CSE151B Project Final Report

Eric Cherny | Lakshmikethan Bethamcharla
echerny@ucsd.edu | lbethamcharla@ucsd.edu
https://github.com/ericcherny/CSE151B-Final

1 Task Description and Background

1.1

Our input consists of an autonomous vehicle's 2D plane position as it navigates in various contexts. This project would be applicable to predicting how AVs navigate across various contexts and show if there are some underlying patterns in the motion of an autonomous vehicle, without direct knowledge of any obstacles.

Solving this kind of task may be important in researching autonomous vehicles and how they navigate when having lost access to vital resources of an AV, such as lack of GPS/signal (so we could simply use the direction traveled from the last point of signal to identify roads from previous travel data of other cars); lack of data (so if a new road is constructed we could perform an update on an AV without having access to a satellite by using the travel data of other cars).

The lack of data problem also plays into how can we automate certain tasks that have been required to be done manually for years, such as instead of drawing new roads manually by identifying them over satellite, maps can now be automatically redrawn through careful analysis of how cars travel, thereby showing us roads, lanes, exits, turns, etc.

1.2

Initially we interpreted this challenge as something related to a time series challenge because we needed to predict a 6 second path given a 5 second path. In class we have seen how RNNs and enccoder/decoder architecture can help predict a x number of time steps given an input. Therefore after making our simple Sequential model of one layer we wanted to explore the lstm idea. Apart from class we also looked at examples of what models would best fist time series and sequence prediction related questions. We found two articles that helped us learn more about the autonomous challenge and inspired us to build our first lstm model [2]. In addition to that we also looked at another article that has a prediction model on autonomous vehicles [1]. It used convolution and some other ideas that we didn't really build upon but looked into to gain a new perspective on how to approach the data. From looking at these two articles we gained new insight on how to look at the our data and analyze it. We expected an lstm would be substantial from the beginning so we took heavy inspiration from [2]. Similar to the article we trained our lstm on the first 5 seconds entirely but used a for loop to output the 6 seconds individually. Each loop would use the last output to predict the next one accounting for the sequence and time relations in the challenge. The first article really helped us a lot in understanding how to interpret and analyze the data from a basic stand point [2]. However, late on we see that our lstm model is in fact not that effective compared to some of the simpler models that we had made earlier.

1.3

Our goal is to correctly predict the next 6 seconds of travel for an AV. One input example consists of 10 (x,y) coordinate positions per second for 5 seconds; thus, 50 (x,y) positions of travel is one input. The output consists of 60 (x,y) positions, therefore capturing 6 seconds of travel. The overall input in one city is therefore shaped as such: (N, 50, 2) where N stands for the number of scenes in a particular city. The overall output in one city is the same but for 60 positions: (N, 60, 2). We can represent our training data is S = (xi, yi) where i ranges from (1,50). A mathematical formula that could represent our input is that given an input ((x1,y1), (x2,y2), ..., (x50, y50)) we have to produce an output of (((x1,y1), (x2,y2), ..., (x60, y60))) for each scene in the dataset.

Solving this kind of problem may be applicable to other datasets that contain data with common tracks/paths. A simple continuation of this problem would be other autonomous vehicles and carriers. For instance trucks typically have to take different routes based on the height of bridges, therefore they can't enter certain roads. A dataset trained on trucks traveling would learn from the paths taken by those trucks to not go those paths. Autonomous drones, too, we can apply this to also capture altitude and extend the problem to (x,y,z) coordinates and learn the paths of those carriers.

As mentioned earlier this can also help navigational/GPS systems predict with higher accuracy speeds of certain vehicles in any given location, we can add a timestamp to every input/output to figure out traffic data and thereby optimize paths based on a current/provided timestamp.

So the general intuition is that by applying the same or similar process from this dataset, we can learn from datasets that contain other variables like altitude, other dimensions, time, and so on.

2 Exploratory Data Analysis

2.1

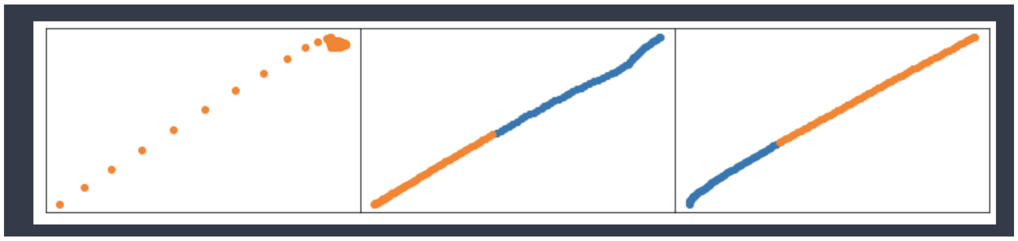
Training:

Palo-Alto: (11993, 50, 2), (11993, 60, 2)
Austin: (43041, 50, 2), (43041, 60, 2)
Miami: (55029, 50, 2), (55029, 60, 2)
Pittsburgh: (43544, 50, 2), (43544, 60, 2)
Dearborn: (24465, 50, 2), (24465, 60, 2)
Washington-DC: (25744, 50, 2), (25744, 60, 2)

Testing:

Palo-Alto: (1686, 50, 2)
Austin: (6325, 50, 2)
Miami: (7971, 50, 2)
Pittsburgh: (6361, 50, 2)
Dearborn: (3671, 50, 2)
Washington-DC: (3829, 50, 2)

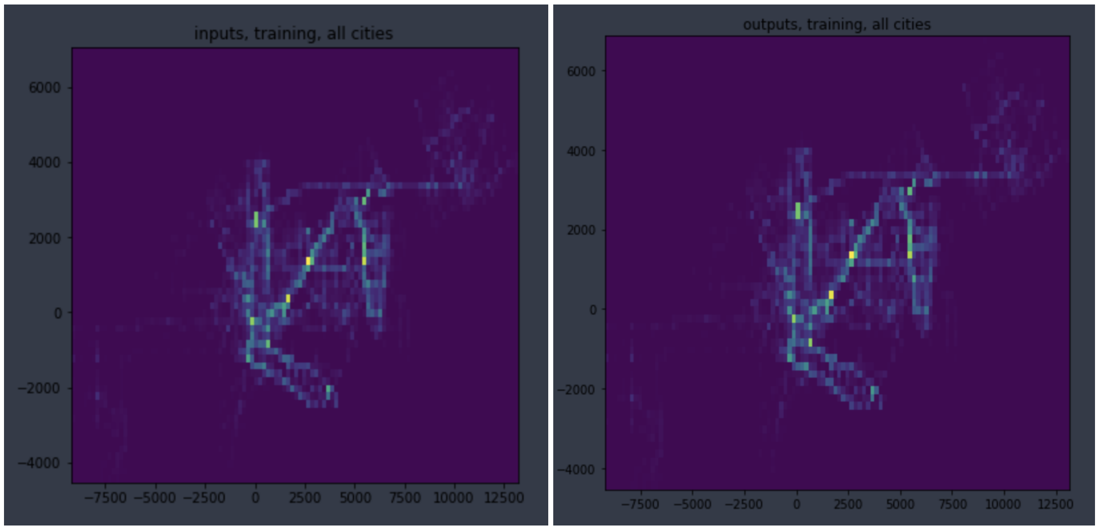
Any input/output combination appears like this:



In the figure above, blue dots represent the 50 inputs and orange dots represent the 60 outputs.

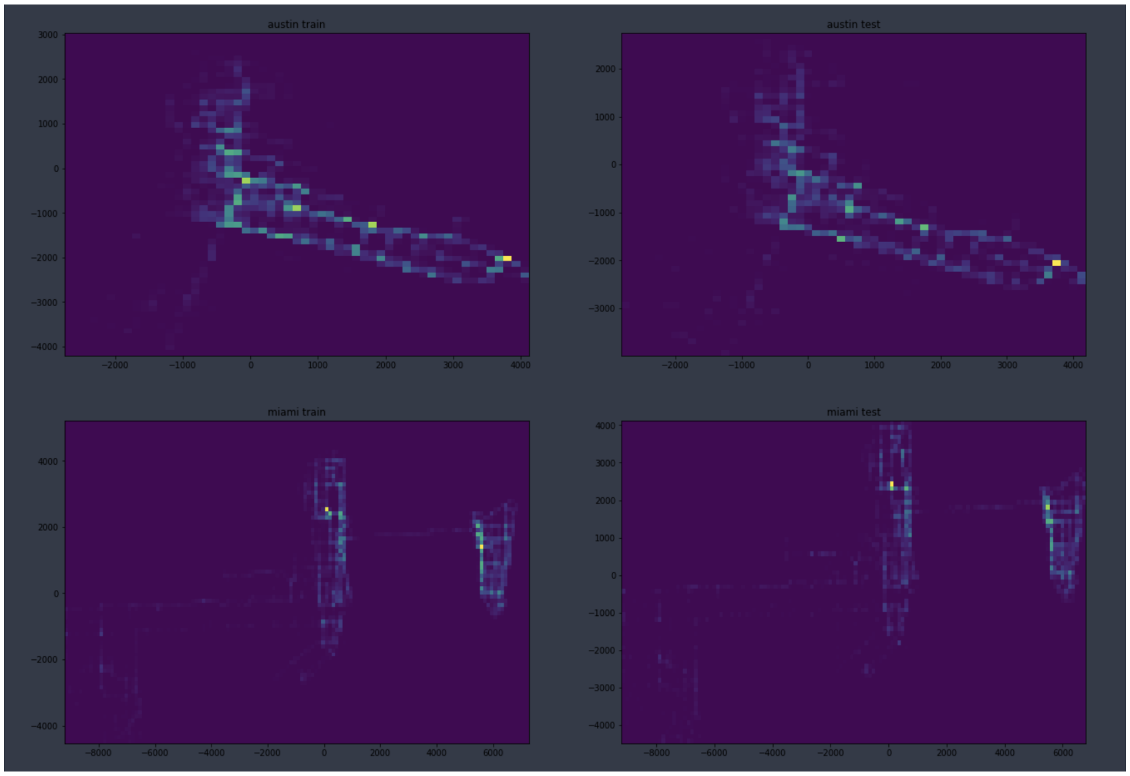
2.2

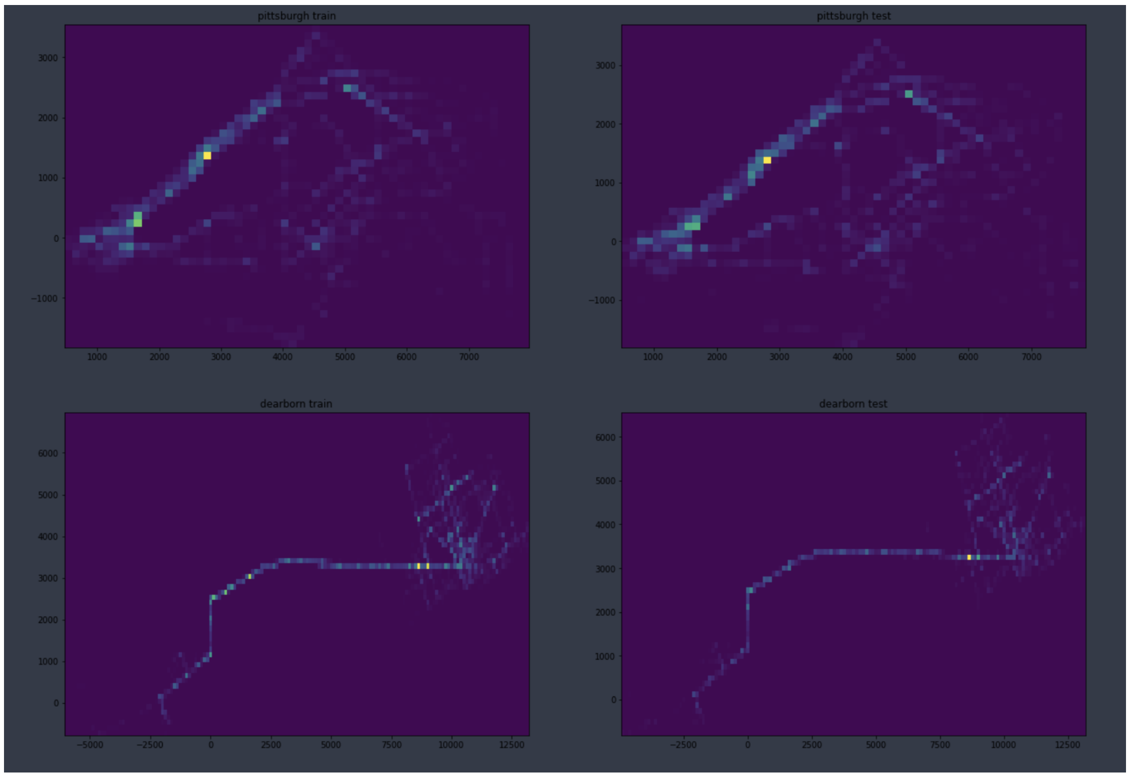
Below are the distributions for inputs and outputs for all the cities combined.

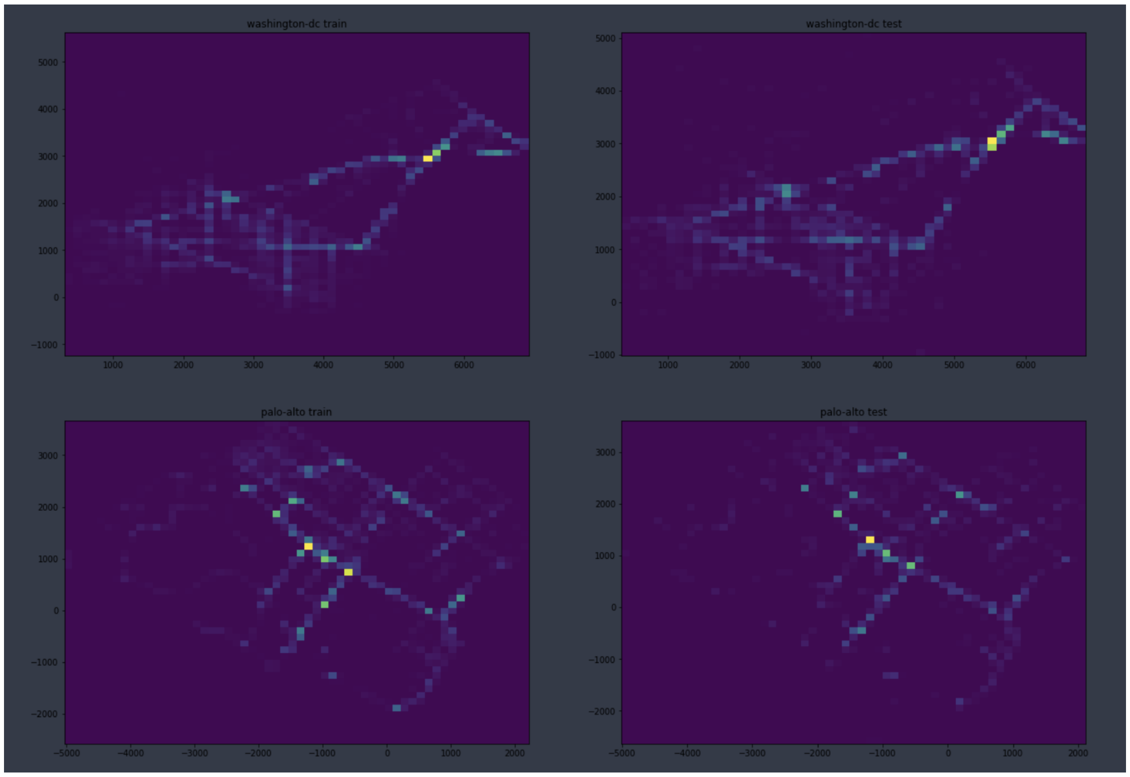


The distribution of inputs and outputs appears similar.

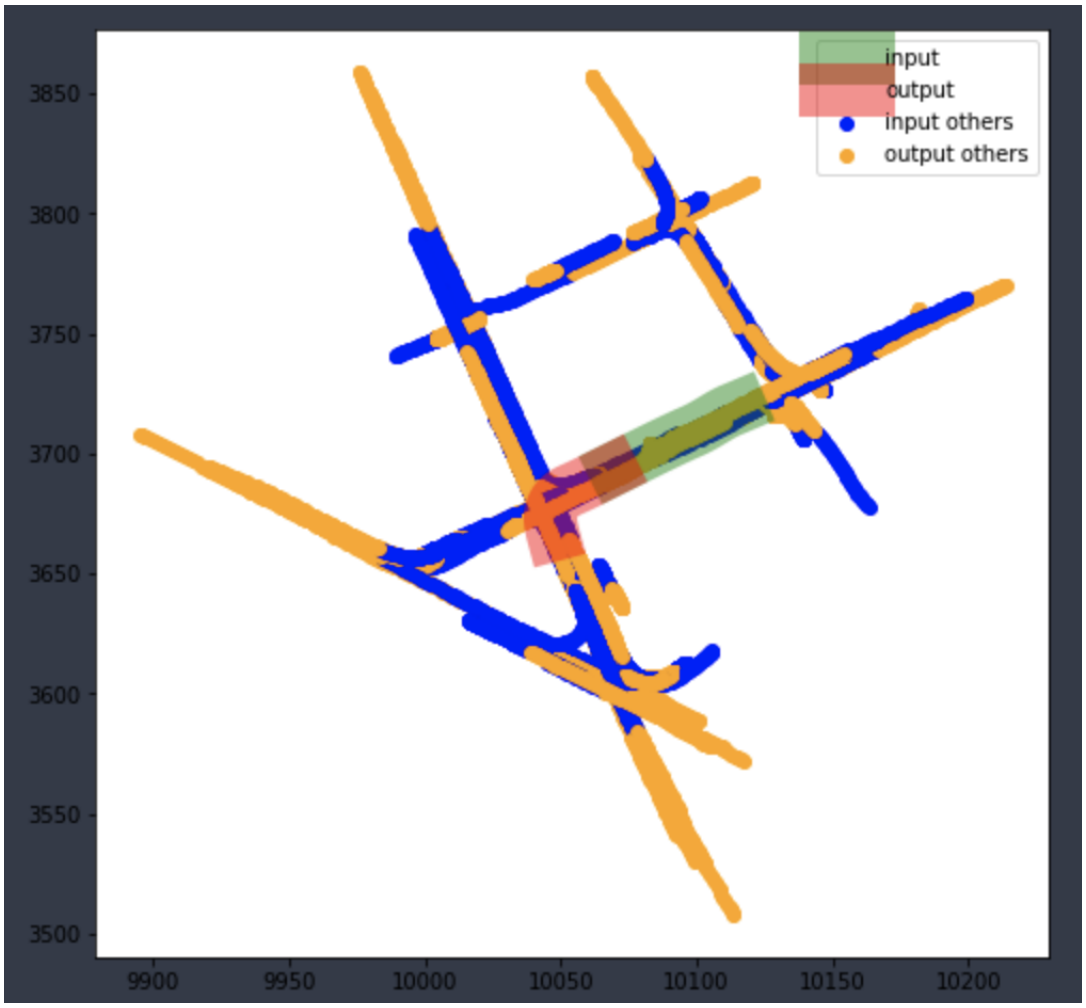
Below we will see the input and output positions for train and test datasets for each city.







From the above it appears that the train and test data are coming from roughly the same distribution. Based on the similar layouts between train and test in each city, we also observe that these are likely city roads, so we further attempt to visualize the roads.



The graph above aggregates 100 pieces of data that are close to one another from a given city vividly producing a road with multiple lanes (see how there are turns, it looks like a road, and cars travel in separate lanes hence the lines are side by side and not always overlapping).

It would have been nice to visualize more precisely the lanes themselves from this data and be able to use that data of lanes to predict future movement of cars. While we didn't get the chance to visualize that, we still attempted to create a feature of 10 closest paths taken for any one given input, so that the training data is trained biasing the closest paths and takes the next closest point iteratively; more on this later.

2.3

Training dataset was 80%, validation was 20% and we did not have a test dataset (other than the testing dataset provided for Kaggle submission).

We used feature engineering as described in section 2.2 to generate paths typically taken in the vicinity of the stated input. We took the average coordinate from any input and using the set of all average coordinates, we assigned 10 other closest inputs/outputs from the training dataset as a feature so that we could learn by biasing those closer paths. The intuition behind that was that if we can identify the closest paths to any one given path that we're trying to predict for, it's more likely that the closest paths would give us information on how the path we're taking in our prediction is constructed, such as whether there's a right turn or left turn or we go straight, and things like speed limits and stop signs and such.

We did not normalize our data because in the models that we built using Linear, Conv and LSTMs, any kind of normalization did not help but only hindered our validation sets. So we simply left it as is without normalization.

We used the city information to perform the feature engineering described above, to generate the closest 10 points for some of our better performing models. However, it's important to note that the feature generated data did not perform as well as we wanted them to perform because they required too much manual model building which we could not finish in time. With the lackluster model that we built using the feature engineered data we got a Kaggle loss of over 1,000, so it did not suffice as a good model for us, since out best model which was simple linear regression had loss of around 27.

3 Machine Learning Model

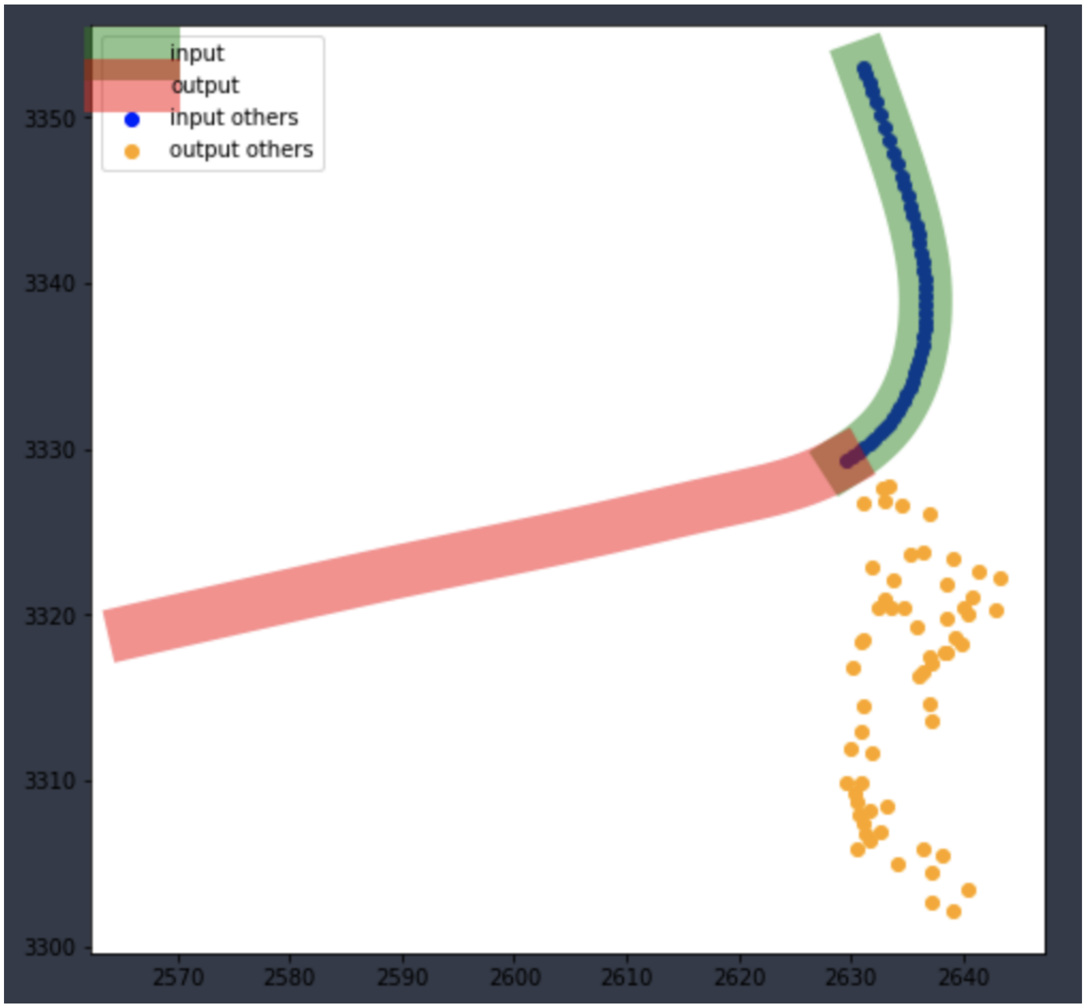
3.1

For our simplest and best performing deep learning model we used a Linear model that performed a (100) to (120) transformation, therefore starting from 50 (x,y) and finishing with 60 (x,y) coordinates. Our loss function was MSE, proportional to RMSE which was used for the Kaggle performance metric. We used the Adam optimizer for that model which performed the best for us.

3.2

We attempted many, many alternative deep learning models.

We started with a linear model that was somehwat convoluted, including many hidden layers. The more hidden layers we took out the better the result we achieved. We started with approximately 6 layers and cut it down to just 1 that would get the best validation result of about 80 RMSE loss. We ran approximately 10,000 epochs on our best performing linear model and iterated the validation set until our validation set's RMSE loss started consistently increasing, at which point we stopped the training and figured that the model was becoming overfit. At that point we submitted the test set and achieved our best deep learning RMSE loss of 102, shown below as typical output.

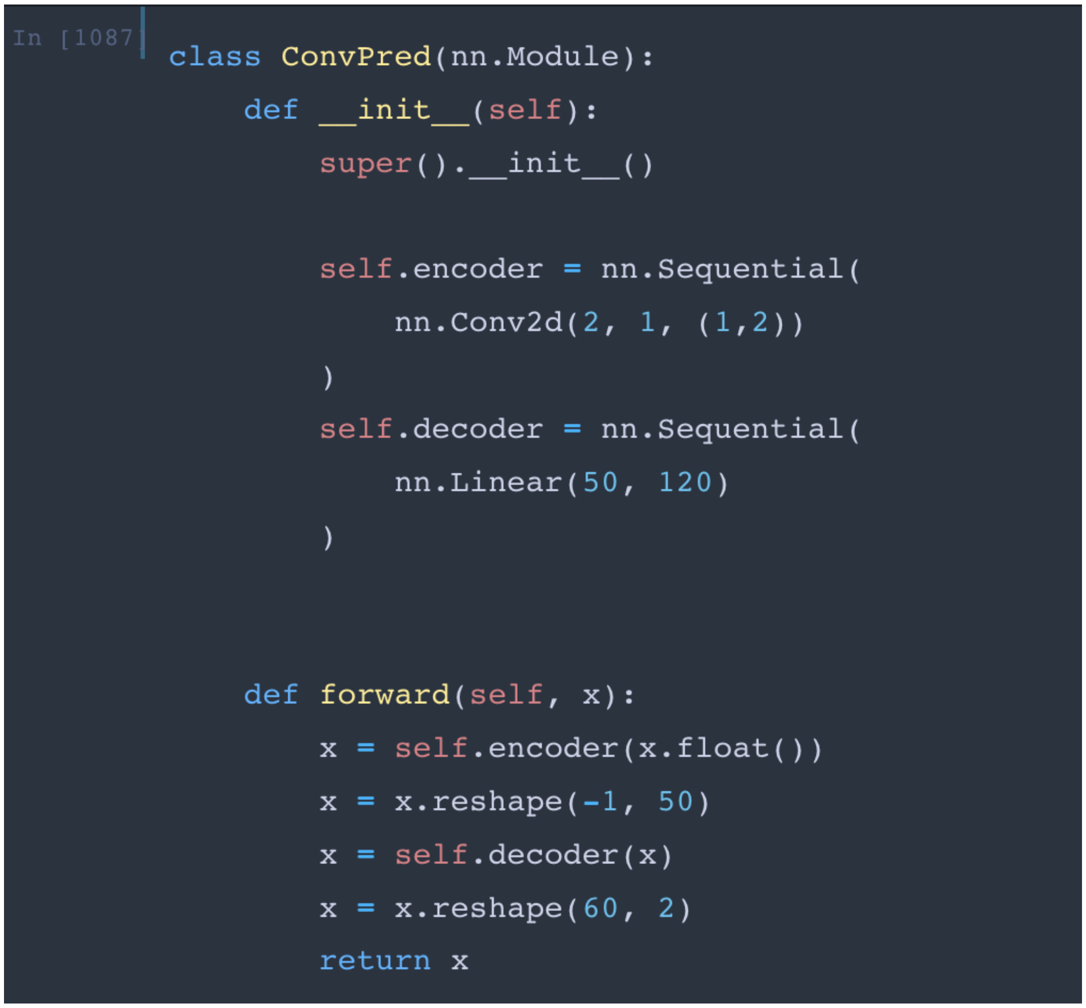


Above we can see that the model was making predictions in the appropriate range of values but it was obvious to the eye that we could have manually made much better predictions. This is why we tried new models.

3.3

We tried improving the prior score by adding features into our model such as the feature engineering steps described earlier, and the best loss RMSE we got was 1300.

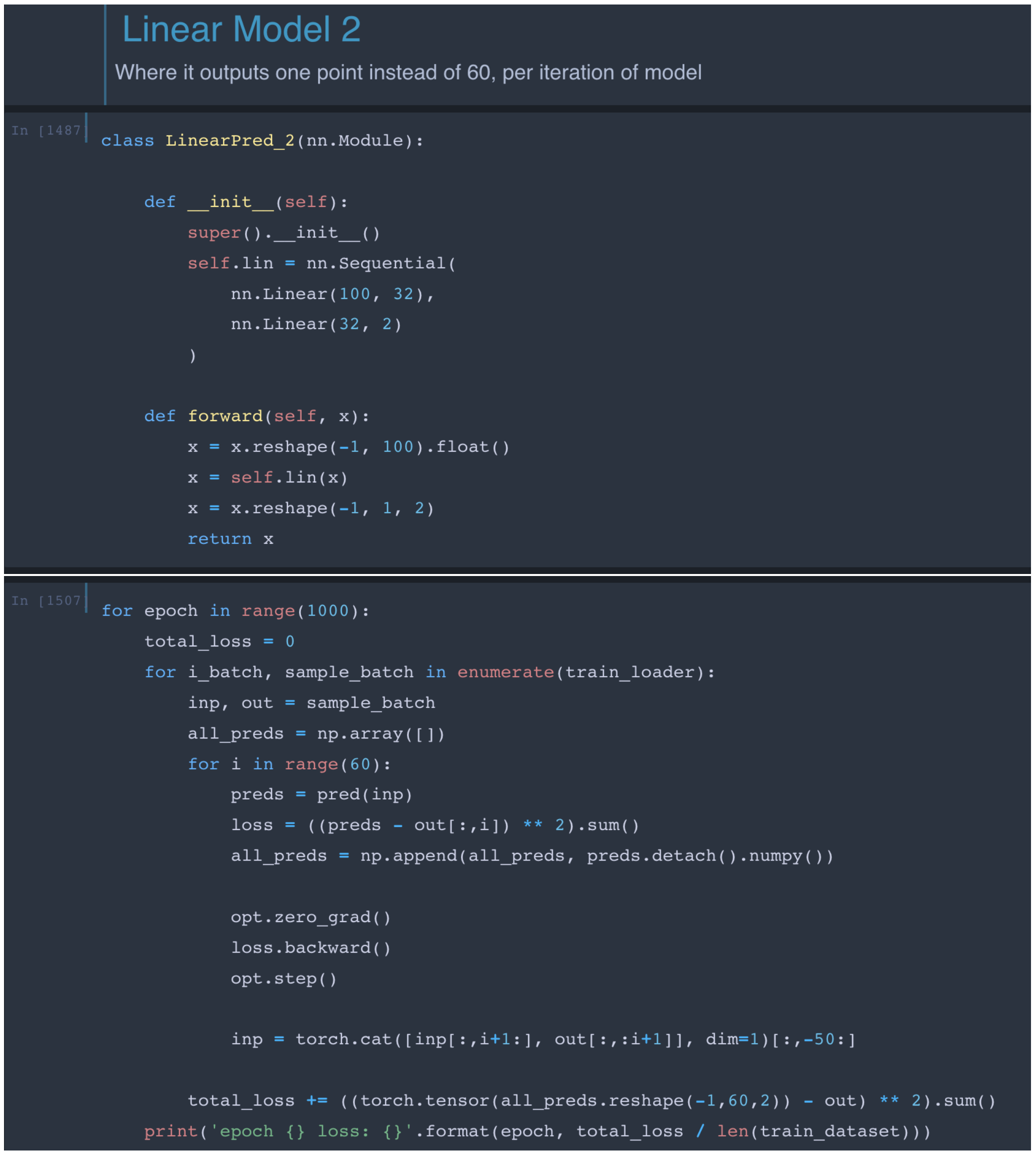
We tried convolution thinking about the architecture that maybe the speeds and directions can be represented as a set of high-degree polynomials that would draw a line as a path. This convolution experimented with multiple layers and multiple different filters and sizes and strides and padding and pooling even. None of the combinations worked as well as the simple linear model, so we stopped with convolution. We attempted the same sequence of events on an LSTM model.



```
In [1087]: class ConvPred(nn.Module):
               def __init__(self):
                   super().__init__()

                   self.encoder = nn.Sequential(
                       nn.Conv2d(2, 1, (1,2))
                   )
                   self.decoder = nn.Sequential(
                       nn.Linear(50, 120)
                   )



               def forward(self, x):
                   x = self.encoder(x.float())
                   x = x.reshape(-1, 50)
                   x = self.decoder(x)
                   x = x.reshape(60, 2)
                   return x
```

Above you can see one of our simpelr convolution models that did not perform well.

We also built a one-by-one model that would only output one coordinate at a time instead of all 60 and that did not work well at all and achieved MSE losses in the billions before we stopped trying with that model.


Linear Model 2

Where it outputs one point instead of 60, per iteration of model

In [1487]
```python
class LinearPred_2(nn.Module):

    def __init__(self):
        super().__init__()
        self.lin = nn.Sequential(
            nn.Linear(100, 32),
            nn.Linear(32, 2)
        )

    def forward(self, x):
        x = x.reshape(-1, 100).float()
        x = self.lin(x)
        x = x.reshape(-1, 1, 2)
        return x
```

In [1507]
```python
for epoch in range(1000):
    total_loss = 0
    for i_batch, sample_batch in enumerate(train_loader):
        inp, out = sample_batch
        all_preds = np.array([])
        for i in range(60):
            preds = pred(inp)
            loss = ((preds - out[:,i]) ** 2).sum()
            all_preds = np.append(all_preds, preds.detach().numpy())

            opt.zero_grad()
            loss.backward()
            opt.step()

            inp = torch.cat([inp[:,i+1:], out[:,:i+1]], dim=1)[:,-50:]

        total_loss += ((torch.tensor(all_preds.reshape(-1,60,2)) - out) ** 2).sum()
    print('epoch {} loss: {}'.format(epoch, total_loss / len(train_dataset)))
```

Above you can see our implementation of the one-by-one output model that did not perform well either.

It felt as if the best performing models were the simplest, at least in the start. So our best performer was actually a simple linear regression model where we grabbed the last 2 points of the input and added the difference between those two points therefore to draw a straight line (regression) using those 2 points with the same speed. That model after a few iterations of how many points to use converged at 27.8 RMSE loss, our best result. We attempted to make better predictions using the features we engineered but that model was difficult to manually implement, and we do acknowledge that it could have been better if we spent more time, but with the time we had we could not achieve a better model performance. The problem with that new custom model was that it stopped at a certain point because not all roads were perfectly drawn out, and sometimes our model decided to switch paths out of the 10 closest paths, ending up in the wrong lane and possibly taking a turn when a straight road was provided, and vice versa. Because the majority of the roads were straight it was better to keep out simple linear regression model. If we drew out better paths from the nearby roads and defined more precise features for where any turn is located and the exact lane our AV is in, we're confident we could have achieved sub-15 RMSE loss.

4 Experiment Design and Results

4.1

Some of our models were extremely slow (the poor performing ones like one-by-one coordinate based, LSTM, Convolution) to train so we attempted an upgraded version of Google Colab with 12 GB of RAM as well as Datahub provided by UCSD with 16 GB of RAM. While they improved the training speeds we still did not achieve convergence, and when we did our models were really bad.

We tried with different optimizers but the majority of our models performed the best with Adam optimizer.

We experimented with batching a lot and it really helped the linear models both converge and get faster, to the point where the 102 loss RMSE we ran on our personal machines with thousands of epochs and no problem. Working with the simpler linear models it took us about a minute for every 100 epochs which we were happy with. We left the personal computers running and the function would break (stop performing backpropagation) when the validation started increasing relatively.

We do not think that most of our limitations and lackluster complicated models were limited by our hardware because when we looked into research behind the number of epochs we had to train our models on required only dozens of epochs to achieve convergent accuracy, and that number of epochs did take about 10-15 minutes but we immediately could see those models are not converging.

4.2

Model Training Times				
Convolution Model	LSTM Model	Linear Regression Model	1 by 1 model	Linear Model
about 300 million	about 700 million	about 729	about 2 billion	about 10k

From this table we can see that the linear regression model performed the best even though it is one of the simplest models we created. When looking into our LSTMs and convolution models we weren't able to make much progress. Our loss was really high and we couldn't decipher the problem regardless of trying multiple deep learning techniques. Therefore we stuck with our linear regression model and continued to improve and develop it the best we could.

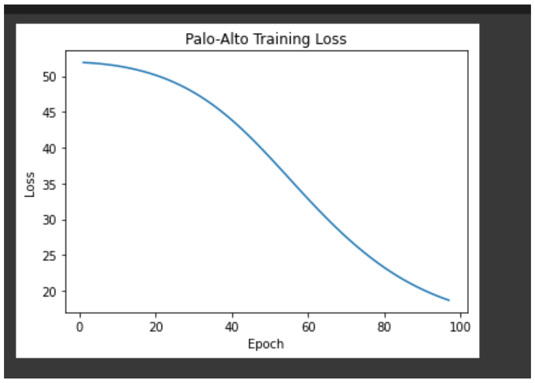
Model Training Times				
Convolution Model	LSTM Model	Linear Regression Model	1 by 1 model	Linear Model
1 minute per epoch	around 45 minutes for 50 epochs	30+ minutes for 5000 epochs	1 minute per epoch	30+ minutes for 5000 epochs

To improve our training times we used larger batches and changed our learning rates. In addition that we used different batching techniques like random batching to see if we could improve performance in conjunction with training times.

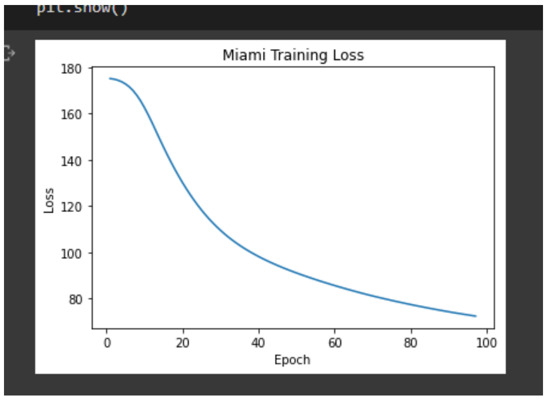
Model Parameters				
Convolution Model	LSTM Model	Linear Regression Model	1 by 1 model	Linear Model
202 parameters	17538 parameters	1 parameter (the number of input coordinates to use for regression prediction)	68 parameters	240 parameters

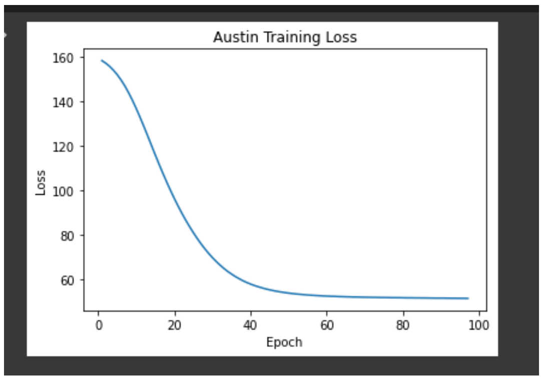
4.3

Our best performing model was actually our most simple model which was linear regression. This was a surprise to us too because when we made convolution and LSTM models the loss was incredibly high. Here are graphs for our loss over the multiple cities.

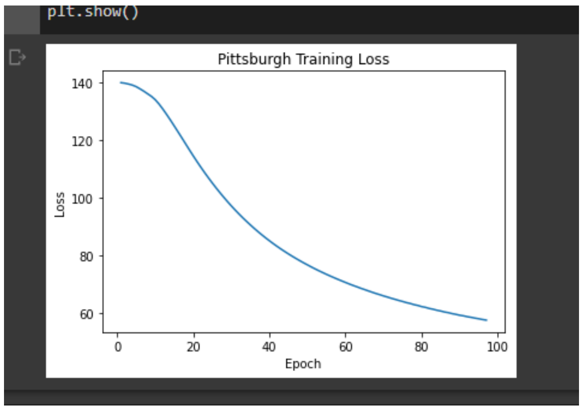


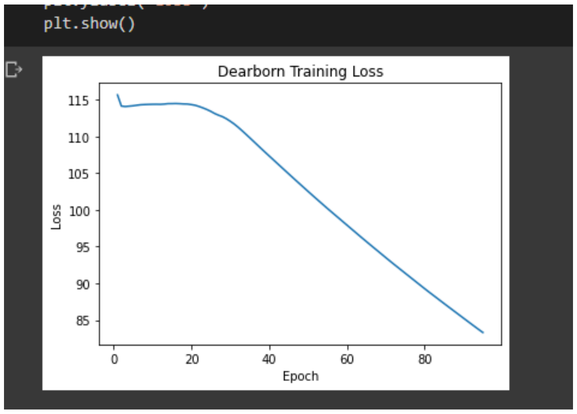
```
plt.show()
```



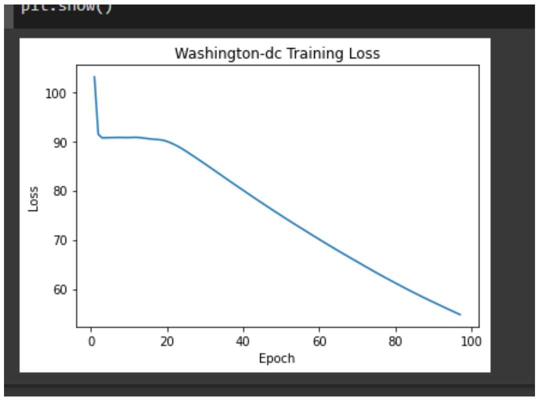


```
plt.show()
```



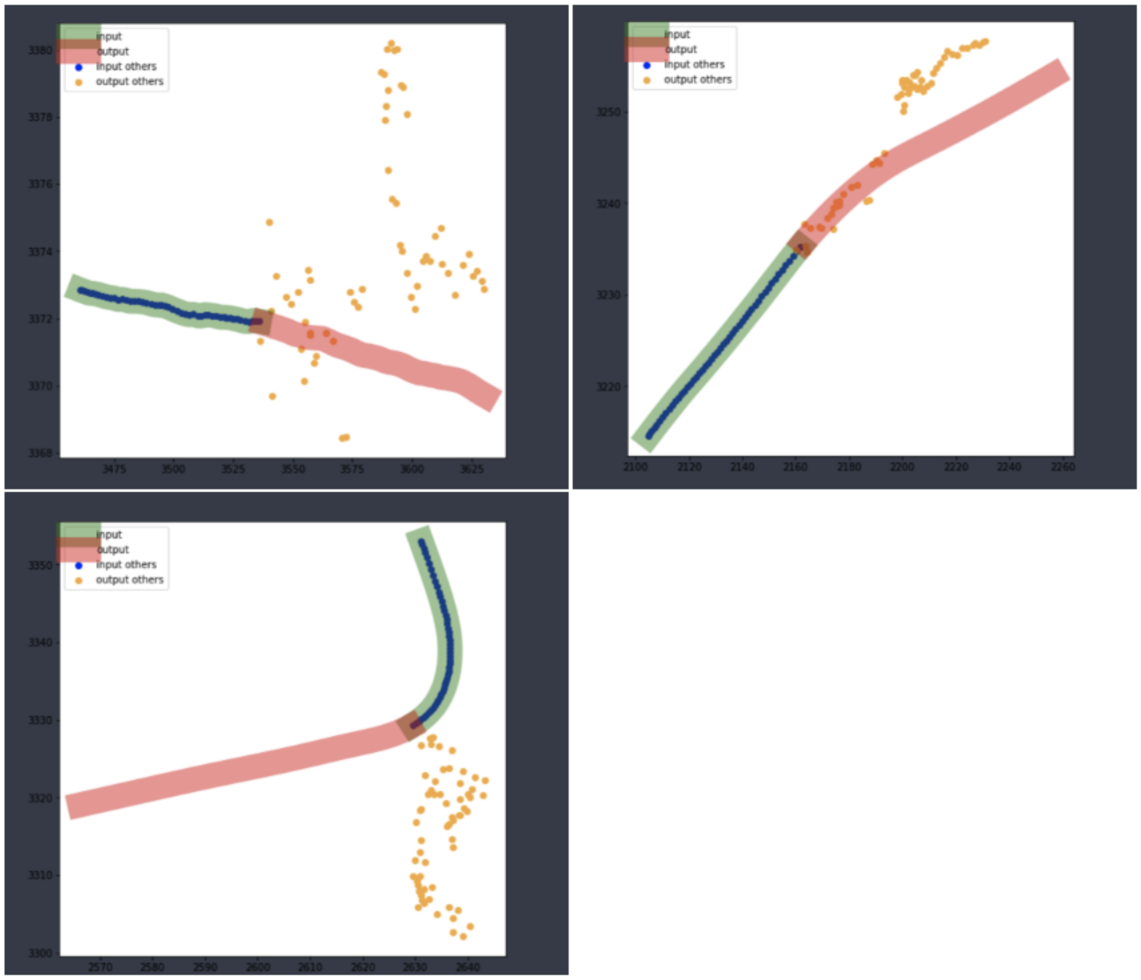


```
plt.show()
```



```
plt.show()
```

From our linear, best-performing deep learning model, we randomly sampled a few samples and here they are in the figure below. The green and red line represent input and correct output, while the orange dots represent the predicted outputs.



In the above figure we acknowledge that this deep learning model did not perform well and we can do better just by performing linear regression. So after this model we performed linear regression and there was no point in visualizing that because we know exactly that it is a line going through the last 2 points in the input.

On Kaggle our final ranking was 20/49 and score 28.18859.

5 Discussion and Future Work

5.1

For feature engineering, to find the most useful features we have to thoroughly analyze the data and test it inside the model. To figure out if a feature is going to be useful we have to firstly define how that feature is going to be placed into the input. Some issues we ran into is that our features were made up of different dimensions, so it was difficult to include them in the deep learning model because it requires features of the same dimensions across every input. So if we were to, say, generate features with varying dimensions then we'd have to use a custom model/function for training on our data, which is what we had to do for the model that resulted in 1300 RMSE loss that used the 10 closest inputs feature we engineered.

Drawing out distributions and learning about the data in general really helped in understanding what the data was. We were going into this project understanding nothing about the use of this dataset (why are we predicting the track of a vehicle without even knowing the obstacles or having any video footage? – for instance). So after doing visualization and understanding that data is geographically related to every city and describes a path/road geographically and with really good precision, we really understood the task better. Sometimes finding use cases for a task can also help understanding the task, such as figuring out why exactly we need to learn without seeing video footage is – what if the car camera breaks? There needs to be an alternative source of information. It can also help our model made architectural decisions, for instance if the data is used for navigational systems where road data changes, we can figure out new roads by biasing toward recordings/paths/inputs taken from more current timestamps rather than older ones.

Time was definitely a constraint in this project and we wish we had more time to work on this because we had so many more ideas on how to finish this project after it was over. For instance we wanted to experiment with a deep learning model that identifies a specific feature that helps us in building a custom model, or maybe even multiple models combined together. For instance we could have built a model that accurately predicts the speed of the vehicle in the future which is a simpler task than predicting the exact 60 coordinates, and we could have used those predicted speeds in the linear model, because with them it'd be much easier to stay on pace with the actual output.

For similar prediction tasks, I'd advise deep learning noobies to try out simpler models first, ones that might not even require deep learning, and attempt to improve that model by making slight alterations to identify key problems. Always understand what the purpose of the model is. You can definitely experiment really quickly with architectures of different models but in our experience only understanding what we're doing and writing custom functions really helped achieve good accuracy. Keep it super simple and add complexity only when you can't get the model to be any simpler.

And other than experimenting with speed, we also want to in the future experiment with regression lines that account for turns, so for instance derive or double derive the direction that any path is moving toward to come up with a different slope for any regression, or some kind of sklearn polynomial rather than linear regression based on 2 coordinates. Even further, we could explore how to define a feature that states the continuation of any given road and the possibilities, for instance defining the average speed limit in any location, defining whether we're in a turn lane, a parking lot, going straight, etc. So just continuing with the feature engineering. Only then bother with deep learning and such.

References

[1] Woven Planet Level 5. How to build a motion prediction model for autonomous vehicles, October 2021.

[2] Charlie O'Neill. Pytorch lstms for time-series data, January 2022.