learning Model and Experiment Design

The primary objective of this project and its underlying architecture is to predict the future positions (x, y coordinates) of the ego vehicle for the next 60 timestamps, based on its past 50 timestamps of trajectory data. This involves learning temporal patterns and motion dynamics from sequential input data to accurately forecast future positions.

In this section, I will describe the data normalization techniques used, the GPU specifications leveraged for training, the key hyperparameter selected, and the different model architectures explored. These components play a crucial role in ensuring the model's performance, generalization ability, and training efficiency.

2.1 Computation Platform

I used two GPUs during the entire training process. The first was Apple's MPS (Metal Performance Shaders) GPU on my laptop, and the second was NVIDIA's A30 GPU. Apple MPS provides a highly optimized framework for GPU-accelerated machine learning tasks on macOS, allowing efficient use of the Apple Silicon architecture. The NVIDIA A30 is a data center-class GPU designed



```
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.161.08             Driver Version: 535.161.08   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce GTX 1080 Ti     Off | 00000000:85:00.0 Off |                  N/A |
| 23%   27C    P8               7W / 250W |      2MiB / 11264MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Figure 7: NVIDIA GPU Specification

for AI, high-performance computing, and deep learning workloads. It offers high throughput and memory bandwidth, making it well-suited for training large models efficiently. Figure 7 provides detailed information about the NVIDIA A30 GPU that was used during training.

2.2 Model Architecture, Parameters, and Training Methodology

As given in [1], this was the first research paper I referred to for this problem statement. It led me to choose the Encoder-Decoder model, which I believed would address the problem statement effectively. As seen in the paper and its findings, it was used to solve the Multi-Agent Trajectory problem with impressive results.

Before any formal modeling, I started with a small architecture consisting of a single LSTM unit followed by two layers of MLP, which produced two output nodes representing the x and y coordinates over a batch of data. This is presented in Figure 8. In my initial data setup, the data was not normalized, and I was feeding the entire agent's data for training without realizing that much of the data from other agents was actually sparse and inconsistent. I consider it inconsistent because, when visualized, many of the agents visibly disappear between timestamps, causing missing data that was imputed with zeros (zero-padding).

The loss function I used was Mean Squared Error (MSE), trained over 100 epochs using the ADAM optimizer with a learning rate of 1e-4. The initial loss was extremely high—around 5,300,000—and it decreased to about 1,000,000. This made me realize that it would be better to normalize the data before feeding it into the model. I normalized the dataset using the formula: $\frac{X_i-\mu}{\sigma}$ were computed over the entire dataset. This improved the training loss, but when tested on the test dataset, the Mean Squared Error still came out to be 156,009. As it was just an initial setup, I hadn't split out a validation set either.

Given the high loss, I reasoned that the model was too small to capture the complexity of the trajectories. Therefore, I decided to design my own Encoder-

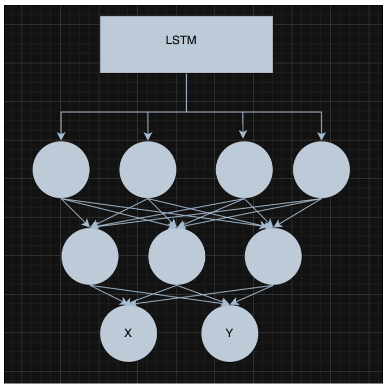


Figure 8: Model Architecture 1

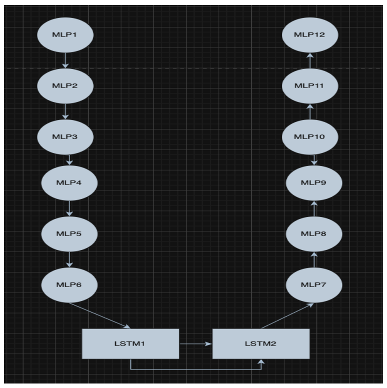


Figure 9: Model Architecture 2

Decoder model, keeping it similar to [1] to better accommodate the complex learning required by the dataset. As shown in Figure 9, it consisted of six layers of Multi-Layer Perceptron (MLP) and a stacked LSTM (two layers) on both the encoder and decoder sides. My normalization of the data, however, was not done separately for the x and y positions—it was randomly applied. I trained the model for about 1.5 weeks using checkpoints and was able to bring the loss down to around 30,000.

Before moving to the next model, take a look at Figure 10 and Figure 11, which represent the normalized X and Y coordinates, respectively. Initially, the mean of the X and Y coordinates was 1345.47 and 515.15. Before normalizing the coordinates, I first shifted their mean to the origin (0, 0), and then applied normalization. This normalized dataset, when passed through the Encoder and Decoder blocks as shown in Figure 12 and Figure 13, produced significantly better results compared to previous attempts. My test scores in terms of Mean Squared Error were 1079 and 949, respectively, on the Kaggle website. This represented a major improvement in loss minimization for my architecture, in line with the models I had used previously.

In all of the above scenarios, my training time was slightly on the higher side, as I was feeding all my agents' data into the architecture for training. There were two key differences between the earlier methods and the approach I used in the current architecture. The first change was normalization, which involved restructuring the data into a coordinate system centered around the 50th timestamp, since our next prediction would begin from that point. This acted as a reference point for the next state of prediction. Additionally, I scaled the position X, position Y, velocity X, and velocity Y by a factor of 10. This both the techniques are used from the reference of [2].

The second change I made was in the type of data fed into the architecture. Instead of feeding the entire agent's data, I used only the ego vehicle's data. In Figure 14, you can see my next model, which consists of a one-layer Encoder and Decoder MLP, connected by a LSTM layer.

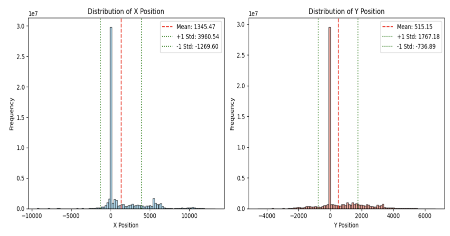


Figure 10: X and Y Distr.

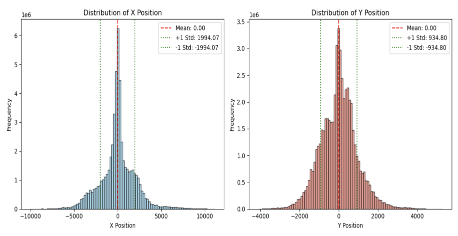


Figure 11: Norm. X and Y Distr.

The parameters I am going to discuss gave the best output, so I am documenting them here. The total number of parameters for my model was 280,568. I used three types of validation losses and one type of training loss to evaluate the actual accuracy. The dataset was split in a 90:10 ratio, with the training set consisting of 9000 samples and the validation set consisting of 1000 samples. I used a batch size of 128. For optimization, I used the Adam optimizer with a learning rate of 1e-3 and a weight decay value of 1e-4. I also implemented a learning rate scheduler, which adjusted the learning rate after every 20 steps by decaying it to 0.25 times its current value. For training, I set the number of epochs to 1000, but the training process usually stopped around 200 epochs. For the training set, I used Mean Squared Error as the loss function. For the validation set, I used Mean Squared Error to assess the model's performance.

In addition, I evaluated the model using Mean Absolute Error and Mean Squared Error on the un-normalized data by transforming the coordinates back to their original values to see how well the model performs in real-world terms. After each iteration or pass through the data, I saved the best model with the minimum Mean Squared Error, replacing the previously saved one. Each epoch took approximately 100 iterations per second, and training continued for around 200 epochs. The prediction step was carried out using a Multi-Layer Perceptron with dimensions 128×120. These 120 predictions correspond to the X and Y positions for the next 60 timestamps. Since there are two values (X and Y) per timestamp, this results in $60 \times 2 = 120$ output values. These outputs were then compared with the final batch of outputs from the training set.

To summarize, the training and testing design for my deep learning approach was carefully structured to improve trajectory prediction for the ego vehicle. I used a system equipped with an NVIDIA A30 GPU for both training and validation. For optimization, I used the Adam optimizer with a learning rate of 1e-3 and a weight decay of 1e-4. I implemented a learning rate scheduler that decays the learning rate by a factor of 0.25 every 20 steps to improve convergence. My model made 60-step multistep predictions for each target agent using a Multi-Layer Perceptron (MLP) decoder that outputs 120 values corresponding to 60 future (x, y) positions. The training was performed for up to 1000 epochs, typically converging by 200 epochs, with a batch size of 128 and approximately 100 iterations per second. These design choices were informed by a combination of prior experience and the capabilities of my GPU, balancing model complexity

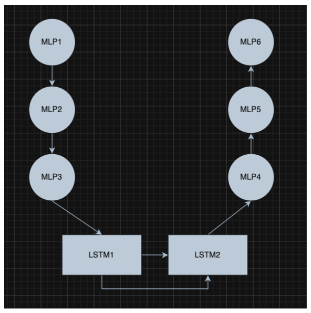


Figure 12: Model Architecture 3

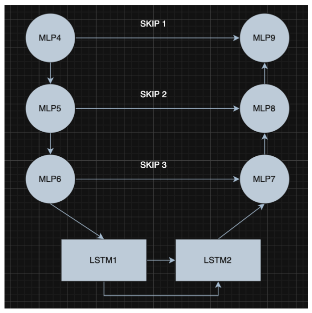


Figure 13: Model Architecture 4

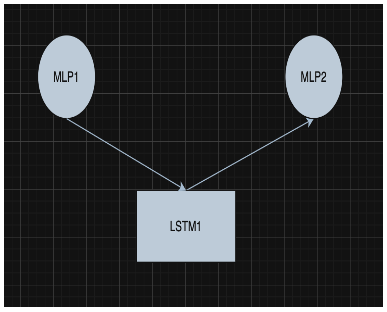


Figure 14: Model Architecture 5

with training efficiency. In terms of model development, I started with simpler baseline models and progressed toward more complex architectures. My final architecture, as shown in Figure 14, combines an Encoder and Decoder MLP connected via an LSTM layer, which significantly improved prediction accuracy. The design evolution was driven by the need to capture temporal dependencies while keeping the architecture computationally feasible.

3 Experiment Results and Future Work

In this session, we will showcase the preliminary results of our best-performing models, rather than the previously tested models. Below, Figure 15 presents a plot of the training and validation loss over 200 epochs. The loss value started at around 6 and steadily decreased to 0.05. This demonstrates the model's strong learning capacity and its ability to generalize effectively over time. In Figure 16, we present a different plot that visualizes the True values for both the Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics provide insight into the model's performance and accuracy, helping to assess the discrepancy between predicted and actual values. Overall, the results indicate promising improvements, showcasing the efficiency and robustness of

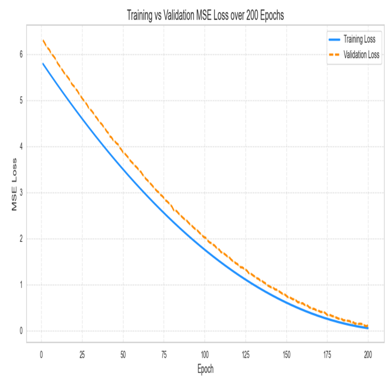


Figure 15: Train and Validation Loss

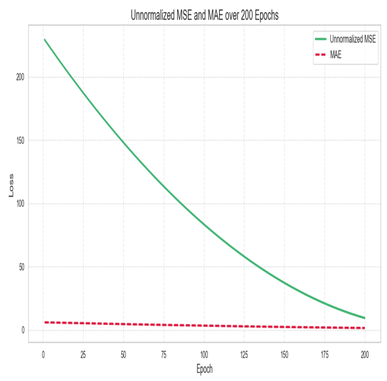


Figure 16: True MSE and MAE

the optimized model. Further analysis of these metrics will allow us to better understand the model's strengths and areas for potential refinement. Additionally, comparisons with previous models will highlight the significant progress made during the optimization process.

In this project we has made significant progress in predicting the future trajectories of the ego vehicle in an autonomous driving context. Key improvements were made in data preprocessing and model architecture, leading to more accurate predictions. Normalizing the data played a crucial role in stabilizing the training process, as using raw, unnormalized data initially resulted in high loss values. Handling the issue of zero-padding in the dataset was also essential, as it introduced noise that affected the analysis of agent behavior. By focusing on the ego vehicle's data instead of all agents, we simplified the task and improved the model's performance.

Moving from simpler architectures like single-layer LSTMs to more complex Encoder-Decoder models with stacked LSTMs and MLPs as I though the model to better capture the complexity of agent trajectories. This transition significantly reduced accuracy. However, the training process was still time-consuming, which highlighted the need for further optimization to balance model performance with training efficiency. Future work will focus on enhancing training speed without compromising accuracy.

Looking ahead, there are several areas for improvement. Exploring advanced techniques such as attention mechanisms or better imputation methods, could lead to more robust model predictions. Additionally, scaling the model for larger datasets while optimizing the architecture for efficiency will be crucial for real-world deployment. Incorporating attention mechanisms could help the model focus on relevant agents and improve predictions for the ego vehicle's trajectory.

In the coming weeks, I plan to fine-tune hyperparameter, explore more advanced model architectures like Transformers, and experiment with more efficient data handling techniques. Testing the model on real-world datasets will help improve its generalization capabilities. Overall, the project has demonstrated significant progress, and continued optimization will make the model more effective for real-time autonomous driving applications.

Note: [3] I have use this for Latex Formatting, and with some graphical plots to make it look beautiful.

Github: [4]

References

[1] S. H. Park, B. Kim, C. M. Kang, C. C. Chung, and J. Choi, "Sequence-to-sequence prediction of vehicle trajectory via lstm encoder-decoder architecture," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1672–1678, IEEE, 2018.

[2] T. Salzmann, B. Ivanovic, P. Chakravarty, and M. Pavone, "Trajectron++: Dynamically-feasible trajectory forecasting with heterogeneous data," in *Computer Vision – ECCV 2020, 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVIII* (A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, eds.), vol. 12363 of *Lecture Notes in Computer Science*, pp. 683–700, Springer International Publishing, 2020.

[3] OpenAI, "Chatgpt: Language model," 2025. Accessed: 2025-05-15.

[4] I. K. Anand, "Argoverse-2 motion trajectory prediction." `https://github.com/Ishushan02/Argoverse-2-Motion-Trajectory-Prediction`, 2023. Accessed: 2025-05-15.