Autonomous Vehicle Trajectory Prediction

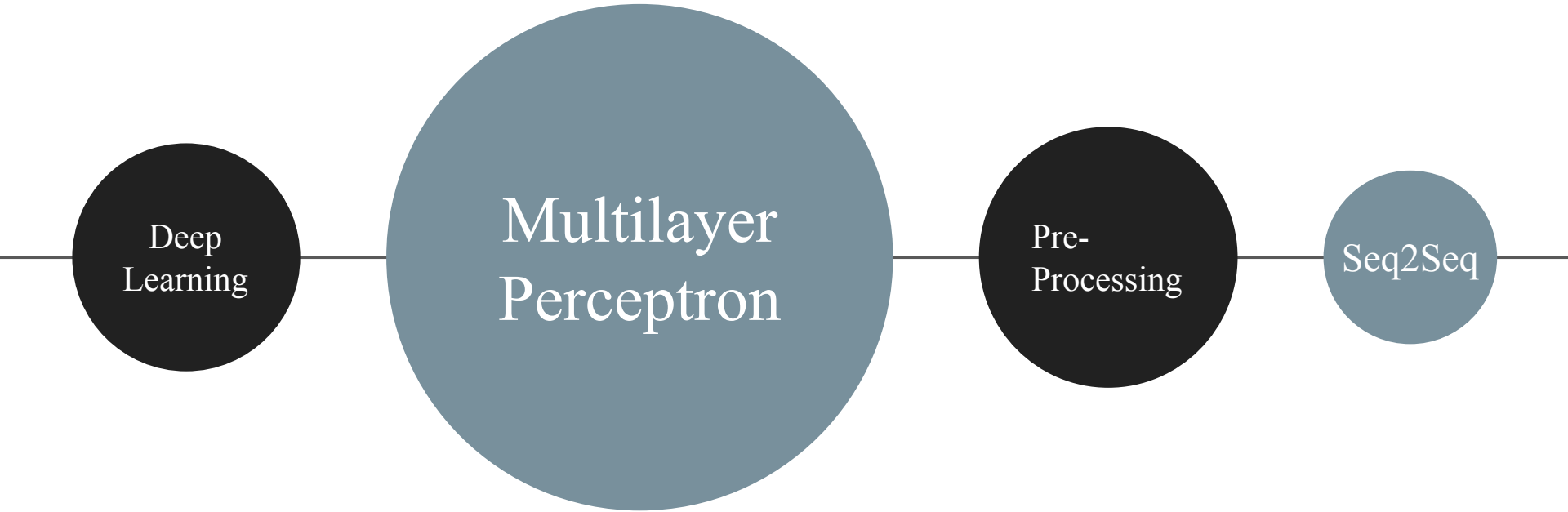
Zairan Xiang Daniel Son

Group: Deep Learned Tacos

Summary

- Who is in your team
  - Zairan Xiang
  - Daniel Son
- How did you solve the problem
  - Linear MLP model with an Encoder Decoder structure
- What have you learned
  - Pre-processing the data had the largest impact in improving the test error
  - Simple model can perform well enough

Key Words


Deep Learning
Multilayer Perceptron
Pre-Processing
Seq2Seq

Introduction

Team Introduction

Daniel Son

- Third year Data Science major

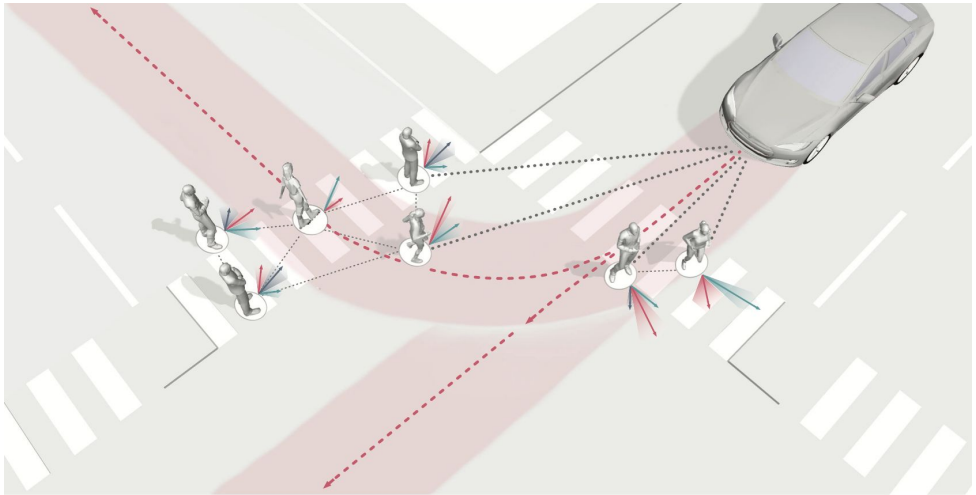
Zairan Xiang

- Third year Data Science major

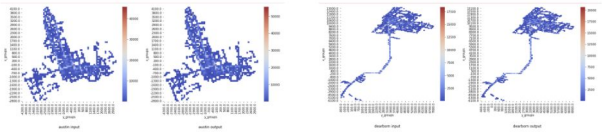
Methodology

Data Processing

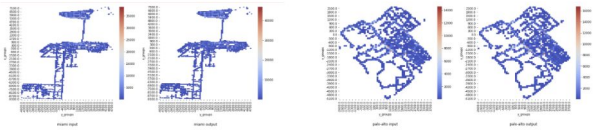
- Converting each datapoint to its relative position to the origin
    - Done by subtracting the position of each data point by the position at the first second of each sequence
    - This was done since the absolute positions did not exhibit any clear pattern while the relative positions did



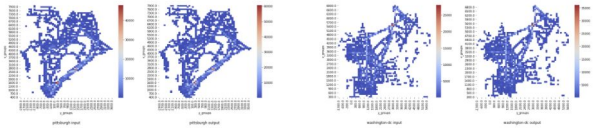
Absolute Positions



(a) Austin (b) Dearborn



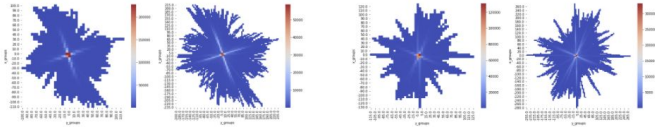
(c) Miami (d) Palo Alto



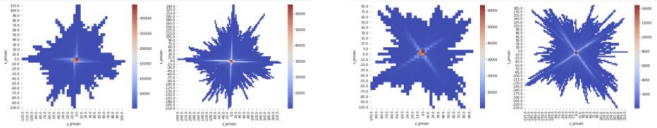
(e) Pittsburgh (f) Washington D.C.

Figure 4: Distribution of Input and Output for each city

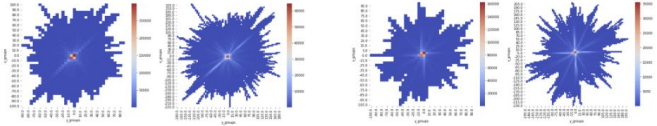
Relative Positions



(a) Austin (b) Dearborn



(c) Miami (d) Palo Alto



(e) Pittsburgh (f) Washington D.C.

Figure 5: Distribution of Relative Input and Output for each city

Deep Learning Model

- Linear MLP w/ Encoder Decoder structure
- Data first pre-processed and feature engineered before being passed into this model
- Optimizer : Adam
- Batch size : 4
- Epochs : 50

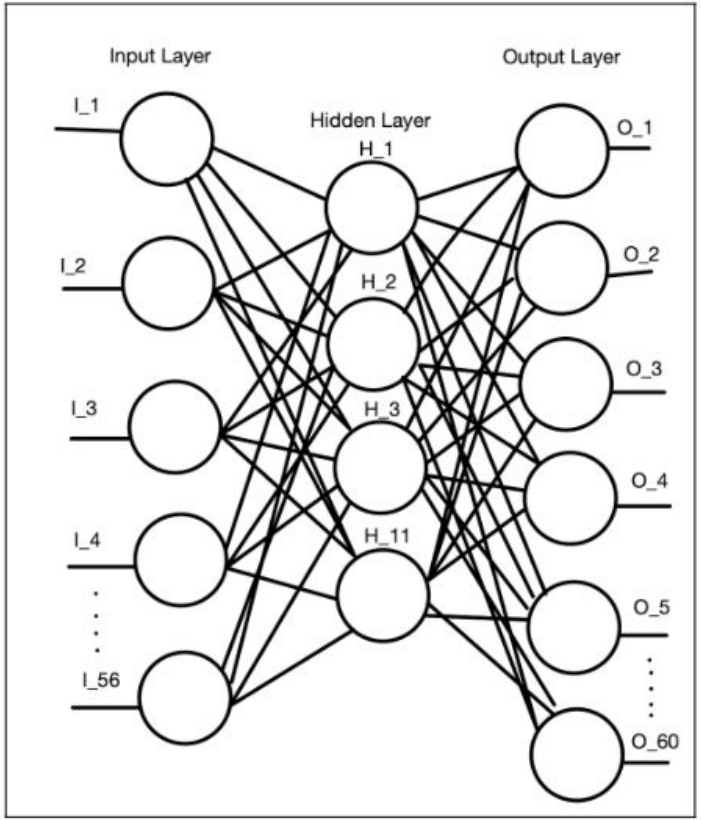
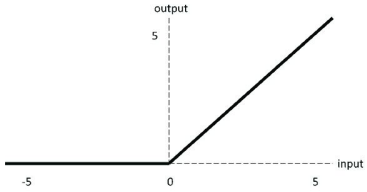


Figure 8: Representation of a Single Layer MLP

Final Model

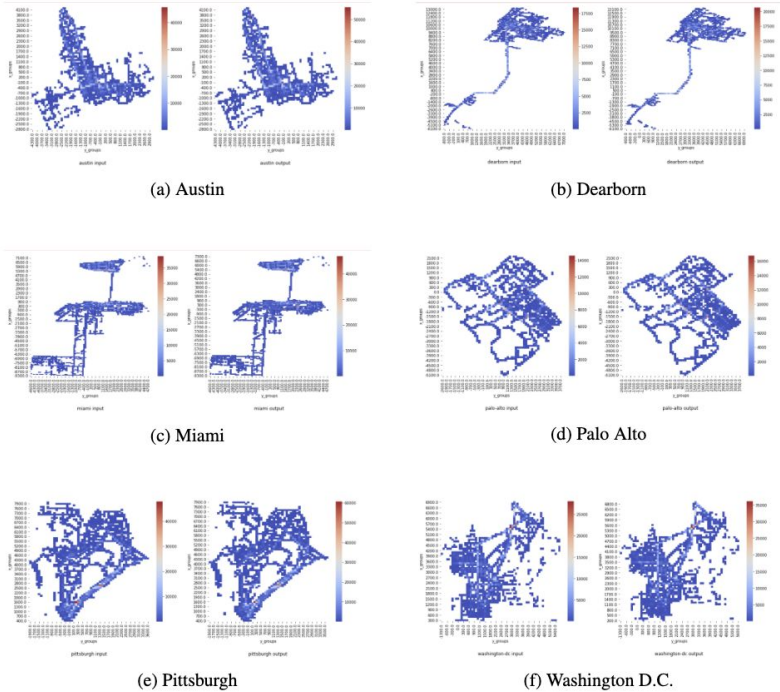


- Multilayer Perceptron with an Encoder Decoder structures, linear layers, one-hot encoded city information, and the data pre-processed to be relative to its origin.
- 5 linear layers in encoder and 4 linear layers in decoder
    - All followed by a ReLU activation function to allow for non-linear predictions

```
class Pred(nn.Module):

    def __init__(self):
        super().__init__()

        self.encoder = nn.Sequential(

            nn.Linear(400, 240),
            nn.ReLU(),
            nn.Linear(240, 96),
            nn.ReLU(),
            nn.Linear(96, 48),
            nn.ReLU(),
            nn.Linear(48, 16),
            nn.ReLU(),
            nn.Linear(16, 16)
        )

        self.decoder = nn.Sequential(

            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, 64),
            nn.ReLU(),
            nn.Linear(64, 120),
            nn.ReLU(),
            nn.Linear(120, 120)
        )

    def forward(self, x):
        x = x.reshape(-1, 400).float()
        x = self.encoder(x)
        x = self.decoder(x)
        x = x.reshape(-1, 60, 2)
        return x
```

Engineering Tricks

- One-hot encoding city information into the data
  - Appended onto every data point in addition to the X and Y coordinates
  - Ex: Austin w/ coordinates 56, 47 will become (56, 47, 1, 0, 0, 0, 0, 0).
  - Need to keep city information since there are varying driving habits between different cities



(a) Austin (b) Dearborn (c) Miami (d) Palo Alto (e) Pittsburgh (f) Washington D.C.

Figure 4: Distribution of Input and Output for each city

```
city_one_hot = {"austin": [1, 0, 0, 0, 0, 0], "miami": [0, 1, 0, 0, 0, 0], \
                "pittsburgh": [0, 0, 1, 0, 0, 0], "dearborn": [0, 0, 0, 1, 0, 0], \
                "washington-dc": [0, 0, 0, 0, 1, 0], "palo-alto": [0, 0, 0, 0, 0, 1]}
```

Experiments

Summary of Past and Final Models

Table 1: Model Designs and Performance

Model Design	MSE Training Loss	MSE Test Loss	Training Time /epoch	Parameters
Encoder Decoder RNN with LSTM	4382047.54824	9993511.61486	5 min	676000
MLP Encoder Decoder run on all cities	24783.68309389	13716.46251	2-3 min	150064
MLP Encoder Decoder run on city separately	821.59873	544.36019	1-2 min	150064
MLP Enc Dec w/ one-hot encoding of city	462.86587	395.99066	2-3 min	150064
MLP Enc Dec w/ one-hot and relative positions	29.87579	26.71983	2-3 min	150064

Experiment 1: Batch Size

- The batch size we chose was 4.
- If we made it larger (50+), the training error was higher.

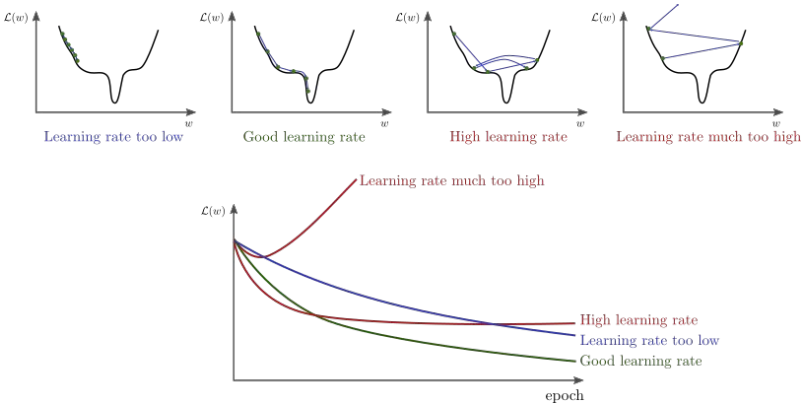
Experiment 2: Number of Epochs

- The number of epochs conducted was 50, which took about a hour to run.
- We found that around 40 epochs, the errors started to converge.
- When testing it with 100 epochs, the error still continued to minimally decrease, but the testing error was much higher than the training error, meaning the model had overfit the data.

Experiment 3: Learning Rate

- For the learning rate, the one that we found worked the best was 1e-3.
- If it was increased to 1e-2, we found that the training error would bounce around and converge earlier at a higher training error.
- If the learning rate was reduced to 1e-4 then we found the model learned very slowly and it required many more epochs to get a similar training error to the one we achieved with a rate of 1e-3.

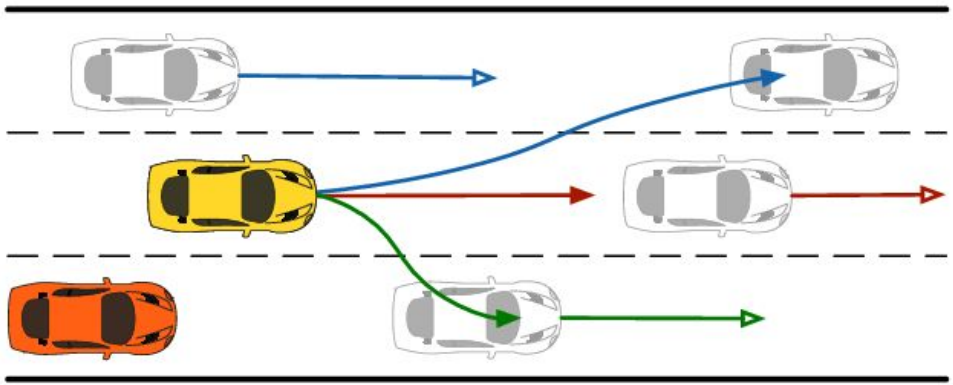
Due to these considerations, we went with a learning rate of 1e-3 for most of our testing and training since it was found to be the most balanced option for most models



Discussion

What have you learned

- Pre-processing the data had the largest effect on improving the testing error
- Batch size and number of epochs also contributes significantly to the model's performance
- Testing the model requires a lot of time for the model to run and lots of computational power
- Simple model classes can still output complex predictions with proper preparation



Future Work

- Other pre-processing and features engineering techniques
- Moving to a more complex model
  - Bidirectional LSTM
    - Theoretically ideal for sequential data such as the current data
- Optimal normalization methods