DLP Homework 1

電控碩 0860077 王國倫

1. Introduction

This lab will implement a two hidden layers neural network shown as Fig 1, and only use Numpy and other standard libraries, the deep learning framework is not allowed to use in this homework.

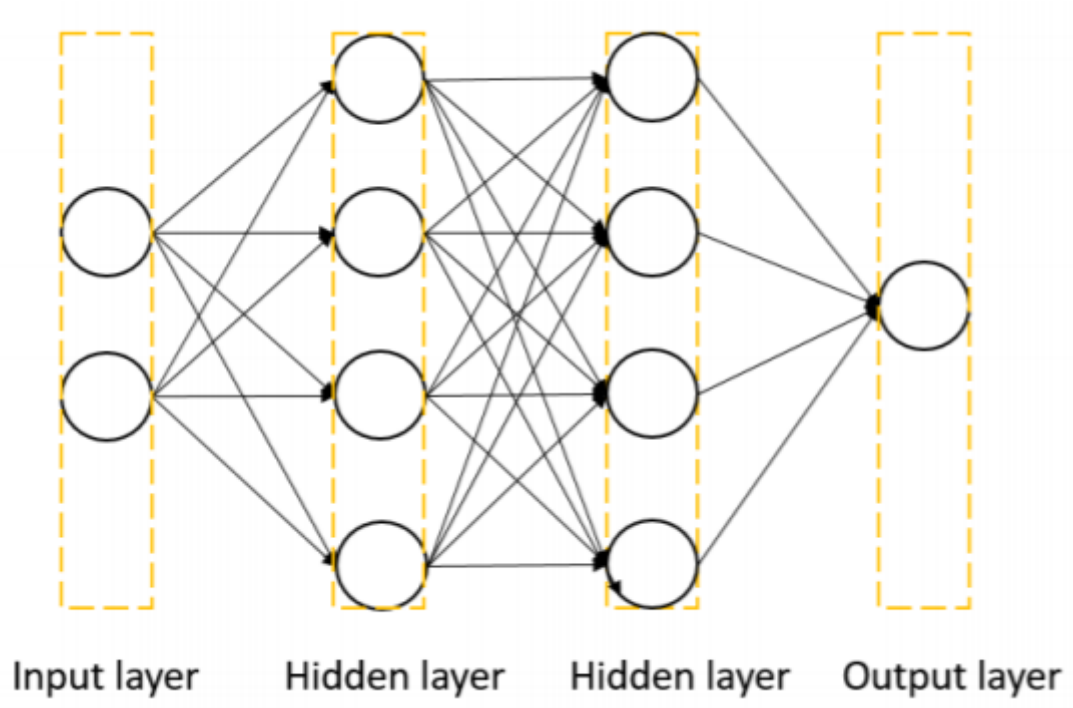


Fig. 1 Two layer network

Iuput data can divided into two parts, one is linear dataset, and the other one is nonlinear dataset, see the Fig. 2.

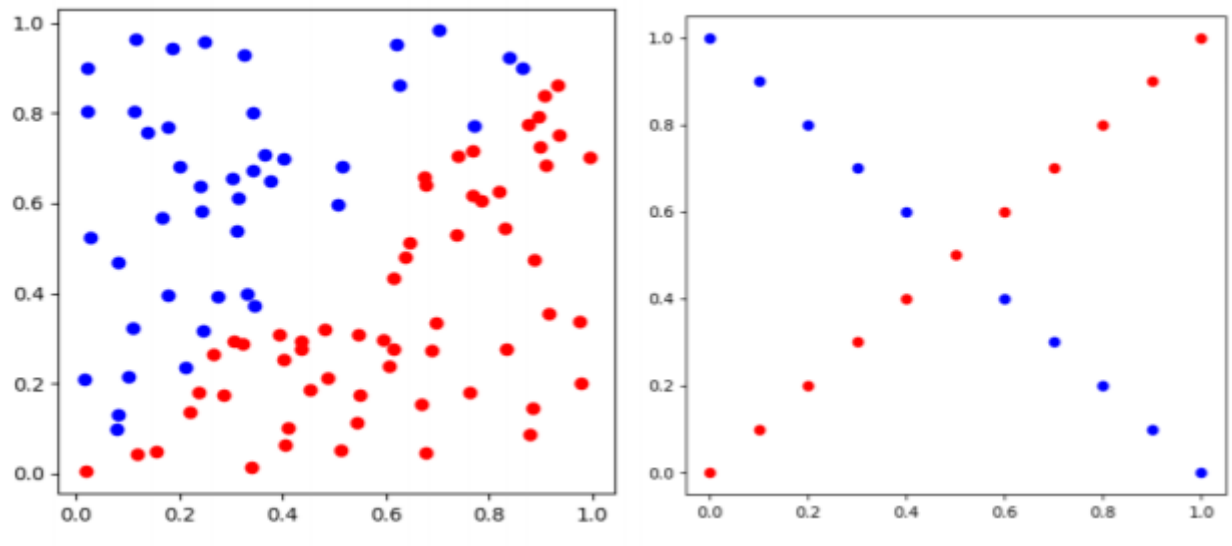


Fig.2 The left-hand side is linear, the right-hand side is nonlinear.

> linear data

```
def generate_linear(n=100):

  point = np.random.uniform(0, 1, (n, 2))
  input = []
  label = []
  for p in point:
    input.append([p[0], p[1]])
    if p[0] > p[1]:
      label.append(0)
    else:
      label.append(1)
  return np.array(input), np.array(label).reshape(n, 1)
```

> nonlinear data

```
def generate_XOR_easy():
  input = []
  label = []

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

    if 0.1 * i == 0.5:
      continue

    input.append([0.1 * i, 1 - 0.1 * i])
    label.append(1)

  return np.array(input), np.array(label).reshape(21, 1)
```

2. Experiment setups

A. Sigmoid functions

The sigmoid functions as activate function on neural network, this function can import generaliza-tion of model to conquer nonlinear problems, likes XOR. The derivation formula are shown as Fig 3, and the Graph of Sigmoid and the derivative of the Sigmoid function can see Fig 4.

$$
\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{(1+e^{-x})-1}{(1+e^{-x})^2} \\
&= \frac{1+e^{-x}}{(1+e^{-x})^2} - \left(\frac{1}{1+e^{-x}}\right)^2 \\
&= \sigma(x) - \sigma(x)^2 \\
\sigma' &= \sigma(1-\sigma)
\end{aligned}
$$

Fig. 3 Sigmoid derivative

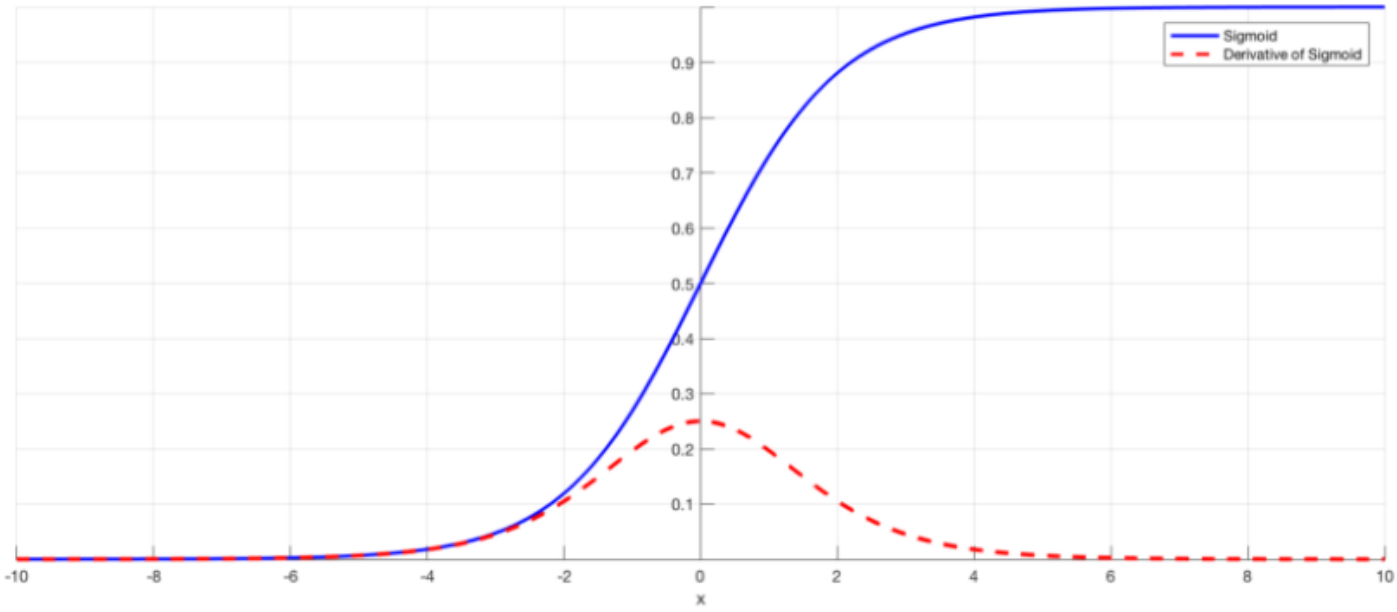


Fig. 4 Sigmoid function and it derivation

B. Neural network

This lab neural network architecture can follow Fig 5.

- x_1 and x_2 are input data
- X is a matrix $m * 2$
- W_1, W_2 and W_2 are model weigtht
- y is predict result, matrix $m * 1$
- $\hat{y}$ is ground truth
- $L(\theta)$ is loss function, this lab I use **MSE**

The hidden layer and predict computations.

$$Z_1 = \sigma(XW_1),\ Z_2 = \sigma(Z_1W_2),\ y = \sigma(Z_2W_3)$$

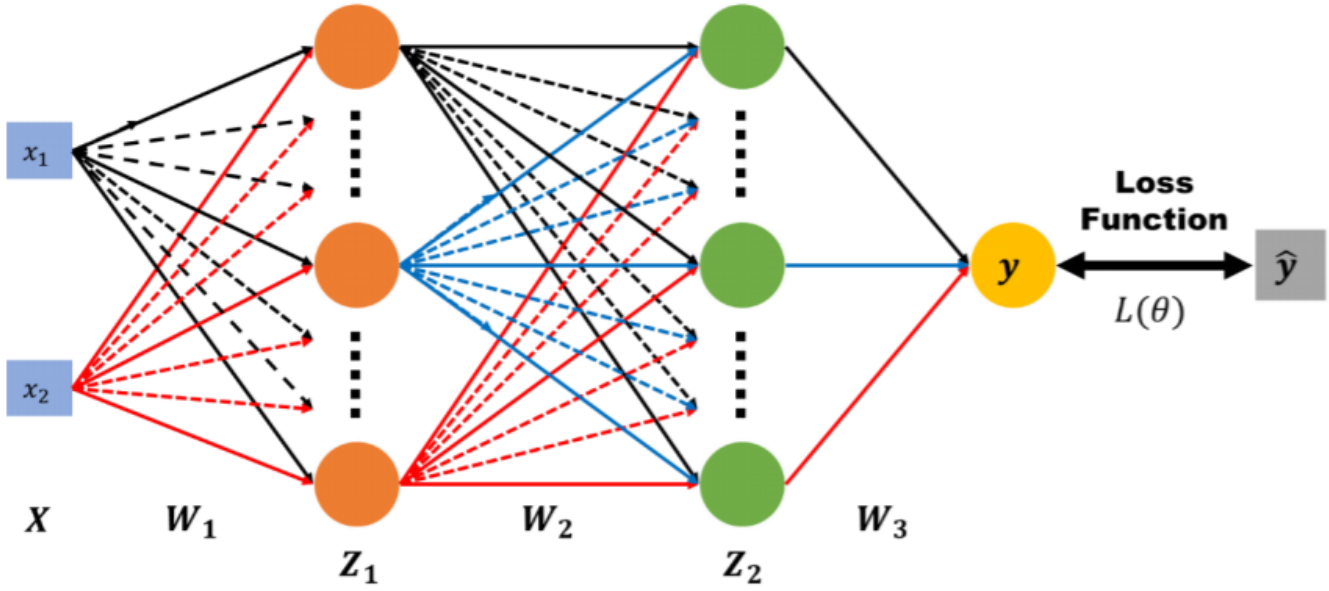


Fig. 5 Forward

C. Backpropagation

In order to derivate the back-propagation, I will take Fig 5 for example.

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial y'} \frac{\partial y'}{\partial W^3} = Z_2^T \cdot \sigma'(y) \circ 2(y - \hat{y})$$

$$y' = Z_2 W_3$$

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial y'} \frac{\partial y'}{\partial Z_2} \frac{\partial Z_2}{\partial Z_2'} \frac{\partial Z_2'}{\partial W^2} = Z_1^T \cdot \sigma'(Z_2) \circ (\sigma'(y) \circ 2(y - \hat{y}) \cdot W_3^T)$$

$$Z_2' = Z_1 W_2$$

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial y'} \frac{\partial y'}{\partial Z_2} \frac{\partial Z_2}{\partial Z_2'} \frac{\partial Z_2'}{\partial Z_1} \frac{\partial Z_1}{\partial Z_1'} \frac{\partial Z_1'}{\partial W^1} = X^T \cdot \sigma'(Z_1) \circ ((\sigma'(Z_2) \circ (\sigma'(y) \circ 2(y - \hat{y}) \cdot W_3^T)) \cdot W_2^T)$$

$$Z_1' = X W_1$$

operator priority: $\circ > \cdot$

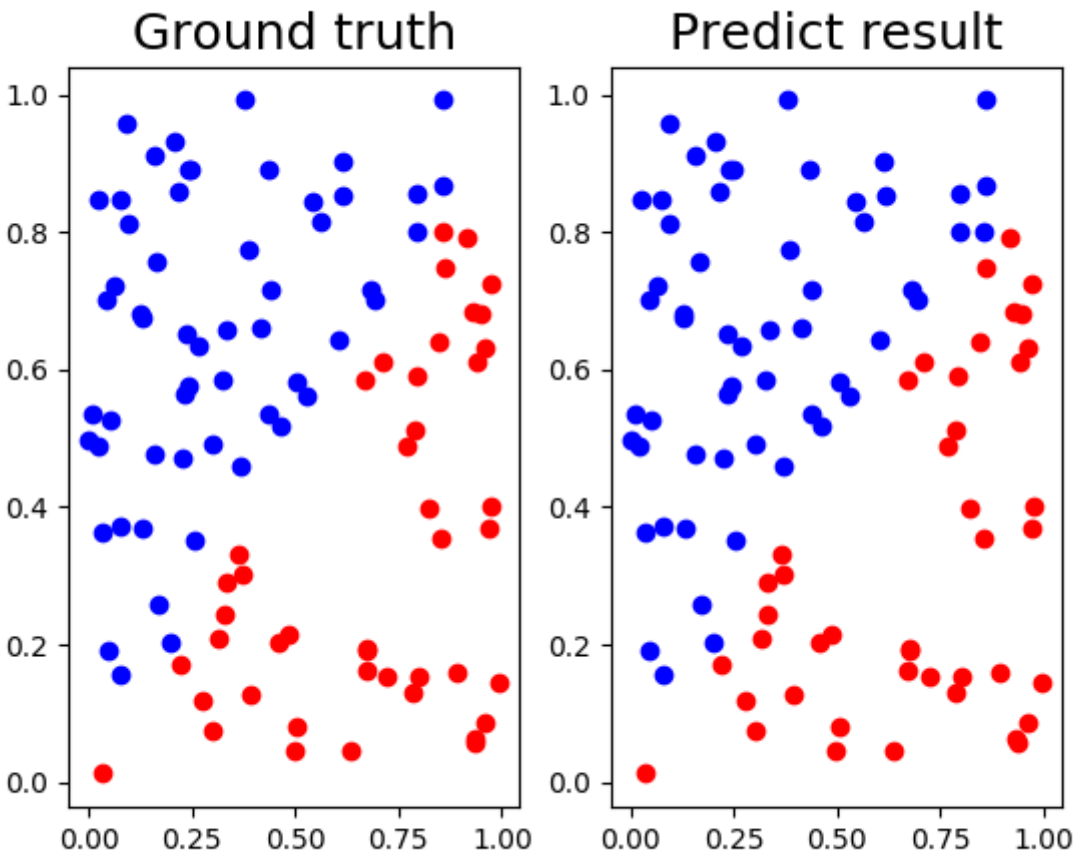
The $\circ$ operation is element-wise product (Hadamard product) in the same dimension.
For example, the Hadamard product for a 3 × 3 matrix A with a 3 × 3 matrix B is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11} b_{11} & a_{12} b_{12} & a_{13} b_{13} \\ a_{21} b_{21} & a_{22} b_{22} & a_{23} b_{23} \\ a_{31} b_{31} & a_{32} b_{32} & a_{33} b_{33} \end{bmatrix}$$

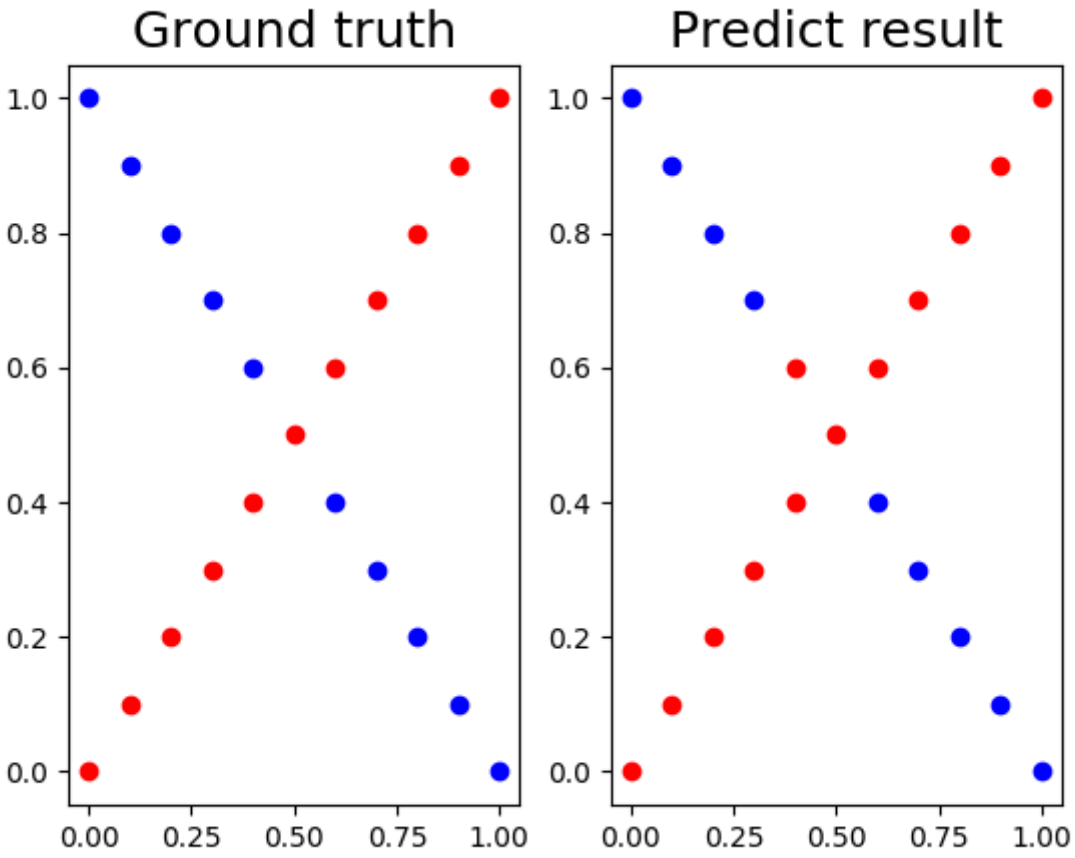
3. Results of your testing

A. Screenshot and comparison figure

1. linear data

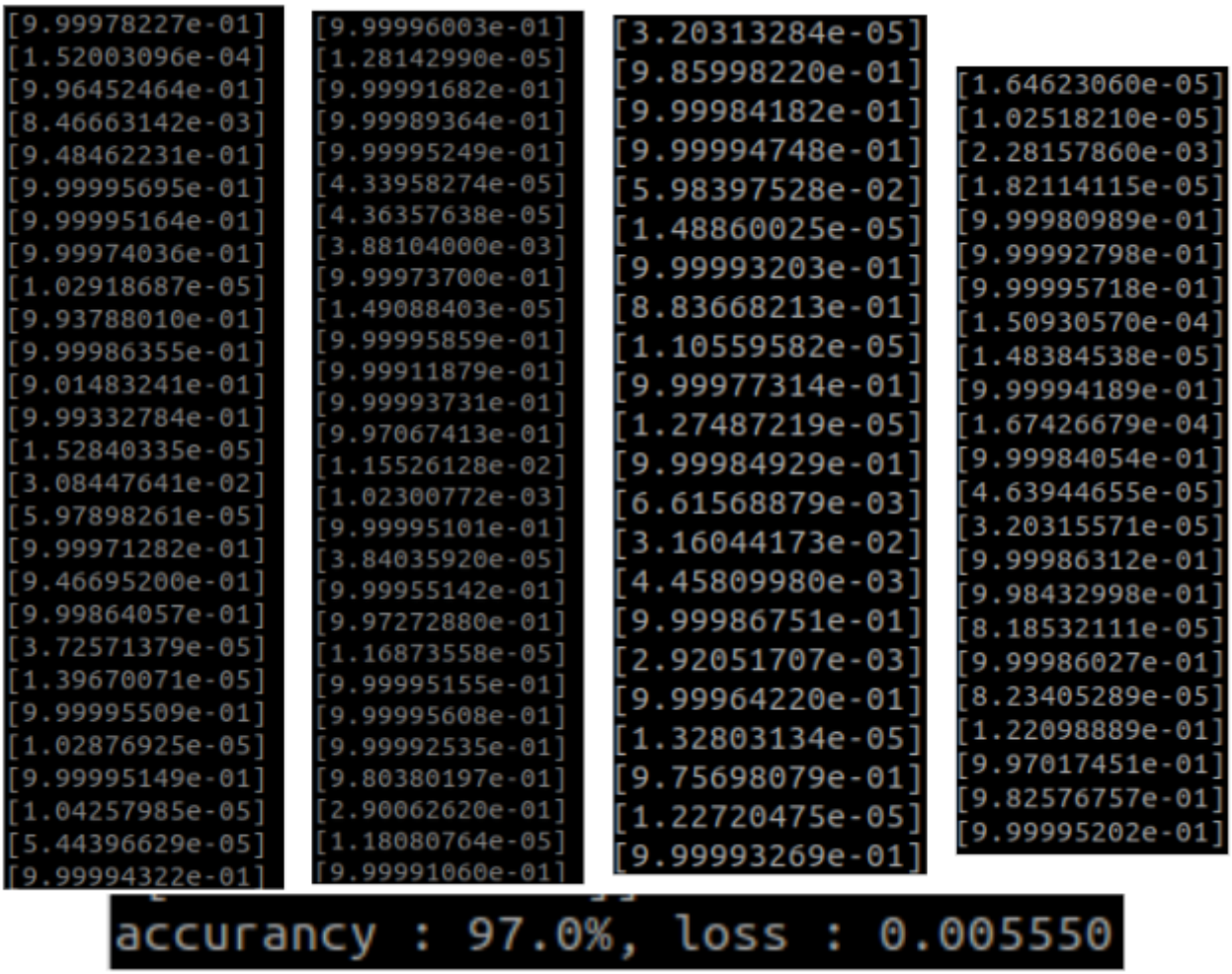


2. nonlinear data

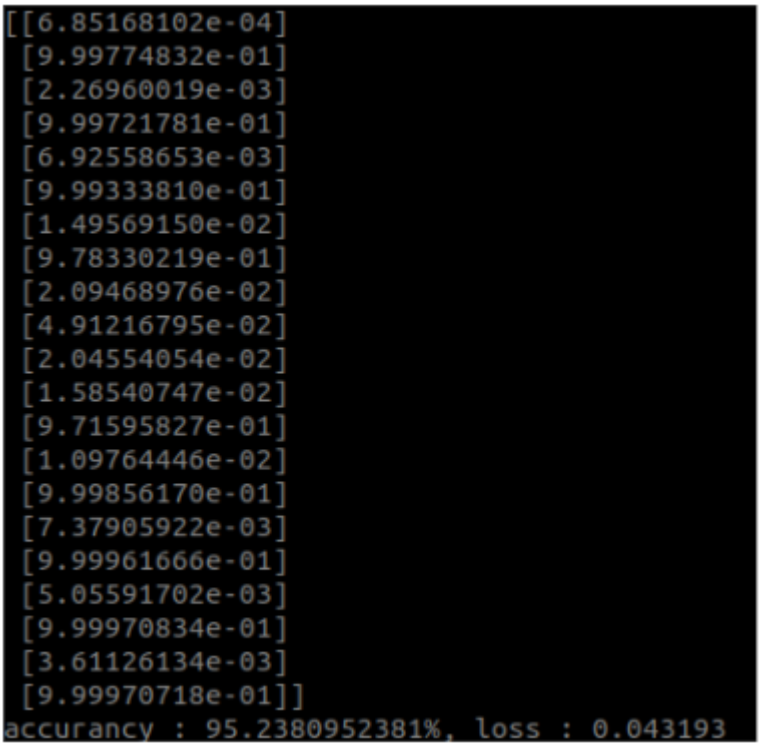


B. Show the accuracy of your prediction

1. linear data

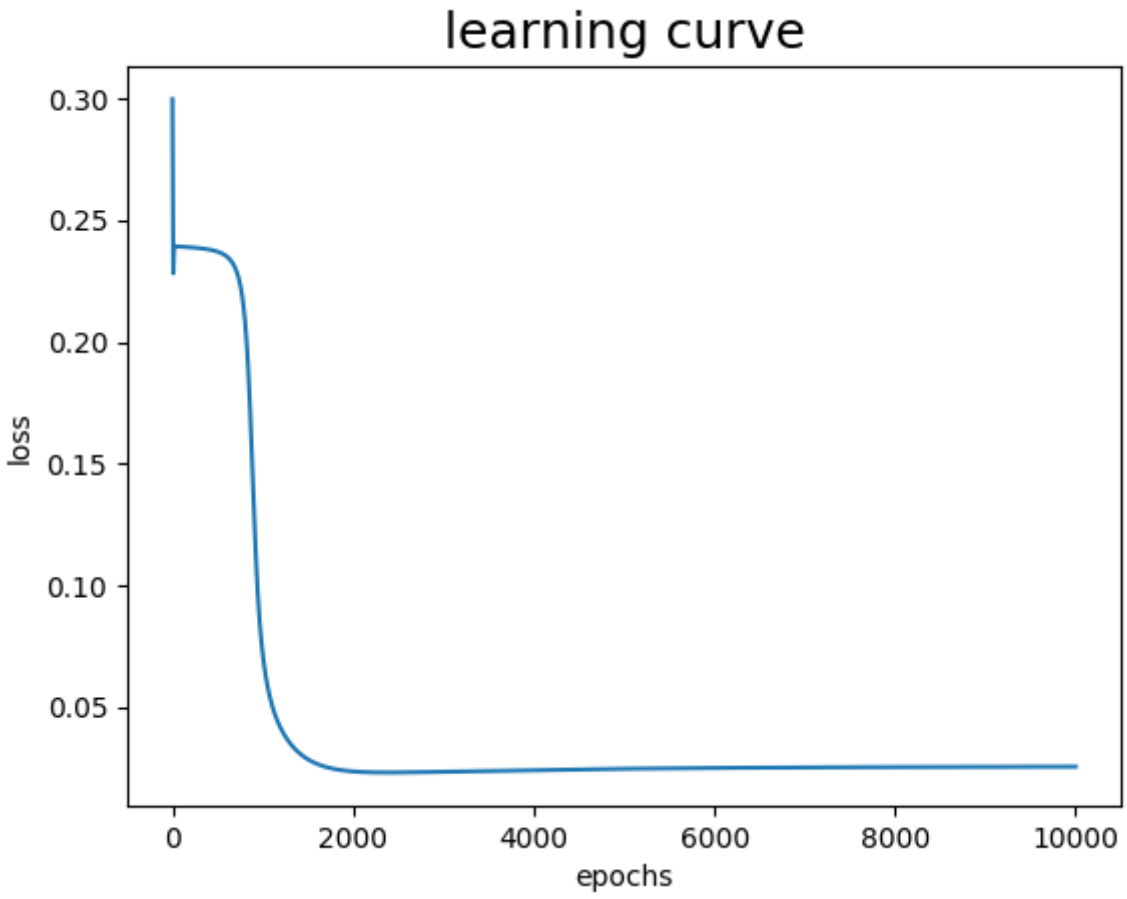


2. nonlinear data

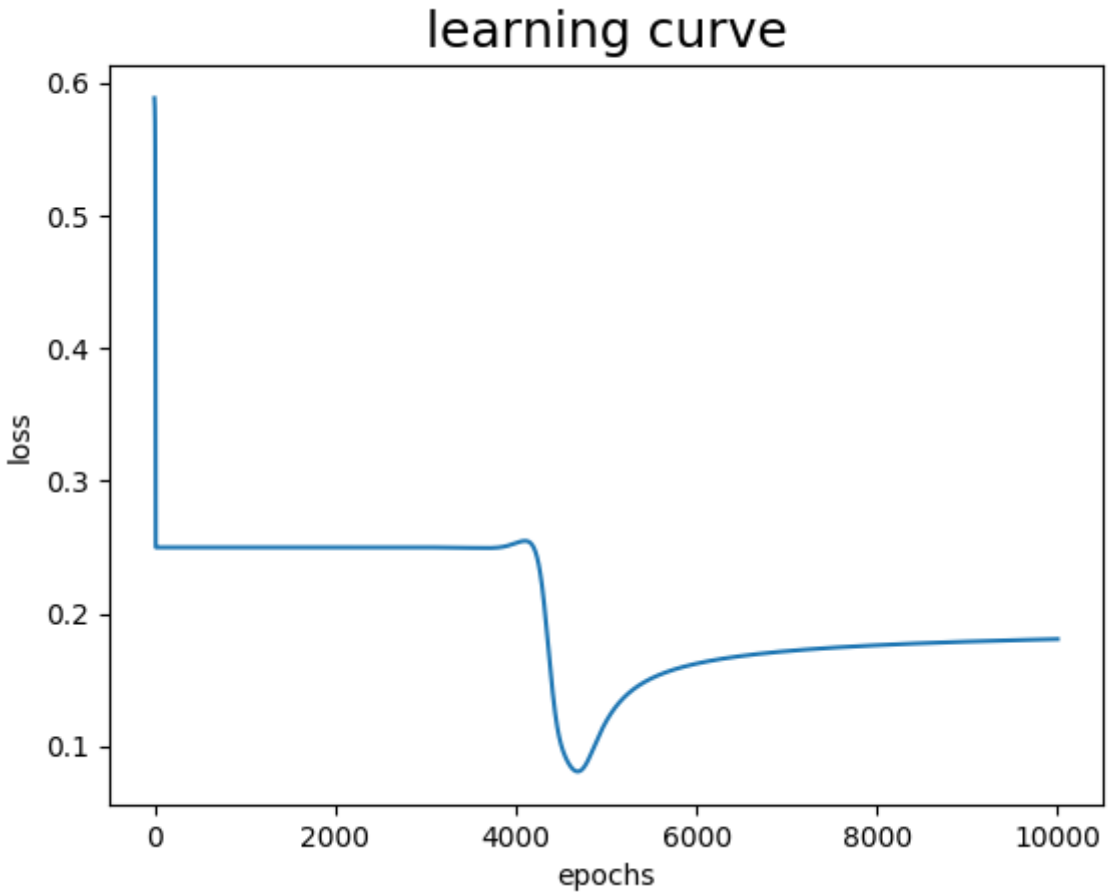


C. Learning curve (loss, epoch curve)

1. linear data



2. nonlinear data



D. anything you want to present

(1) The network and training parameter:

1. linear data
    - hidden layer1 : 5
    - hidden layer2 : 5
    - activate function : sigmoid
    - learning rate : 0.01
    - optimizer : SGD
    - epochs : 10000
    - batch_size : 10
2. nonlinear data
    - hidden layer1 : 6
    - hidden layer2 : 8
    - activate function : sigmoid
    - learning rate : 0.1
    - optimizer : SGD
    - epochs : 10000
    - batch_size : 10

(2) Training, validation, test: I split data into two parts, one is training data, and the other one is validation data, the ratio is 8:2. The whole data will be tested after training model. The training data in charge of train and update parameter, the validation is calculate loss, the test data is predict final result and accurancy.

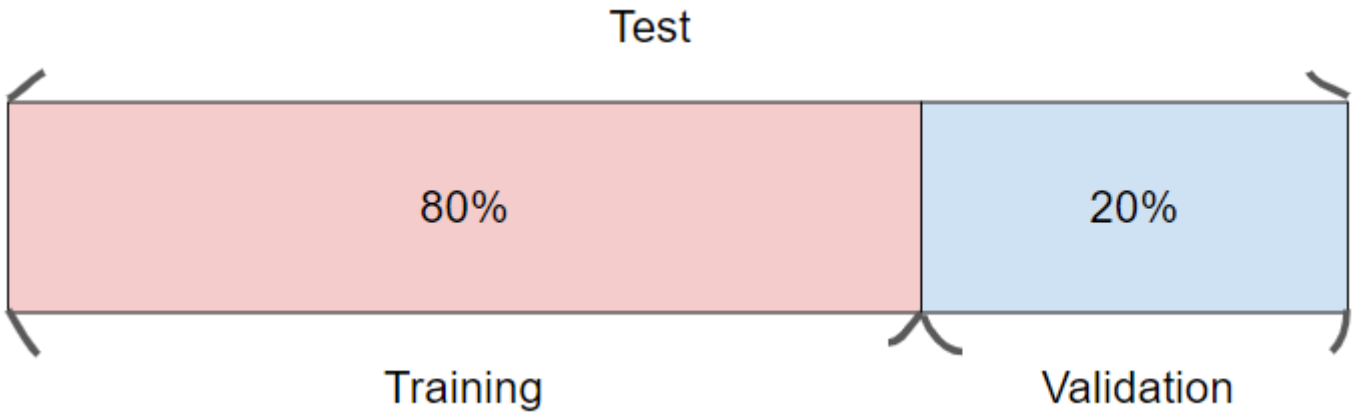


Fig. 6 Training, validation, test dataset

```
    def split_data(self, x, y):

      size = (int)(x.shape[0] * 0.8)
      index = np.array([t for t in range(x.shape[0])])
      np.random.shuffle(index)

      tra_x, val_x = x[index[:size]], x[index[size:]]
      tra_y, val_y = y[index[:size]], y[index[size:]]

      return tra_x, val_x, tra_y, val_y
```

(3) Accurancy: The accurancy use difference of prediction and actual, and it error must less than 0.1, then are considered a successful prediction.

```
  def evaluate(self, prediction, actual):

      diff = np.abs(prediction - actual)
      result = np.zeros(prediction.shape)
      result[diff<=0.1] = 1
      accurancy = np.sum(result == 1) / (float)(actual.shape[0]) * 100

      print("accurancy : {0}%, loss : {1:.6f}".format(accurancy,self.loss(prediction, actual)))
```

4. Discussion

A. Try different learning rates

1. linear data I fixed other parameters, adjust learning rate 1,0.1,0.01.
    - hidden layer1 : 5
    - hidden layer2 : 5
    - activate function : sigmoid
    - optimizer : SGD
    - epochs : 10000
    - batch_size : 10

learning rate	**1**	**0.1**	**0.01**
accurancy	100%	90.48%	97%

2. nonlinear data I fixed other parameters, adjust learning rate 0.2,0.1,0.05.
    - hidden layer1 : 6
    - hidden layer2 : 8
    - activate function : sigmoid
    - optimizer : SGD
    - epochs : 10000
    - batch_size : 10

learning rate	**0.2**	**0.1**	**0.05**
accurancy	90.48%	90.48%	95.24%

B. Try different numbers of hidden units

1. linear data I fixed other parameters, adjust hidden layer1 and layer2.
    - learning rate : 1
    - activate function : sigmoid
    - optimizer : SGD
    - epochs : 10000
    - batch_size : 10

hidden layer	**(5,5)**	**(3,4)**	**(3,3)**
accurancy	100%	99%	49%

C. Try without activation functions

1. linear data I fixed other parameters, and without activate function.
    - hidden layer1 : 5

- hidden layer2 : 3
- learning rate : 0.01
- activate function : none
- optimizer : SGD
- epochs : 10000
- batch_size : 10

```
accurancy : 21.0%, loss : 0.097813
```

2. nonlinear data I fixed other parameters, and without activate function.

- hidden layer1 : 3
- hidden layer2 : 4
- learning rate : 0.01
- activate function : none
- optimizer : SGD
- epochs : 10000
- batch_size : 10

```
accurancy : 9.52380952381%, loss : 0.291206
```

D. Anything you want to share

(1) argparse: In order to easy adjust parameters, I use argparse library to control train and model parameters, then I can use terminal to modify instead of manual adjust code.

```
def create_argparse():

    parser = argparse.ArgumentParser(prog="DLP homework 1", description='This lab implement two

    # model parameters
    parser.add_argument("--input_size", type=int, default=2, help="An integer giving the size of
    parser.add_argument("--hidden_layer1", type=int, default=5, help="An integer giving the size
    parser.add_argument("--hidden_layer2", type=int, default=5, help="An integer giving the size
    parser.add_argument("--output_size", type=int, default=1, help="An integer giving the size o
    parser.add_argument("--activate_function", type=str, default="sigmoid", help="activation fun

    # training hyper parameters
    parser.add_argument("--learning_rate", type=float, default=0.01, help="Learning rate for the
    parser.add_argument("--optimizer", type=str, default="SGD", help="optimizer(SGD, momentum, a
    parser.add_argument("--epochs", type=int, default=10000, help="The number of the training, d
    parser.add_argument("--batch_size", type=int, default=10, help="Input batch size for testing

    # other parameters
    parser.add_argument("file_name", type=str, default="linear", help="save result picture with
    parser.add_argument("data", type=str, default="linear", help="select input data(linear or XO
```

```
    return parser
```

5. Extra

A. Implement different optimizers

In this lab, I implement momentum and adam optimizer, the momentum formula shown as following, where β is hyperparameter, Typically this value is set to $\mathbf{0.9}$, η is learning rate, θ is model weight.
The adam optimizer formular are list below, where β_1 is the exponential decay of the rate for the first moment estimates, it value set 0.9, β_2 is the exponential decay rate for the second-moment estimates, and its literature value is 0.95, small value ε to prevent zero-division.

> Momentum

$$m \leftarrow \beta m + \eta \bigtriangledown MSE$$
$$\theta \leftarrow \theta - m$$

> Adam

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \bigtriangledown MSE$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \bigtriangledown MSE^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

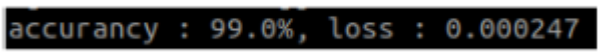
$$\theta \leftarrow \theta - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

```
# momentum optimizer parameter
self.momentum = 0.9
self.vec_W1 = 0
self.vec_W2 = 0
self.vec_W3 = 0

# adam optimizer parameter
self.momentum_decay = 0.9
self.scale_decay = 0.95
self.epsilon = 10 ** -8
self.sca_W1 = 0
self.sca_W2 = 0
self.sca_W3 = 0
```

```
# momentum update weight
self.vec_W1 = self.momentum * self.vec_W1 + self.learning_rate * self.grad_W1
self.vec_W2 = self.momentum * self.vec_W2 + self.learning_rate * self.grad_W2
self.vec_W3 = self.momentum * self.vec_W3 + self.learning_rate * self.grad_W3

self.W1 -= self.vec_W1
self.W2 -= self.vec_W2
self.W3 -= self.vec_W3



# adam update weight
self.vec_W1 = self.momentum_decay * self.vec_W1 + (1 - self.momentum_decay) * self.grad_W1
self.vec_W2 = self.momentum_decay * self.vec_W2 + (1 - self.momentum_decay) * self.grad_W2
self.vec_W3 = self.momentum_decay * self.vec_W3 + (1 - self.momentum_decay) * self.grad_W3

self.vec_W1 = self.vec_W1 / (1 - self.momentum_decay ** epochs)
self.vec_W2 = self.vec_W2 / (1 - self.momentum_decay ** epochs)
self.vec_W3 = self.vec_W3 / (1 - self.momentum_decay ** epochs)

self.sca_W1 = self.scale_decay * self.sca_W1 + (1 - self.scale_decay) * self.grad_W1 ** 2
self.sca_W2 = self.scale_decay * self.sca_W2 + (1 - self.scale_decay) * self.grad_W2 ** 2
self.sca_W3 = self.scale_decay * self.sca_W3 + (1 - self.scale_decay) * self.grad_W3 ** 2

self.sca_W1 = self.sca_W1 / (1 - self.scale_decay ** epochs)
self.sca_W2 = self.sca_W2 / (1 - self.scale_decay ** epochs)
self.sca_W3 = self.sca_W3 / (1 - self.scale_decay ** epochs)

self.W1 -= self.learning_rate * self.vec_W1 / (np.sqrt(self.sca_W1) + self.epsilon)
self.W2 -= self.learning_rate * self.vec_W2 / (np.sqrt(self.sca_W2) + self.epsilon)
self.W3 -= self.learning_rate * self.vec_W3 / (np.sqrt(self.sca_W3) + self.epsilon)
```

1. linear data I use momentum to optimize linear data.

    - hidden layer1 : 4
    - hidden layer2 : 6
    - learning rate : 0.01
    - activate function : sigmoid
    - optimizer : momentum
    - epochs : 10000
    - batch_size : 10

2. nonlinear data I use adam to optimize nonlinear data.

    - hidden layer1 : 4
    - hidden layer2 : 3

- learning rate : 0.01
- activate function : sigmoid
- optimizer : adam
- epochs : 10000
- batch_size : 21

```
accurancy : 100.0%, loss : 0.000000
```

B. Implement different activation functions

In this lab, I use three activation functions, include ReLU, Leaky ReLU, Tanh. There are shown as below.

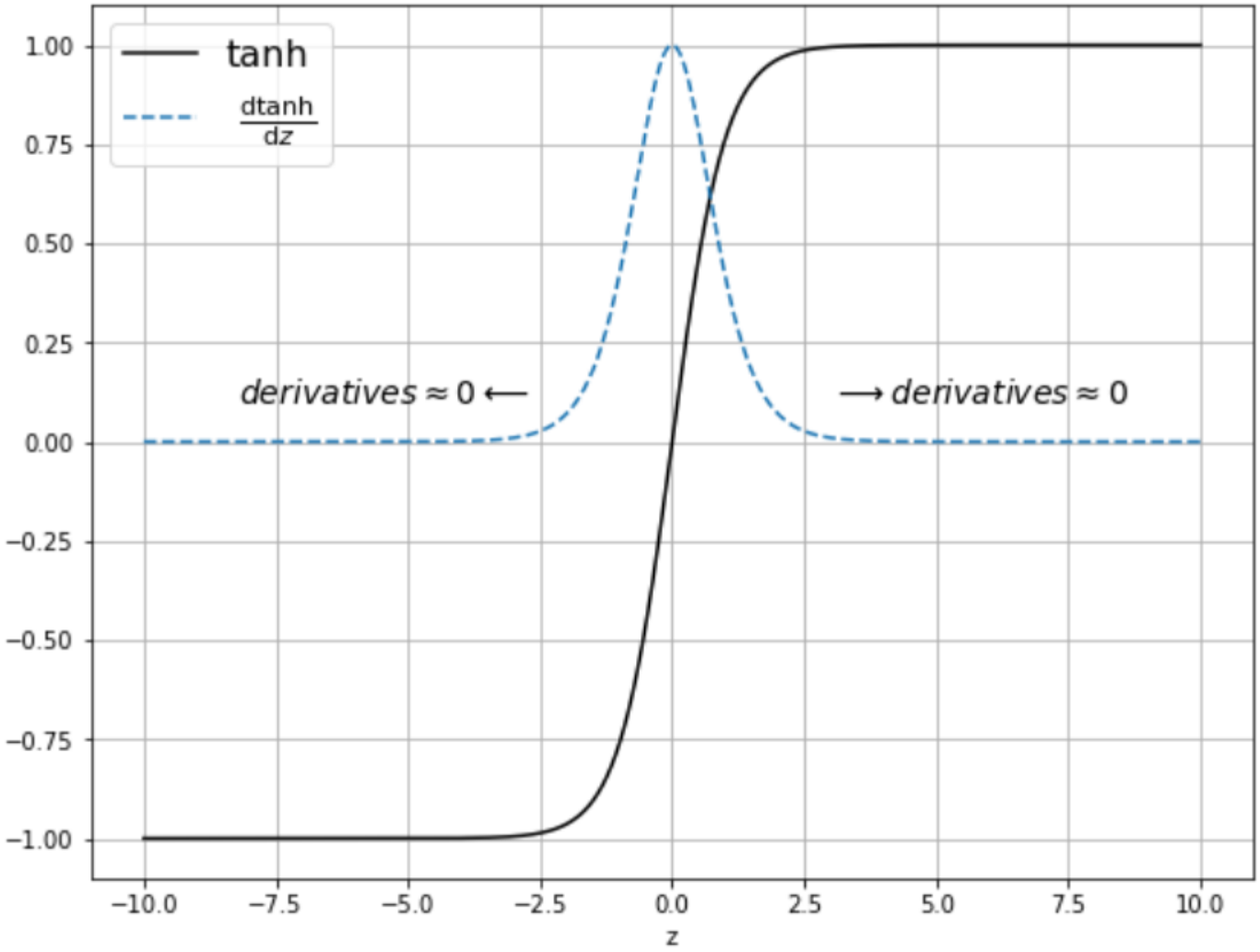
> Tanh

$$g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\frac{d}{dz} g(z) = slope\ of\ g(z)\ at\ z$$

$$\frac{d}{dz} g(z) = \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} = \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$

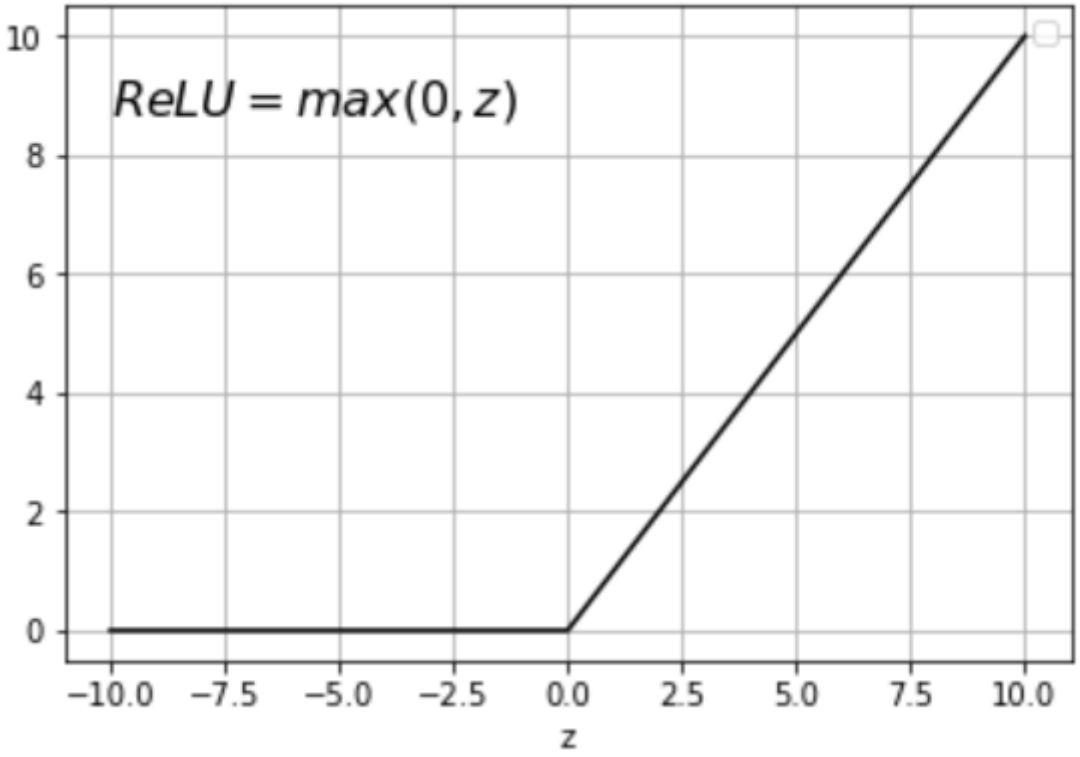
$$\frac{d}{dz} g(z) = \frac{\frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2}}{\frac{(e^z + e^{-z})^2}{(e^z + e^{-z})^2}} = \frac{\frac{1 - tanh(z)^2}{1}}{1} = 1 - tanh(z)^2$$

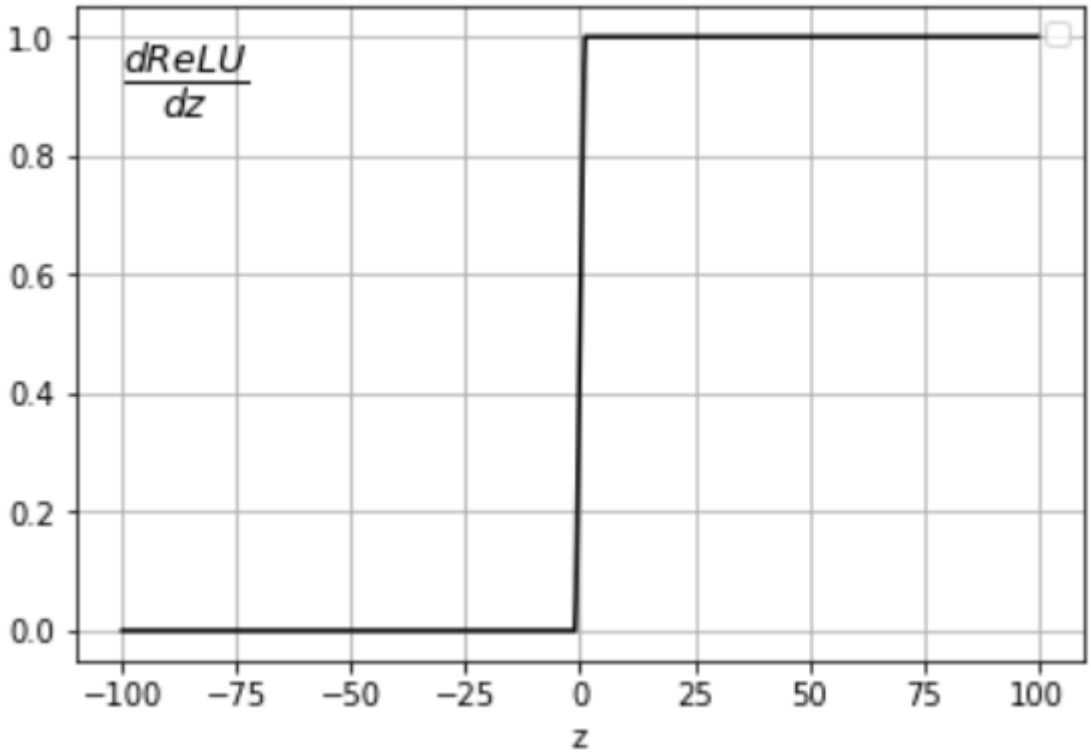


> ReLU

$$g(z) = max(0, z)$$

$$g'(z) = \begin{cases} 1 \ \ if \ \ z > 0 \\ 0 \ \ if \ \ z < 0 \\ undefined \ \ if \ \ z = 0 \end{cases}$$

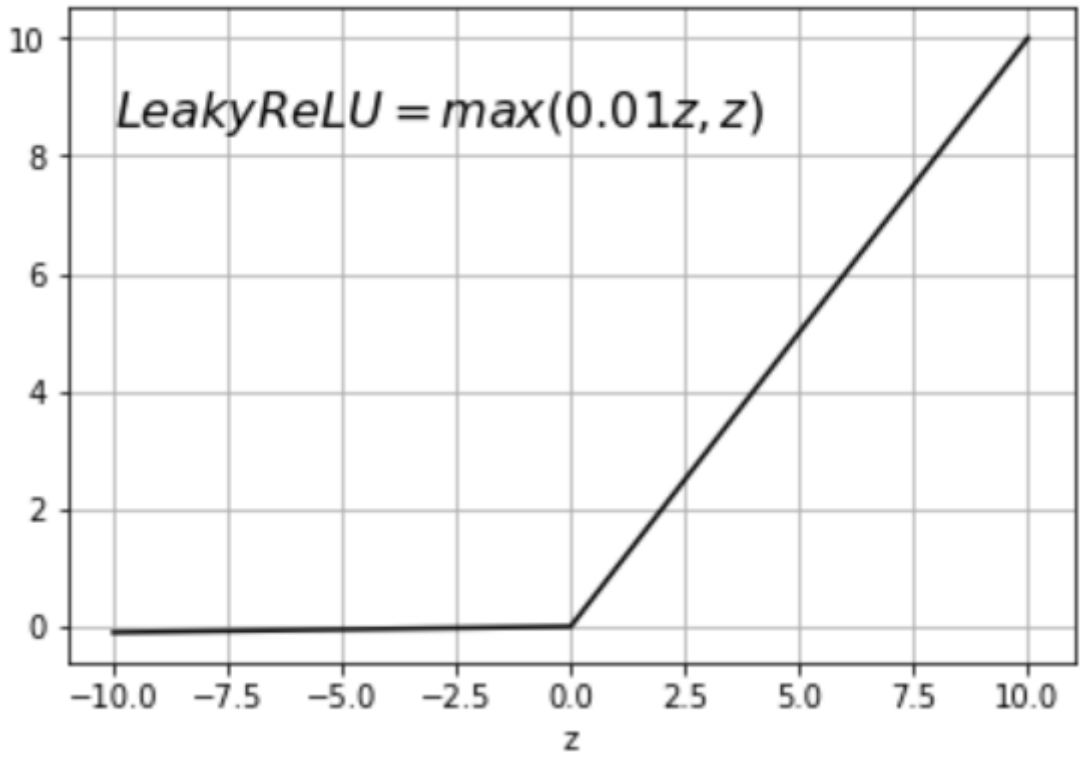

$ReLU = max(0, z)$
10
8
6
4
2
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
z

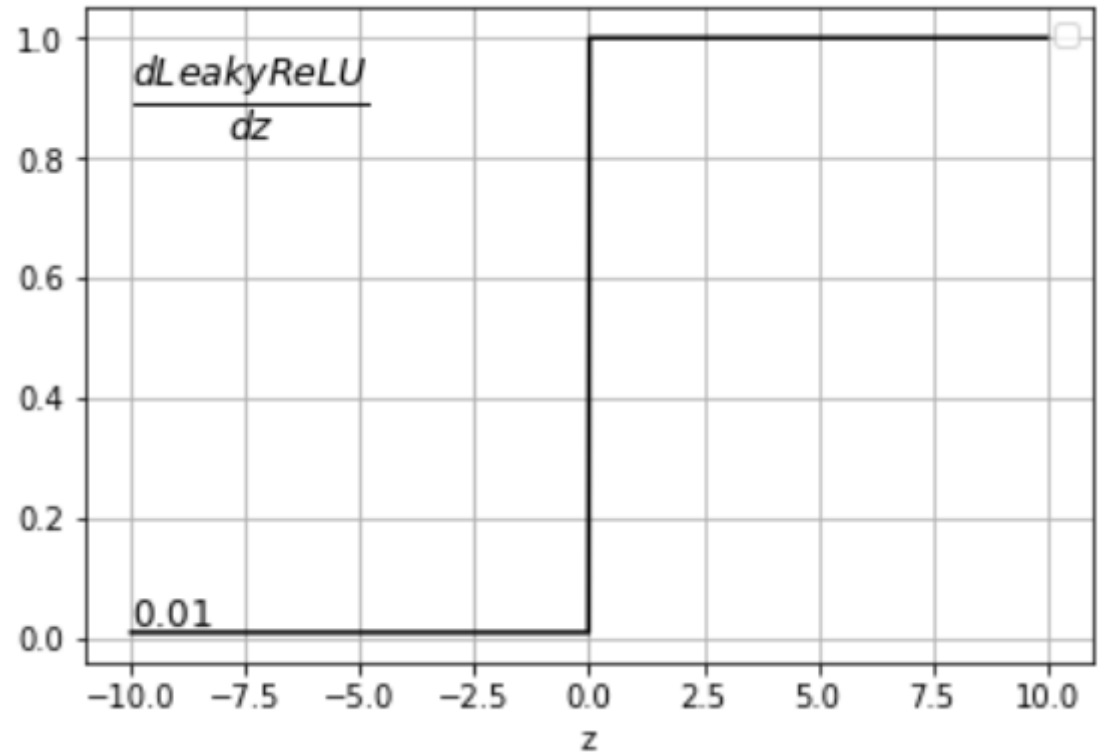

$\frac{dReLU}{dz}$
1.0
0.8
0.6
0.4
0.2
0.0
−100
−75
−50
−25
0
25
50
75
100
z

> Leaky ReLU

$$g(z) = max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01 & if \ \ z < 0 \\ 1 & if \ \ z \geq 0 \end{cases}$$





```
def activate_forward(self):

    if(self.activate == "sigmoid"):
      return lambda x : np.exp(x) / (1.0 + np.exp(x))
    elif(self.activate == "tanh"):
      return lambda x : (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
    elif(self.activate == "ReLU"):
      def relu(x):
        y = np.copy(x)
        y[y<0] = 0
        return y
      return relu
    elif(self.activate == "Leaky_ReLU"):
      def leaky_relu(x):
        y = np.copy(x)
        y[y<0] = 0.01 * y[y<0]
        return y
      return leaky_relu
    else:
      return lambda x : x



  def activate_back(self):

    if(self.activate == "sigmoid"):
      return lambda x : np.multiply(x, 1.0 - x)
    elif(self.activate == "tanh"):
      return lambda x : 1 -  ((np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x)))** 2
    elif(self.activate == "ReLU"):
      def der_relu(x):
        y = np.copy(x)
        y[y>=0] = 1
        y[y<0] = 0
        return y
      return der_relu
    elif(self.activate == "Leaky_ReLU"):
      def der_leaky_relu(x):
        y = np.copy(x)
        y[y>=0] = 1
        y[y<0] = 0.01
        return y
      return der_leaky_relu
    else:
      def one(x):
        y = np.zeros(x.shape)
        y[y==0] = 1
        return y
      return one
```

1. ReLU
    - dataset : linear
    - hidden layer1 : 4

- hidden layer2 : 6
- learning rate : 0.001
- activate function : ReLU
- optimizer : SGD
- epochs : 10000
- batch_size : 10

```
accurancy : 62.0%, loss : 0.082112
```

2. Leaky ReLU
   - dataset : linear
   - hidden layer1 : 4
   - hidden layer2 : 6
   - learning rate : 0.001
   - activate function : Leaky ReLU
   - optimizer : SGD
   - epochs : 10000
   - batch_size : 10

```
accurancy : 52.0%, loss : 0.078608
```

3. Tanh
   - dataset : linear
   - hidden layer1 : 5
   - hidden layer2 : 8
   - learning rate : 0.01
   - activate function : Tanh
   - optimizer : SGD
   - epochs : 10000
   - batch_size : 10

```
accurancy : 99.0%, loss : 0.006092
```

tags: `DLP2021`