Homework 3

Neural Networks - Back-propagation pass

Xiaoyu Xiang(xiang43@purdue.edu)

Platform and Packages

- Python 3.6.5, PyTorch 0.4.0
- Numpy

Code Implementation

neural_network.py

In this script, we define a class NeuralNetwork, to use this class:

```
from neural_network import NeuralNetwork
```

It takes 1 input, that defines the size of input for each layer.

`NeuralNetwork.getlayer(layer_index: int)` can assign the weights for each layer. If not assigned, each layer will take random numbers(mean = 0, std = 1/sqrt(layer size)) as weights.

`NeuralNetwork.forward(input: FloatTensor)` can do the forward pass. For each layer, the output `y` has the following relationship with the input `x` :

`y = Wx + b`

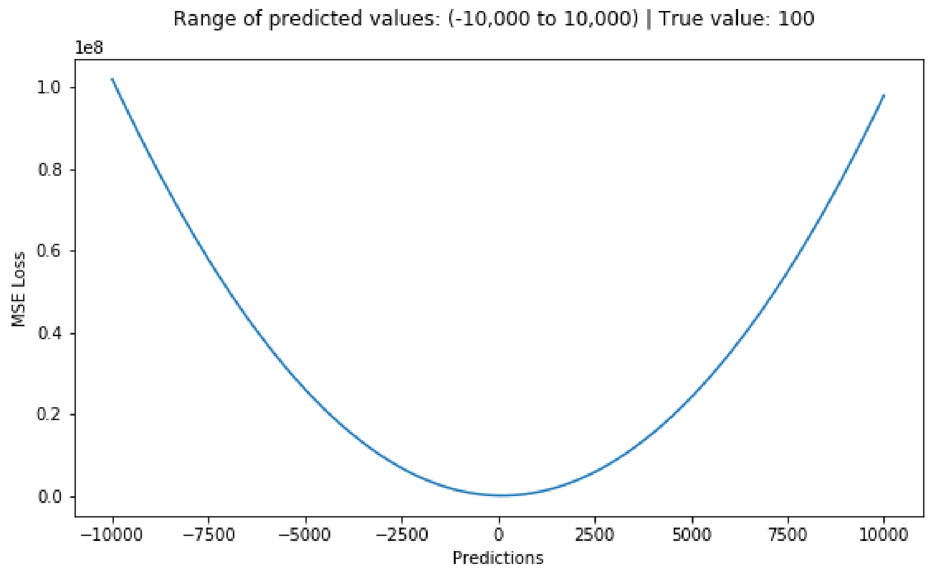
Where the b is the bias node.

`NeuralNetwork.backward(target: FloatTensor, loss: string)` can take 1D or 2D FloatTensor as input, which can compute the gradient of target matrix for back-propagation pass.

`NeuralNetwork.updateParams(eta: float)` is used for update parameters with the calculated backward values and given rate eta.

The loss function used in this update strategy is Mean Square Error Loss, or L2 Loss. Which can be expressed in the formula below.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - y_i^p)^2}{n}$$



Also, the loss function can be changed and even defined by ourselves. That's what we can set through the parameter loss.

If we set it `None` , it will apply the default loss MSE.

By setting it as `CE` , cross-entropy loss is applied:

$$CE = -\sum_x p(x) \log q(x)$$

Where p is the ground truth, an q is the network output(or, prediction result).

logic_gates.py

This script defines 4 classes: `AND` , `OR` , `NOT` , `XOR` .

All the classes take bool input and give bool outputs.

For `AND` , `OR` , `NOT` , they are generated by 1-layer neural network, `XOR` neural network has 2 layers. By transferring the input bool to int, we can apply the linear formula to calculate the output. Where the weights for each logic gates are updated by backward propagation strategy conducted by `logicGates.train()` .

The training process takes 4 input data and labels, which may get over for a few epochs.

Test Result

Run `'python test.py'`

It will build the logic gates, train them, print the training loss and wieghts, and print test results.

For each logic gate, the test results are listed in the tables below.

AND

Input 1	Input2	Output
True	True	True
True	False	False
False	True	False
False	False	False

OR

Input 1	Input2	Output
True	True	True
True	False	True
False	True	True
False	False	False

NOT

Input	Output
True	False
False	True

XOR

Input 1	Input2	Output
True	True	False
True	False	True
False	True	True
False	False	False

Although the results are same with HW02, the theta used in each gates are different:

The manually set weights are:

Gate	b	w1	w2
AND	-10	7	7
OR	-1	7	7
NOT	5	-7	\

XOR Gate:

Layer	b	w1	w2
0	[-5, -5]	[6, -6]	[-6, -6]
1	-5	7	7

The generated weights after training are:

Gate	b	w1	w2
AND	-2.0412	1.3731	1.0525
OR	-0.2708	1.9155	1.3391
NOT	1.2683	-2.7087	\

XOR Gate:

Layer	b	w1	w2
0	[5.1251, 1.8911]	[-3.6829, -6.3788]	[-3.6962, -6.4942]
1	-2.6235	6.1774	-7.2435

When training each gate with SGD, we can see that, the training loss decreases after epochs: