CSE 151B Project Final Report

Robert Dunn
rdunn@ucsd.edu
github.com/rdunnUCSD/cse151b_project

1 Task Description and Background

1.1 The task and its importance

The goal of this task is to predict three seconds of movement for an autonomous vehicle given a two second observation of the vehicle as well as other objects present in the scene. The importance of this task is a result of change in our society from manually driven vehicles to the autonomous vehicles that are being used in this task. As these vehicles are not being driven with manual input, it is necessary to the safety of the individuals both inside and outside of the vehicle to have accurate movements as to avoid collisions or other potential dangers that we currently face on the road.

As we progress further with technology, this becomes an increasingly important task because it will become necessary to have safe and accurate movement for autonomous vehicles. Realistically, if a good solution is not achieved for this task, then autonomous vehicles will not be able to be integrated within society. However, if this problem is solved, then it is likely that shift from manual to autonomous vehicles could become a reality, which (if done well) could make roads safer than they currently are.

1.2 Methods used in the past

Upon doing research for this task, I found many LSTM based models. For example one team [1] implemented an LSTM as it would serve to makes predictions that consider previous information as well as learn spatial information of the surrounding environment. Another such paper [2] details that in trajectory prediction, "future prediction is usually based on past experience or the history of previous trajectories." Continuing on, it is stated that because of this, LSTMs are likely to perform well for such a task.
As a quick aside, almost all of the research I looked at in terms of trajectory prediction detailed usage of an LSTM. Because of this, I had intended on using an LSTM as well, however I did not have the time to do so.

1.3 Formula for predictive task

We can think about this task as a function of the inputs that output our predictions. For my own model, this means that the inputs are the path and velocities of the target vehicle and the output is the predicted path of the target vehicle. Therefore, we can formulate this as the following:

$$f(x) = b$$

such that x is a tensor representing the x and y coordinates and velocities of the vehicle and b is the tensor representing the predicted path. Furthermore, f represents the transformation applied on x in order to reshape it into b.

Preprint. Under review.

2 Exploratory data analysis

2.1 Initial data analysis

For this task, there are two data sets, a train set and a test set.The train set is built of 205,942 observations while the test set contains 3,200 observations. The difference between the observations in the train and test sets is that the train sets contain both an initial two second observation of the scene as well as the three following seconds, while the test set only contains the initial two second observation. Both the train and test data sets contain the following fields:

- 'city': A three character string detailing the city the observations were taken place.
- 'lane': A (N, 3) array of floats detailing the positions of the lanes in a given scene. In this field, 'N' is equal to the number of lanes which is different depending on the scene.
- 'lane_norm': A (N, 3) array of floats detailing the direction of the lanes in a given scene. In this field, 'N' is equal to the number of lanes which is different depending on the scene. 'N' in this field is always equal to 'N' in the 'lane' field.
- 'scene_idx': A nonzero integer representing the identification number for the scene.
- 'agent_id': A string that represents the identification for the autonomous vehicle.
- 'car_mask': A (60, 1) array of 1's and 0's detailing if an object is being tracked in the nth entry in which a 1 represents an object being tracked and a 0 represents no object being tracked.
- 'track_id': A (60, 30, 1) array containing the identification for all objects being tracked in the scene. For each of the 60 entries, the 30 values all contain the same identification of the given scene.
- 'p_in': A (60, 19, 2) array of floats detailing the x and y coordinates of each object being tracked over nineteen time intervals. Each object is identified such that the index of the object correlates to the same index in the 'track_id' field.
- 'v_in': A (60, 19, 2) array of floats detailing the x and y velocities of each object being tracked over nineteen 0.1 second time intervals. Each object is identified in the same way as 'x_in'.

The following fields are only contained in the train set:

- 'p_out': A (60, 30, 2) array of floats detailing the x and y coordinates of each object being tracked for the following thirty 0.1 second time intervals immediately after the initial observations found in 'p_in'. Each object is identified in the same way as 'x_in'.
- 'v_out': A (60, 30, 2) array of floats detailing the x and y velocities of each object being tracked for the following thirty 0.1 second time intervals immediately after the initial observations found in 'v_in'. Each object is identified in the same way as 'x_in'.

2.2 Statistical data analysis

When looking at the distribution of input positions as shown in Figure 1, we can see that the inputs fall mostly into two different groupings. The first being approximately within the x-range of [0, 1000] and y-range of [1500, 4000]. The second major group is seen to be the diagonal space from the approximate coordinates of (2000, 500) to (4500, 2700). Comparing this to the distribution of output positions as shown in Figure 2, we can see that most of the data is also in the same two major groupings. From this, we can see that it is likely that most vehicles drove along either of the paths that make up these two groupings.

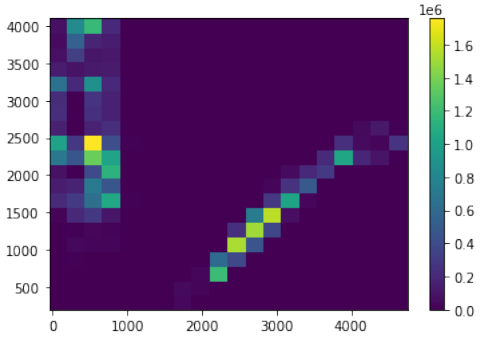

Figure 1: Distribution of input positions

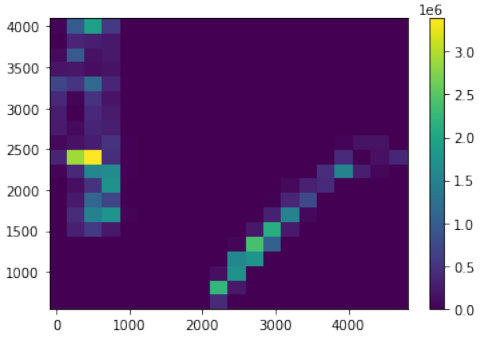

Figure 2: Distribution of output positions

On the other hand, if we look at the distribution of the magnitude of the velocities for all of the agents in the data as shown in Figure 3, we find that most of the data are contained within the x and y ranges of [-20, 20]. Therefore, if we only look at those ranges, two major details are revealed. Firstly, we can see that most of the data lies close to the point (0, 0) which means that the objects either are moving very slowly or are not moving at all. The other important detail is that the rest of the data is distributed in straight lines outwards from (0, 0) which indicates that the objects that are moving are typically moving straight along the x-axis, straight along the y-axis, or approximately at a 45 degree angle.

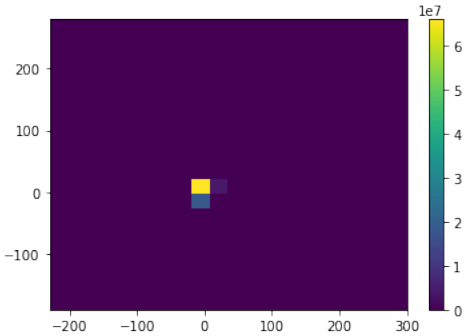

Figure 3: Distribution of magnitude of velocities

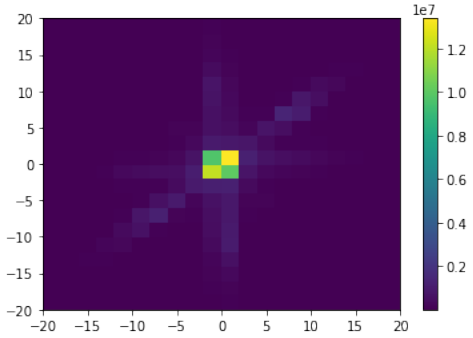

Figure 4: Inner distribution of magnitude of velocities

2.3 Data pre-processing and feature engineering

The only feature engineering that I did for my input data was to normalize the input positions of the target vehicles. Just to reiterate, I only did this for the target vehicle data because I did not use any of the other agents in the scene.
Overall, my normalization process was quite simple. For each of the 19 entries in 'p_in', I subtracted the first entry such that every instances 'p_in' started at the coordinate (0, 0). In doing so, the distance between each observation stayed the same, however their locations had all been shifted relative to their starting positions. An example of this can be seen in Figure 5 which shows that the paths before and after normalization are the same, they just occur in different locations.

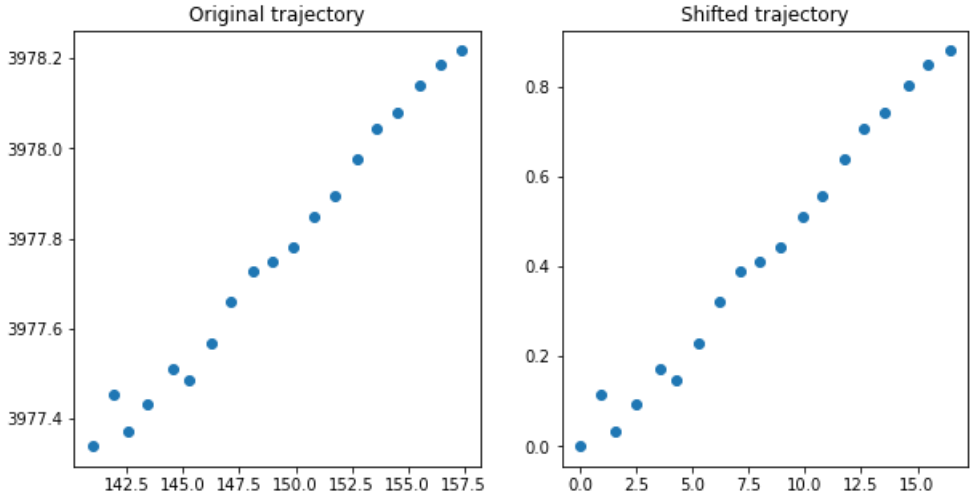

Figure 5: Example path before and after normalization

As for the lane information, I did not utilize this data which did have a negative effect on my model as the model wouldn't be able to predict a lane change assuming the input data is a straight path. In section 5, I will be discussing more details about such predictions.

3 Deep learning model

3.1 Deep learning pipeline

For my model, I ended up using the shifted inputs from the target vehicle as well as the velocities from the target vehicle to create an output of the next 30 expected coordinates. Realistically speaking, the reason I chose to use these inputs is completely based of off simplicity. Going forward with this

task, it would without a doubt be beneficial to add in extra data, although I will talk more about that in Section 6.

I trained my model using the root mean squared error as the loss function, again just to keep things simple. Since the test set would be measured using RMSE, I decided it would be best to then train my model using RMSE.

When designing my model, I had chose not to use regularization techniques such as dropout or pooling because (and I will note my logic may be flawed due to a lack of understanding in this regard) all of the input data was useful in determining the output. Going back to the idea of having more data in my inputs, I do think some of these regularization techniques would become more useful with the increased input size.

For my overall network structure, I decided to use a convolutional neural network. I chose to use this kind of model mostly because it was what I was most comfortable with, and I had an understanding of how to create the desired outputs with the given inputs. Originally, I had plans to start with a CNN and later move to sequence based models (such as LSTM) after creating a decent CNN. However, it took me longer than I had expected to create such a CNN, so in the end that is what I stuck with.

3.2 Previously attempted models

As previously explained, all of my models were some iteration of a CNN. Over time however, my models changed as follows:

1. Original model
    - 35,397 total parameters
    - 3 linear layers
    - 2 convolutional layers
    - 2 normalization layers
2. Second model - adding layers
    - 139,141 total parameters
    - 5 linear layers
    - 3 convolutional layers
    - 3 normalization layers
3. Third model - increase number of parameters
    - 860,485 parameters
    - same layer structure as model 2
    - worse performance than model 2
4. Fourth model - decrease number of parameters
    - 6,233 total parameters
    - same layer structure as model 2
    - significantly worse performance than model 2
5. Fifth model - Add more layers
    - 320,421 total parameters
    - 7 linear layers
    - 5 convolutional layers
    - 5 normalization layers
    - worse performance than model 2
6. Sixth model - model 2 with pre-processing
    - 139,141 total parameters
    - same layers as model 2 + pre-processing
    - significant performance increase

7. Final model
    - 140,165 total parameters
    - very similar to model 6 with minor refinements

While I did not utilize regularization, I did include normalization layers with three batch norm layers and one instance norm layer throughout my network. However, my process for selecting these layers was very antiquated as it was mostly just manual trial and error on small portions of the data set in order to find what combination of normalization layers worked the best. In the end, this ended up being 3 batch norm layers and 1 instance norm layer.

4 Experiment design

4.1 Training and testing design

For training my models, I both trained and tested mainly using my GPU rather than CPU because it greatly accelerated the processes. The specific GPU I am using is the NVIDIA GeForce GTX 1060.

During the training period of my model, I did not split my data into a training and validation set. I do recognize the importance of having a validation set now, although I had not considered this during the actual model building. In a sense, I was utilizing the kaggle submission scores as my validation score. Going forward with this task, it would obviously be beneficial to implement a validation set for my training phase.

The optimizer I ended up using for my final model was RMSprop. I ended up using this optimizer after a small test between five different optimizers which had shown that RMSprop converged the fastest with my model. The results of this test are shown in Figure 6. Initially, I had

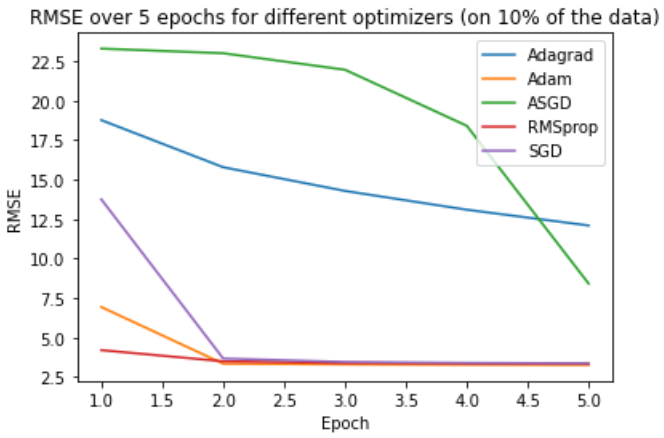

Figure 6: First five epochs on training set with different optimizers

plans to test other parameters as well such as the learning rate, however, I did not have enough time to do so. As a result, I ended up manually trying a few different learning rates and used momentum = 0.9 because that was the recommended value I found in multiple articles from a quick google search.

In order to make multistep predictions for each target, I used multiple convolutional and linear layers to reshape the input data to the desired output shape. Doing so allowed me to use the initial 19 time steps and artificially create the predicted 30 time steps.

For my model, I trained it for seven epochs with a batch size of 32. Each epoch takes approximately 80 minutes to complete. For the batch size, there is not much significance behind using 32, it is realistically only a number that I chose to have a somewhat large batch size. On the other hand, I chose seven epochs because I found that at this point, the model would only make very small improvements per epoch. Therefore, it was not worth the time to continue running the model in order to get such minor improvements in the end.

5 Experiment results

5.1 Comparison of different models

For my previous models, I did not keep the records of the outcomes after creating a better model, so in order to compare my previous models, I have rerun each of the seven models listed above on 5% of the data set for 10 epochs. The results are as follows:

Model	# parameters	Time for 5% epoch	Estimated time for full epoch	Train RMSE
1	35,397	7.2s	4.8 min	88.479
2	139,141	10s	6.7 min	86.335
3	860,485	23s	15.33 min	90.909
4	6,233	6.8s	4.5 min	197.596
5	320,421	13s	8.7 min	87.206
6	139,141	122s	81.3 min	3.444
7	140,165	122s	81.3 min	3.356

One thing to note is that in order to estimate the the full run time per epoch, I used the formula $full = 2 \cdot 20 \cdot partial$ where $full$ is the full run time per epoch and $partial$ is the run time on 5% of the data. The reason I multiplied an extra 2 in this calculation is because when running these models, I believe that the data was saved locally since less space was needed and therefore the access time for this data was much quicker. Since I know that the run time for one epoch in the final model is around 80 minutes, multiplying each estimation by 2 would achieve this time.

In my opinion, the most important thing to take away from this table is the decrease in the train RMSE that occurs from Model 5 to Model 6. As previously mentioned, this is due to the addition of pre-processing the inputs. While the training time is much longer, the increase in performance is definitely worth this trade-off.

Another thing to note in terms of the training time is that I did not have many effective strategies in speeding up my models. In terms of run time, I found that a batch size of around 32 ran the quickest although the speed up from this was not too significant. As I also mentioned earlier, the majority of speed up in my models was due to training on GPU, although that applies to every model I created.

5.2 Final model performance

Again, since I did not think to keep my final result visualizations, so in order to do so, I ran my final model on 10% of the data for 10 epochs and used 1% as a validation set. This looks as follows: As

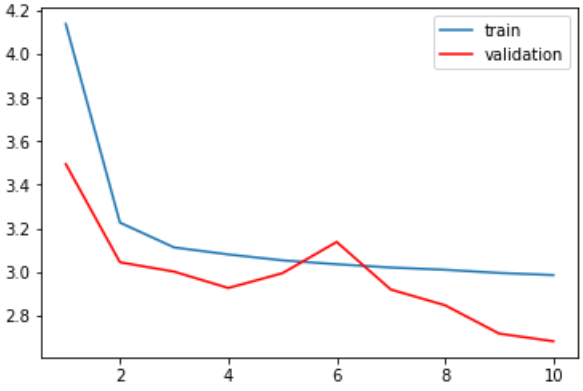

Figure 7: Train and validation RMSE per epoch

for the quality of the predictions, the model is mainly capable of predicting the trajectory for vehicles moving in a straight line. In some cases, the model is able to predict turning vehicles provided that

the vehicle is already turning in the input sequence. When looking at the examples in Figure 8 ('p_in' represented by blue dots, 'p_out' represented by red dots, and predicted 'p_out' represented by orange dots) it can be seen that my model cannot accurately predict turns because the input sequence is moving in a straight line. This makes sense given my model since my model is based only on the input sequence meaning that if the input does not indicate a turn, the model will not assume that the vehicle is turning. This also applies to other events such as lane changes. Regarding the kaggle leader

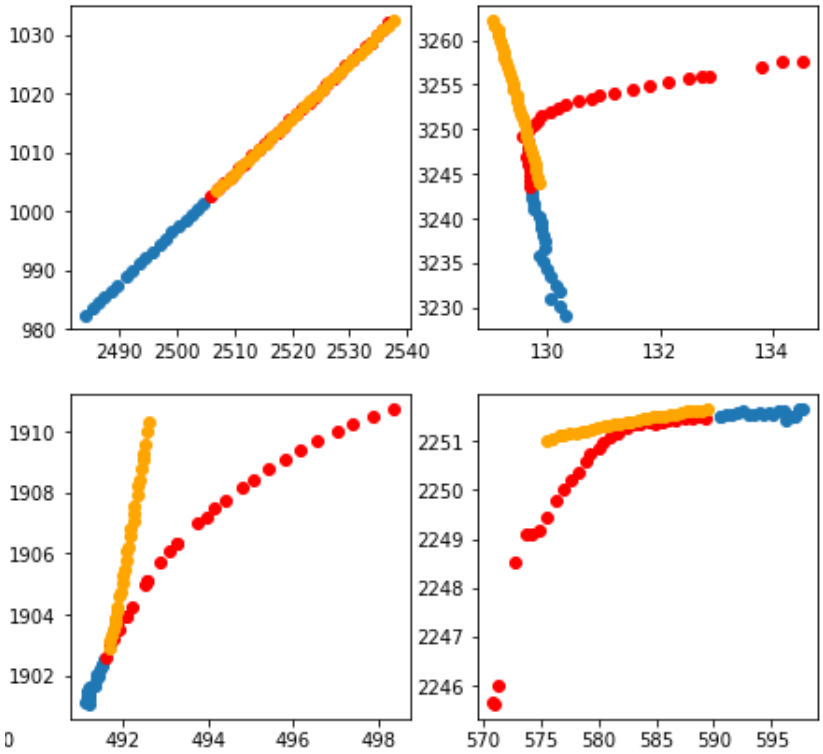

Figure 8: Example predictions on the train set

boards, I am ranked 15th on the public leader boards with a RMSE of 2.342 and 17th on the private leader boards with a RMSE of 2.470.

6 Discussion and future work

6.1 Final conclusions

By far, the most effective strategy in terms of feature engineering is proper data analysis. I say this because this is something that I did not do a good enough job on initially. After finally going back and understanding the data, I realized the importance of normalizing the input positions in order to gain better predictions. Without a doubt, doing so provided the most significant increase in terms of my models performance.

In terms of the bottlenecks over the course of my project, the two biggest were the training times of my model and learning how to implement cuda functionality. The runtime bottleneck is pretty self explanatory, as most of my models would take 30 minutes to an hour long per epoch, so often times I found myself making little progress while the models trained. On the other hand, at the start of the project, I found numerous forum posts which stated that training on GPU is faster than training on CPU. However, I ended up having lots of trouble getting cuda functioning properly and ended up spending a few days trying to do so. Eventually I did get it working and I can say that it was definitely worth it in the end,

The best advice given what I went through with this project is three things.

1. Start early and work often
2. Start with a simple model
3. Train on small sets of the entire data set during the day and train the model on the entire data set overnight

After everything I have tried, one thing I want to attempt is some sort of sequence model, such as an LSTM. While I am content with the overall performance of my model, I think that it would make sense to actually treat the inputs as a sequence. Also, it would be beneficial just to expand out and try new models in general. The other thing I want to implement is adding more data to the input. I think that this is the biggest fault of my model because as mentioned in section 5, my model is not very effective in predicting the trajectories for vehicles that turn or merge lanes, especially after an initial straight path. By adding in extra information about the surroundings of the vehicle, I believe that my model (or potentially a different model) would be able to better predict these kinds of events.

References

[1] Shaobo Wang, Pan Zhao, Biao Yu, Weixin Huang, and Huawei Liang. *Vehicle Trajectory Prediction by Knowledge-Driven LSTM Network in Urban Environments.* https://www.hindawi.com/journals/jat/2020/8894060/

[2] Zhicheng Gu, Zhihao Li, Xuan Di, and Rongye Shi. *An LSTM-Based Autonomous Driving Model Using a Waymo Open Dataset.* https://www.mdpi.com/2076-3417/10/6/2046/htm