Lab1 : back-propagation

Name : 鄭謝廷揚 Student ID : A073501

1. Introduction :

In this lab, I write a simple neuron network with forward pass and backpropagation using two hidden layers. Our goal is target to classify input data into two classes. In order to classification I use sigmoid function to transform data from linearity to non-linearity, and use cross-entropy as loss function. After training, input data can be precise classified.

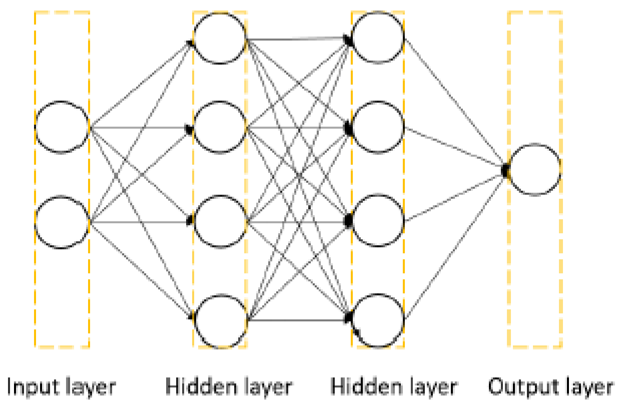
2. Experiment setups :

A. Sigmoid functions

```
def sigmoid(Z):
    return 1.0/(1.0+np.exp(-Z))

def sigmoid_backward(dA, Z):
    sig = sigmoid(Z)
    return dA * sig * (1 - sig)
```

B. Neural network

Neuron network architecture:



Implement:

i. Setting parameters

```
nn_architecture = [
    {"input_dim": 2, "output_dim": 4},
    {"input_dim": 4, "output_dim": 4},
    {"input_dim": 4, "output_dim": 1}
]
params_values = init_layers(nn_architecture)
```

ii. Initiation layers

```
def init_layers(nn_architecture, seed = 99):
    np.random.seed(seed)
    number_of_layers = len(nn_architecture)
    params_values = {}

    for idx, layer in enumerate(nn_architecture):
        layer_idx = idx + 1
        layer_input_size = layer["input_dim"]
        layer_output_size = layer["output_dim"]
        params_values['W' + str(layer_idx)] = abs(np.random.randn(layer_output_size, layer_input_size) * 0.1)
    return params_values
```

np.random.randn(m,n) creat mxn matrix with gaussian distribution of mean 0 and variance 1between 0~1

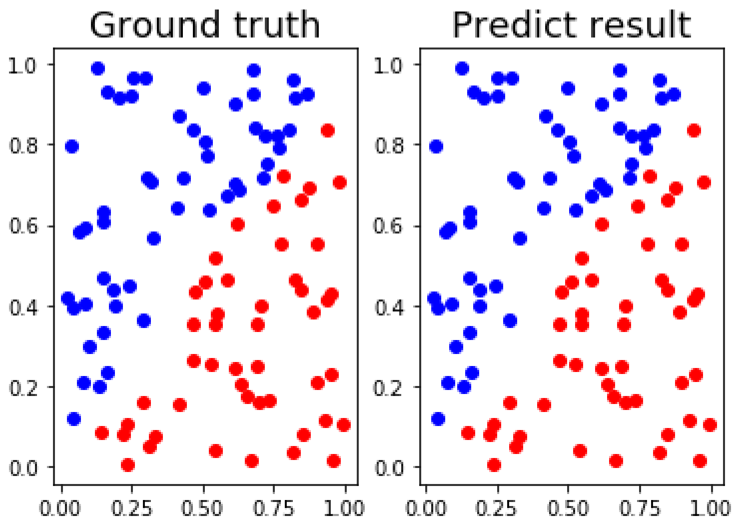
C. Backpropagation

```
def single_layer_backward_propagation(dA_curr, W_curr, Z_curr, A_prev):
    # number of examples
    m = A_prev.shape[1]

    # calculation of the activation function derivative
    dZ_curr = sigmoid_backward(dA_curr, Z_curr)
    # derivative of the matrix W
    dW_curr = np.dot(dZ_curr, A_prev.T) / m
    # derivative of the matrix A_prev
    dA_prev = np.dot(W_curr.T, dZ_curr)
    return dA_prev, dW_curr

def full_backward_propagation(Y_hat, Y, memory, params_values, nn_architecture):
    grads_values = {}
    m = Y.shape[1]
    Y = Y.reshape(Y_hat.shape)

    dA_prev = - (np.divide(Y, Y_hat) - np.divide(1 - Y, 1 - Y_hat));

    for layer_idx_prev, layer in reversed(list(enumerate(nn_architecture))):
        layer_idx_curr = layer_idx_prev + 1
        dA_curr = dA_prev
        A_prev = memory["A" + str(layer_idx_prev)]
        Z_curr = memory["Z" + str(layer_idx_curr)]
        W_curr = params_values["W" + str(layer_idx_curr)]

        dA_prev, dW_curr = single_layer_backward_propagation(
            dA_curr, W_curr, Z_curr, A_prev)
        grads_values["dW" + str(layer_idx_curr)] = dW_curr
    return grads_values
```

3. <u>Results :</u>

i. Result of input data from generate_linear(100) function

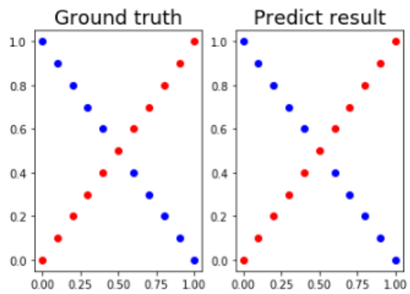
```
epoch =0 cost =0.692340616534196 accu =0.0
epoch =10000 cost =0.001757337621467805 3 accu =0.98
epoch =20000 cost =0.000657957880913636 accu =0.98
epoch =30000 cost =0.0003683228413383988 accu =0.98
epoch =40000 cost =0.0002452570679116494 accu =0.98
epoch =50000 cost =0.000179724711 7116938 accu =0.98
epoch =60000 cost =0.00013988027494183131 accu =0.98
epoch =70000 cost =0.00011344760842384414 accu =0.98
epoch =80000 cost =9.479830557664427e-05 accu =1.0
```



ii. Result of input data from generate_XOR_easy()

```
epoch =0 cost =0.6963296601571213 accu =0.0
epoch =10000 cost =0.6920129836417577 accu =0.0
epoch =20000 cost =0.6920129809443268 accu =0.0
epoch =30000 cost =0.6920129785064699 accu =0.0
epoch =40000 cost =0.6920129762836073 accu =0.0
epoch =50000 cost =0.6920129742392286 accu =0.0
epoch =60000 cost =0.6920129723430485 accu =0.0
epoch =70000 cost =0.6920129705696305 accu =0.0
epoch =80000 cost =0.6920129688973456 accu =0.0
epoch =90000 cost =0.6920129673075736 accu =0.0
epoch =100000 cost =0.6920129657840743 accu =0.0
epoch =110000 cost =0.6920129643124925 accu =0.0
epoch =120000 cost =0.6920129628799536 accu =0.0
epoch =130000 cost =0.6920129614747315 accu =0.0
epoch =140000 cost =0.6920129600859685 accu =0.0
epoch =150000 cost =0.6920129587034326 accu =0.0
epoch =160000 cost =0.6920129573173011 accu =0.0
epoch =170000 cost =0.6920129559179607 accu =0.0
epoch =180000 cost =0.6920129544958187 accu =0.0
epoch =190000 cost =0.6920129530411137 accu =0.0
epoch =200000 cost =0.6920129515437217 accu =0.0
epoch =210000 cost =0.6920129499929498 accu =0.0
epoch =220000 cost =0.6920129483773082 accu =0.0
epoch =230000 cost =0.6920129466842514 accu =0.0
epoch =240000 cost =0.6920129448998769 accu =0.0
epoch =250000 cost =0.6920129430085681 accu =0.0
epoch =260000 cost =0.6920129409925604 accu =0.0
epoch =270000 cost =0.6920129388314065 accu =0.0
epoch =280000 cost =0.6920129365013095 accu =0.0
epoch =290000 cost =0.6920129339742769 accu =0.0
epoch =300000 cost =0.6920129312170332 accu =0.0
epoch =310000 cost =0.6920129281896067 accu =0.0
epoch =320000 cost =0.6920129248434623 accu =0.0
epoch =330000 cost =0.6920129211190059 accu =0.0
epoch =340000 cost =0.6920129169421907 accu =0.0
epoch =350000 cost =0.6920129122198333 accu =0.0
```

```
epoch =350000 cost =0.6920129122198333 accu =0.0
epoch =360000 cost =0.6920129068330274 accu =0.0
epoch =370000 cost =0.69201290062 7702 accu =0.0
epoch =380000 cost =0.6920128934007828 accu =0.0
epoch =390000 cost =0.69201288487 9408 accu =0.0
epoch =400000 cost =0.6920128746888305 accu =0.0
epoch =410000 cost =0.6920128623012384 accu =0.0
epoch =420000 cost =0.6920128469510917 accu =0.0
epoch =430000 cost =0.6920128274888889 accu =0.0
epoch =440000 cost =0.6920128021152918 accu =0.0
epoch =450000 cost =0.69201276786 6766 accu =0.0
epoch =460000 cost =0.6920127195410929 accu =0.0
epoch =470000 cost =0.6920126472223046 accu =0.0
epoch =480000 cost =0.69201252979 2613 accu =0.0
epoch =490000 cost =0.6920123145581815 accu =0.0
epoch =500000 cost =0.6920118330099354 accu =0.0
epoch =510000 cost =0.6920102305876679 accu =0.0
epoch =520000 cost =0.6919897954362207 accu =0.0
epoch =530000 cost =0.001281345476056 5155 accu =1.0
```

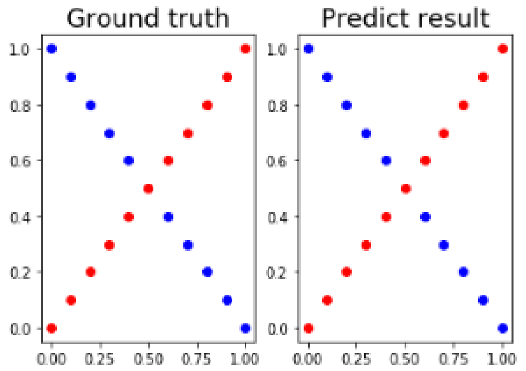


iii. Another result of input data from generate_XOR_easy()

This test use very high learning rate(=5). However, learning rate higher than one is wrong, but in this case can have good result.

```
X,Y =

nn_architecture = [
    {"input_dim": 2, "output_dim": 4},
    {"input_dim": 4, "output_dim": 4},
    {"input_dim": 4, "output_dim": 1}
]
params_values = init_layers(nn_architecture)

learning_rate = 5
for i in range(100000000):
    Y_hat, cashe = full_forward_propagation(X,params_values, nn_architecture)
    #print(Y_hat,Y)
    cost = get_cost_value(Y_hat, Y.T)
    grads_val = full_backward_propagation(Y_hat, Y, cashe, params_values, nn_architecture)
    params_val = update(params_values, grads_val, nn_architecture, learning_rate)
    if(i%10000==0):
        accu = get_accuracy_value(Y_hat, Y.T)
        print('epoch ='+str(i)+' cost ='+str(cost)+' accu ='+str(accu))
        if(accu ==1):
            break;
    #print("\n",Y_hat.shape,"\n",Y_hat)
show_result(X, Y,Y_hat)
```

```
epoch =0 cost =0.6963296601571213 accu =0.0
epoch =10000 cost =0.6920129741167282 accu =0.0
epoch =20000 cost =0.6920129662057138 accu =0.0
epoch =30000 cost =0.6920129596006269 accu =0.0
epoch =40000 cost =0.6920129529839482 accu =0.0
epoch =50000 cost =0.6920129452178303 accu =0.0
epoch =60000 cost =0.6920129347326011 accu =0.0
epoch =70000 cost =0.6920129184131419 accu =0.0
epoch =80000 cost =0.6920128879728743 accu =0.0
epoch =90000 cost =0.6920128111822125 accu =0.0
epoch =100000 cost =0.6920123961145279 accu =0.0
epoch =110000 cost =0.0005878595727402181 accu =1.0
```



4. Discussion :

In this lab, I have learned that how to use Jupyter Notebook and how to use numpy package to set and calculate matrix. Moreover, we need to choose appropriate activate function and loss function for different type of input/output data. For input of this lab using sigmoid function has good accuracy. However, I have tried to use different activate function (hidden layer1 – ReLU, hidden layer2 - sigmoid) to train XOR function input, but it result in bad accuracy.