Assignment 3 - Multi-class Classification and Neural Network

> **FULL MARKS = 10**

In this assignment, you are going to implement your own neural network to do multi-class classification. We use one-vs-all strategy here by training multiple binary classifiers (one for each class).

Please notice that you can only use numpy and scipy.optimize.minimize. **No** library versions of other method are allowed. . Follow the instructions, you will need to fill the blanks to make it functional. The process is similar to the previous assignment.

In [1]:
```
from sklearn import datasets
from scipy.optimize import minimize
import numpy as np
```

In [2]:
```
def load_dataset():
    iris = datasets.load_iris()
    X = iris.data
    y = iris.target

    return X, y
```

In [3]:
```
def train_test_split(X, y):
    idx = np.arange(len(X))
    train_size = int(len(X) * 2/3)
    np.random.shuffle(idx)
    X = X[idx]
    y = y[idx]
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]

    return X_train, X_test, y_train, y_test
```

In [4]:
```
def init_weights(num_in, num_out):
    '''
    :param num_in: the number of input units in the weight array
    :param num_out: the number of output units in the weight array
    '''

    # Note that 'W' contains both weights and bias, you can consider W[0, :] as bias
    W = np.zeros((1 + num_in, num_out))

    ####################################################################################
    # Full Mark: 1                                                                     #
    # TODO:                                                                            #
    # Implement Xavier/Glorot uniform initialization                                   #
    #                                                                                  #
    # Hint: you can find the reference here:                                           #
    # https://www.tensorflow.org/api_docs/python/tf/keras/initializers/GlorotUniform  #
    ####################################################################################

    x_uniform = np.sqrt(6.0 / (num_in + num_out))
    W = np.random.uniform(-x_uniform,x_uniform,W.shape)

    ####################################################################################
    #                            END OF YOUR CODE                                      #
    ####################################################################################

    return W
```

In [5]:
```
def sigmoid(x):
    '''
    :param x: input
    '''

    ####################################################################################
    # Full Mark: 0.5                                                                   #
    # TODO:                                                                            #
    # Implement sigmoid function:                                                      #
    #                               sigmoid(x) = 1/(1+e^(-x))                          #
    ####################################################################################

    res = 1 / ( 1 + np.exp(-x))

    ####################################################################################
    #                            END OF YOUR CODE                                      #
    ####################################################################################

    return res
```

In [6]:
```
def tanh(x):
    '''
```

```python
    :param x: input
    '''

    ###########################################################################
    # Full Mark: 0.5                                                          #
    # TODO:                                                                   #
    # Implement tanh function:                                                #
    #                     tanh(x) = (e^x-e^(-x)) / (e^x+e^(-x))               #
    ###########################################################################

    res = np.tanh(x)

    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################

    return res
```

In [7]:

```python
def sigmoid_gradient(x):
    '''
    :param x: input
    '''

    ###########################################################################
    # Full Mark: 1                                                            #
    # TODO:                                                                   #
    # Computes the gradient of the sigmoid function evaluated at x.           #
    #                                                                         #
    ###########################################################################

    grad = sigmoid(x)*(1-sigmoid(x))

    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################

    return grad
```

In [8]:

```python
def tanh_gradient(x):
    '''
    :param x: input
    '''

    ###########################################################################
    # Full Mark: 1                                                            #
    # TODO:                                                                   #
    # Computes the gradient of the tanh function evaluated at x.              #
    #                                                                         #
    ###########################################################################

    grad = (1 - tanh(x)**2)

    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################

    return grad
```

In [9]:

```python
def forward(W, X):
    '''
    :param W: weights (including biases) of the neural network. It is a list containing both W_hidden and W_outpu
    :param X: input. Already added one additional column of all "1"s.
    '''

    # Shape of W_hidden: [num_feature+1, num_hidden]
    # Shape pf W_output: [num_hidden+1, num_output]
    W_hidden, W_output = W

    ###########################################################################
    # Full Mark: 1                                                            #
    # TODO:                                                                   #
    # Implement the forward pass. You need to compute four values:            #
    # z_hidden, a_hidden, z_output, a_output                                  #
    #                                                                         #
    # Note that our neural network consists of three layers:                  #
    # Input -> Hidden -> Output                                               #
    # The activation function in hidden layer is 'tanh'                       #
    # The activation function in output layer is 'sigmoid'                    #
    ###########################################################################

    # convert to numpy array
    W_hidden = np.array(W_hidden)
    W_output = np.array(W_output)

    # Forward pass through hidden layer
    z_hidden = np.dot(X,W_hidden)
    a_hidden = tanh(z_hidden)
```

```python
    # Add bias column
    a_hidden = np.concatenate([np.ones(( X.shape[0], 1)), a_hidden], axis=1)

    # Forward pass through Output layer
    z_output = np.dot(a_hidden,W_output)
    a_output = sigmoid(z_output)

    ##################################################################################
    #                       END OF YOUR CODE                                         #
    ##################################################################################

    # z_hidden is the raw output of hidden layer, a_hidden is the result after performing activation on z_hidden
    # z_output is the raw output of output layer, a_output is the result after performing activation on z_output
    return {'z_hidden': z_hidden, 'a_hidden': a_hidden,
            'z_output': z_output, 'a_output': a_output}
```

In [10]:

```python
def loss_funtion(W, X, y, num_feature, num_hidden, num_output, L2_lambda):
    '''
    :param W: a 1D array containing all weights and biases.
    :param X: input
    :param y: labels of X
    :param num_feature: number of features in X
    :param num_hidden: number of hidden units
    :param num_output: number of output units
    :param L2_lambda: the coefficient of regularization term
    '''

    m = y.size

    # Reshape W back into the parameters W_hidden and W_output
    W_hidden = np.reshape(W[:num_hidden * (num_feature + 1)],
                          ((num_feature + 1), num_hidden))

    W_output = np.reshape(W[(num_hidden * (num_feature + 1)):],
                          ((num_hidden + 1), num_output))

    # Initialize grads
    W_hidden_grad = np.zeros(W_hidden.shape)
    W_output_grad = np.zeros(W_output.shape)

    # Add one column of "1"s to X
    X_input = np.concatenate([np.ones((m, 1)), X], axis=1)

    ###########################################################################################
    # Full Mark: 3                                                                            #
    # TODO:                                                                                   #
    # 1. Transform y to one-hot encoding. Implement binary cross-entropy loss function        #
    # (Hint: Use the forward function to get necessary outputs from the model)                #
    #                                                                                         #
    # 2. Add L2 regularization to all weights in loss                                         #
    # (Note that we will not add regularization to bias)                                      #
    #                                                                                         #
    # 3. Compute the gradient for W_hidden and W_output (including both weights and biases)   #
    # (Hint: use chain rule, and the sigmoid gradient, tanh gradient you have                 #
    # implemented above. Don't forget to add the gradient of regularization term)             #
    ###########################################################################################

    # One hot encoding
    shape = (y.size, y.max()+1)
    one_hot = np.zeros(shape)
    rows = np.arange(y.size)
    one_hot[rows, y] = 1

    # forward pass
    f_pass = forward((W_hidden,W_output),X_input)
    a_hidden = f_pass['a_hidden']
    a_output = f_pass['a_output']

    # cross-entropy loss
    logprobs = np.multiply(one_hot, np.log(f_pass['a_output'])) + np.multiply((1 - one_hot), np.log(1 - f_pass['a
    L = (-1.0/m) * np.sum(logprobs)

    # L2 regularize
    sum_w = np.sum(np.square(W_hidden)) + np.sum(np.square(W_output))
    L = L + (sum_w * (L2_lambda/(2*m)))
    L = np.squeeze(L)

    ##
    ### Chain Rule
    ##

    # Gradient at output layer
    dZ2 = sigmoid_gradient(a_output)
    dW2 = np.dot(a_hidden[:,1:].T,dZ2)/m + ((L2_lambda / m) * W_output[1:,:])
    dB2 = np.sum(dZ2, axis=0)/m

    # Gradient at hidden layer
```

```python
    dZ1 = np.multiply(dZ2.dot(W_output[1:,].T), tanh_gradient(a_hidden[:,1:]))
    dW1 = (np.dot(X.T, dZ1))/m + ((L2_lambda / m) * W_hidden[1:,:])
    dB1 = np.sum(dZ1, axis=0)/m

    # Reshape Bias Gradient matrix
    dB1 = dB1.reshape(1,10)
    dB2 = dB2.reshape(1,3)

    # Final Gradients of weights
    W_hidden_grad = np.concatenate((dB1, dW1), axis = 0)
    W_output_grad = np.concatenate((dB2, dW2), axis = 0)

    ##################################################################################
    #                        END OF YOUR CODE                                        #
    ##################################################################################

    grads = np.concatenate([W_hidden_grad.ravel(), W_output_grad.ravel()])

    return L, grads
```

In [11]:
```python
def optimize(initial_W, X, y, num_epoch, num_feature, num_hidden, num_output, L2_lambda):
    '''
    :param initial_W: initial weights as a 1D array.
    :param X: input
    :param y: labels of X
    :param num_epoch: number of iterations
    :param num_feature: number of features in X
    :param num_hidden: number of hidden units
    :param num_output: number of output units
    :param L2_lambda: the coefficient of regularization term
    '''

    options = {'maxiter': num_epoch}

    ##########################################################################################
    # Full Mark: 1                                                                           #
    # TODO:                                                                                  #
    # Optimize weights                                                                       #
    # (Hint: use scipy.optimize.minimize to automatically do the iteration.)                 #
    # ref: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html) #
    # For some optimizers, you need to set 'jac' as True.                                    #
    # You may need to use lambda to create a function with one parameter to wrap the         #
    # loss_funtion you have implemented above.                                               #
    #                                                                                        #
    # Note that scipy.optimize.minimize only accepts a 1D weight array as initial weights,   #
    # and the output optimized weights will also be a 1D array.                              #
    # That is why we unroll the initial weights into a single long vector.                   #
    ##########################################################################################

    # initialize W_final
    W_final = np.copy(initial_W)

    # Function to run at each epoch
    def fun(W):
        global W_final
        L, grad = loss_funtion(W, X, y, num_feature, num_hidden, num_output, L2_lambda)
        W_final = W - grad
        return L

    # Optimize (Run Epochs)
    minimize(fun, initial_W, options=options)

    ##################################################################################
    #                        END OF YOUR CODE                                        #
    ##################################################################################

    # Obtain W_hidden and W_output back from W_final
    W_hidden = np.reshape(W_final[:num_hidden * (num_feature + 1)],
                          ((num_feature + 1), num_hidden))
    W_output = np.reshape(W_final[(num_hidden * (num_feature + 1)):],
                          ((num_hidden + 1), num_output))

    return [W_hidden, W_output]
```

In [12]:
```python
def predict(X_test, y_test, W):
    '''
    :param X_test: input
    :param y_test: labels of X_test
    :param W: a list containing two weights W_hidden and W_output.
    '''

    test_input = np.concatenate([np.ones((y_test.size, 1)), X_test], axis=1)

    ##################################################################################
    # Full Mark: 1                                                                   #
    # TODO:                                                                          #
    # Predict on test set and compute the accuracy.                                  #
    # (Hint: use forward function to get predicted output)                           #
```

```
    #                                                                              #
    ################################################################################

    # Forward pass
    cache = forward(W, test_input)

    # One-hot-encode actual values
    shape = (y_test.size, y_test.max()+1)
    one_hot = np.zeros(shape)
    rows = np.arange(y_test.size)
    one_hot[rows, y_test] = 1

    # Encode predicted outputs
    y_hat = np.zeros(shape)
    for i, r in enumerate(cache['a_output']):
        max_val = np.amax(r)
        for j, v in enumerate(r):
            if max_val == v:
                y_hat[i,j] = 1

    # Find Accuracy
    acc = 0
    for i, _ in enumerate(one_hot):
        if np.array_equal(one_hot[i], y_hat[i]):
            acc += 1

    acc = acc / y_test.size

    ################################################################################
    #                        END OF YOUR CODE                                      #
    ################################################################################

    return acc
```

In [13]:

```
# Do not modify this part #
# Define parameters
NUM_FEATURE = 4
NUM_HIDDEN_UNIT = 10
NUM_OUTPUT_UNIT = 3
NUM_EPOCH = 100
L2_lambda = 1

# Load data
X, y = load_dataset()
X_train, X_test, y_train, y_test = train_test_split(X, y)

# Initialize weights
initial_W_hidden = init_weights(NUM_FEATURE, NUM_HIDDEN_UNIT)
initial_W_output = init_weights(NUM_HIDDEN_UNIT, NUM_OUTPUT_UNIT)
initial_W = np.concatenate([initial_W_hidden.ravel(), initial_W_output.ravel()], axis=0)

# Neural network learning
W = optimize(initial_W, X_train, y_train, NUM_EPOCH, NUM_FEATURE, NUM_HIDDEN_UNIT, NUM_OUTPUT_UNIT, L2_lambda)

# Predict
acc = predict(X_test, y_test, W)
print("Test accuracy:", acc)
```

```
Test accuracy: 0.74
```