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.

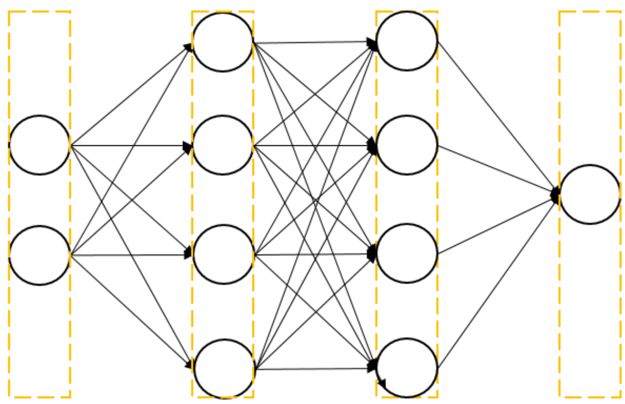


Figure 1. Two-layer neural network

Important Date:

1. Experiment Report Submission Deadline: 3/11 (Tue) 23:59

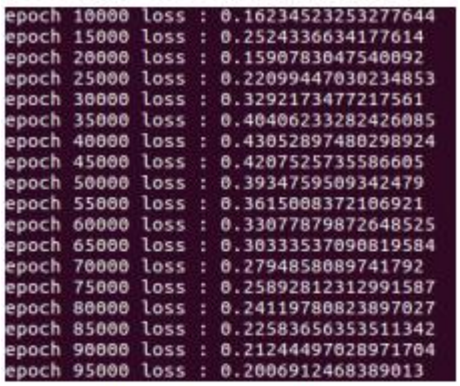
Turn in:

1. Experiment Report (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_陳敬中.zip」

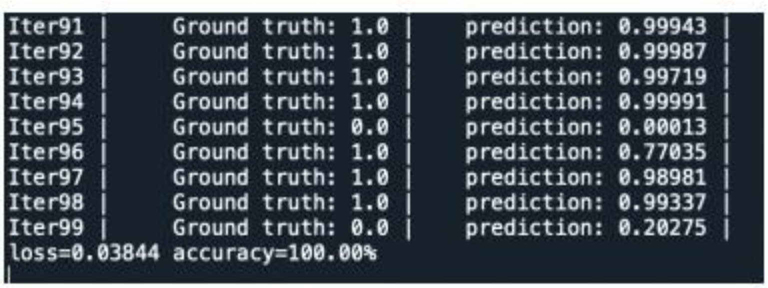
Requirements:

1. Implement simple neural networks with two hidden layers.
2. Each hidden layer needs to contain at least one transformation (CNN, Linear …) and one activate function (Sigmoid, tanh, relu…).
3. You must perform backpropagation on this neural network and can only use Numpy and other python standard libraries to implement.
4. Plot your comparison figures that illustrate the predicted results and the groundtruth.
5. Print the training loss and testing result as the figure listed 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 : 0.4207525735586605
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
```

Figure. a (training)



```
Iter91 |     Ground truth: 1.0 |     prediction: 0.99943 |
Iter92 |     Ground truth: 1.0 |     prediction: 0.99987 |
Iter93 |     Ground truth: 1.0 |     prediction: 0.99719 |
Iter94 |     Ground truth: 1.0 |     prediction: 0.99991 |
Iter95 |     Ground truth: 0.0 |     prediction: 0.00013 |
Iter96 |     Ground truth: 1.0 |     prediction: 0.77035 |
Iter97 |     Ground truth: 1.0 |     prediction: 0.98981 |
Iter98 |     Ground truth: 1.0 |     prediction: 0.99337 |
Iter99 |     Ground truth: 0.0 |     prediction: 0.20275 |
loss=0.03844 accuracy=100.00%
```

Figure. b (testing)

Implementation Details:

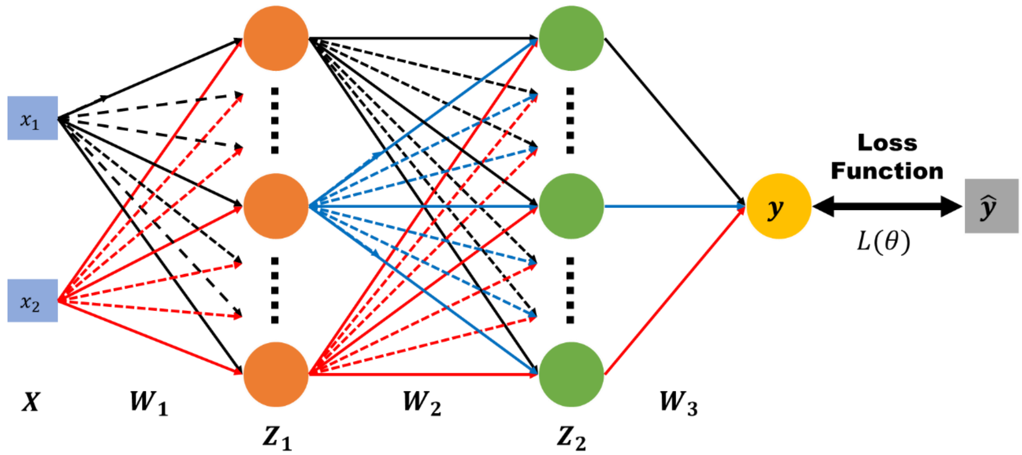


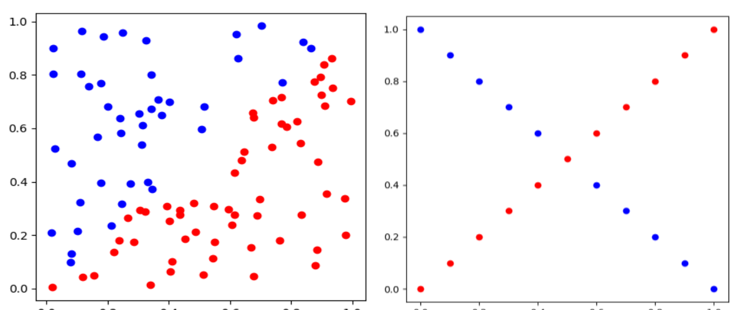
Figure 2. Forward pass

- 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. $L(\theta)$: $loss\ function$
  6. W_1, W_2, W_3 : $weight\ matrix\ of\ network\ layers$

- Here are the computations represented:
  $Z_1 = \sigma(XW_1)$ $\qquad$ $Z_2 = \sigma(Z_1W_2)$ $\qquad$ $y = \sigma(Z_2W_3)$

- In the equations, the σ is sigmoid function that refers to the special case of the **logistic** function and defined by the formula:
  $$\sigma(x) = \frac{1}{1+e^{-x}}$$

- **Input / Test:**
  There are two types of inputs as the following.



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

```python
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)
```

```python
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

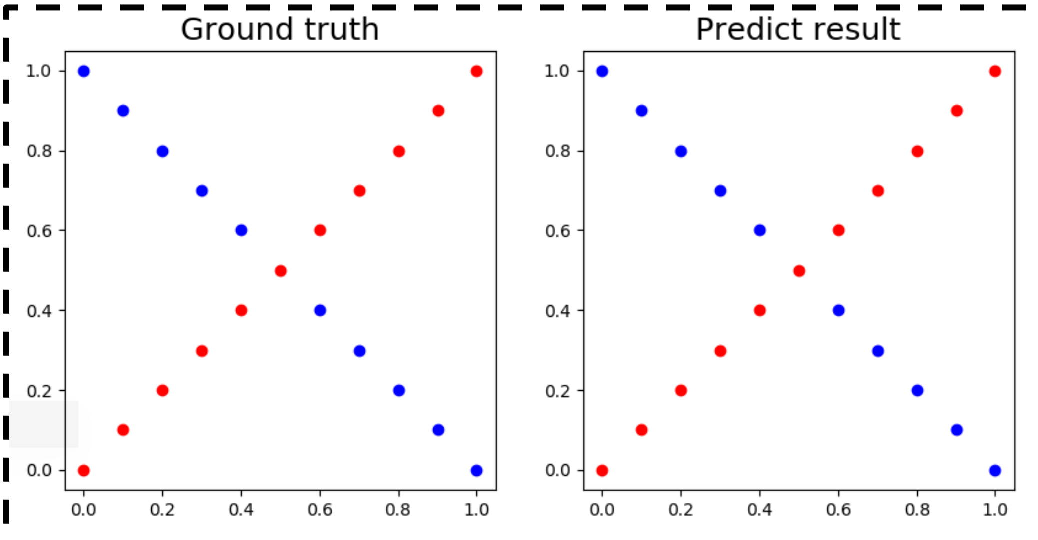
```python
x, y = generate_linear(n=100)
x, y = generate_XOR_easy()
```

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 : 0.4207525735586605
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
```

```
Iter91 |     Ground truth: 1.0 |     prediction: 0.99943 |
Iter92 |     Ground truth: 1.0 |     prediction: 0.99987 |
Iter93 |     Ground truth: 1.0 |     prediction: 0.99719 |
Iter94 |     Ground truth: 1.0 |     prediction: 0.99991 |
Iter95 |     Ground truth: 0.0 |     prediction: 0.00013 |
Iter96 |     Ground truth: 1.0 |     prediction: 0.77035 |
Iter97 |     Ground truth: 1.0 |     prediction: 0.98981 |
Iter98 |     Ground truth: 1.0 |     prediction: 0.99337 |
Iter99 |     Ground truth: 0.0 |     prediction: 0.20275 |
loss=0.03844 accuracy=100.00%
```

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

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

3. (hint) The derivative of sigmoid function

```
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 $L(\theta)$ (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 (5%)

   本實驗為使用 numpy 去實作一個簡單的 NN，包含 2 層 hidden layer，且必須實作 forward 及 backward propagation。

   問題分成兩種，一個是線性可分之 linear 問題、另一個是線性不可分之 XOR 問題

2. Implementation Details (15%):

   A. Sigmoid function

      sigmoid function 會將 input map 到 (-1, 1)之間
      sigmoid function = 1.0/(1.0+ e^-x)
      derivative sigmoid function = x(1-x)

   B. Neural network architecture

      兩層的 hidden layer，一開始會先初始化每層之 weight 和 bias，而 Loss function 使用 Binary cross-entropy(BCE)。
      為什麼不用 MSE？
      因 BCE 較 MSE 適合應用於二元分類問題。
      我的 NN output layer 的 activation function 是 sigmoid，代表 A3 的輸出介於(0,1)，表示屬於類別 1 之機率，而 MSE 適合用在輸出是連續值，不適合用在機率分布。
      且 MSE 對於 sigmoid 的梯度下降非常緩慢，Loss 的懲罰也較 BCE 小。

   C. Backpropagation

      主要透過以下四個公式計算出每層之 weight 與 bias 的 gradient

      BP(1) $$\delta^L = \nabla_a C \cdot \sigma'(z^L)$$

      BP(2) $$\delta^L = ((w^{L+1})^T \cdot \delta^{L+1}) \cdot \sigma'(z^L)$$

      BP(3) $$\partial C / \partial b^L = \delta^L$$

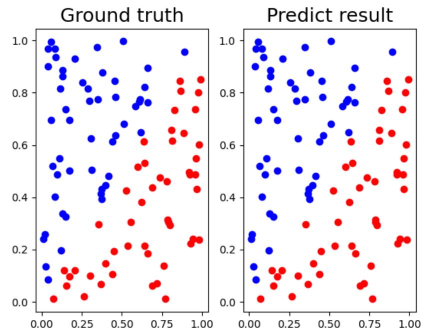
      BP(4) $$\partial C / \partial w^L = a^{L-1} \cdot \delta^L$$

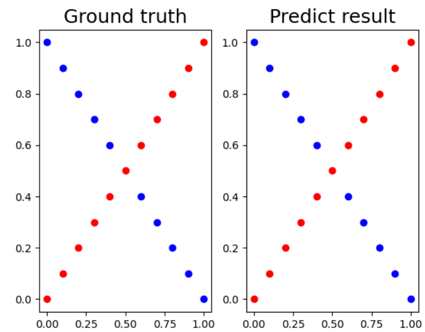
      Backpropagation 演算法四大公式

      Reference: https://datasciocean.tech/deep-learning-core-concept/backpropagation-explain/#google_vignette
      並使用 SGD 去更新權重，表示美進行一次 forward 與 backward 就更新一次權重。。

3. Experimental Results (45%)

A. Screenshot and comparison figure





B. Show the accuracy of your prediction (40%)
(achieve 90% accuracy)

Linear:

```
Epoch 0, Loss: 2.7133
Epoch 500, Loss: 0.2232
Epoch 1000, Loss: 0.1473
Epoch 1500, Loss: 0.1127
Epoch 2000, Loss: 0.0925
Epoch 2500, Loss: 0.0790
Epoch 3000, Loss: 0.0692
Epoch 3500, Loss: 0.0618
Epoch 4000, Loss: 0.0560
Epoch 4500, Loss: 0.0512
```

```
Iter96 | Ground truth: 1.0 | prediction: 0.88449
Iter97 | Ground truth: 0.0 | prediction: 0.00153
Iter98 | Ground truth: 1.0 | prediction: 0.99998
Iter99 | Ground truth: 0.0 | prediction: 0.30631
loss=3.30300 accuracy=100.00%
```

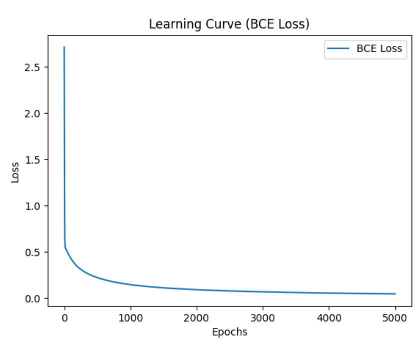
XOR:

```
Epoch 0, Loss: 2.9241
Epoch 500, Loss: 0.6041
Epoch 1000, Loss: 0.4958
Epoch 1500, Loss: 0.3917
Epoch 2000, Loss: 0.3102
Epoch 2500, Loss: 0.2519
Epoch 3000, Loss: 0.2105
Epoch 3500, Loss: 0.1799
Epoch 4000, Loss: 0.1564
Epoch 4500, Loss: 0.1376
```

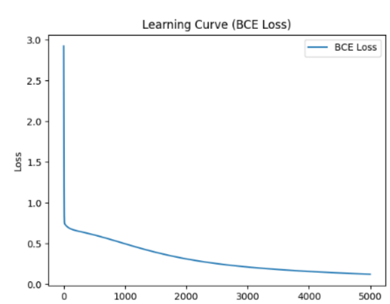
```
Iter17 | Ground truth: 0.0 | prediction: 0.01900
Iter18 | Ground truth: 1.0 | prediction: 0.99266
Iter19 | Ground truth: 0.0 | prediction: 0.01383
Iter20 | Ground truth: 1.0 | prediction: 0.99630
loss=0.12203 accuracy=100.00%
```

C. Learning curve (loss-epoch curve)

Linear:



XOR:



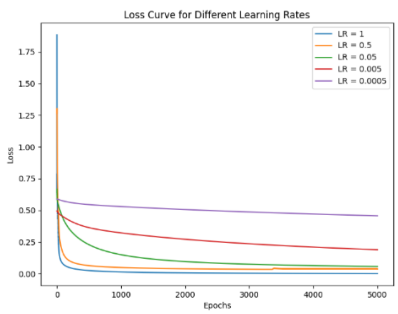
觀察此兩張 Learning curve 可以發現，Linear 的初始 Loss 比較小，且 Loss 下降速度較快，並且發現 XOR 的 Learning curve 在 Epoch=5000 時還在下降，因此可以在 XOR problem 增加 Epochs 或 hidden layer 或 Learning rate 或 hidden units。

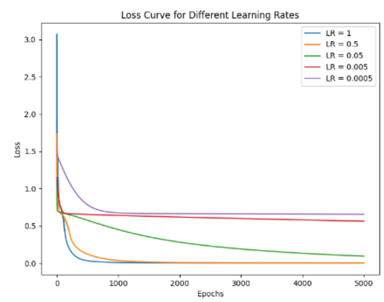
D. Anything you want to present

4. Discussion (15%)

A. Try different learning rates

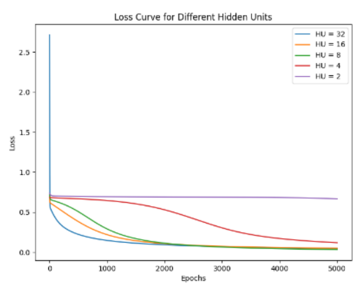
Linear: XOR:

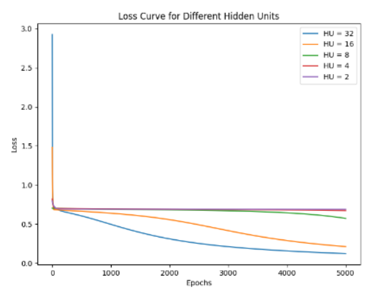




B. Try different numbers of hidden units

Linear: XOR:





C. Try without activation functions

D. Anything you want to share

5. Questions (20%)

A. What is the purpose of activation functions? (6%)

目的是為了引入非線性，使 NN 能夠學習到更複雜的關係。

B. What might happen if the learning rate is too large or too small? (7%)

太大的話：可能會發散，使 Loss 值不斷增加，導致訓練失敗；抑或是導致 Loss 無法收斂，模型的權重過度更新，導致 Loss 值不斷震盪。

太小的話：可能導致學習效率過慢，模型需要花很長的時間學習；也可能會卡在 local minima，無法找到 global minima。

C. What is the purpose of weights and biases in a neural network? (7%)

1. Weight:代表著 Neuron 之間的連結強度，決定了一個 Neuron 對另一個 Neuron 的影響程度。
2. Bias:讓 Neuron 可以調整 activation function 的輸出範圍，即使所有輸入都是 0，也仍然可以激活 Neuron。

6. Extra (10%)

A. Implement different optimizers. (2%)

B. Implement different activation functions. (3%)

C. Implement convolutional layers. (5%)

Score:

If there are any format errors, 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
8. https://www.geeksforgeeks.org/activation-functions-neural-networks/