Lab 1 Back-propagation

林穎志 411551007

1. Introduction

The main objective of this lab is to implement the back-propagation progress in neural network(nn). Which is a method to calculate the gradients in order to update and calculate the weights of neurons in nn. The main idea of how the nn works is that we input the training data and ground truth into nn, it will first learn the representation forwardly then back-propagate according the loss between ground truth and prediction. Letting the nn to update its weights iteratively to learn better on the input features.

2. Experiments

In this lab, we implement the the back-propagation method. Besides, we do experiments on different learning rate, hidden units, optimizers and activation functions to observe the difference on the training result.

A. Sigmoid function

The formula of Sigmoid and derivative Sigmoid functions are shown as below. It is a non-linearity activation function, making the output value located between 0 and 1. The value of Sigmoid function is shown on Fig.1 .

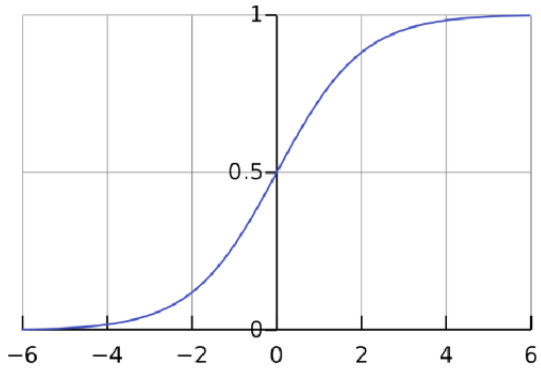


Fig. 1 Sigmoid function

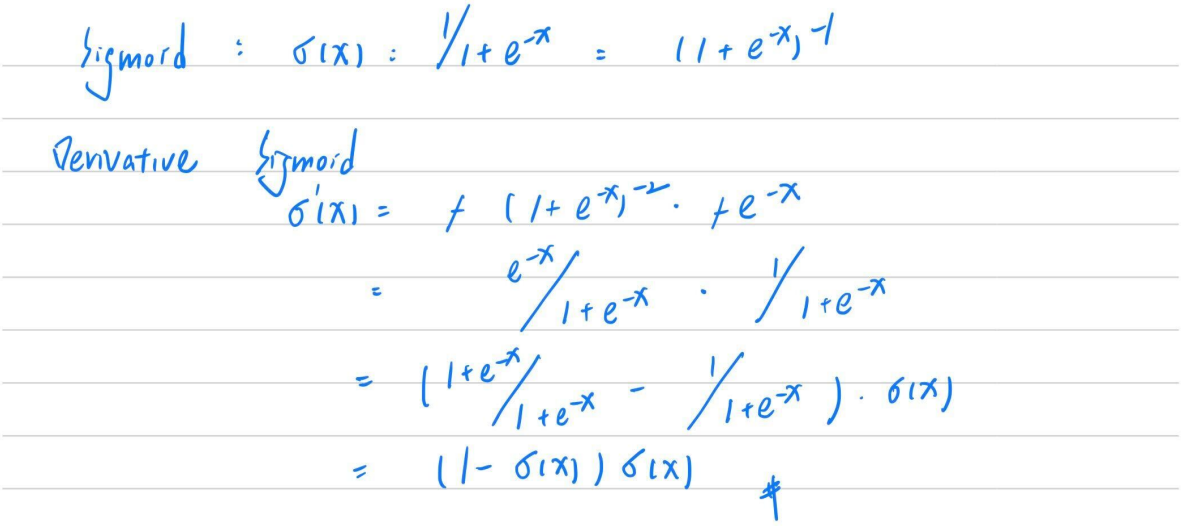


Fig. 2 Sigmoid function and derivative Sigmoid function

```
# ----------------------
#   activation function & loss
# ----------------------

# Sigmoid function
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
# derivative sigmoid
def derivativeSigmoid(y):
    return np.multiply(y, 1.0 - y)
```

B. Neural network

The nn we implemented is shown on Fig. 4. Including 2 layers of hidden layers but different hidden units. The result of comparing different hidden units will be shown in result. The main architecture is show as below. We chose MSE as our loss function and train on the given dataset. The foward passing parameters are defined as:

$$a_1 = \sigma(XW_0)\ ;\ a_2 = \sigma(a_1W_1)\ ;\ \hat{y} = \sigma(a_2W_2)$$

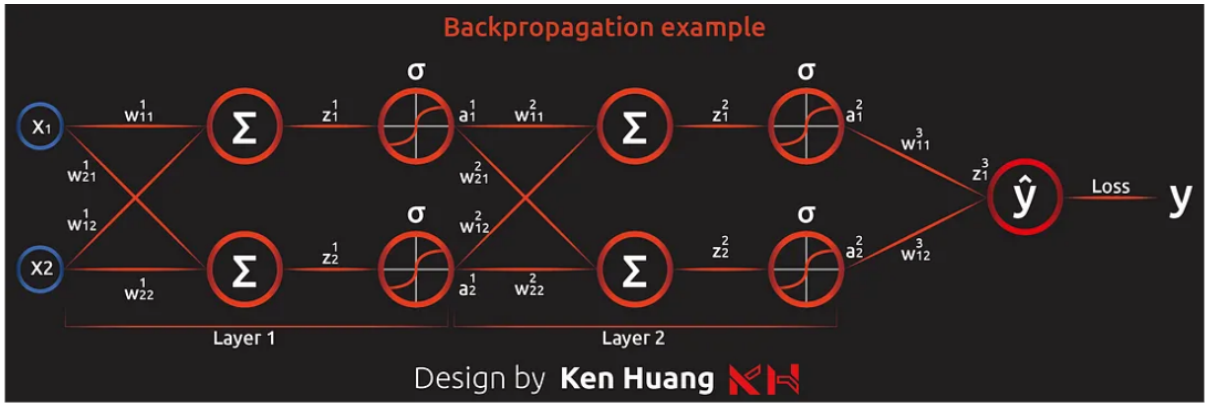


Fig. 4 Neural network

```
class network_h2():
    def __init__(self,  hid_num = 2, neuro_num = 10, activation = 'relu'):

        self.activation = activation

        self.hid_num = hid_num
        self.neuro_num = neuro_num

        self.wsum_0 = 0
        self.wsum_1 = 0
        self.wsum_2 = 0

        self.w0 = np.random.normal(0, 1, (2, self.neuro_num))
        self.w1 = np.random.normal(0, 1, (self.neuro_num , self.neuro_num))
        self.w2 = np.random.normal(0, 1, (self.neuro_num, 1))

        self.momentum0 = np.zeros((2, self.neuro_num))
        self.momentum1 = np.zeros((self.neuro_num , self.neuro_num))
        self.momentum2 = np.zeros((self.neuro_num, 1))

    def forward(self, x):


    def backward(self, y):
        '''
        backward propagation
        '''

    def update(self, lr, optimizer=None):
        '''
        Update weight
        '''

# Loss function
def derivative_mse(y, pred_y):
    return -2 * (y - pred_y) / y.shape[0]

def loss_mse(y, pred_y):
    return np.mean((y - pred_y)**2)

# Train
def train(model,  epoch = 1000, lr = 1e-3, optimizer = 'sgd', dataType = 'linear'):
```

C. Back-Propagation

A back-propagation example is shown on Fig. 5. The algorithm is based on chain rule, which the partial derivative on the loss function of w11 and w12 for example can be defined as:

$$\frac{\partial L}{\partial w_{11}^1} = \frac{1}{n}(y - z_1^3)[\sum_i^2 (w_{1i}^3)\sigma(z_i^2)(1-\sigma(z_i^2))(w_{i1}^2)]\sigma(z_1^2)(1-\sigma(z_1^2))x_1$$

$$\frac{\partial L}{\partial w_{12}^1} = \frac{1}{n}(y - z_1^3)[\sum_i^2 (w_{1i}^3)\sigma(z_i^2)(1-\sigma(z_i^2))(w_{i1}^2)]\sigma(z_1^2)(1-\sigma(z_1^2))x_2$$

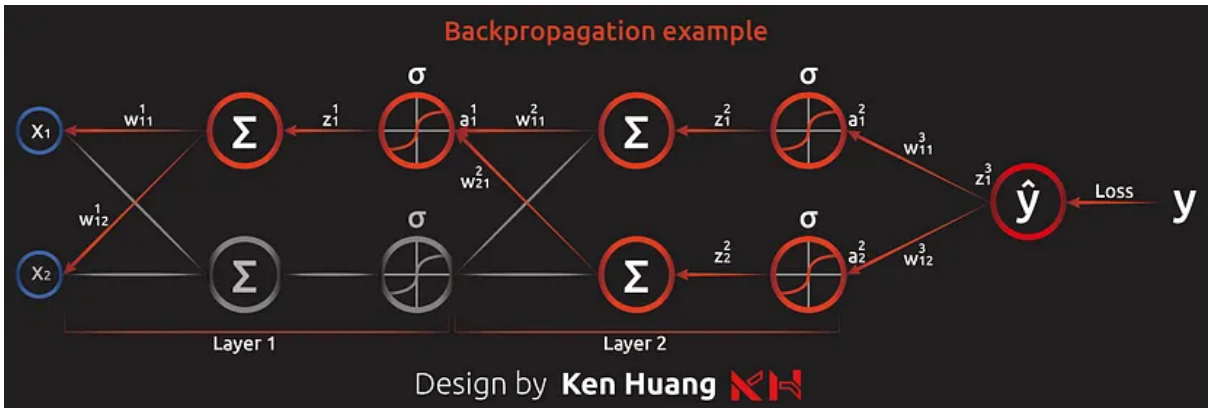


Fig. 5 Main architecture

The nn will update all the weights base on the gradients and according to the algorithm of optimizer. The code is shown as below:

```
def forward(self, x):
    self.x = x
    self.z1 = x @ self.w0

    if self.activation =='sigmoid':
        self.a1 = sigmoid(self.z1)
        self.z2 = self.a1 @ self.w1
        self.a2 = sigmoid(self.z2)
        self.z3 = self.a2 @ self.w2

    if self.activation is not None:
        self.yhat = sigmoid(self.z3)
    else:
        self.yhat = self.z3

    return self.yhat


def backward(self, y):
    '''
    backward propagation
    '''
    #output

    self.de_dy = derivative_mse(y, self.yhat) #loss

    if self.activation is not None:
        #W2
        self.d2_act = derivativeSigmoid(self.yhat)  #act the output of sigmoid
```

```
        self.bw2 = np.multiply(self.d2_act, self.de_dy) #loss*act (G1)
        # self.backwardGrad_w2 = np.matmul(self.backwardGrad, self.w_dic['w2'].T)  #bw2 = (loss*act)@w2
        self.fw2 = self.a2.T @ self.bw2 #(10, 1)

    if self.activation =='sigmoid':
        #w1
        self.d1_act = derivativeSigmoid(self.a2) #act() the output of sigmoid
        self.bw1 = np.multiply(self.d1_act, np.matmul(self.bw2, self.w2.T))  #act(z2) * bw2  (G2)
        self.fw1 = self.a1.T @ self.bw1 #(10, 10)

        #w0
        self.d0_act = derivativeSigmoid(self.a1) #act() the output of sigmoid #(100, 10)
        self.bw0 = np.multiply(self.d0_act, np.matmul(self.bw1, self.w1.T))  #act(z1) * bw1  (G3)
        self.fw0 = self.x.T @ self.bw0  #shape shoulb be (2, 10)


def update(self, lr, optimizer=None):
    '''
    Update weight
    '''
    self.lr = lr
    # print('optimizer = ', optimizer)

    if optimizer == 'adagrad':
        self.wsum_0 += (self.w0**2)
        self.wsum_1 += (self.w1**2)
        self.wsum_2 += (self.w2**2)

        self.w0 +=  -self.lr * self.fw0 / (self.wsum_0**0.5)
        self.w1 +=  -self.lr * self.fw1 / (self.wsum_1**0.5)
        self.w2 +=  -self.lr * self.fw2 / (self.wsum_2**0.5)

    elif optimizer == 'sgd':
        self.w0 +=  -self.lr * self.fw0
        self.w1 +=  -self.lr * self.fw1
        self.w2 +=  -self.lr * self.fw2

    elif optimizer == 'momentum':
        self.momentum0 = 0.9 * self.momentum0 - self.lr * self.fw0
        self.momentum1 = 0.9 * self.momentum1 - self.lr * self.fw1
        self.momentum2 = 0.9 * self.momentum2 - self.lr * self.fw2

        self.w0 +=  self.momentum0
        self.w1 +=  self.momentum1
        self.w2 +=  self.momentum2
```

The chain rule mathemetics of my code is show as below:

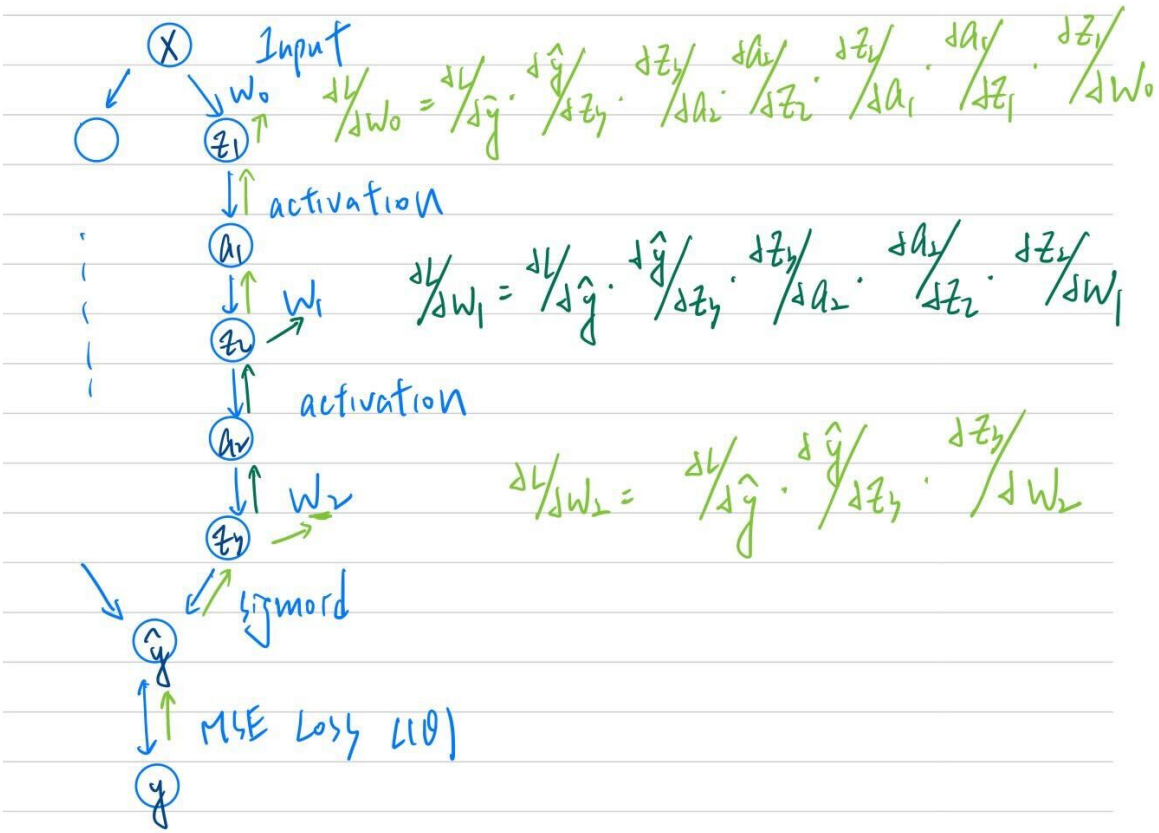


Fig. 6 Chain rule

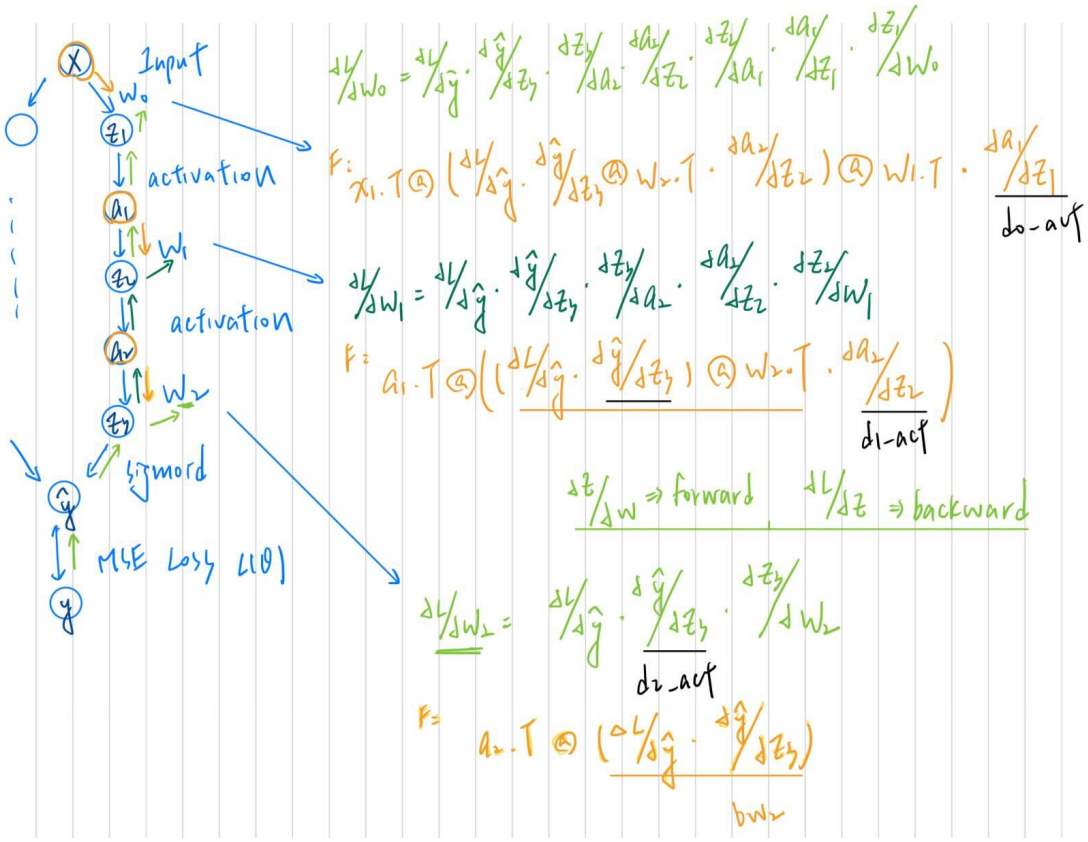


Fig. 7 Foward and backward gradients

We can then update the weights by the SGD algorithm, the algorithm is defined as:

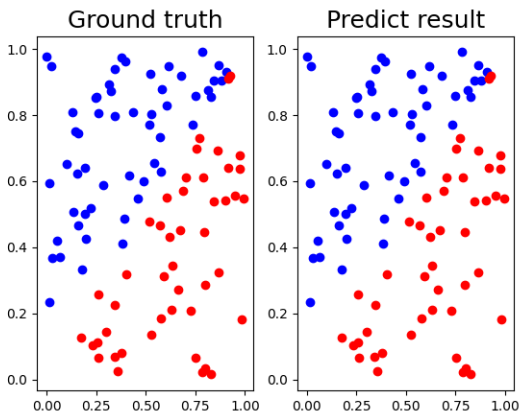
$$W_{n+1} = W_n - lr * \nabla W_n$$

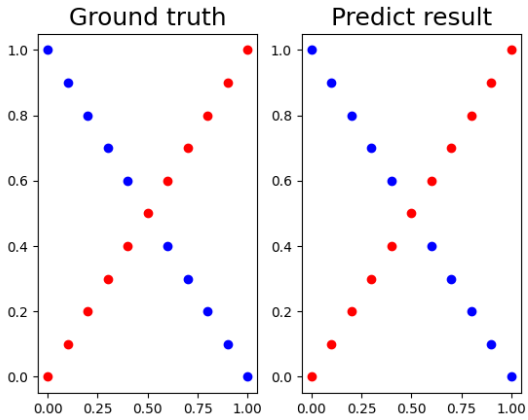
which *lr* is learning rate.

3. Result

A. Sceenshots and Comparisons

For both linear and XOR data, the comparisons are shown as below:



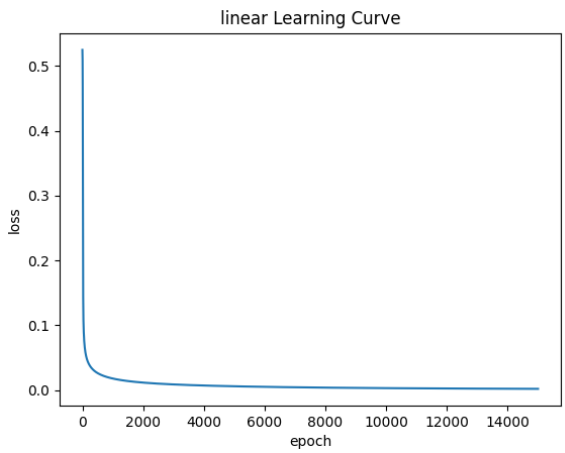


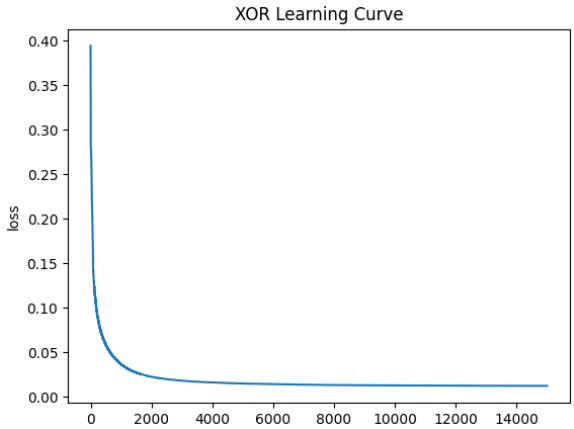
B. Accuracy of prediction

```
[1.00000000e+00]
[4.83431943e-06]
[9.03665551e-15]
[3.42304921e-03]
[3.41819607e-01]
[7.96220940e-10]
[4.38117410e-12]
[1.00000000e+00]
[3.03631637e-05]
[3.15107929e-13]
[9.98828808e-14]
[5.35084754e-15]
[1.98113754e-09]
[6.37989165e-07]
[1.00000000e+00]
[9.99999987e-01]
[2.07654842e-09]]

Datatype: linear, Optimizer: sgd, Activation fun: relu
Test loss :  0.0019769508203095532
Test Acc: 100.0%
```

```
[1.48850095e-02]
[9.99998919e-01]
[1.85390465e-03]
[9.99933190e-01]
[2.28257593e-04]
[9.68957578e-01]
[2.80635935e-05]
[3.44972967e-06]
[9.68286675e-01]
[4.24050395e-07]
[9.99999970e-01]
[5.21253191e-08]
[1.00000000e+00]
[6.40737055e-09]
[1.00000000e+00]
[7.87609483e-10]
[1.00000000e+00]]

Datatype: XOR, Optimizer: sgd, Activation fun: relu
Test loss :  0.012579873591404047
Test Acc: 100.0%
```

C. Learning curve





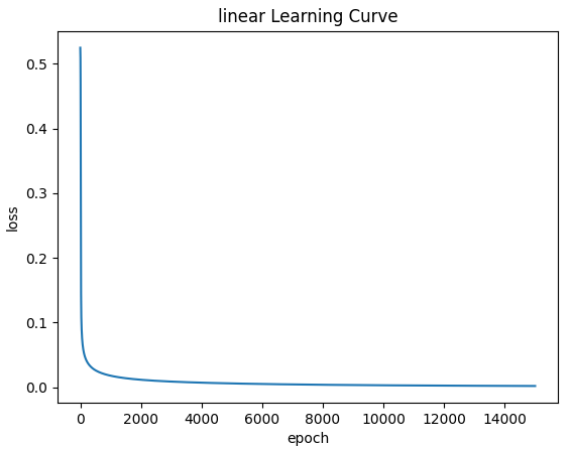
D. Anything to say

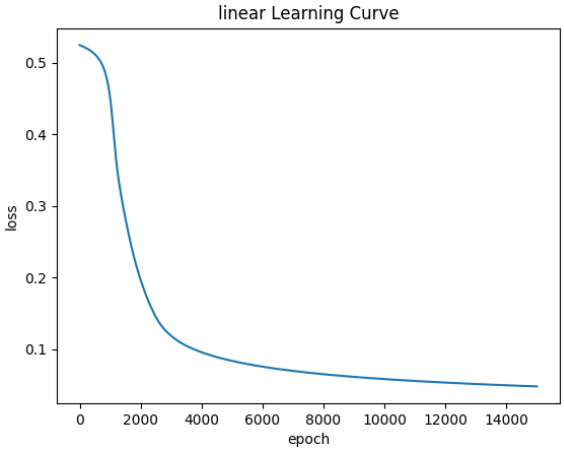
In this task, we designed a network with 2 hidden layers with 10 neurons for each hidden layer. The activation ReLu is chosen and the output activation function is Sigmoid. We can observe that our proposed architecture achieved a hundred percent of accuracy on both linear and XOR data. Furthermore, the model converged slower on XOR data compared to linear data.

4. Discussion

A. Different learning rate

The initial setting of network included 2 layers of hidden layers with 10 neurons of each, ReLu activation function and SGD optimizer. The datatype is linear. The left figure shows the result of learning rate = 1e-1, and the right is 1e-3. The accuracy is 100% (left) and 97% respectively. We can observe that with lower learning rate, the learning curve conveges slower, one may need more epochs to train to achieve 100% of accuracy.

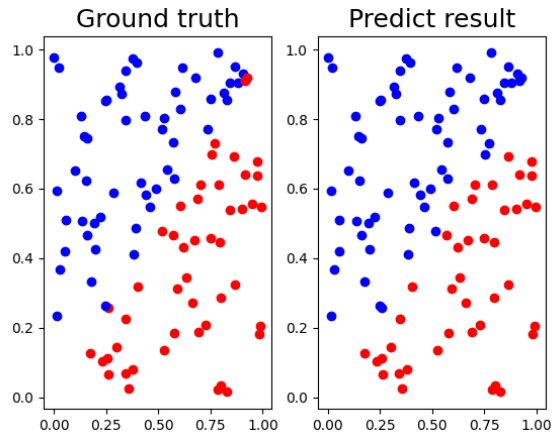


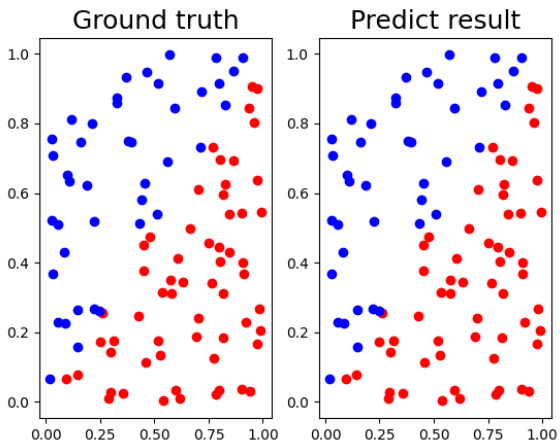


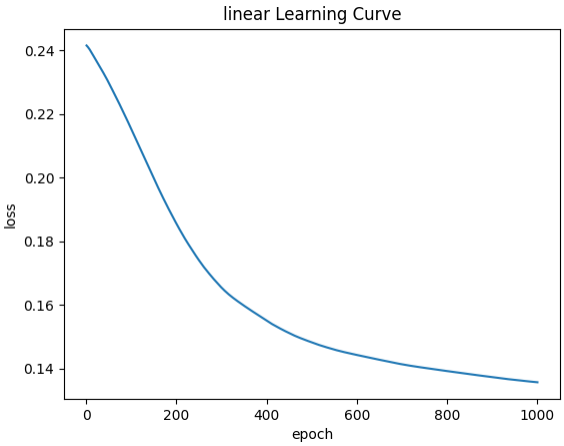
B. Different numbers of hidden units

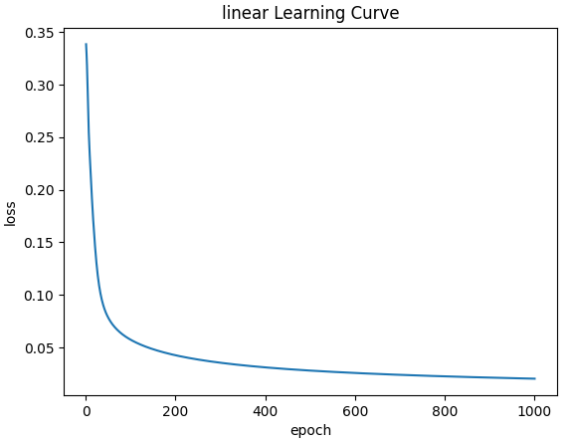
For both linear and XOR data, we can observe that the nn with more neurons appears to have a better performance, choosing the amount of the neurons is important.

Datatype: Linear	Left	Right
Accuracy	**94%**	**100%**
Hidden units	**2**	**10**
Hidden layers	2	2
Activation function	ReLu	ReLu
Optimizer	SGD	SGD
Epoch	1000	1000
Learning rate	0.1	0.1

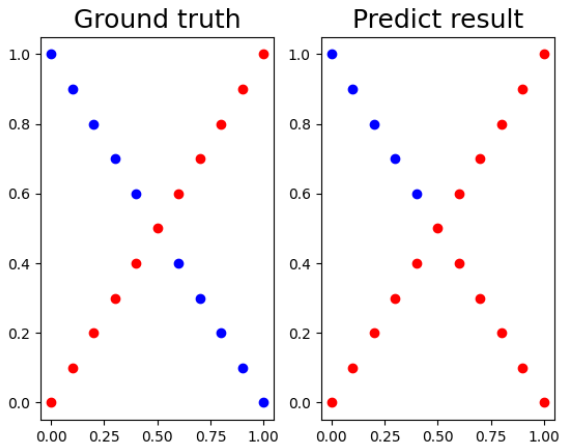


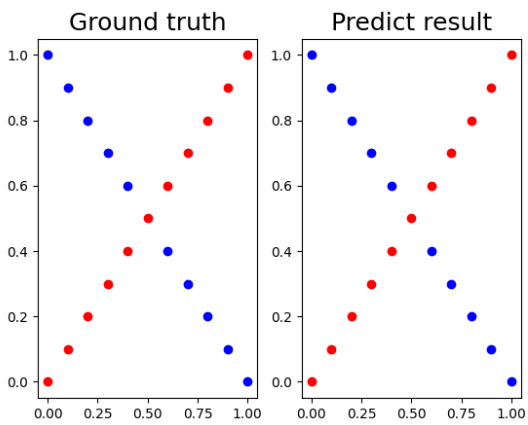


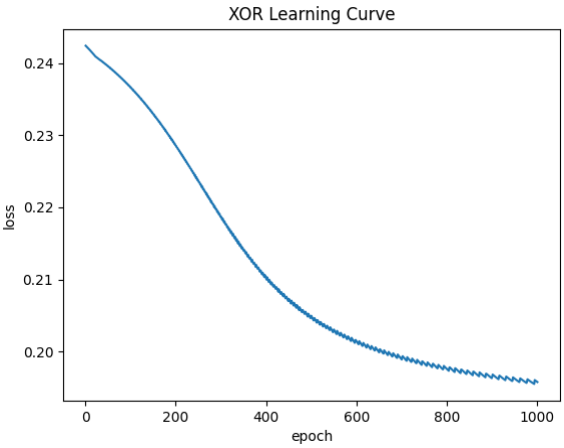


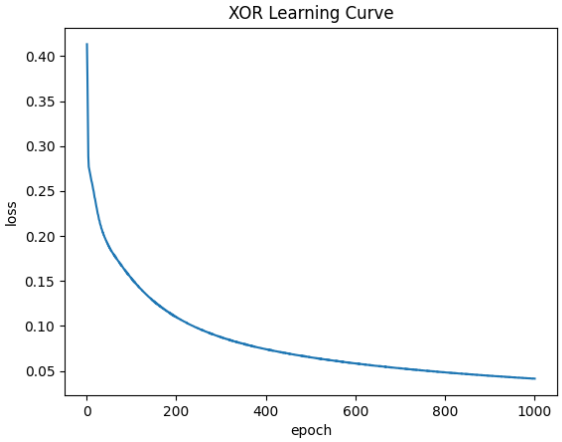


Datatype: XOR	Left	Right
Accuracy	**76%**	**100%**
Hidden units	**2**	**10**
Hidden layers	2	2
Activation function	ReLu	ReLu
Optimizer	SGD	SGD
Epoch	1000	1000
Learning rate	0.1	0.1





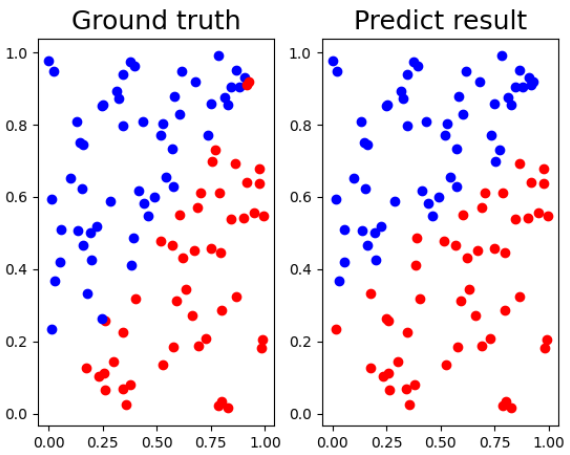


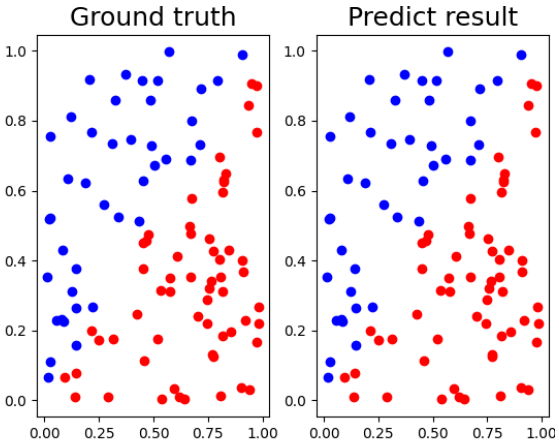


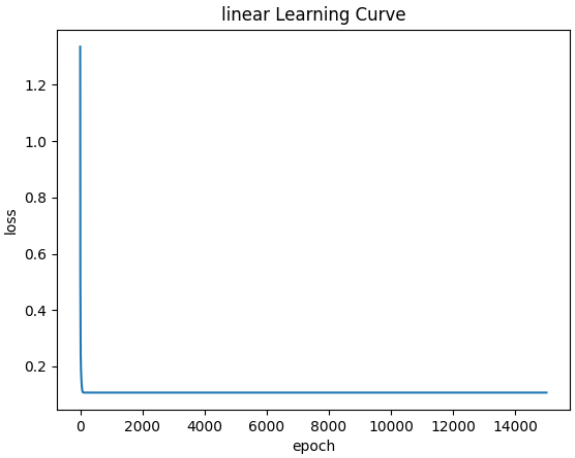
C. Try without activation functions

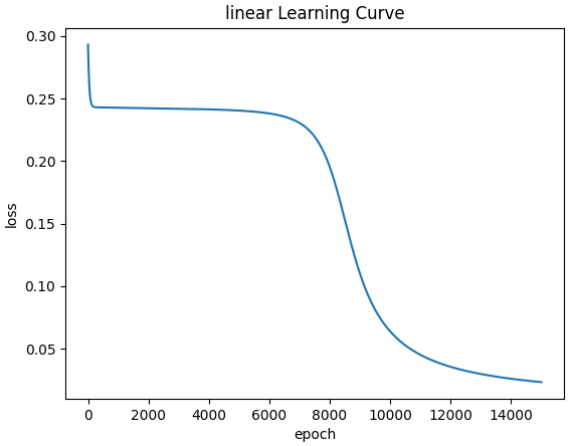
For both linear and XOR data, we can observe that the nn with an activation function performs better, and if the hidden neurons are too big, the loss will be extremly large causing NaN in nn without activation function.

Datatype: Linear	Left	Right
Accuracy	**91%**	**100%**
Activation function	**None**	**Sigmoid**
Hidden units	2	2
Hidden layers	2	2
Optimizer	SGD	SGD
Epoch	15000	15000
Learning rate	0.1	0.1

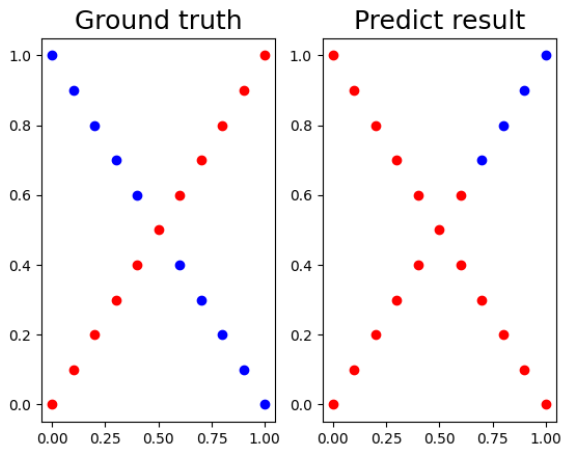


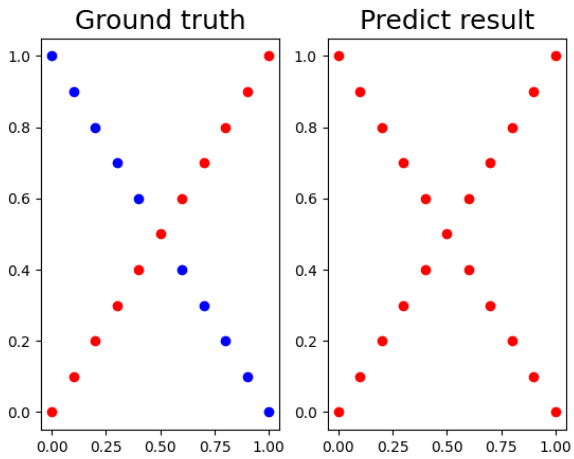


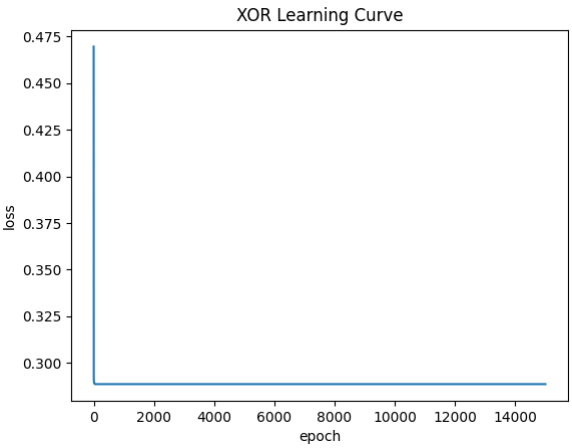


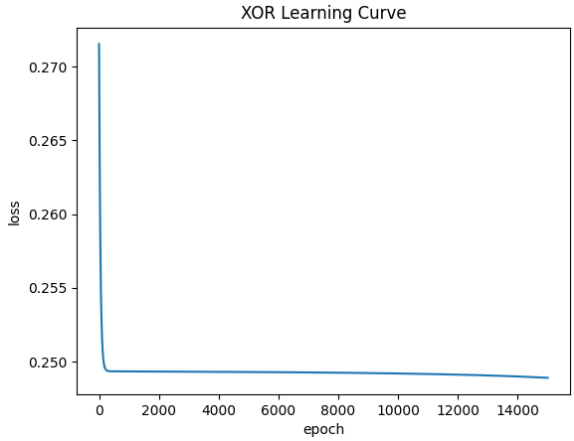


Datatype: XOR	Left	Right
Accuracy	**33.33%**	**52.38%**
Activation function	**None**	**Sigmoid**
Hidden units	2	2
Hidden layers	3	3
Optimizer	SGD	SGD
Epoch	15000	15000
Learning rate	0.1	0.1









D. Anything to share

In ths section, we experimented different settings of hyperparameters, knowing why the basic nn should be design as this architecure. Besides, we know how the hyperparameters influence the model performance.

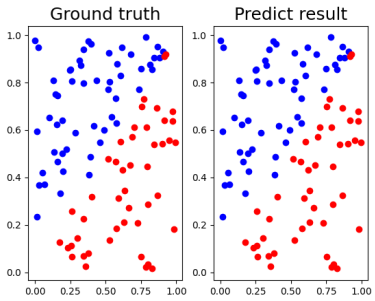
5. Extra

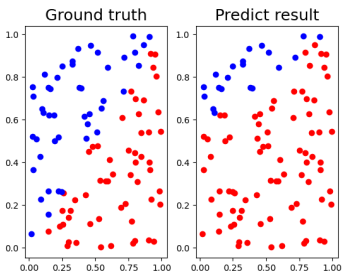
A. Different optimizers

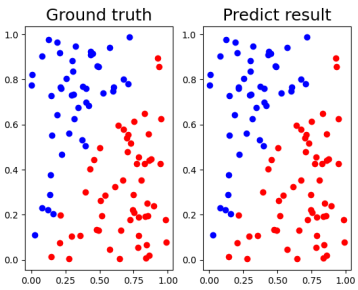
In this task, we observe that the Momentum and SGD perform better than Adagrad. Besides, Momentum and Adagrad have a smoother learning curve, owing to the algorithm can prevent oscilliation and enhance converge speed.

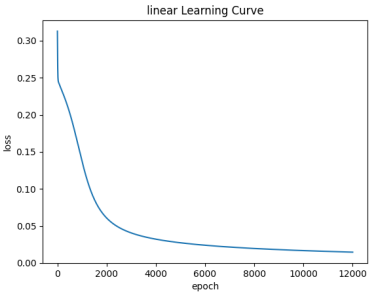
Datatype: Linear	Left	Middle	Right
Accuracy	**100%**	**52.38%**	**100%**
Optimizer	**SGD**	**Adagrad**	**Momentun**
Hidden units	5	5	5

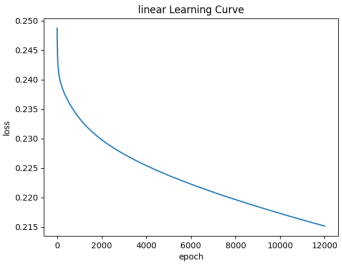
Hidden layers	10	10	10
Activation function	Sigmoid	Sigmoid	Sigmoid
Epoch	12000	12000	12000
Learning rate	0.1	0.1	0.1

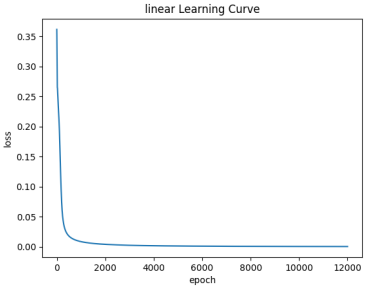




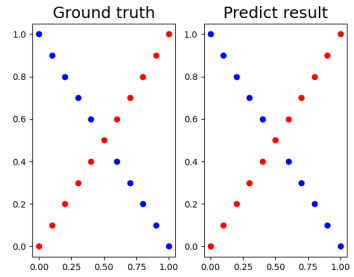


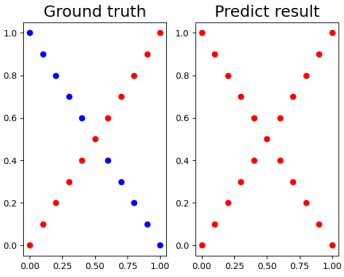


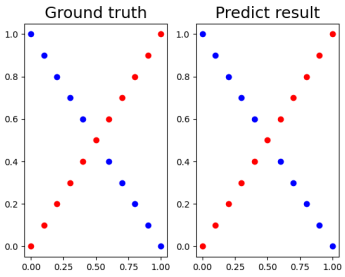


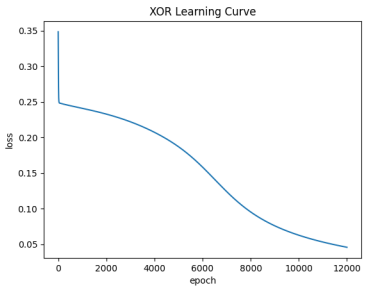


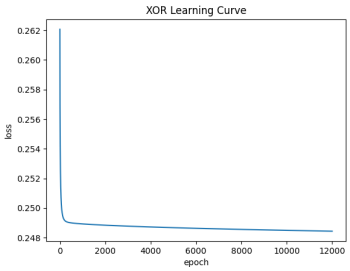
Datatype: Linear	Left	Middle	Right
Accuracy	**100%**	**75%**	**100%**
Optimizer	**SGD**	**Adagrad**	**Momentun**
Hidden units	5	5	5
Hidden layers	10	10	10
Activation function	Sigmoid	Sigmoid	Sigmoid
Epoch	12000	12000	12000
Learning rate	0.1	0.1	0.1

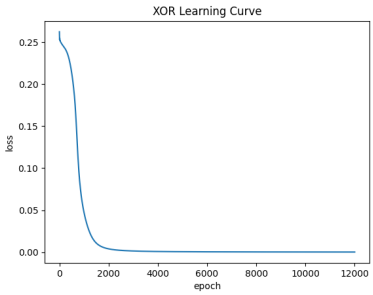








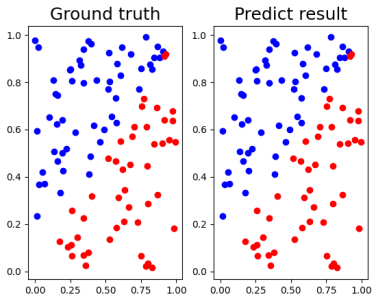


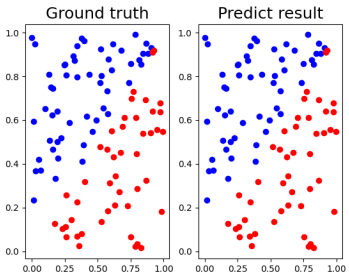


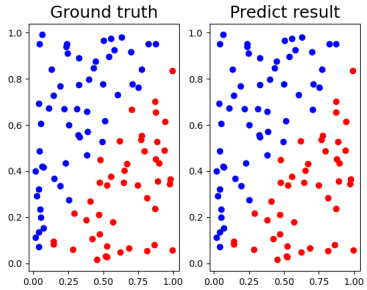
B. Different activation functions

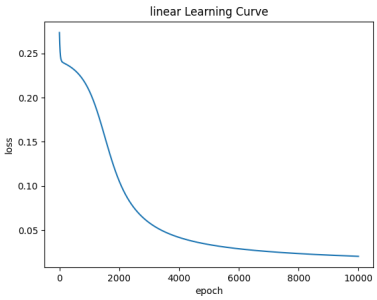
We can oberve that ReLu and Tanh have a well performance in both XOR and linear data. Furthermore, the learning curve of which is smoother and converge faster compares to Sigmoid.

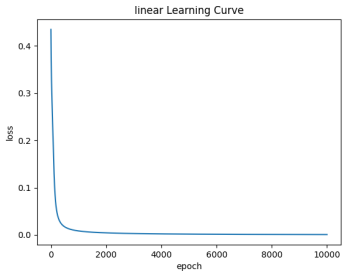
Datatype: Linear	Left	Middle	Right
Accuracy	**100%**	**100%**	**99%**
Activation function	**Sigmoid**	**Tanh**	**ReLu**
Hidden units	5	5	5
Hidden layers	2	2	2
Optimizer	SGD	SGD	SGD
Epoch	10000	10000	10000
Learning rate	0.1	0.1	0.1

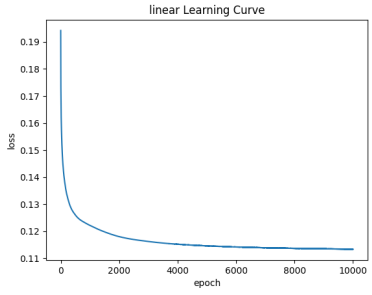












Datatype: XOR	Left	Middle	Right
Accuracy	**90%**	**100%**	**100%**

Activation function	**Sigmoid**	**Tanh**	**ReLu**
Hidden units	5	5	5
Hidden layers	2	2	2
Optimizer	SGD	SGD	SGD
Epoch	10000	10000	10000
Learning rate	0.1	0.1	0.1