CSE 151B Project Final Report

Binghong Li, Zimo Peng, Yong Liu, Brooks Niu
bil004@ucsd.edu, z5peng@ucsd.edu, yol146@ucsd.edu, rniu@ucsd.edu

1 Task Description and Background

1.1 Task and real world example

The task is to predict the future 6 seconds of positions (split into 60 data points) of the target agent based on the previous 5 seconds of positions (50 input data points). This task is important because it can help solve real-life autonomous-vehicle safety issues. These kinds of tasks play a very important role in a world where autonomous vehicles would gradually replace most human-driving vehicles.

Looking forwards, we can apply the predictions to the autonomous vehicles, such as autonomous trucks, to avoid collisions and improve moving routes. In this case, we can optimize the safety of the passengers and reduce real-world traffic accidents and protect the safety of the properties. The enhancement of the moving route can maybe save gas and keep the moving route clean.

We can also analyze the data of other moving vehicles, moving pedestrians, and random moving objects. We predict their movements and allow the autonomous vehicle to be ready for potential dangers caused by the random behavior of other objects on the road. We can use all these data to even re-architect the traffic systems for better safety and efficiency when travelling in day-to-day lives.

We can also apply the prediction for military use: hitting moving objects, blocking missiles, and spying on valuable targets.

1.2 previous research

- *Transformer Networks for Trajectory Forecasting* [2] talks about the advantages of using transformers for trajectory forecasting. Although the trajectory here is for human agents the authors' approach did not include the interactions with the environment or other agents. This makes the method viable in this vehicle motion forecasting challenge since here we also only predict single-vehicle agents' trajectories with only the location data. They compared LSTM, transformer, linear models; and TF(transformer) outperformed other models in 4 out of 5 data sets. Both TF and LSTM made great predictions but TF lines up better with the ground truth. The TF network has 6 layers in both encoder and decoder with self-attention and normalization. Thanks to the complexity and attention mechanism TF can neglect the missing observations if any while preserving its relative time stamp.

- *Spatio-Temporal Graph Transformer Networks for Pedestrian Trajectory Prediction* [1] discusses trajectory prediction among multiple agents with interactions. Although the model includes the interactions among multiple agents, it is also capable of predicting single agents with high accuracy, which fits the goal of this challenge. They used a transformer-based graph convolution mechanism to model agents' interactions. The data is encoded with both a temporal and a spatial transformer first and then fed into another sequential spatial-temporal encoder which can write to the graph memory that can learn the attention for the first temporal encoder. Thus, the model is capable of learning where to focus in the future from the spatial and temporal information which is stored in the graph memory.

Preprint. Under review.

1.3 The input and output

- The input for our refined MLP model:

$$Input = \{c_{x_i}, c_{y_i}, v_{x_k}, v_{y_k}, a_{x_l}, a_{y_l}, j_{x_m}, j_{y_m}\}_{i,k,l,m=1}^{50,49,48,47}$$

, c_x refers to the x coordinate and c_y refers to the y coordinate, together describing the position of the target agents. Since our input includes the trajectories of the first 5 seconds, it was slit into 50 timestamps (each for a tenth of a second). We have raw data with the size of 50 x 2 as inputs. To use the MLP model, We have to flatten it to a 100 x 1 dimension list. We also add velocity, acceleration, and jerk to this list with a final length of 388 as our model's input feature.

- The output $\{c_{x_i}, c_{y_i}\}_{i=1}^{60}$, c_x refers to the first coordinate and c_y refers to the second coordinate, together describing the position of the target agents. Since our input includes the trajectories of the later 6 seconds, it was slit into 60 timestamps (each for a tenth of a second). We have raw data with the size of 60 x 2 as outputs as prediction tasks. To use the MLP model, We have to flatten it to a 120 x 1 dimension list.
  Define training set as $S = \{x_i, y_i\}_{i=1}^{N}$

  Our predictive task is to find function f s.t.

$$\arg\min \frac{1}{N} \sum_{i=1}^{N} (f(x_i) - y_i)^2 \tag{1}$$

  is the minimized, as our competition uses Mean Square Error (MSE) to determine the final score.

  Our model utilizes the simple but powerful Multi-Layer-Perceptron, and there are many problems that can be solved with our model. MLP can be used to solve classification problems that are not linearly seperable. One classic example is the XOR problem, and essentially if we have enough layer of MLP, we can solve a lot of classification problems. Some other good examples: classification within criminal-justice problems, healthcare/insurance problems, loan-qualification problems, etc. Also the model can solve other trajectory prediction problems like boat trajectory prediction and airplane trajectory prediction because they share similar physical properties (i.e. displacement, acceleration, etc.) and can have similar abstraction as our data.

2 Exploratory Data Analysis

2.1 details of the dataset

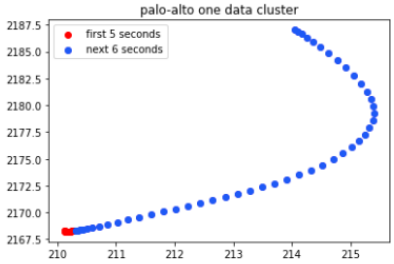
The train/test data size, and the dimensions of inputs/outputs

austin	– train: 43041	test: 6325
miami	– train: 55029	test:7971
pittsburgh	– train: 43544	test: 6361
dearborn	– train: 24465	test: 3671
washington-dc	– train: 25744	test: 3829
palo-alto	– train: 11993	test:1686

Each has num_test and num_train data clusters, and in each data cluster is a tuple of inputs and outputs. The dimension of inputs are 50×2 and the dimension of outputs are 60×2. This indicates that there are 50 timestamps of (x,y) position coordinates for input and 60 timestamps of (x,y) position coordinates for output. Figure 1 includes visualizations for single data samples.

2.2 statistical analysis

- The distribution of input positions for all agents



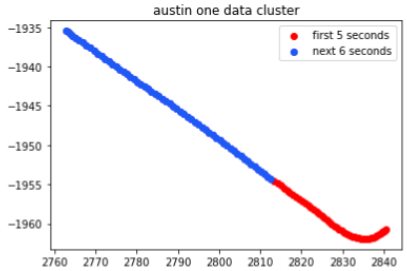


Figure 1: data samples

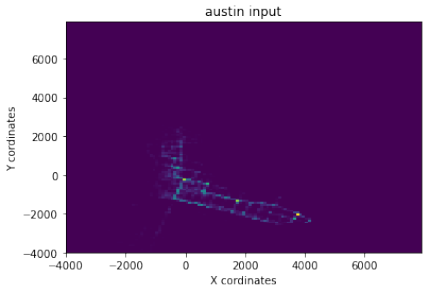


Figure 2: Input of Austin

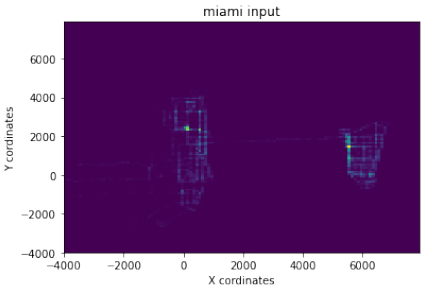


Figure 3: Input of Miami

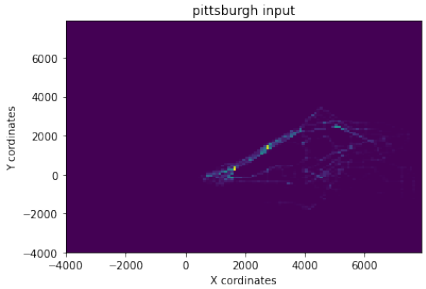


Figure 4: Input of Pittsburgh

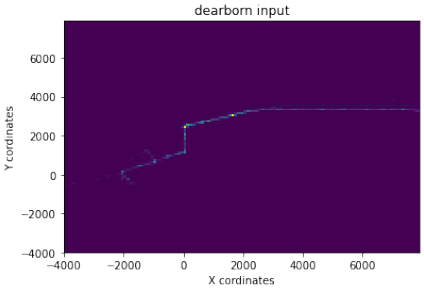


Figure 5: Input of Dearborn

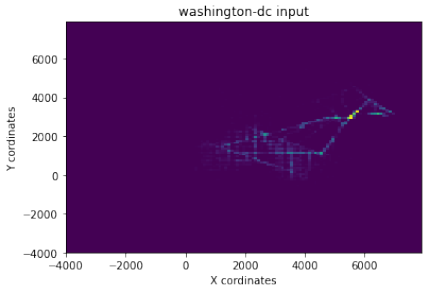


Figure 6: Input of Washington D.C.

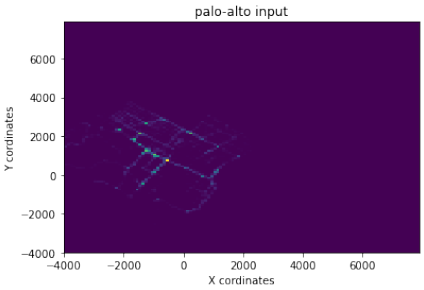


Figure 7: Input of Palo-alto

- The distribution of output positions for all agents

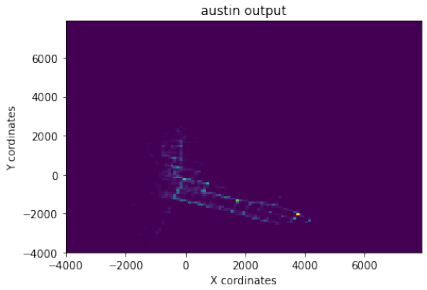


Figure 8: Output of Austin

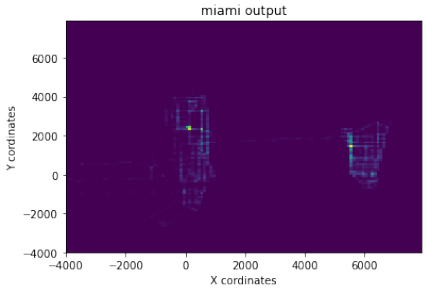


Figure 9: Output of Miami

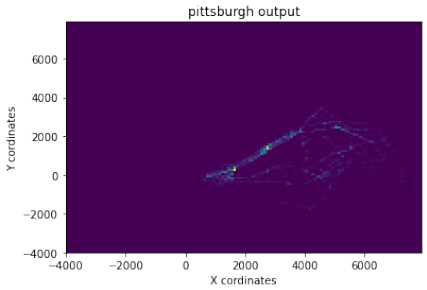


Figure 10: Output of Pittsburgh

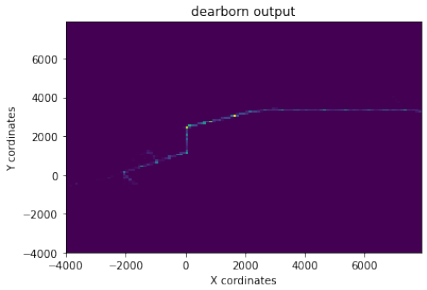


Figure 11: Output of DearBorn

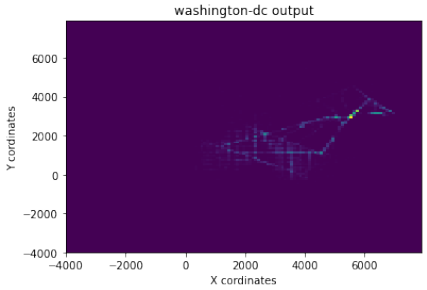


Figure 12: Output of Washington D.C.

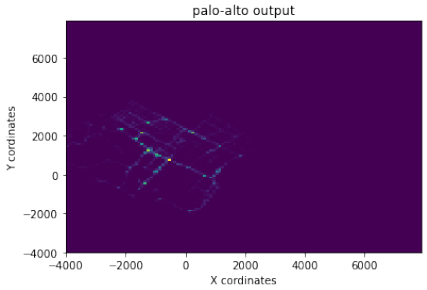


Figure 13: Output of Palo-alto

- The distributions of positions for different cities

Austin:
x coordinates mainly distributed in the range (-1000, 4500); y coordinates mainly distributed in the range (-2500, 2000). The heat map shows that the target agent moves in a routine of continuous small rectangular blocks

Miami:
x coordinates mainly distributed in the range (-1000, 1000), (5000, 7000); y coordinates mainly distributed in the range (-2000, 4000). The heat map shows that the target agent moves in a routine of two major rectangular blocks

Pittsburgh:
x coordinates mainly distributed in the range (0, 6000); y coordinates mainly distributed in the range (-500, 2500). The heat map shows that the target agent moves in a routine composed of one major repetitive trajectory and some dispersed locations

Dearborn:
x coordinates mainly distributed in the range (-2000, 8000); y coordinates mainly distributed in the range (0, 4000). The heat map shows that the target agent moves in a routine composed of one major repetitive trajectory.

Washington DC:
x coordinates mainly distributed in the range (500, 8000); y coordinates mainly distributed in the range (0, 4000). The heat map shows that the target agent moves in a routine composed of two major repetitive trajectories and some sub-trajectories

Palo-alto:
x coordinates mainly distributed in the range (-2000, 1000); y coordinates mainly distributed in the range (-2000, 3000). The heat map shows that the target agent moves in a routine composed of several repetitive trajectories.

We also did exploratory analysis beyond the above questions:
The data distribution of each city is affected could have been affected by multiple factors. Two would we like to point out are: the urban planning and social/population distributions, in other words, how city road systems are built and where the business/entertainment centers of the cities are. Some cities are organized by rigid small street blocks, while others are organized by blocks of random shapes. Some cities have one or several major streets which attract all the traffic, while others might be distributed. Most cities do have some central areas: some may have just one, while others could have multiple of those. All those above ideas could play into how and why our data is distributed like the presented heatmaps.

2.3 data processing

We used a 0.3 validation fraction to split the train data and validation data using the Scikit-Learn package. That is we use 30% of the train data as validation data set.

For feature engineering, we observed the given coordinates and the trajectories. Since the time stamps are also given, and since we want to provide more useful information when training the model, the first intuition is to add the velocity. The velocity feature did lower the loss by a great portion (60%). Then we were thinking acceleration would also make sense, and it did. For fun, we took one step ahead and look at jerk (the third derivative of distance with respect to time). Adding jerk in most cases stabilized the loss changes in different trainings with the same parameters. When we tried further derivatives, we experienced the "vanishing/exploding gradient problem" first-handed-ly, where the Jounce (4th derivative of displacement) and even more could become exponentially large, which is harmful to our process. So we stopped at the jerk.

We firstly tried to standardize the data but it did not work so well because standardizing data for ML problems is not helpful when the data does not follow the assumption of normal distribution, and while the distribution of data points within each data clusters were not as similar.

Then, we normalized our data using translation and rotation. By translation, we subtract the first coordinate from all of the positions to make a relative position for the path (all paths starting from origin). By rotation, we multiply each position vector with a specific vector to rotate the whole path (all paths starting in the same direction). Both of these methods can make our data more uniform and reduce the data noise when training our model.

We discovered that each city has different sizes of data and different patterns within the trajectories. With those in mind, we decided that we would take advantage of this information and train separate models based on data for each city, optimizing the advantage of knowing the specific patterns and characteristics of the cities individually.

3 Machine Learning Model

3.1 start with MLP

We start with simple MLP model with raw data as input: $\{c_{x_i}, c_{y_i}\}_{i=1}^{50}$, the length is 100. The output: $\{c_{x_i}, c_{y_i}\}_{i=1}^{60}$, the lenght is 120. The logic for this model is simply linear regression$Y = W\dot{x} + w_0$
We use sickit learn MLP package with parameters : activation function : Relu ; Learning rate: 0.005;

optimizer: Adam; 5-layer architecture; (32,32,64,64,128); By using mean squared error as loss function, we reach to 830 loss by now.

3.2 final model MLP

After several experiment with LSTM and SeqtoSeq, we end up with MLP model by applying engineering tricks, which is including shifted and rotated position data, x and y velocities, accelerations and jerks. The input feature now is: $Input = \{c_{x_i}, c_{y_i}, v_{x_k}, v_{y_k}, a_{x_l}, a_{y_l}, j_{x_m}, j_{y_m}\}_{i,k,l,m=1}^{50,49,48,47}$ with the length of 388. The output features remain the same as the baseline MLP. We also apply normalization to our input and output feature. After tuning the parameter, we set our parameter as these: activation function : Relu ; Learning rate: 0.001; 4-layer architecture; (32,64,64,128); By using mean squared error as loss function, we reach to 19.3 loss finally.The training log-loss curve for one of the cities' target agents looks like this:

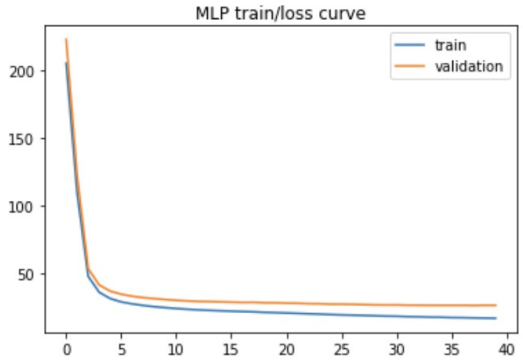


Figure 14: Training Loss Curve

3.3 alternative Model we tried

- **Multi-layer Perceptron**
  - 4-hidden-layer perceptron with $hidden\ dimension = (32, 64, 64, 128)$
  - ReLU activation function, Adam optimizer, adaptive learning rate (begin with 0.001), early stopping (max epoch = 400), MSE Loss
  - Input: normalized x,y position (shifted to (0,0) and rotated toward (1,1)); x,y displacement; x,y velocity; x,y jerk (difference of velocity between time stamps); 50 timestamps in total
  - Output: normalized x,y position; 60 timestamps in total
  - Reference: sklearn
- **Vanilla Long-short Term Memory**
  - 1 layer Vanilla LSTM
  - learning rate = 0.0005, 120 epochs with $hidden\ dimension = 512$
  - Input: x,y displacement; 50 timestamps in total
  - Output: x,y displacement; 60 timestamps in total
  - Reference: pyTorch
- **Sequence to Sequence**
  - 1 layer LSTM encoder, decoder
  - learning rate = 0.0005; 1200 epochs with $hidden\ dimension = 512$; teacher-forcing ratio = 0.5 with mixed teacher forcing
  - Input: x,y displacement; 50 timestamps in total
  - Output: x,y displacement; 60 timestamps in total

- Reference: lkulowski

We did not design new model architectures. Instead, we adopted the established model architectures and focused on data processing and hyper-parameter tuning.

In data processing, we first shift all data to be starting at (0,0), which reduces the different location noises so model can better focus on the pattern of the paths. We also rotated each path so they all start moving in the same direction as vector (1,1). This servers the same purpose in reducing unnecessary noise from data. We tried normalization (i.e. convert the data into a fixed range) but since the paths can vary a lot in lengths this approach added noises instead of reducing them. We also tried dropout layers for LSTM and Seq2Seq but it did not improve the models' performance either.

LSTM enabled us to use the information from the previous layers to predict the next 60 steps for each target agent. We trained our model to predict the 110th step based on the 109 steps given. Then we set the unroll length = 60 in the generation function to let the model generate 60 times based on previous generations to make the multistep prediction.

For seq2seq, we used the LSTM encoder layer to encode the first 50 timesteps, then pass the last hidden state into the LSTM decoder layer. Then the model decides whether to use the previous prediction or last step's ground truth to train based of the teacher-forcing ratio.

At first we chose a relatively big learning rate of 0.001 for each model. But as the training progresses, the total loss was oscillating during the training of latter 20 epochs. So we reduced the learning rate to 0.0005 to ensure both efficiency and accuracy. We also tried lower learning rate for different models but 0.0005 turned out to be the most generic one to use.

4 Experiment Design and Results

4.1 Our best-performing design

In our process of training and testing, we mainly had two platforms, local PCs and UCSD Datahubs. Local environments were constructed with GeForce-RTX-2070-Super and i7-9800x CPU/AMD 5800x. On UCSD Datahub the configurations were: 2022.2-stable, Python 3, nbgrader (1 GPU, 8 CPU, 16G RAM), GeForce GTX 1080, around 2GB. For different phases of the project, while we were using self-developed deep learning models with pytorch, we operated on both local and UCSD Datahub environments for good GPU quality. As we finalized our model with Scikit-Learn's built-in MLP, we decided to operate more on local environments for better CPU power.

For the final submission, as for our optimizer, we used the vanilla Adam optimizer. We did not use momentum as it is not recommended by Scikit-Learn with Adam solver. We started with a learning rate of 0.001 and used the package's "adaptive" learning rate decay method, such that in situations of two consecutive epochs fail to decrease training loss by at least tolerance of 0.0001, or fail to increase validation score by at least the same tolerance, the current learning rate is divided by 5. We also utilized early stopping to better accommodate the validation results to combat over-fitting. Most of the other parameters stayed default to the Scikit-Learn's MLP package. Some caveats were alpha (strength of the L2 regularization) as 0.001, validation fraction as 0.3, and the hidden layers as (32,64,64,128). Most of the design choice came from prior experience when building our MLP and some fiddling with the later model.

To predict the 60-step prediction into the future, although some other models required explicit information of the output dimensions, for the final MLP, the MLP package was able to know that we needed 120 (60×2 after processed) output.

While analyzing the city information, after plotting out the heat-maps and taking some random samples of the paths, we decided that the dataset were different enough from each other and took two approaches to deal with the difference: 1) one hot encoding with a length 6 vector, and 2) training different models with each of the dataset.

Since we used MLP and used learning rate decay and set early stopping to be true, while we set the max-epoch to be 400, it usually ran bout 35-45 epochs before early stopping(using validation fraction of 0.3) with batch size of 32.

4.2 Representative models

	MLP	LSTM	Seq2seq
Training Time	Under 10 min (total)	4 hours (total)	3-4 hours (total)
Final Loss	19.3 (final private)	137.23 (validation)	37(final private)
Num of Parameters	8	6	11

Figure 15: 4.2 Table

Loss value progressions	
Approaches	**MSE Loss**
Naive MLP (Raw data)	800-850
MLP Adding standardization attempt (failed)	> 2000
LSTM (80% data)	200-300
MLP Adding velocity to data	150-250
MLP Adding Jerk to data	100-200
MLP Adding derivatives beyond jerk to data	> 300
Seq2Seq with LSTM (20% translated & rotated data)	30-40
MLP Adding Coordinate translation	22-25
MLP Adding Trajectory Rotation (**FINAL**)	19.3
MLP Adding path length normalization	Unfinished

Figure 16: 4.2 Table

To improve the training speed of MLP and LSTM, we enabled early stopping, which both saved many epochs and prevented over fitting. For Seq2Seq, we only used 20% of total data, which lessened the training time from 12 hours to 4 hours for six cities.

Typically the more complex structure is able to perform better on learning complex patterns when given sufficient data, the lower initial loss when no fine-tune of the model is applied. LSTM and Seq2seq models are able to reach the loss range of 30-140 without fine-tuning or adding features. MLP performs poorly without fine-tuning, receiving a loss of 800.

MLP has 8 parameter (activation="relu", solver="adam", early stopping=False, learning rate="adaptive", max iter=400,alpha=0.001,hidden layer sizes = (32,64,64,128)), LSTM has 6 parameter (learning rate = 0.0005, epochs = 50, input dim = 2, hidden dim = 256, layer count = 1, output dim = 2), and Seq2seq has 11 parameters (hidden layers, drop out rate, input tensor, target tensor, n epochs, target len, batch size, training prediction = 'recursive', teacher forcing ratio = 0.5, learning rate = 0.01, dynamic tf = False).

4.3 **Problem C [1 points]:** Play with different designs of your features and models. Report the following for your best-performing model design:

Our final rank is 8 with a loss value(MSE) of 19.30814 (public), 19.33329 (private)

Figure 17-19 are three of the predicted trajectories with our refined MLP. Our refined final multi-layer perception generates predictions quit well and can learn non-linear paths somewhat efficiently as well. We had a final private MSE loss of 19.31, and our best one had a private MSE of 19.33.

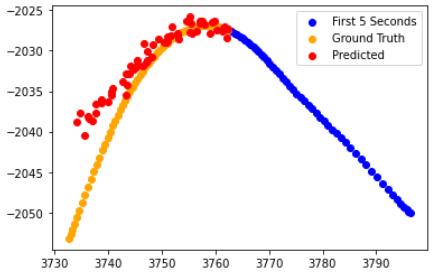


Figure 17: MLP predicted trajectory

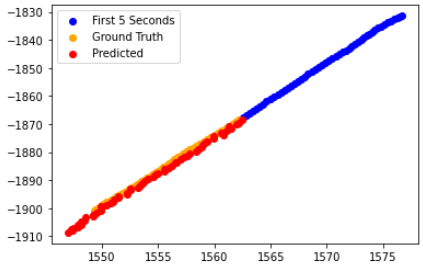


Figure 18: MLP predicted trajectory

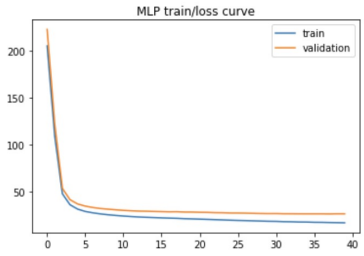


Figure 19: MLP train/val loss curve

Figure 20: Kaggle Ranking

5 Discussion and Future Work

5.1 Result Analysis

In our opinion, the most effective feature-engineering strategy is to compute velocity, acceleration, and jerk and add them to the input string. We also thought about doing so for the angular velocity, acceleration, etc. but did not get to implement them. One hot encoding consisting of the city information could be helpful as well. From the first time we submitted our results, we improved our rank quite a bit. In order to do so, we mostly followed the procedure: 1) first minimize the loss on local training and testing process by tuning hyper-parameters or altering the model, then 2) upload on Kaggle for official evaluation. We plotted loss curves along with prediction v.s. ground-truth graphs to help us determine the time and pace for the loss to converge and allows us to modify the model for better convergence, and also to combat potential over-fitting.

Along the way, our biggest bottleneck occurs when debugging the LSTM and Seq2seq models. These models take a long time to train and thus require a longer span to debug and improve (things are indeed quite different in Deep-Learning settings). Our first suggestion to deep-learning beginners is to get familiar with the task and the given data. Data visualization and primary data analysis helped us get a basic idea of the data and it might even have been possible to observe some obvious patterns, which could provide great inspiration for later data processing and adding features. Then, the beginner should try to organize and normalize the given data, reducing noises and concentrating the focus. From there, the beginner should start with simple models like MLP. The idea behind this is that, it is always a good idea to start with a simple model and gain primary experience in processing the data and training the model. The simple MLP model also has a lower complexity in implementing and debugging too, along with its fast training time. On the other hand, multi-layer perceptron actually performed pretty well in these kinds of prediction tasks for our case. Train the model and observe the process of training; use the validation data to determine if we are going in the right direction. Repeat and improve the model in similar patterns.

If we have more resources in terms of time, ideas, and training resources, we would try some more complex models like transformers, auto-regressive models, fusion models, or Seq2seq with attention. Combining different models and applying tuning techniques would make huge enhancements. We would try to inverse the starting point and the endpoint of the trajectory. We would try to add more features with the given feature. For example, we could add angular velocity or acceleration. Regarding our final submission of MLP, we would try another approach: train the model with all data of the six cities at once and then fine-tune a model according to each city.

References

[1] Cunjun Yu et al. "Spatio-temporal graph transformer networks for pedestrian trajectory prediction". In: *European Conference on Computer Vision*. Springer. 2020, pp. 507–523.

[2] Francesco Giuliari et al. "Transformer networks for trajectory forecasting". In: *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE. 2021, pp. 10335–10342.