Skeleton for Question 2 Answer

1 Team

Team Members	*Oliver Meng d8u2b, Yilin Yang e4l2b, Austin He v0b1b, Yuan Ou k3k1b*
Kaggle Team Name	*LaiGenHuaZi*

2 Introduction (3 *points*)

We are given data of a ego vehicle and it's 9 nearest surrounding objects' movement in the past 1000ms. We try to predict the movement of the ego vehicle in the next 3000ms, conditioning on the past data, using a ego-centric model. The past data contains the ego vehicles movement in response to the surrounding agents in different scenarios. The model is trained based on these different scenarios in order to predict its movement in the next 3 seconds.

3 Summary (12 *points*)

Several paragraphs describing the approach you took to address the problem.

We first looked at the data and realized each .csv file is a scenario where agent is being observed and the dataset includes positions of the agent and its neighbors over a period of 1 second. And we want to predict the trajectory of the agent over the next 3 seconds.

We ended up using a MLP model to train the dataset. We preprocessed the data to form X and y for the model. X in our case is a matrix where each agent is the example (row of the matrix) and each example has features: 1. positions of the agent over the first second 2. positions of 2 randomly chosen agent's neighbors over the first second. For y, we used the positions of the agent over the next 3 seconds. To have better performance, we trained the x coordinate and y coordinate separately - we used two MLP models: one trains on x positions, the other trains on y positions, which give a better result than putting them together in one model. The reason why we used MLP model is that it is highly customized and it introduces a non-linear activation function to fit the non-linear dataset, which we think fits this question because vehicle's movement can be non-linear (turning right, turning left, merging, etc.)

4 Experiments (15 *points*)

Several paragraphs describing the experiments you ran in the process of developing your Kaggle competition final entry, including how you went about data prepossessing, feature engineering, model, hyper-parameter tuning, evaluation, and so forth.

We preprocessed the data to form X and y for the model. X in our case is a matrix where each agent is the example (row of the matrix) and each example has features: 1. positions of the agent over the first second 2. positions of 2 randomly chosen agent's neighbors over the first second. For y, we used the positions of the agent over the next 3 seconds. To have better performance, we trained the x coordinate and y coordinate separately - we used two MLP models: one trains on x positions, the other trains on y positions, which give a better result than putting them together in one model.

We primarily selected only the position data as features because these are the data that matter the most and a single MLP model might not be able to handle binary features such as "present" very well.

In terms of model selection, we considered many other options such as the Vector autoregression model

from our midterm Kaggle, but we didn't use it because it was not intuitive for us to form X and y for the model. We also considered linear regression where similar scenarios will have a same trajectory, but this proved to be difficult as we had no way of accounting other agents and we would have to manually separate the sceanarios.

We tuned the hyper-parameters of our model manually. We found that our model is pretty quick so we ended up using normal gradient descent instead of SGD, so the hyper-parameters of our model is mainly the hidden layer sizes for x position model and y position model. We tuned them manually and selected the hyper-parameters with the lowest score. We ended up using the parameters hidden layer [50] and **max_iter = 10000** for our x positions and hidden layer [30] and **max_iter = 10000** for our y positions.

5 Results (5 *points*)

Team Name	Kaggle Score
LaiGenHuaZi	*1.85844*

6 Conclusion (5 *points*)

Several paragraphs describing what you learned in attempting to solve this problem, what you might have changed to make the solution more valuable, etc.
We learned many data preprocessing techniques while researching for possible approaches, such one-hot-encoding. And we also learned that we can separately learn two features like what we did for x and y positions. If we had more time to expand on this, we would research more about MLP on time series. And we might even use ensemble method to put together multiple models for better prediction accuracy. Also, feature engineering could have been done with techniques learned from lectures. And validation could have been used to tune our hyper-parameters since validation dataset is provided. We could've also tried different combinations of layer size and number of layers to see if it generates more accurate outputs.

7 Code

Include all the code you have written for the autonomous driving prediction problem.
The code can also be found in our zip files.

```
import os
import argparse
import time
import pickle
import time
# 3rd party libraries
import numpy as np
import pandas as pd
import datetime as datetime
import matplotlib.pyplot as plt
from scipy.optimize import approx_fprime
from sklearn.tree import DecisionTreeClassifier
import utils
import linear_model
import sklearn.metrics
from glob import glob
from neural_net import NeuralNet
```

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('-q','--question', required=True)

    io_args = parser.parse_args()
    question = io_args.question

    if question == "2":
        # Data set
        X_training_filenames = glob(os.path.join('..','..','train','X','X_*.csv'))
        X_training_dataframes = [pd.read_csv(f) for f in X_training_filenames]

        y_training_filenames = glob(os.path.join('..','..','train','y','y_*.csv'))
        y_training_dataframes = [pd.read_csv(f) for f in y_training_filenames]

        # X_validate_filenames = glob(os.path.join('..','val','X','X_*.csv'))
        # X_validate_dataframes = [pd.read_csv(f) for f in X_validate_filenames]

        # y_validate_filenames = glob(os.path.join('..','val','y','y_*.csv'))
        # y_validate_dataframes = [pd.read_csv(f) for f in y_validate_filenames]

        X_test_filenames = glob(os.path.join('..','..','test','X','X_*.csv'))
        X_test_dataframes = [pd.read_csv(f) for f in X_test_filenames]

        print("Finished reading and formatting data!")

        agent_position = 0
        X_x = np.zeros((len(X_training_dataframes), 33))
        X_y = np.zeros((len(X_training_dataframes), 33))
        X_test_x = np.zeros((len(X_test_dataframes), 33))
        X_test_y = np.zeros((len(X_test_dataframes), 33))
        y_x = np.zeros((len(X_training_dataframes), 30))
        y_y = np.zeros((len(X_training_dataframes), 30))

        for x in range(len(X_training_dataframes)):
            result_x_positions = y_training_dataframes[x].loc[:, ' x'].to_numpy()
            result_y_positions = y_training_dataframes[x].loc[:, ' y'].to_numpy()
            result_x_positions = np.resize(result_x_positions, (30,))
            result_y_positions = np.resize(result_y_positions, (30,))

            y_x[x] = result_x_positions
            y_y[x] = result_y_positions

            other_car_limit_train = 2
            other_car_limit_test = 2

            x_positions = np.array([])
            y_positions = np.array([])

            x_positions_test = np.array([])
```

```
    y_positions_test = np.array([])

    agent_found_flag = False
    agent_found_flag_test = False

    for i in range(10):

        if X_training_dataframes[x].iloc[0, 6 * i + 2] == " agent":
            temp_x = X_training_dataframes[x].iloc[:, 6 * i + 2 + 2].to_numpy()
            temp_y = X_training_dataframes[x].iloc[:, 6 * i + 2 + 3].to_numpy()
            x_positions = np.append(temp_x, x_positions)
            y_positions = np.append(temp_y, y_positions)
            agent_found_flag = True

        # select 2 non-agent vehicles
        elif other_car_limit_train > 0:
            x_positions = np.append(x_positions, X_training_dataframes[x]
            .iloc[:, 6 * i + 2 + 2].to_numpy())
            y_positions = np.append(y_positions, X_training_dataframes[x]
            .iloc[:, 6 * i + 2 + 3].to_numpy())

            other_car_limit_train -= 1

        elif other_car_limit_train == 0 and agent_found_flag == True:
            X_x[x] = x_positions
            X_y[x] = y_positions

        if x < len(X_test_dataframes) and
        X_test_dataframes[x].iloc[0, 6 * i + 2] == " agent":
            temp_x = X_test_dataframes[x].iloc[:, 6 * i + 2 + 2].to_numpy()
            temp_y = X_test_dataframes[x].iloc[:, 6 * i + 2 + 3].to_numpy()
            x_positions_test = np.append(temp_x, x_positions_test)
            y_positions_test = np.append(temp_y, y_positions_test)
            agent_found_flag_test = True

        # select 2 non-agent vehicles
        elif x < len(X_test_dataframes) and other_car_limit_test > 0:
            x_positions_test = np.append(x_positions_test, X_test_dataframes[x]
            .iloc[:, 6 * i + 2 + 2].to_numpy())
            y_positions_test = np.append(y_positions_test, X_test_dataframes[x]
            .iloc[:, 6 * i + 2 + 3].to_numpy())

            other_car_limit_test -= 1

        elif x < len(X_test_dataframes) and other_car_limit_test == 0 and
        agent_found_flag_test == True:
            X_test_x[x] = x_positions_test
            X_test_y[x] = y_positions_test


model1 = NeuralNet([50], max_iter=10000)
```

```
        model1.fit(X_x, y_x)

        y_hat_x = model1.predict(X_test_x).flatten()

        model2 = NeuralNet([30], max_iter=10000)
        model2.fit(X_y, y_y)

        y_hat_y = model2.predict(X_test_y).flatten()

        y_hat = np.insert(y_hat_y, np.arange(len(y_hat_x)), y_hat_x)
        pd.DataFrame(y_hat).to_csv("output.csv")

import numpy as np
import findMin
import utils

# helper functions to transform between one big vector of weights
# and a list of layer parameters of the form (W,b)
def flatten_weights(weights):
    return np.concatenate([w.flatten() for w in sum(weights,())])

def unflatten_weights(weights_flat, layer_sizes):
    weights = list()
    counter = 0
    for i in range(len(layer_sizes)-1):
        W_size = layer_sizes[i+1] * layer_sizes[i]
        b_size = layer_sizes[i+1]

        W = np.reshape(weights_flat[counter:counter+W_size],
        (layer_sizes[i+1], layer_sizes[i]))
        counter += W_size

        b = weights_flat[counter:counter+b_size][None]
        counter += b_size

        weights.append((W,b))
    return weights

def log_sum_exp(Z):
    Z_max = np.max(Z, axis=1)
    return Z_max + np.log(np.sum(np.exp(Z - Z_max[:,None]), axis=1)) # per-colmumn max

class NeuralNet():
    # uses sigmoid nonlinearity
    def __init__(self, hidden_layer_sizes, lammy=1, max_iter=100):
        self.hidden_layer_sizes = hidden_layer_sizes
        self.lammy = lammy
        self.max_iter = max_iter

    def funObj(self, weights_flat, X, y):
        weights = unflatten_weights(weights_flat, self.layer_sizes)
```

```
        activations = [X]
        for W, b in weights:
            Z = X @ W.T + b
            X = 1/(1+np.exp(-Z)) # sigmoid
            # X = np.maximum(0, Z) # ReLU
            activations.append(X)

        yhat = Z
        if self.classification: # softmax- TODO: use logsumexp trick to avoid overflow
            tmp = np.sum(np.exp(yhat), axis=1)
            # f = -np.sum(yhat[y.astype(bool)] - np.log(tmp))
            f = -np.sum(yhat[y.astype(bool)] - log_sum_exp(yhat))
            grad = np.exp(yhat) / tmp[:,None] - y
        else:  # L2 loss
            f = 0.5*np.sum((yhat-y)**2)
            grad = yhat-y # gradient for L2 loss

        grad_W = grad.T @ activations[-2]
        grad_b = np.sum(grad, axis=0)

        g = [(grad_W, grad_b)]

        for i in range(len(self.layer_sizes)-2,0,-1):
            W, b = weights[i]
            # grad[Z < 0] = 0 # ReLU gradient
            grad = grad @ W
            grad = grad * (activations[i] * (1-activations[i])) # gradient of
            logistic loss
            grad_W = grad.T @ activations[i-1]
            grad_b = np.sum(grad,axis=0)

            g = [(grad_W, grad_b)] + g # insert to start of list

        g = flatten_weights(g)

        # add L2 regularization
        f += 0.5 * self.lammy * np.sum(weights_flat**2)
        g += self.lammy * weights_flat

        return f, g


    def fit(self, X, y):
        if y.ndim == 1:
            y = y[:,None]

        self.layer_sizes = [X.shape[1]] + self.hidden_layer_sizes + [y.shape[1]]
        self.classification = False # assume it's classification iff y has more
        than 1 column
```

```
        # random init
        scale = 0.01
        weights = list()
        for i in range(len(self.layer_sizes)-1):
            W = scale * np.random.randn(self.layer_sizes[i+1],self.layer_sizes[i])
            b = scale * np.random.randn(1,self.layer_sizes[i+1])
            weights.append((W,b))
        weights_flat = flatten_weights(weights)

        # utils.check_gradient(self, X, y, len(weights_flat), epsilon=1e-6)
        weights_flat_new, f = findMin.findMin(self.funObj, weights_flat, self.max_iter,
        X, y, verbose=3)

        self.weights = unflatten_weights(weights_flat_new, self.layer_sizes)

    def fit_SGD(self, X, y, epoch=10, alpha=1e-3, batch_size=500):
        if y.ndim == 1:
            y = y[:,None]

        self.layer_sizes = [X.shape[1]] + self.hidden_layer_sizes + [y.shape[1]]
        self.classification = False # assume it's classification iff y has more
        than 1 column

        # random init
        scale = 0.01
        weights = list()
        for i in range(len(self.layer_sizes)-1):
            W = scale * np.random.randn(self.layer_sizes[i+1],self.layer_sizes[i])
            b = scale * np.random.randn(1,self.layer_sizes[i+1])
            weights.append((W,b))
        weights_flat = flatten_weights(weights)

        # compute SGD here
        n = X.shape[0]
        for t in range(epoch*n//batch_size):
            if t*batch_size % n == 0:
                f, g = self.funObj(weights_flat, X, y)
                print("Epoch %d, Loss = %f" % ((t*batch_size)//n+1, f))

            batch = np.random.choice(n,size=batch_size,replace=False)

            f, g = self.funObj(weights_flat, X[batch], y[batch])
            weights_flat = weights_flat - alpha * g

        self.weights = unflatten_weights(weights_flat, self.layer_sizes)


    def predict(self, X):
        for W, b in self.weights:
            Z = X @ W.T + b
            X = 1/(1+np.exp(-Z))
```

```
if self.classification:
    return np.argmax(Z, axis=1)
else:
    return Z
```