
CSE 151B Project Final Report

Jonathan Zamora-Anaya
Computer Science and Engineering
University of California San Diego
La Jolla, CA 92093
jzamora@ucsd.edu

Edward Yang
Computer Science and Engineering
University of California San Diego
La Jolla, CA 92093
e7yang@ucsd.edu

1 Task Description and Background

1.1 Problem A

Autonomous Driving is largely considered to be one of the defining pillars of modern Artificial Intelligence in the real world. The prevalence of autonomous vehicle research has spread from centralized AI research companies and institutions to traditional motor-vehicle companies, and this is not by accident. In fact, moving forward, the goal behind all this research is to have safe autonomous vehicles driving on the road so humans are relieved of the task of driving a vehicle themselves. More importantly, though, by having autonomous vehicles that drive better than humans on the road, the likelihood of vehicle-to-vehicle accidents will be significantly reduced. Thus, the motivation to contribute to Autonomous Vehicle research, at its core, is fueled by this goal to make the world safer for everyone.

Argoverse Motion Forecasting

A dataset to train and validate motion forecasting models

Argoverse Motion Forecasting is a curated collection of 324,557 scenarios, each 5 seconds long, for training and validation. Each scenario contains the 2D, bird's-eye-view centroid of each tracked object sampled at 10 Hz.

To create this collection, we sifted through more than 1000 hours of driving data from our fleet of self-driving test vehicles to find the most challenging segments – including segments that show vehicles at intersections, vehicles taking left or right turns, and vehicles changing lanes.

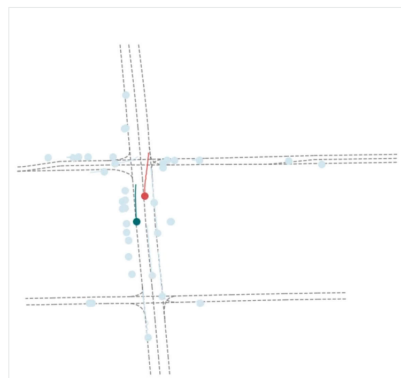
What makes this dataset stand out?

This dataset is far larger than what can currently be mined from publicly available self-driving datasets, and our HD maps make it easier to predict the motion of objects.

SEGMENT DURATION:
5 seconds

TOTAL NUMBER OF SEGMENTS:
323,557

TOTAL TIME:
320 hours



A still frame from one of the motion forecasting sequences, showing the trajectories for the agent of interest (red), self-driving vehicle (green), and all other objects of interest in the scene (light blue).

Figure 1: Figure from <https://www.argoverse.org/data.html>

In order to accomplish this dream, though, there is still much work left to be done in the field of Autonomous Vehicle research. One important piece of the puzzle for the Autonomous Vehicle dream is the task of motion prediction. The task of motion prediction aims to predict where agents in a scene will move at a given time in the future given their current position and behavior. To aid with the research progress being done to solve this problem, *Argo AI* curated and shared a public dataset of highly detailed maps to test, experiment, and teach self-driving vehicles how to understand the world around them [1]. This dataset is popularly known as the *Argoverse Motion Forecasting* dataset, and this dataset serves as the basis for the work we have done over the past quarter as part of CSE 151B - Deep Learning.

Our work, then, aims to develop an efficient and effective deep learning-based solution for the task of Autonomous Vehicle Motion Forecasting through the use of the Argoverse Motion Forecasting dataset. We hope that by sharing our approach and solution to this problem, future work in the field of motion forecasting will be made more intuitive so autonomous vehicles can better learn from scene-level information.

1.2 Problem B

While we are certainly not the first group of researchers or students to work on the task of motion forecasting, our work is deeply inspired by a number of related papers and methods that have previously worked on this task.

To aid with the original Argoverse Motion Forecasting competition, a group of researchers developed a repository of baseline solutions for this task to help give the research community a head start on how to think about the problem at hand. Of the presented methods in this repository, the baseline models included the following approaches:

1. **Constant Velocity** – The model learns from physics-based velocity calculations and does not rely on a deep learning architecture
2. **K-Nearest Neighbors** – The model uses a KNN-based approach to identify nearby agents in a scene and make predictions based off the KNN's provided information
3. **LSTM** – The model uses a Long Short-Term Memory approach to learn long-term dependencies in the provided data and make trajectory predictions based off these predicted dependencies

To aid the baseline approaches further, they computed social and map features for the training, validation, and test datasets they would eventually use in their models. While they mentioned that this feature computation was an expensive computational task, the long-term benefits were well-worth it since their models could better interpret and learn from the provided scene-level information. This concept of computing social features is not new, though. In the "Social GAN: Socially Acceptable Trajectories with Generative Adversarial Networks" paper, the authors presented a novel approach that utilizes Generative Adversarial Networks to better understand human motion behavior. Here, they use an LSTM-based generator to generate trajectories and then have a discriminator determine whether those trajectories are correct.

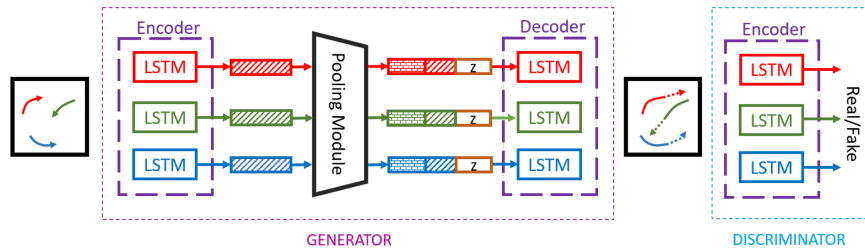


Figure 2: Social GAN Model Architecture from <https://arxiv.org/pdf/1803.10892.pdf>

Through the use of a Social GAN, trajectory prediction models moving forward developed an improved understanding of the scene around them. Further, as a result, the utilization of social

features can be seen being put into practice in the provided baseline models for the Argoverse competition.

With these methods and approaches considered, we developed a firm understanding of the existing methods for solving motion forecasting problems, preparing us to tackle this task ourselves.

1.3 Problem C

For the task of motion prediction, our model inputs will be (p_{in}, v_{in}) , a batch of initial 2-second observations for the positions of each vehicle and their corresponding velocities sampled at a 10Hz rate. Formally, the inputs for this task will be two $(60, 19, 2)$ tensors of floating point values that represent $(px_{(i,j)}, py_{(i,j)})$ coordinates and $(vx_{(i,j)}, vy_{(i,j)})$ velocities (respectively) for $i = 1$ to $i = 60$ where i is one of the 60 tracked vehicle objects in the scene and $j = 1$ to $j = 19$ is the time step at which a position / velocity observation was derived. For the velocities in particular, they can take on values that are positive, negative, or zero that range between $[-200, 200]$. For the positions, they can take on values values between 0 and 4800.

The output of this task will be a tensor with dimensions $(N, 30, 2)$ and it will contain the predicted (x, y) positions of N vehicle objects across a 3-second sequence with 30 time steps.

With the input and output defined for our task, we come to a broader realization about how we can treat our approach as an abstraction to solve related problems. Considering that our model predicts sequences of numbers essentially, we can change the data type of this problem from numbers to words and instead produce word sequence predictions. Such word sequence predictions can take many contexts, such as reproducing the style of an author’s writing by predicting three words in a sentence given two words from the author’s real work. Additionally, our model setup can also be used for the task of translating one word sequence into a different language. With our setup, given a sequence of 2 French words from a French book (hypothetically), we can predict the next 3 words from the French author in English. In fact, the nature of our problem setup is deeply inspired by the Seq2Seq architecture since both our approach and Seq2Seq use an encoder and decoder to produce predictions. An example of this work can be found in the following PyTorch Tutorial that uses Seq2Seq to translate French word sequences into English word sequences.

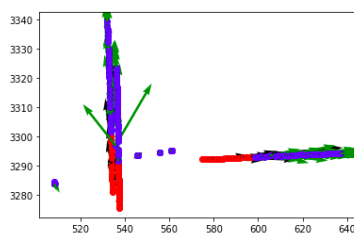
As we can see, the abstractions we developed for our model input and output are reasonable since they are reinforced by existing related works with similar approaches.

2 Exploratory Data Analysis

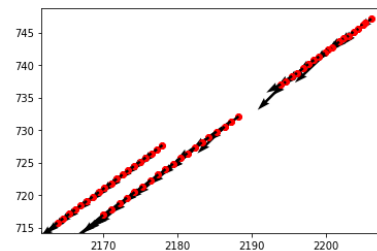
2.1 Problem A

As part of the Motion Forecasting competition for this course, we were generously provided with a Jupyter Notebook that contained some foundational code for us to use during the Exploratory Data Analysis component of our project. Within this provided notebook, we partitioned the provided data into two components – a training set component and a validation set component. Across the 205,942 samples of training data and 3200 samples of test data, each sample had the following attributes:

- **Lane ($k, 3$)** - Stores the (x,y,z) coordinates for the k lanes in the data
- **Lane Norm ($k, 3$)** - Stores the normalized k lanes
- **Car mask ($60, 1$)** - Stores a binary array of whether each of the vehicles (max: 60) is being recorded or not where 1 is recorded and 0 is not
- P_{in} ($60, 19, 2$) - (x, y) position data for up to 60 vehicles for 1.9 seconds [what we are provided as input]
- P_{out} ($60, 30, 2$) - (x, y) position data for up to 60 vehicles for 3 seconds [what we want to predict]
- v_{in} ($60, 19, 2$) - (x, y) velocity data for up to 60 vehicles for 1.9 seconds [what we are provided as input]
- v_{out} ($60, 30, 2$) - (x, y) velocity data for up to 60 vehicles for 3 seconds [what we want to predict]
- $scene_{idx}$ - id of the scene (same as .pkl name)
- $agent_{id}$ - id of the agent in the scene
- $track_{id}$ - id of all vehicles in the scene
- **City** - city where data was collected



(a) Sample Training Data Scene



(b) Sample Validation Data Scene

Figure 3: Example Scenes from Argoverse Data (arrows are velocity vectors, points are positions)

In addition to exploring the provided data's attributes, we also created Plot (a) and Plot (b) in Figure [3] to demonstrate what an example training scene and validation scene look like with all the recorded vehicles in an individual scene.

2.2 Problem B

With an understanding of the data attributes and some sample scene visualizations, we proceed by conducting a statistical analysis on our data to further understand the properties of our data.

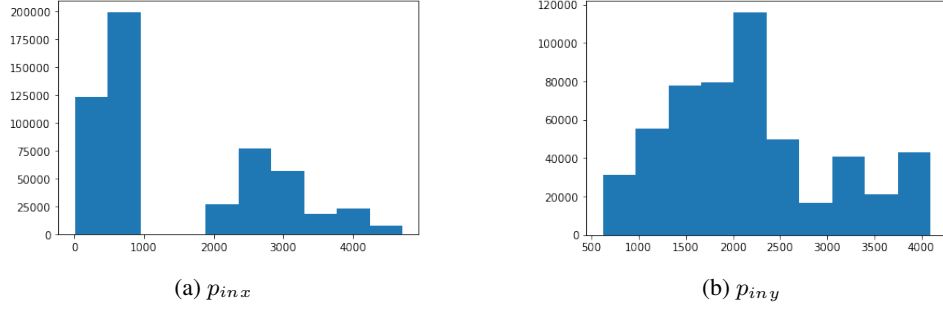


Figure 4: Distributions for Input Positions

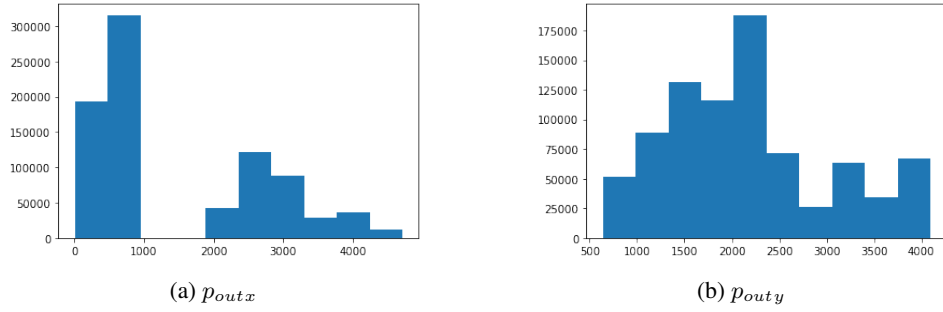


Figure 5: Distributions for Output Positions

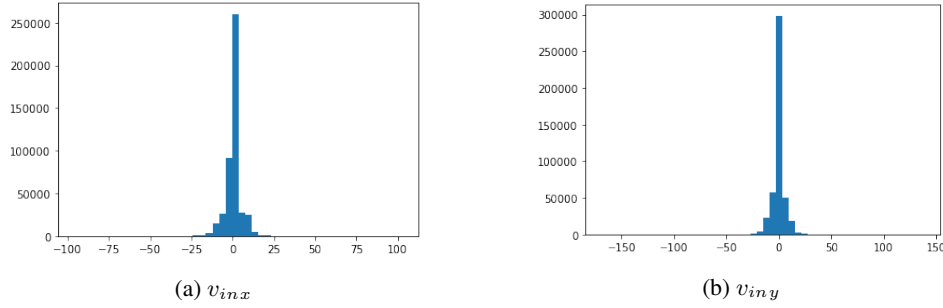


Figure 6: Distributions for Input Velocities

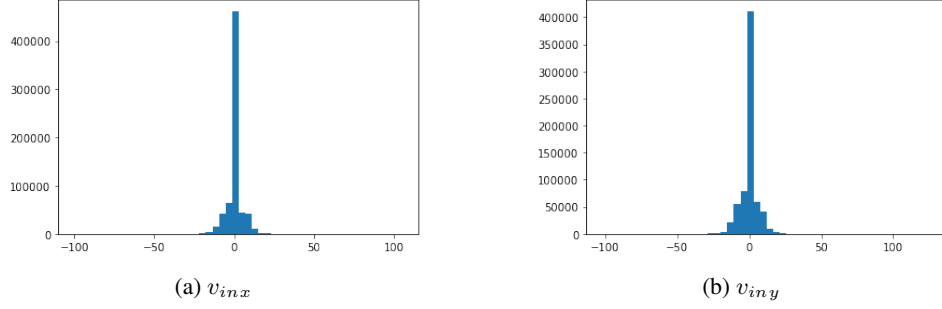


Figure 7: Distributions for Output Velocities



Figure 8: Distribution for Sample Target Agent's Input and Output Positions

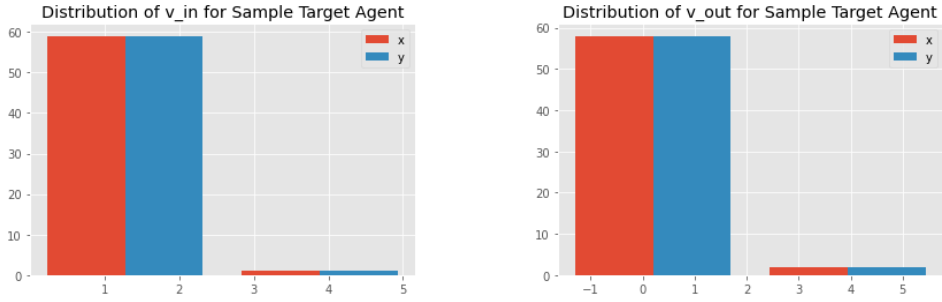


Figure 9: Distribution for Sample Target Agent's Input and Output Velocities

In particular, we examined the distributions of input positions and velocities, output positions and velocities, and target agent positions and velocities. Upon inspecting the distributions for the data, while the data presented here is shown in 2D, we are working with 3D data in our model. As a result of working with these layers of data in 3 dimensions, we had to ensure in the next step for feature engineering we took these considerations into account.

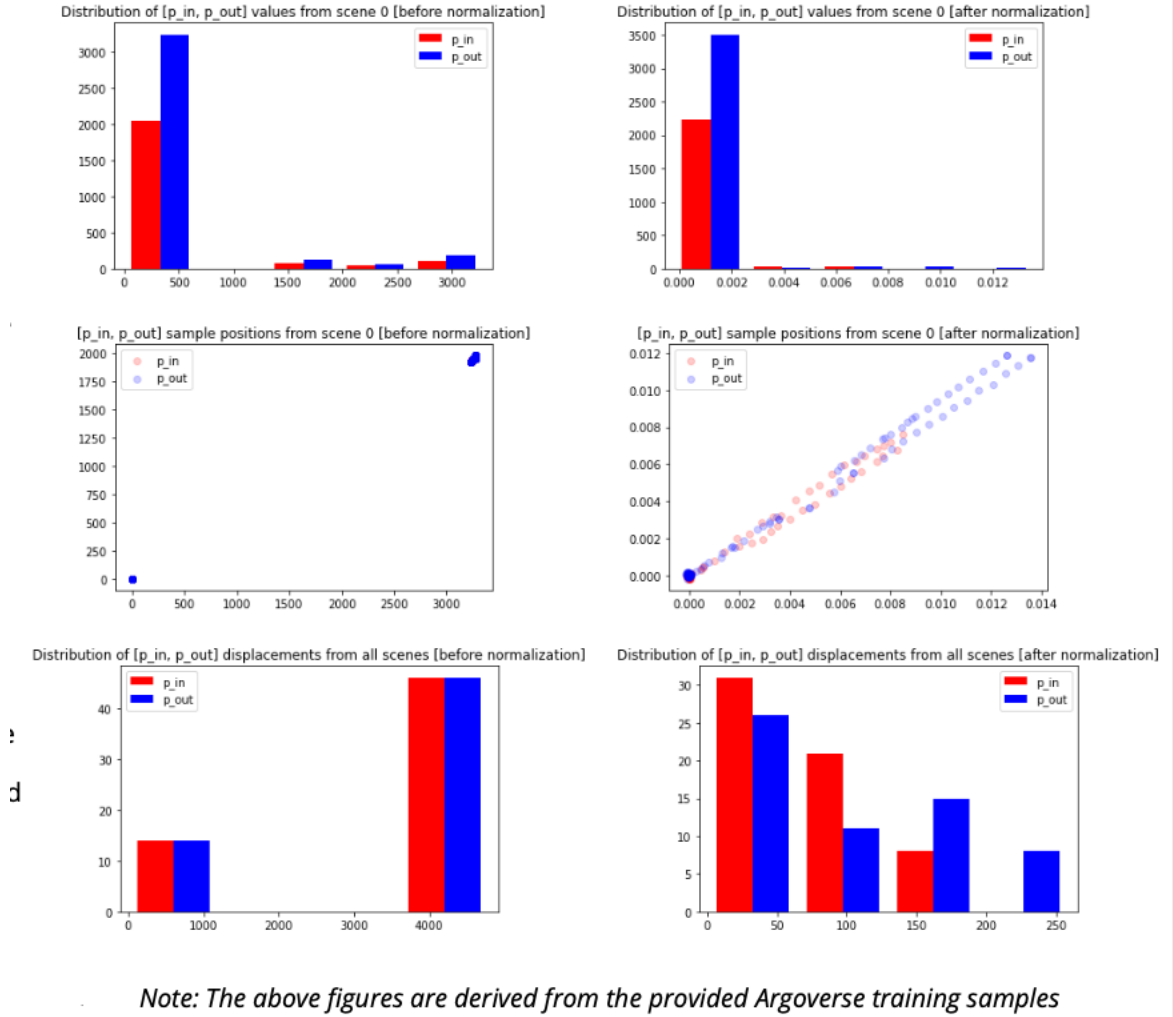


Figure 10: Before and After Normalization Distributions

2.3 Problem C

Before building a great motion prediction model, we must first ensure that our data is engineered such that our model can learn meaningful predictions from it. To do this, we performed feature engineering on the position and velocity data we were provided. We preprocessed our data and normalized it such that all paths start at the origin and then we divided the paths by 200 which we found to be the largest absolute displacement so that we could get the values between $[-1, 1]$. Through our analysis, we found that fastest absolute velocity is 85 so we divided our velocity input by 85 to get the values between $[-1, 1]$. We chose this normalization scheme and designed our features in this way so that we could increase the magnitude of the difference between points so that it would be more apparent to the network. By doing so, the network can more easily learn from the data as there is most change in the activation functions between these bounds. Not only that, normalization standardizes the data so that all data would be evaluated equally without some data having more weight just because the positions or velocities are higher.

We did not use lane information provided in the dataset.

3 Deep Learning Model

<https://github.com/jonzamora/MPogNet>

Figure - Link to our 151B Project Repository

3.1 Problem A

Preliminary tests showed that if we used our network to predict velocity in addition to position we were able to get better performance which is shown in the graphs in the following section. This intuitively makes sense because motion prediction is a problem involving not just position but also velocity and acceleration. If we only used position loss, the network would need to learn "velocity" and "acceleration" implicitly. This is difficult since acceleration is the second derivative of position and velocity is the first derivative. However, by using velocity as a loss also, the network is able to learn the task of motion prediction much easier.

We used RMSE loss of predicted velocity and position against the scaled ground truth velocity out and position out. Using RMSE loss of predicted position/ velocity against ground truth seems to be the most common metric for evaluation on this task after we had surveyed several papers. Note that, in our graphs, we only display position loss as that is what we want to predict in the competition.

We chose our final model out of all the experimental models depending on which one had best validation performance with all hyperparameters held constant. To reiterate, the input to all our models was p_{in} and v_{in} and the output is p_{out} and v_{out} since our preliminary research showed that using velocity was important regardless of the model architecture. The motivation behind this is the fact that, although different networks have different numbers of parameters, we observed that the training loss decreases by smaller amounts as the epoch increases to where the decrease is not significant. By taking the best validation score after a large number of epochs, we can assume that the networks had sufficiently learned and any difference in performance is attributed to the network's ability to predict and not because of the amount of time it was trained.

3.2 Problem B

Because we observed no overfitting in our networks since the validation/ train loss were still dropping, we chose not to add in regularization methods such as dropout or maxpooling. In fact, in some of the papers we surveyed, they reported that batch normalization in LSTM models actually hurts performance. The provided itemized list of model architectures details the progress of our network where we used previous architectures as inspiration for subsequent ones.

- Naive Encoder Decoder Architecture
 - Encoder LSTM
 - * Takes in p_{in} and v_{in} information
 - * Learns the vehicle's "intent" ie. the planned trajectory of the vehicle
 - Decoder LSTM
 - * Takes in the hidden states from encoder which store "intent"
 - * Outputs predicted p_{out} and predicted v_{out}
 - Since cars have acceleration (second derivative of position) using only position to predict position is difficult. By using velocity also, the LSTM has an easier time learning acceleration (first derivative of velocity)
 - Total parameters: 232644
- Bigger Encoder Decoder Architecture
 - Motivated by our previous success we expanded our network to use "LSTM blocks"
 - * 2 LSTM layers with the first LSTM layer expanding states while second layer maintains states
 - * Splits the tasks of encoding/ decoding across multiple LSTMs so each layer doesn't need to do multiple tasks at once

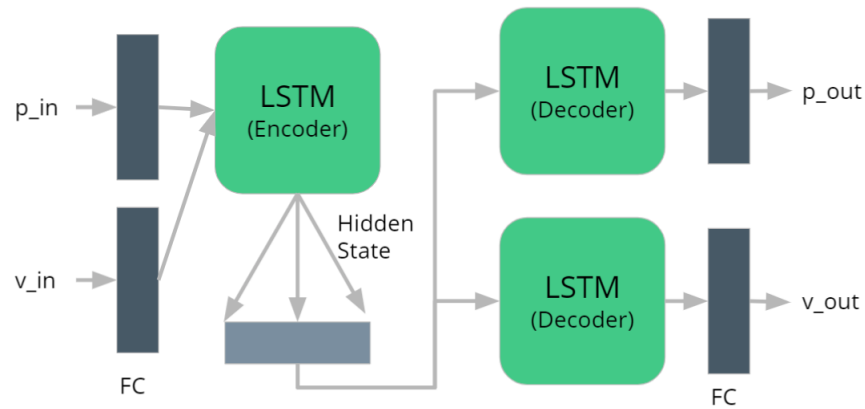


Figure 11: Naive Model

- * Inspired by convolutional blocks in classic CNN architectures (2 CNN layers followed by FC layer)
- Allowed network to be bigger while maintaining reasonable feature expansion
- Total parameters: 2141124

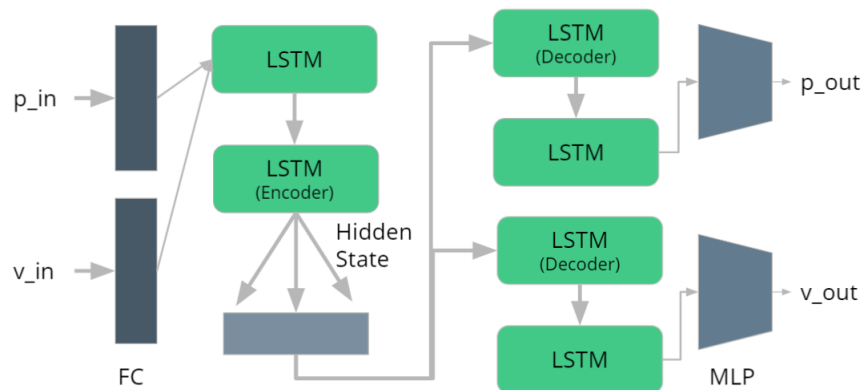


Figure 12: Bigger Model

- Big Big Encoder Decoder Architecture
 - Theorized that an even bigger network with an FC layer between encoder and decoder would have better performance
 - Total parameters: 7884740
- Simple Zero Fill Architecture
 - Instead of directly passing in “intent” to decoder, we load it as the states of the decoder
 - Pass in input to encoder to decoder also as initial trajectory. Decoder would have stronger sense of trajectory based on “intent” and positions and could learn trajectory agnostic “intent.”
 - Total parameters: 397188
- Bigger Zero Fill Architecture
 - Bigger version of Simple Zero Fill that has the same neurons per layer distribution as Bigger Encoder Decoder
 - Total parameters: 5425284
- Recursive Model

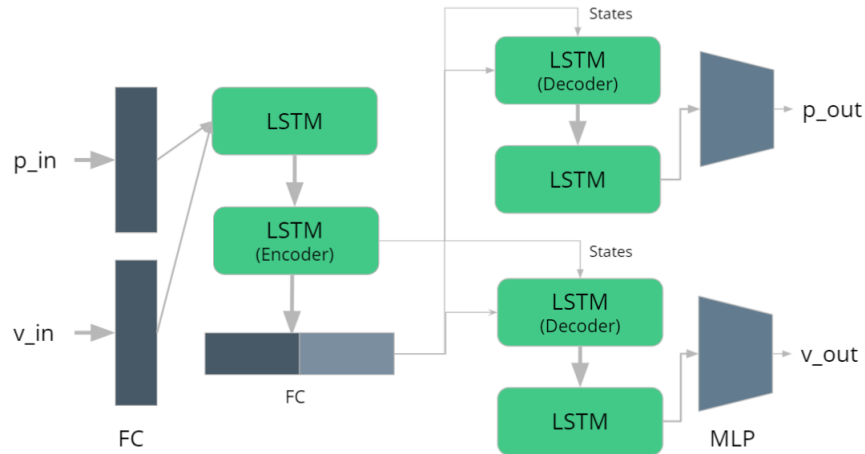


Figure 13: Zero Fill Bigger

- Utilizes a recursive decoder state + input feature inspired by Seq2Seq
- Instead of predicting all 30 points at once, we predict one at a time, recursively. As a result, the network doesn't need to learn as much and just needs to learn next point instead of next 30 which is an easier task
- Cascading errors if the first point is noisy, the noise will be magnified by subsequent points
- Total parameters: 100484

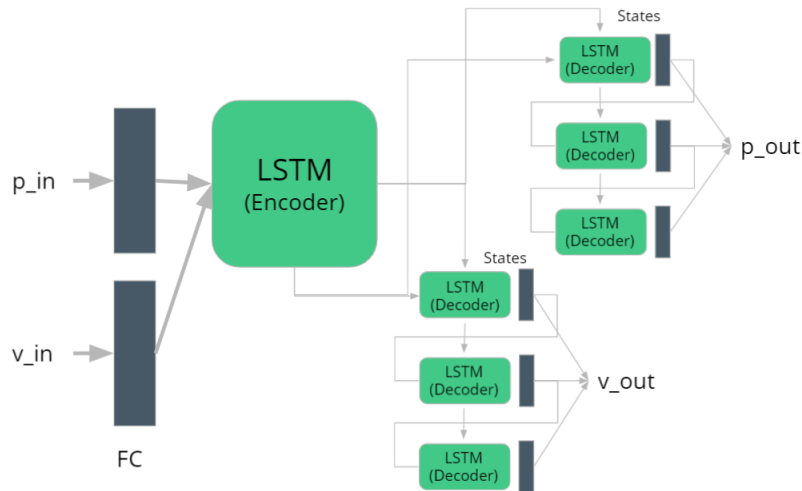


Figure 14: Recursive Model

4 Experiment Design

4.1 Problem A

Most of our network design choices were motivated by our past experiences and from surveying literature in the motion prediction field. The choices of optimizer and learning rate were mainly motivated by our previous experience training models for other tasks. The choice of multistep prediction was motivated by what we learned in class about LSTMs and how they are able to "remember" data over time.

For the training and testing of our deep learning model, we used two computational platforms and GPU types. First, we trained and tested our model using PyTorch with CUDA acceleration on an NVIDIA RTX 3080 Max-Q Mobile GPU with 16 GB of VRAM. This training and testing process was done on a personal computer with a personally-owned GPU, and the model was developed within a Conda environment that satisfied all the required dependencies for our model. Second, we also used the Google Colab Pro platform to train and test our deep learning model. Through Colab Pro, we have access to a Tesla V100 GPU with 16 GB of VRAM.

We used a standard 80:20 train/ validation split to verify that our network wasn't overfitting on the input data and then trained on the whole dataset for submission prediction. We decided to go with the Adam optimizer as it seemed that many other models were using it for this task. We started with a learning rate of $1e-4$ which also seems to be the standard for this task. By using the Adam optimizer which updates each parameter with an individual learning rate, we did not do any manual learning rate decay or momentum since we observed that the training/ validation losses didn't converge even after running the program for more than 5 hours.

In order to make multistep prediction of each target agent, we leveraged LSTMs in our model since LSTMs are able to predict data points over time. Effectively, we would input an $(19, N, 2)$ tensor of position in data and feed it to our network which outputs a $(N, 30, 2)$ tensor which will be our output position prediction. To generate the submissions, we input a $(19, 60, 2)$ tensor corresponding to all the position data in a scene and output a $(60, 30, 2)$ tensor and access the index of the target agent.

We trained for 30 epochs with batch size 100 since we found that the performance increase vs. training time tradeoff was optimal with those parameters. Further, we chose such large numbers since the computational platforms that we used are able to handle large batch sizes. On top of that, by using larger batch sizes, the network will learn faster since the time bottleneck seems to be loading in the data. Larger batch sizes will also help the network generalize better since it won't finetune towards any particular sample. It takes us approximately 7 minutes to train a single epoch.

5 Experiment Results

5.1 Problem A

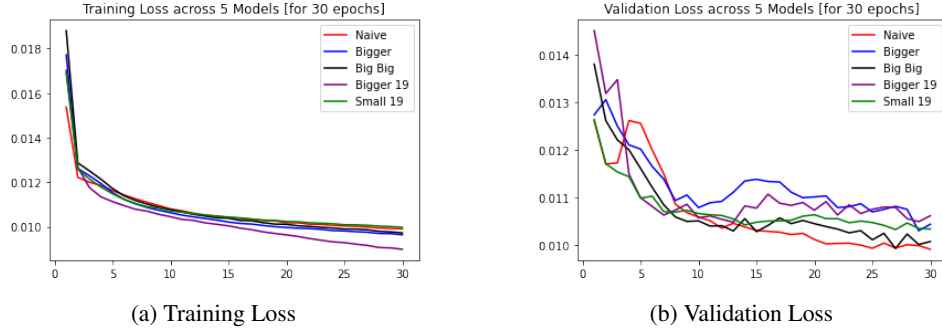


Figure 15: Plot of training/ validation loss across 30 epochs

All data taken at 30 epochs with batch size 100 on same 80:20 train/ validation split using Adam optimizer and a learning rate of $1e-4$

	Naive	Bigger	Biggest	19 Smaller	19 Bigger	Recursive
Train Loss	$9.91 * 10^{-3}$	$9.64 * 10^{-3}$	$9.73 * 10^{-3}$	$9.98 * 10^{-3}$	$8.989 * 10^{-3}$	$1.02 * 10^{-2}$
Validation Loss	$9.91 * 10^{-3}$	$1.04 * 10^{-2}$	$1.01 * 10^{-2}$	$1.03 * 10^{-2}$	$1.06 * 10^{-2}$	$1.03 * 10^{-2}$
Training Time	310s	310s	310s	310s	310s	480s
Parameters	232644	2141124	7884740	397188	5425284	100484

Through this table we see that the most promising network is Bigger Zero Fill which has the lowest training loss despite the fact that it has the second largest number of parameters. The validation loss across all the networks seems to be similar so we based off of training loss. The graphs and figures below are of that model.

Through multiple tests we found that a batch size of 100 had the best performance increase vs. training time tradeoff using 2 workers which was the limit for our devices.

5.2 Problem B

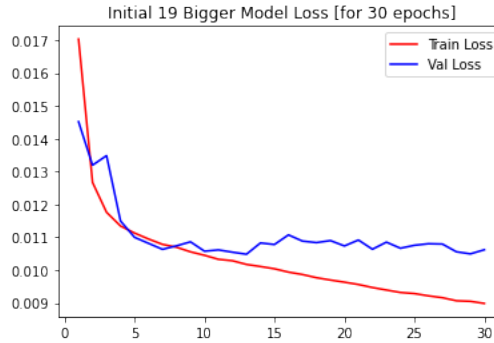


Figure 16: Bigger 19 Model

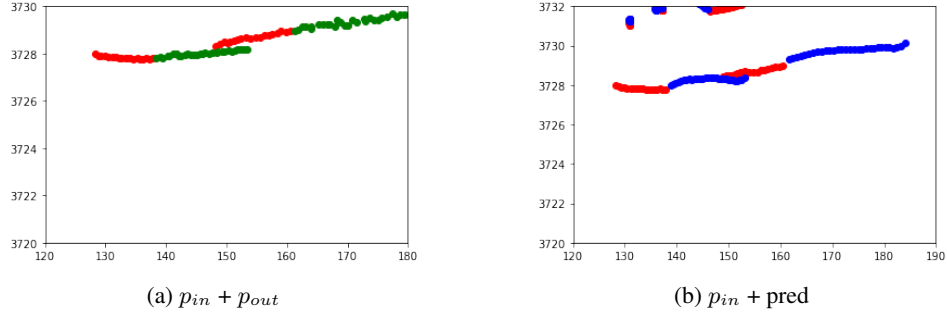


Figure 17: 100.pkl: Ground Truth and Prediction Samples after Training from Training [Red: Prediction, Green = Ground Truth]

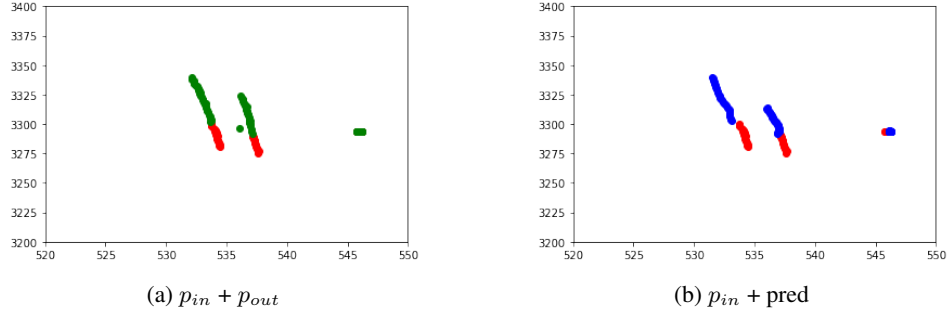


Figure 18: 1000.pkl: Ground Truth and Prediction Samples after Training from Training [Red: Prediction, Green = Ground Truth]

We found that our best performing design to be bigger zero fill which we used in our final submission.
























7	▼ 1	Higher Brothers	   	2.15846	54	4d
8	▼ 1	PikaPika	   	2.19582	35	6d
9	—	Me, Myself, and I	 	2.24610	29	4d
10	—	SuperCats.punch()	  	2.30110	12	2d
11	▲ 7	MPogNet	 	2.30705	19	2d
12	▼ 1	Sarah		2.35131	71	3d
13	▼ 1	CtoZ	  	2.38143	90	4d
14	▼ 1	Alvengers	   	2.42321	33	2d

Figure 19: Final ranking on leaderboard

6 Discussion and Future Work

6.1 Problem A

We think that the most effective feature engineering strategy is data normalization where we normalize the data between $[0,1]$ or $[-1,1]$ and start all the paths at the origin. By doing so, the network can more easily learn from the data as there is most change in the activation functions between these bounds. Not only that, normalization standardizes the data so that all data would be evaluated equally without some data having more weight just because the positions or velocities are higher.

Data visualization helped improve our score the most as we were able to see unique aspects of the input data that we could leverage such as the fact that the largest absolute displacement from the starting point of any path is 200 and the fastest velocity is 85. Using these facts and others we were able to normalize the input more efficiently since we saw that our networks had more or less the same performance regardless of hyperparameter tuning.

The biggest bottleneck in our project was data normalization as increasing our position in the top 20 was difficult since they were off by less than .1. By using effective data normalization we were able to get our network to learn paths better (within 1 unit of the ground truth path) by increasing the magnitude of the difference between points so that it would be more apparent to the network which allowed us to get to position 11.

I would advise a deep learning beginner to try out simple models before more complicated models as simple models are able to get pretty good performance already and complicated models are used to get top tier performance. By using a simple model, they can ensure that their training pipeline works before diving into a more complicated model where there can potentially be bugs in the model also, making it harder to debug and fix.

If we had more resources, we would like to explore the idea of social pooling which we have seen in some of the papers we've surveyed. Since there are multiple cars in a scene, we could potentially use the trajectories of other cars as a "prior" for the trajectory of the target car. Essentially, the network would learn scene- level information such as the direction and lengths of the roads and the paths of other cars to better predict the path of the target car. We also want to try other approaches such as using a GAN or reinforcement learning to solve the task. By using a GAN, we would generate multiple potential trajectories for the target car and the discriminator tries to guess which of these trajectories seems to be the most "reasonable." This may have increased performance since the network learns trajectory and "intent" more strongly as it also will see fake paths along with the real paths, reducing the affect of noise in the data. We also want to try formatting the task as a reinforcement learning task where we simulate an agent and the reward would be how well the agent travels along the actual path given the input path.