DL_HW1

0512204 Yu-Ting Yen

April 2020

1 Introduction

This assignment uses a simple neural network to approximate the labels of given points. We should implement the forward and backward part to complete the sample code. There are two kinds of training data provided by TA, Linear and Xor. The same network architecture can predict the correct labels depends on different training data and testing data.

2 Experimental Setup

The python version is 3.7.3. CPU is 1.4 GHz Intel Core i5. RAM is 8 GB. GPU is Intel Iris Plus Graphics 645 1536 MB.

2.1 Sigmoid functions

It's an activation function. The three layer(two hidden layer and output layer) all use this function. The formula is:

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

2.2 Neural network

The network has two hidden layers and one output layer. Input is 2×1, denoted by i, . The first hidden layer is 100×1, denoted by z_1. The second hidden layer is 100×1, denoted by z_2. The output layer is 1×1, denoted by y. The weights between layers are denoted by w_1,w_2,w_3. The initial of wight random by $N(0, 1)$. In the forward part, I compute the result y by the following formula:

$$z_1 = \sigma(i \cdot w_1)$$

$$z_2 = \sigma(z_1 \cdot w_2)$$

$$y = \sigma(z_2 \cdot w_3)$$

```python
def forward(self, inputs):
    """ Implementation of the forward pass.
    It should accepts the inputs and passing them through the network and return results.
    """
    # print("input", inputs.shape)
    self.input = inputs
    self.z1 = sigmoid(np.matmul(inputs, self.w1))
    self.z2 = sigmoid(np.matmul(self.z1, self.w2))
    self.y = sigmoid(np.matmul(self.z2, self.w3))

    return self.y
```

Figure 1: Code of forward part

2.3 Back-propagation

The weights can be updated by Back-propagation. We should estimate the error and compute the gradient from the back and propagate to the previous layer. Therefore, we should starts from the output layer first. The following formula is error estimation of three layers. δ^l represents the error of l layer. $\odot$ represents element-wise multiplication.

$$\delta^3 = y - label$$

$$\delta^2 = (\delta^3 \cdot w_3^T) \odot \sigma'(z_1 \cdot w_2)$$

$$\delta^1 = (\delta^2 \cdot w_2^T) \odot \sigma'(i \cdot w_1)$$

With the error, we can compute the gradient and update weights easily.

$$w_3 = w_3 - z_2^T \cdot \delta^3$$

$$w_2 = w_2 - z_1^T \cdot \delta^2$$

$$w_1 = w_1 - i^T \cdot \delta^1$$

```python
def backward(self):
    """ Implementation of the backward pass.
    It should utilize the saved loss to compute gradients and update the network all the way to the front.
    """
    # print("error", self.error)

    s3 = self.error
    s2 = np.matmul(s3, self.w3.T) * der_sigmoid(self.z2)
    s1 = np.matmul(s2, self.w2.T) * der_sigmoid(self.z1)
    grad_w3 = np.matmul(self.z2.T, s3)
    grad_w2 = np.matmul(self.z1.T, s2)
    grad_w1 = np.matmul(self.input.T, s1)

    self.w3 -= grad_w3
    self.w2 -= grad_w2
    self.w1 -= grad_w1
```

Figure 2: Code of backward part

3 Experimental Result

3.1 Screenshot and comparison figure

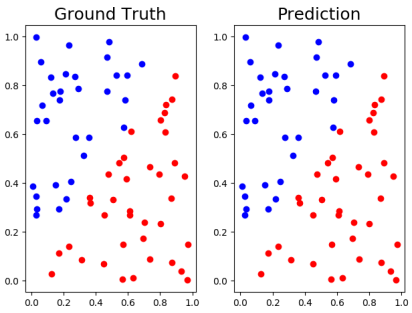

Figure 3: Results of data "Linear"

```
Epochs 0:
accuracy: 75.94%

Epochs 100:
accuracy: 100.00%

Epochs 200:
accuracy: 100.00%

Training finished
accuracy: 100.00%
```

Figure 4: Accuracy of data "Linear"

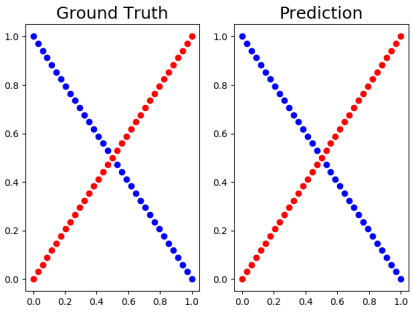

Figure 5: Results of data "XOR"

```
Epochs 0:
accuracy: 49.24%

Epochs 100:
accuracy: 99.93%

Epochs 200:
accuracy: 99.99%

Epochs 300:
accuracy: 99.99%

Epochs 400:
accuracy: 100.00%

Training finished
accuracy: 100.00%
```

Figure 6: Accuracy of data "XOR"

3.2 Anything you want to share

The number of neural is set to 100. I have tried different numbers of neurons. 20 or 50 can also predict well, but it needs more epochs. Therefore, I think 100 is a appropriate number.

4 Discussion

In the experiment, I find the "XOR" data is more challenge than "Linear". I observe this from the number of converge epochs. The "Linear" one often

converge at 200 epochs, and the "XOR" one often converge at 400 epochs. When the number of neurons decreases, this phenomenon is more apparent.