Lab1 : back-propagation

Lab Objective:

In this lab, you will need to understand and implement simple neural networks with forwarding pass and backpropagation using two hidden layers. Notice that you can only use **Numpy** and the python standard libraries, any other frameworks (ex : Tensorflow、PyTorch) are not allowed in this lab.

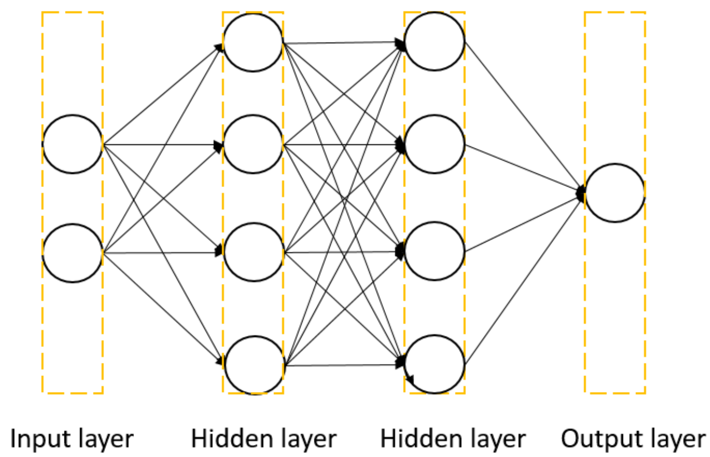


Figure 1. Two-layer neural network

Important Date:

1. Experiment Report Submission Deadline: 7/14 (Thu) 11:59 p.m.
2. Demo date: 7/14 (Thu)

Turn in:

1. Experiment Report (.pdf)
2. Source code

Notice: zip all files in one file and name it like「DL_LAB1_your studentID_name.zip」, ex: 「DL_LAB1_310551109_謝宏笙.zip」

Requirements:

1. Implement simple neural networks with two hidden layers.
2. You must use backpropagation in this neural network and can only use Numpy and other python standard libraries to implement.
3. Plot your comparison figure that show the predicted results and the ground-truth.

Implementation Details:

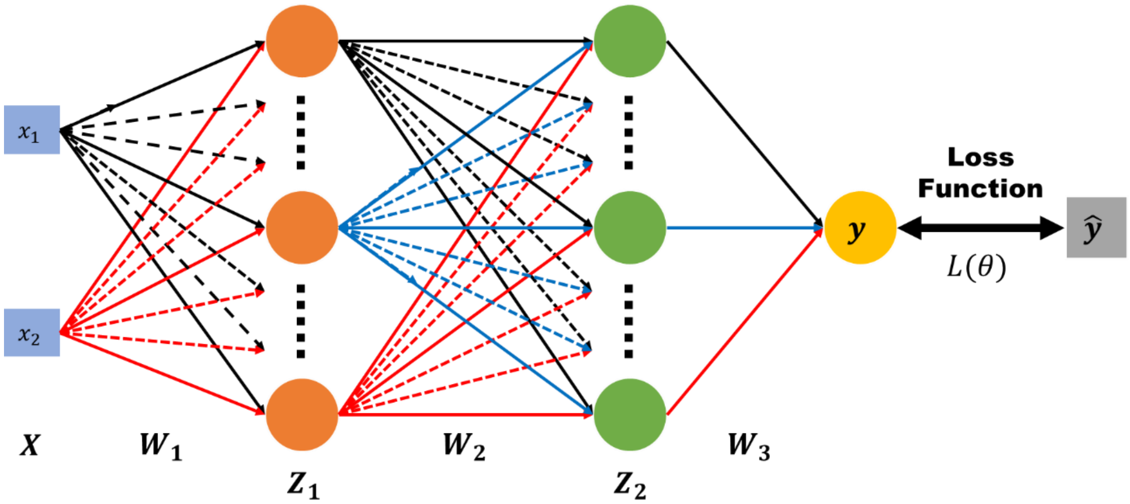


Figure 2. Forward pass

- In the figure 2, we use the following definitions for the notations:
  1. x_1, x_2 : $nerual\ network\ inputs$
  2. X : $[x_1, x_2]$
  3. y : $nerual\ network\ outputs$
  4. $\hat{y}$: $ground\ truth$
  5. $\mathrm{L}(\theta)$: $loss\ function$
  6. W_1, W_2, W_3 : $weight\ matrix\ of\ network\ layers$

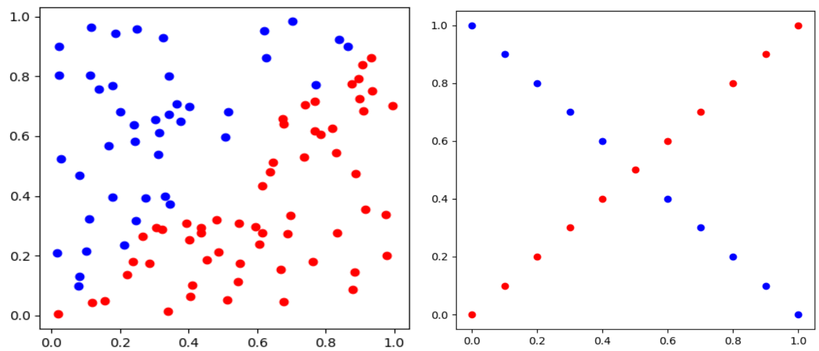
- Here are the computations represented:

  $\boldsymbol{Z_1} = \boldsymbol{\sigma}(\boldsymbol{X}W_1)$ $\qquad$ $\boldsymbol{Z_2} = \boldsymbol{\sigma}(\boldsymbol{Z_1}W_2)$ $\qquad$ $\boldsymbol{y} = \boldsymbol{\sigma}(\boldsymbol{Z_2}W_3)$

- In the equations, the $\boldsymbol{\sigma}$ is sigmoid function that refers to the special case of the **logistic** function and defined by the formula:

  $$\boldsymbol{\sigma}(\mathrm{x}) = \frac{1}{1+e^{-x}}$$

- **Input / Test:**
  The inputs are two kinds which showing at below.



You need to use the following generate functions to create your inputs x, y.

```
def generate_linear(n=100):
    import numpy as np
    pts = np.random.uniform(0, 1, (n, 2))
    inputs = []
    labels = []
    for pt in pts:
        inputs.append([pt[0], pt[1]])
        distance = (pt[0]-pt[1])/1.414
        if pt[0] > pt[1]:
            labels.append(0)
        else:
            labels.append(1)
    return np.array(inputs), np.array(labels).reshape(n, 1)
```

```
def generate_XOR_easy():
    import numpy as np
    inputs = []
    labels = []

    for i in range(11):
        inputs.append([0.1*i, 0.1*i])
        labels.append(0)

        if 0.1*i == 0.5:
            continue

        inputs.append([0.1*i, 1-0.1*i])
        labels.append(1)

    return np.array(inputs), np.array(labels).reshape(21, 1)
```

Function usage

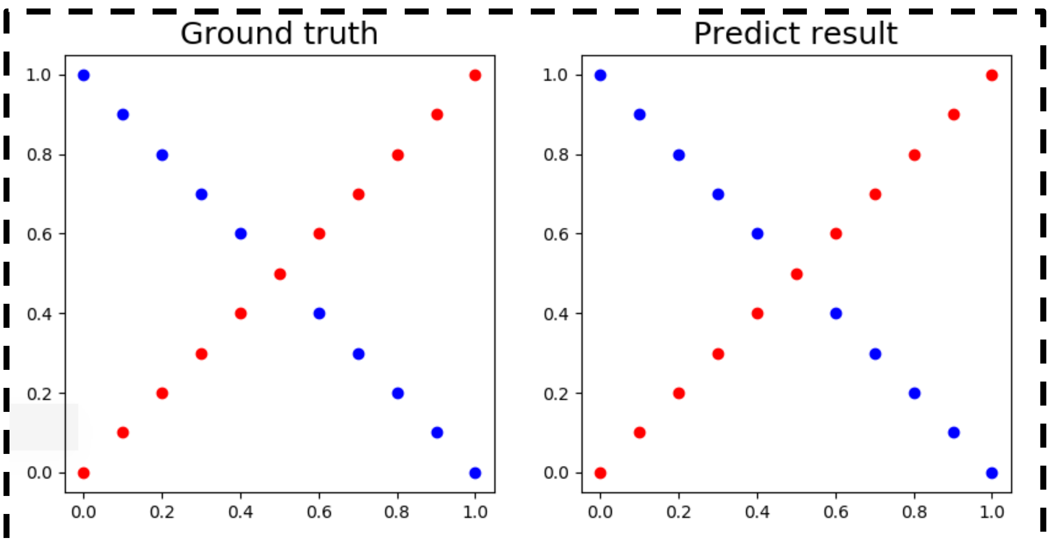
```
x, y = generate_linear(n=100)
x, y = generate_XOR_easy()
```

In the training, you need to print the loss values; In the testing, you need to show your predictions as shown below.

```
epoch 10000 loss : 0.16234523253277644
epoch 15000 loss : 0.2524336634177614
epoch 20000 loss : 0.1590783047540092
epoch 25000 loss : 0.22099447030234853
epoch 30000 loss : 0.3292173477217561
epoch 35000 loss : 0.40406233282426085
epoch 40000 loss : 0.43052897480298924
epoch 45000 loss : 0.420752573558605
epoch 50000 loss : 0.3934759509342479
epoch 55000 loss : 0.3615008372106921
epoch 60000 loss : 0.33077879872648525
epoch 65000 loss : 0.30333537090819584
epoch 70000 loss : 0.2794858089741792
epoch 75000 loss : 0.25892812312991587
epoch 80000 loss : 0.24119780823897027
epoch 85000 loss : 0.22583656353511342
epoch 90000 loss : 0.21244497028971704
epoch 95000 loss : 0.2006912468389013
```

```
[[0.01025062]
 [0.99730607]
 [0.02141321]
 [0.99722154]
 [0.03578171]
 [0.99701922]
 [0.04397049]
 [0.99574117]
 [0.04162245]
 [0.92902792]
 [0.03348791]
 [0.02511045]
 [0.94093942]
 [0.01870069]
 [0.99622948]
 [0.01431959]
 [0.99434455]
 [0.01143039]
 [0.98992477]
 [0.00952752]
 [0.98385905]]
```

Visualize the predictions and ground truth at the end of the training process.
The comparison figure should like example as below.



You can refer to the following visualization code
x: inputs (2-dimensional array)
y: ground truth label (1-dimensional array)
pred_y: outputs of neural network (1-dimensional array)

```python
def show_result(x, y, pred_y):
    import matplotlib.pyplot as plt
    plt.subplot(1,2,1)
    plt.title('Ground truth', fontsize=18)
    for i in range(x.shape[0]):
        if y[i] == 0:
            plt.plot(x[i][0], x[i][1], 'ro')
        else:
            plt.plot(x[i][0], x[i][1], 'bo')

    plt.subplot(1,2,2)
    plt.title('Predict result', fontsize=18)
    for i in range(x.shape[0]):
        if pred_y[i] == 0:
            plt.plot(x[i][0], x[i][1], 'ro')
        else:
            plt.plot(x[i][0], x[i][1], 'bo')

    plt.show()
```

- **Sigmoid functions:**
    1. A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. It is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point. In general, a sigmoid function is monotonic, and has a first derivative which is bell shaped.
    2. (hint) You may write the function like this:

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

    3. (hint) The derivative of sigmoid function

    ```python
    def derivative_sigmoid(x):
        return np.multiply(x, 1.0 - x)
    ```

- **Back Propagation (Gradient computation)**
  Backpropagation is a method used in artificial neural networks to calculate a gradient that is needed in the calculation of the weights to be used in the network. Backpropagation is a generalization of the delta rule to multi-layered feedforward networks, made possible by using the chain rule to

iteratively compute gradients for each layer. The backpropagation learning algorithm can be divided into two parts; **propagation** and **weight update**.

Part 1: Propagation

Each propagation involves the following steps:

1. Propagation forward through the network to generate the output value
2. Calculation of the cost L(θ) (error term)
3. Propagation of the output activations back through the network using the training pattern target in order to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons.

Part 2: Weight update

For each weight-synapse follow the below steps:

1. Multiply its output delta and input activation to get the gradient of the weight.
2. Subtract a ratio (percentage) of the gradient from the weight.
3. This ratio (percentage) influences the speed and quality of learning; it is called the **learning rate**. The greater the ratio, the faster the neuron trains; the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates where the error is increasing, this is why the weight must be updated in the opposite direction.

Repeat part. 1 and 2 until the performance of the network is satisfactory.

Pseudocode:

```
initialize network weights (often small random values)
do
   forEach training example named ex
      prediction = neural-net-output(network, ex)  // forward pass
      actual = teacher-output(ex)
      compute error (prediction - actual) at the output units
      compute Δw_h for all weights from hidden layer to output layer  // backward pass
      compute Δw_i for all weights from input layer to hidden layer   // backward pass continued
      update network weights // input layer not modified by error estimate
until all examples classified correctly or another stopping criterion satisfied
return the network
```

Report Spec

1. Introduction (20%)
2. Experiment setups (30%):

   A. Sigmoid functions

   B. Neural network

   C. Backpropagation
3. Results of your testing (20%)
   A. Screenshot and comparison figure
   B. Show the accuracy of your prediction
   C. Learning curve (loss, epoch curve)
   D. anything you want to present
4. Discussion (30%)
   A. Try different learning rates
   B. Try different numbers of hidden units
   C. Try without activation functions
   D. Anything you want to share
5. Extra (10%)
   A. Implement different optimizers. (2%)
   B. Implement different activation functions. (3%)
   C. Implement convolutional layers. (5%)

Score:
60% demo score (experimental results & questions) + 40% report
For experimental results, you have to achieve at least 90% of accuracy to get the demo score.
If the zip file name or the report spec have format error, you will be punished (-5)

Reference:

1. Logical regression:
   http://www.bogotobogo.com/python/scikit-learn/logistic_regression.php

2. Python tutorial:
   https://docs.python.org/3/tutorial/

3. Numpy tutorial:
   https://www.tutorialspoint.com/numpy/index.htm

4. Python Standard Library:
   https://docs.python.org/3/library/index.html

5. http://speech.ee.ntu.edu.tw/~tlkagk/courses/ML_2016/Lecture/BP.pdf
6. https://en.wikipedia.org/wiki/Sigmoid_function
7. https://en.wikipedia.org/wiki/Backpropagation