CPSC 340 Machine Learning Take-Home Final Exam (Fall 2020)

1 Team

Team Members	Ningyuan Xu (74660614) m0t1b Mathew Tang (67759563) f8r1b Grace Yin (48450150) e6k0b Kerry Zhou (28597540) q2k2b Yining Wang (27257245) z8o1b
Kaggle Team Name	*CNN everywhere*

2 Introduction (3 *points*)

The autonomous driving prediction problem, which comes from a subset of data in US Interpret Challenge, requires participants to predict the future trajectory of a self-driving vehicle (ego vehicle) in the next 3000ms, based on the historical moving trajectories of itself and other nine nearest agents (either pedestrians or other vehicles). The problem extracts its data from twenty tracks in an unsigned traffic intersection in US, MA, and divide them into 2307 training, 523 validation, and 20 test sets. Within each training/test/validation example, X contains the x, y coordinates (the origin of coordinates is at the position of ego vehicle at 0s) of all ten agents, types of agents, an indicator whether they are at intersection, and etc. Y represents moving paths of the ego vehicle in the future three seconds, also in terms of x, y coordinates. The test set does not contain Y, and that is what the autonomous driving problem aims at predicting.

3 Summary (12 *points*)

Before fitting models for training data, our group first conducted several exploratory analysis. We visualized the data set using the python script provided on the interaction-dataset Github. Several plots were also made to visualize the data, including distributions of xy points and distributions of car velocities. Finally, we intended to write two distinct models to solve this problem, one in MLP and the other in CNN, and ensemble both results to get the final predictions.

The MLP approach will transform all the training data into a large matrix X and Y, where each row in X and Y represents information in one CSV file that is flattened into a long row. The X row contains x, y coordinates of the predicted vehicle, whether it is at intersections, and the information of the nearest three agents (either pedestrian or vehicle) sorted in ascending order by distance to the ego vehicle. Each time stamp in every row of X, from -1000 ms to 0 ms, will follow the same pattern of column arrangement. At the same time, each row in Y represents the future trajectory (3000ms) of the predicted vehicle also flattened into a row. Data processing in Python is needed to remove examples with missing values. Afterwards, the X and Y matrix will become the input and out matrix for the multi-layer neuron network (MLP). The validation set and test set will undergo the same pre-processing procedures, except that the validation set will be used to tune hyper parameters and the test set will predict the final results for Y. The implementations of MLP model will be the one provided in the assignment code, with few modifications on the predict function.

The CNN works similarly to the MLP model, except that each example CSV becomes an individual matrix with row numbers equal to the number of time periods. Then each of the training matrix will be fed into the CNN model individually to predict the final results. For the CNN implementation, we write code for the filter layer (convolution layer), max pooling layers, and the softmax layers. Forward and backward propagations for these layers are also implemented. The MLP and CNN approaches are similar in the way of matrix transformation and model fitting. (The draft model will be provided in the Appendix section).

4 Experiments (15 *points*)

4.1 Data Visualization and Model Fitting

x_agent	y_agent	if it is at Intersection	x_j, y_j, is vehicle, is at intersection	repeat the former (-1000ms ~ 0ms)
x_1	y_1	$i \in \{0, 1\}$	$\forall j \in$ {nearest three agent to the agent car}	---
x_2	y_2	---	---	---
x_3	y_3	---	---	---
---	---	---	---	---
x_N	y_N	$i \in \{0, 1\}$	---	---

x,y (100ms)	x,y(200ms)	---	x,y (3000ms)
x_{1_1}, y_{1_1}	x_{1_2}, y_{1_2}	---	x_{1_M}, y_{1_M}
x_{2_1}, y_{2_1}	x_{2_2}, y_{2_2}	---	x_{2_M}, y_{2_M}
---	---	---	---
x_{N_1}, y_{N_1}	x_{N_2}, y_{N_2}	---	x_{N_M}, y_{N_M}

Figure 1: Matrix X and matrix y for MLP and CNN models
The top table is matrix X while the bottom table is matrix Y in our model fitting.

Due to the difficulties of implementing gradient functions in CNN, we only make our MLP model fully working in the end. Thus, we will only discuss the MLP approach in this section. The model assumption has been introduced previously. Figure 1 displays the form of matrix X and matrix Y. As a starting point, we combined all the training dataset X_i. Since the type of dataset is time-series, we have to consider them into 2297 subsamples with period 11. For each subsample, we extracted three features of the car whose role is agent: x, y and if the car is at the intersection. x and y are the coordinates of the car, and we believe that the position of a car to the intersection could play a significant role in autonomous driving prediction. In addition, We also extracted these same three features of the three cars closest to the agent car, and added the feature "if it is vehicle ". The reason for us to do such feature selection is that the nearest three cars are the most sensitive and decisive factors of autonomous driving. Moreover, there are many missing values for the "other cars", and there is no way to predict the missing values accurately, so it is reasonable to select the features of the closet three cars. Figure 2 demonstrates the 2D scatter plot of the x-coordinate and y-coordinate of agent cars.

After building matrix X, we cleaned the combined Y training dataset. Again, due to missing value problems and the lack of information for imputation, we dropped the missing values in the combined Y training dataset and removed the corresponding data points from combined training dataset X. After cleaning, the dimension of matrix X matches that of Y.

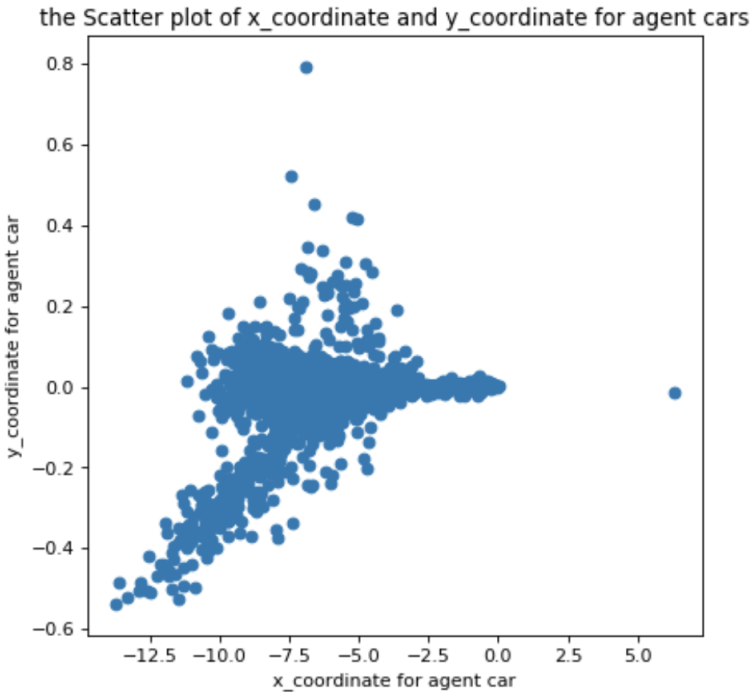

Figure 2: Scatter plot of x and y for Agent Cars

4.2 MLP

Conducting the same transformation of combined X validation dataset and Y validation dataset, we fitted a Multi-layer Perceptron model. The codes are referred to Question 2 in assignment 6. We make some modifications to the MLP implementations by removing unnecessary code for the Softmax loss function. The Softmax loss function is used in MLP models to particularly solve classification problems, such as MNIST classification. Therefore, MLP that relies on Softmax loss function is also called MLPClassifier. The problem of predicting autonomous driving vehicles is NOT a classification problem, but a regression problem. Thus, the MLP will reply on loss functions for linear regression, and this type of problem is also called MLPRegressor. Not only code about Softmax is removed from the original implementations, but also the return variables in the predict function. Instead of returning argmax (which is the case for classification problem), we return the matrix Z equal to the multiplication of weight W and X plus bias vector b. When training the loss function, we can apply gradient descent or stochastic gradient descent. Stochastic gradient descent will be computationally faster, but cannot guarantee the optimal solution (convergence of loss function). In contrast, normal gradient descent will usually lead to a globally optimal and more accurate prediction. Therefore, we used gradient descent instead of stochastic gradient descents in our model.

4.3 Hyperparameter Tuning

Our MLP model contains two hidden layers, each with size 50. For hyperparamter tuning, we attached an importance to regularization parameter λ which is helpful to avoid overfitting, and the maximum iteration. Doing a permutation test, for $\lambda \in \{0.0001, 0.001, 0.01, 0.1\}$ and the maximum iteration $i \in \{100, 200, 300, 400, 500\}$, we noticed that when $\lambda = 0.01$, and the maximum iteration is 100, we obtained the smallest validation error which is 2.60. Therefore, we set $\lambda = 0.01$ and max iteration of 100 for our model for the final prediction.

Figure 3 is the permutation function for hyperparameter Tuning.

```
hidden_layer_choice = [50, 50]
iteration_choice = [100, 200, 300, 400, 500]
lammy_choice = [1,0.1,0.01,0.001,0.0001]

lowest_err = np.inf
best_param = []
for i in iteration_choice:
    for l in lammy_choice:
        model = NeuralNet(hidden_layer_sizes=hidden_layer_choice, max_iter=i, lammy=l)
        model.fit(Xtrain, y_train)
        y_pred_val = model.predict(Xval)
        val_err = compute_error(y_validate, y_pred_val)
        if val_err < lowest_err:
            lowest_err = val_err
            best_param = [i, l]
print(lowest_err, best_param)
```

Figure 3: Hyper parameter tunning

5 Results (5 *points*)

Team Name	Kaggle Score
CNN everywhere	*0.93014*

6 Conclusion (5 *points*)

Predicting future paths of an autonomous driving vehicles is a complicated supervised learning problem that requires advanced neuron network architecture and theories of imitation and deep reinforcement learning. During the competition, our group starts to gain interest in this area and appreciate how difficult it is for data scientists to predict trajectories of moving objects in real life. The model our group uses is deep neuron networks (MLP and CNN). However, MLP and CNN, like traditional neuron networks, has limitations that it cannot predict and modify states of objects and the learning policies during the process of training. Therefore, our model's validation error is not perfect (a bit higher than expected). One way we can make the solution more valuable is to attempt different neuron network architecture, such as recurrent neural network (RNN) and long short-term memory (LSTM). These models are perfectly suited for machine learning problems that involve sequential data, since they have an internal state of "for loop" that allow information to persist in the neuron network. Consequently, recurrent neuron networks better allow the use of previous actions (events) to infer the future ones, which perfectly fits the context of autonomous driving predictions. In fact, machine learning researchers nowadays are continuously looking for more efficient models, and they create attention models, such as ResNet, which may become even more efficient than RNN and LSTM. In conclusion, we still have to try more models to enhance accuracy and improve our test results.

For data preprocessing and matrix transformation, we can also improve our methods by adding more feature columns. Currently, our X matrix only considers three nearest agents to the ego vehicle. In the future we can increase that number to 5–8 to increase dimensions and complexities of the model. We can also add columns indicating whether the nearest agents are pedestrian or vehicles, and so on. In terms of missing data, instead of simply ignoring, we may also leave it as n/a and use preprocessing packages to handle those

missing values. Although these possible enhancement may not actually improve the prediction scores, they are good practice for data scientist and machine learning engineers in the real world. Finally, borrowing ideas from reinforcement learning, we can treat the position to predict as a continuous action space. In the literature, to predict continuous actions, the model usually predict a mean and variance of the action, and the action will be sampled from the Gaussian distribution with that mean and variance. This could better represent how decisions are made in reality and could yield better performance.

7 Code

Note: We reused the MLP implementations from assignment 6, so the MLP code will not be attached below but included in the final zip file submitted.

```
import numpy as np
import pandas as pd
from mlp import NeuralNet

# Exploratory analysis
x_coordinate = X[:,0]
y_coordinate = X[:,1]
plt.figure(figsize=(6, 6), dpi=80)
plt.scatter(x_coordinate,y_coordinate)
plt.title("the Scatter plot of x_coordinate and y_coordinate for Agent Cars")
plt.xlabel("x_coordinate for agent car")
plt.ylabel("y_coordinate for agent car")
plt.draw()


def L2_norm(x1, x2, y1, y2):
    return ((x1 - y2) ** 2 + (y1 - y2) ** 2) ** 0.5


def find_agent(x1):
    n = 5
    k = 3 + 5 * 4
    matrix = np.zeros((11, k))
    for i in range(61):
        if x1[:, i][1] == ' agent':
            matrix[:, 0] = x1[:, i + 2]
            matrix[:, 1] = x1[:, i + 3]
            matrix[:, 2] = x1[:, i + 4]
    return matrix


def find_dist(x1, matrix):
    dist = np.zeros((11, 10))

    for i in range(9):
        k = i * 6 + 4
        dist[:, i] = L2_norm(x1[:, k], x1[:, k + 1], matrix[:, 0], matrix[:, 1])
    #     print("k",dist.shape)
    return dist


def get_small(dist):
    small_index = np.zeros((11, 5))
    #     print("dist",dist)
    for i in range(11):
        cur_dist = dist[i]
        k = 6
        result = np.argsort(cur_dist)
        #         print("small_index",result[1:k])
```

```
        small_index[i] = result[1:k]

    return small_index


def complete_matrix(x1):
    matrix = find_agent(x1)
    dist = find_dist(x1, matrix)
    small_index = get_small(dist)
    N, D = small_index.shape
    #     print(len(small_index))
    for j in range(N):
        for i in range(D):

            raw = small_index[j][i]

            true = int(raw * 6 + 4)

            matrix[j][3 + i * 4] = x1[j][true]
            #              print(i,matrix[j,3+i*2])
            matrix[j][4 + i * 4] = x1[j][true + 1]
            matrix[j][5 + i * 4] = x1[j][true + 2]
            #         print("x1",x1[:,true-1])
            if x1[0][true - 1] == ' car':
                matrix[j][6 + i * 4] = 1
            else:
                matrix[j][6 + i * 4] = 0

    return matrix


def compute_matrix(newdata, n):
    result = []
    for i in range(n):
        x = newdata[i]
        temp = complete_matrix(x)
        temp = temp.flatten()
        result.append(temp)
    # print(result)

    result = np.asarray(result)
    return result


def compute_error(matrix1, matrix2):
    size = matrix1.shape[0] * matrix1.shape[1]
    diff = np.sum(np.square(matrix1 - matrix2)) / size
    return np.sqrt(diff)


if __name__ == '__main__':

    data = pd.read_csv('Xtrain_combined.csv', low_memory=False)
    data.head()
    n=np.shape(data)[1]
    newdata=data.values
    N,D = newdata.shape
    n = int(N/11)
    newdata = np.array_split(newdata, n)
    Xtrain =compute_matrix(newdata,n)

    data = pd.read_csv('Xvalidate_combined.csv', low_memory=False)
    data.head()
    n=np.shape(data)[1]
    newdata=data.values
```

```
N,D = newdata.shape
n = int(N/11)
newdata = np.array_split(newdata, n)
Xval =compute_matrix(newdata,n)
# print(Xval, Xval.shape)

data = pd.read_csv('Xtest_combined.csv', low_memory=False)
data.head()
n=np.shape(data)[1]
newdata=data.values
N,D = newdata.shape
n = int(N/11)
newdata = np.array_split(newdata, n)
Xtest =compute_matrix(newdata,n)

y = pd.read_csv('ytrain_combined.csv')
y = y.values
index_list = np.where(y[:,0] == 3000)[0]
N,D = y.shape
y_train = np.zeros((2297,60))
numitr = 2308
out_index = 0
for i in index_list:
    matrix = y[i-29:i+1,1:]
    matrix = matrix.flatten()
    y_train[out_index] = matrix
    out_index = out_index+1

x_out = np.zeros((2297,99))
place_dist = []
ptr1 = 0
ptr2 = 0
n,d = x_out.shape
org_list = np.where(y[:,0] == 100)[0]
while ptr1 < 2308:
    if y[org_list[ptr1]][1] == y_train[ptr2][0]:
        ptr1 = ptr1+1
        ptr2 = ptr2+1
    else:
        place_dist.append(ptr1)
        ptr1 = ptr1+1
place_dist = np.array(place_dist)
x_out = np.delete(Xtrain, place_dist, 0)
Xtrain = x_out

y = pd.read_csv('yvalidate_combined.csv')
y = y.values
index_list = np.where(y[:,0] == 3000)[0]
l = len(index_list)
N,D = y.shape
y_validate = np.zeros((l, 60))
numitr = Xval.shape[0]
out_index = 0
for i in index_list:
    matrix = y[i-29:i+1,1:]
    matrix = matrix.flatten()
    y_validate[out_index] = matrix
    out_index = out_index+1

x_out2 = np.zeros((l,99))
place_dist = []
ptr1 = 0
ptr2 = 0
n,d = x_out2.shape
org_list = np.where(y[:,0] == 100)[0]
```

```
    while ptr1 < numitr:
        if y[org_list[ptr1]][1] == y_validate[ptr2][0]:
            ptr1 = ptr1+1
            ptr2 = ptr2+1
        else:
            place_dist.append(ptr1)
            ptr1 = ptr1+1
    place_dist = np.array(place_dist)
    x_out2 = np.delete(Xval, place_dist, 0)
    Xval = x_out2

    print(Xtrain.shape, y_train.shape, Xval.shape, y_validate.shape, Xtest.shape)

    hidden_layer_choice = [50, 50]
    iteration_choice = [100, 200, 300, 400, 500]
    lammy_choice = [1,0.1,0.01,0.001,0.0001]

    # lowest_err = np.inf
    # best_param = []
    # for i in iteration_choice:
    #     for l in lammy_choice:
    #         model = NeuralNet(hidden_layer_sizes=hidden_layer_choice, max_iter=i, lammy=l)
    #         model.fit(Xtrain, y_train)
    #         y_pred_val = model.predict(Xval)
    #         val_err = compute_error(y_validate, y_pred_val)
    #         if val_err < lowest_err:
    #             lowest_err = val_err
    #             best_param = [i, l]
    # print(lowest_err, best_param)

    model = NeuralNet(hidden_layer_sizes=[50,50], max_iter=100, lammy=0.01)
    model.fit(Xtrain, y_train)
    y_pred = model.predict(Xtest)
    y_final = y_pred.flatten().T
    pd.DataFrame(y_final).to_csv("./result.csv", header=None, index=None)
```

8 Appendix

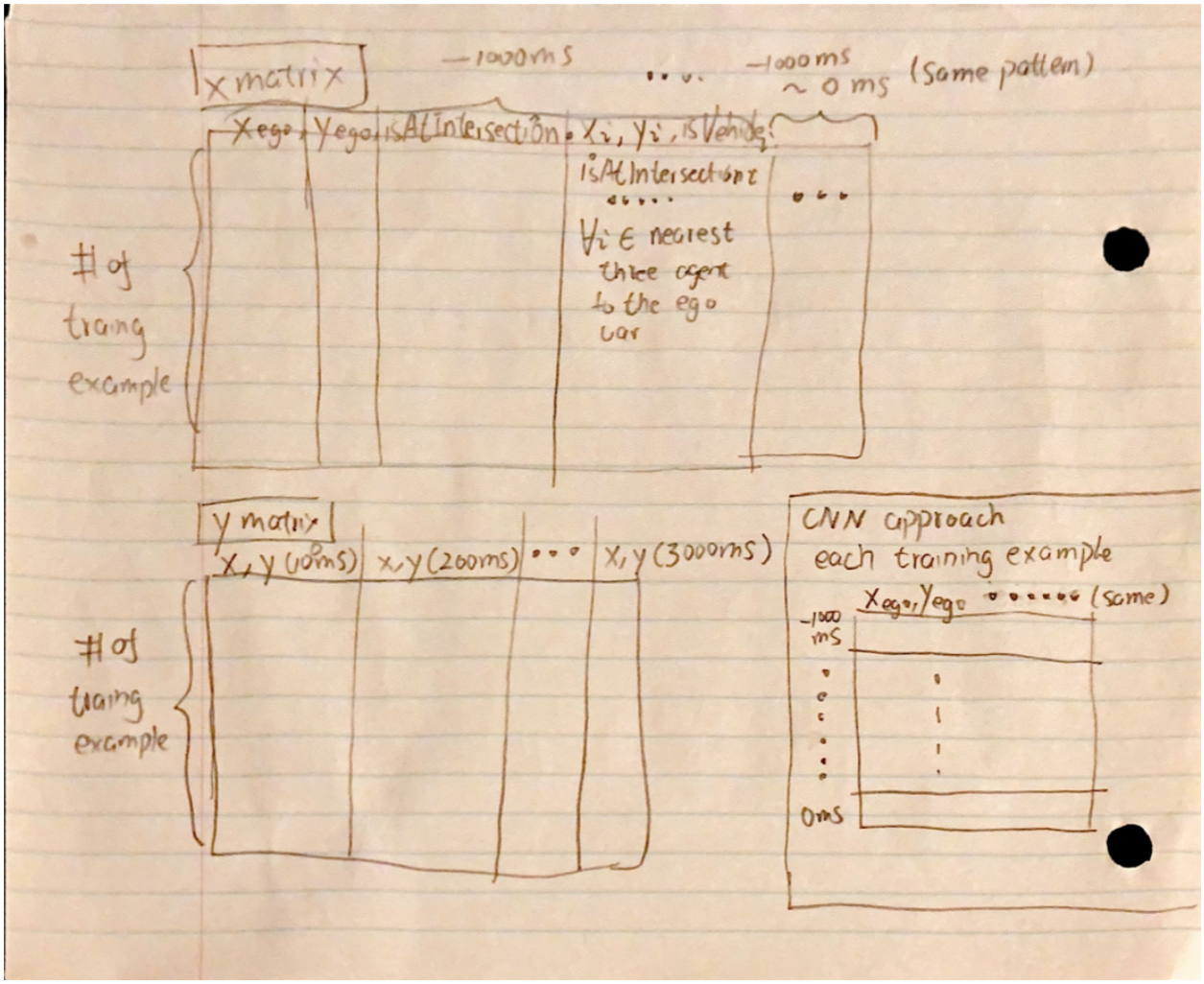

Figure 4: Draft of MLP and CNN model