Damascus University

Information Technology Engineering

Homework of

Neural Networks:

Levenberg Marquardt Algorithm

ITE-4th Year

By:

إيمان الغدير

تسنيم عجاج

راما ريحاوي

إيمان البلخي

Supervised by: م. لين قويدر

Levenberg Marquardt Algorithm

خوارزمية تدريب، هدفنا الأساسي في هذه الخوارزمية هو إيجاد النهاية الصغرى لتابع الخطأ بالنسبة للأوزان، أي إيجاد مجموعة الأوزان التي تجعل تابع الخطأ أصغر ما يمكن.

تعتمد على خوارزمية نيوتن الرياضية لإيجاد النهاية الصغرى لتابع أي هي تعبير عن خوارزمية نيوتن أو تطبيق لها في مجال الشبكات العصبونية.

تستخدم لحل المشكلات غير الخطية، تتصف بالسرعة وتقاربها مضمون ومستقر، مفيدة في المشكلات الصغيرة والمتوسطة الحجم.

تدمج بين طريقتين:

- Gradient Descent: تتصف بالاستقرار.
- Gauss newton algorithm: تتصف بالسرعة.

فيما يلي توصيف Neural Network تخضع لخوارزمية تدريب (levenberg):

data.txt: ملف يحوي الـ data التى ستدرب عليها الشبكة

```
data=pd.read_csv("data.txt")
X=data.iloc[:,:-1].values.T
y=data.iloc[:,[2]].T
X=np.array(X)
y=np.array(y)
#np.squeeze(X).shape
(a,b)=X.shape
(a,b)

```

```
(2, 100)
```

split_data(X,y,test_ratio):

- X: الدخل
- Y: الخرج
- Data_spilt_ratio: نسبة التقسيم

الهدف من التابع قسم الـ data إلى قسمين (training, test)

```
def split_data(X,y,data_split_ratio):
    m=X.shape[1]
    arr=np.array(list(range(m)))
    np.random.shuffle(arr)
    train_arr=arr[range(int(np.floor((1-data_split_ratio)*m)))]
    test_arr=arr[range(int(np.floor((1-data_split_ratio)*m)),m)]
    X_train=X[:,train_arr]
    y_train=y[:,train_arr]
    X_test=X[:,test_arr]
    y_test=y[:,test_arr]
    return (X_train,y_train,X_test,y_test)

```

```
split_data(X,y,0.5)
```

Activation Function:

قمنا باستخدام العديد من Activation function لحساب خرج الطبقة حيث يقوم المستخدم بتحديد ال Activation لكل طبقة:

1- Sigmoid function:

$$f(x) = \frac{1}{1+e^{-x}}$$

Derivative:

$$f'(x) = f(x)(1-f(x))$$

```python
def sigmoid(z,derivative=False):
    a=1.0/(1.0+np.exp(-z))
    if(derivative):
        return np.multiply(a,(1-a))
    return a
```

2- Tanh function:

$$f(x) = \frac{2}{1+e^{-2x}} - 1$$

Derivative:

$$f'(x) = 1 - f(x)^2$$

```python
def tanh(z,derivative=False):
    a=(np.exp(z)-np.exp(-z))/(np.exp(z)+np.exp(-z))
    if(derivative):
        return 1-np.power(a,2)
    return a

```

3- Relu function:

$$f(x) = \begin{cases} 0 & for \quad x < 0 \\ x & for \quad x \geq 0 \end{cases}$$

Derivative:

$$f'(x) = \begin{cases} 0 & for \quad x < 0 \\ 1 & for \quad x \geq 0 \end{cases}$$

```python
def relu(z,derivative=False):
    if(derivative):
        return np.array([list(map(lambda x:int(x>=0),item)) for item in z])
    return np.array([list(map(lambda x:max(x,0),item)) for item in z])
```

```python
def leaky_relu(z,derivative=False):
    if(derivative):
        return np.array([list(map(lambda x:0.01 if x<0 else 1,item)) for item in z])
    return np.array([list(map(lambda x:max(x,0.01*x),item)) for item in z])
```

4- Soft max function:

$$\sigma(z)_j = \frac{e^{zj}}{\sum_{k=1}^{K} e^{zk}} \quad for\ j = 1, \ldots, K.$$

```python
def softmax(z):
    A=np.zeros(z.shape)
    m=z.shape[1]
    for i in range(m):
        A[:,i]=np.divide(np.exp(z[:,i]),np.sum(np.exp(z[:,i])))
    return A
```

Activation_Function:

- Z: القيمة المراد تطبيق تابع ال Activation عليها.
- name_activation_function: اسم تابع التنشيط.
- Derivative: هي قيمة بوليانية تأخذ حالتين عندما ياخذ False يرد الActivation Function أما عندما يأخذ القيمة True يرد Derivative Function.

```python
def activation_Function(z,name_activation_function,derivative=False):
    if(name_activation_function=="sigmoid"):
        return sigmoid(z,derivative)
    elif(name_activation_function=="tanh"):
        return tanh(z,derivative)
    elif(name_activation_function=="relu"):
        return relu(z,derivative)
    elif(name_activation_function=="leaky_relu"):
        return leaky_relu(z,derivative)
    elif(name_activation_function=="softmax"):
        return softmax(z)
    else:
        return z
```

Cost Function:

تم استخدام ثلاث توابع لحساب التكلفة حيث يقوم المستخدم بتحديد تابع التكلفة عند تدريب الشبكة

- MSE (Mean squared error)

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} h_\theta\left(x^{(i)} - y^{(i)}\right)^2$$

```python
def mse(y,A):
    (t,m)=y.shape
    return np.multiply((1.0/m),(np.sum(np.power((y-A),2),1)))
```

- MAE (Mean absolute error)

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_j - \hat{y}_j|$$

```python
def mae(y,A):
    (t,m)=y.shape
    return np.multiply((1.0/m),(np.sum(np.abs((y-A),2))))
```

- Cross entropy

$$J = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - a^{(i)}\right)$$

```python
def cross_entropy_cost(y,A):
    (t,m)=y.shape
    return  np.multiply(-1.0/m,np.sum(np.multiply(y, np.log(A)) +
                                      np.multiply((1 - y), np.log(1 - A))))
```

cost_functions:

- Y: الخرج
- A: الخرج المتوقع عن الشبكة
- cost_function: اسم تابع التكلفة

يتم التعبير عن الخطأ المقابل للاوزان Wight ومدخلات الشبكة X بأحدى التوابع الاتية :

```python
def cost_functions(y,A,name_cost_function):
    if(name_cost_function=="mse"):
        return mse(y,A)
    elif(name_cost_function=="mae"):
        return mae(y,A)
    else:
        return cross_entropy_cost(y,A)
```

Claculate_accuracy:

يستحدم هذا التابع في حساب دقة توقعاتنا بالاعتماد على الخرج المتوقع حيث يتم اختيار proplem_type

Problem type: Regression, binary classification, multi class calassification

```
#accuracy function
#problem_type:(Multi class classification, Regression, binary classification )
```

```
def calculate_accuracy(y,A,problem_type):
    if(problem_type=="Multi_class_classification"):
         A=np.argmax(A,axis=0).reshape(A.shape[1],1).T
         y=np.argmax(y,axis=0).reshape(y.shape[1],1).T
    (t,m)=y.shape
    return np.squeeze(((m-np.sum(np.abs(y-A),1))/m)*100)
```

يستخدم هذا التابع في تحويل مصفوفة ثنائية إلى list

```
def unroll(listt):
    return listt.ravel()
```

بناء صف للشبكة يحتوي على عدة واصفات:

- network_hyperParameters: يحتوي على وصف لجميع hyperParameters التى تؤثر على عملية التعلم حيث يتم ادخالها بشكل يدوي وملاحظة تغيرالمنحي بناءاً على هذه القيم
- network_parameters: يحتوي على وصف ل parameters الشبكة
- training_info: يحتوي على وصف لجميع معلومات التدريب

```
class neural_network:
    def __init__(self):
        self.network_hyperParameters={}
        self.network_parameters={}
        self.trainig_info={}

```

Parameters_network

- X
- Y
- Arr_hiden_neurons: مصفوفة بعدد الطبقات المخفية كل عنصر فيها يمثل عدد النيرونات في طبقة معينة.
- Arr_activation_function: مصفوفة كل عنصر من عناصرها يمثل تابع التنشيط الخاص بطبقة معينة.

```
    def parameters_network(self,X,y,arr_hidden_neurons,arr_activation_functions):
         (n,m)=X.shape
         (o,m)=y.shape
         Wight={}
         Biase={}
         arr_hidden_neurons=[n]+arr_hidden_neurons+[o]
         num_of_layerss=len(arr_hidden_neurons)-1
         for i in range(num_of_layerss):
             wight=np.random.rand(arr_hidden_neurons[i+1],arr_hidden_neurons[i])*0.001
             Wight[i]=wight
             biase=np.zeros((arr_hidden_neurons[i+1],1))
             Biase[i]=biase
         self.network_parameters={"Wight":Wight,"Biase":Biase,"num_of_layers":num_of_layerss,
                                  "arr_activation_functions":arr_activation_functions,"m":m};

```

Forward_propagate:

يقوم بحساب الخرج الخاص بكل طبقة من طبقات الشبكة ويقوم بإرجاع التابع (A,Z) حيث

$$Z_k = X * W_k + b$$

A= Activation _Function(Z)

حيث قيمته بالنسبة لنيرون معين هو الجداء السلمي لخرج نيرونات الطبقة السابقة مضروباً بالأوزان لتلك النيرونات:

```
    def forward_propagate(self,X,y):
        Wight=self.network_parameters["Wight"]
        Biase=self.network_parameters["Biase"]
        num_of_layers=self.network_parameters["num_of_layers"]
        arr_activation_functions=self.network_parameters["arr_activation_functions"]
        #Z=Wight*X+Baise
        #A=activation_Function(Z)
        A={}
        Z={}
        Z[0]=X;
        A[0]=X;
        for i in range(1,num_of_layers+1):
            Z[i]=np.dot(Wight[i-1],A[i-1])+Biase[i-1]
            A[i]=activation_Function(Z[i],arr_activation_functions[i-1])
        return {"A":A,"Z":Z}

```

Deltas_allLayer:

- Dic_forward: هو عبارة عن dictionary يحوي القيم التى قام التابع forward_propagate بأرجاعها
- X
- Y

مهمة التابع حساب الخطأ الذي ترتكبه كل طبقة من الطبقات عدا طبقة الدخل:

$$Delta_k = (A_k - y) * Derivative\ Function_{k-1}(Z_k)$$

```
    def deltas_allLayer(self,dic_forward,X,y):
        Wight=self.network_parameters["Wight"]
        num_of_layers=self.network_parameters["num_of_layers"]
        arr_activation_functions=self.network_parameters["arr_activation_functions"]
        A=dic_forward["A"]
        Z=dic_forward["Z"]
        delta={}
        delta[num_of_layers]=np.multiply(A[num_of_layers]-y,activation_Function(Z[num_of_layers]
                                                                                ,arr_activation_functions[num_of_layers-1],True))
        for i in reversed(range(1,num_of_layers)):
            delta[i]=np.multiply(np.dot(Wight[i].T,delta[i+1]),activation_Function(Z[i]
                                                                                   ,name_activation_functions[i-1],True))
        return delta
```

First_drivative_dWight_and_dBiase:

مهمة التابع إرجاع المشتقات الأولية (dWight,dBiase) ويعتمد هذا التابع في عمله على تابع deltas_allLayer

$$w_{new} = w_{old} + \Delta w$$

$$b_{new} = b_{old} + \Delta b$$

$$\Delta w = -\eta \frac{\partial E}{\partial w} = \frac{-\eta}{N} \sum_{k=1}^{N} (y_k - d_k) \frac{\partial f(z_k)}{\partial z_k} x_k \qquad y_k = f(z_k)$$

$$\Delta b = -\eta \frac{\partial E}{\partial b} = \frac{-\eta}{N} \sum_{k=1}^{N} (y_k - d_k) \frac{\partial f(z_k)}{\partial z_k}$$

- المشتق الأول:

$$g = \left[\frac{\partial E(w)}{\partial w_1} \quad \frac{\partial E(w)}{\partial w_2} \quad \dots \quad \frac{\partial E(w)}{\partial w_n}\right]^T$$

المركّبة الأولى هي المشتق بالنسبة لـ w_1

المركّبة الثانية هي المشتق بالنسبة لـ w_2

المركّبة n هي المشتق بالنسبة لـ w_n

```
    def first_drivative_dWight_and_dBiase(self,dic_forward,X,y):
        m=self.network_parameters["m"]
        num_of_layers=self.network_parameters["num_of_layers"]
        A=dic_forward["A"]
        delta=self.deltas_allLayer(dic_forward,X,y)
        dWight={}
        dbiase={}
        for i in reversed(range(num_of_layers)):
            dWight[i]=np.multiply(1.0/m,np.dot(delta[i+1],A[i].T))
            dbiase[i]=1.0/m*np.sum(delta[i+1],1)
            dbiase[i]=dbiase[i].reshape((dbiase[i].shape[0],1))
        return {"dWight":dWight,"dbiase":dbiase}
```

Error_network:

- Wight: العنصر منه يمثل مصفوفة الأوزان بين طبقتين.

- Biase: العنصر منه يمثل مصفوفة الـ biase الخاصة بطبقة معينة.

$$E_k = (y_k - A_k)$$

```
70
71      def Error_network(self,X,y,Wight,Biase):
72          num_of_layers=self.network_parameters["num_of_layers"]
73          arr_activation_functions=self.network_parameters["arr_activation_functions"]
74          A_prev=X
75          A=None
76          Z=None
77          for i in range(1,num_of_layers+1):
78              Z=np.dot(Wight[i-1],A_prev)+Biase[i-1]
79              A=activation_Function(Z,arr_activation_functions[i-1])
80              A_prev=np.double(A)
81          return y-A
82
```

Wight_jacobian:

يقوم بحساب dWight الخاص بطبقة معينة ولكن قبل حسابه يقوم بحساب مصفوفة الjacobian

$$J = \begin{bmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \cdots & \frac{\partial e_1}{\partial w_n} \\ \frac{\partial e_2}{\partial w_1} & \frac{\partial e_2}{\partial w_2} & \cdots & \cdots & \frac{\partial e_2}{\partial w_n} \\ & & \vdots & & \\ \frac{\partial e_P}{\partial w_1} & \frac{\partial e_P}{\partial w_2} & \cdots & \cdots & \frac{\partial e_P}{\partial w_n} \end{bmatrix}$$

(e): يمثل الخطأ المرتكب عن الشبكة تم حسابه في تابع ال Error network.

(J): مصفوفة تمثل المشتقات الأولى gradients لكل مركبة من الشعاع e بالنسبة لجميع الأوزان حيث كل سطر منها يحوي مشتقات إحدى مركبات الشعاع e بالنسبة لجميع الأوزان.

ثم يقوم بضربها بمنقولها ثم يجمع الناتج للمصفوفة القطرية مضروبة بعدد صغير جداً M وفقاً للقانون:

$$H \approx J^T.J + \mu.I$$

والذي سنقوم بتعديل قيمته في كل تكرار حسب قيمة الخطأ الذي ترتكبه الشبكة، فعند الاقتراب من النهاية الصغرى نقوم بجعل القيمة أكبر وعندها يبدأ أثر مصفوفة الjacobian بالتلاشي وتصبح الخوارزمية أقرب لخوارزمية الgradiant descent في عملها، أما عندما نكون بعيدين عن النهاية الصغرى ومن أجل أن نتحرك بأسرع ما يمكن باتجاهها نقوم بجعل قيمة M متناهية في الصغر لجعل الخوارزمية أقرب لطريقة نيوتن في عملها، وبالنسبة لمشتق التابع e بالنسبة لكل وزن في كل طبقة تم حساب مشتق عددي تقريبي.

```python
    def Wight_jacobian(self,X,y,current_layer,first_derivative_dwight):
        Wight=copy.deepcopy(self.network_parameters["Wight"])
        epsilon=self.network_hyperParameters["epsilon"]
        M=self.network_hyperParameters["M"]
        m=self.network_parameters["m"]

        cur_Wight=Wight[current_layer]
        P_W=cur_Wight.shape[0]*cur_Wight.shape[1]
        J_W=np.zeros((m,P_W))

        for j in range(P_W):
            cur_Wight=unroll(cur_Wight)
            cur_Wight[j]=cur_Wight[j]+epsilon
            cur_Wight=cur_Wight.reshape(self.network_parameters["Wight"][current_layer].shape)

            A_pos=self.Error_network(X,y,Wight,self.network_parameters["Biase"])
            cur_Wight=unroll(cur_Wight)
            cur_Wight[j]=cur_Wight[j]-2*epsilon
            cur_Wight=cur_Wight.reshape(self.network_parameters["Wight"][current_layer].shape)

            A_neg=self.Error_network(X,y,Wight,self.network_parameters["Biase"])
            cur_Wight=unroll(cur_Wight)
            cur_Wight[j]=cur_Wight[j]+epsilon

            deltaA=((A_pos-A_neg).T).reshape((A_pos.shape[0],A_pos.shape[1]))
            deltaA=np.sum(deltaA,0)
            J_W[:,j]=deltaA/(2*epsilon)

        I=np.identity(J_W.shape[1])
        cur_Wight=cur_Wight.reshape(self.network_parameters["Wight"][current_layer].shape)
        dWight=np.dot(np.linalg.inv(np.dot(J_W.T,J_W)+M*I),first_derivative_dwight.reshape((P_W,1)))
        dWight=np.sum(dWight,1)
        dWight.reshape((Wight[current_layer].shape))
        return dWight
```

bias_jacopian:

نفس آلية عمل التابع السابق ولكن يتم من خلاله حساب dB الخاصة بطبقة معينة.

```
    def bias_jacopian(self,X,y,current_layer,first_derivative_dbiase):
        Biase=copy.deepcopy(self.network_parameters["Biase"])
        epsilon=self.network_hyperParameters["epsilon"]
        M=self.network_hyperParameters["M"]
        m=self.network_parameters["m"]

        cur_Biase=Biase[current_layer]
        P_b=cur_Biase.shape[0]*cur_Biase.shape[1]
        J_b=np.zeros((m,P_b))

        for j in range(P_b):
            cur_Biase=unroll(cur_Biase)
            cur_Biase[j]=cur_Biase[j]+epsilon
            cur_Biase=cur_Biase.reshape(self.network_parameters["Biase"][current_layer].shape)

            A_pos=self.Error_network(X,y,self.network_parameters["Wight"],Biase)
            cur_Biase=unroll(cur_Biase)
            cur_Biase[j]=cur_Biase[j]-2*epsilon
            cur_Biase=cur_Biase.reshape(self.network_parameters["Biase"][current_layer].shape)

            A_neg=self.Error_network(X,y,self.network_parameters["Wight"],Biase)
            cur_Biase=unroll(cur_Biase)
            cur_Biase[j]=cur_Biase[j]+epsilon

            deltaA=((A_pos-A_neg).T).reshape((A_pos.shape[0],A_pos.shape[1]))
            deltaA=np.sum(deltaA,0)
            J_b[:,j]=deltaA/(2*epsilon)

        I=np.identity(J_b.shape[1])
        cur_Biase=cur_Biase.reshape(self.network_parameters["Biase"][current_layer].shape)
        dbiase=np.dot(np.linalg.inv(np.dot(J_b.T,J_b)+M*I),first_derivative_dbiase)
        dbiase=np.sum(dbiase,1)
        dbiase.reshape((Biase[current_layer].shape))
        return dbiase
```

Set_training_info:

- Training_algorithm: خوارزمية التعلم
- Epochs: عدد عصور التدريب
- problem_type: Regression, classification, multi class classification
- cost_function: اسم تابع الكلفة الذي اختاره المستخدم

يحدد من خلاله المستخدم بعض التفصيل المتعلقة بعملية التدريب قبل بدئها.

```
155
156     def set_training_info(self,training_algorithm,epochs,problem_type,cost_function):
157         if(not "batch_size" in self.trainig_info):
158             self.trainig_info["batch_size"]=0
159         self.trainig_info={"training_algorithm":training_algorithm,
160                            "epochs":epochs,
161                            "problem_type":problem_type,
162                            "cost_function":cost_function,
163                            "batch_size":self.trainig_info["batch_size"]}
164
165
```

Set_batch_size:

- Batch_type: full, mini, stochastic.
- Batch_size: يقوم فيه المستخدم بتحديد نوع الbatch وعدد عناصر الbatch الواحد قبل البدئ بعملية التدريب.

```
165
166     def set_batch_size(self,batch_type,batch_size=None):
167         if(batch_type=="full"):
168             batch_size=self.parameters["m"]
169         elif(batch_type=="stochastic"):
170             batch_size=1
171         self.trainig_info["batch_size"]=batch_size
172
173
174
```

Set_hyperparameters:

يأخذ التابع كوسيط dictionary يحوي الhyperparameters والذي يحددهم المستخدم حسب نوع خوارزمية التعلم.

```
174
175     def set_hyperparameters(self,network_hyperParameters):
176         self.network_hyperParameters=network_hyperParameters
177
```

Divide_batches:

يقوم بتقسيم البيانات إلى batches قبل البدئ بعملية التدريب.

```
178
179     def divide_batches(self,X,y,batch_size):
180         training_set_X={}
181         training_set_y={}
182         m=X.shape[1]
183         start=0
184         batch_count=int(np.ceil(m/batch_size))
185         for i in range(batch_count):
186             end=int(min(m,start+batch_size))
187             training_set_X[i]=X[:,range(start,end)]
188             training_set_y[i]=y[:,range(start,end)]
189             start=int(start+batch_size)
190         return training_set_X,training_set_y
191
192
```

Shuffle_batches:

يقوم بتغير ترتيب الbatches قبل كل عصر تدريب.

```
192
193     def shuffle_batches(self,training_set_X,trainig_set_y):
194         batch_count=len(training_set_X)
195         indecies=np.array(list(range(batch_count)))
196         np.random.shuffle(indecies)
197
198         trainig_set_X_shuffled=[(training_set_X[index]) for index in indecies]
199         trainig_set_y_shuffled=[(trainig_set_y[index]) for index in indecies]
200
201         return trainig_set_X_shuffled,trainig_set_y_shuffled
202
203
```

Levenberg:

يقوم باستدعاء عدة توابع
(forward_parameters,first_drivative_dWight_and_dBiase,Wight_jacobian,Biase_jacopian)

وحساب المشتق من المرتبة الثانية وفقا للقانون:

$$[\nabla^2 E(w)]_{k,j} = \frac{\partial^2 E(w)}{\partial w_k \partial w_j} = \frac{\partial}{\partial w_k}\frac{\partial E(w)}{\partial w_j}$$
$$= \frac{\partial}{\partial w_k}\left(\sum_{p=1}^{P} e_p . \frac{\partial e_p}{\partial w_j}\right)$$

أي ما قمنا به هو أخذ ال gradient، واشتقاقه مرة أخرى بالاعتماد على قاعدة مشتق جداء تابعين.

- العلاقة السابقة من أجل عنصر من عناصر $\nabla^2 E(w)$ وهو العنصر $[\nabla^2 E(w)]_{k,j}$، ومن أجل جميع العناصر تصبح العلاقة:

$$\nabla^2 E(w) = J^T . J + S$$

حيث أن الحد الثاني $\sum_{p=1}^{P} e_p . \frac{\partial^2 e_p}{\partial w_k . \partial w_j}$ من أجل جميع العناصر يصبح $\sum_{p=1}^{P}\left(e_p \nabla^2 e_p\right)$، وسنسميه S:

$$S = \sum_{p=1}^{P}\left(e_p . \nabla^2 e_p\right)$$

وأما الحد الأول $\sum_{p=1}^{P}(\frac{\partial e_p}{\partial w_k} . \frac{\partial e_p}{\partial w_j})$ فيمكن البرهان رياضياً أنه من أجل جميع العناصر يساوي $J^T J$.

وبالتالي أصبحت العلاقة من أجل جميع العناصر كما ذكرنا:

$$\nabla^2 E(w) = J^T . J + S$$

```
205
206  def levenberg(self,X,y):
207      num_of_layers=self.network_parameters["num_of_layers"]
208      dic_forward=self.forward_propagate(X,y)
209      gradient=self.first_drivative_dWight_and_dBiase(dic_forward,X,y)
210
211      first_derivative_dwight=gradient["dWight"]
212      first_derivative_dbiase=gradient["dbiase"]
213
214      dBiase={}
215      dWight={}
216      for i in range(num_of_layers):
217          dWight[i]=self.Wight_jacobian(X,y,i,first_derivative_dwight[i]).reshape(self.network_parameters["Wight"][i].shape)
218          dBiase[i]=self.bias_jacopian(X,y,i,first_derivative_dbiase[i]).reshape(self.network_parameters["Biase"][i].shape)
219      return {"dWight":dWight,"dBiase":dBiase}
220
221
```

Levenberg_model:

يقوم باستدعاء التابع levenberg في كل عصر تدريب من أجل حساب مشتقات كل مصفوفة من مصفوفات الأوزان بين الطبقات ثم تعديل قيم هذه الأوزان.

مناقشة اختيار قيمة μ:

1. ففي حال اخترنا μ كبيرة جداً نكون قد اقتربنا من gradient descent.
يكون الحد $J_k^T J_k$ صغير نسبياً أمام الحد $\mu_k I$ ومنه:

$$w(k+1) \cong w(k) - [\mu_k I]^{-1}.J_k^T e_k = w(k) - \frac{1}{\mu_k} J_k^T e_k$$

$$g_k = J_k^T e_k \; ; \alpha_k = \frac{1}{\mu_k}$$

حيث:

g_k: هي المشتق الأول لتابع الخطأ بالنسبة للأوزان.

α_k: معدل التعلم.

ومنه:

$$w(k+1) = w(k) - \alpha_k . g_k$$

وهي تمثل خوارزمية الـ gradient descent.

2. وفي حال اخترنا μ صغيرة جداً نكون قد اقتربنا من newton method.

$$w(k+1) = w(k) - H_k^{-1} g_k$$

```
222
223   def levenberg_model(self,X,y):
224       #info
225       epochs=self.trainig_info["epochs"]
226       num_of_layers=self.network_parameters["num_of_layers"]
227       learning_rate=self.network_hyperParameters["learning_rate"]
228       M=self.network_hyperParameters["M"]
229       Beta=self.network_hyperParameters["Beta"]
230       cost=np.zeros(epochs)
231       Wight=self.network_parameters["Wight"]
232       Biase=self.network_parameters["Biase"]
233       m=self.network_parameters["m"]
234       cost_function=self.trainig_info["cost_function"]
235       problem_type=self.trainig_info["problem_type"]
236
237       for i in range(epochs):
238           A=self.forward_propagate(X,y)["A"]
239           y_pred=A[num_of_layers]
240           cost[i]=cost_functions(y,y_pred,cost_function)
241           gradient=self.levenberg(X,y)
242
243           for i in range(num_of_layers):
244               Wight[i]=Wight[i]-learning_rate*gradient["dWight"][i]
245               Biase[i]=Biase[i]-learning_rate*gradient["dBiase"][i]
246           self.network_parameters["Wight"]=Wight
247           self.network_parameters["Biase"]=Biase
248
249           if(cost[i]>=cost[i-1] and M<1):
250               M=M/Beta
251           elif(cost[i]<cost[i-1] and M>1e-9):
252               M=M*Beta
253       if(problem_type!="Regression"):
254             for k in range(m):
255               if(y_pred[0][k]>=0.6):
256                   y_pred[0][k]=1
257               else:
258                   y_pred[0][k]=0
259             acc=calculate_accuracy(y,y_pred,problem_type)
260             print(" calculate_accuracy :%.2f"%acc)
261       else:
262           print(cost[epochs-1])
263       plt.plot(range(epochs),cost)
264
```

Train:

يتم من خلاله تحديد التفاصيل المتعلقة بعملية التدريب كما ذكرنا سابقاً، ثم يقوم باستدعاء التابع الذي يمثل خوارزمية التعلم التي اختارها المستخدم ثم تدريب الشبكة بالاعتماد عليها.

```
266
267   def train(self,X,y,trainig_algorithm,problem_type,epochs,cost_function):
268       self.set_training_info(trainig_algorithm,epochs,problem_type,cost_function)
269       if(trainig_algorithm=="levenberg"):
270           self.levenberg_model(X,y)
271
```

قبل تدريب الشبكة الخاصة بنا على training set معين نقوم أولاً بتعريف object من النوع Network سيعبر عن شبكتنا بالشكل.

ثم نقوم بتحديد شكل الشبكة (كعدد طبقاتها وعدد نيورونات كل طبقة) إضافة إلى توابع التفعيل المستعملة في كل طبقة ومنح قيم ابتدائية لمصفوفات الأوزان بين الطبقات، نقوم باستعمال التابع network_parameters للقيام بذلك (سبب قيامنا بتمرير X,y كوسطاء هو حاجتنا لمعرفة أبعاد المصفوفة X لتحديد أبعاد مصفوفة الأوزان بين طبقة الدخل والطبقة المخفية الأولى وأيضاً حاجتنا للمصفوفة y لتحديد أبعاد مصفوفة الأوزان بين الطبقة المخفية الأخيرة وطبقة الخرج) ويتم استعمال الmethod بالشكل التالي:

```
Network.network_parameters(X,y,[],["sigmoid"])
```

حيث نحدد هنا أننا نريد شبكة لا تحتوي على أي طبقة مخفية وفي طبقة الخرج تابع التفعيل هو sigmoid.

في الخطوة التالية نقوم بتحديد نوع الpatch لو أردنا ذلك:

Full patch: يكون لدينا batch وحيد وهو الtraining set كاملاً.

```
network=neural_network()
network.parameters_network(X,y,[],["sigmoid"])
dic_forward=net.forward_propagate(X,y)

network.set_hyperparameters({"learning_rate":0.05})
network.set_batch_size("full")
print(net.trainig_info)

```

في الخطوة التالية نقوم بتحديد قيم الhyper parameters ...

في حالة الlevenberg نقوم بتحديد الlearning rate إضافة للمتحول epsilon الذي نعتمد على قيمته في تقريب المشتق، والمتحول M الذي نقوم بضربه بالمصفوفة القطرية قبل جمعها لناتج ضرب مصفوفة الJacopian بمنقولها (لجعل المصفوفة الناتجة قابلة للقلب دوماً) والمتحول Beta سنستعمله لتكبير أو تصغير القيمة M وبالتالي جعل طريقة levenberg تعمل بطريقة أقرب لخوارزمية الgradiant descent أو newton method ويتم تعديل القيم اعتماداً على قربنا أو بعدنا من النهاية الصغرى

والآن أصحبنا قادرين على تدريب الشبكة على بيانات معينة وذلك عن طريق التابع train والذي نحدد له التالي:

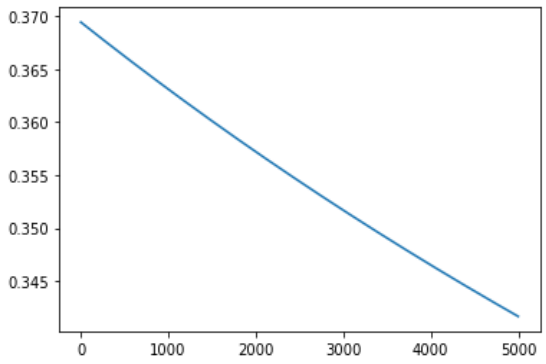
1. Input
2. Expected output
3. Training algorithm
4. Problem type: Regression, binary classification, multi class calassification

5. Epochs

6. Cost function

```
network.set_hyperparameters({"learning_rate":0.17,"epsilon":0.1,"M":0.00001,"Beta":0.0001})
network.train(X,y,"levenberg","Classfication",5000,"cross_entropy_cost")
```

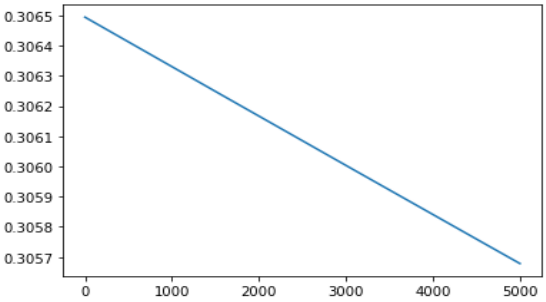
```
calculate_accuracy :87.00
```



حصلنا على دقة 87%

```
network.set_hyperparameters({"learning_rate":0.01,"epsilon":0.1,"M":0.00001,"Beta":0.0001})
network.train(X,y,"levenberg","Multi_class_classification",5000,"cross_entropy_cost")
```

```
calculate_accuracy :100.00
```



حصلنا على دقة : 100%

```
network.set_hyperparameters({"learning_rate":0.1,"epsilon":0.1,"M":0.00001,"Beta":0.0001})
network.train(X,y,"levenberg","Regression",5000,"cross_entropy_cost")
```

```
0.36766042588702724
```