DESIGNING ARTIFICIAL NEURAL NETWORKS FOR CLASSIFICATION OF LBW CASES FROM SCRATCH

PROBLEM STATEMENT:

Low Birth weight (LBW) acts as an indicator of sickness in newborn babies. LBW is closely associated with infant mortality as well as various health outcomes later in life. Various studies show strong correlation between maternal health during pregnancy and the child's birth weight.
We use health indicators of pregnant women such as age, height, weight, community etc., in order for early detection of potential LBW cases. This detection is treated as a classification problem between LBW and not-LBW classes. You have been provided with a Dataset consisting of data collected from a hospital which classifies the patient as cases of LBW and cases of non-LBW. This is a design assignment that requires you to design a neural network from scratch using only numpy.

OBJECTIVES:

Designing an Artificial Neural Network for classification of LBW Cases.

DATA PREPROCESSING:

1) Handling NA and missing values:

The NA values are replaced with the mean(if the column is a number value) or mode(if the column is a binary value) of all the values of the respective column.

2) Feature Scaling:

Feature scaling is a method used to normalize the range of independent variables or features of data. In this step data is normalized.

TRAINING DATA SET AND TEST DATA SET

The dataset is split into training and test set in the ratio of 8:2, i.e.,
80% of data is used as training data and remaining 20% is used as the test data.
For the splitting scikit-learn package is used.

DESIGN:

It is a 2 layer neural network

Description of the various layers

Input layer : 9 inputs
Hidden layer : 6 nodes
Output layer : 1 node

If output node > 0.5, then y = 1 otherwise 0.
Activation function used:
1) sigmoid for the last layer
2) relu for the remaining layers

Dimensions of various hyperparameters

```
# X - 9 x 77

# W1 - 6 x 9
# b1 - 6 x 1

# Z1 = W1 @ X + b1
# Z1 - 6 x 77
# A1 - 6 x 77

# W2 - 1 x 6
# b2 - 1 x 1
# Z2 = W2 @ A1 + b2
# Z2 - 1 x 77
# Y2 - 1 x 77
```

CODE:

```
'''
Design of a Neural Network from scratch

*************<IMP>*************
Mention hyperparams used and describe functionality in detail in this space
- carries 1 mark

Hyperparameters

It is a 2 layer neural network
Input layer : 9 inputs
Hidden layer : 6 nodes
Output layer : 1 node

If output node > 0.5, then y = 1 otherwise 0.
Activation function used:
1) sigmoid for the last layer
2) relu for the remaining layers

Dimensions of various hyperparams
	# X - 9 x 77

	# W1 - 6 x 9
	# b1 - 6 x 1

	# Z1 = W1 @ X + b1
	# Z1 - 6 x 77
	# A1 - 6 x 77

	# W2 - 1 x 6
	# b2 - 1 x 1

	# Z2 = W2 @ A1 + b2
	# Z2 - 1 x 77
	# Y2 - 1 x 77
'''
import numpy as np
import pandas as pd

class NN:
   #Sigmoid function
   def sigmoid(self,z):
      '''
```

```
        Initialize the weights from a random normal distribution
        '''
        a=1/(1+np.exp(-z))
        cache=z
        return a,cache

    #relu function
    def relu(self,z):
        '''
        The ReLufunction performs a threshold
        operation to each input element where values less
        than zero are set to zero.
        '''
        a=np.maximum(0,z)
        cache=z
        return a,cache

    #fit function
    def fit(self):
        '''
        Function that trains the neural network by taking x_train and y_train samples as input
        '''
        #Importing dataset
        dataset = pd.read_csv('LBW_Dataset.csv')

        #Handling missing values and NA values
        dataset['Age']=dataset['Age'].fillna(round(dataset['Age'].mean()))
        dataset['Weight']=dataset['Weight'].fillna(dataset['Weight'].mean())
        dataset['Education']=dataset['Education'].fillna(dataset['Education'].mode()[0])
        dataset['Delivery phase']=dataset['Delivery phase'].fillna(dataset['Delivery phase'].mode()[0])
        dataset['HB']=dataset['HB'].fillna(dataset['HB'].mean())
        dataset['BP']=dataset['BP'].fillna(dataset['BP'].mode()[0])
        dataset['Residence']=dataset['Residence'].fillna(dataset['Residence'].mode()[0])
        X = dataset.iloc[:, :-1].values
        y = dataset.iloc[:, -1].values

        #Splitting the dataset into Training and Test set
        from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 1)

        # Feature Scaling
        from sklearn.preprocessing import StandardScaler
        sc = StandardScaler()
        X_train = sc.fit_transform(X_train)
        X_test = sc.transform(X_test)
```

```
def relu_backward(da,cache):
    z=cache
    dz=np.array(da,copy=True)
    dz[z<=0]=0
    assert(dz.shape==z.shape)
    return dz

def sigmoid_backward(da,cache):
    z=cache
    s,_=self.sigmoid(z)
    dz=da*s*(1-s)
    assert(dz.shape==z.shape)
    return dz

#Initializing the params
def intialize_params_deep(layer_dims):
    params={}
    l=len(layer_dims)
    for i in range(1,l):
        params['w'+str(i)]=np.random.randn(layer_dims[i],layer_dims[i-    1])/np.sqrt(layer_dims[i-1])
        params['b'+str(i)]=np.zeros((layer_dims[i],1))
    return params

#Functions for forward propagation
def linear_forward(a,w,b):
    z=np.dot(w,a)+b
    cache=(a,w,b)
    assert(z.shape==(w.shape[0],a.shape[1]))
    return z,cache

def linear_activation_forward(a_prev,w,b,activation):
    if(activation=="sigmoid"):
        z,l_cache=linear_forward(a_prev,w,b)
        a,activation_cache=self.sigmoid(z)
    elif(activation=="relu"):
        z,l_cache=linear_forward(a_prev,w,b)
        a,activation_cache=self.relu(z)
    cache=l_cache,activation_cache
    return a,cache

def l_model_forward(x,params):
    caches=[]
    a=x
    l=len(params)//2
    for i in range(1,l):
        a_prev=a
```

```
a,cache=linear_activation_forward(a_prev,params['w'+str(i)],params['b'+str(i)],activation='relu')
            caches.append(cache)
         al,cache=linear_activation_forward(a,params['w'+str(l)],params['b'+str(l)],activation='sigmoid')
         caches.append(cache)
         return al,caches

      def compute_cost(al,y):
         m=y.shape[0]
         Totalcost=(-1/m)*np.sum(np.multiply(y,np.log(al))+np.multiply(1-y,np.log(1-al)))
         return Totalcost

      #Functions for backpropagation
      def linear_backward(dz,cache):
         a_prev,w,b=cache
         m=a_prev.shape[1]
         dw=1/m*np.dot(dz,a_prev.T)
         db=1/m*np.sum(dz,axis=1,keepdims=True)
         da_prev=np.dot(w.T,dz)
         return da_prev,dw,db

      def linear_activation_backward(da,cache,activation):
         l_cache,activation_cache=cache
         if(activation=="relu"):
            dz=relu_backward(da,activation_cache)
            da_prev,dw,db=linear_backward(dz,l_cache)
         elif(activation=="sigmoid"):
            dz=sigmoid_backward(da,activation_cache)
            da_prev,dw,db=linear_backward(dz,l_cache)
         return da_prev,dw,db

      def l_model_backward(al,y,caches):
         grads={}
         l=len(caches)
         dal=-(np.divide(y,al)-np.divide(1-y,1-al))
         m=len(layer_dims)
         current_cache=caches[m-2]
         grads['da'+str(m-1)],grads['dw'+str(m-1)],grads['db'+str(m-
1)]=linear_activation_backward(dal,current_cache,activation="sigmoid")
         for i in reversed(range(l-1)):
            current_cache=caches[i]

da_prev_temp,dw_temp,db_temp=linear_activation_backward(grads["da"+str(i+2)],current_cache,activ
ation="relu")
            grads['da'+str(i+1)]=da_prev_temp
            grads['dw'+str(i+1)]=dw_temp
```

```
        grads['db'+str(i+1)]=db_temp
    return grads

  #Function to update params
  def update_params(params,grads,learning_rate):
    for i in range(len_update-1):
        params['w'+str(i+1)]=params['w'+str(i+1)]-(learning_rate*grads['dw'+str(i+1)])
        params['b'+str(i+1)]=params['b'+str(i+1)]-(learning_rate*grads['db'+str(i+1)])
    return params
  X_train=np.reshape(X_train,[X_train.shape[1],X_train.shape[0]])
  X_test=np.reshape(X_test,[X_test.shape[1],X_test.shape[0]])

  def l_layer_model(X,Y,layer_dims,learning_rate,num_iterations,print_cost=False):
    print("Training...")
    costs=[]
    params=intialize_params_deep(layer_dims)
    for i in range(0,num_iterations):
        al,caches=l_model_forward(X,params)
        cost=compute_cost(al,Y)
        grads=l_model_backward(al,Y,caches)
        params=update_params(params,grads,learning_rate)
        costs.append(cost)
    return params
  layer_dims=[9,256,512,2048,512,256,1]
  len_update=len(layer_dims)
  params=l_layer_model(X_train,y_train,layer_dims,learning_rate=0.001,num_iterations=1000)
  pred=self.predict(X_test,params)
  self.CM(y_test,pred)

def predict(self,X_test,params):
  '''
  The predict function performs a simple feed forward of weights
                and outputs yhat values
                yhat is a list of the predicted value for df X
  '''

  z1=params['w1'].dot(X_test)+params['b1']
  a1,_=self.relu(z1)
  z2=(a1.T.dot(params['w2'].T)).T+params['b2']
  a2,_=self.relu(z2)
  z3=(a2.T.dot(params['w3'].T)).T+params['b3']
  a3,_=self.relu(z3)
  z4=(a3.T.dot(params['w4'].T)).T+params['b4']
  a4,_=self.relu(z4)
  z5=(a4.T.dot(params['w5'].T)).T+params['b5']
  a5,_=self.relu(z5)
```

```
        z6=(a5.T.dot(params['w6'].T)).T+params['b6']
        a6,_=self.sigmoid(z6)
        return a6[0]

    def CM(self,y_test,y_test_obs):
        for i in range(len(y_test_obs)):
            if(y_test_obs[i]>0.6):
                y_test_obs[i]=1
            else:
                y_test_obs[i]=0
        cm=[[0,0],[0,0]]
        fp=0
        fn=0
        tp=0
        tn=0
        for i in range(len(y_test)):
            if(y_test[i]==1 and y_test_obs[i]==1):
                tp=tp+1
            if(y_test[i]==0 and y_test_obs[i]==0):
                tn=tn+1
            if(y_test[i]==1 and y_test_obs[i]==0):
                fp=fp+1
            if(y_test[i]==0 and y_test_obs[i]==1):
                fn=fn+1
        cm[0][0]=tn
        cm[0][1]=fp
        cm[1][0]=fn
        cm[1][1]=tp
        p= tp/(tp+fp)
        r=tp/(tp+fn)
        f1=(2*p*r)/(p+r)
        print("Confusion Matrix : ")
        print(cm)
        print("\n")
        print(f"Precision : {p}")
        print(f"Recall : {r}")
        print(f"F1 SCORE : {f1}")
res=NN()
res.fit()
```

OUT OF BOX IMPLEMENTATION:

In the implementation of the above model, we have split the training and test data in the ratio of 8:2 respectively, but instead if we split the data in the ratio of 7:3 , then the performance of the model low when compared to the actual model.

It is clearly visible in the outputs below.

Output 1:

```
Command Prompt
C:\Python3\PES2201800618_AmruthaBS>python Neural_Net.py
Training...
Confusion Matrix :
[[0, 6], [3, 20]]


Precision : 0.7692307692307693
Recall : 0.8695652173913043
F1 SCORE : 0.8163265306122449

C:\Python3\PES2201800618_AmruthaBS>_
```

Output 2:

```
C:\Python3\PES2201800618_AmruthaBS>python Neural_Net.py
Training...
Confusion Matrix :
[[0, 5], [3, 21]]


Precision : 0.8076923076923077
Recall : 0.875
F1 SCORE : 0.8400000000000001

C:\Python3\PES2201800618_AmruthaBS>
```

The low performance of the model might be due to insufficient data in the training set, and hence the model is unable to capture the pattern and fit the data efficiently. This can be overcome by increasing the data in the training set.
In this model we use 80% of data as training set and the remaining 20% of the data as test set.
This improve the performance of the model. It can be clearly observed from the pics below.

EXPERIMENTAL RESULTS:

The final results of the model is shown below.
The accuracy of the model is more than 90% for both the outputs

Output 1:
Precision: 0.9444444444444444
Recall: 0.8947368421052632
F1 SCORE: 0.918918918918919

```
C:\Python3\PES2201800618_AmruthaBS>python Neural_Net.py
Training...
Confusion Matrix :
[[0, 1], [2, 17]]


Precision : 0.9444444444444444
Recall : 0.8947368421052632
F1 SCORE : 0.918918918918919

C:\Python3\PES2201800618_AmruthaBS>
```

Output 2:
Precision: 1.0
Recall: 0.9
F1 SCORE: 0.9473684210526316

```
C:\Python3\PES2201800618_AmruthaBS>python Neural_Net.py
Training...
Confusion Matrix :
[[0, 0], [2, 18]]


Precision : 1.0
Recall : 0.9
F1 SCORE : 0.9473684210526316

C:\Python3\PES2201800618_AmruthaBS>
```

CONCLUSION:

Successfully designed and implemented an Artificial Neural Network for classification of LBW Cases.