Lab 1: Back Propagation

1. Introduction

   In this lab, we need to use numpy and other python standard library to implement neural network and back-propagation. I use a modular approach to complete this lab, and the implementation includes:

   a. Base Class **Layer**: an abstract class to replace __call__ method with .forward()

```python
class Layer:
    def __call__(self, *args, **kwargs):
        return self.forward(*args, **kwargs)

    def forward(self, *args, **kwargs):
        pass

    def backward(self, *args, **kwargs):
        pass
```

   b. **Normal Module:** Sequential, Linear
   c. **Activation:** Sigmoid, Tanh, SoftSign
   d. **Optimizer:** SGD, Adam

2. Implementation Details

   a. Sigmoid Function

For sigmoid function, its definition is $\sigma(x) = \frac{1}{1+e^{-x}}$, and split it into two function to prevent overflow:

$$\sigma(x) = \begin{cases} \dfrac{e^x}{1+e^x}, \text{if } x < 0 \\ \dfrac{1}{1+e^{-x}}, \text{if } x \geq 0 \end{cases}$$

When we need to calculate the gradient, we can use the provided hint to get

$$\nabla\sigma(x) = \sigma(x) * \big(1 - \sigma(x)\big)$$

And the implementation is really intuitive:

```
class Sigmoid(Layer):
    def __init__(self):
        self.output = None

    def forward(self, x):
        self.output = np.piecewise(
            x,
            [x < 0, x >= 0],
            [lambda x: np.exp(x) / (1 + np.exp(x)), lambda x: 1 / (1 +
np.exp(-x))],
        )
        return self.output

    def backward(self, grad_output):
        return grad_output * np.multiply(self.output, 1 - self.output)
```

b. Neural network architecture

I will explain the neural network in two parts

1. **Linear**:

   For Linear, we just do two things, forward and backward. Forward is easy, just multiply the input with the weights and plus the bias.

   For backward, the gradient of current layer's weights is gradient from the next layer multiplied by the inputs, and the gradient of bias is just the sum of the gradient from next layer (since we can see it as a vector with ones). After the calculation, the gradient is multiplied with current weights, which represents the gradient of the

output with respect to the layers from current layer.

Implementation:

```
class Linear(Layer):
    def __init__(self, input_dim, output_dim):
        self.weights = np.random.randn(input_dim, output_dim)
        self.biases = np.zeros(output_dim)
        self.grad_weights = None
        self.grad_biases = None
        self.x = None

    def forward(self, x):
        self.x = x
        return self.x @ self.weights + self.biases

    def backward(self, grad_output):
        self.grad_weights = self.x.T @ grad_output
        self.grad_biases = np.sum(grad_output, axis=0)
        grad_input = grad_output @ self.weights.T
        return grad_input
```

2. **Sequential**:

Sequential is a module which connects the layers in a list, and a convenient interface to calculate the gradient of each layer.

Forward is also easy to implement, just stream the input into all the layers.

Backward is only a little more complicated, we need to get the gradient of the loss, and stream it to the layers in a reverse way.

Implementation:

```python
class Sequential(Layer):
    def __init__(self, layers):
        self.layers = layers

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def backward(self, grad_output):
        for layer in reversed(self.layers):
            grad_output = layer.backward(grad_output)
        return grad_output
```

c. Back-propagation

Back propagation is easy given the modules above, since they've handled each forward and backward, and calculate the gradients for us. So, the last thing we need to do is define the **loss** and implement its forward and backward, then use an **optimizer** to update.

For the **loss**, I choose MSELoss since this task can be seen as a regression task, and the calculation and gradient of MSELoss is really intuitive and easy to implement:

$$L(\hat{y}, y) = (\hat{y} - y)^2$$
$$\nabla L(\hat{y}, y) = 2(\hat{y} - y)$$

Implementation:

```python
class MSELoss(Layer):
    def forward(self, y_pred, y):
        return np.mean((y_pred - y) ** 2)

    def backward(self, y_pred, y):
        return 2 * (y_pred - y) / y.shape[0]
```

For the substract, I implemented a simple SGD, which is just subtract gradient * learning from the weight and bias for each layer.

Implementation:

```
class SGD:
    def __init__(self, model, lr=1e-2):
        self.model = model
        self.lr = lr

    def step(self):
        for layer in self.model.layers:
            if hasattr(layer, "weights"):
                layer.weights -= self.lr * layer.grad_weights
                layer.biases -= self.lr * layer.grad_biases
```

3. Experimental Results

   The base experiments are set with the following hyper-parameters:

   a. Network structure:

```
class Model:
    def __init__(self, optimizer="SGD", lr=1e-2, hidden_1=128, hid-
den_2=64, activation="Sigmoid"):
        if activation == "Sigmoid":
            activation = Sigmoid()
        elif activation == "Tanh":
            activation = Tanh()
        elif activation == "SoftSign":
            activation = SoftSign()

        layers = [
            Linear(2, hidden_1),
            activation,
            Linear(hidden_1, hidden_2),
            activation,
            Linear(hidden_2, 1),
            activation,
        ]
        self.nn = Sequential(layers)
        self.loss = MSELoss()
        if optimizer == "Adam":
            self.optimizer = Adam(self.nn, lr=lr)
        else:
            self.optimizer = SGD(self.nn, lr=lr)
```

   b. Loss function: MSE Loss

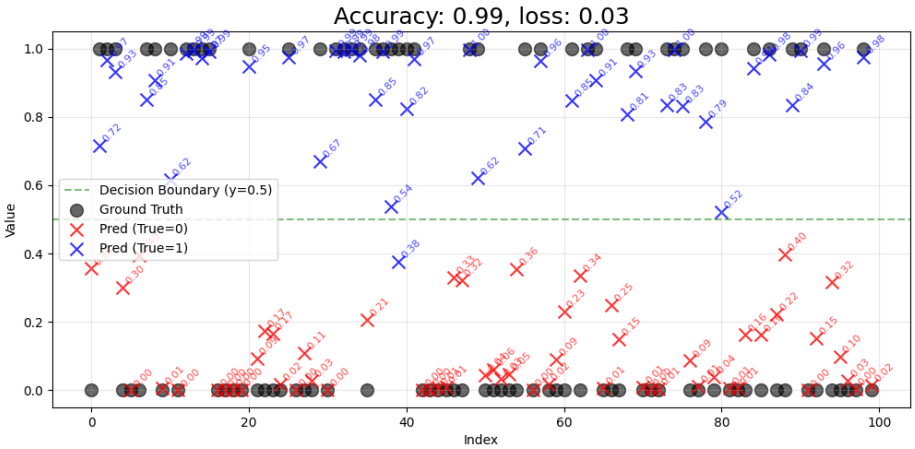
   c. Optimizer: SGD

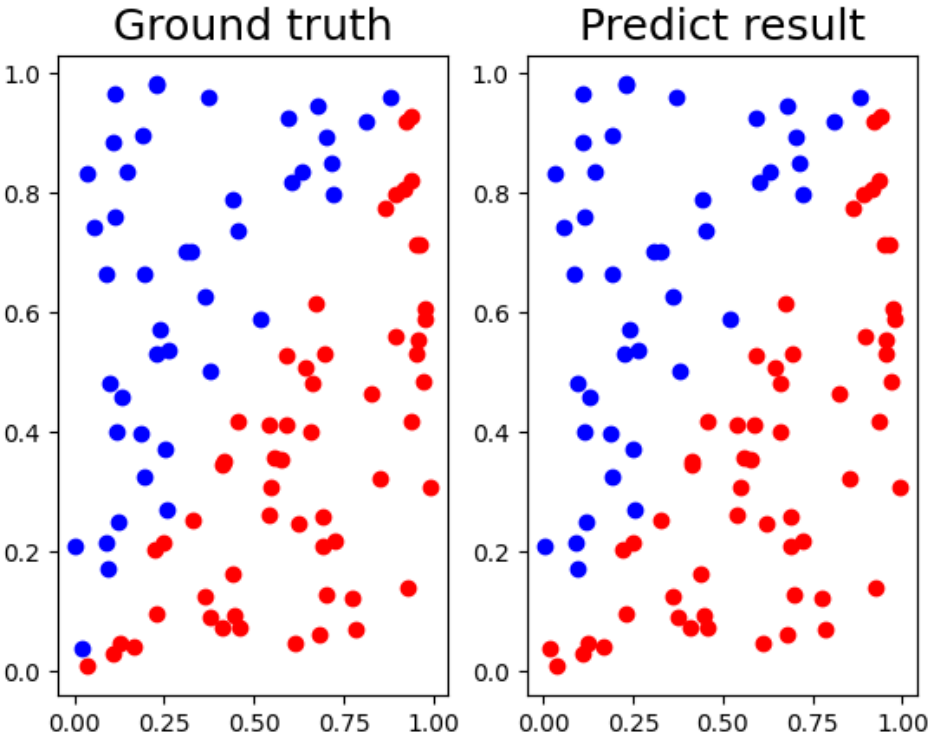
   d. Learning rate: 0.01

   Results:

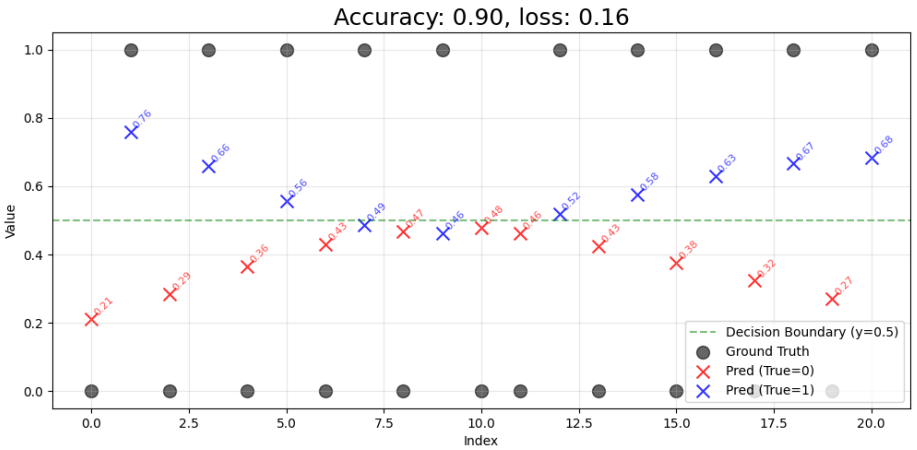
a. Prediction (Accuracy and Loss on the title)

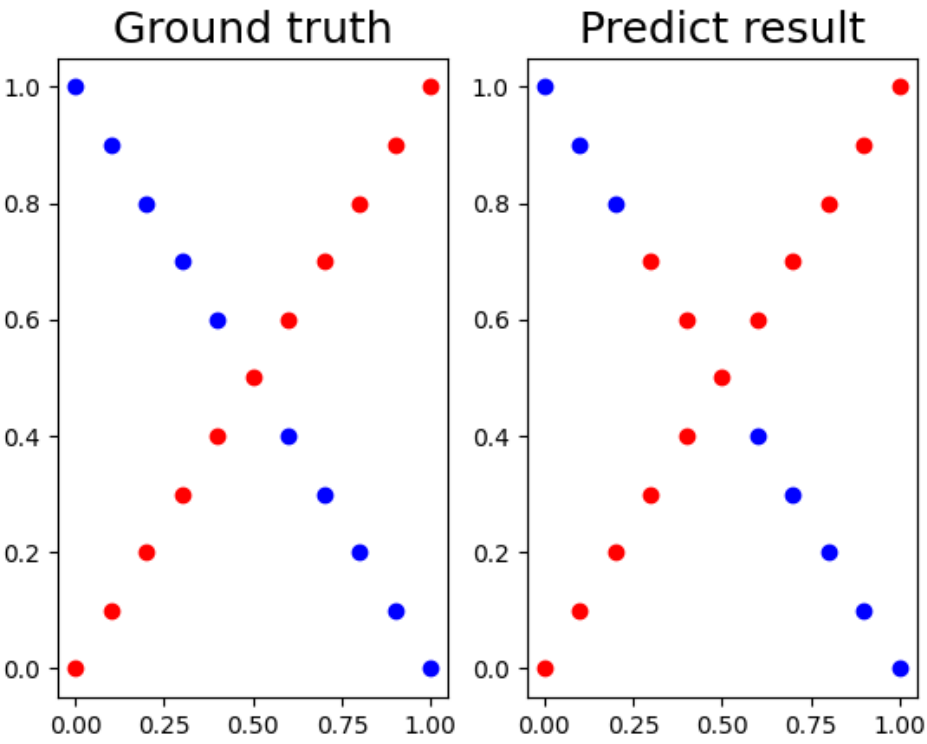
a. Linear





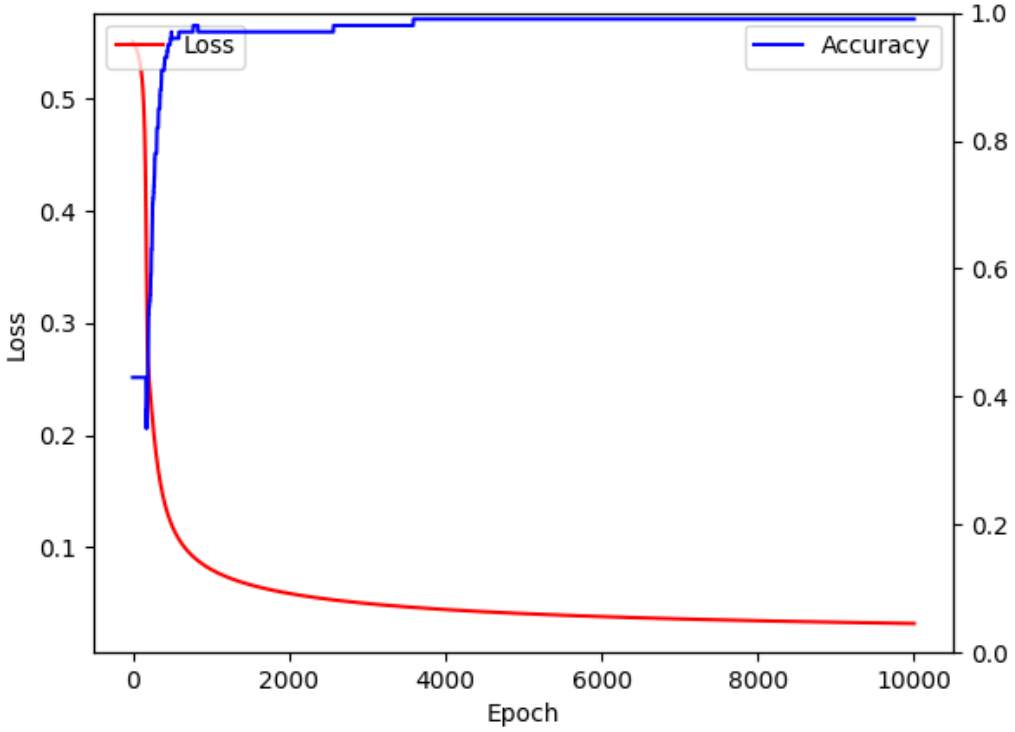
b. XOR:



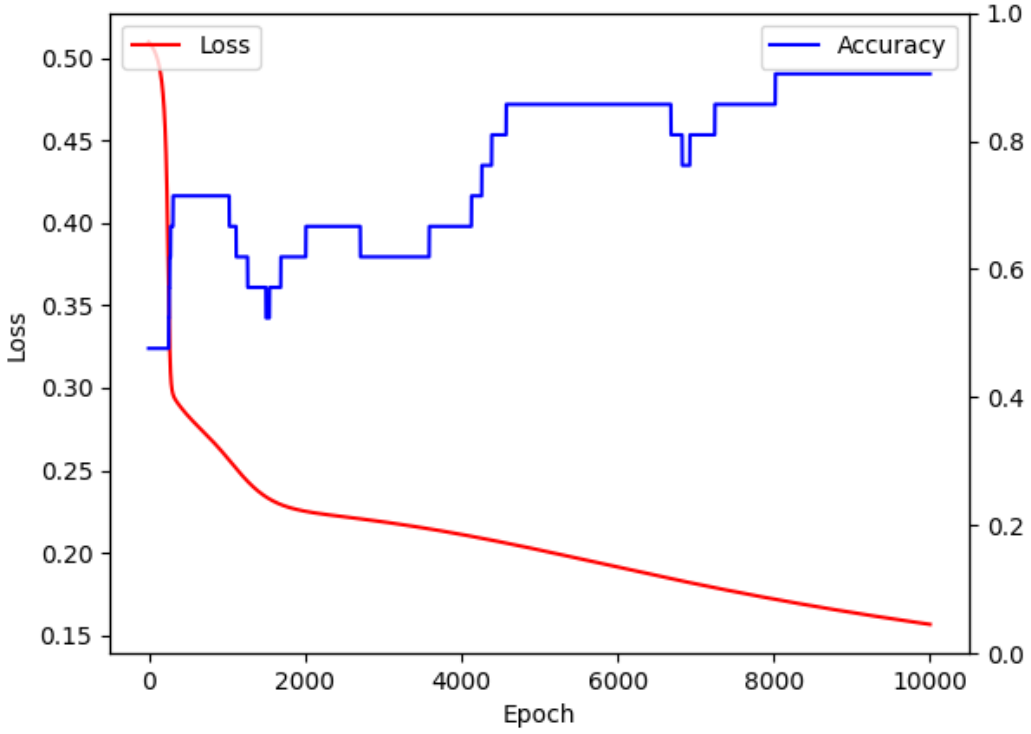


b. Learning curve

a. Linear:

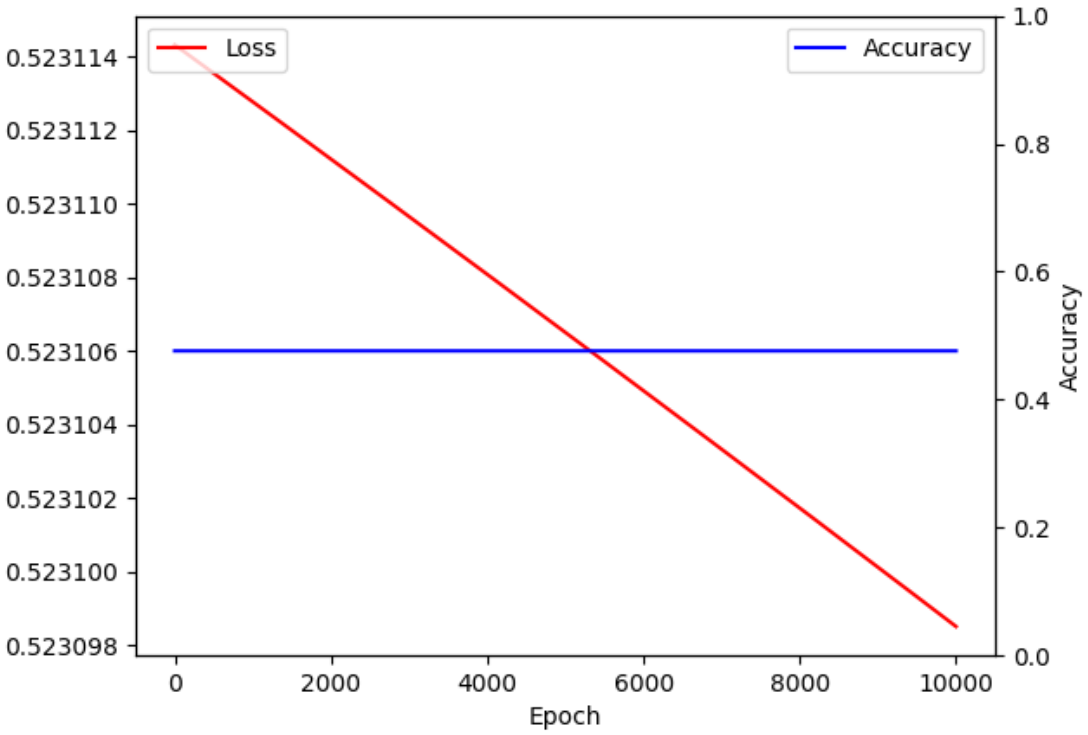


b. XOR:

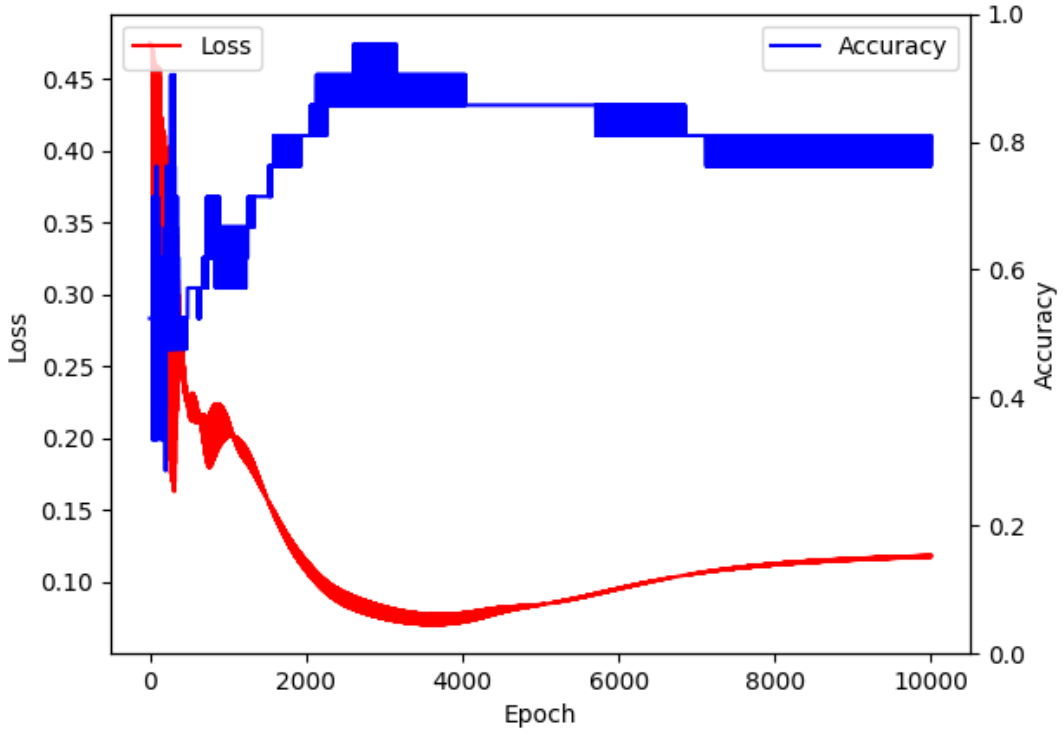


4. Discussion
    a. Learning rates: I choose 1e-2 as the default learning

rate, when I change the learning rate to 1e-4 (a small learning rate), it takes forever to converge to the optimal point. When I change to 1 (a large learning rate), the loss curve is really unstable, showing that it shoots out of the optimal point and do not converge.

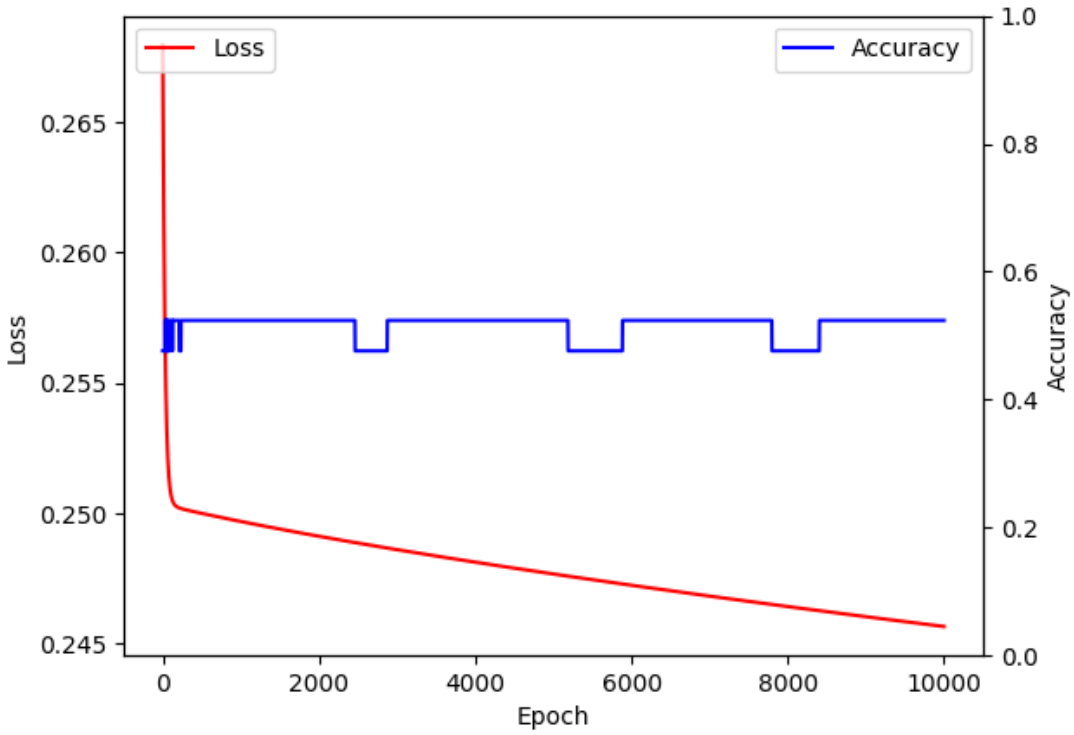


(lr = 1e-4)



(lr=1)

b. Hidden states: I tried to use 32x16 to train XOR dataset instead of 128x64, and the accuracy grows slowly, showing that its ability to represent this type of complicated data is not enough to converge.



c. Without activation

If without activation, there is nothing to regularize the values, which would lead to large output and even overflow, so the training procedure become really unstable (not plot since it overflows every time). Also since there are only linear layers, so it can't learn the XOR (not-linear) dataset.

5. Questions

a. What is the purpose of activation functions?

To regulate the output value into a range so that we can perform some specific tools on the output, e.g., Sigmoid can turn a value into [0, 1], and we can treat it as a probability and use Cross-Entropy loss.

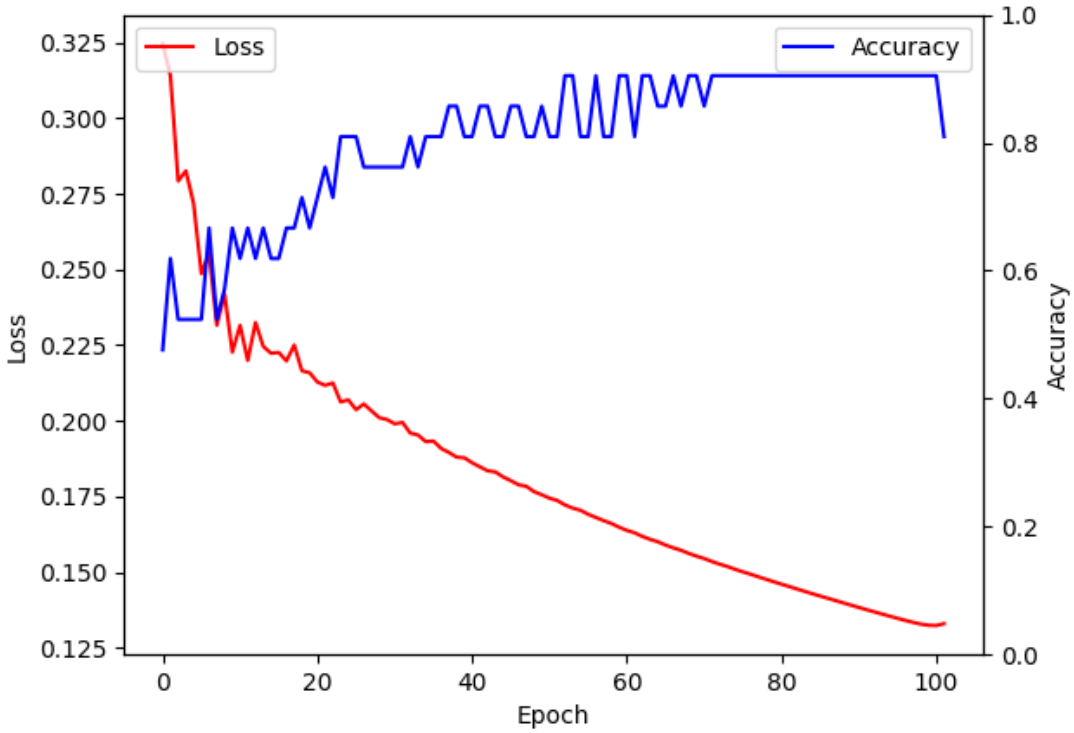
b. What might happen if the learning rate is too large or too small?

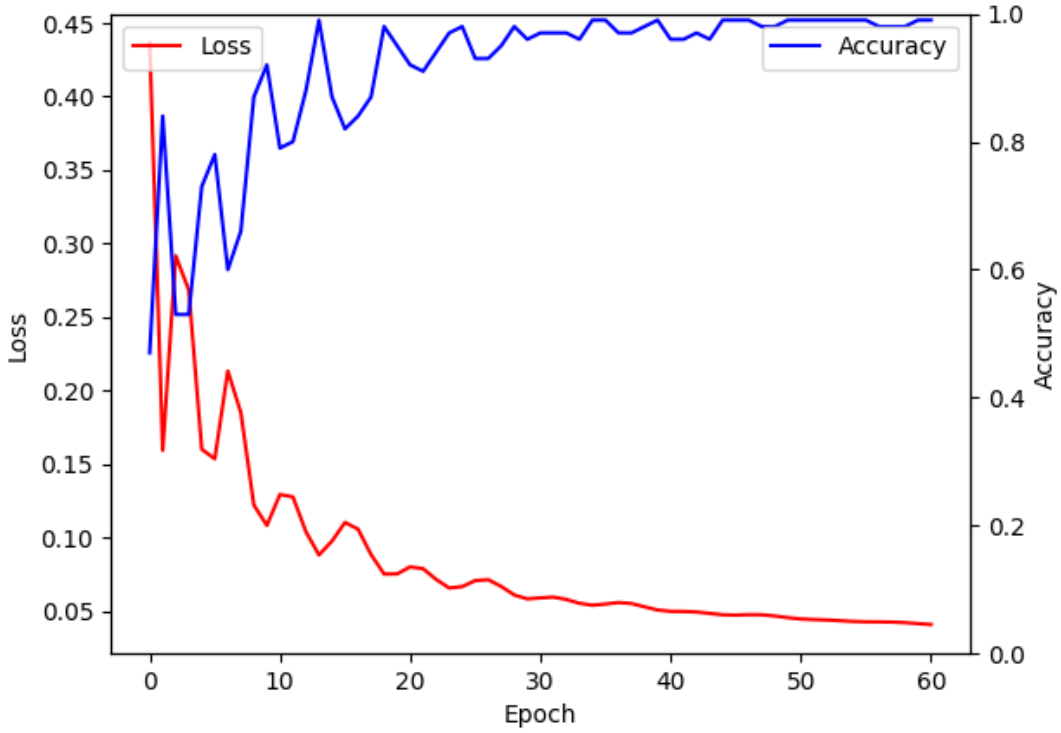
As I mentioned at section 4.a, when the learning rate too large, the weight would shoot out of the optimal point, and make the learning procedure

unstable. When the learning rate is too slow, it might take a lot of time to converge to the optimal point, or it can't escape the sub-optimal point.

c. What is the purpose of weights and biases in a neural network?
   The weights and biases are to perform linear transform on the input.

6. Extra
   a. Optimizers
      I implement another famous optimizer **Adam**, it outperforms SGD on both dataset with same parameter setup on converge speed, it takes about only 100 epochs to converge.

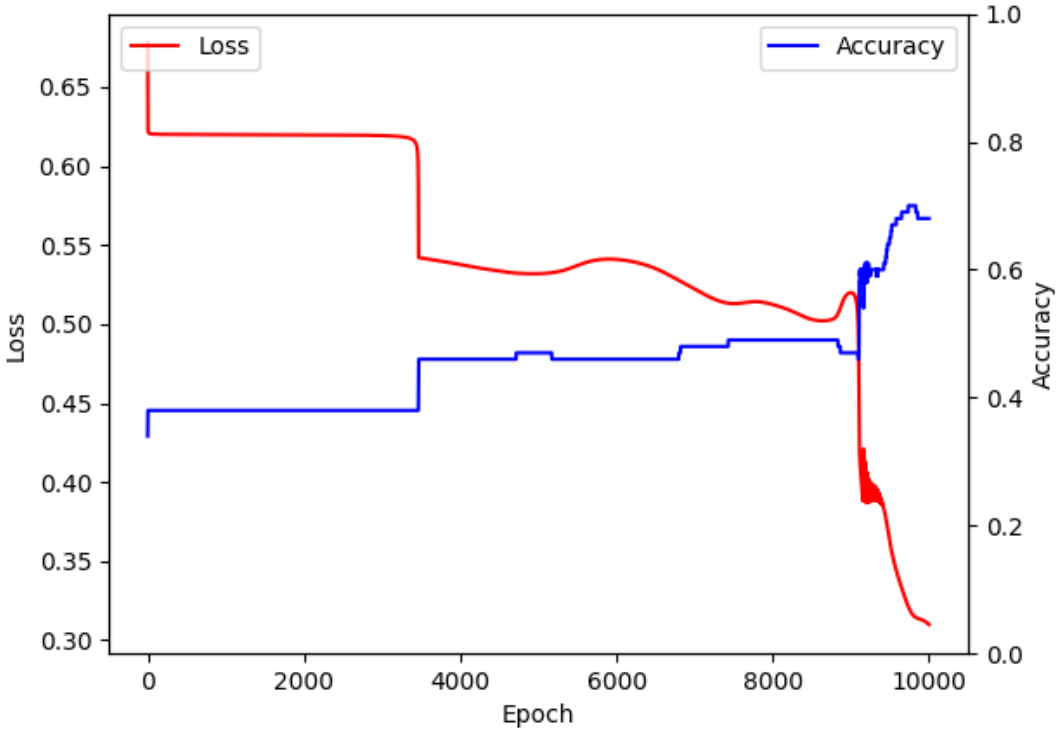




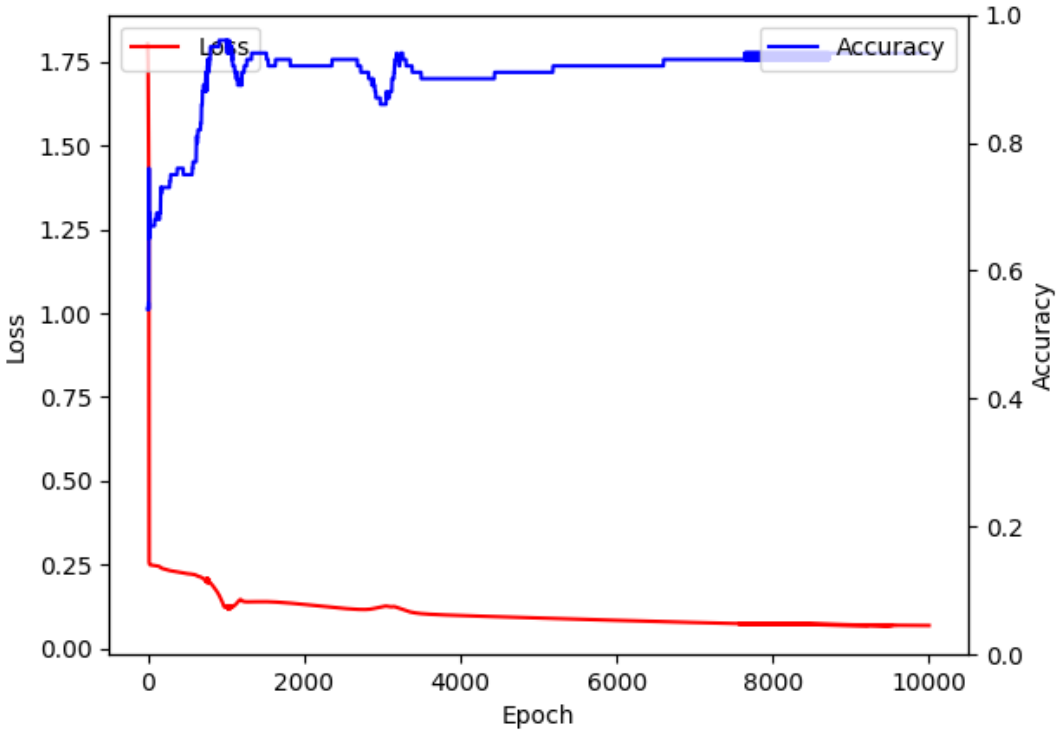
b. Activation functions

I Implemented SoftSign and Tanh activation to compare their performance on same parameter setting.

Tanh performs bad since I choose 0.5 as the baseline between labels, and the target label is 0 and 1, but Tanh converts the input into [-1, 1], which might be hard to generate a value consistently near 0 (target):



SoftSign has a curve like Tanh, but its slope is smoother around 0, so its performance when regressing to a label 0 is better than Tanh.



c. Convolution layer (not implemented)

7. Appendix: Implementations

a. Adam

```
class Adam:
    def __init__(self, model, lr=1e-2, beta1=0.9, beta2=0.999,
epsilon=1e-8):
        self.model = model
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = {}
        self.v = {}
        self.t = 0

        for layer in self.model.layers:
            if hasattr(layer, "weights"):
                self.m[layer] = np.zeros_like(layer.weights)
                self.v[layer] = np.zeros_like(layer.weights)

    def step(self):
        self.t += 1

        for layer in self.model.layers:
            if hasattr(layer, "weights"):
                grad = layer.grad_weights

                self.m[layer] = self.beta1 * self.m[layer] + (1 -
self.beta1) * grad

                self.v[layer] = self.beta2 * self.v[layer] + (
                    1 - self.beta2
                ) * np.square(grad)

                m_hat = self.m[layer] / (1 - self.beta1**self.t)

                v_hat = self.v[layer] / (1 - self.beta2**self.t)

                layer.weights -= self.lr * m_hat / (np.sqrt(v_hat) +
self.epsilon)
```

b. Tanh

```python
class Tanh(Layer):
    def __init__(self):
        self.x = None

    def forward(self, x):
        self.x = x
        return np.tanh(x)

    def backward(self, grad_output):
        return grad_output * (1 - np.tanh(self.x) ** 2)
```

c. SoftSign

```python
class SoftSign(Layer):
    def __init__(self):
        self.x = None

    def forward(self, x):
        self.x = x
        return x / (1 + np.abs(x))

    def backward(self, grad_output):
        return grad_output / (1 + np.abs(self.x)) ** 2
```