CSE 251B Project Final Report

Edward Jin ecjin@ucsd.edu
Saeji Hong sahong@ucsd.edu
Aniket Pratap apratap@ucsd.edu
Derek Ma d5ma@ucsd.edu

1 Introduction

1.1 The Task

Given a dataset of time frames, the task is to predict the trajectory of the ego vehicle in regards to the environment. The ego vehicle is the main vehicle we are predicting for——which could be a Waymo car, for example. This object reacts to the environment in various ways and proceeds along its desired path given by the dataset. We aim to predict the trajectory of the ego vehicle, (x, y), for 60 time frames using the first 50 time frames given to us. This prediction is critical for enabling safe behavior in autonomous vehicles. This concept can also be used in robotics to maneuver the environment, and any task where there is an autonomous machine traversing the real world.

1.1.1 Data Pre-processing

The starter code exhibited the following techniques for data pre-processing:

- **Train Test Split:** 110 time frames split to 50 train, 60 test This was done given the problem constraints, given 50 time stamps from the test, predict the next 60. This data was not shuffled within the time frame because order matters, but it was shuffled in the order the time frames were given to the model.
- **Anchor Point:** last known position of ego vehicle was the basis point for the next predictions. Intuitively, this makes sense because we want to predict the trajectory from the last *known* location given the information from the previous trajectories. In essence, it's saying: given the current position of the vehicle and its previous movements, where will it go next?
- **Normalization:** By making the last known position $(0, 0)$, prepossessing techniques were deployed to normalize the data by 7. The last known distance is set as the origin, the distance is calculated, and then the scaling feature is applied. This normalization is used so that the data is within a consistent range and so that the gradients don't explode.
- **Data Augmentation:** was used so that the model can *generalize* better. If the model were given the exact normalized dataset, it would only memorize the trajectories. Meaning that it will only memorize left turn or going straight, for example. By adding noise, we are adding more data that the model can learn from. Real driving isn't perfect. What if you take a wide turn, for example? This is the purpose of data augmentation.

1.1.2 Start and Code and Problems

The starter code provides a linear, MLP, and LSTM model. The Linear model uses a linear regression to predict the future (x, y) coordinates, while the MLP creates layers to achieve the same thing. Both these models are explained in detail throughout the report, but the main model in the starter code uses a simple LSTM model with a hidden dimension size of 128. This model learns long-term details, but it only *considers* the ego agent. While this is a good strategy, a downside is that it doesn't consider the relationships of other vehicles. For example, the velocity and movement of one object can affect our ego vehicle in a certain way, rather than just looking at the ego vehicle. Our key contributions were as follows:

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

- **Attention:** was the main contribution because it effectively learned relevant information about the surrounding environment. What features affected the ego vehicle? Maybe a car close by caused the ego vehicle to react in some way. Attention finds these connections and places a high emphasis on relevant information. This solves the issue with only focusing don't the ego vehicle.
- **Optimizer:** was changed to AdamW. While a huge change wasn't noticed we used AdamW due to its usage in other research projects and the success it attained.
- **Batch Size**: was changed to be larger, we believed that a larger batch allowed for the model to train faster and generalize more.
- **Hidden Dimension** was increased to 256 to possibly learn more features and detail. We believed that the current model wasn't complex enough to learn the intricacies of the data.
- **Scheduler:** was changed to have a continuous decrease in the learning rate based on how successful the loss was. We believed to only decrease the learning rate if we we consistently doing bad for several runs——opposed to decreasing it after a certain number of epochs.

2 Related Work (Bonus)

- Li and Yu [2024] uses transformers to build a prediction framework
- Madjid et al. [2025] talks about different methods to predict trajectory
- Jiang et al. [2019] uses LSTM architecture for trajectory prediction

3 Problem Statement

The input was normalized and scaled with a factor of 5 to allow the data to be closer together when un-normalizing, compared to 7 and the validation was set to 5%

Input: The training dataset is a 4D tensor of shape $(10000, 50, 110, 6)$, where:

- 10000 = number of scenes,
- 50 = number of agents per scene (including the ego vehicle),
- 110 = number of time steps per agent (sampled at 10Hz),
- 6 = features per time step: position (x, y), velocity (v_x, v_y), heading angle θ, and object type.

Mathematically, the input can be written as:

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

Output: Given the first 50 time steps of the ego vehicle (i.e., $X[:, 0, : 50, :]$), the goal is to predict its next 60 positions (i.e., for time steps 51–110). The model should produce a tensor of shape $(2100 \times 60, 2)$ for the test set, where:

- 2100 = number of test scenes,
- 60 = predicted future time steps,
- 2 = (x, y) position coordinates per step

Formally, the prediction task can be described as learning a function:

$$f : \mathbb{R}^{50\times 50\times 6} \rightarrow \mathbb{R}^{60\times 2}$$

that maps the first 5 seconds (50 time steps) of a scene to the future 6 seconds (60 time steps) of the ego vehicle's trajectory.

The final output can be flattened into a matrix:

$$Y_{\text{pred}} \in \mathbb{R}^{126000\times 2}, \quad \text{where } 126000 = 2100 \times 60$$

.

4 Methods

4.1 LSTM + Attention Architecture

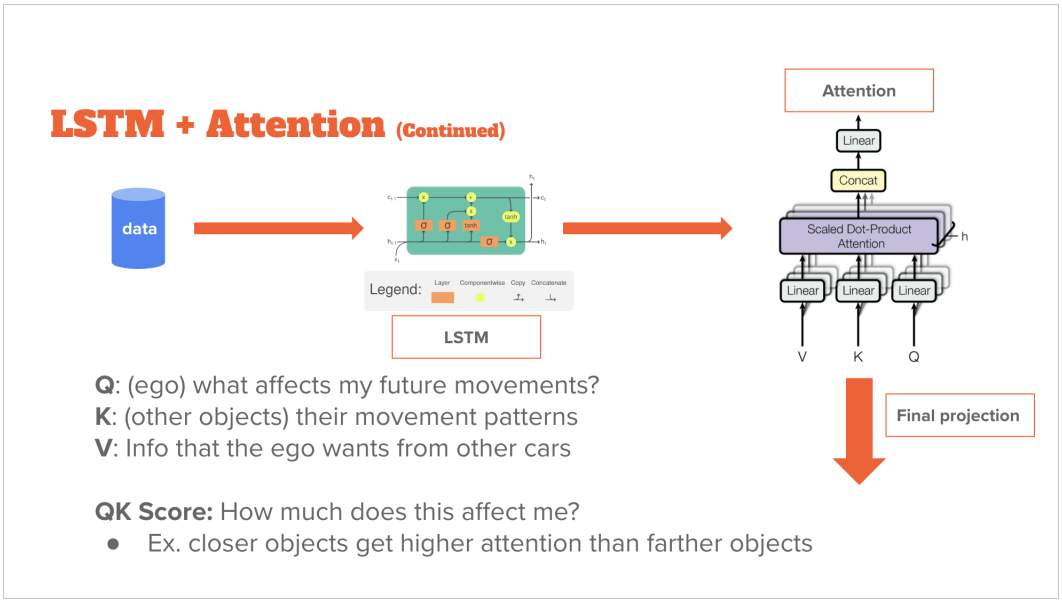


Figure 1: LSTM + Attention Architecture

The meat of our approach relied on the concept of *attention*. The pipeline can be represented as follows

1. Split, normalize, and augment data: This was done given the starter code so nothing much was changed
2. Pass through an LSTM model: The starter code also provided an LSTM model. We tuned the amount hidden dimensions it has. The benefit of using an LSTM is that that is excels in capturing information from sequential representations. Due to the format of the data, it being time frames, we chose to keep the current LSTM model. The goal is for the LSTM to learn the weights for 110 timesteps and pass those weights to attention
3. The learned weights will then pass through an attention mechanism and in our case, the attention mechanism will be represented by 4 heads. Given that we already set the last known distance as the starting point for our prediction, the attention heads receive the previous movements along with the current location. The ego vehicle then queries: what affects my future movements? The keys for this could be something like a lane change of a motorcycle, a red light, similar speed vehicles, etc. The attention score is then calculated, to see which of these keys or events have the most influence on the ego vehicle. Since there are 4 heads, 4 of these queries are asked to get the best representation of our environment
4. A final projection layer is used to convert what we learned through the attention mechanism to the final output that is required for the assignment

4.2 Prediction Task as Optimization Task

The goal is to decrease the loss between the true trajectories of the ego vehicle and the predicted trajectories. This means decreasing the MSE for every position in the time frame for the ego vehicle. MSE was used to exaggerate huge differences between the true and predicted trajectories. Once the models were trained, we used the starter code to observe the train and validation normalized MSEs and MAE, and MSE. The equations for the optimization problems:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{1}$$

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{2}$$

5 Experiments

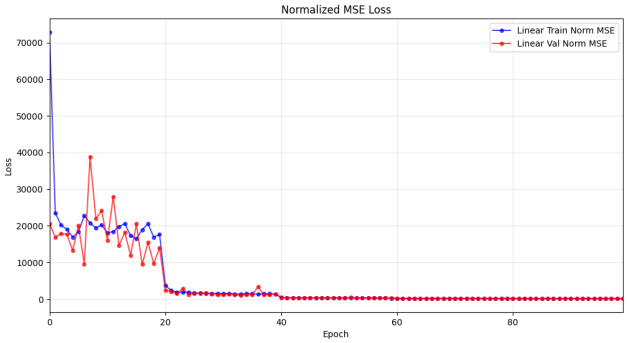
For validation in each experiment, we use the last 5% of the data.

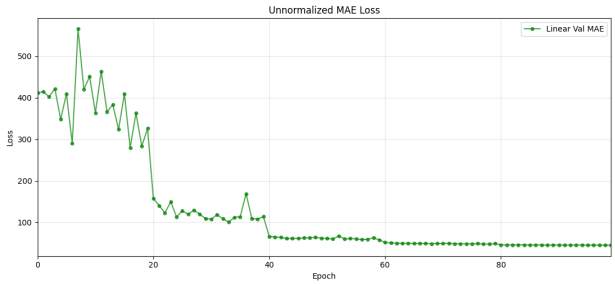
5.1 Baselines

Note that Linear Regression, MLP, and LSTMS had the same:

optimizer	learning rate	weight decay	scheduler	step size	gamma	early stopping
Adam	$1e-3$	$1e-4$	StepL	20	.25	10

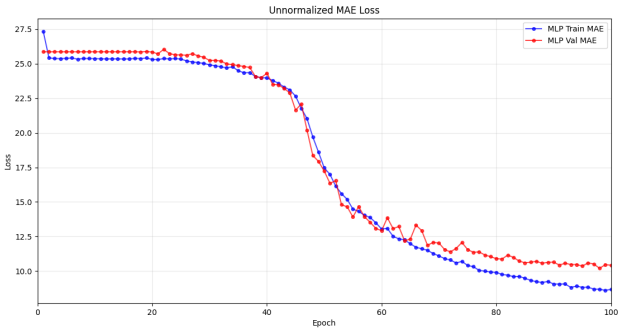
5.1.1 Linear Regression





Linear regression was our first model. We simply input a dimension of size $50 \times 50 \times 2$——essentially only looking at the (x, y) of the 50 times stamps, and the output dimension was 60×2——predicting the (x, y) for 60 time stamps. We observed an extremely high train and validation MSE——meaning that this was not a strong model. Not only that, but the model also shows to have high variance——as we can see by the fluctuating validation MSE——which then converges. This alone showed us that we didn't need to explore this model further and instead focus on another architecture.

5.1.2 MLP



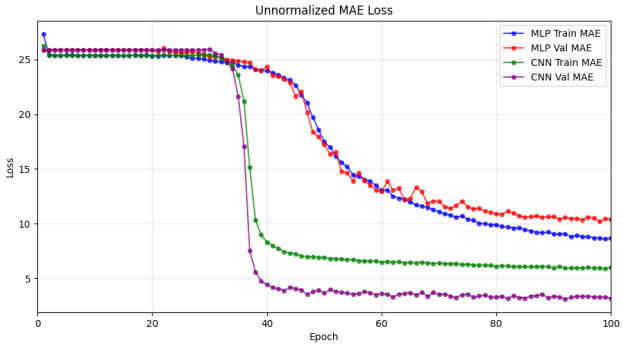
Architecture:

Layer 1	Layer 2	Layer 3	Drop Out
1024	512	256	.1

To start with a simple initial design, we defined a multi-layer perceptron with 3 hidden layers. It first flattens the input data and then passes it through a series of fully connected layers. The flattened input is projected to 1024 dimensions, then down to 512, then down to 256, and finally down to the specified number of output features (120, or 60 x 2 for the ego agent). Each linear transformation is followed by a ReLU activation to introduce non-linearity and a Dropout layer with a probability of 0.1 to help prevent overfitting by randomly zeroing out a fraction of the activations during training. As mentioned, the final output is reshaped to a tensor of shape (batchSize, 60, 2).

While it performed substantially better in terms of MAE, the loss was still far above 0. But the trend of the loss going down showed that we were going in the right direction. The MLP model did a decent job, but we figured that the high loss could be because the model was only learning small parts of the scene. The idea was to use CNNs to take advantage of the transitional and spatial invariance to get a wider representation of what the car will do at each time stamp. Note that the epochs and hyper params were kept the same for both models so far.

5.1.3 CNN



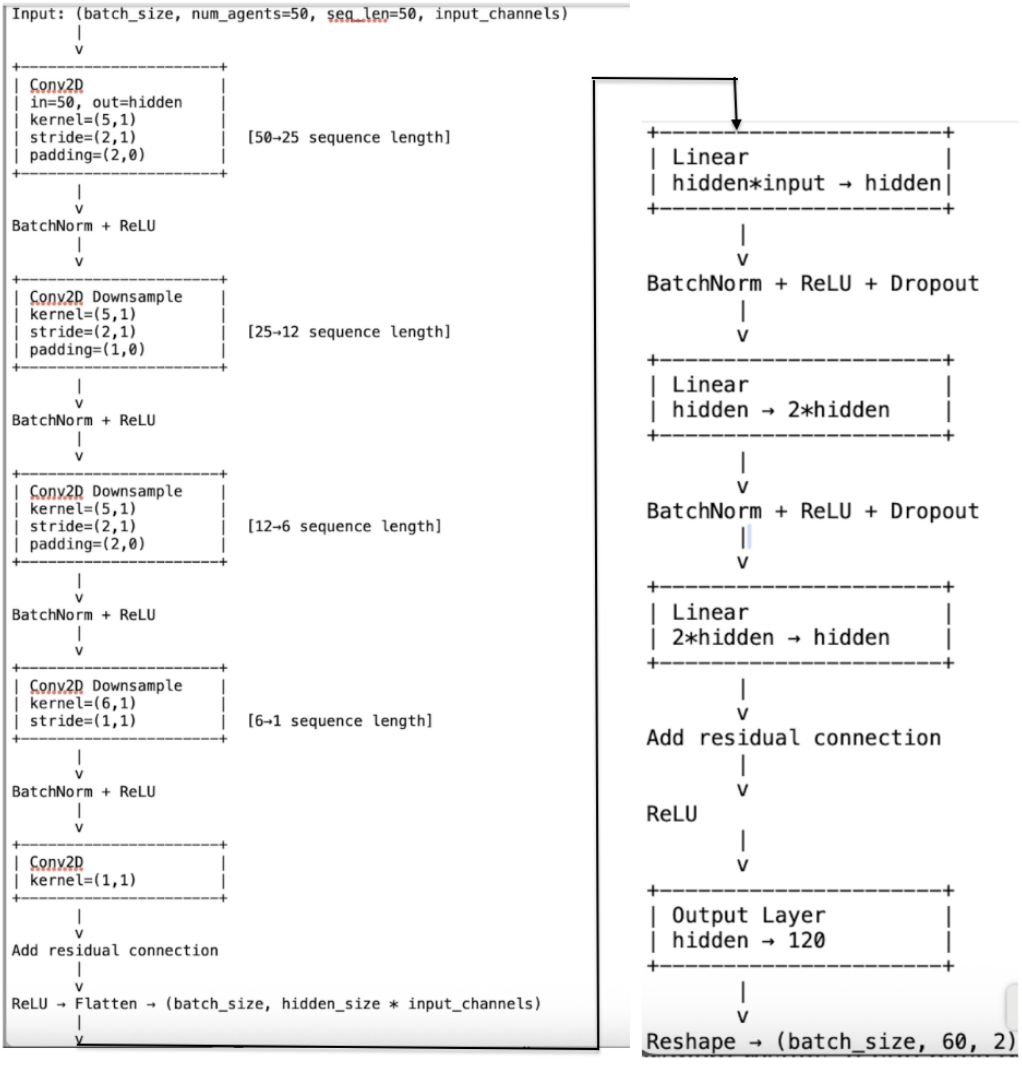
To build on the very basic MLP model, we decided to introduce CNNs. The model now begins by taking as input (batchSize, numAgents, seqLen, inputChannels) and applying two 2D convolutional layers that each convolve across the sequence dimension. Each convolution uses a kernel size of "(5, 1)" with a stride of "(2, 1)" and padding of "(2, 0)", effectively reducing the sequence length while preserving feature dimensions. Each convolution layer is followed by ReLU activations to introduce non-linearity. After the convolutional feature extraction, the resulting tensor is flattened into a 1D vector per example, producing a feature vector that captures the spatiotemporal dynamics of the agents. This vector is then passed through a pair of fully connected layers. The first fully connected

layer reduces the feature dimensionality to a hidden size, followed by a ReLU activation and dropout for regularization. The final linear layer outputs a vector of size 120, representing 60 future time steps of (x, y) coordinates. This output is reshaped into a "(B, 60, 2)" tensor, giving the predicted trajectory for the target agent.

Our theory seemed correct as we saw the train and val MAE drastically drop. Our theory of using CNNs to leverage transitional and spatial variance proved to work because we want to generalize to the trajectory of the ego vehicle. This is also why the data is augmented—to account for this generalization. CNNs use the same methodology in terms of translational invariance. And to see what affected our ego vehicle, it made sense to use a window to see what's close by, because that will affect the ego in the environment. To push our model further, we chose to make a deeper CNN to see if we can achieve better performance.

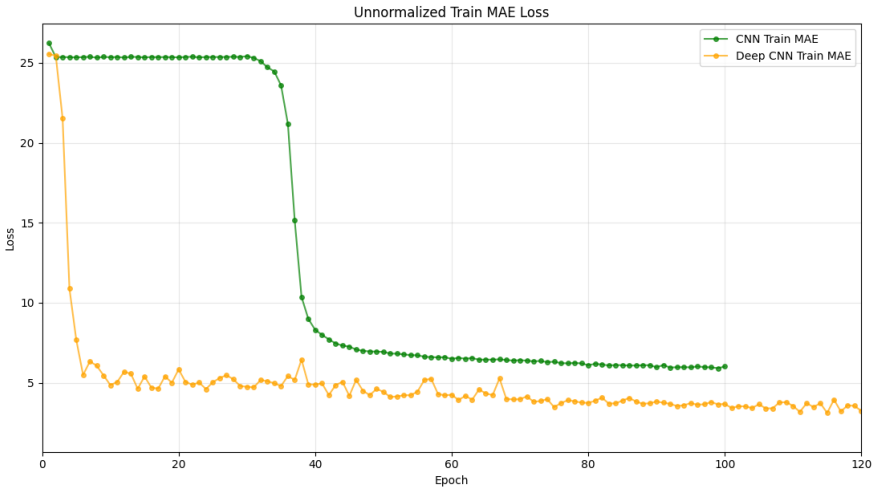
5.1.4 Deeper CNN

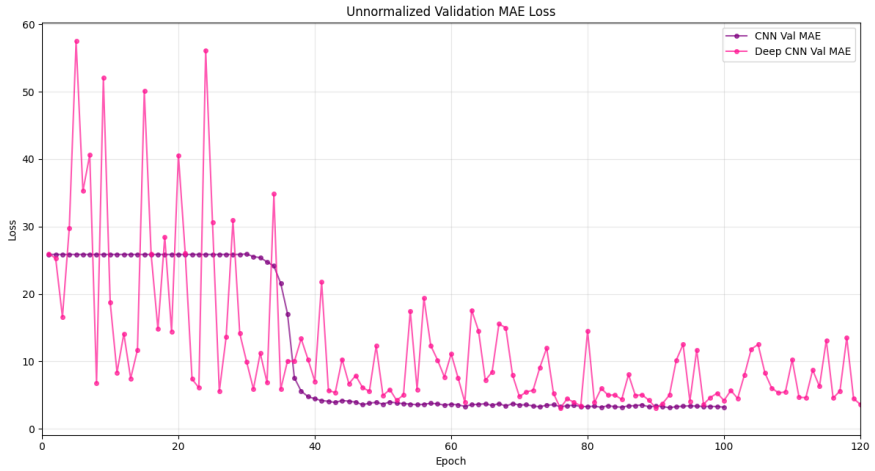
Moving towards an even deeper model, we decided to use 5 CNN layers with 4 fully-connected layers.



This deep model takes as input a tensor of shape "(batchSize, numAgents, seqLen, inputChannels)", then we apply a series of temporal convolutions that operate across the sequence dimension while keeping spatial information intact. We begin by compressing the sequence length using several 2D convolutions with kernel shapes like "(5,1)" and strides of "(2,1)". This choice allows us to downsam-

ple the sequence length dimension from 50 to 25, then to 12, then to 6, and finally to 1. This effectively summarizes the motion history into a compact representation. After each convolutional layer, we apply batch normalization and ReLU activations to stabilize training and introduce nonlinearity. To refine the compressed representation further, we finally use a 1×1 convolution with a residual connection, helping the model learn information across the hidden channels and improving gradient flow. Once we have reduced the sequence dimension to one, we flatten the tensor and transition to a series of fully connected layers. These layers project the flattened tensor into a higher dimension and then project down again to help the model learn high-level representations of the motion dynamics across all agents and channels. We include batch normalization after each fully-connected layer here as well to prevent overfitting and accelerate convergence. Additionally, we incorporate a residual connection between the first and second fully connected layers, promoting feature reuse and deeper learning. Our final output is a prediction of 60 future time steps, each represented as a 2D position vector (x, y), resulting in an output shape of (batchSize, 60, 2).



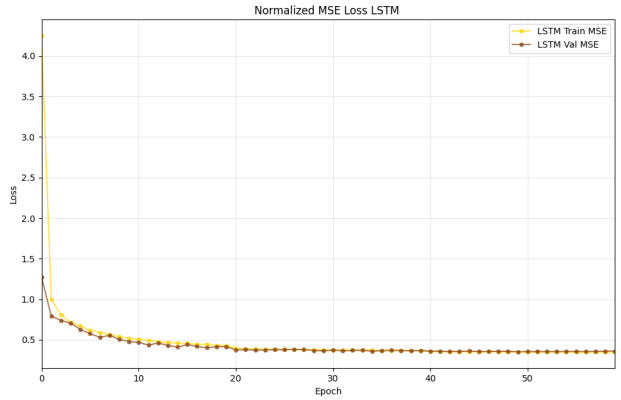


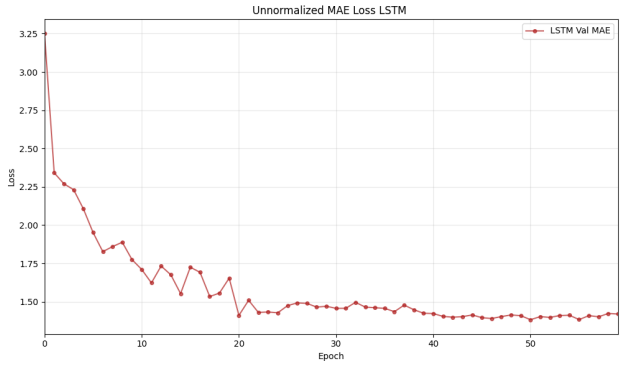
While the training does indeed converge, the validation is spontaneous—— significantly worse than the simple CNN model. This is a clear sign of overfitting and shows that the deeper CNN struggles to generalize the data. This makes sense because when you increase complexity, you are more prone to overfitting. This led us to a new theory: Time stamps are sequential, and CNNs don't account for this sequential nature of data. Could LSTMS be better at generalizing time-based data?

5.1.5 LSTM

Architecture:

Hidden Dim	Out Dim	scale factor
128	120	5





Our evaluation showed that LSTMs were the superior models. They achieved extremely low MSE and MAE compared to other architectures, indicating high accuracy. In addition, their validation metrics remained stable and converged quickly, demonstrating low variance and strong generalization. Given the sequential nature of our data and the need to capture long-term dependencies, LSTMs seemed well suited for this task.

We started with the starter code's LSTM architecture, using a hidden dimension of 128. While our attempts at hyperparameter tuning, such as the scheduler, hidden dimension, and batch size, yielded worse results on the Kaggle leaderboard. This led us to realize that simply tuning hyperparameters wasn't enough. We recognized that while the LSTM excelled at learning temporal sequences, it struggled to identify which specific elements within the scene most influenced the ego vehicle's behavior. This observation paved the way for our next addition: Attention Mechanisms.

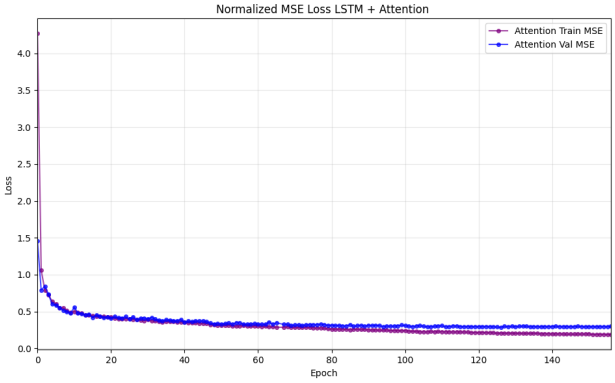
5.1.6 LSTM + Attention

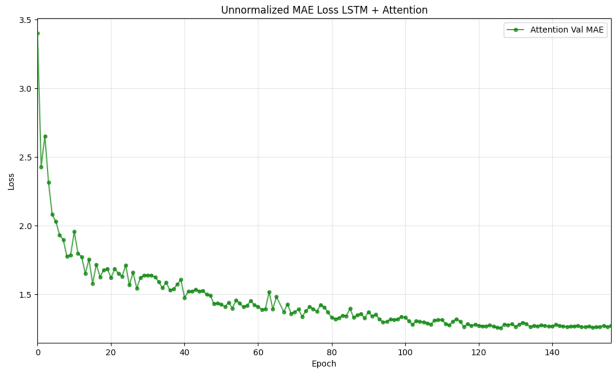
Architecture:

Hidden Dim	Out Dim	attention heads	scaled factor
256	120	4	5

Hyper-parameters

optimizer	learning rate	weight decay	scheduler	factor	patience	early stopping
AdamW	$1e-3$	$1e-4$	ReduceLROnPlateau	.75	5	30





Now, given this graph alone, it is difficult to see any improvement when compared to the LSTM models, given the final model results:

Models	Train Norm. MSE	Val Norm. MSE	Val Unnorm. MAE	Val Unnorm. MSE
LSTM	0.3464	0.3617	1.4197	9.0413
LSTM + Attention	0.1901	0.2996	1.2726	7.4900
Improvement	+.1563	+.0621	+.0421	+1.5513

We notice that the LSTM + Attention model surpasses the LSTM model in every criterion. Specifically, the val normalized values improved—meaning the model has low variance. The training loss is also lower meaning it has high bias. This is generally the sweet spot for models, and the biggest improvement was adding attention. As discussed before, tweaking the hyperparameters and the scheduler only made the model worse when compared to the current baseline. Attention can be seen to give a significant boost to the model's performance. The reasoning for the hyper parameter tuning is in the implementation details section.

5.2 Evaluation

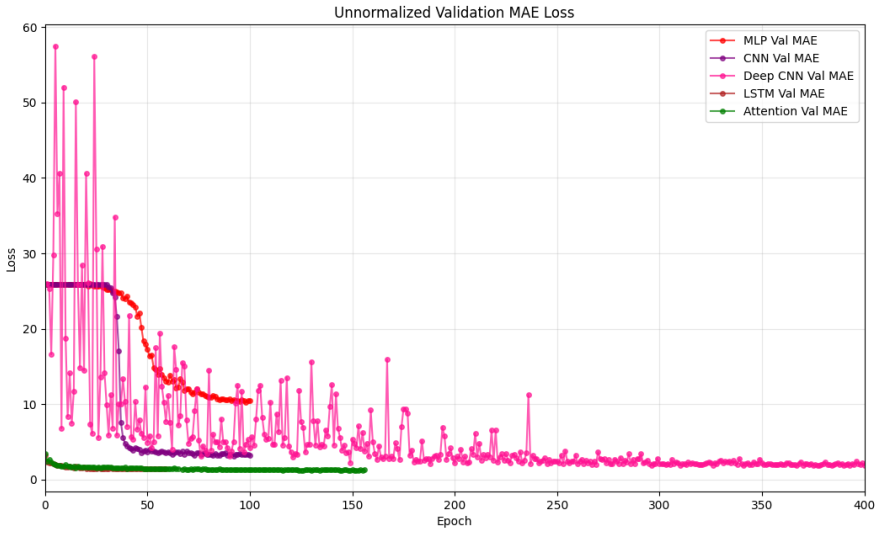
Our evaluation relied on metrics specified in the starter code. We monitored training MSE and MAE to gauge the model's capacity for overfitting, and validation MSE and MAE to assess its generalization

performance. While the Kaggle competition's leaderboard was determined by MSE, we initially prioritized MAE for model comparison because it provided a more interpretable error magnitude, being in the same units as our target. After identifying promising models, our optimization shifted to minimizing the MSE, thus emphasizing the reduction of larger errors.

5.3 Implementation Details

- Apple Silicon GPU—Pytorch mps
- Around 3 minutes per epoch
- Hyper Parameter Tuning
  - Increased hidden dimension from 128 to 256, we believed that this would allow the model to learn richer information, as usually increasing hidden dimensions makes the model more complex and able to fit the data better
  - Increased epoch range to 200 to allow for more iterations through the training data to have a better fit of the data
  - Changed our scheduler from the step learning rate—which was more of a discrete method to a continuous method. We chose to use ReduceLROnPlateau to have it decrease the learning rate by a factor of .75 for every 5 epochs that the loss is the same or increases. We believed a dynamic learning rate would be better in this case because it adapts to the loss.
  - Increased batch size from 32 $\rightarrow$ 64 because bigger batches means faster training. Since we are using attention, speed was important since we weren't using high-end GPUs. Also we believed that increasing the batch size would stabilize the gradients since it's averaged over more samples
  - We increased the early stopping patience from 10 $\rightarrow$ 30 because we wanted more epochs to run and decrease the training loss. To do this, we had to overfit to the training data more—hence the increase in early stopping. Our model didn't complete all 200 epochs but stopped on the 156th iteration
  - Adam $\rightarrow$ AdamW because we read that AdamW fixes some core issues with the current Adam implementation. Since AdamW allowed for consistent regularization, better generalization, and improved convergence
  - Used 4 attention heads—specifically because of our computational platform. increasing attention heads made training longer—mostly due to the score calculation bottleneck. Otherwise, we would have tried increasing the number of heads further

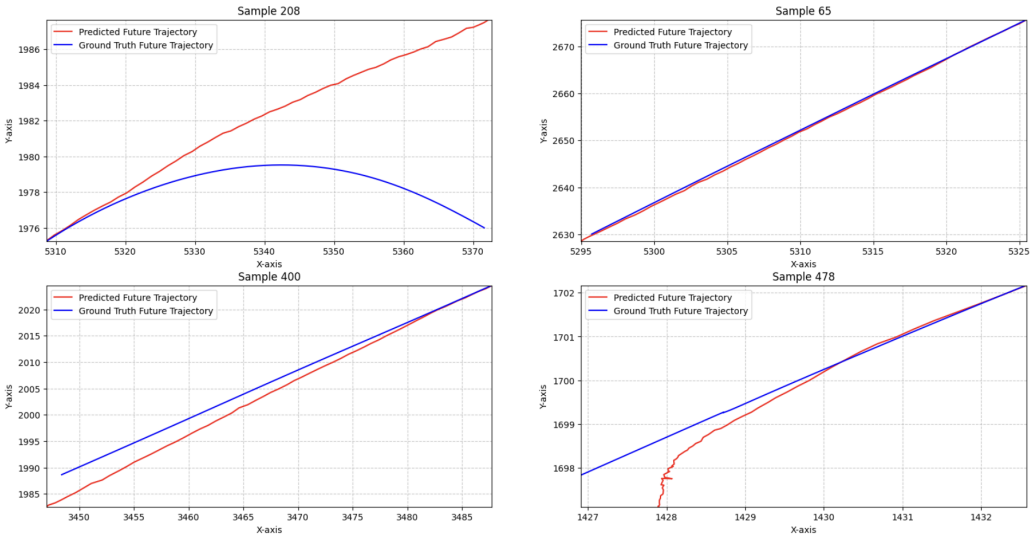
5.4 Final Results



When comparing all the validation MAEs, we can see that the LSTM + Attention model has the best performance for the lowest amount of epochs. Once again, this is due to the LSTM model working well with the sequential nature of this task and the attention heads learning what features should be attended to. Here were the final MAEs:

Models	Val Unnorm. MAE
MLP	10.4320
CNN	3.2123
Deep CNN	2.0415
LSTM	1.4197
LSTM + Attention	1.2726

Some example trajectories of the LSTM + Attention model can be seen below, and it predicts samples 400 and 65 quite well—— almost fitting it perfectly. Sample 208 is where the model falls short, and this could be because of its failure to generalize given the dataset.



All in all, LSTM + Attention allowed us to get 8.01949 on the public leader board and 8.15276 on the private leader board——ranking us 14^{th} place. The drop in the ranking showed that our model failed to generalize enough.

5.5 Ablations

To test whether attention made the model better, we kept the hyper parameters constant between both architectures (LSTM vs. LSTM + Attention) to match. These were the same hyperparameters as our best model. Now, the only difference between the starter code LSTM and our best model is the attention head implementation.

Optimizer	AdamW
Learning Rate	$1e-3$
Weight Decay	$1e-4$
Scheduler	ReduceLROnPlateau
Factor	.75
Patience	5
Early Stopping	30
Batch Size	64
Hidden Dimension	256

Models	Train Norm. MSE	Val Norm. MSE	Val Unnorm. MAE	Val Unnorm. MSE
LSTM	0.2932	0.3405	1.3143	8.5127
LSTM + Attention	0.1901	0.2996	1.2726	7.4900
Improvement	+.1031	+.0409	+.0417	+1.0227

Based on the last row, it can be observed that the LSTM + Attention architecture is better than the standard LSTM architecture provided to us. The difference between the validation MSEs show that the attention model is far superior in terms of generalizing. The training MSE also shows that LSTM + attention overfits to the data substantially more than the standard LSTM model. Given this data analysis, we can conclude that Attention was the sole reason for the increase in performance when comparing architectures.

6 Conclusion

Throughout this project, we learned strategies to progressively build models, improve and tuning them.

6.1 Model Building

A great skill to have is to know what model to build for the current problem. Throughout this project, we learned to start with the most basic model and build our way up to something that is highly desirable for the problem. This was seen by first starting with a linear regression, it did poorly, then moving to the next complex model, and so on. Throughout this phase, we also discussed the pros and cons of certain model——why it makes sense to use LSTMs vs CNNs, for example. By choosing a basic model, and building it up, we learned more about the problem and learned the skill of starting broad, then narrowing down.

6.2 Improving Models

Once we have a great model, it's important to tune the hyperparameters to achieve the best result. We gained an understanding of changing some hyperparameters and why others should remain the same. For example, we can increase the hidden dimension of a model—making it more complex—but we would have to face the problem of it overfitting. Once again, these pros and cons came into play, but we gained the reasoning capabilities to tune the models with a proper justification.

6.3 Limitations and Future Work

One limitation we had was our choice of computational resources. Many of us were not familiar with AWS, so we opted to train locally using Apple Silicon to quickly get an environment setup and running. Placing more of an emphasis on AWS could have potentially allowed us to train a stronger model (or train a similar model faster) and achieve a higher rank on the leaderboard. Another limitation was our experience. Through this course we have certainly been exposed to a lot of core deep learning models, but what truly makes a good model? How do you know? We believe this is an area where years of experience make the difference between developing a good model versus a great model.

For future work, we can opt to directly use a transformer because LSTMs are a stepping stone to transformers, and if we are already using attention, we might as well use the architecture that leverages it the most. Another point could be to augment the data even more: possibly including synthetic data, for example.

Overall, we were able to work with various models, analyze lots of results, and gain a deeper understanding of the nuances between various deep learning models.

7 Contributions

- Model Training: Edward Jin, Aniket Pratap

- Analysis: Derek Ma, Saeji Hong
- Report: Edward Jin, Aniket Pratap, Derek Ma, Saeji Hong

References

Huatao Jiang, Lin Chang, Qing Li, and Dapeng Chen. Trajectory prediction of vehicles based on deep learning. In *2019 4th International Conference on Intelligent Transportation Engineering (ICITE)*, pages 190–195, 2019. doi: 10.1109/ICITE.2019.8880168.

Zhenning Li and Hao Yu. Trajectory prediction for autonomous driving using a transformer network, 2024. URL `https://arxiv.org/abs/2402.16501`.

Nadya Abdel Madjid, Abdulrahman Ahmad, Murad Mebrahtu, Yousef Babaa, Abdelmoamen Nasser, Sumbal Malik, Bilal Hassan, Naoufel Werghi, Jorge Dias, and Majid Khonji. Trajectory prediction for autonomous driving: Progress, limitations, and future directions, 2025. URL `https://arxiv.org/abs/2503.03262`.