Backpropagation

0756079 陳冠聞

March 21, 2019

1 Introduction

The goal of lab 1 is to build a simple neural network and implement forward pass and back-propagation from scratch. I use sigmoid function as the activation function, and mean square error as cost function. For faster computation, I implement both forward pass and back-propagation in matrix operation.

Furthermore, I also implement another version of neural network which use ReLU as the activation function, and binary cross entropy as the cost function, then I compare the difference between two version in the discussion.

2 Experiment setup

2.1 Sigmoid functions

The output for each neuron in network is the weighted sum of its input, and then apply activation function. The sigmoid function is one of the activation functions, and the output of sigmoid function is always fall in the interval $[0, 1]$ as shown in the Figure 1.

Since calculation of the gradient is required in the back-propagation algorithm, it's necessary to know the derivative of the activation function. The derivative of sigmoid function is $\sigma'(z) = \sigma(z) * (1 - \sigma(z))$.

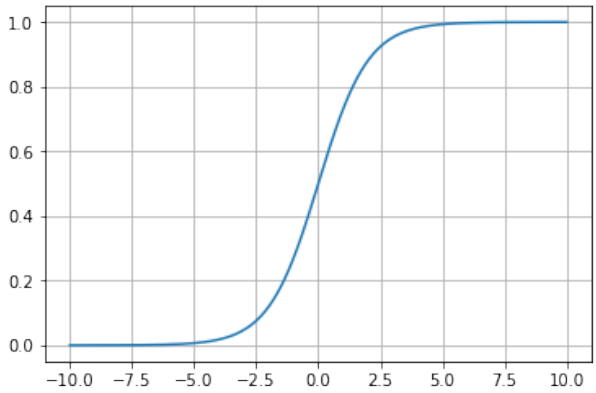


Figure 1: Sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$

2.2 Neural network

In this lab, the neural network takes two input and give one output. I use four neurons each hidden layer, and there are two hidden layers in total. The overall network architecture is shown in Figure 2. Initial weights are set to random numbers drawn from standard normal distribution, and I use sigmoid as activation function for all layers.

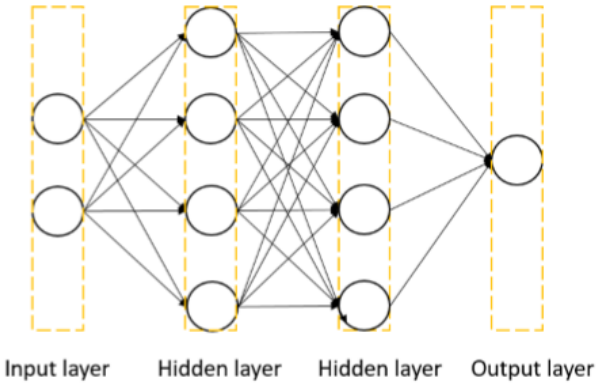


Figure 2: Neural Network Architecture

2.3 Backpropagation

The forward pass can be formulated as $a^l = \sigma(z^l) = \sigma(W^l a^{l-1})$, where a^l is the output of layer l, z^l is the weighted sum of input a^l. After forward pass, the neural network needs to adjust the weight by the gradient of cost function C with respect to the weight w^l, i.e. $\frac{\partial C}{\partial w^l}$. To compute the gradient, I first compute the delta δ^l for layer l, where $\delta^l = \frac{\partial C}{\partial z^l}$. Then by chain rule, I can simply compute the gradient $\frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial z^l}\frac{\partial z^l}{\partial w^l} = \delta^l a^{l-1}$. The implementation detail are in Figure 3

```python
class NeuralNetwork:
    def __init__(self, in_dims=2, hidden_dims=(10, 10), out_dims=1):
        if type(hidden_dims) == int:
            hidden_dims = [hidden_dims]
        self.num_layers = len(hidden_dims)
        self.W = [None]
        self.W.append(np.random.randn(in_dims, hidden_dims[0])) # W_1
        for l in range(1, len(hidden_dims)):
            self.W.append(np.random.randn(hidden_dims[l-1], hidden_dims[l]))
        self.W.append(np.random.randn(hidden_dims[-1], out_dims)) # W_L

        self.activation = sigmoid
        self.derivative_activation = derivative_sigmoid

        self.criterion = MSE_error
        self.derivative_criterion = derivative_MSE
```

(a) init

```python
def forward(self, x):
    self.a = [None for l in range(len(self.W))]
    self.z = [None for l in range(len(self.W))]
    self.a[0] = x
    for l in range(1, len(self.W)):
        self.z[l] = np.matmul(self.a[l-1], self.W[l])
        self.a[l] = self.activation(self.z[l])
    self.out = self.a[-1]
    return self.out
```

(b) forward-pass

```python
def backward(self, y, out, lr=0.1):
    deltas = [None for l in range(len(self.W))]

    # Compute delta for all layer
    deltas[-1] = self.derivative_criterion(y, out) * self.derivative_activation(self.z[-1])
    for l in range(len(self.W)-2, 0, -1):
        deltas[l] = np.matmul(deltas[l+1], self.W[l+1].T) * self.derivative_activation(self.z[l])

    # Compute gradient accroding to delta and update weight
    for l in range(1, len(self.W)):
        gradient = np.matmul(self.a[l-1].T, deltas[l])
        self.W[l] -= lr * gradient
```

(c) back-propagation

Figure 3: Code Snippet

3 Results of testing

When training the neural network, I use mean square error as loss function. Learning rate is set to 0.5, and number of epochs is set to 50,000 for both linear data and XOR data. I print out loss every 5,000 epochs and print the model output of first 10 example at the end of training.

As shown in Figure 4, the loss continues to decreases, which imply the network indeed learn something from data. In addition, we can see that every output is very close to 1.0 or 0.0, and the predict result is exactly the same as the ground truth (Figure 5), which means the network successfully learn the pattern of the data.

```
Epoch |    Loss
 5000 | 0.01192014
10000 | 0.00777360
15000 | 0.00587868
20000 | 0.00482130
25000 | 0.00412157
30000 | 0.00360897
35000 | 0.00321275
40000 | 0.00289528
45000 | 0.00262053
50000 | 0.00218792
```

(a) Linear loss

```
[[0.00008327]
 [0.99998367]
 [0.0000862 ]
 [0.0000543 ]
 [0.00032906]
 [0.99998361]
 [0.0000531 ]
 [0.99993091]
 [0.99998407]
 [0.00005137]]
```

(b) Linear output

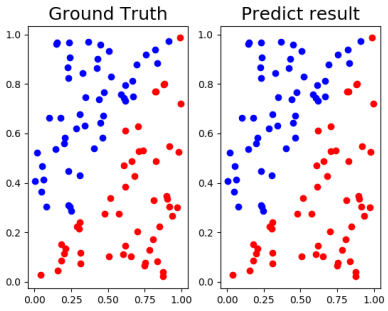
```
Epoch |    Loss
 5000 | 0.11415668
10000 | 0.00750947
15000 | 0.00112232
20000 | 0.00053804
25000 | 0.00034573
30000 | 0.00025252
35000 | 0.00019808
40000 | 0.00016258
45000 | 0.00013767
50000 | 0.00011926
```

(c) XOR loss

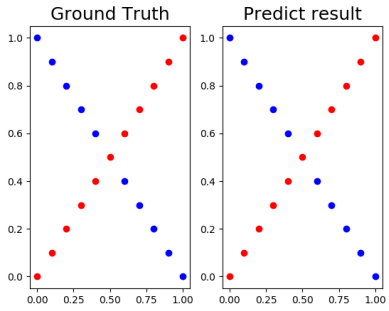
```
[[0.00238463]
 [0.99244   ]
 [0.0048268 ]
 [0.99238323]
 [0.00966902]
 [0.99216764]
 [0.01523903]
 [0.99114481]
 [0.01858549]
 [0.96665629]]
```

(d) XOR output

Figure 4: Loss and output



(a) Linear data



(b) XOR data

Figure 5: Comparison with Ground Truth

4 Discussion

4.1 Difficulties encountered in this lab

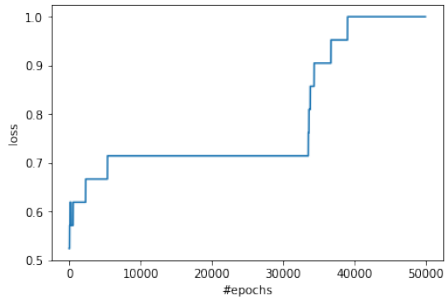
Because the element-wise operation is pretty slower comparing to the matrix operation, so I want to use matrix operation both in forward pass and back-propagation. Unfortunately, most of the formula derivation on the Internet for the back-propagation is about the element-wise operation. In this lab, I spent some time to fully understand the formula, and spent more time to derive the matrix form formula. In the end, training in matrix operation was much faster than training in element-wise operation, and I'm glad my effort was not in vain.

4.2 More exploration beyond the requirement

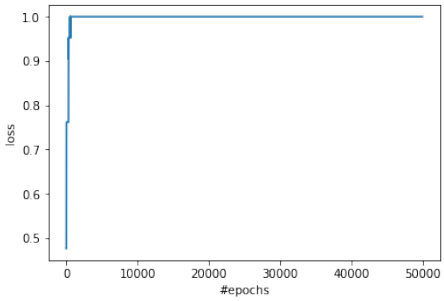
Since the sigmoid unit will saturate when the value is too positive or negative, it's not a popular choice for hidden unit in modern deep learning architecture. Instead, ReLU (Rectified Linear Unit) is the most popular one. Since the task is about binary classification, it's inevitable to use sigmoid unit in output layer. Nonetheless, instead of using mean square loss, using binary cross entropy as the loss function is able to cancel the exponential term in the sigmoid.

To explore the difference between sigmoid unit and ReLU, I design two networks and train on the XOR data. Both networks have two hidden layers, and each layer have 4 neurons, and the activation function in output layer is sigmoid in both networks. The difference is that one network uses sigmoid as activation function for hidden layer with MSE as cost function, another use ReLU as activation function with BCE as cost function.

I train the model for 50,000 epochs with learning rate set to 0.1, as shown in Figure 6, the model with ReLU and BCE can learn to perfectly classify quickly, and the loss drop to minimum value more smoothly (in Figure 7), which imply the model can learn faster and more stably.

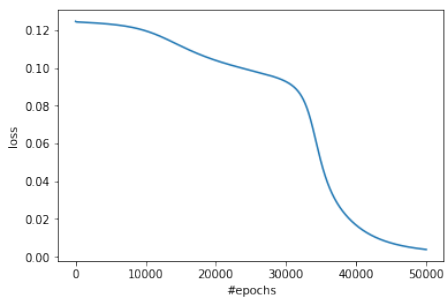


(a) Sigmoid + MSE

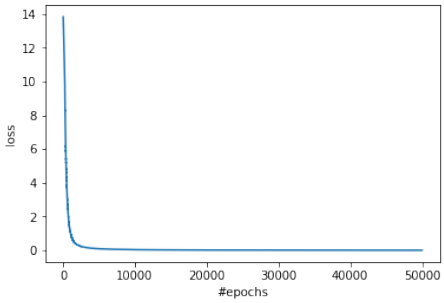


(b) ReLU + BCE

Figure 6: Accuracy History



(a) Sigmoid + MSE



(b) ReLU + BCE

Figure 7: Loss History