

Lab 5: Neural Network*Student: Xun Xu**ID: 1926930*

5.1 Lab introduction

Lab5 is to implement the two-layer MLP (multilayer perceptron) algorithm, which is a classic neural network. In this experiment, we use the publicly dataset to verify our algorithm. Stochastic Gradient Descent (SGD) is used to update the gradient, so that the loss function (that is the error) is minimized.

5.2 Methodology and Coding comment

5.2.1 Read data

Read the dataset into a list and shuffle it with the random.shuffle method. Hint: fix the random seed (e.g. random.seed(17)) before calling random.shuffle.

Code show as below:

```
# Read file
random.seed(17)
f = open('./breast-cancer-wisconsin.data', 'r+')
dataset = f.readlines()
f.close()
print('input data','length:',len(dataset), ',part of data:',dataset[0:2])

random.shuffle(dataset)

# Separate string data according to commas
for i in range(len(dataset)):
    dataset[i] = dataset[i].split(',')
print('split input data','length:',len(dataset[i]),',part of data:',dataset[0:2])

# Convert string data to numeric data
data = []
for i in range(len(dataset)):
    for j in range(len(dataset[0])):
        data.append(int(dataset[i][j]))
# Rearrange to original dimension
new_data = np.asarray(data).reshape(len(dataset), 11)
print('numeric data(raw data)', 'type:', type(new_data[0][0]), ',shape:', new_data.shape)
print('part of data:', new_data[0:2])
```

The result is shown as figure 5.1

```
input data length: 683 ,part of data: ['1000025,5,1,1,1,2,1,3,1,1,2\n', '1002945,5,4,4,5,7,10,3,2,1,2\n']
split input data length: 11 ,part of data: ['1000025', '5', '1', '1', '1', '2', '1', '3', '1', '1', '2\n']
numeric data(raw data) type: <class 'numpy.int32'> ,shape: (683, 11)
part of data:
[[1000025      5      1      1      1      2      1      3      1
      1      2]
 [1002945      5      4      4      5      7     10      3      2
      1      2]]
type <class 'numpy.ndarray'>
```

Figure 5.1: read data (portion)

5.2.2 Read data

Split the dataset as five parts to do cross-fold validation: Each of 5 subsets was used as test set and the remaining data was used for training. Five subsets were used for testing rotationally to evaluate the classification accuracy.

Code show as below:

```

x_data = x[:, 1:x.shape[1]-1]
x_label = x[:, x.shape[1]-1]

index = math.floor(x.shape[0]/5)
data_set0 = x_data[0:index]
data_set1 = x_data[index:index*2]
data_set2 = x_data[index*2:index*3]
data_set3 = x_data[index*3:index*4]
data_set4 = x_data[index*4:]
data_label0 = x_label[0:index]
data_label1 = x_label[index:index*2]
data_label2 = x_label[index*2:index*3]
data_label3 = x_label[index*3:index*4]
data_label4 = x_label[index*4:]
print('type',type(data_set1))

# five sets
print('five data set', '\ndata1 shape',data_set0.shape,'\ndata2 shape',data_set1.shape,'\ndata3
shape',data_set2.shape,'\ndata4 shape',data_set3
.shape,'\ndata5 shape',data_set4.shape)

# five labels
print('five label set', '\nlabel1 shape',data_label0.shape,'\nlabel2 shape',data_label1.shape,'\
nlabel3 shape',data_label2.shape,'\nlabel4 shape
',data_label3.shape,'\nlabel5 shape',data_label4
.shape)

whole_data = np.array([data_set0,data_set1,data_set2,data_set3,data_set4])
whole_label = np.array([data_label0,data_label1,data_label2,data_label3,data_label4])

```

My idea here is to divide the original data set and label set into five parts, and then combine them in one data to facilitate extraction by index, and then extract the required training set and training set by index and stitching during cross-validation Test set (including corresponding tags). The result is shown as figure 5.2

```

five data set
data1 shape (136, 9)
data2 shape (136, 9)
data3 shape (136, 9)
data4 shape (136, 9)
data5 shape (139, 9)
five label set
label1 shape (136,)
label2 shape (136,)
label3 shape (136,)
label4 shape (136,)
label5 shape (139,)

```

Figure 5.2: Split the dataset as five parts

5.2.3 Add bias

All input feature vectors are augmented with the 1 as follows :

$$\hat{X} = \begin{bmatrix} X & 1_{N \times 1} \end{bmatrix},$$

since:

$$w^T x + w_0 = \begin{bmatrix} w^T & w_0 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix}$$

Code show as below:

```
X_train = np.insert(train_set, train_set.shape[1] - 1, values=np.ones(train_set.shape[0]), axis=1)
X_test = np.insert(test_set, test_set.shape[1] - 1, values=np.ones(test_set.shape[0]), axis=1)
```

The result is shown as figure 5.3

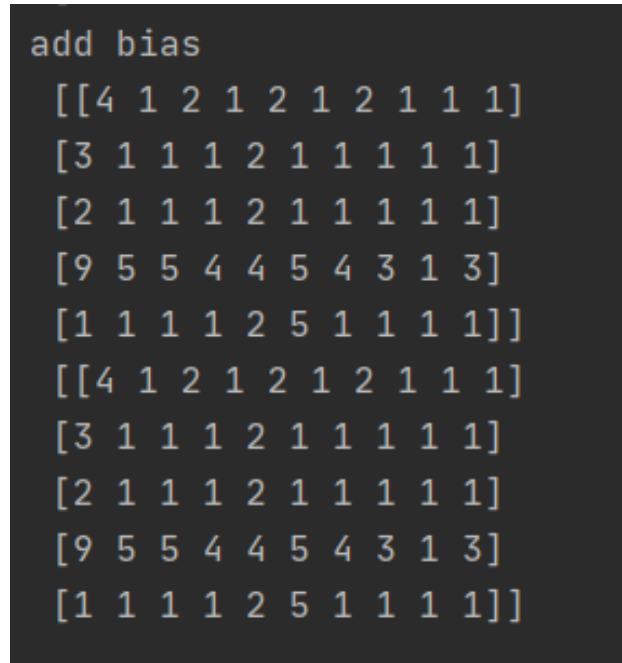


Figure 5.3: add bias (portion)

5.2.4 Scale values

Scale linearly the attribute values x_{ij} of the data matrix \hat{X} into $[-1,1]$ for each dimensional feature as follows:

$$x_{ij} \leftarrow 2 \frac{x_{ij} - \min_i x_{ij} + 10^{-6}}{\max_i x_{ij} - \min_i x_{ij} + 10^{-6}} - 1$$

Code show as below:

```
def scale(training_set):
    # method 1
    max_i = np.max(training_set, axis=0)
    min_i = np.min(training_set, axis=0)
    y = 2 * (training_set - min_i + 10 ** -6) / (max_i - min_i + 10 ** -6) - 1
    return y
```

The result is shown as figure 5.4

```
scale data:
[[-0.33333319 -0.99999978 -0.77777758 -0.99999978 -0.77777758 -0.99999978
 -0.77777758 -0.99999978  1.          -0.99999978]
 [-0.55555538 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.99999978
 -0.99999978 -0.99999978  1.          -0.99999978]
 [-0.77777758 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.99999978
 -0.99999978 -0.99999978  1.          -0.99999978]
 [ 0.77777778 -0.11111099 -0.11111099 -0.33333319 -0.33333319 -0.11111099
 -0.33333319 -0.55555538  1.          -0.55555538]
 [-0.99999978 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.11111099
 -0.99999978 -0.99999978  1.          -0.99999978]]
```

Figure 5.4: Scale values(portion)

5.2.5 One hot coding

The label l_n of the n -th example is converted into a K dimensional vector t_n as follows (K is the number of the classes) :

$$t_{nk} = \begin{cases} +1, & k = l_n \\ 0, & k \neq l_n \end{cases}$$

Code show as below:

```
y_0 = np.array([1 if label == 2 else 0 for label in train_label])
y_0 = np.reshape(y_0, (len(train_label), 1))
y_1 = np.array([1 if label == 4 else 0 for label in train_label])
y_1 = np.reshape(y_1, (len(train_label), 1))
y_onehot = np.hstack((y_0, y_1))
print('y_onehot:', 'shape:', y_onehot.shape, '\n', y_onehot[0:5])
```

After the corresponding value is changed, it can be obtained by splicing. This step is called onehot in the current machine learning, here I use the code I wrote, without using the api. The result is shown as figure ??

```

y_onehot: shape: (136, 2)
[[1 0]
 [1 0]
 [1 0]
 [0 1]
 [1 0]]

```

Figure 5.5: converted labels(portion)

5.2.6 Initialize all weights

Initialize all weights w_{ij} of MLP network such as:

$$w_{ij} \in \left[-\sqrt{\frac{6}{D+1+K}}, \sqrt{\frac{6}{D+1+K}}\right]$$

where D and K is the number of the input nodes and the output nodes (each node is related to a class), respectively.

Code show as below:

```

w_range = (6/(input_size + 1 + num_labels))**0.5
print('\nw_range',w_range)
size = hidden_size * (input_size + 1) + num_labels * hidden_size
print('\nparams_size',size)
# params = np.linspace(-w_range,w_range,size)
params = np.random.uniform(-w_range,w_range,size)
print('\nparams',params[0:5])

```

The result is shown as figure 5.6

```

params_size 48

params
[ 0.5180414 -0.6142434  0.16507882  0.48728362  0.66110906 -0.31581669
-0.211798   -0.26645758  0.42451669 -0.36051109 -0.28212721 -0.19872898
 0.57485642  0.210307   -0.58727763 -0.53379894  0.24918175  0.23745601
-0.20100498  0.12042917  0.28149433  0.24682384 -0.20908699  0.07369369
-0.30552833 -0.32951447  0.31509979  0.2974377  -0.15254348 -0.06929708
-0.22417297  0.37334415 -0.24064177  0.5852053  0.53455479 -0.32328907
 0.70240631  0.34375626 -0.51613855 -0.41654778 -0.64809397  0.27306283
-0.65246729  0.60136879  0.63846412  0.48953245  0.52230332  0.36857275]

```

Figure 5.6: Initialize weights

5.2.7 Forward propagation:

Choose randomly an input vector x to network and forward propagate through the network (H is the number of the hidden units):

$$a_j = \sum_{i=0}^D w_{ji}^{(1)} x_i$$

$$z_j = \tanh(a_j)$$

$$y_k = \sum_{j=0}^H w_{kj}^{(2)} z_j$$

Code show as below:

```

def forward_propagate(X, theta1, theta2):
    a = np.dot(X, theta1.T)
    z = np.tanh(a)
    yk = np.dot(z, theta2.T)
    return a, z, yk

```

The result is shown as figure 5.7

```
forward_propagate:
a shape: (136, 4)
z shape: (136, 4)
yk shape: (136, 2)
```

Figure 5.7: Forward propagation

The number of hidden layers I use for forward propagation here is 4, which can be changed by modifying the foremost hyperparameters of the main function. The number of data set samples I brought here is 136. According to the dimensions of each part, it can be concluded that the result is correct.

5.2.8 Lost function:

To obtain the error rate $E = \frac{1}{2} \sum_{k=1}^K (y_k - t_k)^2$ of the example x .

Code show as below:

```
def cost( h, y):
m = h.shape[0]
J = 0
for i in range(m): # Iterate through each sample
    J += (np.sum(h[i,:] - y[i,:])**2)
J = 0.5*J/m
return J
```

The result is shown as figure 5.8

```
cost of sample 1.8030419465916456
```

Figure 5.8: error(cost function result)

The cost is the error of samples, which is the loss function. Our purpose is to minimize the loss function beauty, that is, the method of using gradient descent. The figure 5.8 shows the error of a single sample after calculation.

5.2.9 Back propagation and gradient descent:

Evaluate the δ_k for all output units:

$$\delta_k = y_k - t_k$$

Backpropagate the δ 's to obtain δ_j for each hidden unit in the network:

$$\begin{aligned}\delta_j &= \tanh(a_j)' \sum_{k=1}^K w_{kj} \delta_k \\ &= (1 - z_j^2) \sum_{k=1}^K w_{kj} \delta_k\end{aligned}$$

The derivative with respect to the first-layer and the second-layer weights are given by:

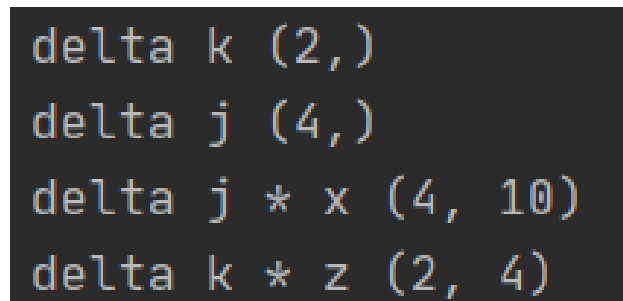
$$\frac{\partial E}{\partial w_{ji}^{(1)}} = \delta_j x_i, \quad \frac{\partial E}{\partial w_{kj}^{(2)}} = \delta_k z_j$$

Code show as below:

```
for t in range(m):
    xt = np.array(X[t,:])
    at = np.array(a[t,:])
    zt = np.asarray(z[t,:])
    ykt = np.array(yk[t,:])
    y_onehott = np.array(y_onehot[t,:])

    dyt = ykt - y_onehott
    dat = np.dot(np.dot(dyt, theta2), tanh_gradient(zt))
    w1 = w1 - learning_rate*(np.dot(dat.reshape((hidden_size,1)), xt.reshape((1,input_size+1))))
    w2 = w2 - learning_rate*(np.dot(dyt.reshape((num_labels,1)), zt.reshape((1,hidden_size))))
```

The result is shown as figure 5.9



```
delta k (2,)
delta j (4,)
delta j * x (4, 10)
delta k * z (2, 4)
```

Figure 5.9: Dimension of back propagation and gradient descent

I integrated these three steps together, we seek the gradient descent after back propagation. Here I show the calculation of each part, the first is the forward propagation (here takes the calculation of a single sample 10×1) as an example):

$$\begin{aligned}a_j &= \sum_{i=0}^D w_{ji}^{(1)} x_i & (4 \times 10) \times (10 \times 1) &= 4 \times 1 \\ z_j &= \tanh(a_j) & (4 \times 1) & \\ y_k &= \sum_{j=0}^H w_{kj}^{(2)} z_j & (2 \times 4) \times (4 \times 1) &= 2 \times 1\end{aligned}$$

For the back propagation, this is the result of the error derivation of the weights:

$$\begin{aligned}
 \delta_k &= y_k - t_k & 2 * 1 - 2 * 1 \\
 \delta_j &= \tanh(a_j)' \sum_{k=1}^K w_{kj} \delta_k & (4 * 4) * (4 * 2) * (2 * 1) = 4 * 1 \\
 &= (1 - z_j^2) \sum_{k=1}^K w_{kj} \delta_k \\
 \frac{\partial E}{\partial w_{ji}^{(1)}} &= \delta_j x_i & (4 * 1) * (1 * 10) = 4 * 10 \\
 \frac{\partial E}{\partial w_{kj}^{(2)}} &= \delta_k z_j & (2 * 1) * (1 * 4) = 2 * 4
 \end{aligned}$$

In fact, in the code, I used the the sample times transposition of the weight, so the specific order may be slightly different from the formula, but the result dimension is the same.

5.3

After repeated training, I found that the accuracy of the experimental results has a great relationship with the initialization results, resulting in very unstable final results. No matter how you change the size of the hidden layer, the learning rate, and the number of iterations, there is no way to optimize the accuracy. I think it has something to do with the amount of data we have. Too little data leads to overfitting. In order to get a better accuracy, I decided to use an outer loop of 100 iterations, to ensure that I can get better data as much as possible and not to waste too much computer performance, and record all the weights for easy extraction later. Code show as below:

```
weight = []
best_weight_index = []
index = 0
acc1 = []
acc2 = []
acc3 = []
acc4 = []
acc5 = []
acc = [acc1, acc2, acc3, acc4, acc5]
# method 1
for p in range(5):
    str = [0, 1, 2, 3, 4]
    str.remove(p)
    whole_data, whole_label = read_data()
    train_set = np.vstack((whole_data[str[0]], whole_data[str[1]], whole_data[str[2]], whole_data[str[3]]))[:,]
    train_label = np.hstack((whole_label[str[0]], whole_label[str[1]], whole_label[str[2]], whole_label[str[3]]))[:,]
    test_set = whole_data[p][:,]
    test_label = whole_label[p][:,]
    X_train = np.insert(train_set, train_set.shape[1] - 1, values=np.ones(train_set.shape[0]), axis=1)
    X_train = scale(X_train)
    X_test = np.insert(test_set, test_set.shape[1] - 1, values=np.ones(test_set.shape[0]), axis=1)
    X_test = scale(X_test)
    y_0 = np.array([1 if label == 2 else 0 for label in train_label])
    y_0 = np.reshape(y_0, (len(train_label), 1))
    y_1 = np.array([1 if label == 4 else 0 for label in train_label])
    y_1 = np.reshape(y_1, (len(train_label), 1))
    y_onehot = np.hstack((y_0, y_1))
    y_test = np.array([1 if label == 2 else 0 for label in test_label])
    y_test = np.reshape(y_test, (len(test_label), 1))

    for i in range(100):
        J1, J2, w, w1 = backprop(params, input_size, hidden_size, num_labels, X_train, y_onehot, learning_rate=0.001)

        params = w1
        weight.append(w)
        theta1 = np.reshape(w[:hidden_size * (input_size + 1)], (hidden_size, (input_size + 1)))
        theta2 = np.reshape(w[hidden_size * (input_size + 1):], (num_labels, hidden_size))
        a2, z2, yk2 = forward_propagate(X_test, theta1, theta2)
        y_pred = np.array(np.argmax(yk2, axis=1))
        correct = [1 if a == b else 0 for (a, b) in zip(y_pred, y_test)]
        accuracy = (sum(map(int, correct)) / float(len(correct)))
        acc[p].append(accuracy)
        index += 1

    if accuracy > 0.95:
        best_weight_index.append(index)

for p in range(5):
    print('acc', acc[p])
```

```
print('best', acc[p][np.argmax(acc[p])])
```

The result is shown as figure 5.10

```
accuracy [0.3897058823529412, 0.6029411764705882, 0.9485294117647058, 0.7279411764705882, 0.4632352941176471, 0.7132352941176471,
best accuracy 0.9485294117647058
accuracy [0.7573529411764706, 0.7941176470588235, 0.5441176470588235, 0.7941176470588235, 0.6617647058823529, 0.7941176470588235,
best accuracy 0.7941176470588235
accuracy [0.6397058823529411, 0.7720588235294118, 0.7867647058823529, 0.7720588235294118, 0.7867647058823529, 0.7720588235294118,
best accuracy 0.7867647058823529
accuracy [0.5735294117647058, 0.8897058823529411, 0.36764705882352944, 0.8897058823529411, 0.36764705882352944, 0.8897058823529411,
best accuracy 0.8897058823529411
accuracy [0.5035971223021583, 0.8633093525179856, 0.60431654676259, 0.8633093525179856, 0.60431654676259, 0.8633093525179856, 0.60431654676259,
best accuracy 0.8633093525179856
```

Figure 5.10: accuracy

But even this result is still based on unstable training, and can only have a little reference significance. In the attachment of code implementation, in order to prevent too much printed information from causing interference, I provide two codes, one is to view the results and dimensions of each step, and the other is to run cross-validation.