DL HW1

Ting-Hsuan, Wang 0756045

March 2019

1 Introduction

Deep learning is currently one of the hottest tools that almost every computer science or non computer science person would use. With the magic of deep learning and the progress of the computing power of GPUs, many problems that were intractable are now solvable. For example, by constructing CNN, we can classify photos into different categories. Or by constructing GAN, we can generate fake human faces which looks like as if they are from real human. However, to many people, deep learning is still considered a black box.
To fully understand the fundamentals of deep learning, we have to dive into the mathmatical part. Briefly speaking, deep learning task can be separated into 3 part: First, using feedforward network to calculate the outputs of each layers. Then, using backpropagation to update the weights. Lastly, iterating through the first and second steps until the weights/results converge. In this homework, we need to implement the above three steps from scratch without using any libraries that will help us calculate derivatives. We will also do experiments on a linear dataset and a XOR dataset to validate our network. Finally, we'll give a discussion on this homework.

2 Experiment setups

In this section, we're going to introduce the activation function and network architecture we'll use later. Furthermore, we'll go through the process of backpropagation and deduce formulas that can be used to update the parameter, weight and bias terms.

2.1 Sigmoid functions

In deep learning model, we want to use activation functions that has derivative everywhere. The function we choose here is sigmoid function, which gives us 1 when x is extremely large and 0 when x is extremely small. Sigmoid function looks like this:

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

By doing derivative on f(x) with respect to x, we can get:

$\frac{\partial f(x)}{\partial x} = (\frac{-1}{(1+e^{-x})^2})(-e^{-x}) = (\frac{1}{1+e^{-x}})(\frac{e^{-x}}{1+e^{-x}})$

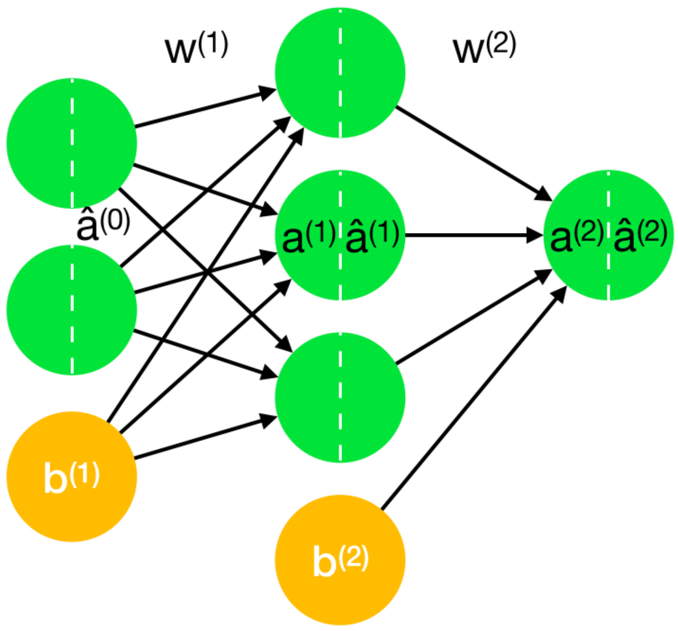
$= (\frac{1}{1+e^{-x}})(\frac{1+e^{-x}-1}{1+e^{-x}}) = f(x)(1-f(x))$

2.2 Neural network

Usually, if the problem we're dealing with is complicate, the network has to be broad and deep, or we won't have enough parameters to model the problem. Since the dataset that we're separating here (2d linear dataset and 2d XOR dataset) is easy, we do not need huge neural network.
We define our linear separation network architecture as follows: 2 neurons as input layer, two 2 neurons as hidden layers, and a neuron as output layer. We also add bias term on input and hidden layer. As for the XOR network: We set 2 neurons as input layer, two 3 neurons as hidden layers, and a neuron as output layer. The activation function we use is sigmoid, and the loss function is cross entropy. To further speed up the learning process, we use SGD momentum as optimizer with batch size equals to the training data, and we also add special initialization, "He initialization($\sqrt{\frac{2}{size^{(l-1)}}}$)", designed specifically for network that uses sigmoid as activation function. We also record the result of predictions and the loss during training phase in the "Results" section.

2.3 Backpropagation

In backpropagation step, we'll have to calculate the gradient for both weight and bias terms. The update degree is decided by the loss of the network's output and the real answer. The larger the loss is, the greater the update degree is. Here, we're going to calculate the gradient using some math. Before we start, let's define some mathematical notation.
Mathematical Notation:
For convenience, we define ith input vector and output vector as $a^{(i)}$ and $\hat{a}^{(i)}$. As for ith weight matrix and bias vector, we define them as $w^{(i)}$ and $b^{(i)}$.
Assume today we're trying to update a simple neural network:



This network has input layer of size 2, a hidden layer with size 3 and an output layer with size 1. Each layer except for the output layer has an bias term respectively, and each layer will pass through an activation function before it goes onto the next layer. The activation function we use here is sigmoid function. As for the loss function, we use cross entropy to calculate the loss, $L(\theta)$, for each iteration.
Steps:
Assume today we know the first input layer, $\hat{a}^{(0)}$. We first randomize all the parameters that the model is going to learn, including weight matrices, $w^{(1)}, w^{(2)}$, and bias terms,$b^{(1)}, b^{(2)}$. Then, we'll walk through forward pass to calculate the output value based on current weight matrices and bias term. Thirdly, we'll do back propagation by calculating the gradient of the weight and bias terms based on the result of forward pass. Lastly, we'll conclude a general formula to update all the parameters.
These is what the input layer and weight matrices and bias term looks like:

$$\hat{a}^{(0)} = \begin{bmatrix} \hat{a}^{(0)}_{11} & \hat{a}^{(0)}_{12} \end{bmatrix}$$

$$w^{(1)} = \begin{bmatrix} w^{(1)}_{11} & w^{(1)}_{12} & w^{(1)}_{13} \\ w^{(1)}_{21} & w^{(1)}_{22} & w^{(1)}_{23} \end{bmatrix}, b^{(1)} = \begin{bmatrix} b^{(1)}_{11} & b^{(1)}_{12} & b^{(1)}_{13} \end{bmatrix}$$

$$w^{(2)} = \begin{bmatrix} w^{(2)}_{11} \\ w^{(2)}_{21} \\ w^{(2)}_{31} \end{bmatrix}, b^{(2)} = \begin{bmatrix} b^{(2)}_{11} \end{bmatrix}$$

For the forward pass, we have to compute $\hat{a}^{(i)} = \sigma(a^{(i)} w^{(i+1)} + b^{(i+1)})$.

$$a^{(1)} = \begin{bmatrix} (\hat{a}^{(0)}_{11} w^{(1)}_{11} + \hat{a}^{(0)}_{12} w^{(1)}_{21}) + b^{(1)}_{11} & (\hat{a}^{(0)}_{11} w^{(1)}_{12} + \hat{a}^{(0)}_{12} w^{(1)}_{22}) + b^{(1)}_{12} & (\hat{a}^{(0)}_{11} w^{(1)}_{13} + \hat{a}^{(0)}_{12} w^{(1)}_{23}) + b^{(1)}_{13} \end{bmatrix}$$

$$= \begin{bmatrix} a^{(1)}_{11} & a^{(1)}_{12} & a^{(1)}_{13} \end{bmatrix}$$

$$\hat{a}^{(1)} = \begin{bmatrix} \sigma(a^{(1)}_{11}) & \sigma(a^{(1)}_{12}) & \sigma(a^{(1)}_{13}) \end{bmatrix} = \begin{bmatrix} \hat{a}^{(1)}_{11} & \hat{a}^{(1)}_{12} & \hat{a}^{(1)}_{13} \end{bmatrix}$$

$$a^{(2)} = \begin{bmatrix} (\hat{a}^{(1)}_{11} w^{(2)}_{11} + \hat{a}^{(1)}_{12} w^{(2)}_{21} + \hat{a}^{(1)}_{13} w^{(2)}_{31}) + b^{(2)}_{11} \end{bmatrix} = \begin{bmatrix} a^{(2)}_{11} \end{bmatrix}$$

$$\hat{a}^{(2)} = \begin{bmatrix} \sigma(a^{(2)}_{11}) \end{bmatrix}$$

For the backward pass, we have to calculate the gradient of each parameters. As for $L(\theta) = L(\hat{a}^{(2)})$, it is the loss function which is predefined before.

$$d\hat{a}^{(2)} = \begin{bmatrix} \frac{\partial L}{\partial \hat{a}^{(2)}_{11}} \end{bmatrix} = \begin{bmatrix} d\hat{a}^{(2)}_{11} \end{bmatrix}$$

$$da^{(2)} = \begin{bmatrix} \frac{\partial L}{\partial a^{(2)}_{11}} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial \hat{a}^{(2)}_{11}} \frac{\partial \hat{a}^{(2)}_{11}}{\partial a^{(2)}_{11}} \end{bmatrix} = \begin{bmatrix} d\hat{a}^{(2)}_{11} (\hat{a}^{(2)}_{11})(1 - \hat{a}^{(2)}_{11}) \end{bmatrix} = \begin{bmatrix} da^{(2)}_{11} \end{bmatrix}$$

$$dw^{(2)} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}^{(2)}} \\ \frac{\partial L}{\partial w_{21}^{(2)}} \\ \frac{\partial L}{\partial w_{31}^{(2)}} \end{bmatrix} = \begin{bmatrix} da_{11}^{(2)} \frac{\partial a_{11}^{(2)}}{\partial w_{11}^{(2)}} \\ da_{11}^{(2)} \frac{\partial a_{11}^{(2)}}{\partial w_{21}^{(2)}} \\ da_{11}^{(2)} \frac{\partial a_{11}^{(2)}}{\partial w_{31}^{(2)}} \end{bmatrix} = \begin{bmatrix} da_{11}^{(2)} \hat{a}_{11}^{(1)} \\ da_{11}^{(2)} \hat{a}_{12}^{(1)} \\ da_{11}^{(2)} \hat{a}_{13}^{(1)} \end{bmatrix} = \begin{bmatrix} dw_{11}^{(2)} \\ dw_{21}^{(2)} \\ dw_{31}^{(2)} \end{bmatrix}$$

$$db^{(2)} = \begin{bmatrix} \frac{\partial L}{\partial b_{11}^{(2)}} \end{bmatrix} = \begin{bmatrix} da_{11}^{(2)} \frac{\partial a_{11}^{(2)}}{\partial b_{11}^{(2)}} \end{bmatrix} = \begin{bmatrix} da_{11}^{(2)}(1) \end{bmatrix} = \begin{bmatrix} db_{11}^{(2)} \end{bmatrix}$$

$$d\hat{a}^{(1)} = \begin{bmatrix} \frac{\partial L}{\partial \hat{a}_{11}^{(1)}} & \frac{\partial L}{\partial \hat{a}_{12}^{(1)}} & \frac{\partial L}{\partial \hat{a}_{13}^{(1)}} \end{bmatrix} = \begin{bmatrix} da_{11}^{(2)} \frac{\partial a_{11}^{(2)}}{\partial \hat{a}_{11}^{(1)}} & da_{11}^{(2)} \frac{\partial a_{11}^{(2)}}{\partial \hat{a}_{12}^{(1)}} & da_{11}^{(2)} \frac{\partial a_{11}^{(2)}}{\partial \hat{a}_{13}^{(1)}} \end{bmatrix}$$
$$= \begin{bmatrix} da_{11}^{(2)} w_{11}^{(2)} & da_{11}^{(2)} w_{21}^{(2)} & da_{11}^{(2)} w_{31}^{(2)} \end{bmatrix} = \begin{bmatrix} d\hat{a}_{11}^{(1)} & d\hat{a}_{12}^{(1)} & d\hat{a}_{13}^{(1)} \end{bmatrix}$$

$$da^{(1)} = \begin{bmatrix} \frac{\partial L}{\partial a_{11}^{(1)}} & \frac{\partial L}{\partial a_{12}^{(1)}} & \frac{\partial L}{\partial a_{13}^{(1)}} \end{bmatrix} = \begin{bmatrix} d\hat{a}_{11}^{(1)} \frac{\partial \hat{a}_{11}^{(1)}}{\partial a_{11}^{(1)}} & d\hat{a}_{12}^{(1)} \frac{\partial \hat{a}_{12}^{(1)}}{\partial a_{12}^{(1)}} & d\hat{a}_{13}^{(1)} \frac{\partial \hat{a}_{13}^{(1)}}{\partial a_{13}^{(1)}} \end{bmatrix}$$
$$= \begin{bmatrix} d\hat{a}_{11}^{(1)}(\hat{a}_{11}^{(1)})(1-\hat{a}_{11}^{(1)}) & d\hat{a}_{12}^{(1)}(\hat{a}_{12}^{(1)})(1-\hat{a}_{12}^{(1)}) & d\hat{a}_{13}^{(1)}(\hat{a}_{13}^{(1)})(1-\hat{a}_{13}^{(1)}) \end{bmatrix}$$
$$= \begin{bmatrix} da_{11}^{(1)} & da_{12}^{(1)} & da_{13}^{(1)} \end{bmatrix}$$

$$dw^{(1)} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}^{(1)}} & \frac{\partial L}{\partial w_{12}^{(1)}} & \frac{\partial L}{\partial w_{13}^{(1)}} \\ \frac{\partial L}{\partial w_{21}^{(1)}} & \frac{\partial L}{\partial w_{22}^{(1)}} & \frac{\partial L}{\partial w_{23}^{(1)}} \end{bmatrix} = \begin{bmatrix} da_{11}^{(1)} \frac{\partial a_{11}^{(1)}}{\partial w_{11}^{(1)}} & da_{12}^{(1)} \frac{\partial a_{12}^{(1)}}{\partial w_{12}^{(1)}} & da_{13}^{(1)} \frac{\partial a_{13}^{(1)}}{\partial w_{13}^{(1)}} \\ da_{11}^{(1)} \frac{\partial a_{11}^{(1)}}{\partial w_{21}^{(1)}} & da_{12}^{(1)} \frac{\partial a_{12}^{(1)}}{\partial w_{22}^{(1)}} & da_{13}^{(1)} \frac{\partial a_{13}^{(1)}}{\partial w_{23}^{(1)}} \end{bmatrix}$$
$$= \begin{bmatrix} da_{11}^{(1)} \hat{a}_{11}^{(0)} & da_{12}^{(1)} \hat{a}_{11}^{(0)} & da_{13}^{(1)} \hat{a}_{11}^{(0)} \\ da_{11}^{(1)} \hat{a}_{12}^{(0)} & da_{12}^{(1)} \hat{a}_{12}^{(0)} & da_{13}^{(1)} \hat{a}_{12}^{(0)} \end{bmatrix} = \begin{bmatrix} dw_{11}^{(1)} & dw_{12}^{(1)} & dw_{13}^{(1)} \\ dw_{21}^{(1)} & dw_{22}^{(1)} & dw_{23}^{(1)} \end{bmatrix}$$

$$db^{(1)} = \begin{bmatrix} da_{11}^{(1)} \frac{\partial a_{11}^{(1)}}{\partial b_{11}^{(1)}} & da_{12}^{(1)} \frac{\partial a_{12}^{(1)}}{\partial b_{12}^{(1)}} & da_{13}^{(1)} \frac{\partial a_{13}^{(1)}}{\partial b_{13}^{(1)}} \end{bmatrix} = \begin{bmatrix} da_{11}^{(1)}(1) & da_{12}^{(1)}(1) & da_{13}^{(1)}(1) \end{bmatrix}$$
$$= \begin{bmatrix} db_{11}^{(1)} & db_{12}^{(1)} & db_{13}^{(1)} \end{bmatrix}$$

$$d\hat{a}^{(0)} = \begin{bmatrix} \frac{\partial L}{\partial \hat{a}_{11}^{(0)}} & \frac{\partial L}{\partial \hat{a}_{12}^{(0)}} \end{bmatrix}$$
$$= \begin{bmatrix} da_{11}^{(1)} \frac{\partial a_{11}^{(1)}}{\partial \hat{a}_{11}^{(0)}} + da_{12}^{(1)} \frac{\partial a_{12}^{(1)}}{\partial \hat{a}_{11}^{(0)}} + da_{13}^{(1)} \frac{\partial a_{13}^{(1)}}{\partial \hat{a}_{11}^{(0)}} & da_{11}^{(1)} \frac{\partial a_{11}^{(1)}}{\partial \hat{a}_{12}^{(0)}} + da_{12}^{(1)} \frac{\partial a_{12}^{(1)}}{\partial \hat{a}_{12}^{(0)}} + da_{13}^{(1)} \frac{\partial a_{13}^{(1)}}{\partial \hat{a}_{12}^{(0)}} \end{bmatrix}$$
$$= \begin{bmatrix} da_{11}^{(1)} w_{11}^{(1)} + da_{12}^{(1)} w_{12}^{(1)} + da_{13}^{(1)} w_{13}^{(1)} & da_{11}^{(1)} w_{21}^{(1)} + da_{12}^{(1)} w_{21}^{(1)} + da_{13}^{(1)} w_{31}^{(1)} \end{bmatrix}$$
$$= \begin{bmatrix} d\hat{a}_{11}^{(0)} & d\hat{a}_{12}^{(0)} \end{bmatrix}$$

After we finish all these calculation, we can conclude a general formula to update weight and bias terms. Note that the "@" used here means dot product.

$d\hat{a}^{(i)} = da^{(i+1)} @ w^{(i+1)}.T$

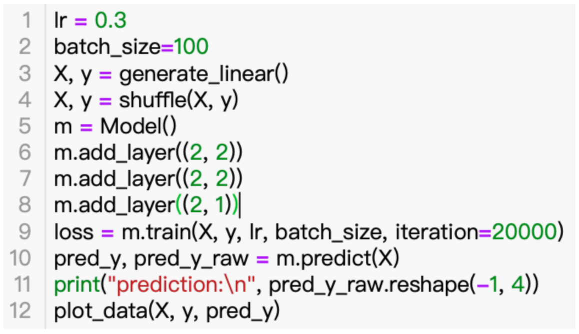
$da^{(i)} = d\hat{a}^{(i)} * \hat{a}^{(i)} * (1 - \hat{a}^{(i)})$

$dw^{(i)} = \hat{a}^{(i-1)}.T @ da^{(i)}$

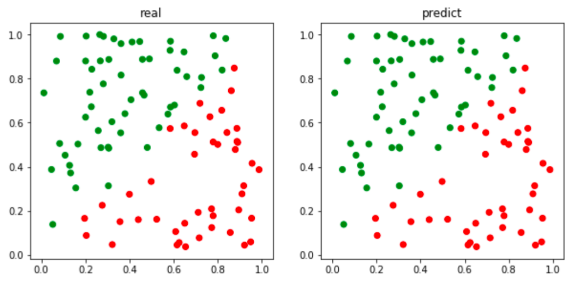
$db^{(i)} = 1 * da^{(i)}$

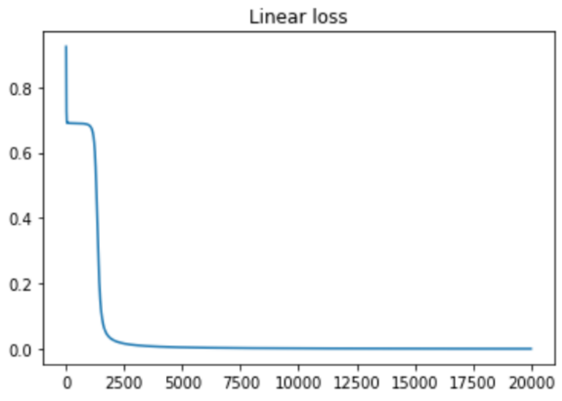
3 Results

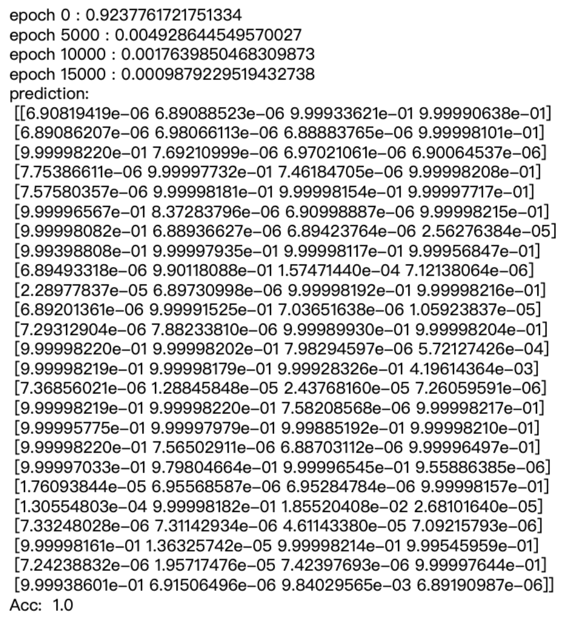
For linear classification, there are 50 data for both class 1 and 2. The learning rate is set to 0.3 and the epoch is set to 20000.



```
lr = 0.3
batch_size=100
X, y = generate_linear()
X, y = shuffle(X, y)
m = Model()
m.add_layer((2, 2))
m.add_layer((2, 2))
m.add_layer((2, 1))
loss = m.train(X, y, lr, batch_size, iteration=20000)
pred_y, pred_y_raw = m.predict(X)
print("prediction:\n", pred_y_raw.reshape(-1, 4))
plot_data(X, y, pred_y)
```

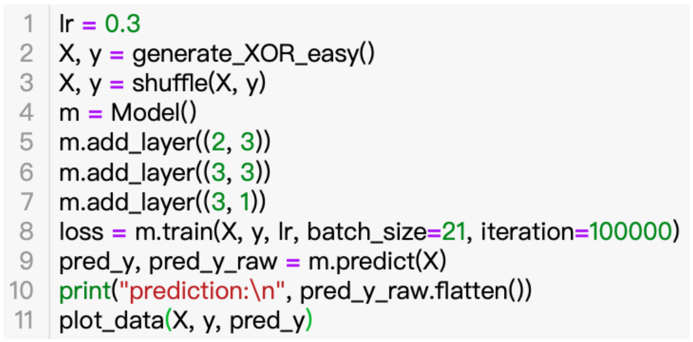




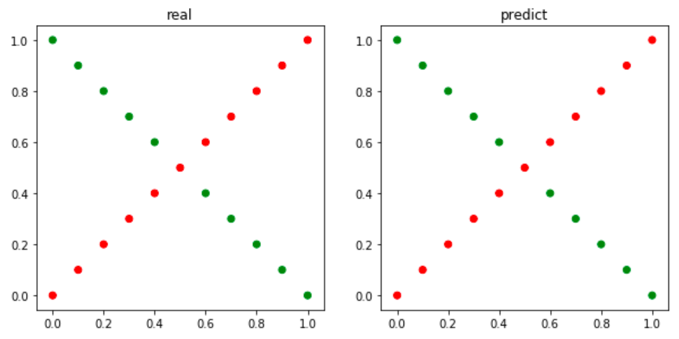


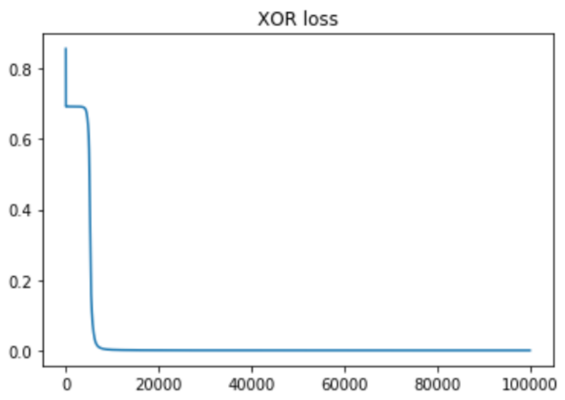
```
epoch 0 : 0.9237761721751334
epoch 5000 : 0.004928644549570027
epoch 10000 : 0.0017639850468309873
epoch 15000 : 0.0009879229519432738
prediction:
 [[6.90819419e-06 6.89088523e-06 9.99933621e-01 9.99990638e-01]
 [6.89086207e-06 6.98066113e-06 6.88883765e-06 9.99998101e-01]
 [9.99998220e-01 7.69210999e-06 6.97021061e-06 6.90064537e-06]
 [7.75386611e-06 9.99997732e-01 7.46184705e-06 9.99998208e-01]
 [7.57580357e-06 9.99998181e-01 9.99998154e-01 9.99997717e-01]
 [9.99996567e-01 8.37283796e-06 6.90998887e-06 9.99998215e-01]
 [9.99998082e-01 6.88936627e-06 6.89423764e-06 2.56276384e-05]
 [9.99398808e-01 9.99997935e-01 9.99998117e-01 9.99956847e-01]
 [6.89493318e-06 9.90118088e-01 1.57471440e-04 7.12138064e-06]
 [2.28977837e-05 6.89730998e-06 9.99998192e-01 9.99998216e-01]
 [6.89201361e-06 9.99991525e-01 7.03651638e-06 1.05923837e-05]
 [7.29312904e-06 7.88233810e-06 9.99989930e-01 9.99998204e-01]
 [9.99998220e-01 9.99998202e-01 7.98294597e-06 5.72127426e-04]
 [9.99998219e-01 9.99998179e-01 9.99928326e-01 4.19614364e-03]
 [7.36856021e-06 1.28845848e-05 2.43768160e-05 7.26059591e-06]
 [9.99998219e-01 9.99998220e-01 7.58208568e-06 9.99998217e-01]
 [9.99995775e-01 9.99997979e-01 9.99885192e-01 9.99998210e-01]
 [9.99998220e-01 7.56502911e-06 6.88703112e-06 9.99996497e-01]
 [9.99997033e-01 9.79804664e-01 9.99996545e-01 9.55886385e-06]
 [1.76093844e-05 6.95568587e-06 6.95284784e-06 9.99998157e-01]
 [1.30554803e-04 9.99998182e-01 1.85520408e-02 2.68101640e-05]
 [7.33248028e-06 7.31142934e-06 4.61143380e-05 7.09215793e-06]
 [9.99998161e-01 1.36325742e-05 9.99998214e-01 9.99545959e-01]
 [7.24238832e-06 1.95717476e-05 7.42397693e-06 9.99997644e-01]
 [9.99938601e-01 6.91506496e-06 9.84029565e-03 6.89190987e-06]]
Acc:  1.0
```

For XOR classification, there are 10 and 11 data for class 1 and 2 respectively. The learning rate is set to 0.3 and the epoch is set to 100000.



```
lr = 0.3
X, y = generate_XOR_easy()
X, y = shuffle(X, y)
m = Model()
m.add_layer((2, 3))
m.add_layer((3, 3))
m.add_layer((3, 1))
loss = m.train(X, y, lr, batch_size=21, iteration=100000)
pred_y, pred_y_raw = m.predict(X)
print("prediction:\n", pred_y_raw.flatten())
plot_data(X, y, pred_y)
```





```
epoch 0 : 0.8556205601001917
epoch 5000 : 0.5711453774455718
epoch 10000 : 0.001993246499807004
epoch 15000 : 0.0007175145539242605
epoch 20000 : 0.0004164768705575398
epoch 25000 : 0.0002877692602987921
epoch 30000 : 0.00021762873576804343
epoch 35000 : 0.00017391792093485889
epoch 40000 : 0.00014424657556554125
epoch 45000 : 0.00012287414287718762
epoch 50000 : 0.00010679321680969488
epoch 55000 : 9.428257650548001e-05
epoch 60000 : 8.428898279397079e-05
epoch 65000 : 7.61332498183649e-05
epoch 70000 : 6.935850088260488e-05
epoch 75000 : 6.364652232278121e-05
epoch 80000 : 5.876912137941219e-05
epoch 85000 : 5.455855125617446e-05
epoch 90000 : 5.0888838506633774e-05
epoch 95000 : 4.766360642509997e-05
prediction:
 [9.99999617e-01 4.99861515e-08 6.65580646e-06 2.03335408e-05
 3.42511670e-07 9.99999462e-01 9.99999678e-01 1.07016141e-05
 9.99999626e-01 9.99746762e-01 9.99999655e-01 2.67489546e-08
 1.86902482e-04 7.17750607e-05 9.99999428e-01 1.34635227e-04
 2.79883321e-06 9.99751300e-01 9.99999673e-01 1.47559646e-06
 9.99999623e-01]
Acc:  1.0
```

In both experiments, the converging speeds are quite fast. They take less than 2500 epochs and 10000 epochs to converge respectively.

4 Discussion

4.1 Vanishing gradient problem

The sigmoid function we use here actually has a problem: it will lead to vanishing gradient problem. The gradient will become smaller and smaller as the process of backpropagation goes. Thus, the layers that are close to input layers may not be updated well. As a result, when the network goes deeper, the gradient vanish more seriously. We can solve this problem by changing activation functions to ReLu, PReLu or ELU.

4.2 Optimizer choosing

Since stochastic gradient descent is slow and takes a lot of iteration to find local minimum, mathematician further use some tricks called optimizers to speed up the updating process. The updating direction of each optimizer is different, and thus there's not always a rule of thumb which optimizer we should use. As for the optimizer here, we choose SGD momentum since this is easy to implement. There're still a lot of choices which we're not implementing, including Adagrad, Adam, Adamax, etc.