CSE 251B Project Final Report

https://github.com/mizoreto/Autonoumous_Driving_Motion_Forecast

Jakob Getzel
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093
jgetzel@ucsd.edu

Jiayin Meng
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093
j3meng@ucsd.edu

Amaar Valliani
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093
avallian@ucsd.edu

Yule Zhang
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093
yuz300@ucsd.edu

Abstract

Autonomous diving has become a defining field in Computer Science in the twenty-first century. Turning towards Machine Learning models to solve the problem is a natural evolution of the task. Accurate trajectory prediction is a fundamental part of solving the problem and we attempt to leverage the ADAPT model to tackle the challenge. Our final submission achieved a score of 7.25389, placing us 7th on the private Kaggle leaderboard.

1 Introduction

Trajectory prediction is an important problem in autonomous driving, where the task is to predict the future positions of agents such as vehicles, pedestrians, and cyclists based on their past motion. Correct trajectory forecasting helps autonomous vehicles to foresee the behavior of surrounding agents and make safer navigation decisions. In real life, successful solutions have broad applications in fields like robotics, crowd simulation, and intelligent transportation systems.

For this project, we worked in the Trajectory Prediction Challenge using the Argoverse 2 Motion Forecasting Dataset in Kaggle competition. The starter code provided a baseline framework with three simple models: a linear regression model over flattened trajectory history, a feedforward MLP, and an LSTM operating only on the ego agent's past trajectory. These models apprehend either simple correlations or short-term temporal dependencies but have significant limitations. They fail to capture complex social dynamics, and struggle with making accurate long-horizon projections in complex and dynamic environments.

To overcome these limitations, we explored a range of advanced deep-learning architecture models. Our final work is inspired by the ADAPT model [1] which combines endpoint-conditioned trajectory prediction with attention-based fusion. Our model introduces an LSTM-based encoder for each agent, followed by a multi-head self-attention module to capture social interactions. To guide long-horizon prediction, we employ a coarse-to-fine endpoint prediction module with gradient detachment, enabling the decoder to generate future trajectories conditioned on a stable goal. This structure provides interpretability and results in improved accuracy and better handling of complex situations.

Our main contributions are as follows:

Preprint. Under review.

- **Multi-agent encoder-decoder architecture with attention-based interaction modeling:**
  We move beyond single-agent sequential models by encoding all agents' trajectories and capturing their interactions via a multi-head attention mechanism, allowing the model to reason about social context.
- **Coarse-to-fine endpoint-conditioned decoding:**
  We incorporate an endpoint predictor that generates a coarse goal followed by a refined offset. The predicted endpoint is then used to guide final trajectory generation, improving prediction stability and accuracy.
- **Demonstrated strong improvements over baselines:**
  Our model consistently achieves lower validation MSE and FDE compared to the linear regression, MLP, and LSTM models provided in the started code, validating the effectiveness of our architectural innovations.

Our final submission achieved a score of 7.25389, placing us 7th on the private Kaggle leaderboard in our final model.

2 Related Work

2.1 ADAPT: Efficient Multi-Agent Trajectory Prediction with Adaption

Recent works have focused on improving multi-agent trajectory forecasting by either increasing model complexity or sacrificing computational efficiency. Aydemir et al.[1] proposed a novel solution that achieves both high prediction accuracy and low inference cost through two key techniques: dynamic weight learning and endpoint-conditioned prediction with gradient stopping. The former enables the model to adaptively adjust the prediction head to each agent's individual reference frame within a scene-centric representation, effectively balancing the trade-off between pose-invariance and computational efficiency. The latter stabilizes training and enhances performance by decoupling the optimization of endpoint prediction and full trajectory generation. With these designs, ADAPT demonstrates strong performance using significantly fewer parameters and faster inference times compared to prior methods on benchmarks such as Argoverse and Interaction datasets. Moreover, its unified architecture supports both single-agent and multi-agent prediction under the same backbone.

Our final model is largely inspired by ADAPT. Specifically, we follow a staged prediction process comprising coarse endpoint prediction, endpoint refinement, and final trajectory generation, where the trajectory is conditioned on both the initial position and the refined endpoint. To stabilize training, we apply gradient stopping between the endpoint and trajectory prediction stages. Since our task focuses on single-agent trajectory prediction, we omit the dynamic weight learning component used in ADAPT. Furthermore, while ADAPT predicts multiple diverse endpoints, our model predicts only a single endpoint.

2.2 AgentFormer: Agent-Aware Transformers for Socio-Temporal Multi-Agent Forecasting

Yuan et al.[2] introduced a transformer-based framework that jointly models both temporal dynamics and social interactions of multiple agents by attending to all agents across all time steps. Unlike prior works that treat the two dimensions separately, AgentFormer enables a richer representation of multi-agent behavior by maintaining agent identity throughout the sequence.

While our architecture does not adopt AgentFormer's structure, we referred to its attention mechanism as inspiration when experimenting with an LSTM encoder–Transformer decoder architecture in early-stage model development.

2.3 State Space Models: Efficiently Modeling Long Sequences with Structured State Spaces

Albert Gu et al. [3] proposed the Structured State Space sequence model (S4) to efficiently model long sequences. This model is based on the state space model (SSM), which maps an input signal to an output through a latent state. The S4 model can be represented as a continuous-time model, a recurrent model, or a convolutional model.

The S4 model has demonstrated strong performance on a variety of benchmarks, including tasks with sequences of up to 16,000 steps, and has shown to be competitive with, and in some cases superior to, traditional models like RNNs and Transformers on tasks such as speech classification and image classification.

3 Problem Statement

3.1 Problem

In this project, we explore how to forecast the trajectory of a vehicle, the ego agent. As context, have past position and velocity information of the ego agent, along with other agents like external vehicles, pedestrians, or cyclists.

This problem aims at assessing how well models can understand road activities and use the understanding to predict agent movements. The strides made in the space of this problem are crucial to the development of self-driving, traffic management, and safety features to avoid crashes.

3.2 Dataset

To work on our problem, we are using the Argoverse 2 dataset. The training dataset is

$$\mathbf{X}^{train} \in \mathbb{R}^{10000\times 50\times 110\times 6}$$

Where $N = 10000$ number of training scenes, $A = 50$ is the number of agents per scene (including padded agents), $T = 110$ is the number of time steps per scene, and $D = 6$ is the features per agent per time step. Looking at the features, we have 1: x-position, 2: y-position, 3: x-velocity, 4: y-velocity, 5: heading angle, and 6: agent type (encoded as an int, one of 10 options). Looking at a specific scene, agent, and time stamp, we have this vector $x_{n,a,t} \in [d_1, d_2, d_3, d_4, d_5, d_6]$. The output for the train data is

$$\hat{\mathbf{Y}} \in \mathbb{R}^{10000\times 60\times 2}$$

We have 10000 scenes and 60 timestamps at each scene, which are the next movements in the coordinates $(x, y) \in \mathbb{R}^2$ for the ego vehicle.

The test dataset is $\mathbf{X}^{test} \in \mathbb{R}^{2100\times 50\times 50\times 6}$. This is mostly the same as the train input, but we have only $N_t = 2100$ test scenes and only $T_t = 50$ time stamps per scene. The output for the test data is $\hat{\mathbf{Y}} \in \mathbb{R}^{10000\times 60\times 2}$.

3.3 Dataset Changes

Given our Argoverse 2 dataset, we decided to split the training dataset into the first 50 timestamps for the X, which is our model input, and the last 60 timestamps for the Y, which is the ground truth of the input. We then split our training dataset into a train/validation split of $90/10$ so our validation set has the size 1000. Thus, the inputs of the model for training are

$$\mathbf{X}^{train} \in \mathbb{R}^{9000\times 50\times 50\times 6}$$

The inputs for validation are

$$\mathbf{X}^{validation} \in \mathbb{R}^{1000\times 50\times 50\times 6}$$

and the output is

$$\hat{\mathbf{Y}} \in \mathbb{R}^{10000\times 60\times 2}$$

3.4 Data Preprocessing

We perform a series of preprocessing steps to normalize and augment the raw trajectory data before feeding it into our model. Each scene contains motion histories for 50 agents over 110 time steps, where the first 50 steps represent the past, and the remaining 60 steps are the future ground truth.

Formally, let the raw input tensor be $\mathcal{S} \in \mathbb{R}^{M\times 50\times 110\times 6}$, where $\mathbf{M}$ is the number of scenes, and each agent is described by a 6-dimensional feature vector containing (x, y) position, (v_x, v_y) velocity, heading angle, and object type.

For each sample, we extract:

- The **historical trajectory** $\mathbf{x} \in \mathbb{R}^{50\times 50\times 6}$ for all agents.
- The **ground-truth future trajectory** $\mathbf{y} \in \mathbb{R}^{60\times 2}$ for the ego agent (index 0).

To improve generalization, we apply random data augmentations during training:

- With 50% probability, we apply a random in-plane rotation $\mathbf{R} \in \mathbb{R}^{2\times 2}$ to all positional and velocity components.
- With 50% probability, we reflect trajectories across the y-axis by negating the x-coordinates.

To remove global positional variance and preserve relative motion, we subtract the final historical position of the ego agent, $\mathbf{x}_{0,49,:2}$, from all agent trajectories:

$$\mathbf{x}_{i,t,:2} \leftarrow \mathbf{x}_{i,t,:2} - \mathbf{x}_{0,49,:2} \quad \mathbf{y}_t \leftarrow \mathbf{y}_t - \mathbf{x}_{0,49,:2} \tag{1}$$

Finally, to stabilize training, we normalize the first four features (position and velocity) by a scale factor α (typically $\alpha = 7.0$):

$$\mathbf{x}_{i,t,:4} \leftarrow \frac{\mathbf{x}_{i,t,:4}}{\alpha}, \quad \mathbf{y}_t \leftarrow \frac{\mathbf{y}_t}{\alpha} \tag{2}$$

During evaluation, the same normalization is applied, but data augmentations are disabled.

3.5 Input Representation

Let $\mathbf{X} \in \mathbb{R}^{B\times N\times T\times C}$ be the input tensor, where $\mathbf{B}$ is the batch size, $\mathbf{N} = 50$ is the number of agents, $\mathbf{T} = 50$ is the number of historical time steps, and $\mathbf{C} = 6$ is the feature dimension.

4 Method

Our model is inspired by ADAPT [1] referenced in Section 2.1, which proposes a coarse-to-fine endpoint prediction framework followed by trajectory interpolation. Our model consists of four main components: a Feature Encoder, an Agent-Agent Attention module, an Endpoint Predictor, and a Trajectory Decoder.

Given a set of $\mathbf{N} = 50$ agent motion histories, each represented by a sequence of $\mathbf{T} = 50$ steps with 6-dimensional features, the encoder embeds each polyline into a fixed-size representation. These encoded features are then fused via a multi-head attention mechanism that models interactions between agents. We then feed the fused ego agent features into the endpoint predictor to first predict a coarse endpoint and then refine it via offset regression. Finally, the trajectory decoder predicts the full future trajectory conditioned on the fused ego agent features and the predicted refined endpoint. Figure 1 illustrates the architecture of our model.

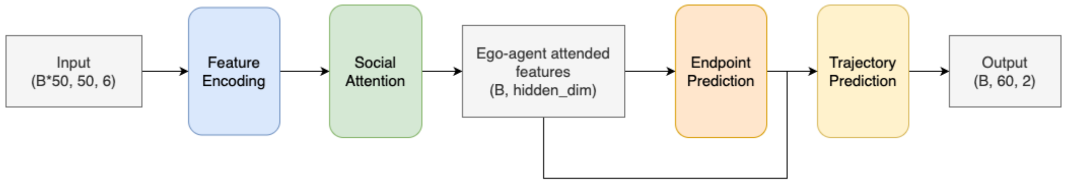

Figure 1: Our Model Architecture

4.1 Feature Encoder

To represent each agent's motion history, we employ a bidirectional LSTM encoder over the polyline trajectory of each agent. We reshape the input $\mathbf{X}$ to $(B \cdot N, T, C)$ and feed it into a bidirectional LSTM:

$$\mathbf{H} = \texttt{FeatureEncoder}(\mathbf{X}) \in \mathbb{R}^{B\times N\times d} \tag{3}$$

The final hidden states from the forward and backward directions are concatenated to obtain a fixed-size embedding for each agent's polyline. This representation captures temporal dynamics and direction-aware behavior, making it suitable for downstream interaction modeling. Here, $\mathbf{d}$ denotes the LSTM's output dimension (i.e., the sum of forward and backward hidden sizes).

4.2 Agent-Agent Interaction

We model social interactions among all N agents using a multi-head self-attention module. Given the embeddings $\mathbf{H} \in \mathbb{R}^{B \times N \times d}$, we apply attention as follows:

$$\mathbf{H}' = \mathbf{H} + \texttt{MultiHeadAttn}(\mathbf{H}, \mathbf{H}, \mathbf{H}) \in \mathbb{R}^{B \times N \times d} \tag{4}$$

This self-attention mechanism enables each agent to attend to all others in the scene, allowing the model to reason about joint behaviors such as merging, yielding, and collision avoidance. We include a residual connection to stabilize training and preserve the original agent features. Unlike a full Transformer block, we omit LayerNorm and position-wise feedforward layers for computational simplicity, as we observed no empirical benefit in our setting.

Finally, we extract the ego agent's context-aware representation from the updated feature tensor as:

$$\mathbf{h}_{\text{ego}} = \mathbf{H}'[:, 0, :] \in \mathbb{R}^{B \times d} \tag{5}$$

This vector serves as the foundation for downstream prediction modules.

4.3 Coarse-to-Fine Endpoint Prediction

To predict the endpoint of the ego agent's future motion, we adopt the coarse-to-fine endpoint prediction strategy, inspired by ADAPT [1]. Our goal is to first predict a rough estimate of the future endpoint and then refine it to improve precision, while decoupling the gradients between the two steps for better training stability.

We begin by applying a MLP, composed of a LayerNorm, a ReLU activation, and a linear projection layer, to the ego embedding:

$$\mathbf{f} = \texttt{MLP}(\mathbf{h}_{\text{ego}}) \tag{6}$$

This transformed feature $\mathbf{f}$ is used to produce a coarse endpoint prediction $\hat{\mathbf{e}}_{\text{coarse}} \in \mathbb{R}^{B \times 2}$ through a linear projection:

$$\hat{\mathbf{e}}_{\text{coarse}} = \mathbf{W}_1 \mathbf{f} + \mathbf{b}_1 \tag{7}$$

To improve upon this initial prediction, we generate a residual refinement offset $\hat{\mathbf{o}} \in \mathbb{R}^{B \times 2}$ from the original ego embedding using another lightweight MLP:

$$\hat{\mathbf{o}} = \mathbf{W}_2 \cdot \texttt{ReLU}(\mathbf{h}_{\text{ego}}) + \mathbf{b}_2 \tag{8}$$

The final refined endpoint is then obtained by adding the offset to the `detached` coarse prediction:

$$\hat{\mathbf{e}}_{\text{refined}} = \texttt{detach}(\hat{\mathbf{e}}_{\text{coarse}}) + \hat{\mathbf{o}} \tag{9}$$

We explicitly detach the coarse prediction from the computation graph to prevent gradients from flowing into the coarse branch during refinement. This design choice allows the refinement module to independently optimize the correction vector, avoiding undesired interference with coarse endpoint learning, which leads to more stable training and better endpoint localization.

4.4 Trajectory Decoder

To generate the final future trajectory, we concatenate the refined endpoint prediction $\hat{\mathbf{e}}_{\text{refined}}$ with the ego feature vector $\mathbf{h}_{\text{ego}}$. Before concatenation, we detach the refined endpoint to prevent gradients from flowing back through it. This concatenated vector is then passed through an MLP to regress the full trajectory:

$$\hat{\mathbf{Y}} = \texttt{MLP}\left([\mathbf{h}_{\text{ego}}, \texttt{detach}(\hat{\mathbf{e}}_{\text{refined}})]\right) \in \mathbb{R}^{B \times 60 \times 2} \tag{10}$$

The MLP consists of a ReLU activation followed by a linear layer that outputs a flat vector of size 60×2, which is reshaped into a sequence of 2D positions over 60 future timesteps. The output $\hat{\mathbf{Y}}$ represents the predicted trajectory of the ego agent in normalized coordinates.

4.5 Loss Function

The training objective combines two components: the full trajectory prediction loss and the endpoint prediction loss. The endpoint loss supervises the predicted coarse endpoint, while the trajectory loss encourages the predicted future sequence to match the ground-truth trajectory.

Let $\mathbf{Y} \in \mathbb{R}^{B\times 60\times 2}$ denote the ground-truth future trajectory, and we have the predicted trajectory $\hat{\mathbf{Y}} \in \mathbb{R}^{B\times 60\times 2}$.

Let $\mathbf{y}_{60} = \mathbf{Y}[:, -1, :] \in \mathbb{R}^{B\times 2}$ be the final position of the ground-truth trajectory.

The overall loss is defined as:

$$\mathcal{L} = \texttt{MSE}(\hat{\mathbf{Y}}, \mathbf{Y}) + \lambda \cdot \texttt{MSE}(\hat{\mathbf{e}}_{\text{coarse}}, \mathbf{y}_{60}) \tag{11}$$

where λ is a balancing weight (set to 1.0 in our experiments). Both terms use the mean squared error (MSE) averaged across the batch.

5 Experiments

Here, we'll describe different ways that we initially explored our problem, and approaches/architectures that were attempted. Ultimately, after exploring many different architectures, our ADAPT based method achieved a lower loss and lower Mean-Squared Error than all of the other approaches that we've tried.

5.1 Baselines

5.1.1 Baseline 1: Constant Velocity

We tried to run the constant velocity baseline algorithm to see how it works. Basically, it calculates the average velocity of the prediction agent during the first 50 time steps and it assumes that the prediction agent would move at a constant velocity for the next 60 time steps. The velocity is the distance the agent would move in both x and y directions in one time step and we calculate where the agent is for the next 60 time steps based on that average velocity and the position of the agent at time step 50.

$$(x, y)_t = (x, y)_{50} + (t - 50) \cdot \text{average velocity}, \quad \text{where } t \text{ is the time step}$$

We use MSE to calculate the training loss and it's around 54.1862. After submitting the predictions for test data to the leaderboard, the test error is around 53.02926.

5.1.2 Baseline 2: MLP

Afterwords, we tried a Multi-Layer Perceptron using ReLU activation functions and Dropout.

Our model starts with a layer that accepts $50 \cdot 50 \cdot 6$ inputs (which is the size of our initial input data, 50 agents per scene, 50 starting time steps, with 6 features per agent), into 1024 neurons. We then use ReLU after summing, and a dropout of 0.1.

We then have a second layer of 512 neurons with the same activation function and dropout after each neuron. Then, a third layer with a size of 256 neurons (with ReLU and Dropout), which lastly goes to our output layer of size $60 * 2$ (60 timestamps of an x and y coordinate for our ego agent).

5.1.3 Baseline 3: LSTM

We then use the default LSTM implementation initially provided to us, before moving onto other various experiments such as using encoder-decoder, state-space models, transformers, and other modifications of the basic LSTM architecture.

5.2 Evaluation

For all models, we evaluated our loss using Kaggle submission scores, as well as locally comparing the Validation Mean-Absolute-Error (L1) loss, and Validation Mean-Squared-Error loss of our different models.

For the pure training/validation loss of most of our baselines, we used MSE loss or WeightedMSE-MAELoss with an alpha of 0.4. We used a different loss function when training our final model (ADAPT) which did not allow for easy comparison on these fronts.

For ADAPT, we trained using ADAPT Loss, which is the MSE loss of the predicted endpoint of the trajectory compared with the ground truth endpoint of the trajectory, added to the MSE of the predicted and ground truth trajectories themselves.

For a more intuitive method of evaluation, we also visualized the predicted trajectories compared to the ground truth trajectories in matplotlib to tell if we were on the right track.

5.3 Implementation Details

For training and testing our ADAPT model, we used Google Colab with an A100 GPU, containing 40GB of VRAM, and 83.5GB of system RAM. On this machine, with our model it takes around 10s in order to train a single epoch.

To train our final ADAPT model, we used Adam with a learning rate of 0.001 with a "ReduceLROn-Plateau" scheduler. The scheduler has a patience of 10 epochs without improvement with a threshold of 0.001 before reducing the learning rate by a factor of 0.5. The scheduler has a learning rate floor of 0.0000001 where it no longer decreases. Our optimizer also used a weight decay value of 0.00005.

We added early stopping with a patience of 40, and as a result ended up training for 153 epochs (at 10 seconds per epoch).Our batch size was 32, and we tried a range of hyper parameter values. Mainly, we toyed with increasing the patience on our learning rate scheduler until it we stopped seeing additional value from it (patience of 1 thorugh 10), the factor by which to decrease the learning rate (0.9 to 0.1) and the starting learning rate (from 1e-1 to 1e-5). We also experimented with weight decays between 1e-4 and 1e-5.

The tuning for the given above was for our ADAPT model, but tuning hyperparameters was different for baselines. For example, when we thought we were potentially overfitting (high validation loss with lower training loss) on a MLP model, we increased weight decay significantly more than we had to for our ADAPT model. Another example is that our transformer decoder model ended up seeing its best performance with a learning rate of 1e-4, a 10x difference compared to our ADAPT learning rate. Our patience was increased for our ADAPT method compared to baselines as well (baselines had a patience of 1), simply because we witnessed better success using this model.

5.4 Results

5.4.1 Quantitative Results

After training our model for 153 epochs and we ended with a training loss of 0.4264, a validation loss of 0.7924, a validation MAE of 1.2681, and a validation MSE of 7.1673. Figure 2 shows how the training and validation loss (using the ADAPT loss) decreased over the epochs.

After predicting the test data on our trained model, our model got a score of 7.22861 on the public test set and a 7.25389 on the private test set. This led to a leaderboard ranking of $6th$ and $7th$ place for our group in the competition.

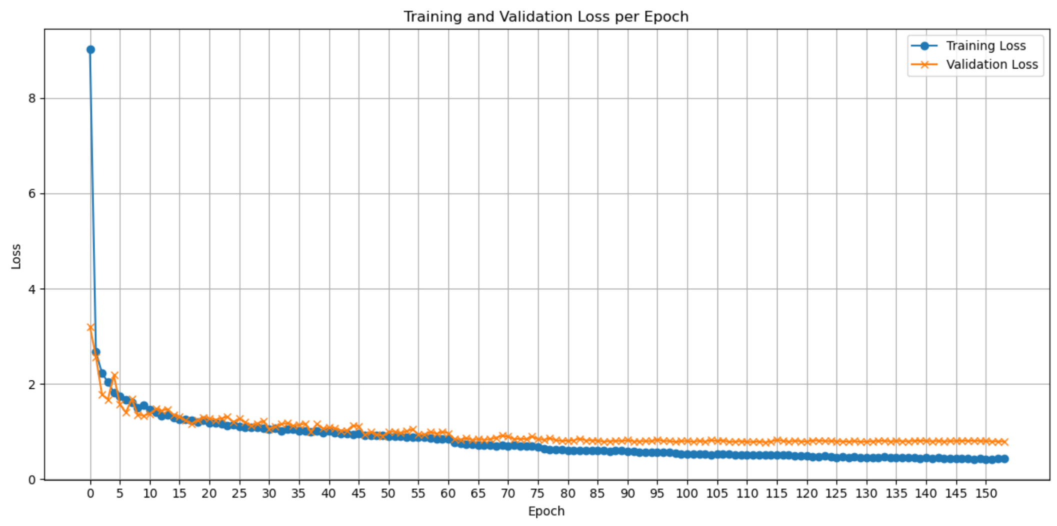

Figure 2: Training and Validation Loss per Epoch

5.4.2 Qualitative Results

We also visualized the predictions of our model and looked at case studies of where the model performed well or incorrectly. Looking at Figure 3, we can see cases where the model performed correctly. We see that the model correctly follows trajectories that move in a straight line, regardless of orientation. It also correctly recognizes wide turns and moves in the general direction to mimic the curvature of the turn.

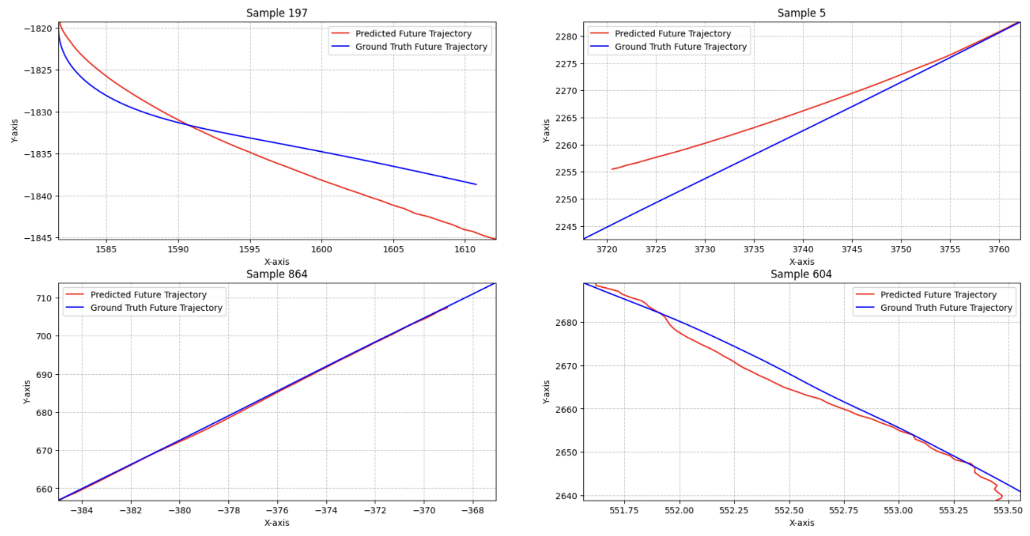

Figure 3: Case Studies where Model is Successful

Figure 4, we can see cases where the model performed poorly. We can see that the model predicts imprecise and high-variance trajectories in scenes where the vehicle does not move and it often does not come to a stop, in scenes where the ego vehicle comes to a stop in the scene. When looking at turns, it does not accurately follow sharp turns and often under-turns compared to the ground truth.

These case studies help us understand how our model performs in the context of the problem and understand its trajectory forecasting shortcomings in a real-world scenario.

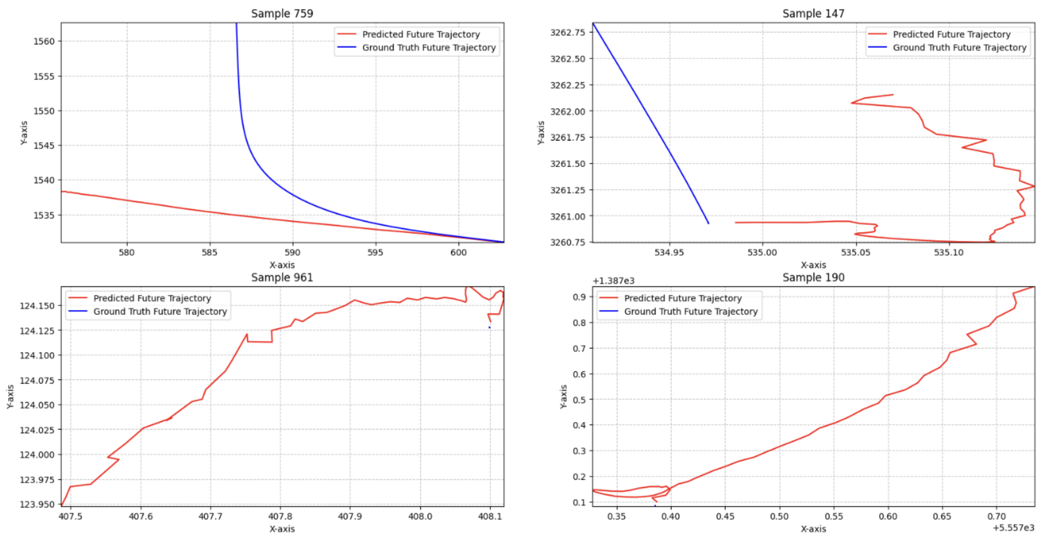

Figure 4: Case Studies where Model needs Improvement

5.5 Ablations

One of our models that we were experimenting with was an LSTM Decoder with a Transformer decoder. We had plenty of twists on this architecture, but for this section we will focus on testing the impact of adding a temporal attention mechanism to a version of this architecture which already contained social attention pooling. We compared this baseline social-attention model against our experimental one, keeping all hyperparameters and training configurations identical to ensure a fair evaluation. The baseline architecture used an LSTM to encode each agent's trajectory, followed by a social attention module to model interactions. The experimental architecture inserted a multi-head temporal attention layer after the LSTM to dynamically weigh the importance of past timesteps for each agent before the social attention stage.

Contrary to our expectations, the addition of the temporal attention module degraded performance. The baseline model achieved a private Kaggle leaderboard score of 8.79274, while the experimental model scored a higher (worse) 9.81890. This result suggests that the increased complexity of the temporal attention layer led to overfitting or introduced noise from less relevant, distant past states. The simpler heuristic of the baseline model, which focused on more recent history, proved to be a more robust and effective strategy. This study demonstrates that increasing architectural complexity does not always yield better performance, and simpler, more targeted mechanisms can be superior.

6 Conclusion

In this project, we learned the challenges of trajectory prediction and the design of deep learning models for real-world forecasting tasks. Our experiments demonstrated that simple models, such as linear regression and basic multilayer perceptrons (MLPs), are inadequate for capturing the temporal and social dynamics in multi-agent traffic environments. The most effective techniques we found were those that modeled the interactions between agents. Integrating attention mechanisms(multi-head self-attention) significantly improved the model's capacity to foresee socially plausible trajectories. In addition, the ADAPT-inspired approach, which predicts a coarse endpoint before refining the full trajectory, proved highly effective. It improved helped stabilize training by decoupling endpoint prediction from full trajectory generation. One surprising finding was that adding useful features, such as acceleration, did not necessarily improve performance. These additional features introduced noise and made the model less stable in training. We also observed that data augmentation strategies, such as random rotations and flips, helped to improve generalization in unseen scenarios. For beginners' advice, we suggest starting with simple sequential models, such as LSTMs, to capture temporal dependencies and then moving to more complex architectures that model agent interactions. Additionally, attention mechanisms are highly beneficial, and integrating intermediate prediction targets (like endpoints) can stabilize and improve long-horizon predictions.

6.1 Limitations

For our work's limitations. First, we did not combine with external map information, such as lane boundaries or traffic signals, which could potentially provide helpful information for trajectory prediction. Then, our models still struggle with rare, complex scenarios, such as sudden stops or aggressive maneuvers, at times. As a result, the project highlighted the importance of particular model design, interaction modeling, and iterative experimentation in developing effective deep learning models for autonomous trajectory prediction.

6.2 Future Work

Future work can explore several areas. We can incorporate high-definition map features, such as lane boundaries and traffic signals, which could enhance contextual understanding and improve prediction accuracy. Next, we can build more advanced models, for example, using graph-based interaction networks or relation-aware attention, which may lead to better representations of agent-to-agent dynamics. Another approach is optimizing models for real-time deployment on edge devices. Finally, we can also enhance the model's ability to handle rare, critical events, such as sudden stops or aggressive maneuvers.

7 Contributions

Jakob Getzel: Designed and implemented the State-Space Model (S4) experiments with different approaches/decoders. Worked on milestone, presentation, and report.

Jiayin Meng: Designed and implemented the final model architecture; Explored multiple modeling approaches; Conducted training, evaluation, and hyperparameter tuning; Worked on milestone, presentation, and report.

Amaar Valliani: Explored Mamba model approach; Worked on milestone, presentation, and report; Defined the problem, explored output of models, and evaluated strengths and weaknesses.

Yule Zhang: Explored and experimented the Graph Neural Network model and different encoders/decoders. Worked on milestone, presentation, and report.

References

[1] Görkay Aydemir, Adil Kaan Akan, and Fatma Güney. Adapt: Efficient multi-agent trajectory prediction with adaptation, 2023.

[2] Ye Yuan, Xinshuo Weng, Yanglan Ou, and Kris Kitani. Agentformer: Agent-aware transformers for socio-temporal multi-agent forecasting, 2021.

[3] Karan Goel Albert Gu and Christopher R ´e. Efficiently modeling long sequences with structured state spaces, 2021.