Deep Learning – Lab 1

Backpropagation

I. Introduction

i. Requirements

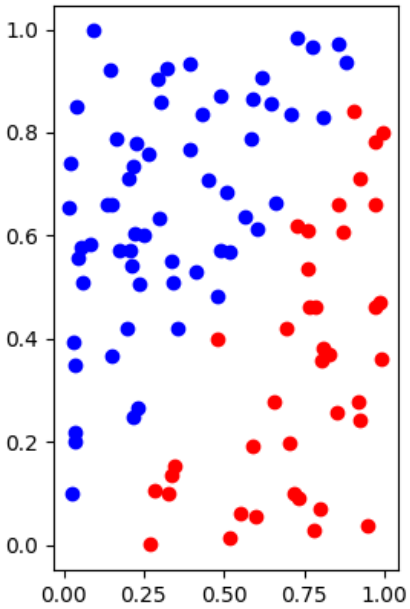
- Implement simple neural networks with two hidden layers.
- Each hidden layer needs to contain at least one transformation (CNN, Linear …) and one activate function (Sigmoid, tanh….).
- You must use backpropagation in this neural network and can only use Numpy and other python standard libraries to implement.

ii. Config

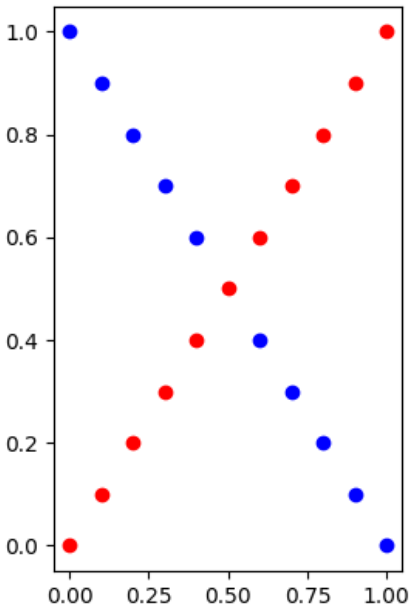
```
class Cfg:
    lr = 0.1
    epoch = 500
    seed = '312551116'
    activate = 'Sigmoid'
    schedular = 'ExponentialLR'
    loss_function = 'MSE'
    epislon = 1e-5
    optimizer = 'Adam'
```

iii. Data

Linear



XOR



II. Experiment setups

i. Sigmoid functions

- Sigmoid

```
lambda x : 1.0/(1.0 + np.exp(-x))
```

- Derivative Sigmoid

```
lambda x: np.multiply(x, 1.0 - x)
```

ii. Neural Network

- **Layer**

1. Layer.forward_once(): Multiply the input by the weights of this layer and add the bias.

```
def forward_once(self, input):
        self.input = np.append(input, np.ones((input.shape[0], 1)), axis = 1)
        self.output = np.matmul(self.input,self.weight)
        self.output = activate(self.output, self.activate)
        return self.output
```

2. Layer.backward_once(): Multiply the pre_gradient by the derivative of the activation function of this layer, and return the value multiplied by the weights to the previous layer.

```
def backward_once(self,pre_gradient):
        self.current_gradient =
np.multiply(pre_gradient,d_activate(self.output,self.activate))
        self.next_gradient = np.matmul(self.current_gradient, self.weight[:-1].T)
        return self.next_gradient
```

3. Layer.update_once(): Multiply the derivative of neuron with respect to the loss by the original input from the forward() function, and update the value of the weights according to the update method specified by the optimizer.

```
def update_once(self,lr, optimizer):
        self.weight_gradient = np.matmul(self.input.T,self.current_gradient)
        self.weight = self.weight - self.optimizer(lr = lr, optimizer = optimizer)
```

- **Network**

1. Network.forward(): Calculate the forward pass from the first layer to the last layer.

```
def forward(self,input):
        for layer in self.network:
            input = layer.forward_once(input)
        return input
```

2. Network.backward(): Calculate the derivatives of the loss with respect to the neurons, starting from the last layer and propagating backwards until the first layer.

```
def backward(self,gradient):
        for layer in reversed(self.network):
            gradient = layer.backward_once(gradient)
```

3. Network.update(): Update the parameters (weights and biases) of the neural network starting from the first layer to the last layer.

```
def update(self, optimizer = Cfg.optimizer):
        for layer in self.network:
            layer.update_once(self.lr, optimizer)
```

iii. Backpropagation

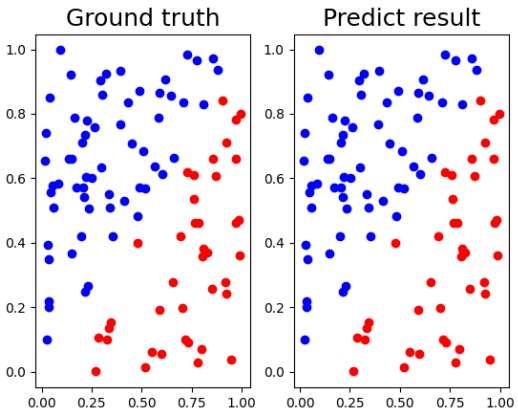
- In our neural network, we perform a forward pass on the input data to obtain the model's predictions. Then, we calculate the loss by comparing the predictions with the ground truth. The ultimate goal of the neural network is to minimize the loss value, for which we need to compute the partial derivatives of the loss function with respect to each parameter in the neural network, and use it to perform gradient descent, updating the parameters in the direction that minimizes the loss function.

- To calculate the partial derivatives of weights and biases in each layer, we will use backpropagation to apply the chain rule.

III. Results of your testing

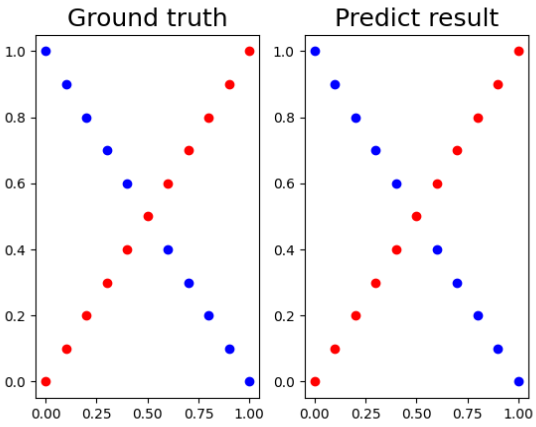
i. Linear data

```
Iter 79  | Ground truth: 1 | prediction: 0.999996537353852
Iter 80  | Ground truth: 1 | prediction: 0.999999990768754
Iter 81  | Ground truth: 0 | prediction: 0.000000006589720
Iter 82  | Ground truth: 0 | prediction: 0.000000388655859
Iter 83  | Ground truth: 1 | prediction: 0.999999993396778
Iter 84  | Ground truth: 1 | prediction: 0.999999993399717
Iter 85  | Ground truth: 1 | prediction: 0.999999990658352
Iter 86  | Ground truth: 0 | prediction: 0.007116593253508
Iter 87  | Ground truth: 0 | prediction: 0.000000028689866
Iter 88  | Ground truth: 1 | prediction: 0.999999993395696
Iter 89  | Ground truth: 1 | prediction: 0.999999993425728
Iter 90  | Ground truth: 1 | prediction: 0.999999980661953
Iter 91  | Ground truth: 0 | prediction: 0.000000006149779
Iter 92  | Ground truth: 1 | prediction: 0.999999993408114
Iter 93  | Ground truth: 0 | prediction: 0.000000006129757
Iter 94  | Ground truth: 1 | prediction: 0.991706060015984
Iter 95  | Ground truth: 0 | prediction: 0.000000877266757
Iter 96  | Ground truth: 0 | prediction: 0.000000010946393
Iter 97  | Ground truth: 0 | prediction: 0.000000006405709
Iter 98  | Ground truth: 1 | prediction: 0.998091826140120
Iter 99  | Ground truth: 1 | prediction: 0.999999993466254
loss = 0.000000937993307 accuracy = 1.0
```

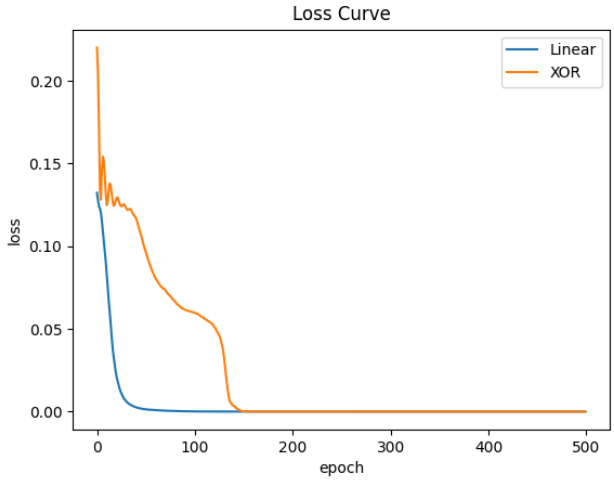


ii. XOR data

```
Iter  0  | Ground truth: 0 | prediction: 0.000773330360594
Iter  1  | Ground truth: 0 | prediction: 0.999978567424830
Iter  2  | Ground truth: 0 | prediction: 0.000784358175385
Iter  3  | Ground truth: 1 | prediction: 0.999993495044803
Iter  4  | Ground truth: 0 | prediction: 0.000957134132540
Iter  5  | Ground truth: 1 | prediction: 0.999996268867731
Iter  6  | Ground truth: 1 | prediction: 0.001897525638473
Iter  7  | Ground truth: 0 | prediction: 0.999996295023869
Iter  8  | Ground truth: 1 | prediction: 0.002359190685489
Iter  9  | Ground truth: 0 | prediction: 0.998546920221962
Iter 10  | Ground truth: 1 | prediction: 0.001890298251466
Iter 11  | Ground truth: 1 | prediction: 0.001509327236308
Iter 12  | Ground truth: 0 | prediction: 0.997340045666940
Iter 13  | Ground truth: 1 | prediction: 0.001296011628824
Iter 14  | Ground truth: 0 | prediction: 0.998383945633731
Iter 15  | Ground truth: 1 | prediction: 0.001174575688239
Iter 16  | Ground truth: 0 | prediction: 0.998389253494939
Iter 17  | Ground truth: 0 | prediction: 0.001100570944510
Iter 18  | Ground truth: 0 | prediction: 0.998389678717665
Iter 19  | Ground truth: 0 | prediction: 0.001052481068266
Iter 20  | Ground truth: 1 | prediction: 0.998389687451612
loss = 0.000001002499884 accuracy = 1.0
```

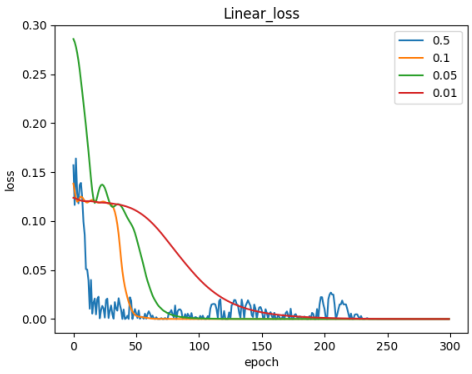


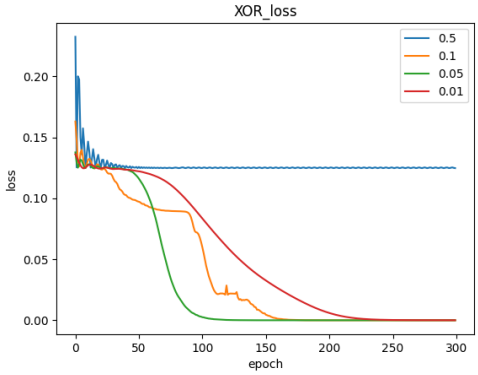
iii. Learning Curve



IV. Discussion

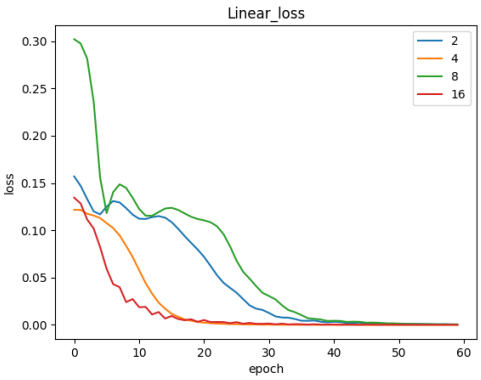
i. Different learning rate

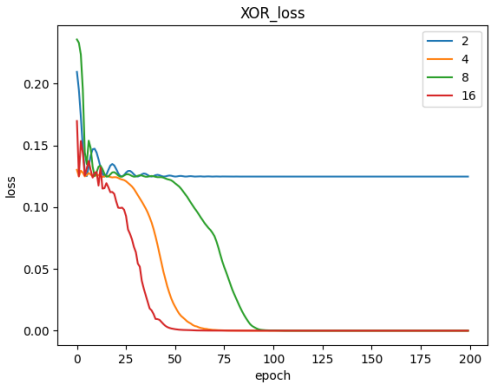




- We can observe that when using high learning rate, rapid convergence occurs in the initial phase; however, it might cause oscillations or zigzag learning patterns, which results in the failure to converge to the optimal solution.

ii. Different number of hidden units

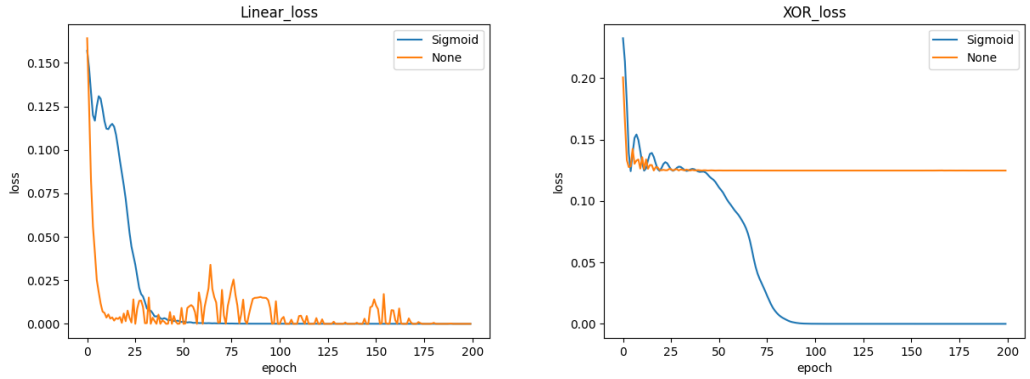




- Increasing the number of hidden units in the model can facilitate faster convergence and improved learning. However, this increase also leads to a higher number of parameters and computational complexity.

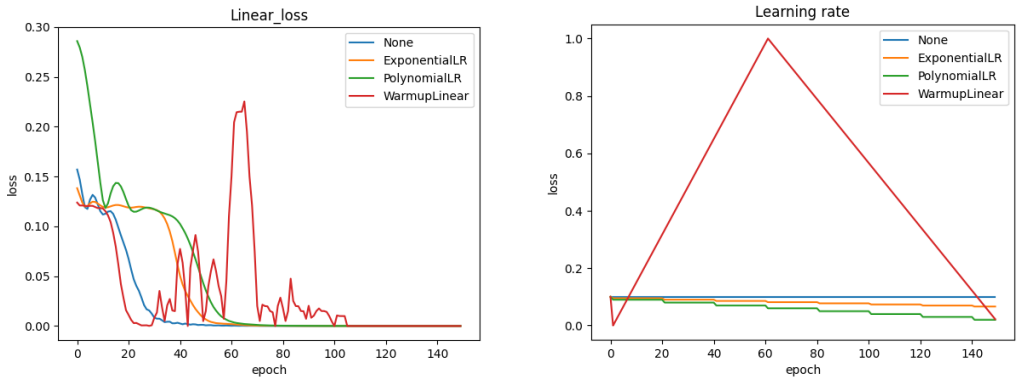
- Additionally, it is observed that using only 2 hidden units may not enable the model to perfectly learn XOR. This could be due to the fact that XOR is a more complex task compared to linear tasks, requiring a more complex model to capture the underlying patterns effectively.

iii. Without activation function



- In the absence of an activation function, we can observe that the model learns faster initially but faces challenges such as oscillatory convergence and high fluctuations in later stages. This can be attributed to the absence of a sigmoid function, which limits the output between 0 and 1 and helps stabilize the learning process by preventing large gradients.
- Furthermore, XOR proves to be difficult to converge without a sigmoid function. This is because XOR is a non-linear task, and without the approximation capabilities of a non-linear activation function, it becomes challenging to classify XOR accurately.

iv. With different scheduler



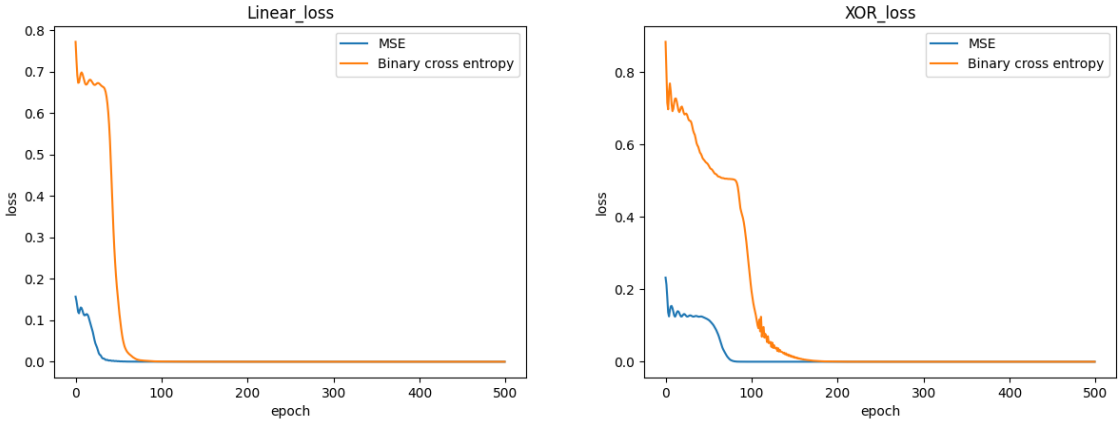
- This part is an additional test that I conducted. I implemented three types of schedulers: Exponential, Polynomial, and WarmupLinear. I tested their effects on a linear task (left figure) and plotted the learning rate curve (right figure).

- From the loss curve, we can see that without using any scheduler, the model converges the fastest. However, when using schedulers, the learning rate is gradually reduced, resulting in a more stable convergence in the later stages.

To sum up, schedulers can be beneficial in the early stages to avoid falling into local minima and achieving more stable convergence and better results in the later stages.

- BTW, WarmupLinear scheduler. From the graph, we can see that its convergence curve exhibits significant oscillations before reaching the highest point of warm-up steps. Afterwards, as the learning rate gradually decreases again, it converges gradually. Although it may not be apparent from the graph, in practice, it can achieve the lowest loss as a scheduler (though it is relatively less stable).

v. With different loss function

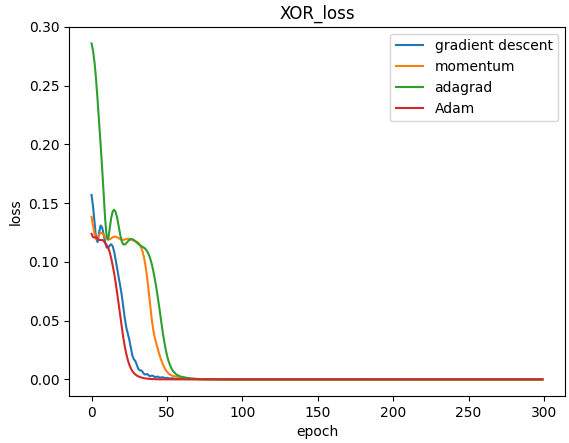


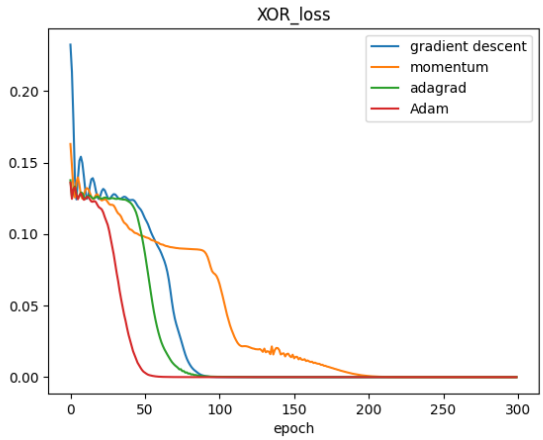
- In this test, I compared the training performance of different loss functions. I initially expected Binary Cross Entropy to converge faster than MSE, but the results turned out to be the opposite, which surprised me.

- In my opinion, the possible reason for this result is that MSE involves squaring the errors, making it more sensitive to larger errors. While this sensitivity allows it to learn faster in the initial stages, it can also lead to instability and difficulties in later stages of training. However, since the tasks in this experiment were relatively simple, both loss functions may have been able to converge easily. As a result, MSE appeared to perform better in this particular task.

V. Extra

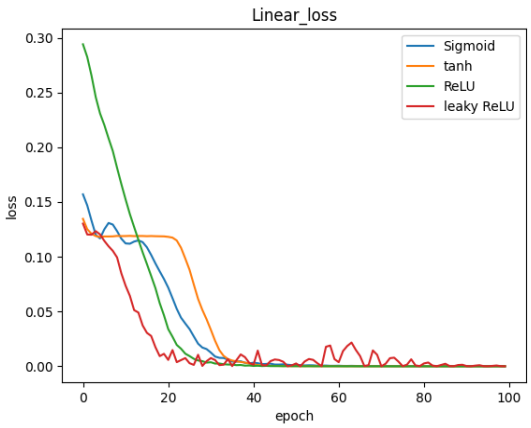
i. Implement different optimizers

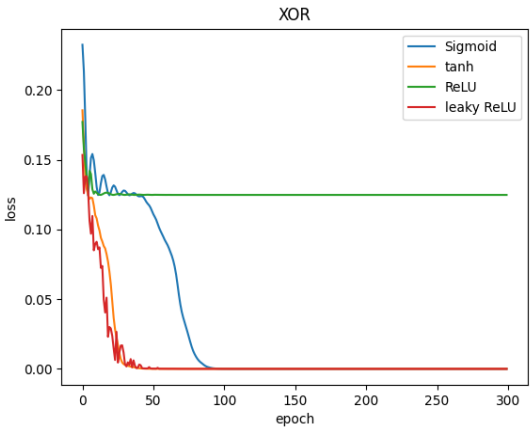




- In the Linear dataset, all four optimizers were able to converge quickly and easily due to the simplicity of the task.
- However, in the XOR dataset, we can clearly observe that Adam converges exceptionally fast, while momentum converges relatively slowly. I think this is because momentum accumulates momentum over time, and the XOR task has a clear decision boundary, making it easier to resulting in larger loss while classification errors near the boundary,.
- Overall, these observations suggest that different optimizers may perform differently depending on the dataset and task complexity. In the case of the XOR task, the clear decision boundary appears to have a significant impact on the convergence speed and performance of the optimizers.

ii. Implement different activation functions





- In the Linear task, all four activations demonstrate good learning performance. However, Leaky ReLU tends to exhibit a jagged learning curve in the later stages. I believe this might be due to its differential gradient values for positive and negative inputs, leading to learning instability.
- In the XOR problem, we can observe that ReLU performs poorly. I believe this is because ReLU has a gradient of zero in the negative region and its function resembles a linear function, which hinders its ability to handle the non-linear nature of the XOR problem effectively.