Autonomous Vehicle Motion Prediction

Ethan Nagola
University of California, San Diego
enagola@ucsd.edu

Tianye Fan
University of California, San Diego
t4fan@ucsd.edu

Rickan Li
University of California, San Diego
ril027@ucsd.edu

Pranay Tulugu
University of California, San Diego
ptulugu@ucsd.edu

Github: https://github.com/ptulugu/AVMotionPrediction

1 Task Description and Exploratory Analysis

A) The task is to predict the position of a car based on cars' positions and velocities around the car. It is important because we could apply this prediction to autonomous cars so that a car that is driving could predict the positions of the cars around it based on their current position and velocity and act accordingly to drive safely based on the environment. The input is given by the dimensions [60,19,2] for both P-in and V-in giving a dimension of [60,19,4] where each of the 60 data points are individual objects in the scene and they have 2 values that locate their position and 2 values that are their velocity at 19 different times. The output is given by the dimensions [60,30,2] for both P-out and V-out giving a dimension of [60,30,4] which is the same 60 data points but with 30 different locations that we need to predict given the input data. Something important to note is that while we can load velocity and position we are only looking at position specifically. In addition to the data that we are pulling there is also data that represents the number of lanes with the dimenstions of [num lanes, 3] where each lane is represented by x,y,z (z is always 0 in this case). Furthermore, the input data has unique identifiers for each car in the scene given by 00000000-0000-0000-0000-0000XXXXXXXX where X are numbers that identify each car and there is an agent-id with a similar form but it defines the car that we want to predict in the scene. Furthermore, the data has a scene id defining which scene it is and a city string that is either "PIT" or "MIA".

B) From researching some resources I found two articles that were interesting: Deep Learning-based Vehicle Behaviour Prediction for Autonomous Driving Applications: a Review by Sajjad Mozaffari, Omar Y. Al-Jarrah, and Mehrdad Dia as well as Prediction in Autonomous Vehicle - All You Need To Know by Atul Singh. The article by Mozaffari et al. talks about many different approaches towards solving the issue including RNNs, CNNs, mixtures of the two, fully connected NN models, and more. The article revealed that RNN models had an overall advantage of processing temporal dependencies, CNN models had an overall advantage of processing spatial dependencies, and that a combination of both had the overall advantage. Furthermore, the article looked at Graph Neural Networks which worked well with the graph-like structure of their traffic data (1). The article by Singh talks about the differences between model based and data based approaches to this problem. He then goes on to talk about their advantages and combines them into an approach that uses the Naive Bayes Classifier to predict (2). Overall, from looking at the different approaches online, we found that there is no such thing as a wrong model and that it is how you approach using that model which is important.

C) Going into evaluating and visualizing our inputs and outputs, our inclinations on what our model/models' abstraction would look like akin to this:
Given a respective agent's, *represented by an arbitrary variable labelled C*, with position and velocity at a given index, we would have (C_{pxi}, C_{pyi}) and (C_{vxi}, C_{pyi}) where p and v are the x,y coordinates for $n-th$ vehicle $p_t^n, v_t^n \in \mathbb{R}$ with given timestamps, t.

Linear Regression-type Model:
The model would take an input of $p^t_{x,y}$ and return an output $p^{t+1...t_{prediction}}_{x,y}$, ultimately we are looking at the next 30 time steps. Training and loss factors addressed by mean-squares $\min_w ||Xw - y||_2^2 + \alpha||w||_2^2$, α is a complexity parameter used for optimization in minimal-cost pruning.
We then also wanted to further embedding with velocity: we noticed that certain agents' (vehicles) had velocity values of 0, and that vehicle's position was not moving. So, prediction positions could be better predicted or enhanced given an initial coordinate and trajectory, then its future position could be predicated with constant velocity.

LSTM Model: Encoder-Decoder model
Wanted to continue with sequential-type feeding, but we wanted the input to be $(p^t_{x,y}, x^t_{x,y})$ and return the same output of positions in the next defined time steps. But, we had various methodologies for training:
We wanted the velocity to be embedded like additional feature to our input, so the most streamless output would be reflective of the input $(p^t_{x,y}, x^t_{x,y})$ following the standard equation ...

$$h_i(t) = \tanh\,(s_i^{(t)})\sigma(b_i^\circ + \sum_j U^\circ_{i,j} x_j^{(t)} + \sum_j W^\circ_{i,j} h_j^{(t-1)})$$

this is derived from PyTorch implementation of LSTM:

$$f_t = \sigma(U_f\ x_t + V_f\ h_{t-1} + b_f)$$

$$i_t = \sigma(U_i\ x_t + V_i\ h_{t-1} + b_i)$$

$$o_t = \sigma(U_o\ x_t + V_o\ h_{t-1} + b_o)$$

$$g_t = \tanh\ (U_g\ x_t + V_g\ h_{t-1} + b_g) \text{ a.k.a. } \tilde{c}_t$$

$$c_t = f_t \circ c_{t-1} + i_t \circ g_t$$

$$h_t = o_t \circ \tanh\ (c_t)$$

We really liked the mathematical model abstraction of the Encoder-Decoder LSTM model, where the prediction output is reliant on conditional probability: So for training,

With given input sequence of features for a given vehicle, C, ($C = \{c_1, ..., c_{time}\}$), and respective output sequence, O, ($O = \{o_1, ..., o_{time'}\}$) which gives us probability of $p(O|C)$. The encoder uses previous cell states to approximate probability with probability distribution of
$\Pi_t^{time'} = (o_t|CellState_t, o_1, ..., o_{t-1})$,
the decoder creates and approximates probability distribution with its own cell state based on previous **decode** iterations on encoded inputs $\Pi_t^{time'} = (o_t|CellState'_{t-1}, o_1, ..., o_{t-1})$. So this means that the decoder isn't predicated on previous outputs from the LSTM, but actually makes probabilistic output decision based on its own probability distribution, for $o't$, to update the decoder state. This obviously has its advantages and disadvantages. We liked this with the impression that it's representative of a learned prediction rather than a more calculated, specified output. We thought it would offer some value of validity to our model and help us during validation.

I think the encoder/decoder, although slightly more complicated implementation, we could do some interesting things during implementation. Standard implementation of the encoder-decoder model

often use library implementation of softmax that already have predefined use of LSTM hidden states, feature sequences, and attention mechanisms. Some ideas we had:

- we would have additional layers in our LSTM, like having a decoder process that also facilitates loss analysis and optimization over its hidden states.
- defining differing feature maps like have sequential outputs standardize the velocity feature as weights, and then train over initial positions and use features like lanes from dataset to check relative position of multiple agents.

This lead to further discussion on computational efficiency suggested dimensional reshaping to flatten position and velocity inputs, which lead to certain worries about overfitting, which we had trouble expressing in math model...
We wanted to additional features and layers to incorporate as much as the data as possible. Certain ideas like more explicit use of hidden states in the LSTM, explicitly initiating them in our model to be fed to our layers, or LSTM cell stacking could benefit the overall accuracy of prediction. But, model performance and overfitting lead us to keep our abstraction simple, and we agreed that being aware and conscious of output values (even dimensions) as we knew the use of Library import for our layer facilitates easier implementation that often leads to naive ideas and ignorance to if layer is adding computational load with no real affect on output.

Math Model on Other Tasks

I think both the Linear Regression and LSTM model abstactions could be used a many other tasks, a lot of time-series prediction tasks and moving average problems. The task to solve is representative on how we fit the features of said task into the model, so instead of position and velocity if we have dates, stock price, and amount sold we can try to predict stock values.

Examples of task prediction:

- Sales of items in future
- Housing cost in future
- Marathon times based on shorter-length races
- Production of items in future
- etc.

2 Exploratory Data Analysis

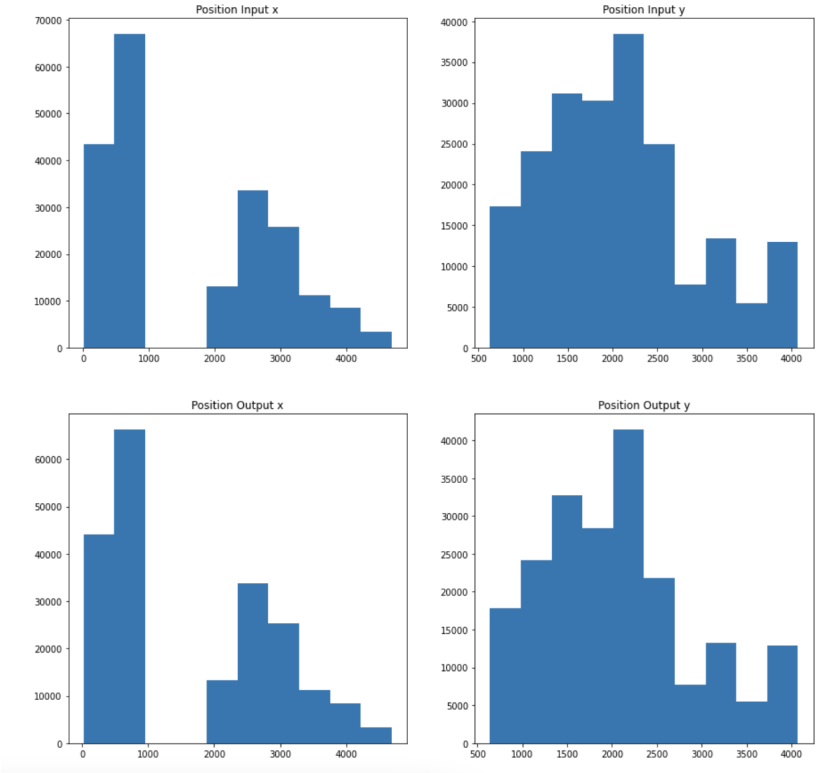
Problem A [0.5 points]: Run the provided Jupyter notebook for loading the data. Describe the details of this dataset:

The size of the training data is 205944, the dimensions of the inputs of the training data consists of a P-in and V-in of [60,19,2] giving us [60,19,4], and the dimensions of the output of the training data consists of a P-out and V-out of [60,30,2]. The size of the test data is 3200, the dimensions of the inputs of the test data is also a P-in and V-in of [60,19,2] giving us [60,19,4] while it seems that there is no output since this is just the validation.

Problem B [0.5 points]: Perform statistical analysis to understand the properties of the data:

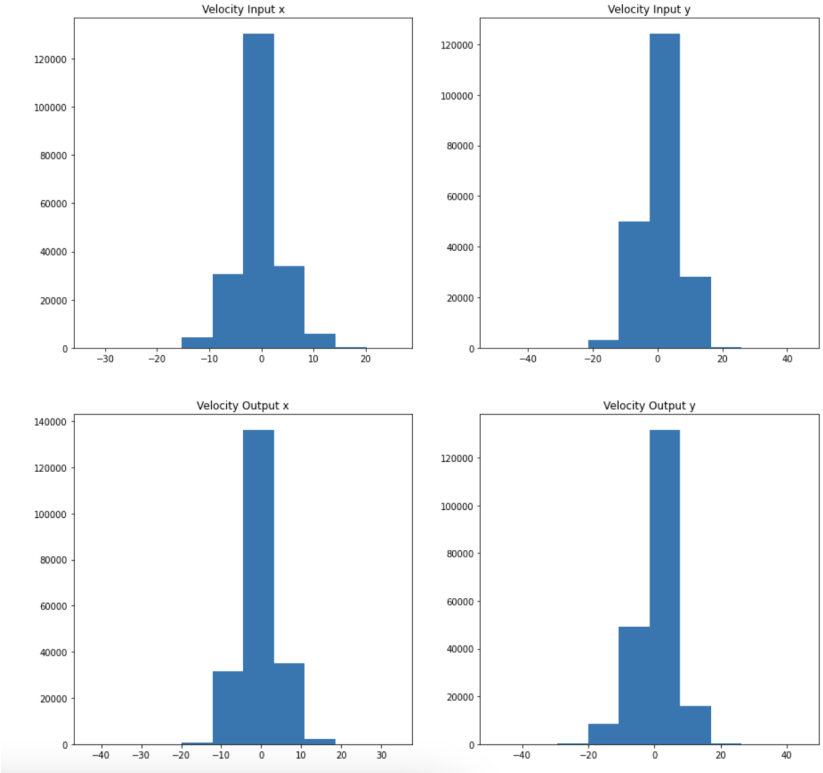
To find the distribution of the input positions we looked at the initial position of each car for every element in the input data by going through and storing it into a csv file that we would read with pandas and graph the x and y positions as a histogram. The distribution of input positions for all agents is bimodal and slightly skewed right in the first column (labeled position input x) and unimodal and slightly skewed right in the second column (labeled position input y) as shown by the position histograms in the first row.

To find the distribution of the output positions we looked at the initial position of each car for every element in the output data by going through and storing it into a csv file that we would read with pandas and graph the x and y positions as a histogram. The distribution of output positions for all agents is bimodal and slightly skewed right in the first column (labeled position output x) and unimodal and slightly skewed right in the second column (labeled position output y) as shown by the position histograms in the second row.



To find the distribution of the input velocity we looked at the initial velocity of each car for every element in the input data by going through and storing it into a csv file that we would read with pandas and graph the x and y velocities as a histogram. The distribution of input velocity for all agents is normally distributed both in the first column (labeled velocity input x) and in the second column (labeled velocity input y) as shown by the velocity histograms in the first row.

To find the distribution of the output velocity we looked at the initial velocity of each car for every element in the output data by going through and storing it into a csv file that we would read with pandas and graph the x and y velocities as a histogram. The distribution of output velocity for all agents is normally distributed both in the first column (labeled velocity output x) and in the second column (labeled velocity output y) as shown by the velocity histograms in the second row.



Problem C [1 points]: Process the data to prepare for the prediction task. Describe the steps that you have taken to process the data:

Our data processing is pretty straight forward and actually didn't deviate much from the example code given to us. Breaking it down...

Preprocessing mainly involved just getting the respective data in the form of the "pickles". Allowing us to identify the data.
Processing involved stacking the position and velocity values (for both input and ouput) into float tensors, indexing them/embedding them with identifiers (agent and track) and putting them into the Pytorch Dataloader. This was done in the batch process, where we viewed each batch like a scene. Then going into the model, we would fit inputs into (4, 19, 60 * 4) and targets into (4, 30, 60 * 4) shapes, which represents their respective timestamps, also reshaping the output from the model and target to check loss on a given "agent" car.

Lane Data: We ultimately did not use the lanes as an effective feature for our model. Beginning the competition we found that data from the lanes could potentially be something very useful. We found that there can be a varying number of lanes but every lane is given an x,y, and z coordinate so we get the dimensions of [number of lanes, 3]. The lanes can be relative to the positions of the cars in the scene and can be useful to use in relation to p-in and P-out for our prediction model. Something else important to note is that z is always 0 so we can ignore it when using the dimension of the lanes. Another important thing to note is that each car has its own unique id to it that allows the data to keep track of the position for each specific car. For each scene with the cars, each car has a unique id (doesn't have to be unique in between different scenes that we want to predict). We definitely saw the lane data as feature used in some way to carry weight for prediction, helping build a better

defined LSTM unit. For a given scene, we could also cars relative position to lane as indicators for surrounding cars or factors that could influence trajectory (i.e. switching lanes).

3 Deep Learning Model

Problem A [1 points]:Describe the deep learning pipeline for your prediction task and answer the following questions:

What are the input/output that you end up using for prediction after pre-processing?

For linear regression model, the input we use for prediction is in dim (2 * 3200 * 19), which contains the positions of the target vehicle. And the output is (2 * 3200 * 30) which is the prediction of the positions of the target vehicle in next 30 timestamps.

What is your loss function? If you have multiple alternatives, discuss your ideas and observations:

We use RMSE as our loss function as it is used in the leaderboard for calculation score. We also think about using Square Loss and it also is a good indicator of how the model is doing.

How did you decide on the deep learning model? Explain the rationale given the input/output.

We started designing the model intimidated by the choices and how to go about making an effective deep model for the dataset. We started with ideas and concepts from class and did further research online. We narrowed and started to finalize our model through consensus on which would work best and as something that everybody in the group understands. Much of the specific features and parameters of our model were fitted based on trial and error. Through this and collaboration we were able to gain clearer and clearer ideas of what our final model would be and how it would work. We are still making adjustments and learning how to make a more effective prediction model.

Problem B [1 points]: Describe the models you have tried to make predictions. You should always start with simple models (such as Linear Regression) and gradually increase the complexity of your model. Include pictures/sketch of your model architecture if that helps. You can also use mathematical equations to explain your prediction logic

Use an itemized list to briefly summarize each of the models, their architecture, and parameters, and provide the correct reference if possible.

Here is the table of our models and corresponding statistics:

Model Name	RMSE	Training Time	Number of Parameters
Linear Regression	~2.5	40 Mins for 20,000 samples (SK learn)	6000 + 6000 = ~12000
MLP Regressor	~3.5	20 Mins for 200,000 samples	~12,000
Feed Forward 1 layer	~8	30 Mins for 200,000 samples	~12,000
Feed Forward multi layer	~6.5	~1 hour for 200,000 samples	~100,000
Simple RNN	~20	~2 hours for 200,000 samples	~135,000
Encoder Decoder RNN	~15	~2.5 hours for 200,000 samples	~240,000
Encoder Decoder using GRU	~12	~3 hours for 200, 000 samples	~250,000

We tried a Linear Regression model to predict the positions in next 30 timestamps based on previous positions of the target agent only. Since the training set is too big, we only fetch 10000 samples to train our model. Furthermore, we separate the x and y positions and will try to use x to predict x in the future, and using y to predict y. So our idea is simple here, feed the model with input positions, and provide positions in next time stamp as output labels. Since each linear regression model only have one output, we will need to create 30 models to predict positions of 30 timestamps with each model providing correct positions for that specific timestamps from output. After training the model, we test it with the validation data. By the way, we are using sklearn.LinearRegression regressor. The RSME for the linear regression model is 2.53.

$Simple\ Linear\ Regression\ Architecture$:
$(input\ x,\ output\ x\ of\ next\ timestamp) - - >\ Linear\ Regression\ Model - - > Predict$
$(input\ y,\ output\ y\ of\ next\ timestamp) - - >\ Linear\ Regression\ Model - - > Predict$
$Repeat\ for\ 30\ times$

Describe different regularization techniques that you have used such as dropoutand max-pooling in your model

We tried encoder and decoder model with LSTM.First we use velocities and positions in to predict, then we find that it is more effective to use positions only. Also, We make two models to predict x and y separately which improve the loss. We choose ADAM as optimiser which is better than ADAGRAD and SGD. For regularization techniques, we use batch-norm which greatly reduced the loss of prediction.

4 Experiment Design

Problem A [1 points]: Describe how you set up the training and testing design for deep learning. Answer the following questions:

What computational platform/GPU did you use for training/testing?

JupyterHub-UCSD Prefer [1 GPU, 8 CPU, 16G RAM]

One of the many conveniences of PyTorch is that it has built-in CUDA support which can utilize GPU for computations — given that it is detected when using the platform. Deep Learning Models and Neural Networks are known to utilize computations that can be executed in parallel. Having a GPU is advantageous in the number of computations that it can do along with its reputation for parallel computing.

Prefacing later analysis, we have tested an RNN (Recurrent Neural Network) to build our model. RNNs use additional recurrent connections which is akin to having "trainable memory", previous outputs are trained as inputs. Creating these layers is somewhat computationally intensive in training an RNN. Having a GPU can help accelerate the training and prediction processes in our RNN model. The decision on which computational platform slightly predates specific needs of a model we choose. Since we do testing for multiple deep learning models: the nuances and subtleties of each one lends to the idea that having a GPU to facilitate timely processing and testing, especially in case of more intensive computational methods, is more sensible than not utilizing the option.

In all honesty, we wanted a Platform option with a GPU under the idea that it will ultimately run faster. Luckily for us, UCSD was so kind to have an accessible Machine Learning Platform for students with a GPU resource (along with a generous amount of CPU and RAM). DSMLP has allotted class notebooks with nodes that have access to run 1080ti's.

The benefits of having a system with a GPU, multiple CPUs, 16G of RAM, and access for all members, was too good to pass up.

How did you split your training and validation set?

Each batch we created in the training set would be split same quantity of test and train, so forever 10 training agents there would 10 testing agents, we did try to randomize indexes whenever we loaded the data, for the sake of nuance and variation, so all "values" could be used as training or testing.

Our validation set was only seen as "training" values, so processing the data was the same, expect there was no "target" set of values.

What is your optimizer? How did you tune your learning rate, learning rate decay, momentum and other parameters?

Our group had a bias in looking for ways to implement mean squared error and stochastic gradient as our loss function and respective optimizer; as we all had a good understanding of the

concept from lecture. We started looking at, specifically, Stochastic gradient descent for our optimizer due to its popularity for computational efficiency and ease of implementation, which our group agreed with. We also thought it would be advantageous for our model to update more frequently given the type of data. The disadvantages of Stochastic gradient optimizers like convergence and high variance in model parameters were outweighed by the benefits; given proper implementation in reducing/updating learning rate and adapting for relative variance, these issues could be curbed. We also are confident given the backing of the extremely helpful machine learning libraries available.

So, with PyTorch, we are able to have access to optim which contains implementations of various optimization algorithms: we chose the ADAM optimizer (Adaptive Moment Estimation) which configures with our consensus to use Stochastic gradient in our model. ADAM is a first-order gradient optimization that uses stochastic objective functions over our data values that we compose as parameter groups.

We then use MSELOSS from the same library that creates a criterion that measures the mean squared error between given elements.

Parameters for ADAM Optimization Algorithm

betas: coefficients used to calculate the running averages of gradient and its square.
The formulas for the decaying past and past squared gradient shown below:
$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
$v_t = \beta_1 v_{t-1} + (1 - \beta_2) g_t^2$
We chose betas=(0.9, 0.999), which is the default value and generally recommended. The betas dictates which gradients we're averaging over, and we generally agreed that averaging over a larger portion of the gradient would lend to a better prediction model.

Learning Rate = 0.5
The learning rate we currently have is more so inline with performance given the number of epochs we've decided on. The ADAM optimizer computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name deriving (ADA)ptive (M)oment estimation. Under the guise that the optimizer changes the lr accordingly, the hyper-parameter in the Torch implementation of 0.5 gave us suitable time and optimal training.

Learning Rate Decay = 1e-4
The worry we had for the model was maybe issues with overfitting; even though the optimizer should adjust accordingly. The adapting decay rate for ADAM is calculated from the betas through the steps. PyTorch implements "weight decay", a term denoting L_2 regularization, with the default value of 0, which we found lended itself to, both, high accuracy loss and time consumption. Our current value of 1e-4 is more optimal, but is very flexible for change.

One thing to note is that with continued improvement of our model, more than likely, adjustments will have to be made for the optimizer values. One adjustment we are working towards for both efficiency and optimal training is adjusting the learning rate and respective decay through a custom scheduler: one idea is exponential learning rate implementation as we step. The values right now are effective but will continue to improve it to make it more adaptive for our data.

How did you make multistep (30 step) prediction for each target agent?

The inputs are passed into our RNN model using a feed-forward prediction method. The "recurrence" aspect occurs in our model's encoder and decoder process, where the processed outputs are fed back into the model.

How many epoch did you use? What is your batch-size? How long does it take to train your model for one epoch (going through the entire training data set once)?

We are currently using five epochs and have a batch-size of 16. These two values are more reflective of trial and error. We found that the batch size of 16 gave us better accuracy while maintaining the same processing time as larger batch sizes. The five epochs were enough to reach a good fit for our training model. We found more epochs were unnecessary and timely, lending to overfitting.

Explain why you made these design choices. Was it motivated by your past experience? Or was it due to the limitation from your computational platform? You are welcome to use screenshots or provide code snippets to explain your design.

We started designing the model intimidated by the choices and how to go about making an effective deep model for the dataset. We started with ideas and concepts from class and did further research online. We narrowed and started to finalize our model through consensus on which would work best and as something that everybody in the group understands. Much of the specific features and parameters of our model were fitted based on trial and error. Through this and collaboration we were able to gain clearer and clearer ideas of what our final model would be and how it would work. We are still making adjustments and learning how to make a more effective prediction model.

5 Experiment Results

Problem A
Here is a table of our models and corresponding RMSE

Model Name	RMSE	Training Time	Number of Parameters
Linear Regression	~2.5	40 Mins for 20,000 samples (SK learn)	6000 + 6000 = ~12000
MLP Regressor	~3.5	20 Mins for 200,000 samples	~12,000
Feed Forward 1 layer	~8	30 Mins for 200,000 samples	~12,000
Feed Forward multi layer	~6.5	~1 hour for 200,000 samples	~100,000
Simple RNN	~20	~2 hours for 200,000 samples	~135,000
Encoder Decoder RNN	~15	~2.5 hours for 200,000 samples	~240,000
Encoder Decoder using GRU	~12	~3 hours for 200, 000 samples	~250,000

Here we were able to conclude that the simpler models did better since most of the data represented linear trajectories. We see that our more complicated models didn't do as well either because:

(i) Our feature vectors were not represented in a way that our model could extract useful data from lanes, city etc.
(ii) We didn't train for longer than 5 epochs due to computational capabilities

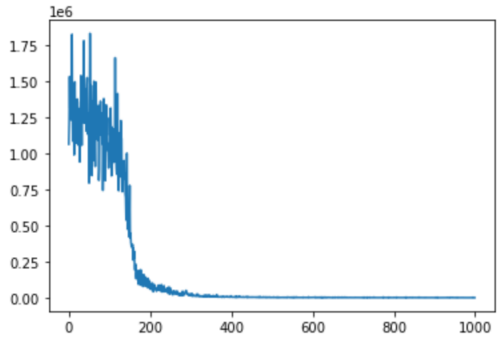
However we see that our RNN models were able to predict trajectories with more nuances and curvatures better than straight trajectories, which can be seen below in our random sampling
Our models took about 1 hour on average to train. They would previously take over 5 hours to train, and this was fixed by normalizing and standardizing the training data, and changing activation function for our FC layer to relu from tanh.
We also notice the more complex our model, the more parameters we see in the model as seen in the table above.

Problem B

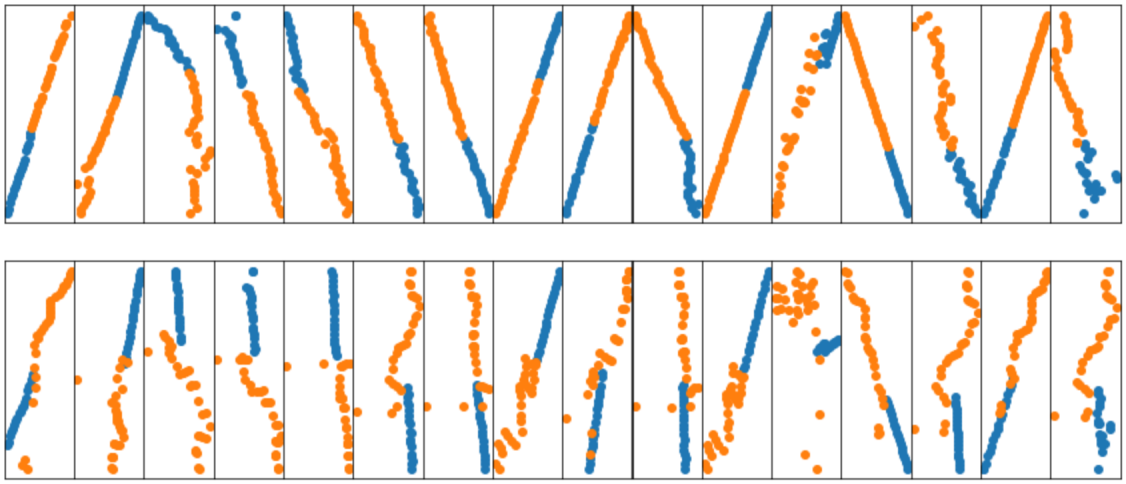
Here is our loss for our encoder decoder LSTM model (Best performing Deep RNN model)

```
4 plt.plot(time, losses[:1000])
```

```
Out[26]: [<matplotlib.lines.Line2D at 0x7fea39ac3bd0>]
```



We randomly sample a few input and visualize our networks output compared to the ground truth model. Here are 16 samples where blue represents the input and orange the output. First row is the Ground Truth and second row is the predictions our model made.



Our current RMSE is 2.526 and ranking is 19

6 Discussion and Future Work

The biggest lesson we learnt so far was how to manage vanishing gradient and normalizing/standardizing our data For the future we want to see how to include other features like lanes, lane_norm better in our models. We also want to experiment with attention mechanism to see if that will help.

For feature engineering, we believe 'rasterizing' the given data to create an image of the scene would best for CNN based modelling. For our RNN models, we loaded batches of 16 of x and y position values and fed these into the RNN models, hoping for the model to recognise sequences.

We did data visualization to see how far our models predictions were from the ground truth prediction. This showed us how a majority of the data followed a linear trajectory, and our RNN was

able to predict the curved examples better, but the linear data worse.
We used grid search for our hyper parameter tuning to get the best results for layer wise regularization, epochs, teacher-force ratio, etc. These two methods combined, as well as normalization, greatly increased our score. We found our model did best with the following:

- Normalizing data: greatly reduces loss
- Higher learning rate: lead to more loss
- Optimiser: ADAM worked better than ADAGRAD and SGD
- Feature Vectors: Using solely p_in values worked better than using both p_in and v_in values. This could be because of the way we designed our feature vectors.
- Model Complexity: The more complex our model, the more loss we get in general. This could be because of vanishing gradient. We are planning to explore solutions for this in the future.
- Regularization: Regularization reduced loss. Batch Norm layer after and before activation increased loss, Dropout slightly increased loss, but layer wise regularization worked the best.
- Activation: RELU worked the best for unnormalized data. For Normalized data, tanh and RELU gave similar results.

If we were to advise a beginner, we would tell them to always start with the simplest model, anaylize how it models the data, and increasing make this model more complex, instead of making assumptions about data and jumping right in. One mistake we made was using the RNN right away instead of looking at linear regression to anaylize how it was modelling. This left us confused when we had high loss scores initially. This was the biggest bottleneck, along with normalization of data.

With more resources, we would like to explore how modeling the scene as in image would help. It was difficult because of the size of the image, but with more resources, we believe this could give us great result.

References

[1] S. Mozaffari, O. Al-Jarrah, M. Dianati, P. Jennings, and A. Mouzakitis, "Deep learning-based vehicle behaviour prediction for autonomous driving applications: a review," 7 2020.

[2] A. Singh, "Prediction in autonomous vehicle - all you need to know," 5 2018.