CSE251B Final Report - Sigmoid Males

Dancheng Liu **Brandon Erickson** **Trevor Tuttle** **Aman Parikh**

1 Task Description and Background

1.1 Introduction

The autonomous motion forecasting challenge is a task in which the future trajectory of a particular agent is predicted based on the past trajectory. More specifically, given the position and velocity of the agent in first 2 seconds, we need to predict the position and velocity of the agent in the next 3 seconds. For the problem, many corresponding factors are given such as agents id, track id, lane/map information and much more.

The goal of the motion forecasting problem is to predict the future movements of other traffic agents, allowing autonomous vehicles to make informed decisions and avoid potential collisions. There are a number of challenges associated with motion forecasting. One challenge is that the movement of traffic agents is often unpredictable. This is due to a variety of factors, such as the behavior of human drivers, the presence of obstacles, and the weather. Another challenge is that the environment in which autonomous vehicles operate is constantly changing. This means that the models used for motion forecasting must be able to adapt to new conditions quickly and accurately. By overcoming the challenges in motion forecasting, we can enhance the overall safety and efficiency of autonomous vehicles, bringing us closer to widespread adoption and integration into existing transportation systems.

Solving such an issue would impact real life by allowing for the full automation of vehicular transport. This could very much save many human lives, if it could be solved to the degree that the AI drives better than humans. Over 40,000 people die from car accidents each year, any incremental improvement solving this problem can save many people, as well as eliminate banes on society such as drunk driving. Aside from this, we can consider that these vehicles would allow for a more efficient transport method (i.e. without human error, vehicles could eliminate any extra time spent waiting at red lights etc).

1.2 Mathematical Formulation

1. Input: The input consists of a sequence of observed position and velocity pairs for tracked agents, denoted as p_{in} = ($p_{in}(x)$,$p_{in}(y)$) and v_{in} = ($v_{in}(x)$,$v_{in}(y)$). Mathematically, the input can be represented as stacking both the position and velocity vectors together as follows :
   Input = X: $\{(p_{in_i}, v_{in_i})\}_{i=1}^{N}$ =$[(p_{in_1}, v_{in_1}), (p_{in_2}, v_{in_2}), ..., (p_{in_N}, v_{in_N})]$
2. Output: The output is a sequence of predicted positions for the main agent over the next three seconds, denoted as p_{out}. Mathematically, the output can be represented as:
   Output = y:$\{p_{out_i}\}_{i=1}^{N}$ = $[p_{out_1}, p_{out_2}, ..., p_{out_N}]$
3. Prediction Task: The prediction task involves learning a function or model that maps the input sequence of observed position and velocity pairs to the corresponding output sequence of predicted positions. Formally, we train a regression model F such that F maps the input sequence to the output sequence with the goal of $min\{MSE(F(p_{in_i}, v_{in_i}) - p_{out_i})\}$, where MSE is the mean-squared-loss.

Figure 1: Visualization of four samples

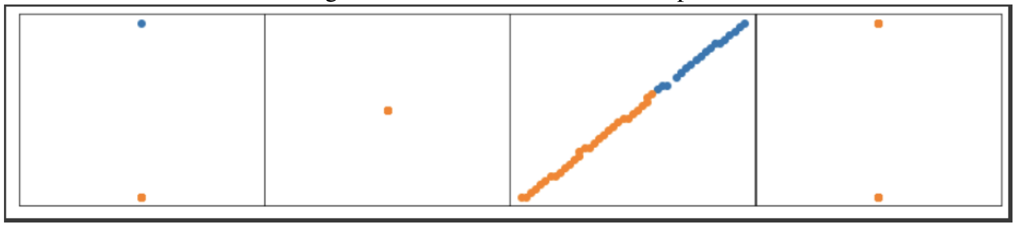
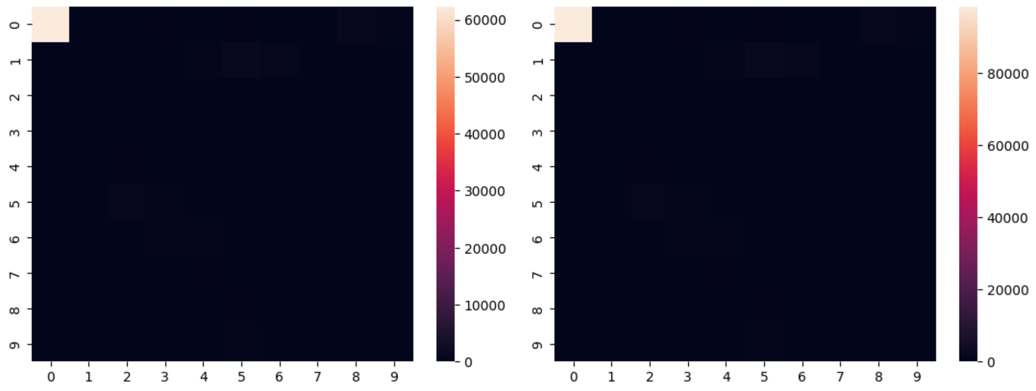


Figure 2: Heatmap for input and output values with non-tracked agents included



> In other words, we aim to develop a deep learning model that can effectively capture the patterns and dynamics of the observed positions and velocities to provide prediction of the future positions of the main agent over the 3 seconds time interval.

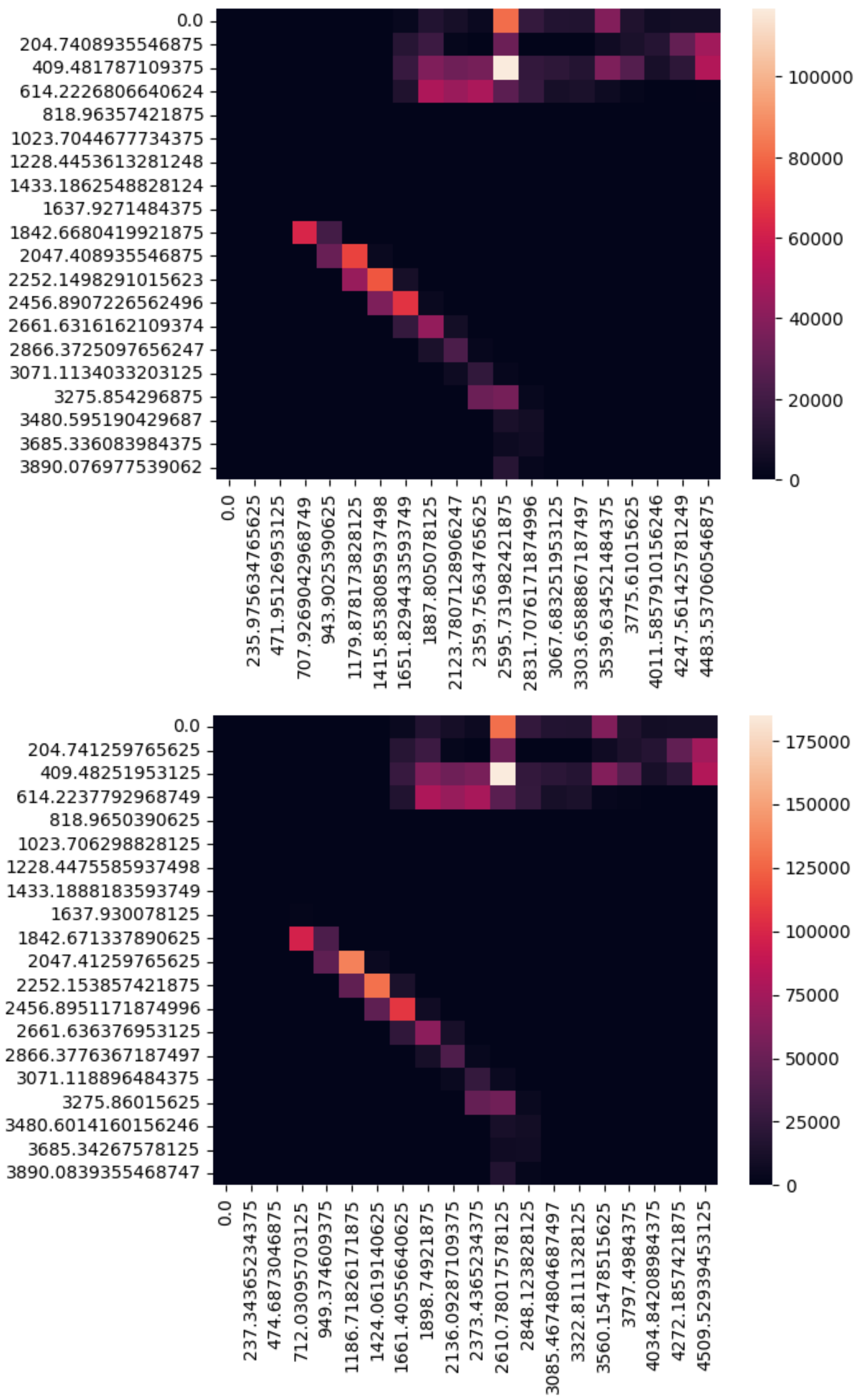
When we consider whether or not our model could generalize to other tasks, we consider that though dimensionality may be a hiccup, it could be used in tasks such as heart rate prediction, stock market prediction, and weather forecasting. Our final model learns based off of both time (in its LSTM components) and off of image data (in its CNN components). As such, time series data can be fed into the model for various tasks, as the aforementioned heart rate and stock market predictors. In the stock market example the positions could be current stock prices, and the velocities could be their change in price, which would be used to predict future price. For similar reasons, weather forecasting could also be a task in which our model performs well, with the position and velocity pairing being wind speed, humidity, or temperature.

2 Exploratory Data Analysis

2.1 Data Statistics

The given train dataset consists of 205942 samples and test dataset has 3200 samples. The input dimensions for the model from the dataset are: $[60, 19, 4]$, which signifies there are position and velocity values for 60 agents over 19 time stamps. The output dimensions are: $[60, 30, 4]$, which signifies there are position and velocity values for 60 agents distributed over 30 time stamps. The visualization of a sample is here 1 In total both the input and output tensors are of 3 dimensions. We use heat map to visualize the distribution of the values in the dataset. As the entire dataset is too large to fit in the memory, we take a batch of data, quantizing them into 10 bins, and use heatmap to visualize their frequencies. As shown by Figure 2, the non-tracked agents dominates the distribution with 0s. If we remove the non-tracked agents from the batch and use a 20 bins and 10000 samples, the distribution is summarized in Figure 3, with the top being the input and bottom being the output. As we can see, the data is still non-uniform-distributed, and it forms two main clusters.

Figure 3: Heatmap for input and output values with non-tracked agents excluded



2.2 Data split and additional features

For all models, we use 80-20 split for training and validation split with sizes 164753 and 41189 respectively. The testing dataset is the 3200 agent test dataset, and the testing result is reported by Kaggle. We only used four features (x and y position, x and y velocity) as the input of our model. We did not consider any additional features from the dataset, though those features could have given more information and led to better performance of our model. Especially, we have not yet incorporated the city information as a part of our final model. We plan to do this as our future work.

2.3 Preprocessing

Based on the starting code, we devised three types of pre-processing techniques. The first one is much similar to the given code, where position and velocity data are extracted from the raw dataset, and 19 frames are used for training while 30 frames are used for prediction. One improvement we noticed from the starting code was that the conversion from list to tensor was very slow, and that significantly slows down the training speed. Thus, we used numpy.stack to convert the lists into numpy ndarrays before converting them to tensors. This processing technique was used for our first CNN architectures that expect a 4D input, where our processed data has the shape of [batch size, 60, 19, 4]. However, our latest CNN autoencoder abandons this design, and choose to predict one frame at a time, with output in the shape of [batch size, 60, 1, 4].

The second pre-processing was inspired by the results obtained CNN autoencoder. We noticed that there are large offsets in the predicted results, and they come from the unbalanced nature of the training set. In the training set, since we used all sixty agents for training, and the majority of them are not active, the training dataset is heavily skewed towards 0. Nevertheless, it is a nontrivial task to simply use the mask and filter out those inactive agents because such approaches leads to an uneven data structure.

Thus, we decided to use individual agent's projectile as the features of the dataset. With the help of the mask from the raw dataset, we are able to identify the active tracked agents, and we now have a 3D input to the model, with the shape of [batch size, 19, 4]. We also considered the limitation of memory when predicting 30 frames all together. Thus, we crop the prediction task into predicting only the next 1 frame, reducing the size of y label into [batch size, 4].

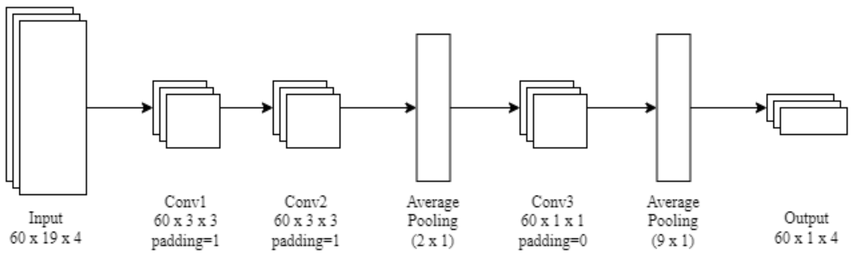
For the last pre-proessing technique, we noticed that the displacement error builds up from predicting only 1 frame at a time, and we hypothesized that predicting 30 frames together would make the model have a better performance. Thus, we designed the pipeline to perform normalization by batch. Such behavior sacrifices signifcant efficiency, but it leads to better performance of the model.

For our best performance model, data normalization has been done by translating and rotation (use first point as origin and rotate such that 19th point lies on the x axis). **Translation of the origin:** By setting the first point as the new origin (0,0), the translated data becomes relative to a common reference point. This eliminates any biases or inconsistencies arising from the absolute positions of the data points. It simplifies the analysis by focusing on the relative positions and relationships between the points, rather than their absolute coordinates. **Rotation for alignment:** Rotating the data such that the 19th point lies on the x-axis allows for a specific alignment of the data. This alignment brings important features or patterns into accordance with the coordinate axes, facilitating their analysis and modeling. Aligning the data with the axes simplifies subsequent calculations, such as measuring distances, angles, or other geometric properties. This transformation enables a more straightforward interpretation of the data and enhances the efficiency of subsequent analytical tasks.

3 Deep Learning Model and Experiment Design

During the experiment, we considered multiple types of models, and we experimented with different input and output feature representations. We will discuss some representative models below. For all models, we use MSE as the loss function.

Figure 4: Three-layer CNN model architecture



3.1 CNN

Many studies have shown that convolutional neural networks (CNNs) can be useful for both lane detection [3] and vehicle detection [4] in autonomous vehicles. These networks exploit spatial patterns in images to detect and identify objects in view of an autonomous vehicle's camera system.

Our first model implementation for the competition was a three-layer CNN as depicted in Figure 4. This model takes a $60 \times 19 \times 4$ tensor as input, which contains the position and velocity of up to 60 agents over 19 time steps. In the first two convolutional layers, the network attempts to learn spatial relationships between agents over time to predict their position and velocity in the next time step. We can then continuously feed the prediction back into the input with the previous 18 time steps to form predictions for the remaining 29 time steps. To speed up training, we used ReLU as the activation function at each convolutional layer. This network contains 68580 parameters for tuning, and requires about 2.5 minutes of runtime per epoch to train in our Colab environment.

3.2 LSTM

With the second pre-processing technique mentioned above, we are able to get a 3D tensor as input for our model. Such tensor is composed of features of 19 timestamps and each timestamp has four values, namely the horizontal and vertical position and velocity of the vehicle. Given the temporal nature of the data, it is natural to think about recurrent models that are proven to be effective on time-series data. Various works have also shown that LSTM and RNN structures are useful for predicting vehicle motions [1,2]. For this task, we consider a LSTM model with two LSTM layers and three dense layers. We choose the hyperparameters such that the first LSTM layer has 64 hidden units, and the second one has 32 hidden units.

3.3 MLP

After careful examination of the training and validation dataset, we found that the dataset is rather simple in that the vehicles seem to mostly move towards one direction. In this case, complex models are super prone to overfitting, which leads to inconsistent performance on the testing dataset. Thus, we proposed several MLP architectures that aim to learn the simple patterns with fewer layers than CNN/LSTM models. One layer MLP (linear regression) might be too simple for the task, so we evaluated the performance of a 3 layer and a 4 layer MLP architecture. For the 3 layer architecture, the output units of the three dense layers are 64, 16, and 4, respectively. This architecture is very simple, and is often used as the last few layers of a complete deep learning pipeline. To the contrast of that, we also evaluated a more complex MLP architecture with 4 layers, and the outputs from dense layers are 256, 256, 64, 4. In this design, we hope the projection from 19*4 feature space to 256 hidden space could help the model extract more information about the motion of the vehicle.

3.4 CLDNN

The most successful model is inspired from works by Emam [5] and Sumen [6]. From previous models, we noticed that the input features of this regression task is similar to the input of multi-channel signals. The four input features contain both spatial invariant characteristics, as well as temporal relationships. In this case, we should apply a model that first uses CNN layers to first extract spatial relationships, then uses LSTM for temporal relationships. Under this design principle, the first version of our CLDNN model is very similar to the one from [5]. It takes four channels of input after the second pre-processing and the first normalization. Since we recognized that four channels might still be in different magnitudes, we use different convolutional filters for each of the channel, and stack them after the convolution is completed. Afterwards, the model applies two LSTM layers, each with 64 hidden units, to the latent representations of the features, and use two dense layers to predict 1 frame of movement.

With experiment, we were surprised that such model is only marginally better than MLP design. This led to the hypothesis that is mentioned above, which is that error builds up during prediction. After several designs and experiments, in version 4 of CLDNN, we switched the prediction from 1 frame to 30 frames, changed the normalization from mean-based to translation and rotation based on the first and last input points, respectively, and added a manual feature extraction in the middle. The design is illustrated with Figure 5. As we can see from the figure, the input remains the same with previous versions. However, after two convolutional layers, we manually extract and combine features based on human perception of how to make vehicle position predictions. In addition to the four input features, we add four manually selected features, and they are: x and y positions combined, x and y velocities combined, position and velocity of x, position and velocity of y. We apply two convolutional layers to those four selected features, and we pass all eight features as the input to a two-layer LSTM model with 512 hidden units and dropout rate of 0.2. After the LSTM model, we apply two dense layers with 256 and 120 units to make the final prediction of 30 frames, each having four features. Throughout the model, ReLU is used as the activation function for hidden features, and MSE is used as the loss function. The total number of parameters for our final CLDNN model was 3,334,208.

It is noteworthy to point out that we experimented with different optimizers, and we found that while Adam optimizer converges faster, it converges to worse position compared to SGD optimizer. As a result, for the sake of better performance, we chose to use SGD for CLDNN.

4 Experiment Design and Results

4.1 Problem Workflow and Setup

As a group of four, we decide to each work on the problem and aggregate findings to achieve the best result. We hold regular meetings to share our ideas and models with each other. Since we work on the problem individually, the setup of each of us is slightly different.

In general, we decided to use virtual machines and cloud computing platforms to perform the training and prediction tasks. We initially launched the data visualization and exploratory analysis on DataHub, but during the training phase, we noticed that DataHub has broken GPUs that caused a very slow training process. As a result, Dancheng, Brandon, and Aman switched to Google Colab, while Trevor borrowed his roommate's computer to perform model training.

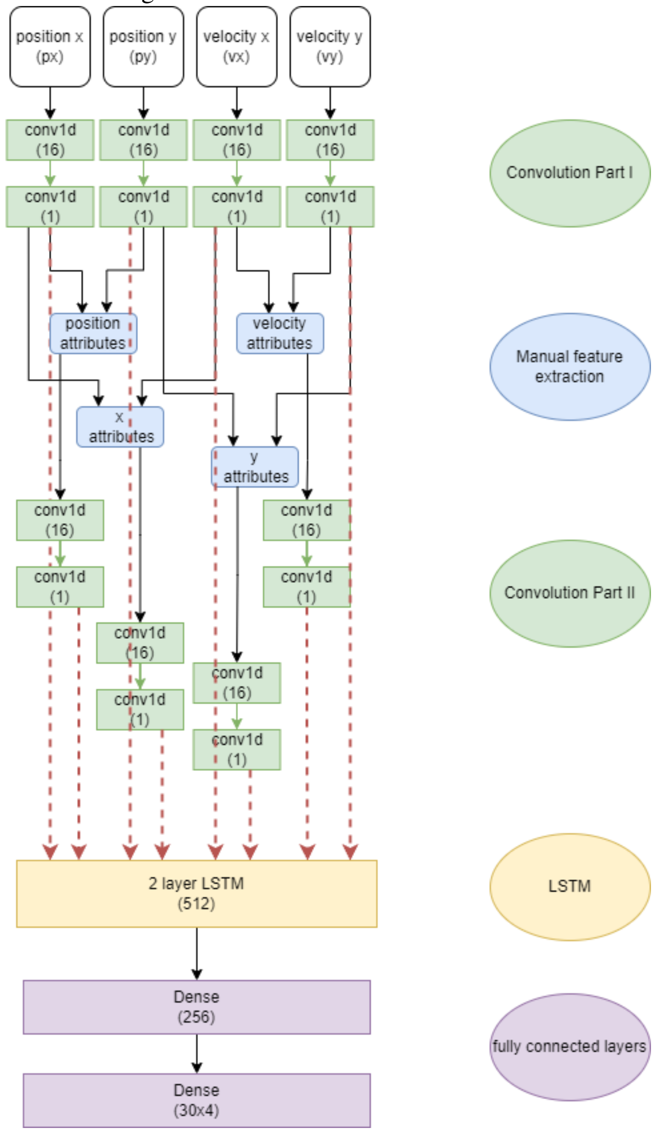
With Google Colab, the models are trained using a randomly selected A100 or T4 GPU, with Xeon 2.3 GHz as CPU. Trevor's roommate has a RTX 3080ti GPU. All code are written with PyTorch and Python 3.8 on a Jupyter notebook. As this is only a milestone progress report, we want to emphasize on the overall performance of a model. Thus, we have not yet fine-tuned the model and used only standard hyperparameters for training. We set the learning rate to be 1e-4, with MSE as the loss metric and using Adam as the optimizer. We run the model for a maximum of 200 epochs, but we use manual early stopping once the model converges on validation loss. The final weight is selected to be the one with least validation loss. The models are trained with batch size of 128.

Since we designed, tested, and reported four types of representative models, and these experiments are done over weeks, we do not have a uniform optimizer and multi-step prediction mechanism. The model with best performance, CLDNNv4, predicts 30 frames all together. It uses a SGD optimizer with learning rate of 1E-3 and momentum of 0.9. During the experiment, we had to perform normalization with the batch due to the limitation of RAM.

Our code is available at [1]

[1]`https://github.com/bjericks/cse251b-project`

Figure 5: CLDNN v4 model illustration



4.2 Performance of some representative models

4.2.1 CNN

One particular run of the proposed CNN model converges to a very high training loss as shown in Figure 6, whereas there have also been more successful runs that converges to much lower values, as seen in Figure 7. This is likely due to a few issues not yet addressed during pre-processing:

- The 3D tensor generated from pre-processing can be very sparse. The input data for a given scene may contain fewer than 60 agents, and the data uses a bitmask to evenly distribute the agents within the tensor. In turn, it can be difficult for filters in the convolutional layer to capture patterns between multiple agents. When the initialization is in a bad shape, it prevents this CNN from learning meaning representations of the data.
- The convolutional layers may be applied across dimensions with little correlation in the 3D tensor such that the filters cannot learn meaningful patterns.

The Kaggle open testing set reports MSEs of 61.98 and 47.02 over two submissions. Improving the performance of this CNN will require some reorganization of the pre-processed data.

Figure 6: CNN model training loss with default random initialization

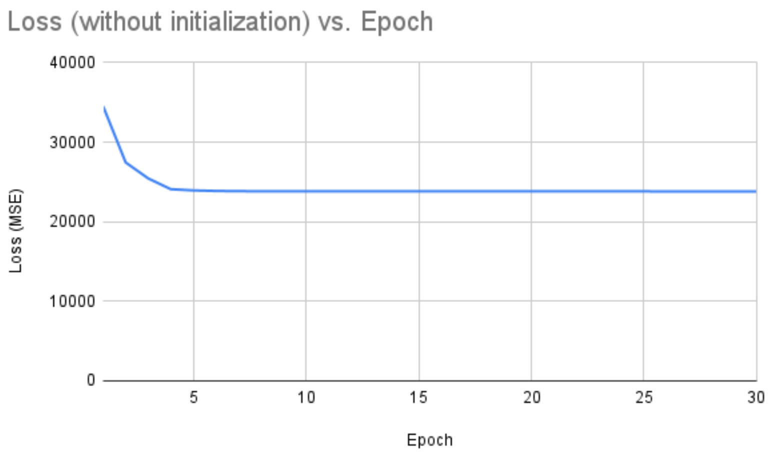
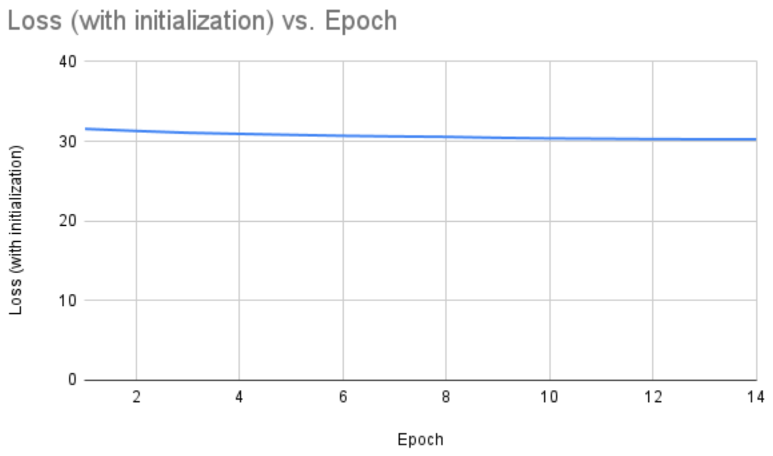


Figure 7: CNN model training loss with pre-trained weights



4.2.2 LSTM

The proposed LSTM model works worse than we expected. The model was able to learn an expoential decay in the training and validation loss in the beginning, but it converges to a relatively high loss at the end. The detailed loss curve is shown in Figure 8. The Kaggle open testing set reports MSE of 127.25.

As mentioned before, previous works have shown that LSTM could be suitable for this task, but our results do not support this claim. We suspect this is because of two possible issues. First, LSTM is beneficial in that it is able to capture long-term dependencies. While vehicle motion prediction seems to be a time-series task, the prediction is only based on 19 time stamps, which hinders the effective usage of long-term dependencies within the time-stamps. Also, LSTM is more sensitive to fine-tuning compared to CNN. We have not yet fine-tuned the model, which might cause severe performance degradation in prediction.

4.2.3 MLP

The three layer MLP model performs the best among all designs. Different from the previous two models, since this design is relatively simple, and the parameter size is very small, the model converges after the first iteration. The training and validation loss are summarized in Figure 9. We get

Figure 8: LSTM training and validation loss

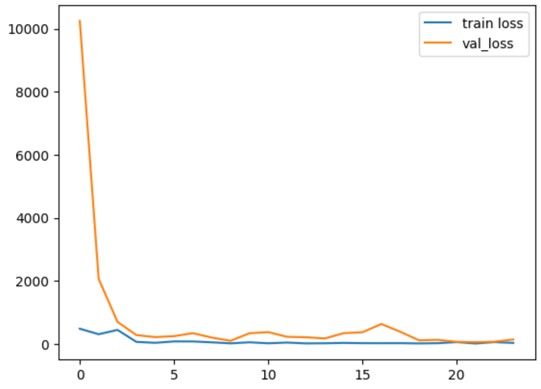
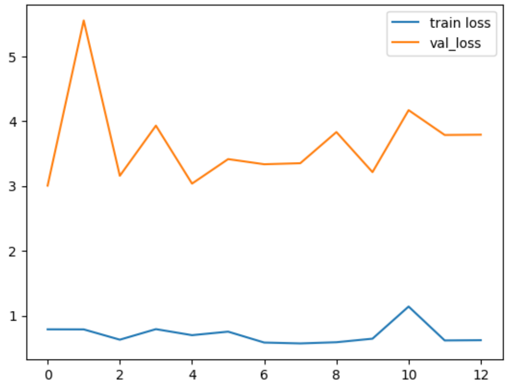


Figure 9: 3-layer MLP training and validation loss



2.12 MSE from the Kaggle open testing set, which ranks us position no.6 when writing the report. We also examined the performance of more complex MLP architecture. The four layer MLP described above performs slightly worse than the 3 layer MLP, achieving MSE of 2.35 on Kaggle.

We also obtained eight visualizations on the three layer MLP model from eight testing samples. The visualizations are shown on Figure 10. As we can see, the model seems to learn a pattern for vehicles that are moving. For example, in case 3 and 7, it makes excellent predictions that the vehicle is moving towards some direction. It also learns an accelerated velocity, though such acceleration might not be true for all samples. Furthermore, the biggest issue currently a MLP model faces is that it does not learn the pattern of a stopping vehicle. We suspect that it is because of the simplicity of the model making it only learn one pattern instead of multiple scenarios that could happen in the dataset.

Figure 10: 3-layer MLP sample illustration

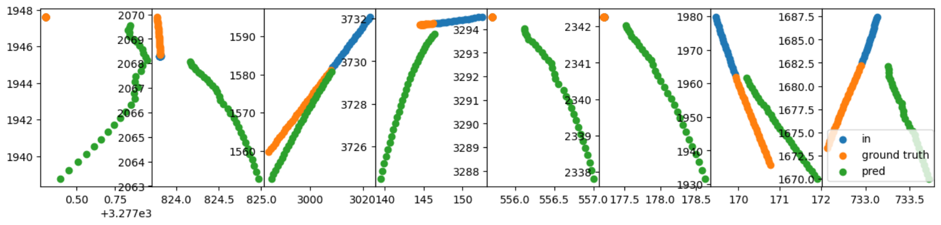


Figure 11: CLDNNv4 training and validation loss curve

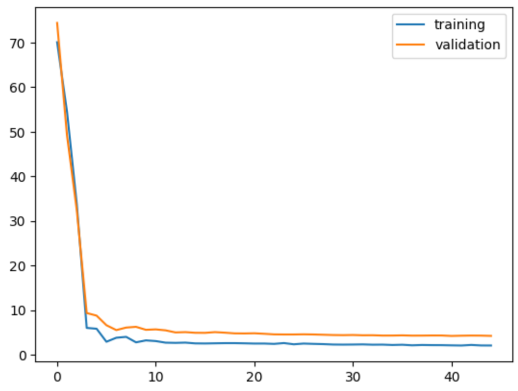
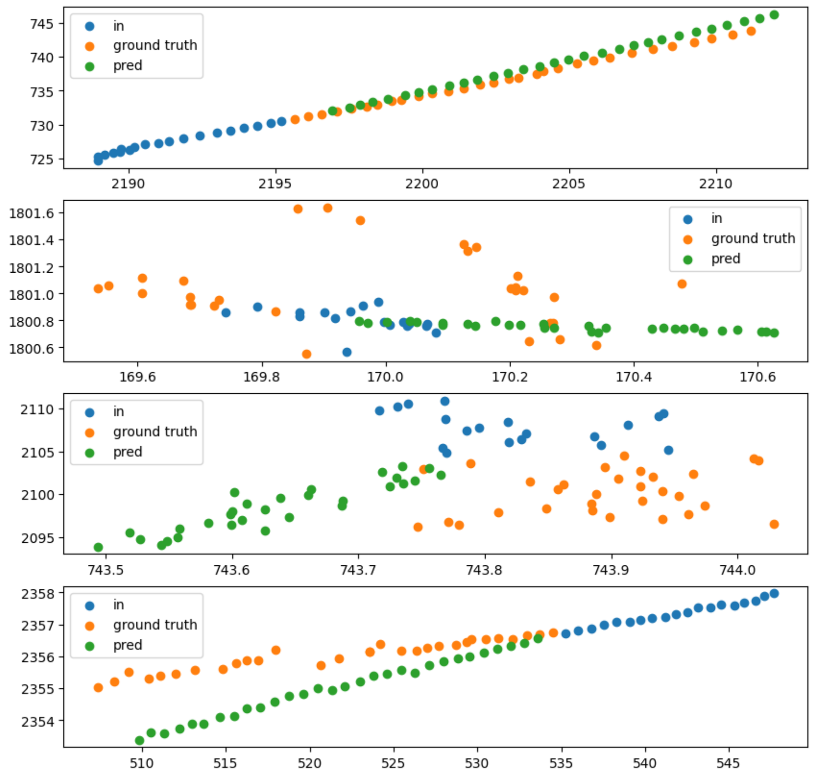


Figure 12: CLDNN Predictions for trajectory



4.2.4 CLDNN

As we mentioned above in the design section, we were surprised that the first version of CLDNN which predicts 1 frame at a time only performs marginally better than MLP model. This is likely due to the accumulated prediction errors from predicting 30 times. As a result, we changed the architecture to predict 30 frames together, and the performance is much better. As shown by figure 11, we could see an exponential decrease of the training and validation loss of the model in the first few epochs. The final validation MSE loss converges to somewhere near 4, and the Kaggle public test dataset reports a RMSE loss of 1.777.

We also obtained four visualizations of the model's predictions compared to the ground truth. As shown by figure 12, we could see that the model does a good job in predicting forward and stationary trajectories.

Table 1: Comparison of models on Kaggle public testing set

Model	Time per epoch (s)
CNN	150
LSTM	43
MLP 3 layer	26
MLP 4 layer	28
CLDNNv4*	450

Table 2: Comparison of models on Kaggle public testing set

Model	RMSE
CNN	47.02
LSTM	127.25
MLP 3 layer	2.12
MLP 4 layer	2.35
CLDNNv1	1.98
CLDNNv4	1.78

4.2.5 Brief comparison among models

In summary, we evaluated the performance of four types of models: a CNN autoencoder, a LSTM model, MLP models, and CLDNN models. Among them, MLP is the fastest, finishing one epoch in less than 30 seconds. The detailed time taken by each model is summarized in Table 1. [2] However, CLDNN performs the best, with the fourth version being significantly better than other models. Their performance on Kaggle public dataset is summarized in Table 2. Their numbers of parameters are reported in Table 3. It is important to point out that while the fourth version has a much larger number of parameters, they are mainly used by the dense layers to account on the larger hidden states of LSTM layers. Number of parameters do not have a direct causation of slowdown due to the high parallelism in GPU acceleration.

5 Discussion and Future Work

When developing our models for this competition, we found that pre-processing the input data with rotational and translational normalization was the most impactful feature engineering technique when it came to improving the overall accuracy of our predictions. All of our models that outperformed our 3-layer MLP baseline used normalization in pre-processing to some degree. In early versions of our CLDNN model, we applied translational normalization using mean-shifted position vectors. Later versions, including our best-performing model, further improved our RMSE scores by adding the rotational normalization component.

We also observed that our models formed more accurate predictions when all 30 output frames were predicted in the same forward pass, rather than predicting agents' positions for the next single frame with every forward pass. We suspect that when one frame is predicted at a time, and the frame predicted is then fed back into the network for the next forward pass, the model generates error that

[2]*: Due to the limitation of memory, we had to preprocess the data together with the model. The reported time is for both pre-processing and training

Table 3: Number of parameters of each model

Model	# of parameters
CNN	68580
LSTM	33684
MLP 3 layer	36500
MLP 4 layer	132676
CLDNNv1	54120
CLDNNv4	3334208

accumulates over time. With the compounded error of single frame prediction, the model is more likely to diverge from the ground truth.

The biggest bottleneck for our team in this competition was our limited access to GPU accelerators in our development environments. Early on, we decided to work primarily with Google's Colab environment, with one member of the team using the Pro version and the rest of the team using the free version. We chose this environment because with the Pro version, we had access to a more powerful accelerator than other free or otherwise affordable options such as Datahub. However, since Colab's authentication system is token-based, we were limited on the extent that we could develop and train our models in a given day. When we created our more complex CLDNN models, this restriction severely impacted our ability to test different model structures and conduct hyper-parameter tuning.

Nevertheless, over the course of this competition, we iteratively developed a CLDNN model that accurately projected the trajectories of agents in autonomous vehicle applications. To any beginner looking to develop deep learning models for similar applications, we suggest the following:

- **Pre-processing is key.** A deep learning model can only learn effectively if it's given feature vectors that lend themselves to learning. Normalization can lead to huge improvement.
- **Remember to have an iterative process for hyper-parameter tuning.** Deep learning requires a large amount of testing and tuning to reach optimal results, and it can be tempting to change multiple hyper-parameters at once to speed up the process. This approach can cause problems, however, when one has to determine what is or isn't helping produce better results.
- **More complex models aren't always better.** Adding complexity to a model can increase the risk of overfitting and lead to less accurate results. We observed this trend in the performance of our 3-layer and 4-layer MLP networks and in the first two versions of our CLDNN networks.

While our CLDNN implementation yields a significant improvement over our baseline MLP model, it does not account for the effects that neighboring agents have on a given agent's trajectory. As a result, we would like to explore architectures and mechanisms that serve to exploit the spatial relationships between agents in future work. For example, we could add an attention mechanism to our existing CLDNN model so that the model factors in nearby agents and lane markers that may influence a vehicle's movement. We are also interested in developing Graph Neural Networks as a means of utilizing these same relationships.

References

[1] Y. Jeong, S. Kim and K. Yi, "Surround Vehicle Motion Prediction Using LSTM-RNN for Motion Planning of Autonomous Vehicles at Multi-Lane Turn Intersections," in IEEE Open Journal of Intelligent Transportation Systems, vol. 1, pp. 2-14, 2020, doi: 10.1109/OJITS.2020.2965969.

[2] Jiang, R.; Xu, H.; Gong, G.; Kuang, Y.; Liu, Z. Spatial-Temporal Attentive LSTM for Vehicle-Trajectory Prediction. ISPRS Int. J. Geo-Inf. 2022, 11, 354. https://doi.org/10.3390/ijgi11070354

[3] M.-T. Duong, T.-D. Do, and M.-H. Le, "Navigating Self-Driving Vehicles Using Convolutional Neural Network," in 2018 4th International Conference on Green Technology and Sustainable Development (GTSD), 2018, pp. 607–610. doi: 10.1109/GTSD.2018.8595533.

[4] I. A. Tarmizi and A. A. Aziz, "Vehicle Detection Using Convolutional Neural Network for Autonomous Vehicles," in 2018 International Conference on Intelligent and Advanced System (ICIAS), 2018, pp. 1–5. doi: 10.1109/ICIAS.2018.8540563.

[5] A. Emam, M. Shalaby, M. A. Aboelazm, H. E. A. Bakr and H. A. A. Mansour, "A Comparative Study between CNN, LSTM, and CLDNN Models in The Context of Radio Modulation Classification," 2020 12th International Conference on Electrical Engineering (ICEENG), Cairo, Egypt, 2020, pp. 190-195, doi: 10.1109/ICEENG45378.2020.9171706.

[6] G. Sümen, B. A. Çelebi, G. K. Kurt, A. Görçin and S. T. Başaran, "Multi-Channel Learning with Preprocessing for Automatic Modulation Order Separation," 2022 IEEE Symposium on Computers and Communications (ISCC), Rhodes, Greece, 2022, pp. 1-5, doi: 10.1109/ISCC55528.2022.9912830.