DLP_HW1_多工所碩一 313553024 蘇柏叡

1. Introduction (20%)

(1)實驗要求說明

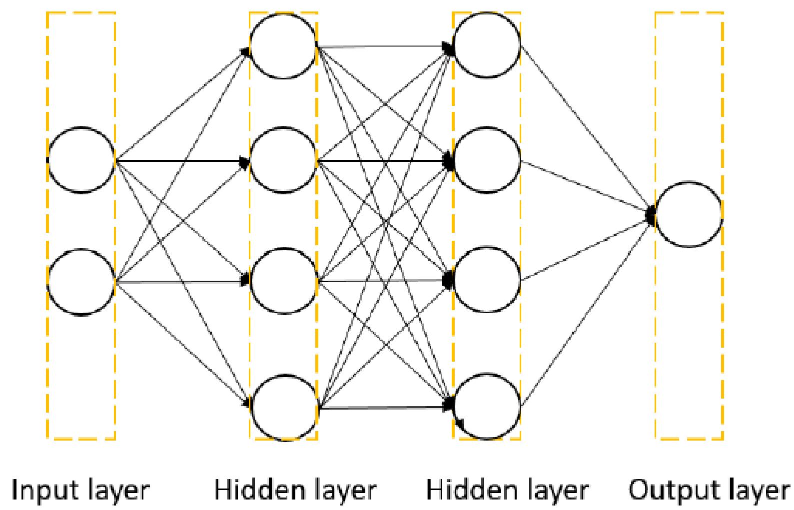
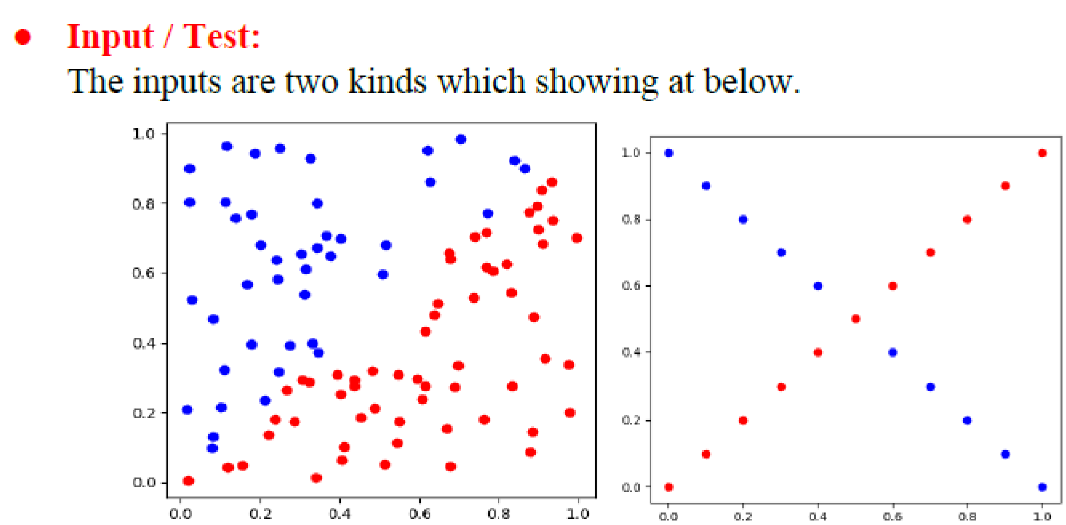


Figure 1. Two-layer neural network

本次Lab要實作的是雙層的神經網路並且使用簡報提供的兩種方法去生成(1)Linear、(2)XOR兩種資料，範例是Linear有100個資料點，XOR是有21個資料點。

資料點圖片示例如簡報所提供(下圖):



本次實驗包含1個輸入層(兩個units)，兩個隱藏層(default各4個units)，一個輸出層，於討論處會討論不同hidden layer units數量會有何結果。此外，本實驗的核心為對

training data在透過訓練過程中不斷試圖降低損失值，並使預測值和groundtruth能夠愈精準。

a. Forward propagation:

輸入的資料點在經過模型後，會依據不同的權重、激活函數…等計算出預測值。

b. Backward propagation:

計算預測值和 Ground Truth 的 error，得到 Loss value，並且透過梯度下降(GD)、Adam、Adagrad…等 optimizer 求出權重更新量，再透過 Update function 進行更新權重。

c. Weight Update:

本 Lab 分為 4 種 optimizer(Momentum、**GD**、**Adam**、Adagrad)，依據各自優化器的算法計算損失並透過 **Backward propagation** 求得梯度，最後根據選擇的優化器算法，調整每層的權重以最小化損失。(註:本 LAB 使用 GD，不過因為 Adam 也是相當廣泛使用，因此本 LAB 會都放)

d. Early Stopping:

設定當 loss < 0.001 時會自動 break 掉，減少訓練 Epoch 數量及時間成本。

2. Experiment setups:

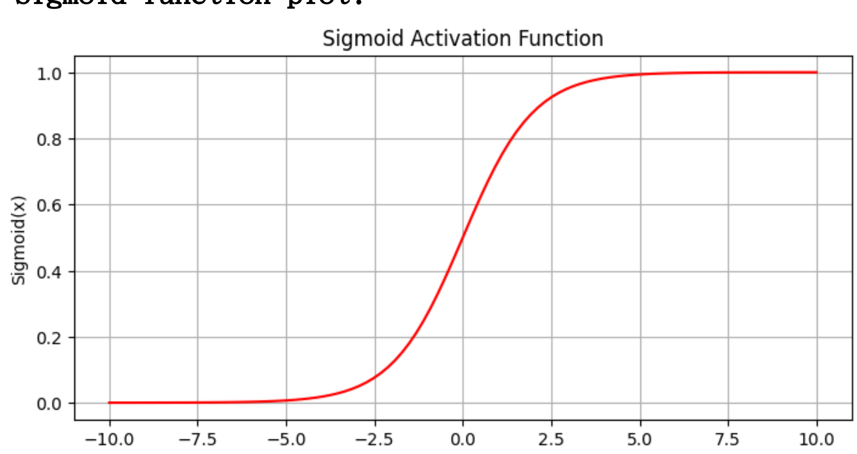
一Sigmoid functions

(1) code:

```
    @staticmethod
    def sigmoid(x: np.ndarray) -> np.ndarray:
        return 1.0 / (1.0 + np.exp(-x))

    @staticmethod
    def derivative_sigmoid(y: np.ndarray) -> np.ndarray:
        return np.multiply(y, 1.0 - y)
```

(2) Sigmoid function plot:



一Neural network

(1) Code(僅截forward、Backward、Update、mse、derivative_mse部分):

```
class Model:
    def __init__(self, epoch: int, learning_rate: float, num_of_hidden_layers: int, input_units: int,
                 hidden_units: int, output_units: int, activation: str, optimizer: str):
    ## 預設2個hidden layer，每個hidden layer 會有4個hidden units，預設activation是sigmoid
        self.num_of_epoch = epoch
        self.learning_rate = learning_rate
        self.hidden_units = hidden_units
        self.activation = activation
        self.optimizer = optimizer
        self.learning_epoch, self.learning_loss = list(), list()

        # Setup input layer
        self.input_layer = Layer(input_units, hidden_units, activation, optimizer, learning_rate)

        # Dynamically create hidden layers
        self.hidden_layers = [Layer(hidden_units, hidden_units, activation, optimizer, learning_rate)
                              for _ in range(num_of_hidden_layers)]

        # Output layer
        self.output_layer = Layer(hidden_units, output_units, 'sigmoid', optimizer, learning_rate)  #
```

```
    def forward(self, inputs: np.ndarray) -> np.ndarray:
        """
        Forward feed through the network
        """
        inputs = self.input_layer.forward(inputs)
        for layer in self.hidden_layers:
            inputs = layer.forward(inputs)
        return self.output_layer.forward(inputs)

    def backward(self, derivative_loss) -> None:
        """
        Backward propagation through the network
        """
        derivative_loss = self.output_layer.backward(derivative_loss)
        for layer in reversed(self.hidden_layers):
            derivative_loss = layer.backward(derivative_loss)
        self.input_layer.backward(derivative_loss)

    def update(self) -> None:
        """
        Update weights in the entire network
        """
        self.input_layer.update()
        for layer in self.hidden_layers:
            layer.update()
        self.output_layer.update()
```

```
    @staticmethod
    def mse_loss(prediction: np.ndarray, ground_truth: np.ndarray) -> np.ndarray:
        return np.mean((prediction - ground_truth) ** 2)

    @staticmethod
    def mse_derivative_loss(prediction: np.ndarray, ground_truth: np.ndarray) -> np.ndarray:
        return 2 * (prediction - ground_truth) / len(ground_truth)
```

一Backpropagation

(1) Layer中的backward function:

a. code:

```python
def backward(self, derivative_loss: np.ndarray) -> np.ndarray:
    # Calculate the gradient of the loss with respect to the output of the layer
    if self.activation == 'sigmoid':
        self.backward_gradient = np.multiply(Activation.derivative_sigmoid(self.output), derivative_loss)
    elif self.activation == 'tanh':
        self.backward_gradient = np.multiply(Activation.derivative_tanh(self.output), derivative_loss)
    elif self.activation == 'relu':
        self.backward_gradient = np.multiply(Activation.derivative_relu(self.output), derivative_loss)
    elif self.activation == 'leaky_relu':
        self.backward_gradient = np.multiply(Activation.derivative_leaky_relu(self.output), derivative_loss)
    else:
        self.backward_gradient = derivative_loss
    return self.backward_gradient @ self.weight[:-1].T
```

b. **說明:**

本function輸入的參數derivative_loss，為後一層傳遞過來的損失函數的導數，此外本function分為四種激活函數進行分別並根據各自定義做multiply(使用如作業說明所說的chain rule)，最後透過矩陣乘法將當前層的輸出梯度和權重矩陣先轉置再相乘(可計算出該層輸入的梯度並將其傳遞到網絡中前一層)，接著再繼續Backpropagation。

(2) Model中的backward function

a. code:

```python
def backward(self, derivative_loss) -> None:

    derivative_loss = self.output_layer.backward(derivative_loss)
    for layer in reversed(self.hidden_layers):
        derivative_loss = layer.backward(derivative_loss)
    self.input_layer.backward(derivative_loss)
```

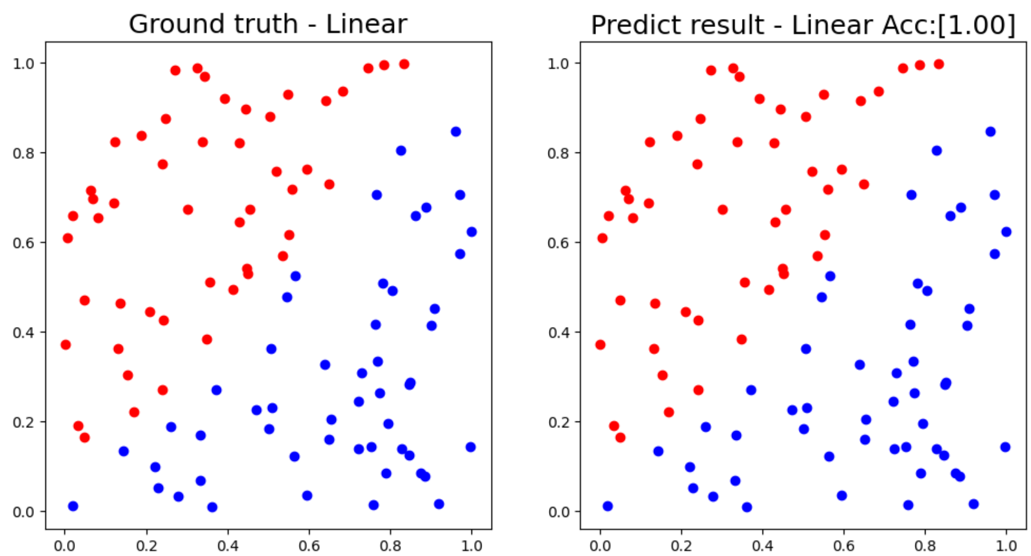
b. **說明:**

首先調用Layer中的 backward function來開始損失導數的Back propagation，然後反向遍歷隱藏層，將每層計算得到的導數傳遞給前一層。最後，損失導數會被傳遞到輸入層，完成Model的梯度計算。

3. Results of your testing (20%)

—Screenshot and comparison figure

(1) Linear Data



(2) XOR Data:

—Show the accuracy of your prediction

(1) Linear Data:

a. 實驗結果細節(Adam):

```
-------Experiment Details -------

Data type: Linear data points
Activation: sigmoid
Learning rate: 0.1
Number of hidden layers: 2
Hidden units: 4
Optimizer: adam
Accuracy: 100.00%
Loss: 0.1106
-------------------------
```

b. Training Loss and Testing Result(Part)

```
Iter: 90        | Ground Truth: [0] | prediction: [0.00676215] |
Iter: 91        | Ground Truth: [0] | prediction: [0.00644663] |
Iter: 92        | Ground Truth: [1] | prediction: [0.99806473] |
Iter: 93        | Ground Truth: [0] | prediction: [0.00645433] |
Iter: 94        | Ground Truth: [1] | prediction: [0.99800735] |
Iter: 95        | Ground Truth: [1] | prediction: [0.99810162] |
Iter: 96        | Ground Truth: [0] | prediction: [0.00645513] |
Iter: 97        | Ground Truth: [1] | prediction: [0.99377277] |
Iter: 98        | Ground Truth: [0] | prediction: [0.00658146] |
Iter: 99        | Ground Truth: [0] | prediction: [0.00648518] |
```

```
Epoch = 0  Loss = [0.39969534]  Accuracy = 0.46
Epoch = 10  Loss = [0.25578894]  Accuracy = 0.54
Epoch = 20  Loss = [0.24750434]  Accuracy = 0.54
Epoch = 30  Loss = [0.24387260]  Accuracy = 0.54
Epoch = 40  Loss = [0.21841295]  Accuracy = 0.54
Epoch = 50  Loss = [0.12192525]  Accuracy = 0.92
Epoch = 60  Loss = [0.03488295]  Accuracy = 1.00
Epoch = 70  Loss = [0.01137883]  Accuracy = 1.00
Epoch = 80  Loss = [0.00527135]  Accuracy = 1.00
Epoch = 90  Loss = [0.00327075]  Accuracy = 1.00
Epoch = 100  Loss = [0.00227610]  Accuracy = 1.00
Epoch = 110  Loss = [0.00174592]  Accuracy = 1.00
Epoch = 120  Loss = [0.00138703]  Accuracy = 1.00
Epoch = 130  Loss = [0.00113086]  Accuracy = 1.00
```

a. 實驗結果細節(GD):

```
Data type: XOR data points
Activation: sigmoid
Learning rate: 0.1
Number of hidden layers: 2
Hidden units: 4
Optimizer: gd
Accuracy: 100.00%
Loss: 0.1505
```

b. Training Loss and Testing Result(Part)

```
Epoch = 0   Loss = [0.45692423]   Accuracy = 0.52
Epoch = 5000   Loss = [0.24901642]   Accuracy = 0.52
Epoch = 10000   Loss = [0.24821503]   Accuracy = 0.52
Epoch = 15000   Loss = [0.24341205]   Accuracy = 0.48
Epoch = 20000   Loss = [0.21606819]   Accuracy = 0.71
Epoch = 25000   Loss = [0.06048855]   Accuracy = 1.00
Epoch = 30000   Loss = [0.01999831]   Accuracy = 1.00
Epoch = 35000   Loss = [0.00651260]   Accuracy = 1.00
Epoch = 40000   Loss = [0.00259872]   Accuracy = 1.00
Epoch = 45000   Loss = [0.00137740]   Accuracy = 1.00
```

```
Iter: 10         | Ground Truth: [0] | prediction: [0.09241769] |
Iter: 11         | Ground Truth: [0] | prediction: [0.02985942] |
Iter: 12         | Ground Truth: [1] | prediction: [0.92824395] |
Iter: 13         | Ground Truth: [0] | prediction: [0.00744278] |
Iter: 14         | Ground Truth: [1] | prediction: [0.99802524] |
Iter: 15         | Ground Truth: [0] | prediction: [0.00312518] |
Iter: 16         | Ground Truth: [1] | prediction: [0.99869671] |
Iter: 17         | Ground Truth: [0] | prediction: [0.00193423] |
Iter: 18         | Ground Truth: [1] | prediction: [0.99878882] |
Iter: 19         | Ground Truth: [0] | prediction: [0.00146984] |
Iter: 20         | Ground Truth: [1] | prediction: [0.99880784] |
```

(2) XOR Data:

a. 實驗結果細節(Adam):

```
-------Experiment Details -------

Data type: XOR data points
Activation: sigmoid
Learning rate: 0.1
Number of hidden layers: 2
Hidden units: 4
Optimizer: adam
Accuracy: 100.00%
Loss: 0.1370
-------------------------
```

b. Training Loss and Testing Result(Part)

```
Epoch = 250  Loss = [0.04596963]  Accuracy = 0.95
Epoch = 260  Loss = [0.04302146]  Accuracy = 0.95
Epoch = 270  Loss = [0.02845436]  Accuracy = 0.95
Epoch = 280  Loss = [0.02522249]  Accuracy = 0.95
Epoch = 290  Loss = [0.01439581]  Accuracy = 1.00
Epoch = 300  Loss = [0.01228130]  Accuracy = 1.00
Epoch = 310  Loss = [0.00807228]  Accuracy = 1.00
Epoch = 320  Loss = [0.00561066]  Accuracy = 1.00
Epoch = 330  Loss = [0.00372465]  Accuracy = 1.00
Epoch = 340  Loss = [0.00263157]  Accuracy = 1.00
Epoch = 350  Loss = [0.00193100]  Accuracy = 1.00
Epoch = 360  Loss = [0.00149407]  Accuracy = 1.00
Epoch = 370  Loss = [0.00120041]  Accuracy = 1.00
Epoch = 380  Loss = [0.00099512]  Accuracy = 1.00
```

```
Iter: 8          | Ground Truth: [0] | prediction: [0.02586511]  |
Iter: 9          | Ground Truth: [1] | prediction: [0.92285478]  |
Iter: 10         | Ground Truth: [0] | prediction: [0.07027517]  |
Iter: 11         | Ground Truth: [0] | prediction: [0.04794331]  |
Iter: 12         | Ground Truth: [1] | prediction: [0.94379344]  |
Iter: 13         | Ground Truth: [0] | prediction: [0.02048664]  |
Iter: 14         | Ground Truth: [1] | prediction: [0.99236262]  |
Iter: 15         | Ground Truth: [0] | prediction: [0.01870874]  |
Iter: 16         | Ground Truth: [1] | prediction: [0.99382538]  |
Iter: 17         | Ground Truth: [0] | prediction: [0.01834371]  |
Iter: 18         | Ground Truth: [1] | prediction: [0.99415511]  |
Iter: 19         | Ground Truth: [0] | prediction: [0.01819998]  |
Iter: 20         | Ground Truth: [1] | prediction: [0.99426877]  |
```

a. 實驗結果細節(GD):

```
-------Experiment Details -------

Data type: XOR data points
Activation: sigmoid
Learning rate: 0.1
Number of hidden layers: 2
Hidden units: 4
Optimizer: gd
Accuracy: 100.00%
Loss: 0.1496
-------------------------
```

b. Training Loss and Testing Result(Part)

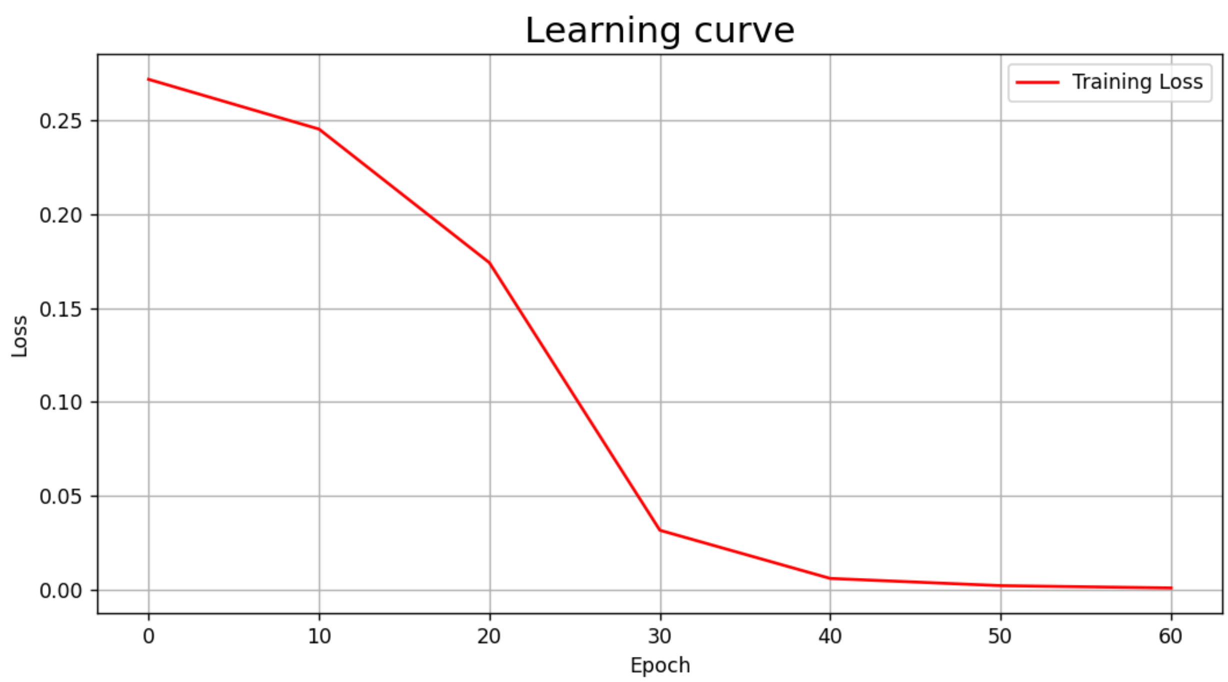
```
Epoch = 49890  Loss = [0.00146334]  Accuracy = 1.00
Epoch = 49900  Loss = [0.00146217]  Accuracy = 1.00
Epoch = 49910  Loss = [0.00146099]  Accuracy = 1.00
Epoch = 49920  Loss = [0.00145982]  Accuracy = 1.00
Epoch = 49930  Loss = [0.00145865]  Accuracy = 1.00
Epoch = 49940  Loss = [0.00145748]  Accuracy = 1.00
Epoch = 49950  Loss = [0.00145631]  Accuracy = 1.00
Epoch = 49960  Loss = [0.00145514]  Accuracy = 1.00
Epoch = 49970  Loss = [0.00145398]  Accuracy = 1.00
Epoch = 49980  Loss = [0.00145282]  Accuracy = 1.00
Epoch = 49990  Loss = [0.00145166]  Accuracy = 1.00
```

```
Iter: 9        | Ground Truth: [1] | prediction: [0.92821685]  |
Iter: 10       | Ground Truth: [0] | prediction: [0.03790754]  |
Iter: 11       | Ground Truth: [0] | prediction: [0.03765693]  |
Iter: 12       | Ground Truth: [1] | prediction: [0.94523835]  |
Iter: 13       | Ground Truth: [0] | prediction: [0.03739893]  |
Iter: 14       | Ground Truth: [1] | prediction: [0.97865566]  |
Iter: 15       | Ground Truth: [0] | prediction: [0.03713349]  |
Iter: 16       | Ground Truth: [1] | prediction: [0.98115412]  |
Iter: 17       | Ground Truth: [0] | prediction: [0.03686084]  |
Iter: 18       | Ground Truth: [1] | prediction: [0.98176202]  |
Iter: 19       | Ground Truth: [0] | prediction: [0.03658145]  |
Iter: 20       | Ground Truth: [1] | prediction: [0.98205938]  |
```

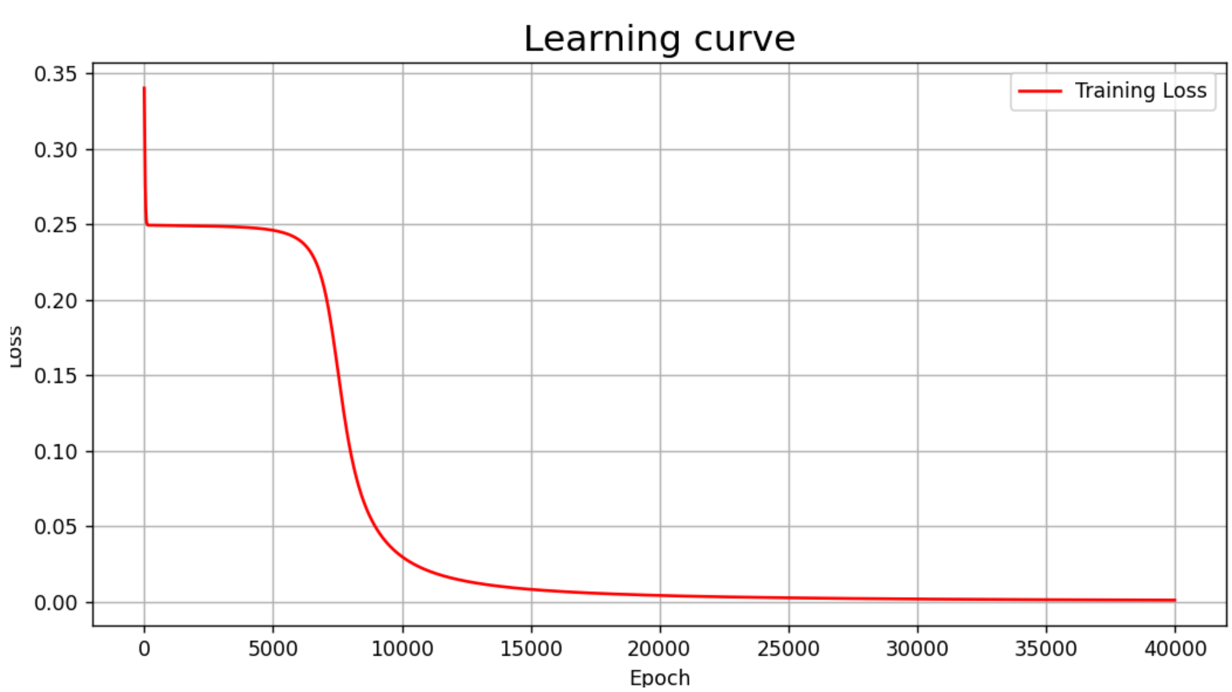
—Learning curve (loss, epoch curve)

(1)Linear Data:

(Lr: 0.1; activation: Sigmoid; optimizer: Adam)

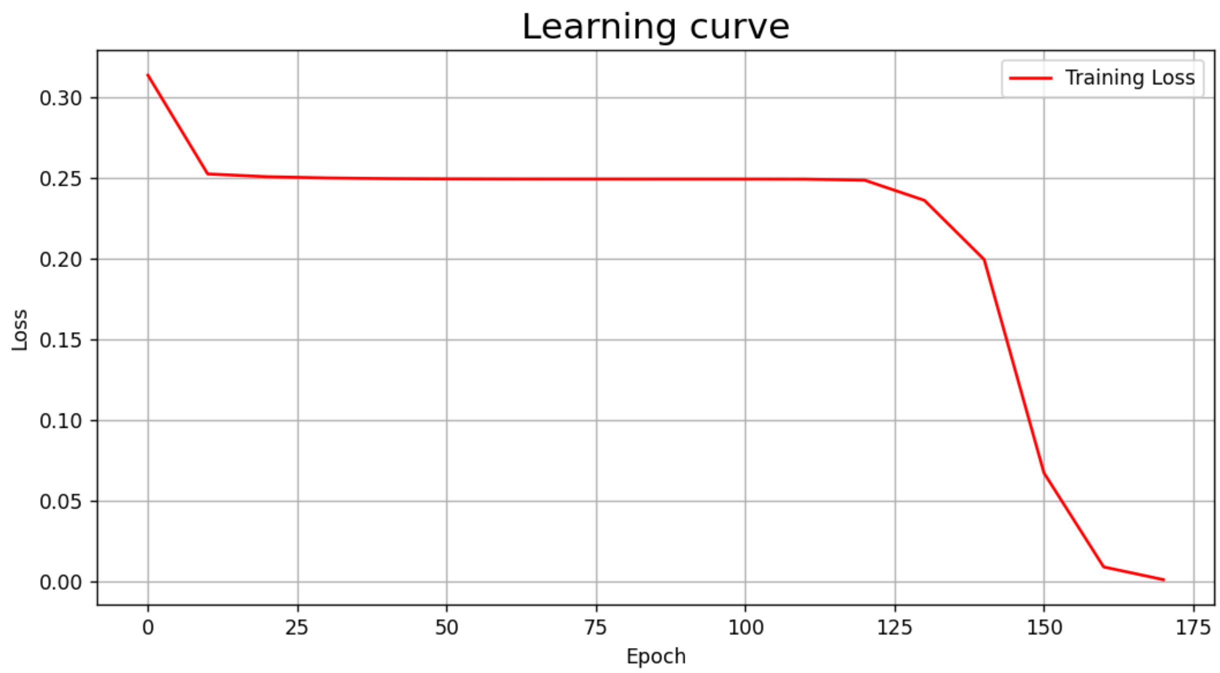


(Lr: 0.1; activation: Sigmoid; optimizer: GD)

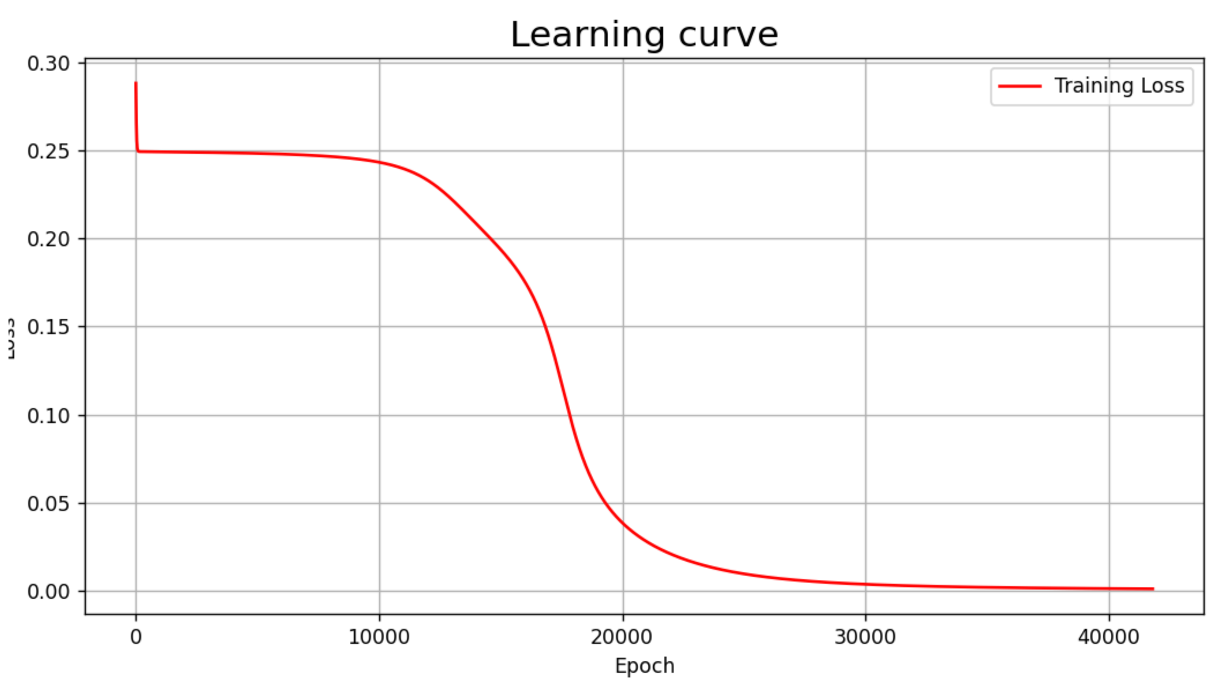


(2) XOR Data:

(Lr: 0.1; activation: Sigmoid; optimizer: Adam)



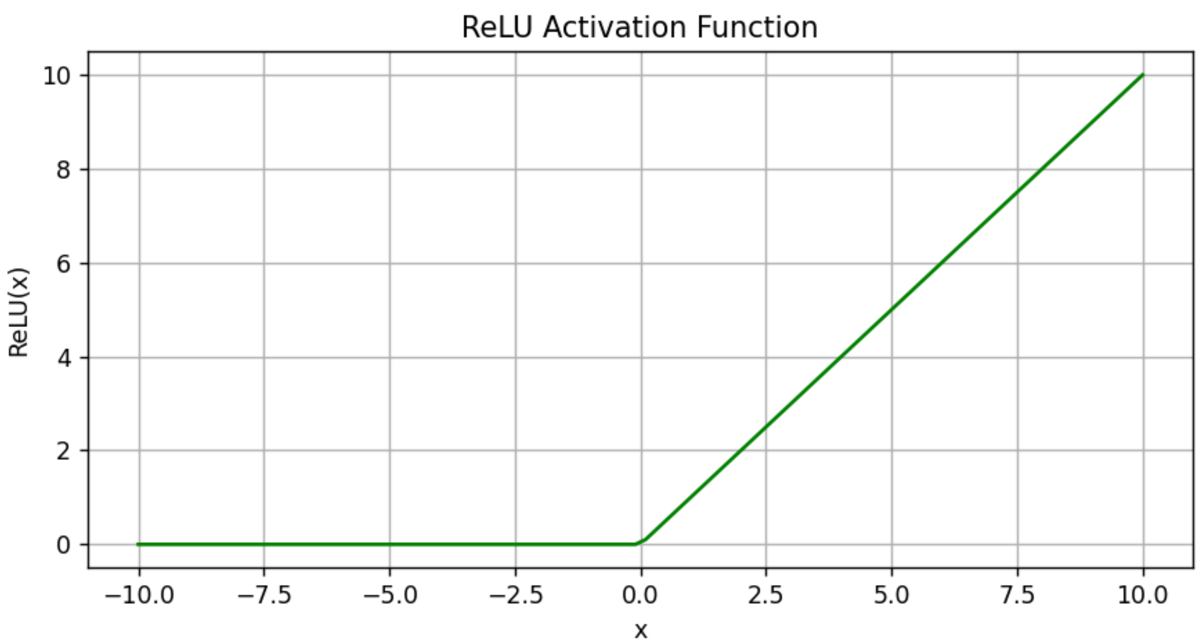
(Lr: 0.1; activation: Sigmoid; optimizer: GD)



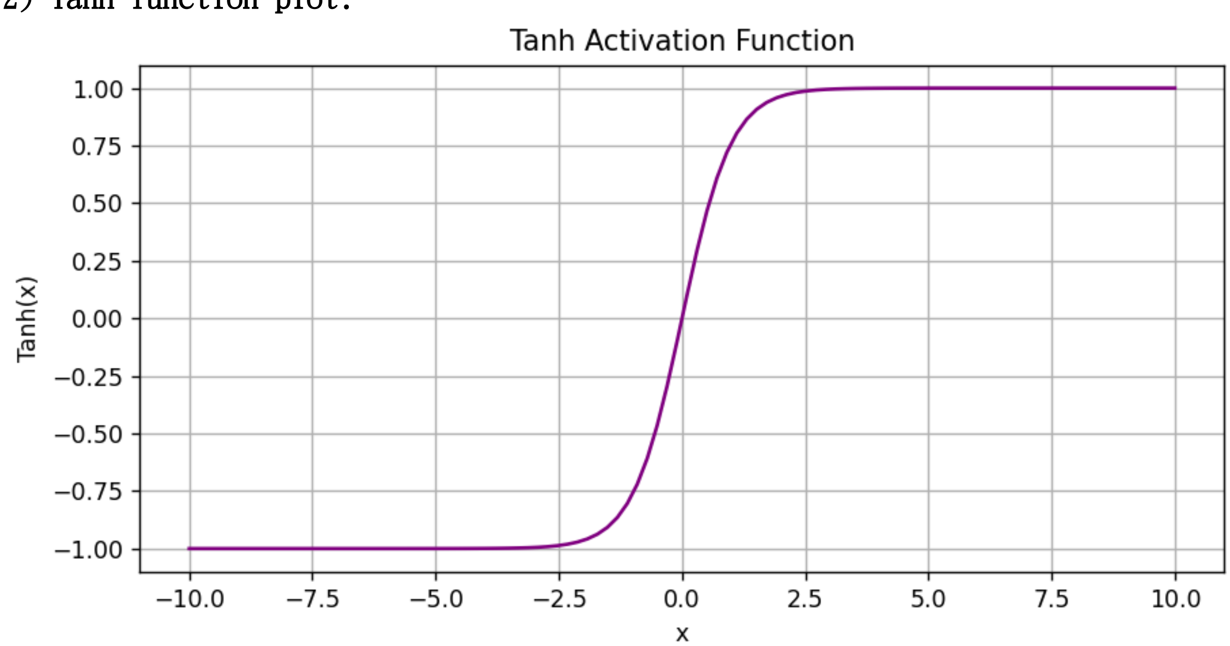
—Anything you want to present

(1) ReLU function plot:

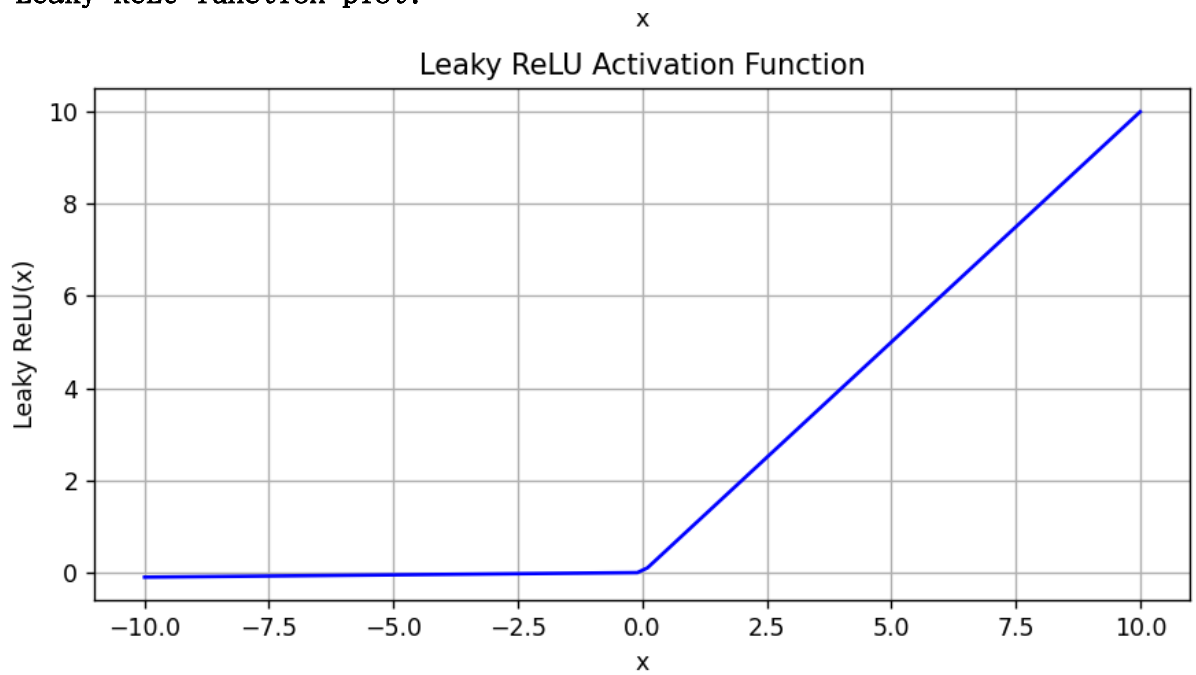
^



(2) Tanh function plot:



(3) Leaky ReLU function plot:



3. Discussion 做表

(1) Try different learning rates

本部分將Learning Rate分為1、0.1、0.01、0.001四種學習率進行比較其訓練結果，本部分在激活函數部分均使用Sigmoid。此外我們可看出當使用Adam學習率過大時(eg: Lr = 1時，可能會因為step過大導致無法有效達到minimum，造成較嚴重之誤判(如XOR)。較有趣的地方是使用GD時若是學習率較小如為0.001反而會有誤判的情況。不過當擁有適當的學習率時，整體而言使用Adam作為優化器，訓練的時間成本會較GD來的低。

1) Linear Data

Learning Rate	Epoch	Accuracy	Loss	Optimizer
1	128	96%	0.1812	Adam
0.1	189	100%	0.0813	Adam
0.01	1000	100%	0.0282	Adam
0.001	6627	100%	0.0645	Adam

Learning Rate	Epoch	Accuracy	Loss	Optimizer
1	2718	100%	0.0307	GD
0.1	62138	100%	0.0466	GD
0.01	172654	100%	0.1458	GD
0.001	1462379	99%	0.1367	GD

2) XOR

Learning Rate	Epoch	Accuracy	Loss	Optimizer
1	50000	52%	0.2495	Adam
0.1	232	100%	0.0859	Adam
0.01	676	100%	0.0855	Adam
0.001	5318	100%	0.0989	Adam

Learning Rate	Epoch	Accuracy	Loss	Optimizer
1	7357	100%	0.1512	GD
0.1	92691	100%	0.1661	GD
0.01	458878	100%	0.1612	GD
0.001	3110247	100%	0.1040	GD

(2) Try different numbers of hidden units

在此本部分在Learning Rate皆設0.001，激活函數部分均使用Sigmoid。且由上面的實驗結果，採用Adam較節省訓練時間成本，因此**優化器部分均選擇使用Adam**。

1) Linear:

Hidden Units	Epoch	Accuracy	Loss
4	6627	100%	0.0645
10	3597	100%	0.0299
50	1740	100%	0.0242

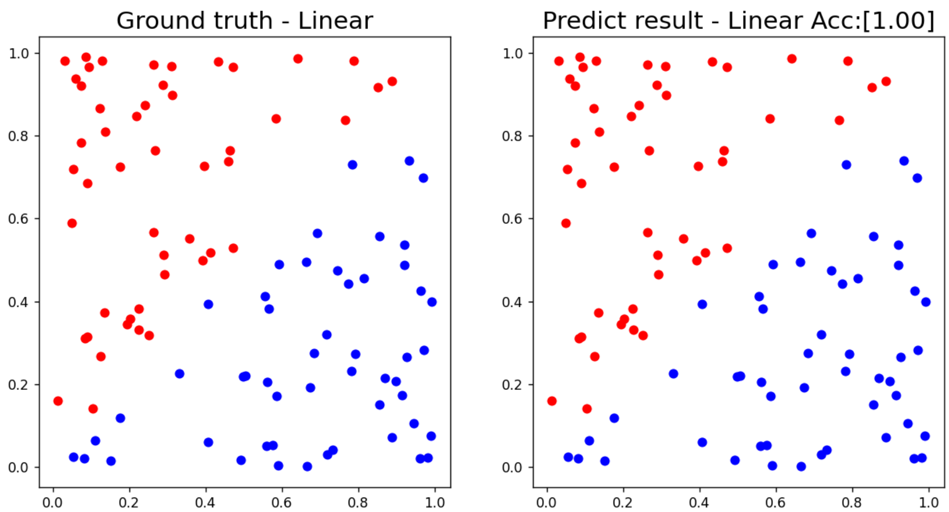
2)XOR

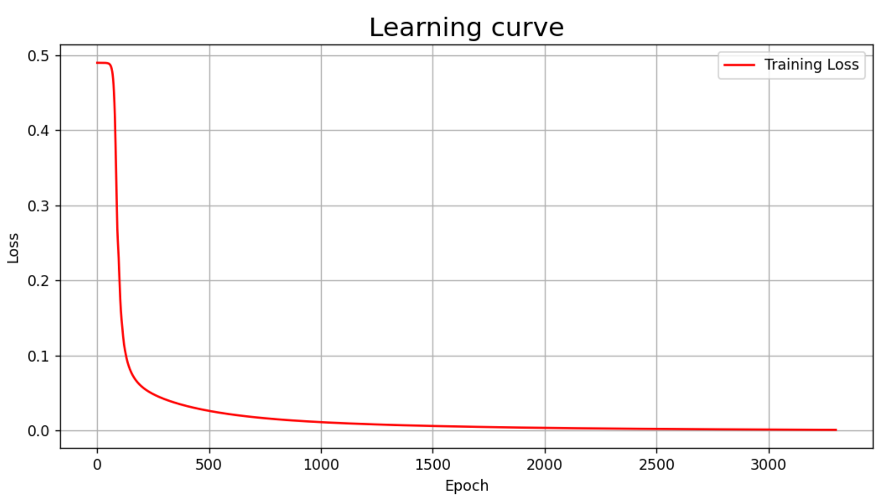
Hidden Units	Epoch	Accuracy	Loss
4	5318	100%	0.0989
10	3172	100%	0.0726
50	978	100%	0.0390

(3) Try without activation functions

a. 使用Adam作為優化器的Linear Data:

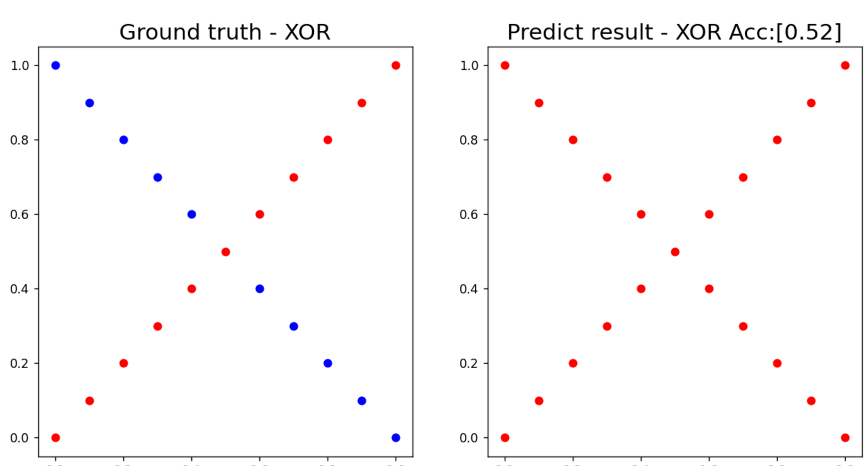
Epoch	Accuracy	Loss	Optimizer
3820	100%	0.0238	Adam

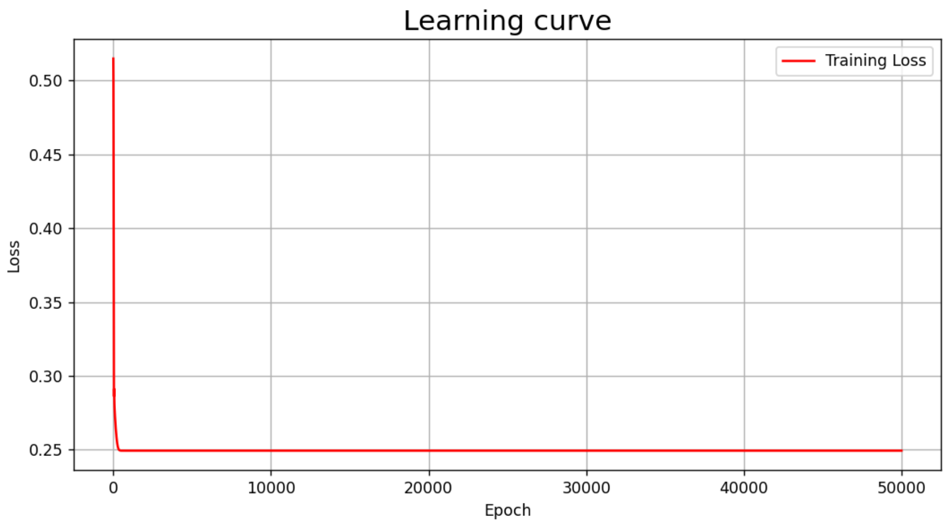




b. 使用Adam作為優化器的XOR Data:(Not work)

Epoch	Accuracy	Loss	Optimizer
50000	52%	0.2497	Adam





說明：
因為activation function主要是引入非線性，使得model能夠學習非線性問題。但由於缺乏activation會導致整個model僅能處理線性問題，故無法有效處理XOR問題，因為XOR本身即是線性不可分。

5. Extra

(1) Implement different optimizers:

Lr = 0.01、Activation : Sigmoid

```
gradient = self.forward_gradient.T @ self.backward_gradient

if self.optimizer == 'adam':
    self.moving_average_m = 0.9 * self.moving_average_m + 0.1 * gradient
    self.moving_average_v = 0.999 * self.moving_average_v + 0.001 * np.square(gradient)
    bias_correction_m = self.moving_average_m / (1.0 - 0.9 ** self.update_times)
    bias_correction_v = self.moving_average_v / (1.0 - 0.999 ** self.update_times)
    self.update_times += 1
    delta_weight = -self.learning_rate * bias_correction_m / (np.sqrt(bias_correction_v) + 1e-8)

elif self.optimizer == 'adagrad':
    self.sum_of_squares_of_gradients += np.square(gradient)
    delta_weight = -self.learning_rate * gradient / np.sqrt(self.sum_of_squares_of_gradients + 1e-8)

elif self.optimizer == 'momentum':
    self.momentum = 0.9 * self.momentum - self.learning_rate * gradient
    delta_weight = self.momentum

elif self.optimizer == 'gd':
    delta_weight = -self.learning_rate * gradient

else: # Default to Gradient Descent
    delta_weight = -self.learning_rate * gradient

self.weight += delta_weight
```

a. Linear

Optimizer	Epoch	Accuracy	Loss
Adam	1005	**100%**	0.0377
adagrad	**50000**	**100%**	0.0337
momentum	**50000**	**100%**	0.0425
GD	**50000**	**98%**	0.2110

b. XOR

Optimizer	Epoch	Accuracy	Loss
Adam	595	**100%**	0.0999
adagrad	**50000**	**100%**	0.0590
momentum	**50000**	**100%**	0.1522
GD	**50000**	**52%**	0.2499

(2) Implement different activation functions.

```python
    @staticmethod
    def tanh(x: np.ndarray) -> np.ndarray:
        return np.tanh(x)

    @staticmethod
    def derivative_tanh(y: np.ndarray) -> np.ndarray:
        return 1.0 - y ** 2

    @staticmethod
    def relu(x: np.ndarray) -> np.ndarray:
        """Calculate relu function."""
        return np.maximum(0.0, x)

    @staticmethod
    def derivative_relu(y: np.ndarray) -> np.ndarray:
        """Calculate the derivative of relu function."""
        return np.heaviside(y, 0.0)
```

```python
    @staticmethod
    def leaky_relu(x: np.ndarray) -> np.ndarray:
        """Calculate leaky relu function."""
        return np.maximum(0.0, x) + 0.01 * np.minimum(0.0, x)

    @staticmethod
    def derivative_leaky_relu(y: np.ndarray) -> np.ndarray:
        """Calculate the derivative of leaky relu function."""
        y[y > 0.0] = 1.0
        y[y <= 0.0] = 0.01
        return y
```

a. Linear

Activation	Epoch	Accuracy	Loss
tanh	2135	**100%**	0.0614
relu	2704	**100%**	0.1039
Leaky relu	4730	**100%**	0.0482

b. XOR

Activation	Epoch	Accuracy	Loss
tanh	2364	**100%**	0.0770
relu	3582	**100%**	0.0401
Leaky relu	4142	**100%**	0.0927