Deep Learning HW1

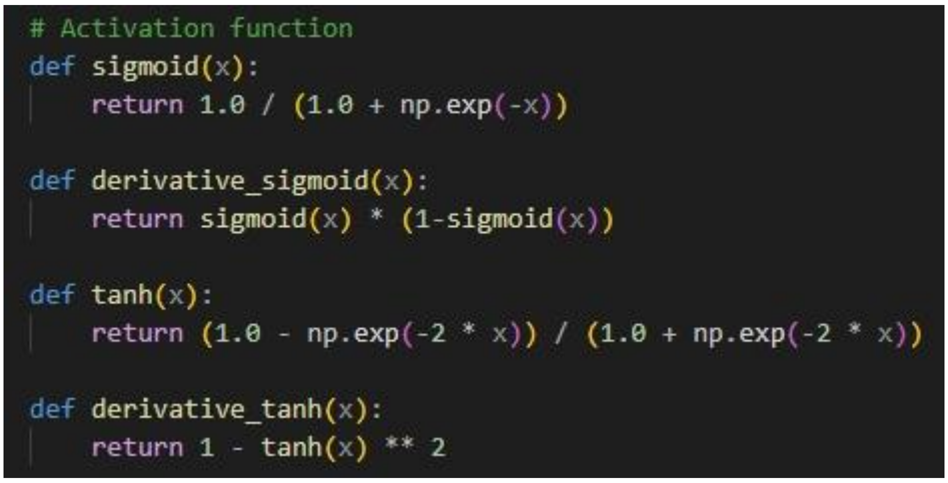
312554054 林辰翰

1. Introduction

Lab1 : Implement a neural network without using PyTorch, which must contain the following:

- Implement a simple neural network with two hidden layers.
- Each hidden layer needs to contain at least one transformation and one activation function.
- Must use backpropagation in this neural network and can only use Numpy and other Python standard libraries for implementation.
- Plot a comparison figure that shows the predicted results and the ground truth.
- Be sure that the output meets the requirement.
- Use datasets returned from the functions "generate_linear" and "generate_XOR_easy".

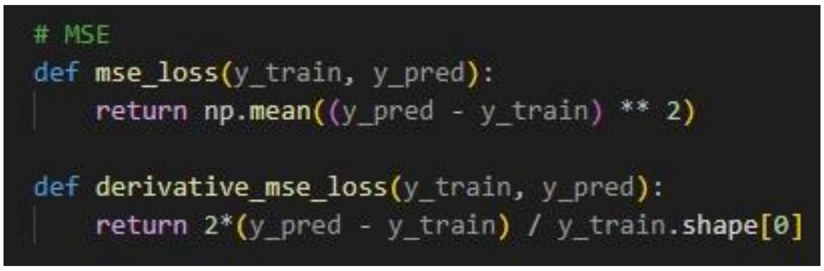
2. Activation function(Sigmoid functions)



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

def derivative_sigmoid(x):
    return sigmoid(x) * (1-sigmoid(x))

def tanh(x):
    return (1.0 - np.exp(-2 * x)) / (1.0 + np.exp(-2 * x))

def derivative_tanh(x):
    return 1 - tanh(x) ** 2
```

In the instruction, we are supposed to use sigmoid functions. For the extra point, I've implemented tanh functions for different activation function.

3. MSE



```
# MSE
def mse_loss(y_train, y_pred):
    return np.mean((y_pred - y_train) ** 2)

def derivative_mse_loss(y_train, y_pred):
    return 2*(y_pred - y_train) / y_train.shape[0]
```

Calculates the average of the squared differences between predicted values and actual values.

The derivative one is used for the backpropagation to update the model's weights.

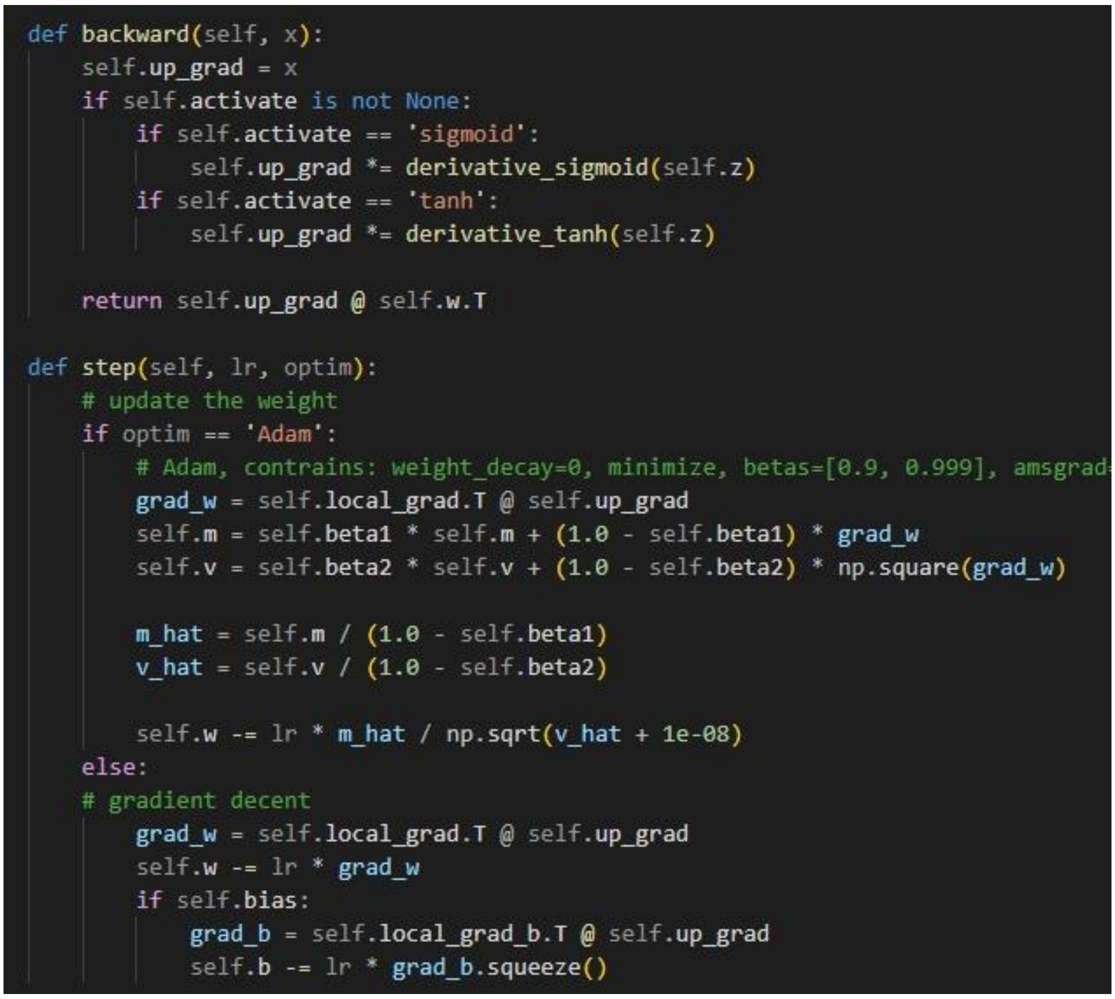
4. Neural Network

In the instructions, we are supposed to implement a simple neural network with two hidden layers.

In this part, I divide the architecture in two parts, "Layers" controls all operations and "Network" passes the result of forward propagation and backward propagation.

(Details have shown in the code)

5. Backpropagation



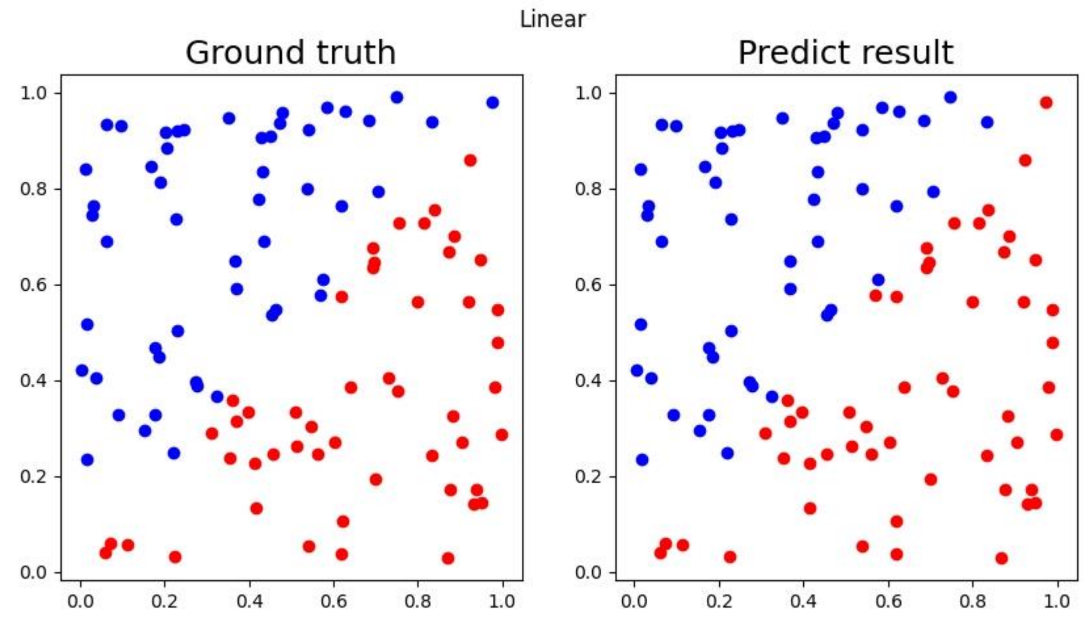
```python
def backward(self, x):
    self.up_grad = x
    if self.activate is not None:
        if self.activate == 'sigmoid':
            self.up_grad *= derivative_sigmoid(self.z)
        if self.activate == 'tanh':
            self.up_grad *= derivative_tanh(self.z)

    return self.up_grad @ self.w.T

def step(self, lr, optim):
    # update the weight
    if optim == 'Adam':
        # Adam, contrains: weight_decay=0, minimize, betas=[0.9, 0.999], amsgrad
        grad_w = self.local_grad.T @ self.up_grad
        self.m = self.beta1 * self.m + (1.0 - self.beta1) * grad_w
        self.v = self.beta2 * self.v + (1.0 - self.beta2) * np.square(grad_w)

        m_hat = self.m / (1.0 - self.beta1)
        v_hat = self.v / (1.0 - self.beta2)

        self.w -= lr * m_hat / np.sqrt(v_hat + 1e-08)
    else:
    # gradient decent
        grad_w = self.local_grad.T @ self.up_grad
        self.w -= lr * grad_w
        if self.bias:
            grad_b = self.local_grad_b.T @ self.up_grad
            self.b -= lr * grad_b.squeeze()
```

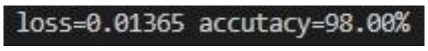
During training, backpropagation is used to compute the gradient of each layer with respect to the loss function. It will receive the gradient from the next layer. These gradients are multiplied by the derivative of the activation function, then it calculates and returns the gradients to be passed back to the previous layer.

6. Result

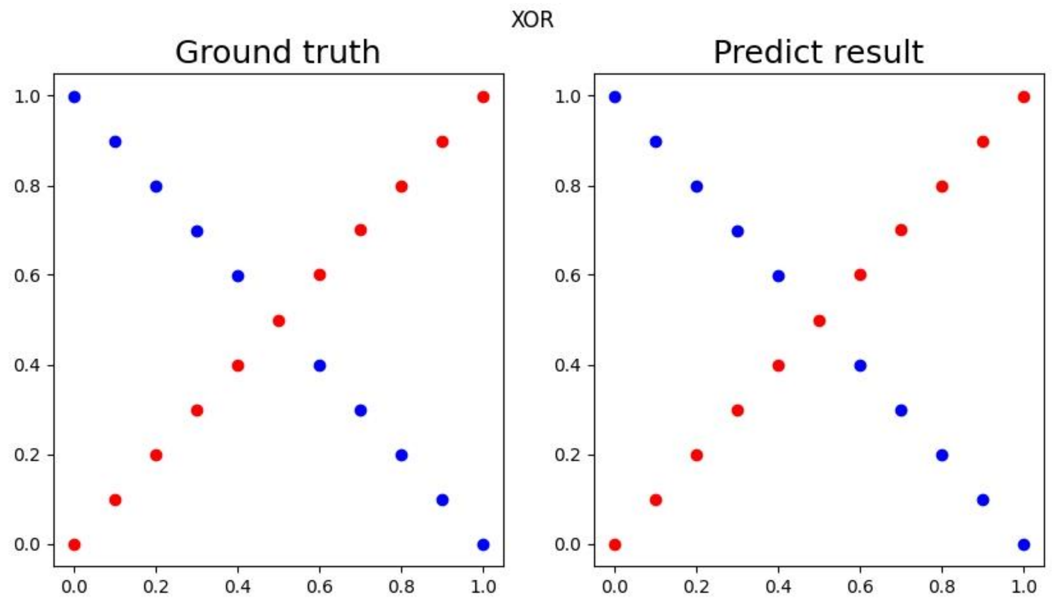
Default settings:

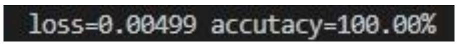
- Learning rate: 1.0.
- Optimizer: Gradient Descent
- Batch size: x_train.shape[0] (Linear: 100, XOR: 21)
- Hidden layer size: 8 (2 hidden layers)
- Training epsilon: 0.005
- Maximum epoch: 100000
- 1 epoch = 500 steps





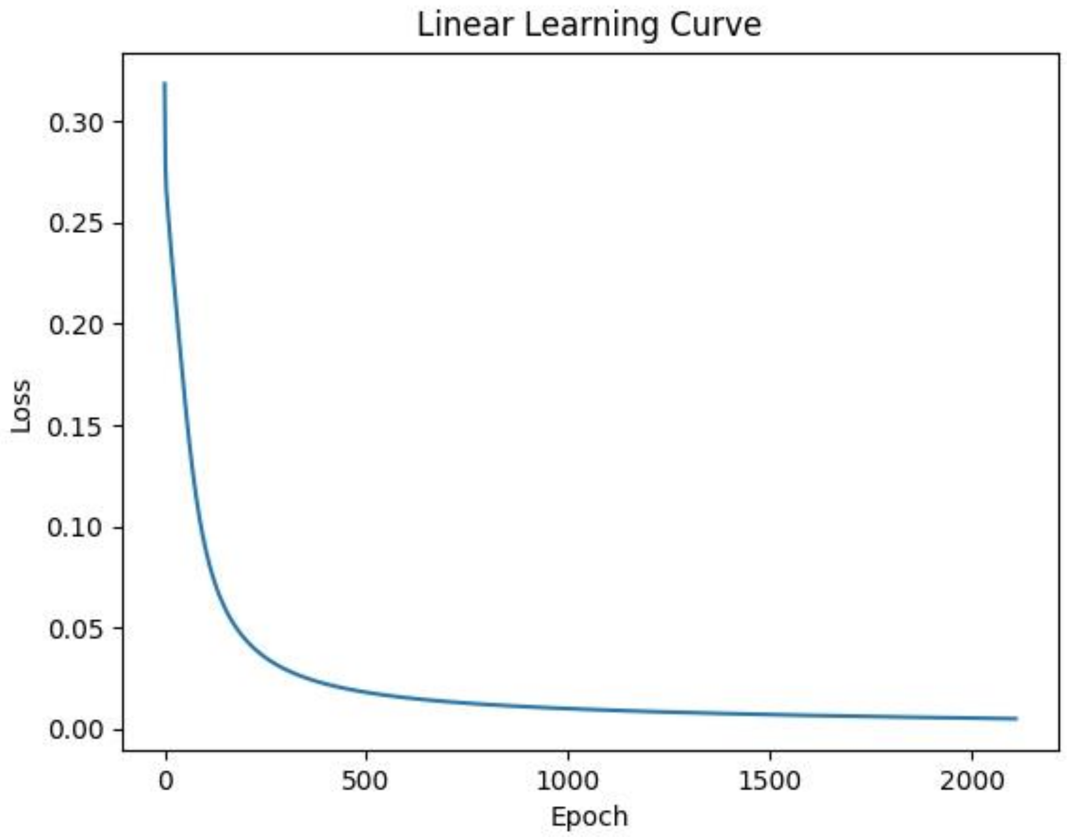
```
loss=0.01365 accutacy=98.00%
```

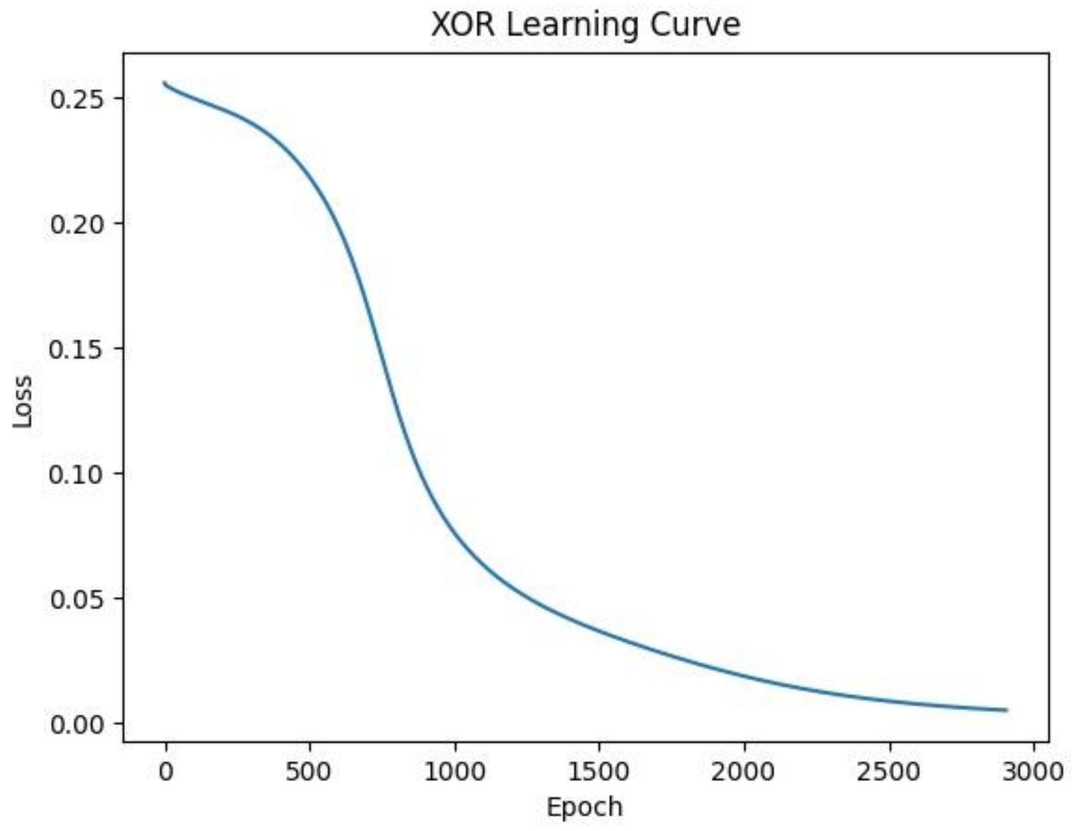




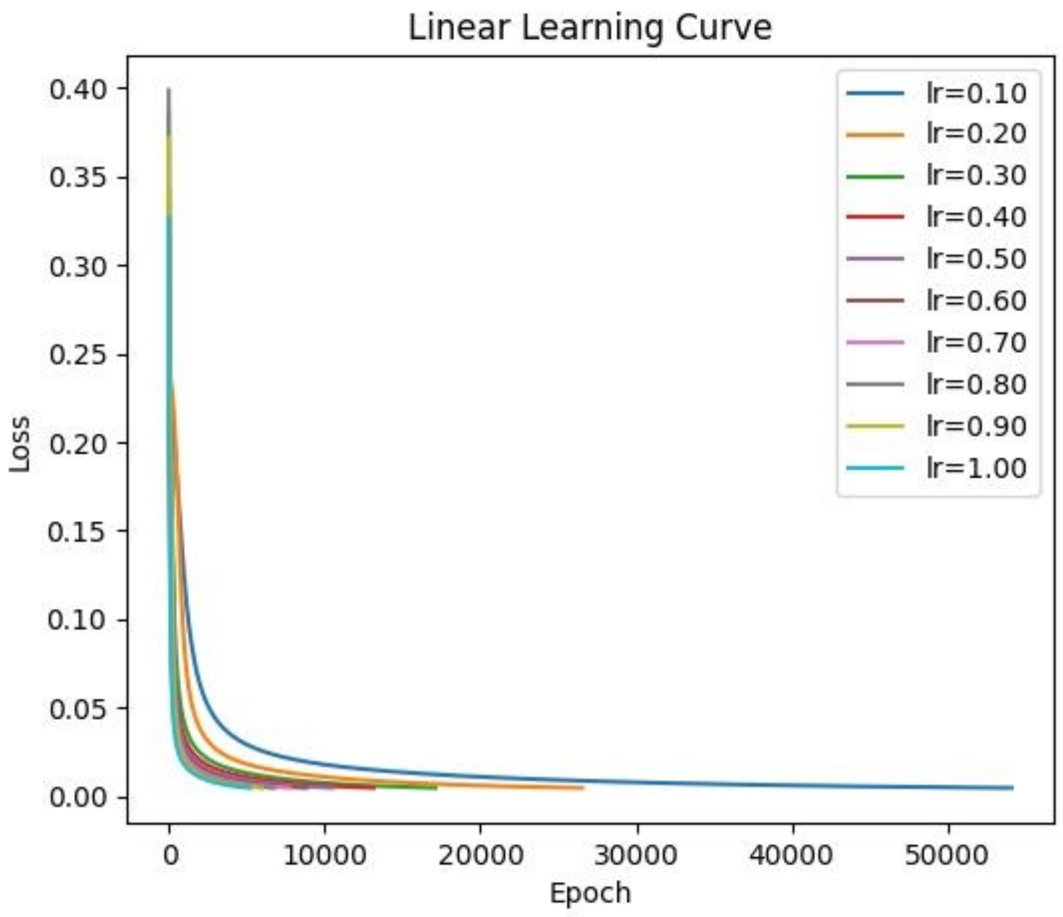
```
loss=0.00499 accutacy=100.00%
```

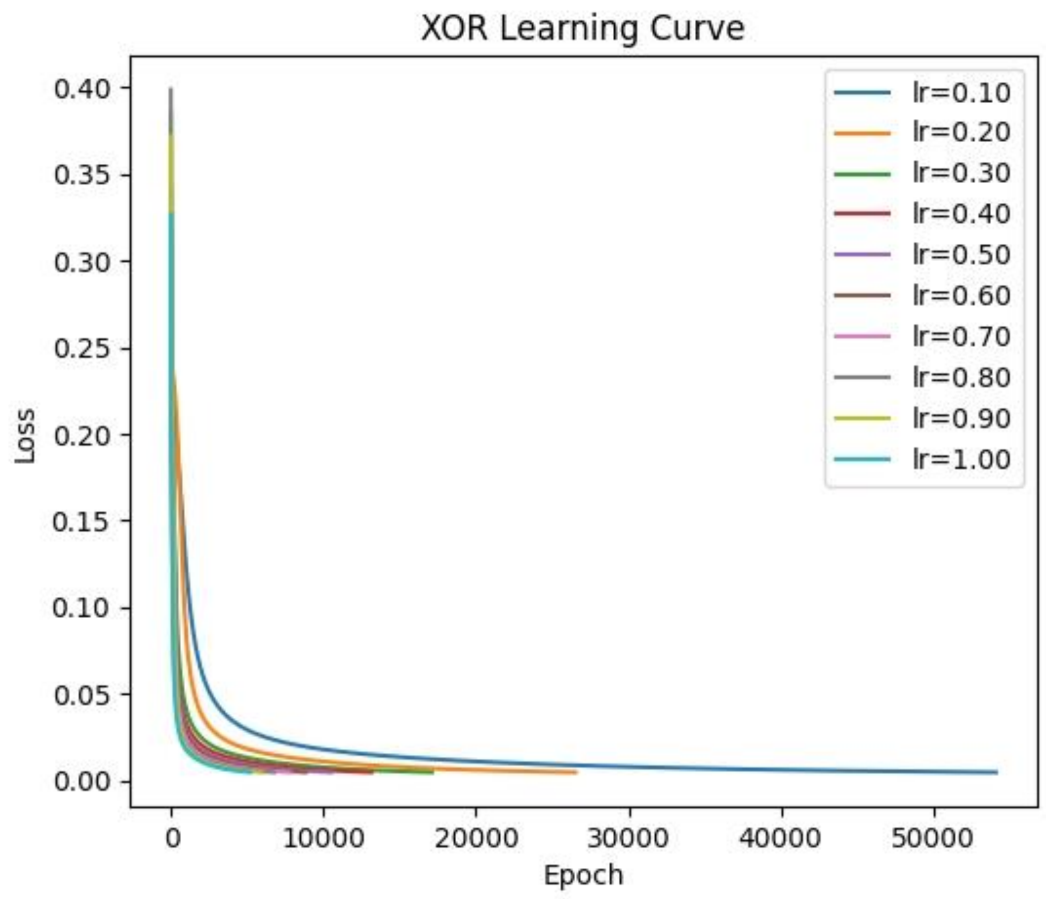
Learning curve:


Linear Learning Curve
Loss
Epoch
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
500
1000
1500
2000


XOR Learning Curve
Loss
Epoch
0.25
0.20
0.15
0.10
0.05
0.00
0
500
1000
1500
2000
2500
3000

Different learning rate:





In the scenario depicted above, regardless of whether it is a linear or XOR problem, the training results improves as the learning rate increases from 0.1 to 1. This suggests that in this case, a higher learning rate leads to better training performance.

Different numbers of hidden units:

ring

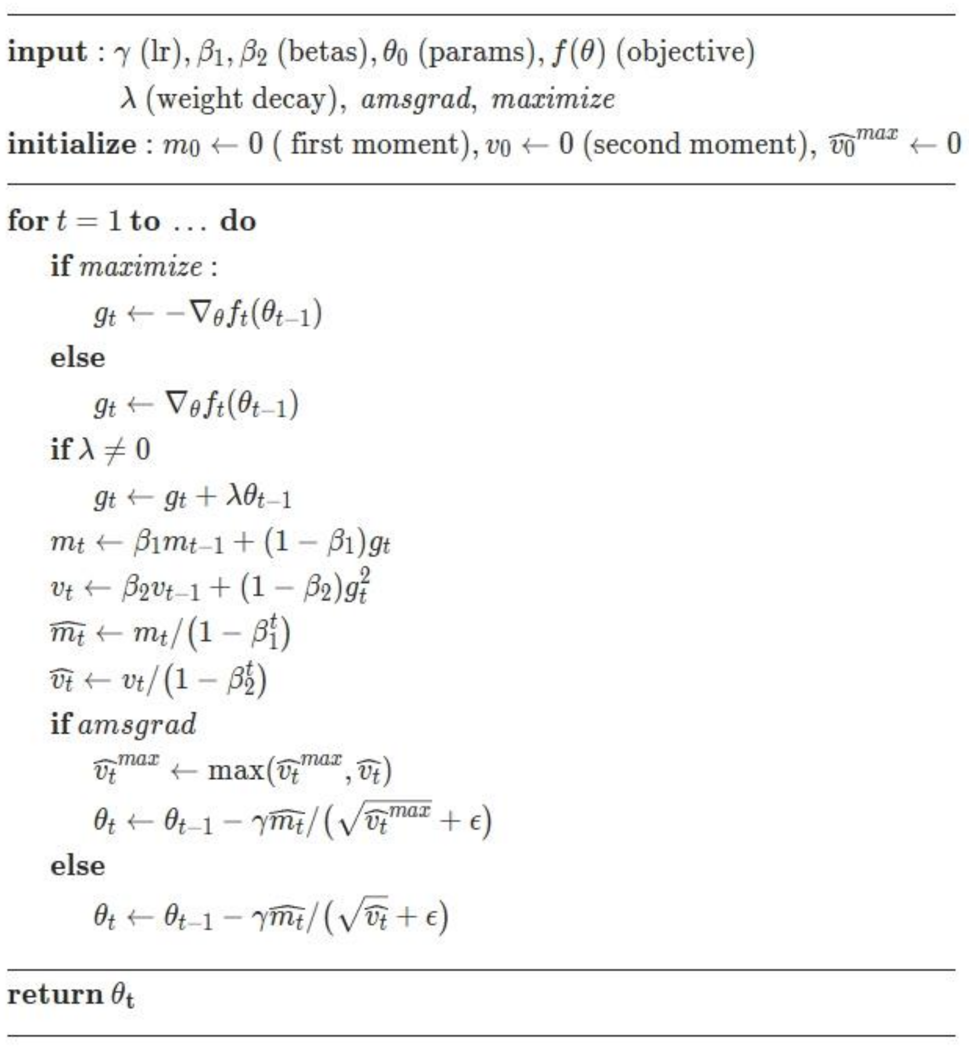
Without activation functions:

If there are no activation functions, the entire model becomes linear. This means the model will use linear regression to solve the problem. However, linear regression requires the assumption that the samples are normally distributed. In our dataset, the samples are binary distributed, so the model will not train properly.

Others:

Different optimizers (Adam):

In the field of machine learning, the most famous and the first optimizer that comes to mind is "Adam". The algorithm is attached below:



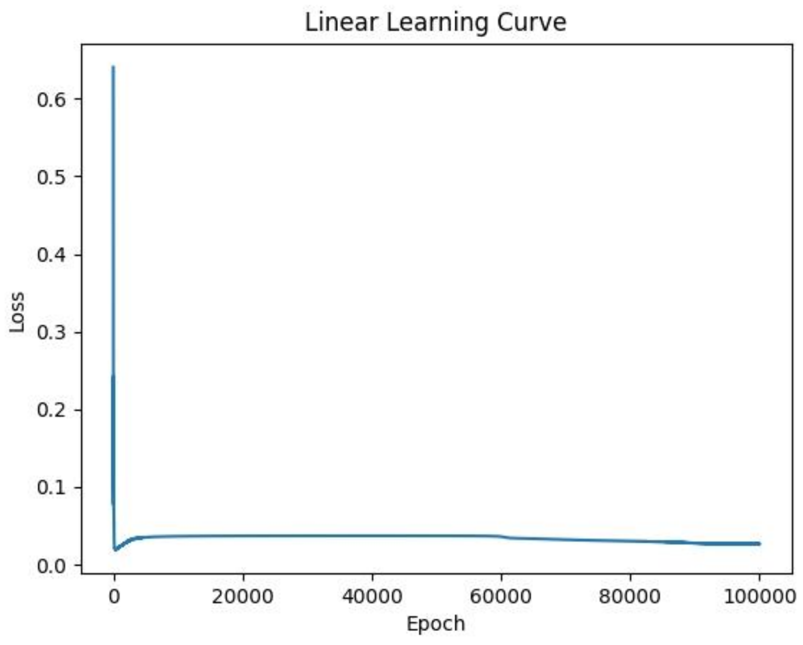
```
**input** : $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_0$ (params), $f(\theta)$ (objective)
            $\lambda$ (weight decay), $amsgrad$, $maximize$
**initialize** : $m_0 \leftarrow 0$ ( first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

**for** $t = 1$ **to** $\ldots$ **do**
    **if** $maximize$ :
        $g_t \leftarrow -\nabla_{\theta} f_t(\theta_{t-1})$
    **else**
        $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$
    **if** $\lambda \neq 0$
        $g_t \leftarrow g_t + \lambda \theta_{t-1}$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big)$
    $\widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big)$
    **if** $amsgrad$
        $\widehat{v_t}^{max} \leftarrow \mathrm{max}(\widehat{v_t}^{max}, \widehat{v_t})$
        $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
    **else**
        $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

**return** $\theta_t$
```

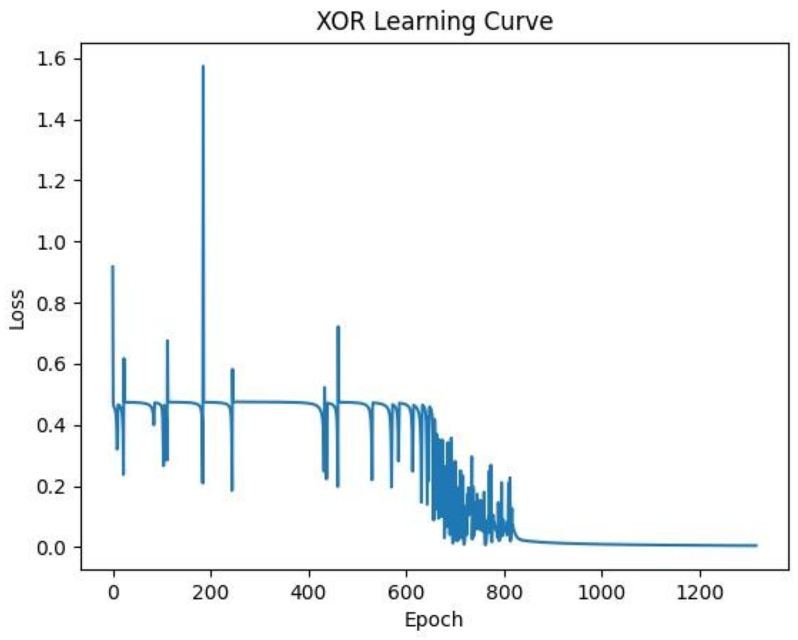
(Reference: Adam — PyTorch 2.2 documentation)

- This optimizer is used for minimize the loss
- Weight decay = 0
- Betas = [0.9, 0.999]
- Amsgrad = False
- Epsilon = 1e-8

Different activation functions:

Due to the binary classification, tanh function would be a good way to implement.


Linear Learning Curve
Loss
Epoch


XOR Learning Curve
Loss
Epoch