NYCU DL Lab1 - Backpropagation

Student ID:313553014 Name:廖怡誠

1. Introduction

In the Lab1, we need to implement the neural network with two hidden layers. With implementing the basic neural network architecture, understanding how NN works such as forward pass, backward pass, and weight updating. In addition, we need to conduct experiments to observe the results when using different settings and hyperparameters, such as the activation function, optimizer, loss function, and learning rate to understand the effect on the model.

The figure below illustrates the structure of my code. All layer functions and activation functions are contained within the same file, following the Pytorch configuration. Additionally, I have implemented a class named 'Model' that manages the forward pass, backward pass, and weight updates accommodating different optimizers, learning rates, and the numbers of hidden units.

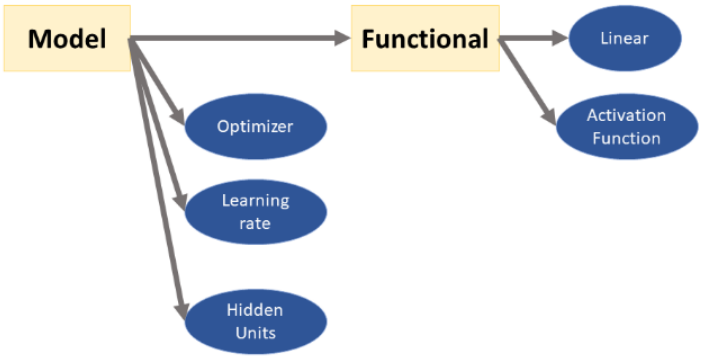


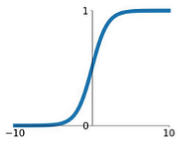
Fig1. The structure of my code

2. Experiment setups

A. Sigmoid function

Sigmoid

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




```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def derivative_sigmoid(x):
    return np.multiply(x, 1.0 - x)
```

The default activation function is the sigmoid function, which converts values to a range between 0 and 1. In the Lab1, the labels of the linear and XOR datasets are either zero or one, representing a binary classification problem, which means the characteristics of the sigmoid function are appropriately applied to this problem.

B. Neural network

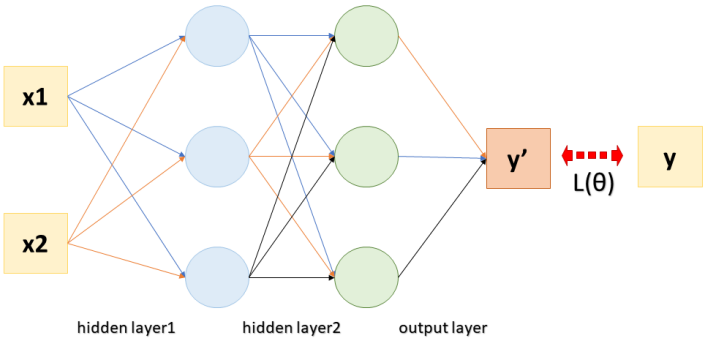


Fig2. Schematic diagram of neural network

My model architecture consists of two hidden layers and one output layer. The first hidden layer has an input feature size of 2 and an output feature size of 10. The second one has an input feature size of 10 and an output feature size of 10. The output layer has an input feature size of 10, and to obtain a single value the model output feature size is set to 1. Additionally, the model can produce different outputs by adjusting the hyperparameter such as learning rate and the activation function. For instance, if there is no activation function in the model, it will behave as linear model, yielding poor result when used with the XOR dataset.

C. Backpropagation

Fig3. Forward pass

Fig4. Backward pass

Backpropagation is a method that employs the chain rule to compute the gradient of the error function, $(L(\theta))$, with respect to the weight of neural network, (∂W). This calculated gradient is then used to update the weight of the layers for better performance. In my implementation, I do not include an activation in the output layer, hence, z3 does not involve the computation of the derivative of the activation function. The calculation of $\partial L(\theta)/\partial w_1$ is presented as an example in the following steps.

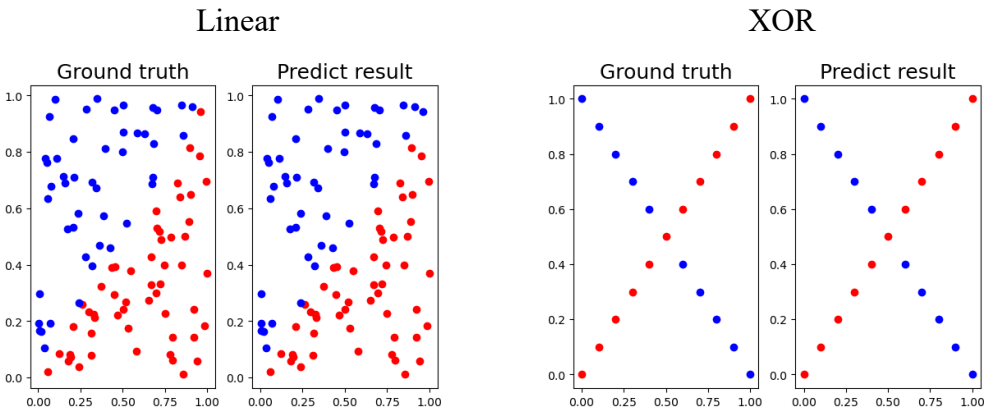
$$\frac{\partial L(\theta)}{\partial w_1} = \frac{\partial L(\theta)}{\partial z_3} * \frac{\partial z_3}{\partial a_3} * \frac{\partial a_3}{\partial z_2} * \frac{\partial z_2}{\partial a_2} * \frac{\partial a_2}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

3. Result of my testing

The hyperparameters are set such that the optimizer is Stochastic Gradient Descent (SGD), the learning rate (LR) is 0.1, the number of epochs is 100000, and the activation function is the sigmoid function.

A. Screenshot and comparison figure

The following two figures are comparison figure of ground truth and predict result of linear and XOR respectively. After training for 100000 epochs using corresponding datasets, the model can correctly classified.



B. Show the accuracy of your prediction

The two figures below demonstrate the accuracy of my model's predictions. After the training phase, the model achieved an accuracy of 99% with loss of 0.008 when tested on the linear dataset, and the accuracy of 100% with a loss of 0.00000881 when tested on the XOR dataset.

```
Iter 97 |  Ground Truth: 1.0 | Prediction: 1.00000 |
Iter 98 |  Ground Truth: 1.0 | Prediction: 1.00000 |
Iter 99 |  Ground Truth: 0.0 | Prediction: 0.00000 |
loss: 0.00893134 accuracy: 99.00%
```

Fig5. Linear- accuracy of your prediction

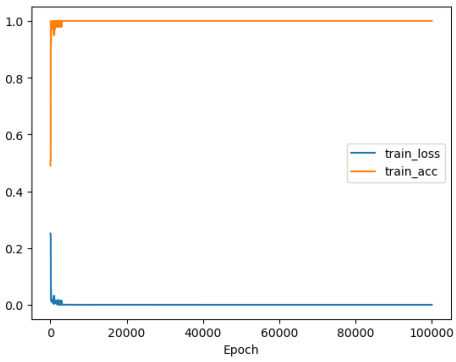
```
Iter 18 |  Ground Truth: 1.0 | Prediction: 0.99887 |
Iter 19 |  Ground Truth: 0.0 | Prediction: 0.00213 |
Iter 20 |  Ground Truth: 1.0 | Prediction: 0.99896 |
loss: 0.00000881 accuracy: 100.00%
```

Fig6. XOR- accuracy of your prediction

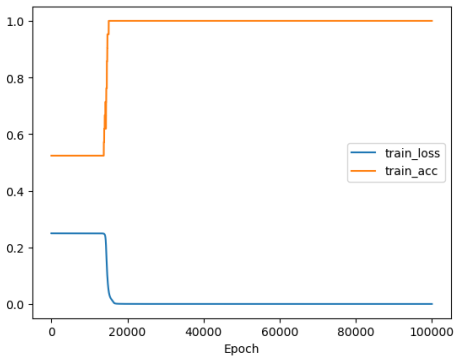
C. Learning curve (loss-epoch curve)

The two figures below illustrate the learning curve (loss versus epoch). During the training phase on the linear dataset, the model achieves the optimal performance around epoch 2000, and on the XOR dataset, the model reaches its peak performance around epoch 16000. Additionally, I observe an inverse relationship between accuracy and loss; in other words, as the loss decreases, the accuracy increases, vice versa.

Linear



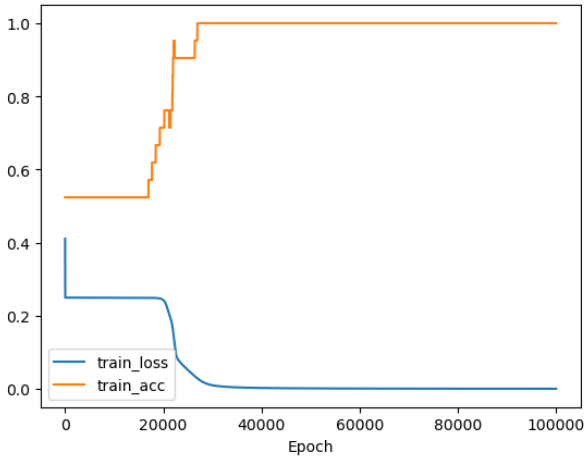
XOR



4. Discussion

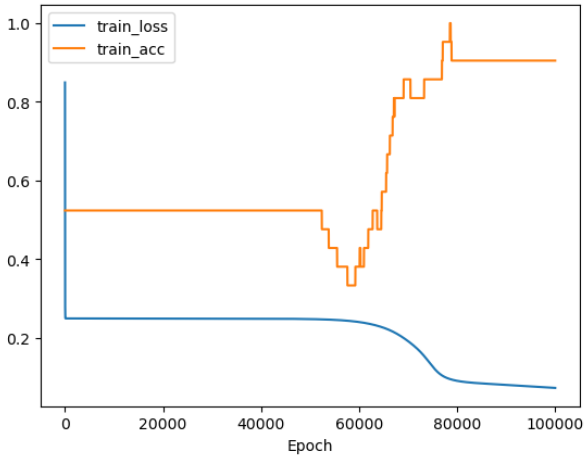
A. Try different learning rates

Learning rate is one hyperparameter that defines the adjustment in the weight of network with respect to the loss gradient descent. It determines how fast or slow model will move towards the optimal weight. Additionally, if the learning rate is too small, the model may get stuck in a local maximum. Conversely, if the learning rate is too large, the model may overshoot the global maximum. I conducted experiments with different learning rates of 0.01, 0.001, and 0.0001, and observed that as the learning rate decreases, the number of epochs required to achieve optimal accuracy increases. On the linear dataset, due to its simplicity, the model achieves optimal performance under all three conditions. However, on the XOR dataset, which is more complex, there is barely any increase in accuracy when the learning rate is set to 0.0001.



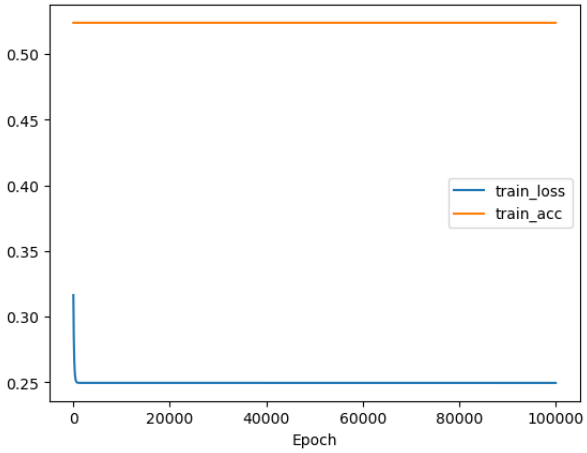

```
Iter 17 |  Ground Truth: 0.0 | Prediction: 0.00091 |
Iter 18 |  Ground Truth: 1.0 | Prediction: 1.00923 |
Iter 19 |  Ground Truth: 0.0 | Prediction: 0.00383 |
Iter 20 |  Ground Truth: 1.0 | Prediction: 1.01161 |
loss: 0.00017660 accuracy: 100.00%
```

Dataset is XOR, learning rate is set to 0.01



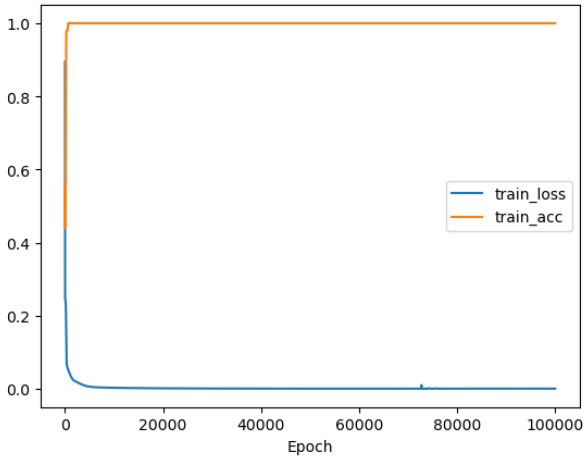

```
Iter 17 |  Ground Truth: 0.0 | Prediction: 0.04377 |
Iter 18 |  Ground Truth: 1.0 | Prediction: 1.08420 |
Iter 19 |  Ground Truth: 0.0 | Prediction: -0.05609 |
Iter 20 |  Ground Truth: 1.0 | Prediction: 1.21657 |
loss: 0.07287238 accuracy: 90.48%
```

Dataset is XOR, learning rate is set to 0.001



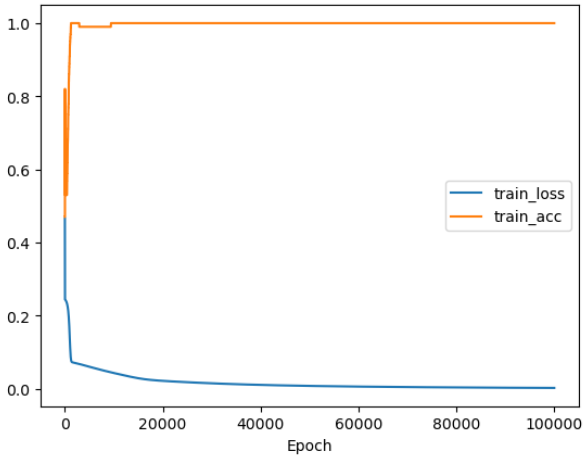

```
Iter 17 |  Ground Truth: 0.0 | Prediction: 0.47463 |
Iter 18 |  Ground Truth: 1.0 | Prediction: 0.47231 |
Iter 19 |  Ground Truth: 0.0 | Prediction: 0.47418 |
Iter 20 |  Ground Truth: 1.0 | Prediction: 0.47138 |
loss: 0.24938505 accuracy: 52.38%
```

Dataset is XOR, learning rate is set to 0.0001



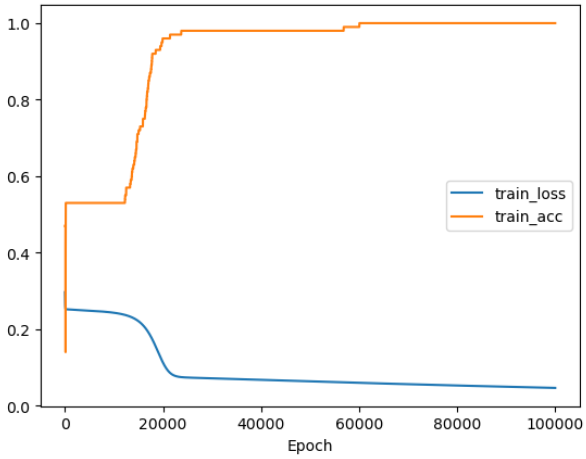

```
Iter 97 |  Ground Truth: 1.0 | Prediction: 0.96756 |
Iter 98 |  Ground Truth: 1.0 | Prediction: 0.96097 |
Iter 99 |  Ground Truth: 1.0 | Prediction: 0.97192 |
loss: 0.00532146 accuracy: 99.00%
```

Dataset is linear, learning rate is set to 0.01




```
Iter 97 |  Ground Truth: 0.0 | Prediction: 0.22367 |
Iter 98 |  Ground Truth: 1.0 | Prediction: 1.00305 |
Iter 99 |  Ground Truth: 1.0 | Prediction: 1.05068 |
loss: 0.02241624 accuracy: 96.00%
```

Dataset is linear, learning rate is set to 0.001



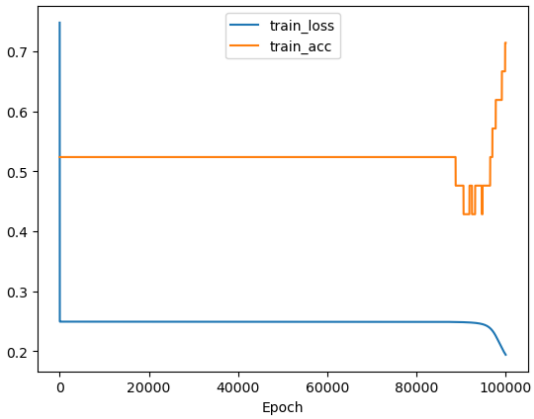

```
Iter 97 |  Ground Truth: 0.0 | Prediction: 0.35044 |
Iter 98 |  Ground Truth: 0.0 | Prediction: 0.24022 |
Iter 99 |  Ground Truth: 1.0 | Prediction: 1.00939 |
loss: 0.04235414 accuracy: 98.00%
```

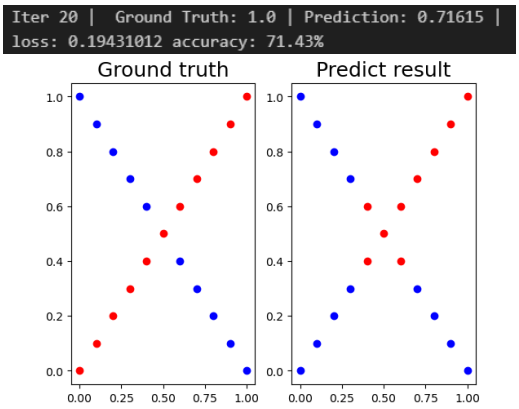
Dataset is linear, learning rate is set to 0.0001

B. Try different numbers of hidden units

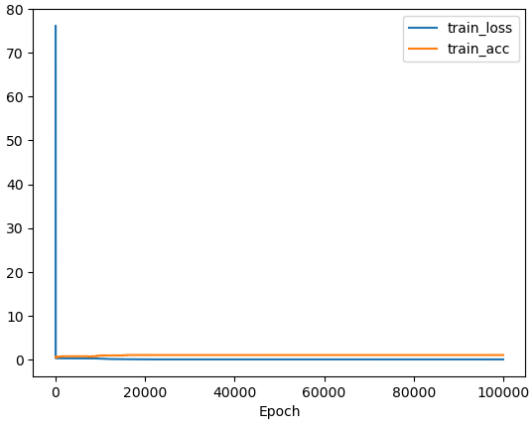
As the numbers of hidden units increases, the complexity of the model also increases. Additionally, because the number of model parameters increases, the computational load increases as well. I conducted experiments with

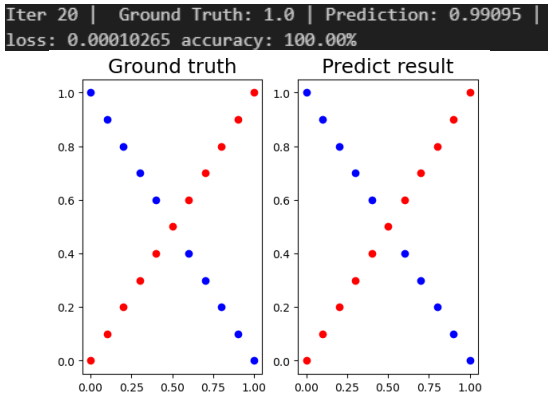
different numbers of hidden units and observed that when there are two hidden units, the training time is 10.9 seconds, and when there are one hundred hidden units, the training time is 25.7 seconds. However, the model reaches optimal performance at an earlier epoch when there are more hidden units. Moreover, due to a complex model with 100 hidden units, the model experiences a high loss value in the early epochs, until the gradient descends towards the correct direction.



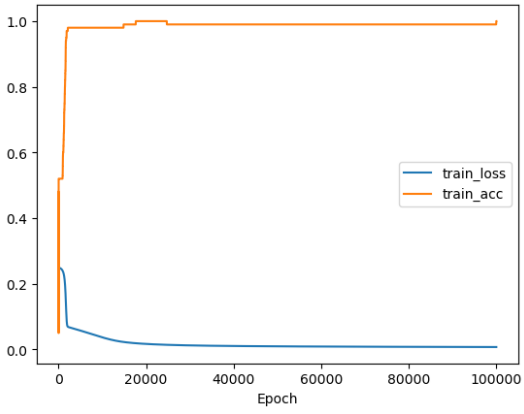


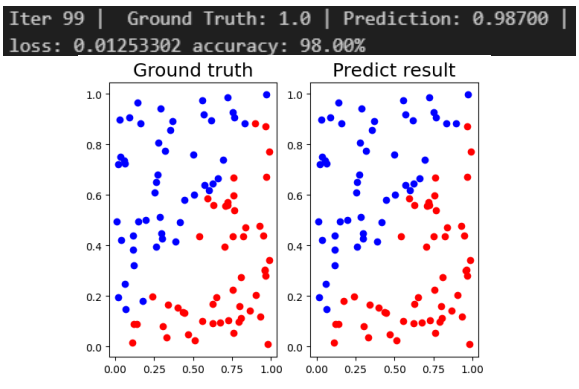
Dataset is XOR, the numbers of hidden units are set to 2.



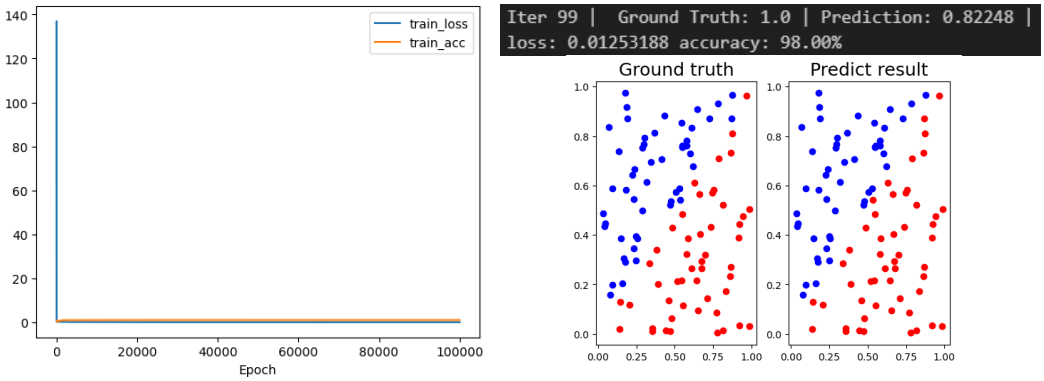


Dataset is XOR, the numbers of hidden units are set to 100.





Dataset is linear, the numbers of hidden units are set to 2.

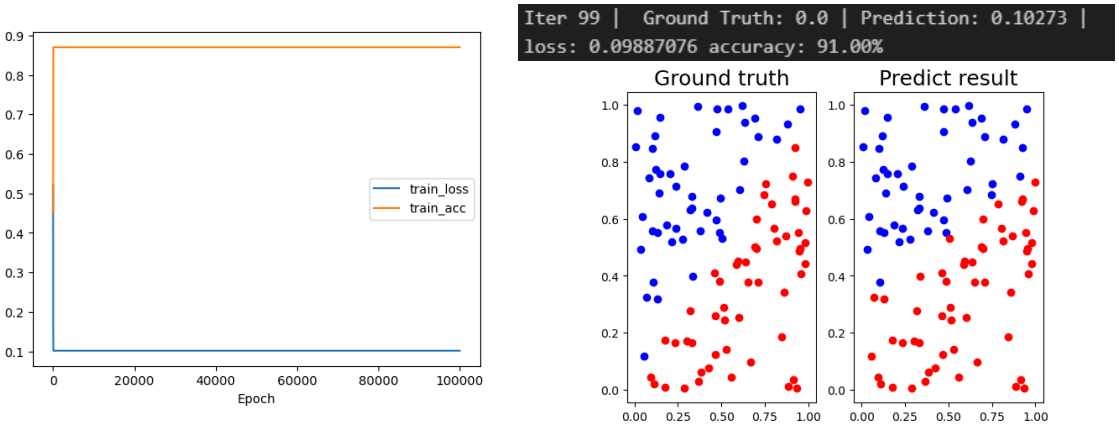


Dataset is linear, the numbers of hidden units are set to 100.

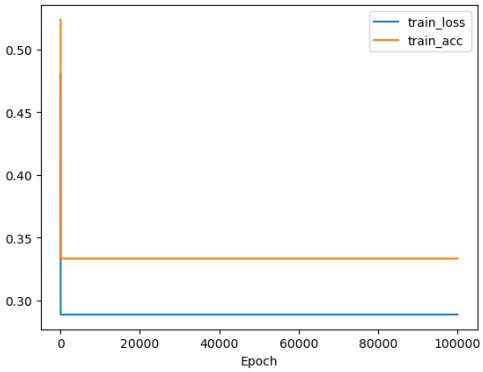
C. Try without activation functions

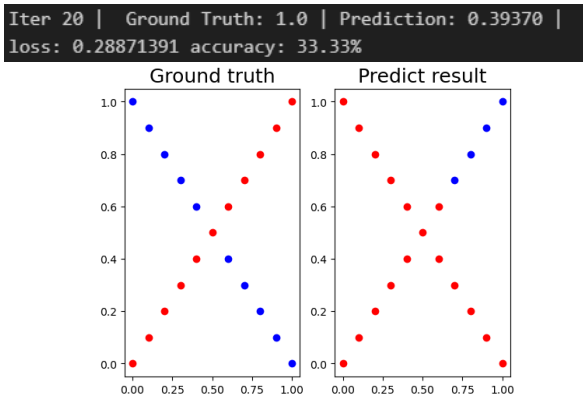
The hyperparameters in the following figure are set such that the optimizer is Stochastic Gradient Descent (SGD), the learning rate (LR) is 0.1, the number of hidden units is 10, the number of epochs is 100000, and there is no activation function.

The model without activation functions is essentially a linear model, which is incapable of solving complex problems. Consequently, while it can achieve an accuracy of 91% on the linear dataset, it fails to produce excellent results on the XOR dataset. This is because the XOR dataset requires a non-linear model capable of classifying data in higher dimension.



Dataset is linear





Dataset is XOR

5. Extra

A. Implement different optimizers

I have implemented two optimizers, Momentum and AdaGrad. Momentum, a variant of SGD, simulates the concept of physical momentum. It accelerates learning in consistent directions and slow down when the direction changes. AdaGrad is an optimizer that adjusts the learning rate according to the gradient. In my experiments, I observed that loss is fluctuates when the optimizer is SGD, whereas the loss is more stable when the optimizer is either Momentum or AdaGrad. Additionally, both Momentum and AdaGrad reach optimal accuracy faster than SGD.

$$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$$

$$W \leftarrow W + V_t$$

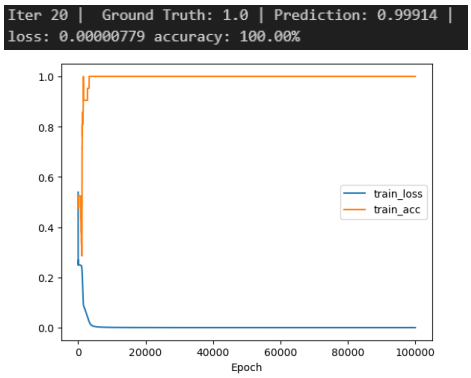
Momentum weight update equation

$$W \leftarrow W - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L}{\partial W}$$

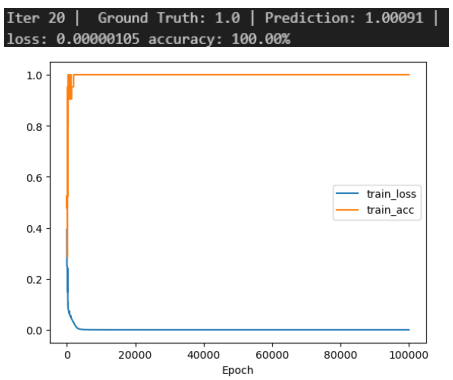
$$n = \sum_{r=1}^{t} \left(\frac{\partial L_r}{\partial W_r}\right)^2$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{\sum_{r=1}^{t} \left(\frac{\partial L_r}{\partial W_r}\right)^2 + \epsilon}} \frac{\partial L}{\partial W}$$

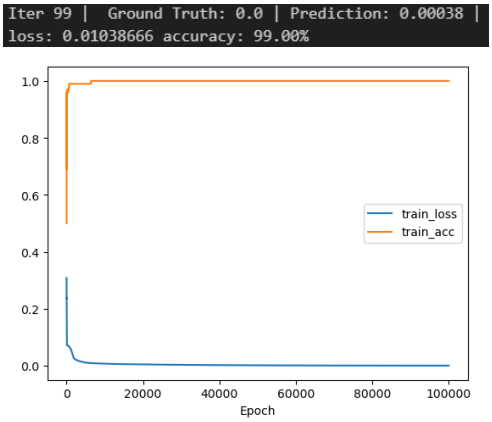
AdaGrad weight update equation



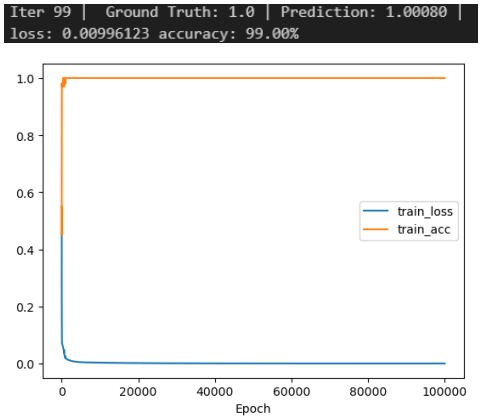
Momentum, XOR



AdaGrad, XOR



Momentum, linear



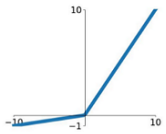
AdaGrad, linear

B. Implement different activation functions

In addition to implementing the sigmoid function, I also implemented LeakyReLU function and tanh function. LeakyReLU is variant of ReLU that helps avoid dead neurons. In my experiments, due to the learning rate being set to 0.1, which implies a slow gradient update, gradient descent does not perform well with LeakyReLU as the activation function. Tanh performs better than sigmoid, and my experimental result support this observation. In summary, LeakyReLU function yields poorer results, while tanh function outperforms it.

Leaky ReLU

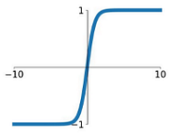
$\max(0.1x, x)$




```python
def LeakyReLU(x, a= 0.01):
    return np.where(x >= 0, x, a * x)

def derivative_LeakyReLU(x, a= 0.01):
    return np.where(x >= 0, 1, a)
```

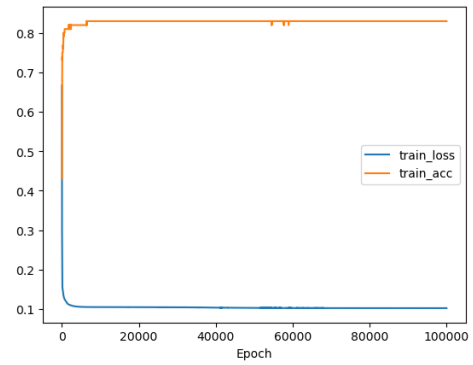
tanh

$\tanh(x)$



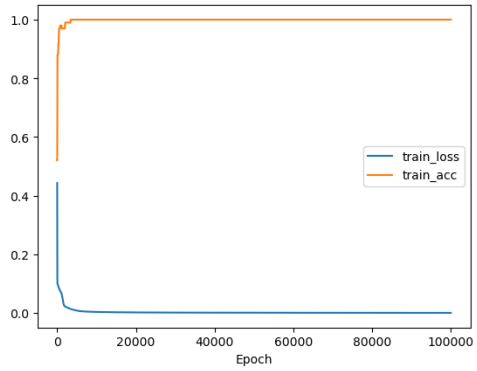

```python
def tanh(x):
    return np.tanh(x)

def derivative_tanh(x):
    return 1 - np.multiply(x, x)
```

```
Iter 99 |  Ground Truth: 1.0 | Prediction: 1.20731 |
loss: 0.06108963 accuracy: 93.00%
```



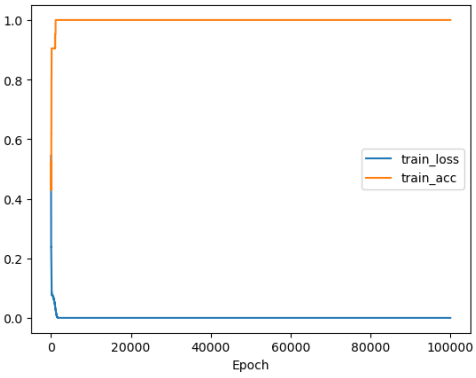
LeakyReLU, linear

```
Iter 99 |  Ground Truth: 0.0 | Prediction: -0.00606 |
loss: 0.01102611 accuracy: 98.00%
```



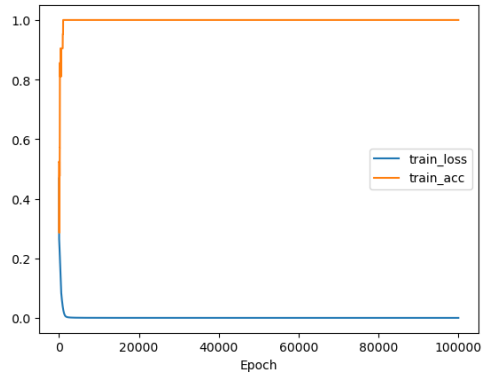
Tanh, linear

```
Iter 20 |  Ground Truth: 1.0 | Prediction: 1.00000 |
loss: 0.00000000 accuracy: 100.00%
```



LeakyReLU, XOR

```
Iter 20 |  Ground Truth: 1.0 | Prediction: 0.99934 |
loss: 0.00000968 accuracy: 100.00%
```



Tanh, XOR