CSE 251B Project Final Report

https://github.com/amritamo/trajectory-prediction

Shayan Boghani	**Amrita Moturi**	**Amit Namburi**	**Iris Zhou**
UCSD	UCSD	UCSD	UCSD
sboghani@ucsd.edu	amoturi@ucsd.edu	anamburi@ucsd.edu	irzhou@ucsd.edu

Abstract

Accurate prediction of future vehicle trajectories is important for safe and reliable autonomous driving. In this work, we propose a Transformer-based model that improves upon standard LSTM baselines by incorporating richer spatiotemporal context and multi-agent interactions. Our model architecture incorporates the data of all 50 agents in the scene using temporal encoders and multi-head self-attention. We introduce an enhanced normalization pipeline that aligns trajectories to the ego agent's heading and decomposes velocity components, resulting in more stable and generalizable training. Additionally, we incorporate a validity mask to handle padded or missing agent data, increasing robustness in real-world settings. Our results demonstrate improved prediction accuracy and reduced error compared to the LSTM baseline, highlighting the effectiveness of our approach on the Argoverse 2 dataset. Our final private Kaggle score was 8.17635 at rank 17, under the name "Autobots Roll Out!!".

1 Introduction

The task is to build a model that is trained on the Argoverse 2 Motion Forecasting Dataset [1] to predict future trajectories of an autonomous vehicle (ego-vehicle) and other surrounding actors which are present in the scene to represent real-life driving scenarios. The training data consists of 10,000 scenes, each with 50 agents' trajectories, and their individual x and y positions, component velocities, and heading angle over 110 timesteps. Specifically, the task involves using the first 50 timesteps of each agent's behavior to make accurate, multi-step predictions over the next 60 timesteps. By accomplishing this task, we contribute to research in motion forecasting, which is an essential part of safe and reliable autonomous driving. Accurately anticipating the behavior of nearby agents is critical for navigation in dynamic environments. This dataset also captures various traffic conditions in diverse cities and environments, which brings us closer toward developing autonomous systems that generalize well and can be deployed at scale.

The starter code implements a basic LSTM model for predicting the future trajectories of agents and includes a data preprocessing step to normalize the input positional data using a scaling factor and the ego agent's last given position. Data augmentation is also applied to the training set to introduce random flipping and rotations, allowing the model to learn how to predict regardless of the orientation. Shortcomings of the starter code are that only positional normalization is applied, which our model addressed by normalizing based on the ego's heading angle and component velocities as well. Since we implemented a transformer model, we also take advantage of the self-attention mechanism to incorporate all 50 agents to better learn multi-agent interactions, rather than the starter code which can only focus on ego's motion patterns. Additionally, the starter code applies a flat LSTM structure that treats the sequence as a linear time series without considering spatial relationships between agents. We improved upon this by encoding each agent's trajectory independently with a temporal MLP

Preprint. Under review.

encoder, followed by positional encodings and a stacked Transformer encoder that allows dynamic interaction modeling across agents. Finally, we incorporate a mask to indicate which agents are actually present and active in each scene, to distinguish between zero-padded agents, ensuring that the model attends only to valid agents. This makes our model more robust to sparse or incomplete data compared to the baseline LSTM.

To reiterate, our main contributions are:

- Contribution 1: Stronger modeling of multi-agent interactions through self-attention over all agents.
- Contribution 2: Improved model architecture using a deep Transformer encoder with temporal feature encoding, positional embeddings, and residual connections for stronger spatiotemporal modeling.
- Contribution 3: Learns more generalizable representations using additional normalization and augmentation of heading angle and component velocities, aligning input data to ego as the reference.
- Contribution 4: Increased robustness to padded data by incorporating a validity mask, which filters out inactive agents during attention computation and prevents overfitting to noisy inputs.

Our final private Kaggle score was 8.17635 at rank 17, under the name "Autobots Roll Out!!".

2 Related Works

Trajectory forecasting is widely studied in the context of autonomous driving, with increasing focus on multi-agent interactions and temporal reasoning as machine learning models have become more advanced in their ability to embed these features. Argoverse 1 [2] was initially introduced as a large-scale benchmark for 3D tracking and trajectory forecasting, providing richly annotated map data and realistic urban driving scenarios. The Argoverse 2 Motion Forecasting Dataset was later developed to enhance training and validation of the models to incorporate scenarios that were approximately twice as long and more diverse [1].

The introduction of LSTM networks brought about significant achievements in trajectory prediction due to the ability to embed temporal data which is critical in modeling time-series data which is used in motion forecasting problems. Work by Alahi et al. proved that Social LSTMs able to capture interactions between agents through a pooling layer. Work by Tang et al. [3] integrated multimodal prediction, generating diverse possible futures. Later approaches leveraged structured scene representations and spatial reasoning. Zeng et al. [4] proposed LaneRCNN, a graph-centric framework incorporating lane features through object-level representations. Additional work using hierarchical graph networks was done by LaneGCN, which was able to model continuous agent dynamics between actors, maps, and lanes [5].

Other works focused on spatiotemporal encoding using point-based or proposal-based architectures. TPCN [6] applied temporal point cloud networks to capture evolving motion patterns, and TPNet [7] utilized trajectory proposals as anchor-based predictions. VectorNet [8] encoded both agent dynamics and high-definition maps using a vectorized representation, enabling fine-grained reasoning about road layouts and interactions.

3 Problem Statement

Let N be the number of scenes, $A = 50$ the number of agents per scene, $T_{\text{in}} = 50$ the number of input timesteps, $T_{\text{out}} = 60$ the number of future timesteps to predict, $D_{\text{in}} = 6$ the number of features per input timestep, and $D_{\text{out}} = 2$ the number of features per output timestep.

The input tensor is defined as:

$$\mathbf{X} \in \mathbb{R}^{N \times A \times T_{\text{in}} \times D_{\text{in}}}$$

where each vector $\mathbf{X}_{n,a,t,:}$ contains the features:

$$(x, y, v_x, v_y, \theta, \text{object}) \in \mathbb{R}^6$$

The output tensor is:

$$\mathbf{Y} \in \mathbb{R}^{N \times A \times T_{\text{out}} \times D_{\text{out}}}$$

where each vector $\mathbf{Y}_{n,a,t,:} = (x, y) \in \mathbb{R}^2$ represents the predicted future position of agent a at timestep t in scene n.

We define the prediction task as the function:

$$f : \mathbb{R}^{A \times T_{\text{in}} \times D_{\text{in}}} \rightarrow \mathbb{R}^{A \times T_{\text{out}} \times D_{\text{out}}}$$

which maps the observed agent trajectories in a single scene to predicted future positions.

The training data shape is $(10000, 50, 110, 6)$, which represents 10,000 scenes which track 50 agents over 110 timestamps across 6 dimensions. Each timestep represents 0.1s of activity. The dimensions include the x-coordinate of the agent's position, y-coordinate of the agent's position, x-component of the agent's velocity, y-component of the agent's velocity, heading angle of the agent in radians, and the encoded object type.

The test data shape is $(2100, 50, 50, 6)$, representing 2,100 different scenes with 50 agents each over 50 timesteps across the 6 dimensions previously mentioned. The model's input dimension will be $(50, 50, 6)$, representing the 50 agents for 50 timesteps, and 6 features per timestep. The model outputs predictions with shape (50, 60, 2), representing future x and y positions for each of the 50 agents over the next 60 timesteps. The input x values range from -9279.713 to 13261.213 and input y values range from -4581.698 to 6653.930.

3.0.1 Positional data

Fig. 1 and 2 show the distribution of input and output positions of all agents over the first 50 and the last 60 timesteps respectively. Visually, the two plots look fairly similar. The distributions of positions in both plots may be highlighting the locations of roads, with brighter areas indicating intersections where more traffic or congestion may occur. The positional data span is quite vast, and as such, it may be wise to pre-process the data by normalizing all of the positions. This will make it easier for the model to predict the position.

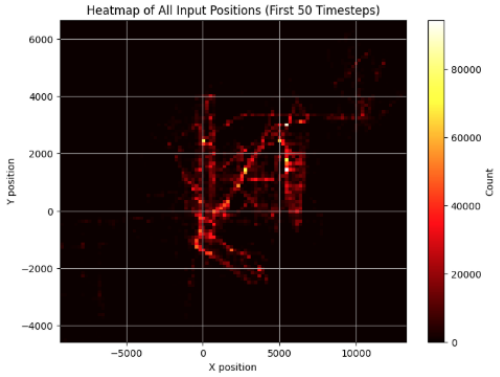

Figure 1: Distribution of input positions for all agents (first 50 timesteps)

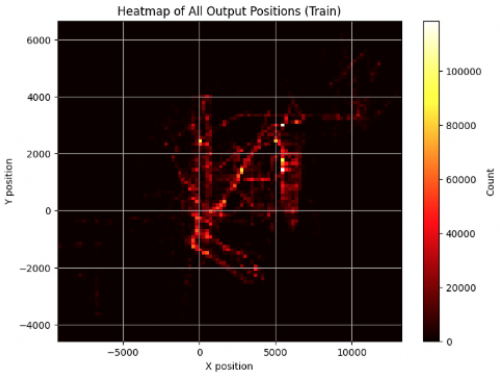

Figure 2: Distribution of output positions for all agents (next 60 timesteps)

3.0.2 Agent data

Looking more closely at the agent types represented in the data provides more insight into how agents may be interacting within and across training scenes. There are 10 unique object classes, including two sub-categories of dynamic and static objects.

Fig. 3 shows the mean distribution of agent types over all training scenes. We can see that vehicles compose the overwhelming majority of agents across scenes, with an average of about 48 vehicles out of 50 total agents. Pedestrians and unlabeled static objects mostly frequently make up the remainder of the agents in the scene, though this varies across scenes, as shown by the error bars.

This distribution provides some context for the input and output positions shown in Fig. 1 and 2, that most agents are vehicles continuously moving along clearly defined roads in the scene.

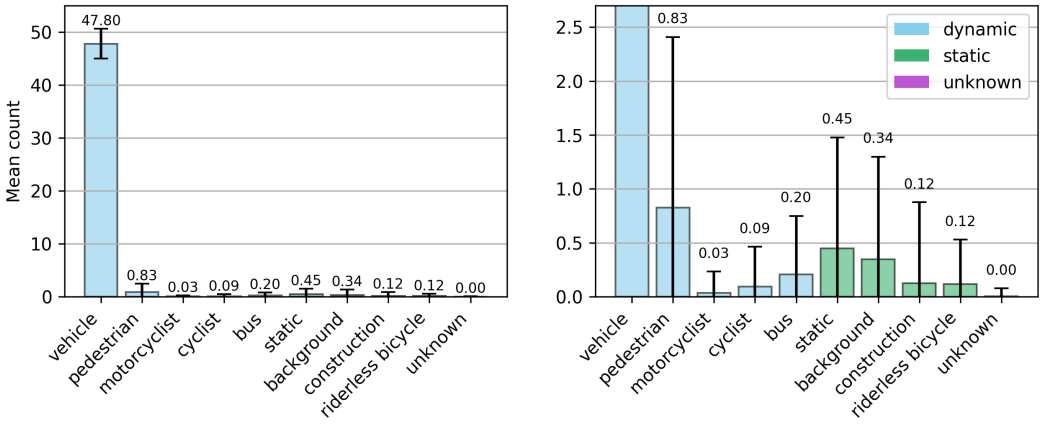

Figure 3: Mean distribution of agent types over all 10000 training scenes, with counts above each bar and standard deviations displayed as error bars. Y-axis of right plot is clipped to show counts of non-vehicular agent types.

The data description provided on the competition page also notes that incomplete trajectories or missing agents may result in zero padding in the data. To gain a better understanding of how much this impacts our data, Fig. 4 shows the mean number of active agents across timesteps in a scene. We can see that the number of active agents tapers toward the beginning and end of each scene, but reaches the maximum number of 50 agents toward the 40th timestep.

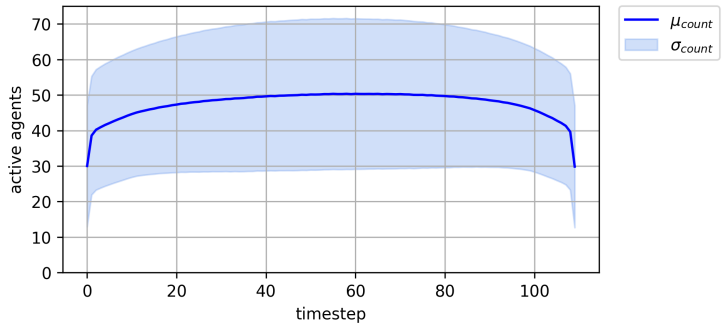

Figure 4: Mean number of active agents over scenes over 110 timesteps, with standard deviation shown as error bands. The number of active agents is computed based on whether an agent's positional data is zero.

3.0.3 Movement of objects

Analyzing the velocities of object types can provide more insight into patterns in motion and inform our models assumptions. Fig. 5 shows the mean speeds of object types over the course of a scene. We see that some dynamic objects, including buses, cyclists, and pedestrians appear to move at a constant speed over the course of a scene. Other dynamic objects like motorcyclists and vehicles may speed up during a scene, though this may be influenced by the presence of zero-padding for incomplete trajectories. Static objects, including construction and riderless bicycles, display little to no movement with some noise as expected.

This analysis confirms that classification of 4 of the object types as being static. To reduce the complexity of the information fed to our models, we could replace the object type indices of static and unknown objects with a single number. For the dynamic objects, we can see that their movement velocity remains fairly distinct from one another, so it makes sense to keep their labels in the "object type" dimension.

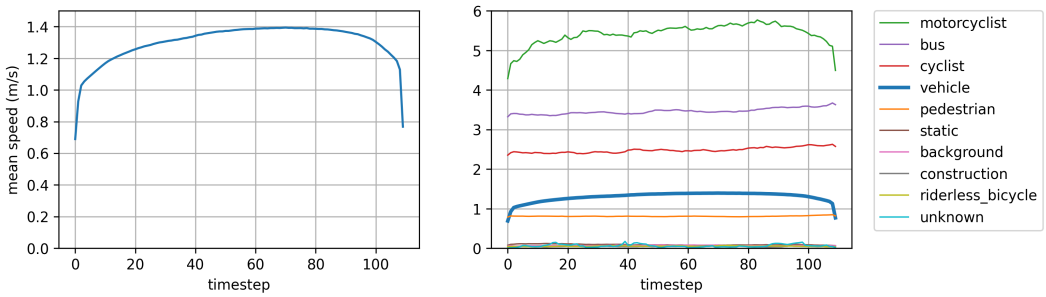

Figure 5: Mean object speeds (m/s) over scenes over 110 timesteps, obtained through object x and y-velocities. Left plot shows just vehicle speed, including speeds of ego vehicles. Right plot shows all objects.

4 Methods

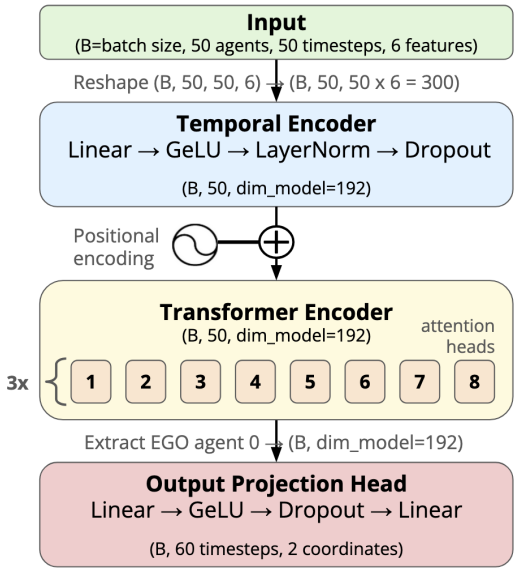

Figure 6: Transformer model architecture

The chosen model was a Transformer-based model, taking in our pre-processed data and outputting the ego prediction for the final timestep. The structure of the model can be seen in Figure 6. This model is based off a simple Transformer Encoder architecture. The reason for choosing this architecture was due to the sequential data format as well as the potential interaction between agents.

The prediction task can be formulated as an optimization problem, where the objective (loss) function is as follows:

$$\mathcal{L}(\theta) = \frac{1}{B \cdot P} \sum_{i=1}^{B} \sum_{j=1}^{P} ||\hat{y}_{i,j} - y_{i,j}||^2$$

Where: θ are the model parameters, B is the batch size, P is the number of predicted steps, $\hat{y}$ is the predicted ego position, and y is the ground truth ego position.

4.1 Pre-processing

To evaluate our model's performance, we randomly split the training data into 90% for training and 10% for validation. This split was controlled using fixed random seeds to ensure reproducibility. Training batches were shuffled, while validation data was kept in a fixed order for consistency during evaluation.

Unlike the starter code, which only normalizes spatial positions using a fixed scale, our model applied additional normalization based on the component velocities, such that The v_x and v_y components were normalized independently. Additionally, the heading alignment was normalized such that input features were rotated to align with the ego agent's final heading angle, providing a consistent local reference frame. A normalization factor of 3.0 was also empirically chosen and applied to all spatial features to stabilize training.

During evaluation, the predicted trajectories were denormalized by reapplying the inverse of these transformations as shown in Fig. 7.

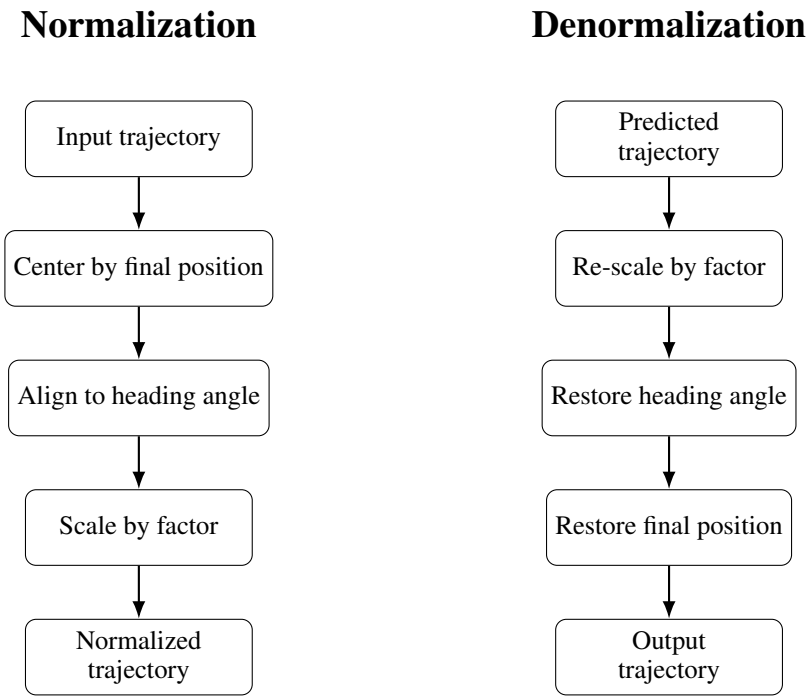

Figure 7: Normalization and Denormalization pipeline applied to input and predicted trajectories.

4.2 Temporal encoding

One key aspect of the data was its time series reliance. The model had to accurately capture these relationships so that it could learn representations that accurately reflected the training data. To do this, a temporal encoder was used. The input historical trajectory data for each agent was flattened and fed into a feedforward encoder to project it into a fixed-dimensional embedding. This is done for all agents.

4.3 Positional encoding

The model also had to capture the position of each agent in a scene. This cannot be done inherently, and as such, a positional encoder can be used. Sinusoidal functions with different wavelengths are used to ensure a unique encoding for each position and helps with deriving relationships between agents and between an agent and the scene. Figures 8 and 9 show the equation for the positional encoders.

$$\mathrm{PE}_{\text{pos},\,2i} = \sin\left(\frac{\text{pos}}{10000^{\frac{2i}{dim_{\text{model}}}}}\right)$$

Figure 8: Even Positional Encoder

$$\mathrm{PE}_{\text{pos},\,2i+1} = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{dim_{\text{model}}}}}\right)$$

Figure 9: Odd Positional Encoder

4.4 Transformer encoder

The transformer encoder is the main part of the model. This set of layers allows for the attention between agents and the model to learn these relationships. A single encoder is used for all the agents and uses a GELU activation function with layer normalization in between layers. Our encoder consists of 3 layers, each with 8 attention heads.

4.5 Output layer

Finally, the ego agent's future trajectories are extracted from the transformer encoder output. This is the 60 time steps for the future trajectory of which the last value is used for evaluation. The output layer is a simple feedforward network which uses a GELU activation and dropout.

5 Experiments

5.1 Baseline

All baseline models were preprocessed by normalizing their position data with respect to EGO vehicle's position at last input timestep, and scaling position and velocity data (dividing) by $\lambda = 10.0$.

- ◇ **Linear Regression:**
  - **Architecture:**
    - Input (batch, 50 agents, 50 timesteps, 2 coordinates)
    - Reshape (batch, $50 \times 50 \times 2 = 5000$)
    - Linear layer ($5000 \rightarrow 120$)
    - Output linear layer (60 timesteps $\times$ 2 coordinates)
  - **Parameters:**
    - N/A

- ◇ **Multi-layer perceptron:**
  - **Architecture:**
    - Input (batch, 50 agents, 50 timesteps, 6 features)
    - Reshape (batch, $50 \times 50 \times 6 = 15000$)
    - Linear layer 1 ($15000 \rightarrow 1024$) $\rightarrow$ ReLU $\rightarrow$ Dropout
    - Linear layer 2 ($1024 \rightarrow 512$) $\rightarrow$ ReLU $\rightarrow$ Dropout
    - Linear layer 3 ($512 \rightarrow 256$) $\rightarrow$ ReLU $\rightarrow$ Dropout
    - Output linear layer ($256 \rightarrow 60 \times 2 = 120$)
    - Reshape (60 timesteps, 2 coordinates)
  - **Parameters:**
    - Dropout (p = 0.1)

◇ **MultiAgent LSTM:**

- **Architecture:**
  - Input: (batch, 50 agents, 50 timesteps, 6 features)
  - Process each agent through shared LSTM: ($6 \rightarrow 128$)
  - Extract last hidden state for each agent
  - Apply validity mask to zero out inactive agents
  - Concatenate agent features: (batch, 50×128)
  - Fuse with MLP: $6400 \rightarrow 256 \rightarrow 128$
  - Output: Linear ($128 \rightarrow 120$)
  - Reshape: (60 timesteps $\times$ 2 coordinates)
- **Parameters:**
  - Hidden size: 128
  - Dropout (p = 0.1)

5.2 Final Transformer Based Model

All input data was preprocessed via ego-centering, heading alignment, and normalization (factor $\lambda = 3.0$). Future predictions are generated only for the ego agent (index 0).

◇ **Transformer:**

- **Architecture:**
  - Input: (batch, 50 agents, 50 timesteps, 6 features)
  - Flatten each agent's history: $(50 \times 6 = 300)$
  - Feedforward encoder: Linear $\rightarrow$ GELU $\rightarrow$ LayerNorm $\rightarrow$ Dropout
  - Add sinusoidal positional encoding (over agents)
  - 3-layer Transformer Encoder (192 dim, 8 heads)
  - Output: feedforward projection (ego agent only): $192 \rightarrow 96 \rightarrow 120$
  - Reshape: $(60$ timesteps $\times 2$ coordinates$)$
- **Parameters:**
  - Model dimension: 192
  - Attention heads: 8
  - Transformer layers: 3
  - Feedforward dim: 384
  - Dropout: 0.05

5.3 Evaluation

Our models and baselines were evaluated based on the Mean Squared Error (MSE) of the validation and test set given the predicted and ground truth positions for the output 60 timesteps. Formally, we define this in the equation below. This equation is adapted from the loss function defined in Methods.

$$MSE = \frac{1}{60N} \sum_{i=1}^{N} \sum_{t=1}^{60} [(\hat{x}_{i,t} - x_{i,t})^2 + (\hat{y}_{i,t} - y_{i,t})^2]$$

The variables are defined as follows:

- N = number of scenes in dataset
- $(\hat{x}, \hat{y})$ = predicted coordinates
- (x, y) = ground truth coordinates

We use the same training/validation/test split across all models, using Kaggle to obtain the test MSE. Additionally, simple qualitative evaluation was done for the models to identify weak points. This meant visually inspecting trajectories and determining which types of trajectories the model performed the worst on, and determining ways to improve this performance.

5.4 Implementation Details

Computational platform: We initially used Google Colab's T4 GPU to do model training, however, we ran into memory issues and GPU usage limits. This led us to JupyterLab from UCSD's Data Science/Machine learning Platform (DSMLP), which provides 1 GPU.

Training time: Our final model trained for 186 epochs, with each epoch taking approximately 14.5 seconds to complete on average.

Optimization and scheduling: For training, we used the Adam optimizer with a learning rate of 1×10^{-3} and weight decay of 1×10^{-4}. A batch size of 32 was chosen to balance memory efficiency and gradient stability. Models were trained for up to 200 epochs, with early stopping triggered after 30 consecutive epochs without validation improvement. We used a StepLR scheduler to decay the learning rate by a factor of 0.25 every 20 epochs to facilitate convergence.

Design choices: Our design choices were based on the initial starter code, as well as optimization searches done from an intuitive baseline. For example, the learning rate was chosen based on its common usage in deep learning applications as being a good middle ground to prevent divergent behavior while still converging in reasonable time. Other values, such as the normalization scale were empirically tested throughout our model iterations. Initially, testing was done to see if larger or smaller values were better compared to the initial value provided (7.0). After training a few models, it was found that smaller scale values tended to improve performance so a series of values were tested to find the optimal for the model architecture (3.0). Batching was chosen due to computational limitations, stability, stochasticity, and generalization properties. For our optimizer, we tested both RMSProp and Adam and found Adam to perform slightly better in our architecture.

We did not do an exhaustive grid search for hyperparameter tuning due to computational constraints, however, we experimented with batch sizes $16, 32, 64, 128$ and learning rates $1e$-2, $1e$-3, $5e$-3, $1e$-4 before settling on the final configuration. Additionally, we experimented with the scaling value for the dataset to determine the best scale factor for the normalization step which came out to 3.0. These settings worked consistently well for both the LSTM baseline and our Transformer-based model.

Gradient clipping with a max norm of 5.0 was also applied to prevent exploding gradients, especially in earlier epochs. We ensured that all models were trained under comparable conditions for fair evaluation.

5.5 Results

The results of the tested models can be seen in Table 1. While these were not the only models tested, they highlight the performance gap between simpler baselines and more advanced architectures.

The MultiAgent LSTM notably outperforms the MLP and constant velocity baselines, demonstrating the benefit of modeling multiple agents jointly rather than in isolation. However, its performance still trails behind the Transformer, which shows superior generalization and accuracy across both validation and test sets.

Figure 10 shows sample predictions generated by the Transformer. Unlike the baselines, which often fail to model even simple linear or turning motions, the Transformer consistently/reasonably captures complex multi-agent dynamics and trajectory curvature.

The Transformer's performance is more stable across diverse scenes, indicating that the ego-aligned normalization and self-attention across agents contribute significantly to robustness and adaptability.

Model Comparison Summary

Model	Val MSE (local)	Test MSE (public)	Remarks
Constant Velocity	53.66	53.03	Baseline only
Linear Regression	>1.1M	>1.2M	Baseline only
MLP	290.75	290.53	Struggles with generalization
MultiAgent LSTM	8.5523	8.94334	Competitive, better than MLP baseline
Transformer	**7.8**	**8.215**	Best-performing

Table 1: Model performance comparison across validation and public leaderboard test MSE scores.

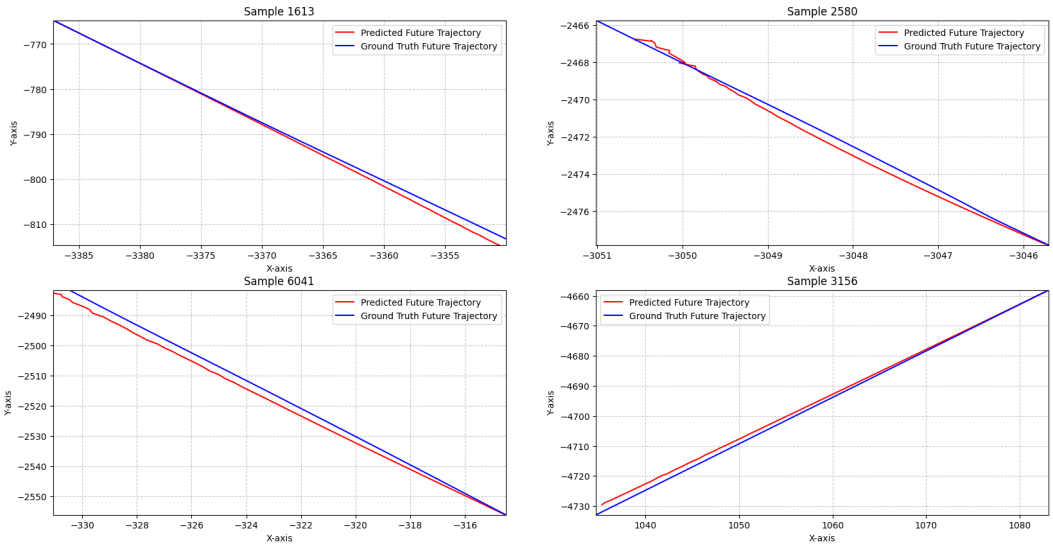

Figure 10: Sample ground truth vs. predicted trajectories using normalized transformer model.

5.6 Ablations

The final model contained a few key components which improved performance significantly. As such, these were the chosen parameters for the ablation study. Table 2 summarizes the results of this study, with the validation MSE of the models displayed in Figure 11. All hyperparameters were kept the same as what is found in the Implementation Details section. Validation MSE may vary from previous reports, as our final Kaggle submission was trained on the entire training set, while the results below use a train/validation split with split=0.9/0.1.

It's also worth noting that due to lack of computing resources, we were not able to perform k-fold cross-validation for more stable results. This means that differences between performance in models could also be due to the inherent stochasticity of model training, although all models were trained using the same hyperparameter and optimization settings.

Model/Changes made	Val MSE
Transformer w/ Full Normalization**	8.0997
ReLU	8.2027
No positional encoding	8.4801
No validity mask	8.1135
Only normalize positions, scale positions and velocity by $\lambda = 10$ (from sample)	10.6050

Table 2: Comparison of local validation accuracy on same train/val split. All parameters and settings other than mentioned changed are kept the same. **Transformer w/ Full Normalization described in Methods section above. Uses GeLU, positional encoding, scale=3.0, normalizes all features, and applies validity mask.

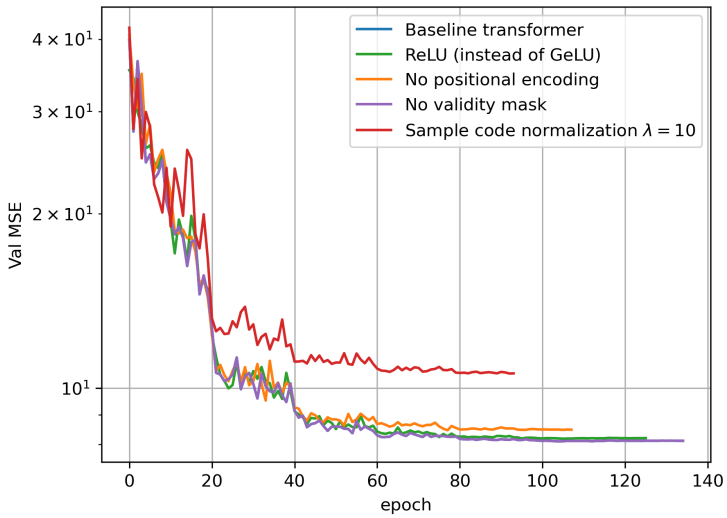

Figure 11: Validation MSE over training epochs for variations on baseline transformer model. Differences between baseline transformer model with ReLU and GeLU are negligible, so baseline performance is hidden under ReLU performance. Y-axis is shown with log-scale.

The first table entry is the best performing model. This can be considered the ablation study's "baseline". The first step was to change the activation function of the model. ReLU is a common activation function used in neural networks, and seemed a good nonlinearity to introduce instead of GELU, the activation function used in the final model. When introduced, ReLU saw model performance drop significantly. This may be because of ReLU's clipping for values less than zero to zero. For any values that were negative, which is common for the dataset, ReLU likely clipped them possibly leading to the "dying ReLU" problem.

Next, the positional encoder was removed. The positional encoder holds key information about position of agents in the scene. As such, this removal was known to potentially cause significant loss. The results show performance drop from 8.0997 to 8.4801. This shows that the model learned significant relationships from the positional information, both for each agent and between agents.

Additionally, we tested the performance of the model without using a validity mask. The validity mask is crucial to informing the model of "false" data introduced by zero-padding for incomplete trajectories or absent agents. Without using a validity mask in the transformer, we saw a small drop in performance from 8.0997 to 8.1135. The comparable performance could be due to randomness in the training process. It is also possible that the impact of zero-padding is negligible, only occurring for a few time steps at the beginning and end of a trajectory as shown in Figure 4.

Finally, the normalization and scaling was reverted back to the sample normalization provided for the baseline. This included normalizing the agent positions and scaling the agent positions and velocities. This saw the greatest drop in performance. The need for normalization arises from how many network architectures work. The dataset which we are using has varying magnitudes for the data, making it hard for a model to learn representations. The solution to this is normalization and scaling, which pushes these varying values into a tighter window and allows for the model to learn quicker. When we remove this normalization, the model now has to adjust weights drastically to attempt to output a value close to the input. This is quite difficult and leads to degraded performance, as demonstrated in the results of reverting our normalization.

From our ablation study, we find that the most impactful aspect of our model training was data normalization and scaling. This led to the greatest improvement in loss, and impacted other features as well.

6 Conclusion

Throughout this project, we found that thoughtful data pre-processing is critical to model performance. In particular, normalizing not just positions, but also aligning heading angles and decomposing velocity components significantly improved our MSE loss by ensuring consistent input representations across diverse driving scenes. Insights from exploratory data analysis (EDA) also played an important role in guiding our design decisions. For example, identifying that zero-padded agents could confuse the model prompted us to explore more robust encodings for stationary or missing agents, such as using a validity mask to filter them during attention computation.

We also discovered that the encoding strategy had a substantial impact on model performance. Encoding one token per timestep for a single agent preserves rich temporal dynamics, which is useful for learning individual motion patterns. In contrast, encoding one token per agent across all timesteps captures interaction between agents but may dilute temporal structure. Understanding the trade-offs allowed us to iterate on our architecture and make changes to account for temporal dynamics and multi-agent interaction considerations.

For deep learning beginners approaching similar prediction tasks, we recommend focusing first on carefully understanding the data inputs and outputs, and how missing data is dealt with before experimenting with complex model architectures. Even simple models like the baseline LSTM can perform well with intentional preprocessing. After establishing a decent baseline, we can move forward with implementing more complex architectures like transformers. Additionally, implementing a clear evaluation loop, visualizing predictions, and monitoring overfitting early in training is very helpful to know how your model is doing throughout the process.

6.1 Limitations

Although out Transformer-based model has good performance, there are still several limitations:

- **Lack of map/lane context:** Our model relies only on agent trajectories (past) and does not use map or lane graph features, which are generally very useful and informative for overall scene understanding in complex scenarios. This could potentially be generated based on global trajectories of agents across all scenes.
- **Fixed input window:** The model has a fixed 50 timestep input window which may not adapt well to variable length sequences or forecasting scenarios / actual live use cases.
- **Single agent prediction:** Only the ego agent's trajectory is predicted, which simplifies the overall task but ignores the use of joint prediction for all agents which might give it more information regarding the future time steps coming up.
- **One shot prediction:** Predictions for a trajectory are made using all information and all future time steps are predicted. This can lead to propagating bias which can lead to divergence. The use of shorter prediction frames could improve the outcome of the trajectory prediction for the final timestep.
- **Computational cost:** Transformer layers produce very high memory and computation overhead. When scaling to more agents or live use case, it might involve a lot of computational cost.
- **Limited training time:** Due to compute constraints, we trained for fewer epochs than ideal and did not perform full hyperparameter sweeps or grid search to comprehensively evaluate the best parameters. Additionally, we were not able to perform extensive k-fold validation testing to get the best possible results from our training.

6.2 Future Work

We aim to explore several strategies to improve our model's performance in the trajectory prediction task. One example of this is feature augmentation. In future iterations of our model, we plan to use higher-order motion features such as acceleration, jerk (the temporal derivative of acceleration), and yaw rate (change in heading). These additions can provide the model with more context about an

agent's motion state, potentially improving its ability to capture more nuanced behaviors like turning, deceleration, and unexpected maneuvers.

Another possible addition to improve our model's performance is data augmentation. While our current approach applies flips and rotations to our input data during training as in-place transformations, we seek to extend this by appending augmented examples as additional training samples. This change would increase the size dataset and improve the model's robustness to spatial variation. We also would want to train the model for more epochs using adaptive learning rate schedules and apply K-fold cross-validation to produce more reliable performance estimates and reduce sensitivity to dataset splits.

Additionally, gaining access to more compute resources would greatly improve the ability to tune parameters, train for more epochs, and test more extensive training methods to gain the best performance of the existing model.

Finally, we are interested in making architectural changes that combine the strengths of Graph Neural Networks (GNNs) and Transformers. A hybrid GNN and Transformer model could encode agent-agent interactions through graph structures while maintaining the attention mechanism that allows for scalable and flexible modeling of dynamic multi-agent environments.

7 Contributions

Amit focused on model training, optimization and worked alongside other team members to build different type of architectures for the challenge. Amrita experimented with different model architectures like multi-agent LSTMs and performed model hyperparameter tuning on the final model architecture. Iris established the baseline performance from simpler models and experimented with model architectures like autoregressive LSTMs and transformers. Shayan worked on data normalization, ego-centric and multi-agent LSTMs, and transformer model architectures with an emphasis on hyperparameter tuning.

For the final report, Amit contributed to the Experiments and Methods. Amrita contributed to the Abstract, Introduction, Related Works, Conclusion, and Problem Statement. Iris contributed to the Problem Statement, Methods, and Experiments. Shayan contributed to the Methods, Conclusion, and Results.

References

[1] B. Wilson, W. Qi, T. Agarwal, J. Lambert, J. Singh, S. Khandelwal, B. Pan, R. Kumar, A. Hartnett, J. K. Pontes, D. Ramanan, P. Carr, and J. Hays, "Argoverse 2: Next generation datasets for self-driving perception and forecasting," 2023.

[2] M.-F. Chang, J. Lambert, P. Sangkloy, J. Singh, S. Bak, A. Hartnett, D. Wang, P. Carr, S. Lucey, D. Ramanan, and J. Hays, "Argoverse: 3d tracking and forecasting with rich maps," 2019.

[3] Y. C. Tang and R. Salakhutdinov, "Multiple futures prediction," *arXiv preprint arXiv:1911.00997*, 2019.

[4] W. Zeng, M. Liang, R. Liao, and R. Urtasun, "Lanercnn: Distributed representations for graph-centric motion forecasting," *arXiv preprint arXiv:2101.06653*, 2021.

[5] M. Liang, B. Yang, R. Hu, Y. Chen, R. Liao, S. Feng, and R. Urtasun, "Learning lane graph representations for motion forecasting," in *European Conference on Computer Vision (ECCV)*, pp. 541–556, Springer, 2020.

[6] M. Ye, T. Cao, and Q. Chen, "Tpcn: Temporal point cloud networks for motion forecasting," *arXiv preprint arXiv:2103.03067*, 2021.

[7] L. Fang, Q. Jiang, J. Shi, and B. Zhou, "Tpnet: Trajectory proposal network for motion prediction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6790–6799, 2020.

[8] J. Gao, C. Sun, H. Zhao, Y. Shen, D. Anguelov, C. Li, and C. Schmid, "Vectornet: Encoding hd maps and agent dynamics from vectorized representation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11525–11533, 2020.