DLP Lab1

1. Introduction

In this lab, we will delve into the implementation of simple neural networks while focusing on fundamental concepts such as the forward pass and backpropagation. Our primary goal is to construct a neural network with two hidden layers (three weights), utilizing only Numpy and Python's standard libraries.

2. Experiment setups

A. Sigmoid functions

Derivation

$$f(x) = \frac{1}{1+e^{-x}}$$

$$1-f(x) = \frac{e^{-x}}{1+e^{-x}}$$

$$f'(x) = \cancel{-1}\left(1+e^{-x}\right)^{-2} \times e^{-x} \times \cancel{-1}$$

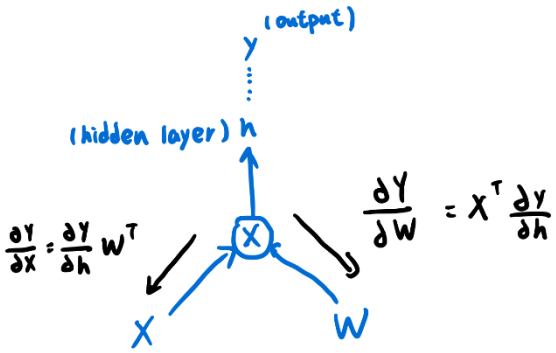
$$= f(x)(1-f(x))$$

Implementation

```
class Sigmoid():
    def __init__(self):
        self.output = None
    def forward(self, x):
        self.output = 1 / (1 + np.exp(-x))
        return self.output
    def backward(self, gradient):
        return gradient * np.multiply(self.output, (1-self.output))
```

B. Neural network

Derivation



Implementation

```
class Linear():
    def __init__(self, input_size, output_size):
        self.W = np.random.randn(input_size, output_size)
        self.input = None
    def forward(self, x):
        self.input = x
        return np.dot(x, self.W)

    def backward(self, gradient, lr):
        gradient_out = np.dot(gradient, self.W.T)
        self.W -= lr * np.dot(self.input.T, gradient)
        return gradient_out
```

C. Backpropagation

Pass the gradient backwards.

The first gradient was derived from the loss function.

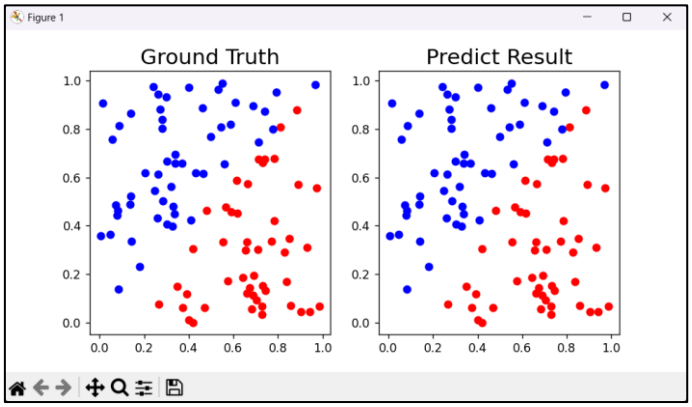
Implementation

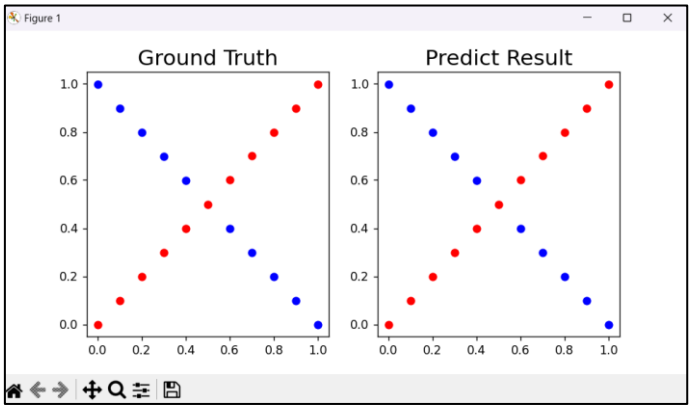
```
def backward(self, y):
    gradient = 2 * (self.predictions - y)
    gradient = self.activation3.backward(gradient)
    lr = self.optimizer3.getlr(gradient)
    gradient = self.L3.backward(gradient, lr)
    gradient = self.activation2.backward(gradient)
    lr = self.optimizer2.getlr(gradient)
    gradient = self.L2.backward(gradient, lr)
    gradient = self.activation1.backward(gradient)
    lr = self.optimizer1.getlr(gradient)
    gradient = self.L1.backward(gradient, lr)
```

3. Results of your testing

A. Screenshot and comparison figure

Config: lr = 0.1, optimizer = SGD, hidden unit = 3, epoch = 30000

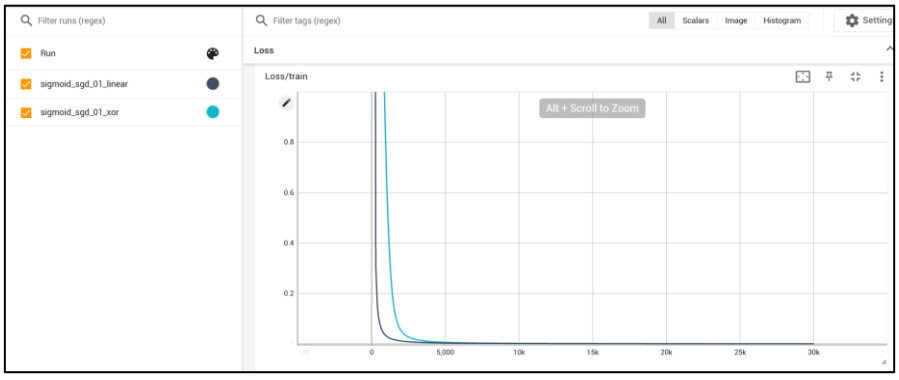




B. Show the accuracy of your prediction

```
Iter92 |    Ground truth: [1] |    prediction: 0.9999996354946171
Iter93 |    Ground truth: [1] |    prediction: 0.9999673233760434
Iter94 |    Ground truth: [0] |    prediction: 2.9822574013483736e-06
Iter95 |    Ground truth: [1] |    prediction: 0.9999997172719413
Iter96 |    Ground truth: [0] |    prediction: 2.0371862579389577e-05
Iter97 |    Ground truth: [1] |    prediction: 0.9999996601554417
Iter98 |    Ground truth: [0] |    prediction: 2.6347350329018343e-06
Iter99 |    Ground truth: [1] |    prediction: 0.9999997028006258
Loss=[0.00081561] accuracy=1.0
Iter12 |    Ground truth: [1] |    prediction: 0.9887897107282196
Iter13 |    Ground truth: [0] |    prediction: 0.0012571320188278924
Iter14 |    Ground truth: [1] |    prediction: 0.9999107688220383
Iter15 |    Ground truth: [0] |    prediction: 0.0006436239231531381
Iter16 |    Ground truth: [1] |    prediction: 0.9999420839027361
Iter17 |    Ground truth: [0] |    prediction: 0.0003640453760244666
Iter18 |    Ground truth: [1] |    prediction: 0.999942811312714
Iter19 |    Ground truth: [0] |    prediction: 0.0002293483226061319
Iter20 |    Ground truth: [1] |    prediction: 0.9999406717004354
Loss=[0.00049774] accuracy=1.0
```

C. Learning curve (loss, epoch curve)



Linear

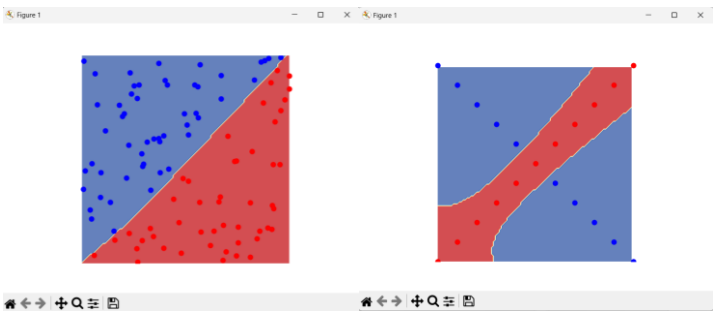
```
epoch 0 loss : 26.447101626359572
epoch 5000 loss : 0.008049239671506647
epoch 10000 loss : 0.003234553575851024
epoch 15000 loss : 0.00192894716728296
epoch 20000 loss : 0.0013450778473084056
epoch 25000 loss : 0.0010202553881928432
```

XOR

```
epoch 0 loss : 6.345911385963441
epoch 5000 loss : 0.006305692253848721
epoch 10000 loss : 0.002115832297511607
epoch 15000 loss : 0.0012079531574402207
epoch 20000 loss : 0.0008287986679365939
epoch 25000 loss : 0.0006243714999382504
```

D. Anything you want to present

Decision boundaries

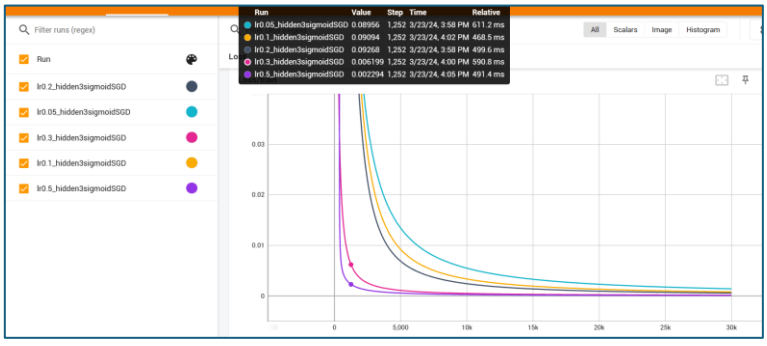


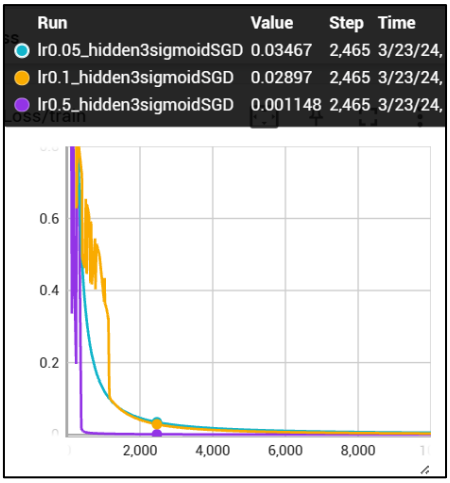
4. Discussion

A. Try different learning rates

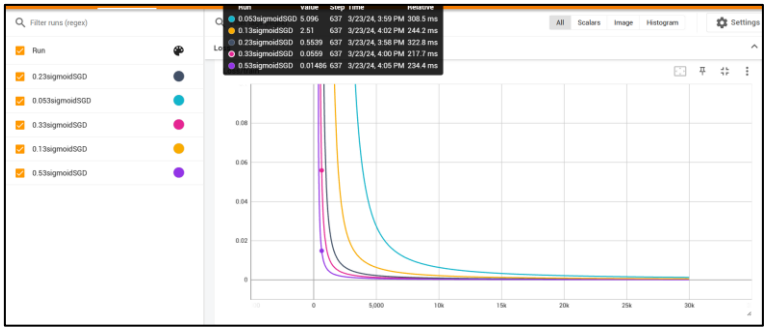
Due to the simplicity of dataset, the loss still converges in a good value even when learning rate is big. The graph shows that the higher learning rate is, the faster it converges, and the harder the training curve vibrates.

Linear





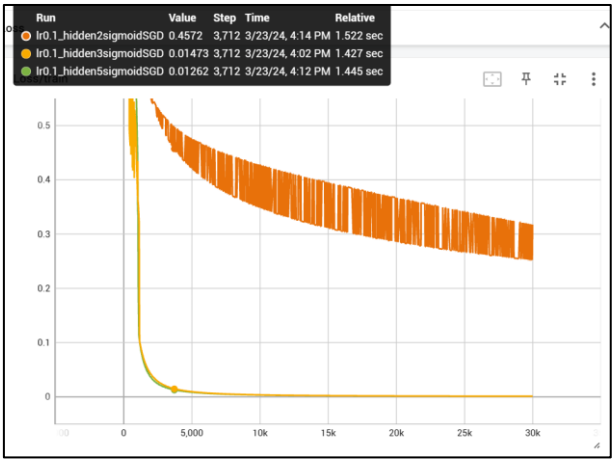
XOR



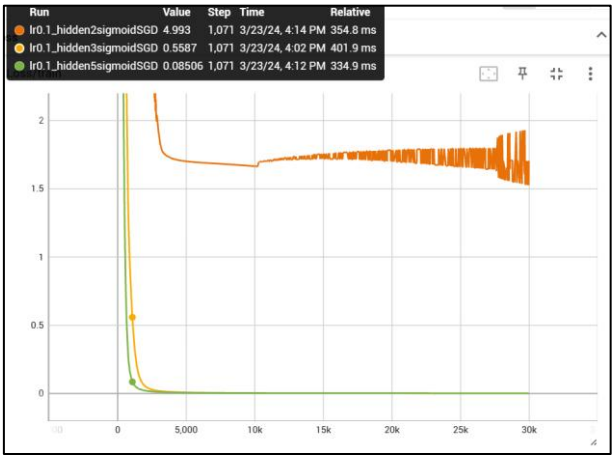
B. Try different numbers of hidden units

The result shows that it is harder for the nn having only 2 hidden units per layer to converge. The curve with 5 hidden units is more stable than the one with 3 hidden units.

Linear

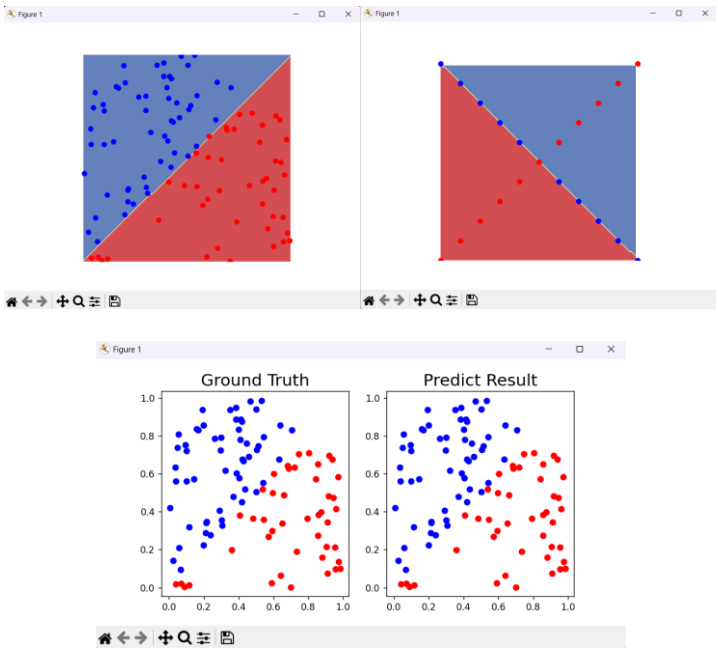


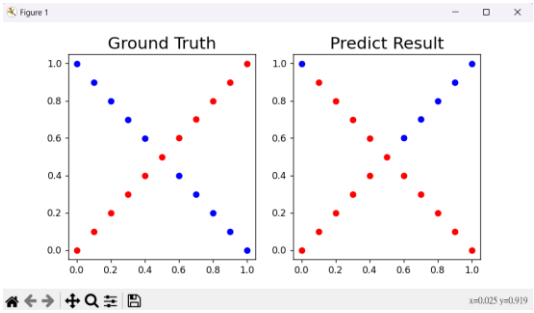
XOR



C. Try without activation functions

I took away all 3 sigmoid layers and use the original setting except for the optimizer. I tried using SGD and tune the learning rate from 0.1 to 0.01 and they all failed to converge, so I used Adagrad as the optimizer in this experiment. Since there is no sigmoid function in this experiment, the gradient will make the backpropagation process overflow, so I set 1 if output > 0.5 and otherwise 0 on the prediction when calculating loss. This nn fits the first data well because the data can be linearly classified. However, it is hard to fit the second data well because the data is hard to be linearly classified.





D. Anything you want to share

Normally we will think that the more stabler the training curve is, the better hyperparameter it is, but some curve vibrates strongly, and it converges fast, so maybe it depends to the dataset.

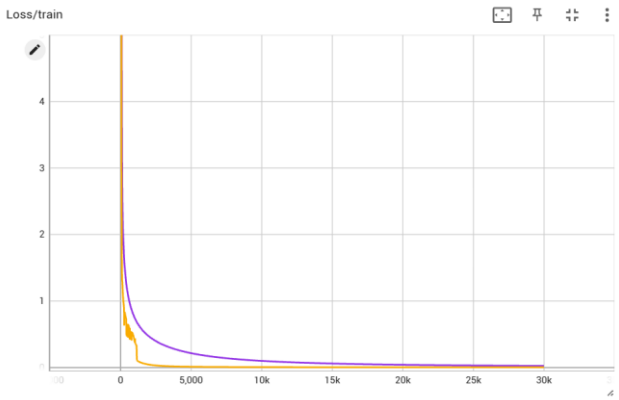
5. Extra

A. Implement different optimizers

I implemented adagrad to return updated learning rate.

```
class Adagrad():
    def __init__(self, lr=0.01):
        self.lr = lr
        self.G = None
    def getlr(self, gradient):
        if self.G is None:
            self.G = 1
        self.G += np.mean(gradient**2)
        return self.lr / np.sqrt(self.G + 1e-7)
```

Adagrad converges slower but much smoother than SGD.

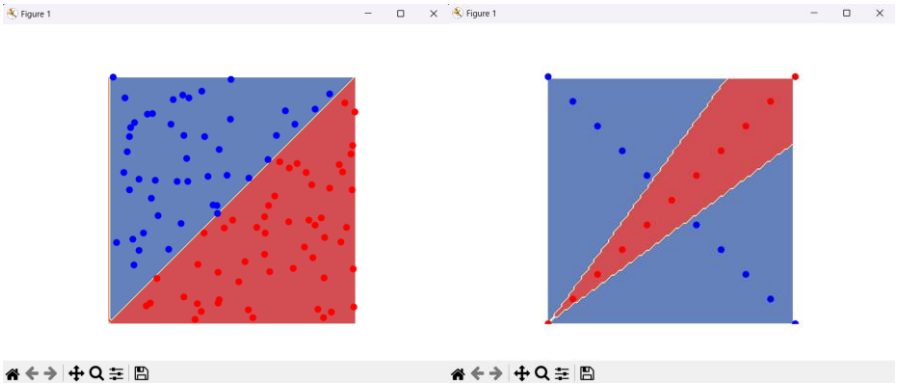


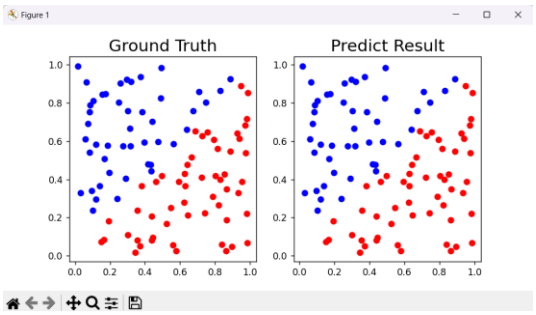
B. Implement different activation functions

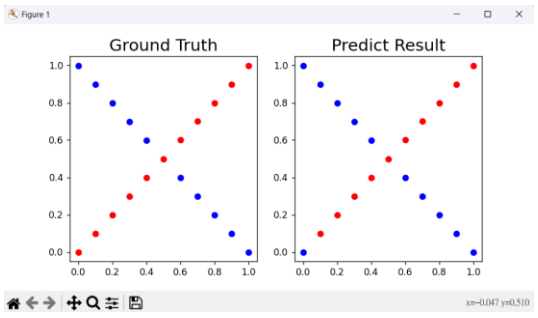
```
class ReLU():
    def __init__(self):
        self.output = None
    def forward(self, x):
        self.output = np.maximum(0, x)
        return self.output
    def backward(self, gradient):
        return gradient * np.where(self.output > 0, 1, 0)
```

I tried training with three relu but it failed to converge, so I modified the first

two layers into relu and use the same setting. Linear data fits perfectly and only one data point is wrong in the xor data. This is normal because data point(0,0) passing through the final sigmoid is going to have value 0.5. This error can be solved by adding bias values in the network or simply set value 0.5 to false.







C. Implement convolutional layers

X