1 Team

Team Members	*Ali Seyfi (aliseyfi), Milad Jalali(miladj7),Parsa Delavari(p6e7a), Mohammad Taha Askari (d0z5d), Shanny Lu(n7o0d)*
Kaggle Team Name	*The world is not logical*

2 Introduction (3 *points*)

A few sentences describing the autonomous driving prediction problem.

To build a predictive model of vehicle motion based on various features, such as past positions for past ten time steps (1000ms) of both the ego vehicle and closest 10 agents moving about the same intersection, and predict its future position in 3 seconds, $y_i \in \mathbb{R}^{60}$ given X_i.

3 Summary (12 *points*)

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

After formulating the Autonomous driving problem as a supervised learning problem with multiple outputs, with the provided 2307 + 523 pairs dataset, we first considered a simple model for predicting the position of the agent on each time step based on features, and then we considered a chained model to predict all 60 outputs. The core model is a single-layer neural network with "relu" or "sigmoid" activation function. The inputs of the model are trajectories and other features of all cars/pedestrian/bicycles in the intersection for all past time steps including the trajectory of the agent in the recent time steps. In other words, chained model consists of several one-layer neural networks; each of them will predict the position (x-coordinate and y-coordinate) of the agent in one time step. The input of these models is the data about all cars in the first one seconds, and trajectory of the agent in the all past time steps. As a result, the input size of these models increases, as the time steps goes on. The output size of each of these models is two (x-coordinate and y-coordinate).

One drawback of this method is that the error propagates through the model. Since the output of each model serves as input of the next model, if the first model makes some error in predicting the agent's position, next models will also predict wrong based on this wrong input. One solution to this problem is considering several independent neural networks; each for predicting the trajectory of the agent in one time step using the input data of first one second. This approach will eliminate the dependency of the models, and as a result the error in first models doesn't affect the other models.

The approaches mentioned above, predict the x-coordinate and y-coordinate of agent at each time step simultaneously. It means that the model tries to minimize the L2-norm of the error for predicting the position on 2 axis, and not minimizing the error for each axis. As this may be problematic, we also considered the model in which we can predict each coordinate separately. In fact, we implemented the basic neural network in a flexible way, so we can change the number of outputs as a hyper-parameter.

Using these approaches, we need to fit 30 or 60 neural networks which is time-consuming. As a result, we considered an alternative approach of considering only one neural network which predicts all 60 output values once. So, the input of this model will be the trajectories and other related features of all cars in the first one second, and the output of the model will be the trajectory of the agent in the next 3 seconds. We implemented two class of chained neural network (for dependent models approach)and multiple neural network (for independent models approach), and compared the validation error of them after hyper-parameter tuning. More details will be delivered in the sections below.

4 Experiments (15 *points*)

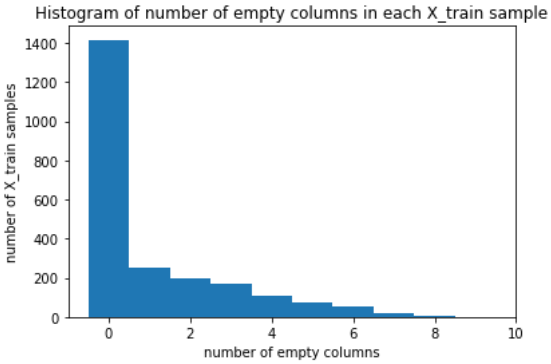
Several paragraphs describing the experiments you ran in the process of developing your Kaggle competition final entry, including how you went about data prepossessing, feature engineering, model, hyper-parameter tuning, evaluation, and so forth.

As observing the original data, we can find out that there exists some redundant data or empty column (all zero values). Thus, it is necessary to conduct data prepossessing and feature selection first. During this progress, we **deleted unwanted columns, converted categorical values to numerical values, adding extra features, filled empty columns with useful information, and merged the whole data together** on the each dataset X_i and y_i.

Specifically, first we moved the 'agent' related columns to the first columns and then dropped 'role' column because it was coded by index. At the same time, we got rid of all categorical values. The 'type' column were transformed into numerical values, with vehicle = 1 and pedestrian = 2. Secondly, we considered adding following **extra features**:

1. **distance** to agent: For every object we computed its distance to the 'agent' as a new feature. The reason is that the decisions are more related to neighboring vehicles than others.
2. **speed**: We computed speed in x and y directions using discrete approximation of derivative of location to calculate overall speed.
3. **direction**: We also added direction as the angle of speed vector. But because direction was discrete at 2π, we decided to use both $\sin(direction)$ and $\cos(direction)$ instead of direction itself. These two features are continuous and have the information of direction.
4. **acceleration**: We added acceleration by computing discrete approximation of derivative of speed. This feature tells us whether a car is braking or increasing its velocity.
5. **turning**: This feature is the derivative of direction, which actually is a continuous version of direction that we needed to obtain first. This feature tells us if a car is turning to left or right and how much.

Thirdly, we noticed that 888 out of 2308 X_train files have at least one empty column, there are approximately 1.1 empty columns per each X_train file, and 436 X_train samples have more than 2 empty column. Therefore, it is important to handle these empty columns. However, it is obvious that we cannot simply delete them because our final version of X_train files should have the same number of features in each row. We came up with an idea to deal with empty column without deleting them. For example, when we have only 6 present cars in one frame we can assume that the other four cars are far away and do not move, thus one rational approach is to give those columns constant value x_s and y_s. Consequently the four extra features speed, direction, acceleration and turning get zero as they do not move, which is a good point. We chose those constant values in the same order of magnitude of max(x) and max(y) to tell the model that they are far from the 'agent'.



As for **model selection**, as mentioned in above session, since this autonomous driving problem is a supervised learning with multiple outputs. We first considered a simple model based on selected features to predict position of agent in next 3 seconds, and then came up with using a chained model to predict all 60 outputs (x and y values in the next 30 timestamps) by fitting 30 or 60 neural networks. Upon consideration of the tradeoff between training time and model accuracy, we picked an alternative approach of implementing one neural network which could predict 60 outputs once. The model input is the trajectories and other selected features of all cars in the most recent one second, and output are the predictive trajectories in three seconds. However, the RMSE of this model is around 1.4, which is not satisfactory. Ultimately, we tested on some other model and observed that the ridge regression led to a surprisingly good result. The most surprising part is that, in this model, we didn't even consider other features except for only agent's position (x, y) in the past. Position x and y in the next 30 timestamp were respectively predicted based on given position (x or y) and new predicted position (x or y). In another word, the output of each model serves as input of the next model in a chained rule.

Due to the fact that our final model is super simple, the various values of **hyperparatmeter** in our final model actually didn't matter too much and were all set to one. But for our previous tested neural network model, we tuned it in a grid search way by finding out an elbow value in the figure of RMSE vs. hyperparameter values.

The **evaluation** approach we applied to pick best model and tuned hyperparameter is to conduct cross validation by separating our merged test data after prepossessing and feature engineering into two parts, training dataset (2307 row data) and validation dataset (523 row data). More specifically, dimension of X_train, y_train, X_val, and y_val are (2307, 990), (2307, 60), (523, 990) and (523, 60) respectively. Afterward, based on these datasets, model selection and hyper-parameter tuning are conducted. Need to mention that we picked the hyper-parameter values without overfitting hopefully. Also, our validation accuracy were measured and compared against the actual positions using Root Mean Square Error (RMSE).

5 Results (5 *points*)

Team Name	Kaggle Score
The world is not logical	**0.40043**

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.

Why did we choose this team name, The world is not logical? Long story short, we tried whatever we thought that may yield to a better validation error. We normalized our features, but the result was not satisfactory; We tried to adding various innovative features, but the results also got worse; we tried creative models, such as chaining neural networks, but the results were still not achievable. At the end, we decided to simplify our methods and features. The result? It was getting perfect! Getting rid of all other features except for only two features, x and y of the agent car, made the results getting better.

During the process of solving this problem, we were astonished but finally accepted the fact that sometimes we just made a simply problem over complication, just like we originally considered complicated chained neural network with fancy features, which led to around 1.4 RSME, while our final model, a simply linear model, chained ridge regression model, improved our error to only 0.4. Also, tackling issues with an open mind is important. If we kept tracked on our fancy features and complicated deep neural network model by simply tuning hyperparameters, the results wouldn't improve such dramatically.

As for how to improve our model in this autonomous driving prediction, since the data we have is limited and, too some extent, not sufficient, we concluded that if larger dataset or more data will be given, then our neural network based models will have better performance, due to the fact that deep neural networks usually require tons of data to build a satisfactory model. We still believe that neural network will beat simple linear model with larger dataset.

7 Code

Include all the code you have written for the autonomous driving prediction problem.
In addition to the bellow codes, there is a main script that runs different models and compute their accuracy.

- Ridge regression Class : A class which implemented for applying ridge regression on the data.

```
class Ridge:

    def init(self, lambda):
        self.lambda = lambda

    def fit(self,X,y):
        n, d = X.shape
        lambda = self.lambda
        a = np.array([[1]]*n)
        X_new=np.append(X, a, axis=1)
        self.w = solve(X_new.T@X_new + lambda*np.eye(d+1), X_new.T@y)

    def predict(self, X):
        n, d = X.shape
        a = np.array([[1]]*n)
        X_new=np.append(X, a, axis=1)
        return X_new@self.w
```

Listing 1: final_model.py - Ridge

- Chained Ridge : This is our final model which give us the best score. It is basically a chain of ridge regression models.

```
def chained_Ridge(X_train, y_train, X_test):
    model_x = Ridge()
    model_x.fit(X_train, y_train[:,0])
    x_hat_train = model_x.predict(X_train)
    x_hat_test = model_x.predict(X_test)

    model_y = Ridge()
    model_y.fit(X_train, y_train[:,1])
    y_hat_train = model_y.predict(X_train)
    y_hat_test = model_y.predict(X_test)

    y_pred_train = np.concatenate((x_hat_train.reshape(-1,1), y_hat_train.reshape(-1,1)
    ),axis=1)
    y_pred_test = np.concatenate((x_hat_test.reshape(-1,1), y_hat_test.reshape(-1,1)),
    axis=1)

    for i in range(2,60):
        XN_train = np.concatenate((X_train,y_pred_train[:,i-2].reshape(-1,1)),axis =1)
        XN_test = np.concatenate((X_test,y_pred_test[:,i-2].reshape(-1,1)), axis = 1)

        model = Ridge()
        model.fit(XN_train, y_train[:,i])
        hat_train = model.predict(XN_train)
        hat_test = model.predict(XN_test)

        y_pred_train = np.concatenate((y_pred_train, hat_train.reshape(-1,1)),axis=1)
        y_pred_test = np.concatenate((y_pred_test, hat_test.reshape(-1,1)),axis=1)

    return y_pred_train, y_pred_test
```

Listing 2: final_model.py - chained_Ridge

- Continuous angle function: As we want to take derivative of angle to compute the turning, we need to make angle continuous.

```
def continuous_angle(x):

    last = 0
    out = []

    for angle in x:
        while angle < last-np.pi: angle += 2*np.pi
        while angle > last+np.pi: angle -= 2*np.pi
        last = angle
        out.append(angle)

    return np.array(out)
```

Listing 3: add_features.py - continuous_angle

- Distance to agent function: The name of function says what it is wrote for.

```
def dist2agent(data):

    for i in range(1,10):

        x = data[' x%d'%i]
        y = data[' y%d'%i]
        xa = data[' x0']
        ya = data[' y0']
        data[' dist%d'%i] = np.sqrt((x-xa)**2+(y-ya)**2)
    return data
```

Listing 4: add_features.py - dist2agent

- Poly 2 function : We compute X^2, y^2, and Xy in this function.

```
def poly2(data):

    for i in range(10):


        x = data[' x%d' % i]
        y = data[' y%d' % i]

        data[' x2%d' % i] = x**2
        data[' y2%d' % i] = y**2
        data[' xy%d' % i] = x*y

    return data
```

Listing 5: add_features.py - poly2

- Speed direction : Computing direction of speed.

```
def speed_direction(data):

    for i in range(10):

        speed = np.zeros(11)
        sin_dir = np.zeros(11)
        cos_dir = np.zeros(11)

        x = data[' x%d' % i]
        y = data[' y%d' % i]

        speed[0] = np.sqrt((x[1]-x[0])**2+(y[1]-y[0])**2)
        direction = np.arctan2(y[1]-y[0],x[1]-x[0])
        sin_dir[0] = np.sin(direction)
        cos_dir[0] = np.cos(direction)

        speed[10] = np.sqrt((x[10]-x[9])**2+(y[10]-y[9])**2)
        direction = np.arctan2(y[10]-y[9],x[10]-x[9])
        sin_dir[10] = np.sin(direction)
        cos_dir[10] = np.cos(direction)

        for t in range(1,10):

            speed[t] = np.sqrt((x[t+1]-x[t-1])**2+(y[t+1]-y[t-1])**2)/2
            direction = np.arctan2(y[t+1]-y[t-1],x[t+1]-x[t-1])
            sin_dir[t] = np.sin(direction)
            cos_dir[t] = np.cos(direction)


        data[' speed%d' % i] = speed
        data[' sin(dir)%d' % i] = sin_dir
        data[' cos(dir)%d' % i] = cos_dir

    return data
```

Listing 6: add_features.py - speed_direction

- Acceleration : Computing acceleration.

```
def acceleration(data):

    for i in range(10):

        a = np.zeros(11)

        speed = data[' speed%d' % i]

        a[0] = speed[1]-speed[0]
        a[10] = speed[10]-speed[9]

        for t in range(1,10):
            a[t] = (speed[t+1]-speed[t-1])/2


        data[' acceleration%d' % i] = a

    return data
```

Listing 7: add_features.py - acceleration

- Turning: Computing derivative of direction of the vehicle

```
def turning(data):

    for i in range(10):

        turn = np.zeros(11)

        sin_dir = data[' sin(dir)%d' % i]
        cos_dir = data[' cos(dir)%d' % i]
        direction = np.arctan2(sin_dir, cos_dir)
        direction = continuous_angle(direction)

        turn[0] = direction[1]-direction[0]
        turn[10] = direction[10]-direction[9]

        for t in range(1,10):
            turn[t] = (direction[t+1]-direction[t-1])/2


        data[' turning%d' % i] = turn

    return data
```

Listing 8: add_features.py - turning

- Replace the agent : Moving agent to the first column and removing role column

```
def replace_agent(data):
    for i in range(10):
        if data[' role%d' % i][0] == ' agent':
            temp = data[[' id0',' role0',' type0',' x0',' y0',' present0']]
            data[[' id0',' role0',' type0',' x0',' y0',' present0']] = data[[' id%d'%i,
    ' role%d'%i,' type%d'%i,' x%d'%i,' y%d'%i,' present%d'%i]]
            data[[' id%d'%i,' role%d'%i,' type%d'%i,' x%d'%i,' y%d'%i,' present%d'%i]]
    = temp


    return data
```

Listing 9: add_features.py - replace_agent

- Empty fixing : fill empty parts of the data.

```
def empty_fix(data, x_max=30, y_max=10):
    xs = np.array([2*x_max,2*x_max,-2*x_max,-2*x_max,3*x_max,3*x_max,-3*x_max,-3*x_max
    ,4*x_max])
    ys = np.array([2*y_max,-2*y_max,2*y_max,-2*y_max,3*y_max,-3*y_max,3*y_max,-3*y_max
    ,-4*y_max])
    j = 0
    for i in range(1,10):
        if data[' present%d'%i][0] == 0:
            data[' x%d'%i] = xs[j]*np.ones(11)
            data[' y%d'%i] = ys[j]*np.ones(11)
            j += 1
            data[' role%d'%i] = ' others'
            data[' type%d'%i] = ' car'
    return data
```

Listing 10: add_features.py - empty_fix

- Create output : reformat the output to fit the Kaggle's format.

```
def create_output(location):

    xs = [str(i) + "_x_" + str(j)  for i in range(20) for j in range(1,31)]
    ys = [str(i) + "_y_" + str(j)  for i in range(20) for j in range(1,31)]

    ids = []
    for i in range(len(xs)):
        ids.append(xs[i])
        ids.append(ys[i])

    data = {'Id': ids,
            'location': location}

    df = pd.DataFrame (data, columns = ['Id','location'])
    df.to_csv('ytest.csv', index=False)
```

Listing 11: utils.py - create_output

- Chained class : This class get a model as input and fit a chain model as described earlier.

```
class chained():

    def __init__(self, model, *args):

        self.model = model
        self.args = args

    def fit(self,X,Y):

        n,d = X.shape
        n,k = Y.shape
        k = int(k/2)
        self.k = k
        print('k=',k)
        x_models = []
        y_models = []
        for i in range(k):
            model = self.model(*self.args)
            model.fit(np.concatenate((X,Y[:,0:2*i]), axis=1),Y[:,2*i])
            x_models.append(model)
            model = self.model(*self.args)
            model.fit(np.concatenate((X,Y[:,0:2*i]), axis=1),Y[:,2*i+1])
            y_models.append(model)
            print(i)

        self.x_models = x_models
        self.y_models = y_models

    def predict(self,X):
        n,d = X.shape
        x_models = self.x_models
        y_models = self.y_models
        k = self.k

        Y = np.zeros((n,2*k))

        for i in range(k):
            Y[:,2*i] = x_models[i].predict(np.concatenate((X,Y[:,0:2*i]), axis=1))
            Y[:,2*i+1] = y_models[i].predict(np.concatenate((X,Y[:,0:2*i]), axis=1))
            print(i)
```

Listing 12: utils.py - chained

- getX and gety functions : in these functions we compute features using add_features file functions and add store them together.

```
def get_Xtrain(mypath):
    X_train=[]

    directory = os.path.join(mypath,'train','X')
    for root,dirs,files in os.walk(directory):
        for file in files:
           if file.endswith(".csv"):
               with open(os.path.join(mypath,'train','X',file), 'rb') as f:
                   data = pd.read_csv(f)
                   data = add_features.replace_agent(data)
                   data = add_features.empty_fix(data)
                   data = preprocess.clean_data_X(data)
                   # data = add_features.dist2agent(data)
                   # data = add_features.poly2(data)
                   data = add_features.speed_direction(data)
                   data = add_features.acceleration(data)
                   data = add_features.turning(data)
                   data = np.array(data)
                   data = data.flatten()
                   X_train.append(data.tolist())
                   print(file)

    X_train = np.array(X_train)

    X_train = np.vectorize(float)(X_train)
    return X_train
```

Listing 13: utils.py - get_Xtrain

- Cleaning X data : In order to clean x data, we just remove time step column and then change alphabetic features to numeric ones. At the end we remove id, present, and role columns.

```
def clean_data_X(data):
    data = data.drop(columns = ['time step'])

    data = data.replace({" car": 1, " pedestrian/bicycle": 2})
    data = data.replace({" agent": 1, " others": 2})

    for i in range(10):
        data = data.drop(columns = [' id' +str(i), ' present'+str(i), ' role'+str(i)])


    return data
```

Listing 14: preprocess.py - clean_data_X

- Cleaning y data : In some samples of y dataset we have lower number of rows than 30. One approach is to delete them but a better way is to predict the missing rows to also make use of those samples. This function finds the number of missing rows and then predicts them.

```
def clean_data_y(data):
    size = 30
    if size != data.shape[0]:
        diff=size-data.shape[0]
        x = data[' x'].values
        y = data[' y'].values
        t = data['time step'].values

        model_x = AutoReg(x, 5)
        model_y = AutoReg(y, 5)
        model_t = AutoReg(t, 5)
        predictions_x = model_x.fit().predict(start=len(x), end=len(x)+diff-1, dynamic=
    False)
        predictions_y = model_y.fit().predict(start=len(y), end=len(y)+diff-1, dynamic=
    False)
        predictions_t = model_t.fit().predict(start=len(t), end=len(t)+diff-1, dynamic=
    False)
        d = np.concatenate((predictions_t.reshape(-1,1),predictions_x.reshape(-1,1),
    predictions_y.reshape(-1,1)),axis=1)
        d = pd.DataFrame(d, columns=['time step',' x', ' y'])
        data = data.append(d)

    return data.drop(columns = ['time step'])
```

Listing 15: preprocess.py - clean_data_y

- Cleaning y : When we plotted the coordinates of the y dataset points, we noticed that the last point of each y sample seem to be outliers. This function deletes those points and predicts the correct values.

```
def clean_y(y_test,y_val):

    n = y_test.shape[0]

    y = np.concatenate((y_test,y_val),axis=0)
    y1 = np.zeros(y.shape)
    index = np.array(range(0,60,2))
    xs = y[:,index]
    ys = y[:,index+1]
    model = MLPRegressor()
    model.fit(xs[:,:28],xs[:,28])
    x_new = model.predict(xs[:,1:29])
    model = MLPRegressor()
    model.fit(ys[:,:28],ys[:,28])
    y_new = model.predict(ys[:,1:29])

    y1[:,:58] = y[:,:58]
    y1[:,58] = x_new
    y1[:,59] = y_new
    return y1[:n], y1[n:]
```

Listing 16: preprocess.py - clean_y

- Neural Net chain : Another implementation of the neural net chain that we talked about earlier.

```
class NeuralNet_Chain():
    def __init__(self, hidden_layer_sizes, classification = True, lammy=1, s=2,
    max_iter=100, verbose=True):
        self.hidden_layer_sizes = hidden_layer_sizes    # number of neurons for each
    model in chain
        self.lammy = lammy  # regularization hyper-param for each model in chain
        self.max_iter = max_iter    # maximum number of iterations
        self.classification = classification    # classification or regression
    indicator
        self.s = s  # chain stride
        self.verbose = verbose

    def fit(self, X, y):

        _, self.k = y.shape # output dimension of chained model

        # repeating the hyper-parameters when are not assigned
        while self.lammy.shape[0] != np.int(self.k/self.s):
            self.lammy = np.append(self.lammy,self.lammy[-1])

        # repeating the hyper-parameters when are not assigned
        while self.max_iter.shape[0] != np.int(self.k / self.s):
            self.max_iter = np.append(self.max_iter, self.max_iter[-1])

        # repeating the hyper-parameters when are not assigned
        while len(self.hidden_layer_sizes) != np.int(self.k/self.s):
            self.hidden_layer_sizes.append(self.hidden_layer_sizes[-1])

        # fitting dependent models repetitively
        models = []
        model = NeuralNet(hidden_layer_sizes=self.hidden_layer_sizes[0],classification=
    self.classification, lammy=self.lammy[0],max_iter=self.max_iter[0], verbose=self.
    verbose)
        model.fit(X,y[:,0:self.s])
        models.append(model)

        for i in range(1,np.int(self.k/self.s)):

            model = NeuralNet(hidden_layer_sizes=self.hidden_layer_sizes[i],
    classification=self.classification,
                              lammy=self.lammy[i], max_iter=self.max_iter[i], verbose=
    self.verbose)
            model.fit(np.concatenate((X,y[:,0:i*self.s]),axis=1), y[:, i*self.s:(i+1)*
    self.s])
            models.append(model)

        self.models = models

    def predict(self, X):

        # going forward through the chain
        y = self.models[0].predict(X)
        for i in range(1,np.int(self.k/self.s)):
            y_new = self.models[i].predict(np.concatenate((X,y),axis=1))
            y = np.concatenate((y,y_new),axis=1)

        return y
```

Listing 17: neural_net.py - NeuralNet_Chain

- Neural Net Multiple : implementation of the neural net but this time independent layers that we talked about earlier.

```
class NeuralNet_Multiple():
    def __init__(self, hidden_layer_sizes, classification = True, lammy=1, s=2,
    max_iter=100, verbose=True):
        self.hidden_layer_sizes = hidden_layer_sizes    # number of neurons for each
    model
        self.lammy = lammy  # regularization hyper-param for each model
        self.max_iter = max_iter    # maximum number of iterations
        self.classification = classification    # classification or regression
    indicator
        self.s = s  # chain stride
        self.verbose = verbose

    def fit(self, X, y):

        _, self.k = y.shape # output dimension of multiple model

        # repeating the hyper-parameters when are not assigned
        while self.lammy.shape[0] != np.int(self.k/self.s):
            self.lammy = np.append(self.lammy,self.lammy[-1])

        # repeating the hyper-parameters when are not assigned
        while self.max_iter.shape[0] != np.int(self.k / self.s):
            self.max_iter = np.append(self.max_iter, self.max_iter[-1])

        # repeating the hyper-parameters when are not assigned
        while len(self.hidden_layer_sizes) != np.int(self.k/self.s):
            self.hidden_layer_sizes.append(self.hidden_layer_sizes[-1])

        # fitting independent models repetitively
        models = []
        model = NeuralNet(hidden_layer_sizes=self.hidden_layer_sizes[0],classification=
    self.classification, lammy=self.lammy[0],max_iter=self.max_iter[0], verbose=self.
    verbose)
        model.fit(X,y[:,0:self.s])
        models.append(model)

        for i in range(1,np.int(self.k/self.s)):

            model = NeuralNet(hidden_layer_sizes=self.hidden_layer_sizes[i],
    classification=self.classification,
                              lammy=self.lammy[i], max_iter=self.max_iter[i], verbose=
    self.verbose)
            model.fit(X, y[:, i*self.s:(i+1)*self.s])
            models.append(model)

        self.models = models

    def predict(self, X):

        # going forward through several independent models
        y = self.models[0].predict(X)
        for i in range(1,np.int(self.k/self.s)):
            y_new = self.models[i].predict(X)
            y = np.concatenate((y,y_new),axis=1)

        return y
```

Listing 18: neural_net.py - NeuralNet_Multiple