CSE 151B Project Final Report

Shmuel Yishai Silver
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92023
ssilver@ucsd.edu

1 Task Description and Background

1.1 Problem A

As the world shifts from using robotics in tightly controlled and predictable spaces to using robotics in highly dynamic and unpredictable spaces, it is becoming increasingly apparent that such systems will be required to make decisions of their own. In the realm of self-driving vehicles, this capacity — or the lack thereof — is quite literally a matter of life and death.

One fundamental component in the decision-making process is the understanding and extrapolation of information within the relevant problem's context. In the space of autonomous vehicles, this entails being aware of an environment's state (both past and present) as well as being able to predict the future states of that environment. This project entails to do just that using deep learning. More specifically, we are tasked with predicting a traffic agent's trajectory (car, cyclist, pedestrian, etc.) within the context of a large, feature-rich scene; we are given scene information over the course of 2 seconds and are asked to predict a specific traffic agent's trajectory over the next 3 seconds.

There are many ways in which solving this task could have a huge impact on daily life. Most obviously, it could help create safer autonomous vehicles and thus usher in a new era of cheap transportation. If this were the case, it may make more economic sense for people to use ride-sharing services as opposed to owning or leasing cars. Likewise, the pace at which home delivery is used might accelerate as more and more people find it convenient and cost-effective to order goods online.

1.2 Problem B

One of the first papers I came across presented DESIRE, a Deep Stochastic IOC RNN Encoder-decoder framework [1]. As far as I can understand it, what it is essentially doing is training an auto-encoder to randomly predict a large number of plausible trajectories based on the given input. These trajectories are then evaluated by an RNN, which assigns each one a probabilistic score. The one with the highest score is then chosen as the network's prediction. They chose to go this route due to issues with a small amount of data that is widely available as well as the tendency for networks to begin averaging their trajectories.

I noticed that one of the authors of the above paper was actually an assistant professor at UCSD. I started looking into his research and found another relevant paper: SMART: Simultaneous Multi-Agent Recurrent Trajectory Prediction [2]. Essentially, they are using convolutional LSTMs to predict future trajectories. They do so by creating input maps using scene information as well as agent information. This is akin to predicting video.

One of the latest papers I found was Spatio-Temporal Look-Ahead Trajectory Prediction using Memory Neuron Network [3]. Essentially, they used a novel type of RNN called a Memory Neuron Network to predict trajectories. And while they did this, the paper was really about the Memory Neuron Network, in which each layer memorizes not just it's own state, but also the states of those

Preprint. Under review.

before it. Indeed, it was only tangentially related to this task, but it was interesting for me to see such a new development.

1.3 Problem C

Predict the trajectory of a vehicle $V_i = (X_i, Y_i)$, where $X_i = (x_i^t, y_i^t)$ for time-steps: $t \in [1, \ldots, t_{obs}]$ and where $Y_i = (x_i^t, y_i^t)$ for time-steps: $t \in [t_{obs} + 1, \ldots, t_{pred}]$ [6]. Additionally, we are provided scene context: C. In our case, $t_{obs=19}$ and $t_{pred} = 50$.

As we can see, when the problem is put in this term, my model can be widely applied to a wide variety of things. For example, it could likely be used to predict stock market prices so long as x_i represents the price at any given moment and y_i represents the number of trades at and given moment. Likewise, I imagine my model could be used to predict air-traffic and naval-traffic trajectory. The context might be different, but the task itself would be relatively unchanged. Lastly, I believe it would be possible for my agent to predict weather conditions: so long as x_i and y_i represent important features such as temperature and humidity, I see no reason as to why this shouldn't be the case. Indeed, I believe my model can have a large range of applications.

2 Exploratory Data Analysis

2.1 Problem A

The data provided contains scenes, whereas the test data (the data that we submit to Kaggle) contains 3600 scenes. Each scene consists of various fields that represent pieces of information within the scene that are relevant to either the input or output.

For input, we have several values [4]:

- "city": This is a string value taking on the values "PIT" or "MIA", representing whether or the scene represents somewhere in Pittsburgh or Miami.
- "scene_idx": This takes on an integer value and is unique among all provided scenes.
- "agent_id": This is a string that dictates the ID belonging to the agent we are trying to track.
- "car_mask": This is a 60x1 matrix that shows which of the 60 possible agents are, in reality agents — as opposed to being empty, dummy values.
- "track_id": This is an array containing the IDs of all the agents within the scene.
- "p_in": This is a 60x19x2 array containing the positions of all 60 (or less) agents within the scene over all 19 time-steps.
- "v_in": This is a 60x19x2 array containing the velocities of all 60 (or less) agents within the scene over all 19 time-steps.
- "lane": This a kx3 matrix containing the x, y, and z coordinates for the k lanes described in the scene
- "lane_norm": This is a kx3 matrix containing the norms of the x, y, and z, coordinates described in the scene.

As for our output (which is provided only for the training data), we have:

- "p_out": This is a 60x30x2 array containing the positions of all 60 (or less) agents within the scene over all 30 time-steps that should be predicted following the initial 19 time-steps.
- "v_out": This is a 60x30x2 array containing the velocities of all 60 (or less) agents within the scene over all 30 time-steps that should be predicted following the initial 19 time-steps.

We can see a sample of such a scene in figure 1.

2.2 Problem B

As we can see in the figures 2.a and 2.c, the input and output positions are largely the same. This makes sense as the positions are global and we would likely be tracking agents on streets and areas for which we know certain attributes (lanes, lane norms, etc.). Likewise, it makes sense that the

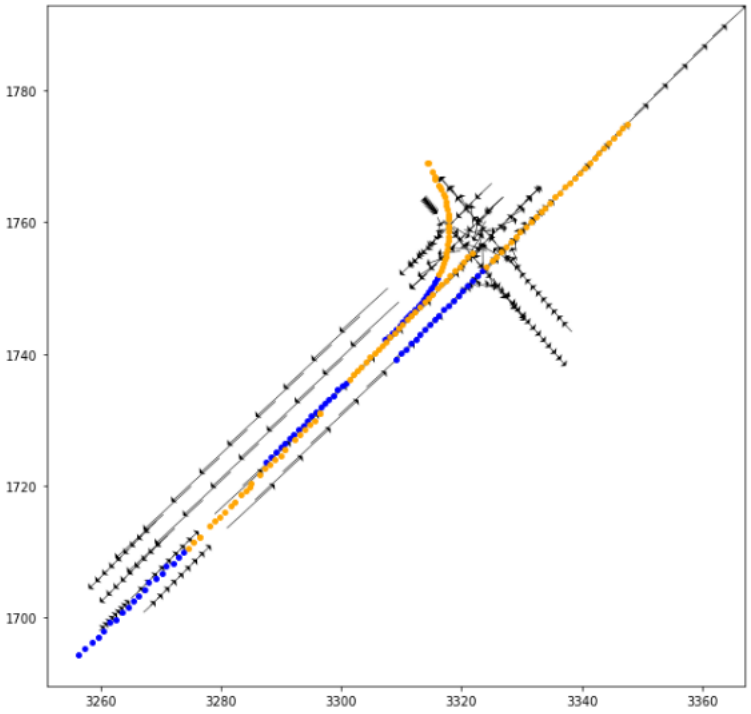


Figure 1: A Sample Scene

distribution is roughly the same as output positions should be close to the input positions that they're associated with.

Additionally, if we plot the positions as opposed to displaying them on a histogram, we can see that the two groupings of positions are, in fact, the two separate cities and their respective roadways, as seen in figure 3.

For the distribution of velocities shown in figures 2.b and 2.d, I think both charts make quite a bit of sense when considering the type of agents there are — cars, cyclists, and pedestrians — as well as the types of agents most relevant to autonomous vehicles (the one's we'd like to track) — cyclists and cars. Likewise, it makes sense that the target agents largely way on the faster side since pedestrians — slower agents — are more often then not on the sidewalk and thus not of immediate interest to our goal of providing safer systems to the realm of autonomous vehicles.

2.3 Problem C

2.3.1 Data Normalization

It's important to understand the task at hand: predicting the behavior of human-driven traffic agents. And, as such, it's important to acknowledge how humans themselves treat this problem. For starters, almost no-one deliberately uses a coordinate space, and especially not a global one like GPS (this focuses on immediate behavior and excludes getting directions). From my experience, it is much more common to use an agent-centered perspective. I.e. where am I now relative to where I just was a second ago (or some other relatively arbitrary unit of time).

With all of this in mind, I chose to convert my agents' positions to inter-time-step displacements. As an example, for agent i, this would look like:

$$\Delta X_i^0 = (0, 0) \tag{1}$$

$$\Delta X_i^t = X_i^t - X_i^{(t-1)} \tag{2}$$

$$\Delta Y_i^0 = Y_i^0 - X_i^{t\_obs} \tag{3}$$

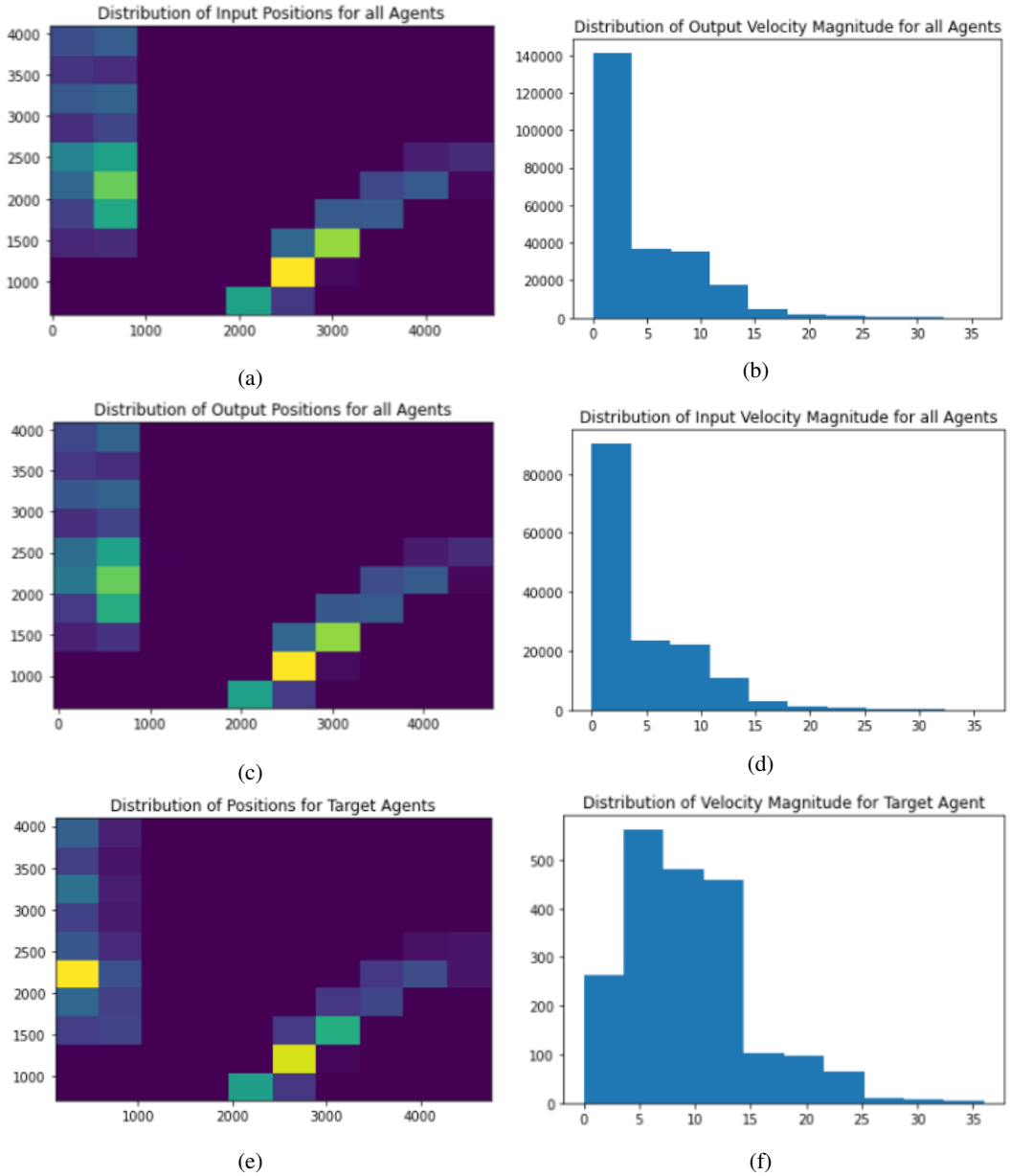


Figure 2: Distributions of Inputs and Outputs and Agent Characteristics

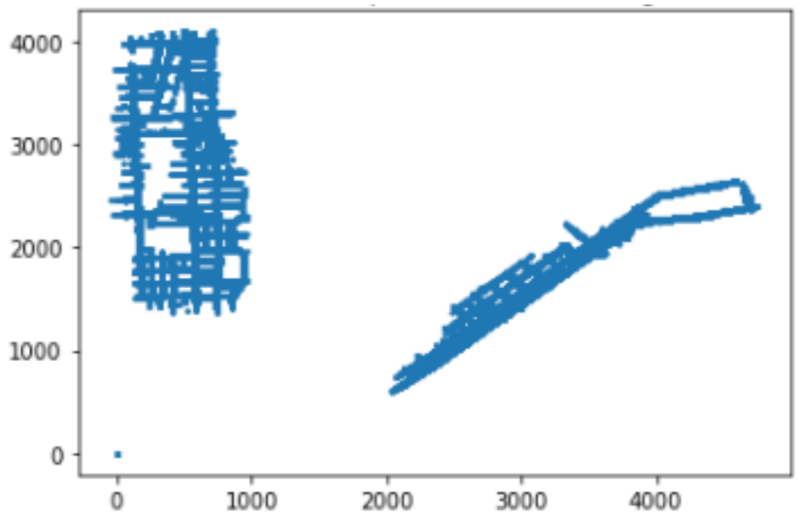


Figure 3: Plotted Positions

$$\Delta Y_i^t = Y_i^t - Y_i^{(t-1)} \tag{4}$$

I found that using this approach to data normalization significantly improved the accuracy of my networks. It should also be mentioned that I found a paper in my preliminary research that discussed predicting human trajectories; this paper also suggested such a method of normalizing the data [5].

2.3.2 Lane Information

I also tried to use lane information. There were many reasons for this, but generally speaking I thought that knowing about characteristics of the path an agent was currently on might lead to better predictions when it came to the agent's trajectory.

The first issue that came to mind was the inconsistent nature of how many lane centers and lane norms were provided and the impact that this might have when it came to how I would provide this information to my neural network. But, an even larger issue soon came to light: When I converted my positions to displacements I eliminated the use of a consistent coordinate system — feeding in lane information might be useless as I'm not necessarily telling my network where the agent is or where it's headed relative to lanes. That being the case, I decided to create an image that would contain both lane information as well as the agents' positions relative to such lanes.

My first attempts were centered around using the information provided by the "lane" and "lane_norm" fields by creating a 2 channel image. The first channel contained the norm at a certain point in the x direction and the second in the Y direction. Ultimately, however, I found that this data was too sparse when it came to properly depicting lanes. I tried addressing this issue by using convolution with a kernel to "connect the dots," but this often resulted in some confusing structures. Take, for example, the lanes depicted in figures 4.a and 4.b, which contains two vertical lanes.

My second attempts used the positional inputs and the velocity inputs provided in the data-set to depict lanes. Essentially, I created a 2x5000x5000 array and then iterated over all positions in the data-set: for each position I converted the coordinates to integers. I then used these integers as indices for the array I created and added the velocities associated with those indices' positions. This approach gave me much better results, as depicted in figures 4.c and 4.d. It should be noted that the scene depicted in these figures is not the same as the scene depicted in figures 4.a and 4.b.

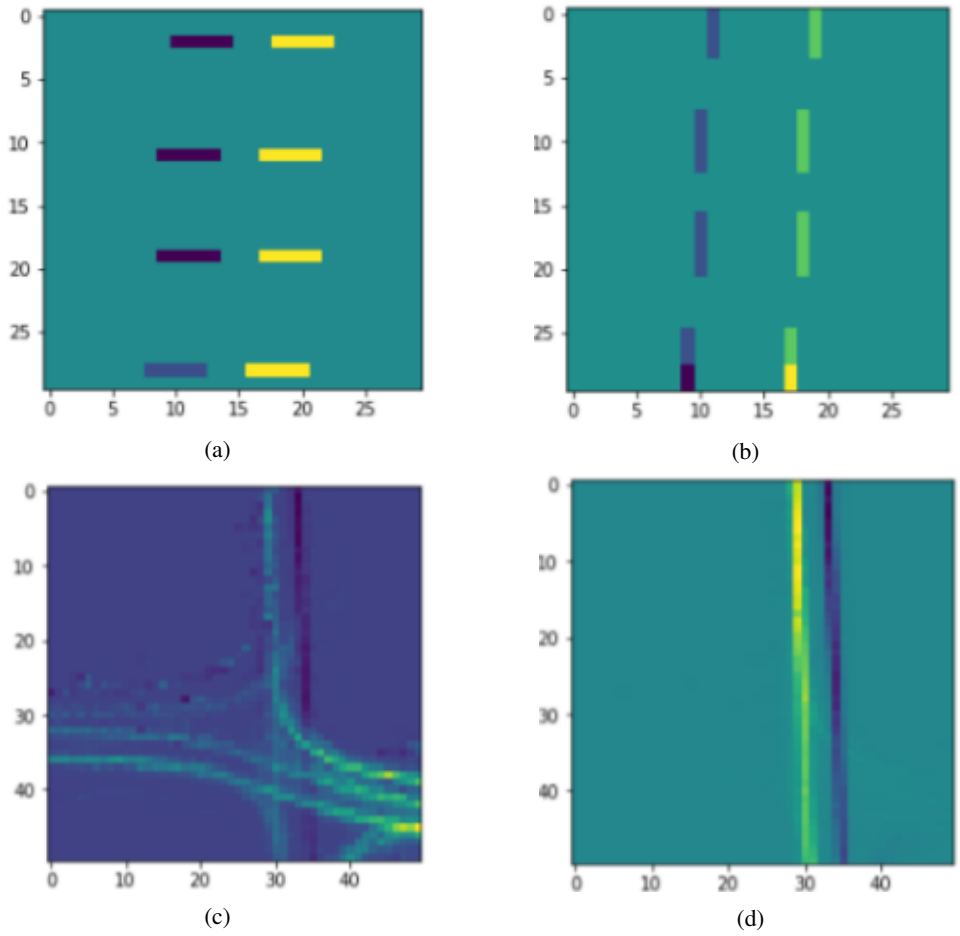


Figure 4: Visualized Lane Images

3 Deep Learning Model

3.1 Problem A

Due to suggestions from other papers as well as the sequential/temporal nature of the task, almost all of my networks were variations of the seq2seq architecture. As such, all of my architectures were fed information as a sequence of 19 time-steps. They would then iteratively predict the next 30 time-steps and append each prediction to a list. This list was then used to form a single tensor with which I found the loss using Mean Square Error (MSE), which is proportional to Root Mean Square Error (RMSE).

To elaborate, I would create a 19 x $(2 \cdot i)$, where i is the number of agents I was tracking, and feed it to my neural network. Within the encoder part of my neural network, the network would iterate over all 19 vectors of size $(2 \cdot i)$ to create an encoding. Then, during the decoder part of my network, the network would iteratively create 30 predictions of size $(2 \cdot i)$ and then use them to create a single tensor of size $(60 \cdot i)$.

There was an exception to this, however, in that the first model I created took a single input vector of size $(19 \cdot 60 \cdot 2)$, or 2280, that contained all input positions and then outputted a vector of size $(30 \cdot 60 \cdot 2)$, or 3600.

3.2 Problem B

I used a variety of architectures. The most important of which are as follows:

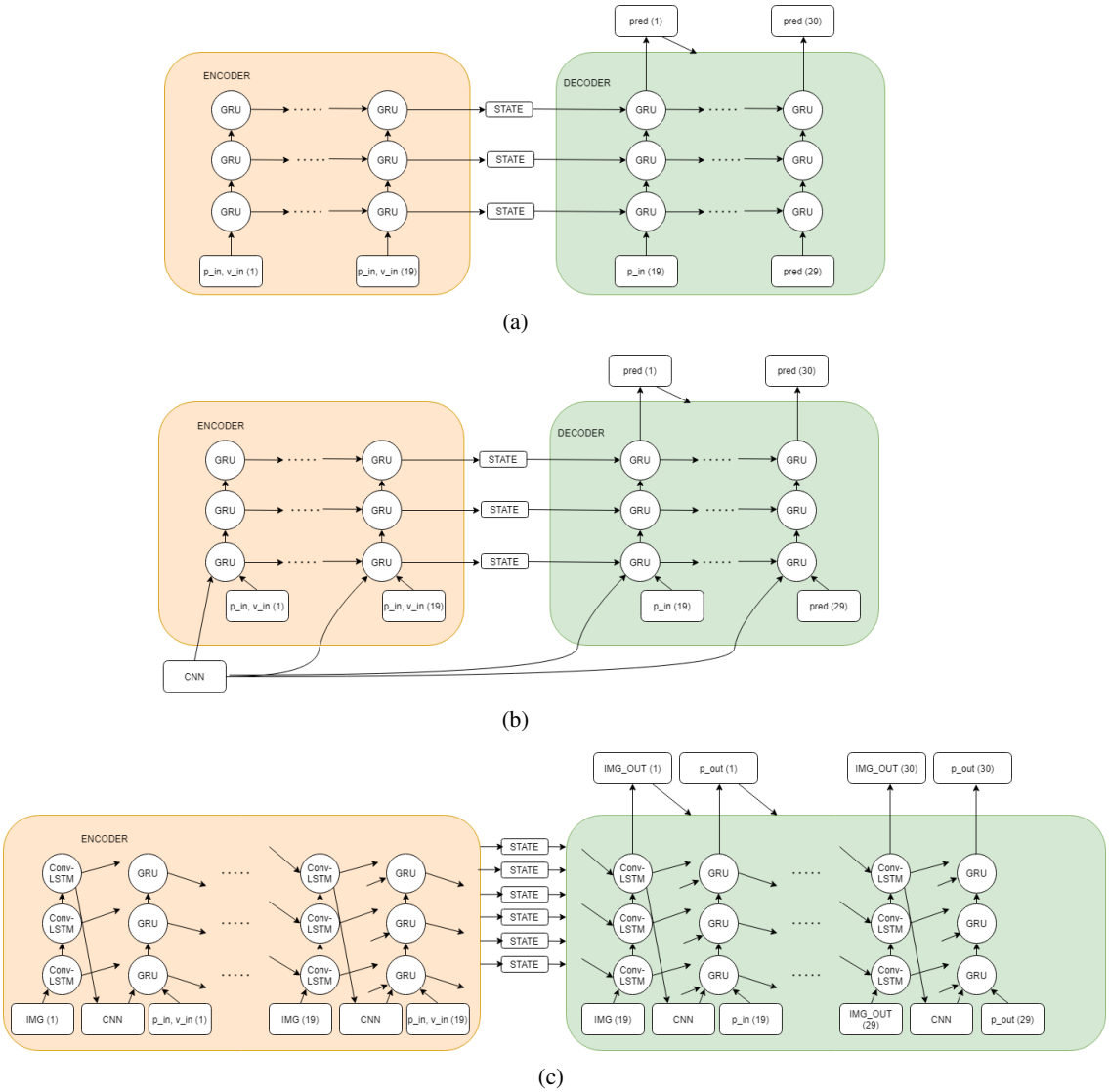


Figure 5: A Seq2Seq (a), a Seq2Seq With Convolution (b), and a Seq2Seq With ConvLSTM (c)

- **Linear Regression**: My first model was a fully connected network of 6 layers, the first three being of sizes 2280, 3500, and 4500, while the last three are 5500, 4500, 4096 (I was originally trying to work in factors of 2 and had accidentally left it as 4096), and then, finally, an output layer of size 3600. It should be mentioned that ReLU was applied only to the first 3 layers as I wanted my network to be able to model more than just linear functions while also being able to output negative numbers.
- **RNN**: This network tracked all 60 agents, thus the input and output sizes of 120. It began with two layers of LSTM cells, the first being of size 120, and the second being of size 256. I feed the outputs of these layers into a small fully connected network of 2 layers. The first layer has 512 dimensions and the second has 1024 dimensions. The output of this fully connected network is then fed into another two layers of LSTM cells, the first having 1024 dimensions and the second having 512 dimensions. Lastly, I feed the output from these LSTM cells to another fully connected network of two layers: the first having 256 dimensions and the second having 120 dimensions. During forwarding, I apply ReLU activation functions to the first two fully connected layers so that the network can model non-linear functions. Likewise, I also apply dropout in the middle to encourage generalization.

- **LSTM Seq2Seq**: This model included three layers of LSTM cells of sizes 256, 512, and 1024, respectively, as an encoder network and then three layers of LSTM cells of the same size as a decoder network. Finally, there an additional three layers of fully connected neurons of sizes 512, 256, 128, and then, finally, an output layer. The sizes of the input and output varied, but were generally between 2 and 6. Some examples would be predicting for only one agent. Another would be predicting for a single agent's position and velocity. Another would be predicting for a single agent's position by taking into accound it's position and velocity (in which case it would have an input of size 4 and an output of size 2). This required splitting the last known time-step into position and velocity and then feeding position alone into the decoder network. You can see a visualization of this architecture in figure 5.a. It should be noted that I also experimented with using Gated Recurrent Units (GRU) instead of LSTMs, as depicted in my figures.
- **LSTM Seq2Seq With Convolution**: This model was an extension of the LSTM Seq2Seq model described above. First and foremost, it included a convolutional neural network that can be described as follows: a convolutional layer that takes in an image of size 50 x 50 that has 4 channels (2 representing lane information and 2 representing the agent's velocity at certain positions) and then outputs 16 channels. It has a kernel of size 5. This output is then passed through ReLU normalization before it is passed through a max-pooling layer of size 2 x 2. This is then repeated 3 times for two more convolutional layers that have 32 and 16 output channels, respectively. This is then passed through a small fully connected network that has three layers of sizes 128, 64, and 64. The vector produced by this network is concatenated to the input fed into the LSTM seq2seq model described above at each time-step. I.e. convolution is only performed once during each call to forward. You can see a visualization of this architecture in figure 5.b.
- **LSTM Seq2Seq With ConvLSTM**: This model was an extension of the LSTM Seq2Seq With Convolution model described above. The idea is simple: rather than use a single image, use 19 input images that have 3 channels each (2 for lane information, which is static, and 1 for agent position). During the encoding stage, we feed these 19 images into 3 ConvLSTM layers of channel size 64, 64, and 3, respectively. All of these have a kernel size of 5. The output of these ConvLSTMs are then fed into the convolutional neural network created for LSTM Seq2Seq With Convolution, which is now run at every time-step. Then, for the decoder part, we use the latest image to generate a new image using the ConvLSTMs and then use that image as input for the convolutional neural network. It should be mentioned that this network has two outputs: one for the predicted positions and one for the predicted images. You can see a visualization of this architecture in figure 5.c.

4 Experiment Design

My training and testing was done on Datahub using the GPU provided, an Nvidia Geforce GTX 1080 Ti. I did this for two reasons: this GPU is more powerful than my own (an Nvidia Geforce GTX 1060 with Max-Q Design) as well as the fact that doing this would allow me to use my computer for other things during training and testing.

For optimization I initially chose the optimizer Adam. This was primarily because, in my experience with Deep Learning, this is a common choice among developers. That being said, it also minimized the number of hyper-parameters that I had to deal with. In fact, the only real hyper-parameter that I tuned in relation to my optimizer was my learning rate. It was originally 0.01, but after discussing it with a tutor as well as testing it for myself, I found that 0.001 provided much better results. Moving on from these learnings, I eventually found an article that explained why AdamW would be a better choice and I adopted that optimizer. Despite AdamW offering a default value of 0.01 for weight_decay, I did experiment with other values.

For the training process itself, I chose to use 10 epochs with a batch size of 100. I did this because I found that my loss was beginning to plateau after about 8 epochs and I was worried about over-fitting to the training data. As for the batch size, I chose 100 because I wanted my networks to efficiently generalize the data while not taking too long to train, and I found that I could accomplish this with 100. Indeed, with an training time of about 9 minutes per epoch (for both of my primary models), I found that that the training time was significantly shorter than smaller batch sizes (10 and 4), while about as long as a larger batch size (1000).

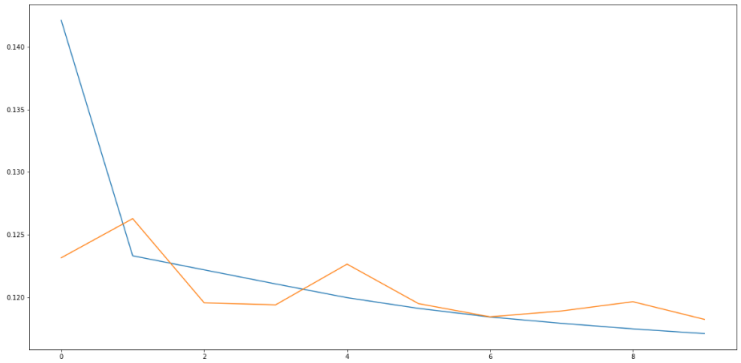


Figure 6: Loss Over 10 Epochs for Training (Blue) and Validation (Orange)

Lastly, I've described how my various networks make their predictions above.

5 Experiment Results

5.1 Problem A

Model Design	Training Time (Mins)	Kaggle Score (RMSE)	Number of Parameters
DNN	60	13.405	106,433,296
RNN	75	4.852	7,573,880
Seq2Seq (LSTM)	70	2.139	17,993,986
Seq2Seq (GRU) w/ CNN	80	2.641	14,331,010

My initial strategy when it came to getting my input and output variables was to assemble them inside my training loop. I.e. I would have my loader give me all of the fields for each batch and then I would use whatever information was relevant to me to create my input and output variables. Eventually, however, I just moved this process to the loader itself. Doing this vastly decreased the amount of time used by each iteration. Likewise, and as I mentioned earlier, increasing the batch size to a reasonable number also sped up this process quite a bit. Thirdly, increasing the number of workers in my data-loader also sped up training (which makes sense, considering how computationally intensive disc operations are).

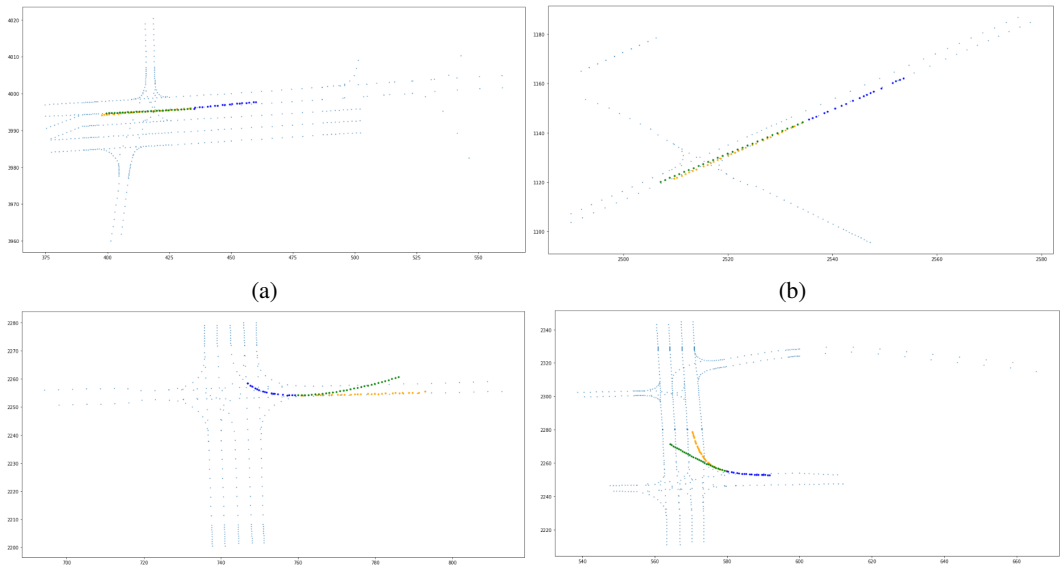
5.2 Problem B

You can see the training and validation loss over 10 epochs in figure 6, and you can see various predictions my model made in figure 7. These figures are in relation to my LSTM seq2seq model.

My current ranking on the leaderboard is 9^{th} place. That being said, my ranking on the leaderboard at the beginning of class on Tuesday of week 10 (when the top 10 were locked into place) was 6^{th}. I am extremely satisfied with this placement, especially in-consideration of my ranking at the time of the milestone report: 22^{nd}.

6 Discussion and Future Work

I think by-far-and-away normalizing my data via displacements yielded the best results. Not doing so resulted in RMSE's that were significantly higher. That being the case, this strategy did have its issues. Most notably, my initial algorithm had a slight bug inside of it. This bug resulted in me calculating my first expected output incorrectly and, ultimately, a significantly higher RMSE. That being said, utilizing data-visualization helped me figure out that this (as opposed to my network) was the issue and address it as necessary. Indeed, data-visualization helped drop my Kaggle RMSE from 22.72 to 2.56.



(a) (b) (c) (d)

Figure 7: Visualized Input (Blue), Ground Truth (Orange), and Predictions (Green)

Another technique that greatly improved my Kaggle score was hyper-parameter tuning. Despite all of my efforts, none of the work I did with map information resulted in a lower RMSE than my work with hyper-parameters, which lowered my Kaggle RMSE from about 2.3 to 2.14.

For me I found that the biggest bottle-neck was Datahub. I had dedicated a full Saturday to working on this project, but Datahub crashed the day before and I was unable to do any work for the entire weekend (week 9). And, due to poor time-management skills, I was unable to reorganize my schedule and allocate different time-slots to this class. I think part of the reason for this was that I genuinely enjoyed this project and I found it relaxing. Without it, I just found other avenues (video games) for relaxation.

If I were to give a deep learning beginner advice for a similar project, I think I'd tell him to do whatever he thought was most fun and to be competitive about the leaderboard. For me, this meant building new architectures. For others, I guess this meant playing around with hyper-parameters. Regardless, I think passion was definitely a major key to success in this project. Secondly, I would definitely advise him to be completely aware of what would be required for the reports and record information as needed. I.e. save numpy files containing the loss for each model trained.

As for what I would do if I had more resources (time, computation, etc.), I think I would definitely continue my work with map data. For one, I think there is a ceiling to how good a simple neural network (i.e. seq2seq) can get with only positional inputs. I was unable to within the time-frame of this project, but I'd definitely like to be able to break that ceiling through the use of map information. Likewise, I would also like to include time-stamp information as part of my input. I was unable to do so until now due to a lack of time, but that is definitely something I'd like to explore.

References

https://github.com/yishaiSilver/Traffic-Agent-Motion-Prediction

[1] Lee, Namhoon, et. al., *DESIRE: Distant Future Prediction in Dynamic Scenes with Interacting Agents* https://arxiv.org/pdf/1704.04394.pdf

[2] Chandraker, Manmohan, et. al., *SMART: Simultaneous Multi-Agent Recurrent Trajectory Prediction* https://www.ecva.net/papers/eccv$_2$020/$papers_E CCV/papers/123720460.pdf$

[3] Rao, Nishanth, Sundaram Suresh, *Spatio-Temporal Look-Ahead Trajectory Prediction using Memory Neural Network* https://arxiv.org/pdf/2102.12070.pdf

[4] Allen, Justin. https://www.kaggle.com/c/cse151b-spring/discussion/233734

[5] Huang, Yingfan, et. al., *STGAT: Modeling Spatial-Temporal Interactions for Human Trajectory Prediction* https://openaccess.thecvf.com/content_ICCV_2019/papers/Huang_STGAT_Modeling_Spatial-Temporal_Interactions_for_Human_Trajectory_Prediction_ICCV_2019_paper.pdf