4.1

(1) 答：不可以，因为三输入XOR问题仍然是线性不可分的，感知机无法处理。

(2) 答：可以，一个三输入、单隐层的MLP可以处理线性不可分的三输入XOR问题。

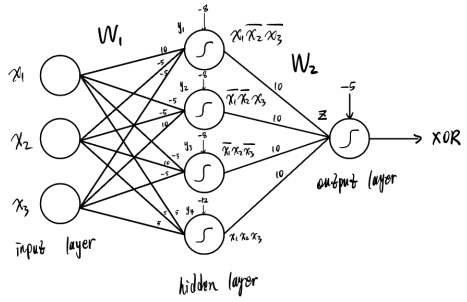
(3)

设计一个解决三输入XOR问题的单隐层MLP

解：由于 $A \oplus B \oplus C = ABC + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}$

∴ 可以让隐层有4个神经元

具体设计如下：



其中：

$$\begin{cases} y_1 = 10x_1 - 5x_2 - 5x_3 - 8 \\ y_2 = -5x_1 - 5x_2 + 10x_3 - 8 \\ y_3 = -5x_1 + 10x_2 - 5x_3 - 8 \\ y_4 = 5x_1 + 5x_2 + 5x_3 - 12 \\ z = 10y_1 + 10y_2 + 10y_3 + 10y_4 - 5 \end{cases} \quad W_1 = \begin{bmatrix} 10 & -5 & -5 \\ -5 & -5 & 10 \\ -5 & 10 & -5 \\ 5 & 5 & 5 \end{bmatrix}_{4\times 3} \quad W_2 = \begin{bmatrix} 10, 10, 10, 10 \end{bmatrix}_{1\times 4}$$

$$b_1 = \begin{bmatrix} -8 \\ -8 \\ -8 \\ -12 \end{bmatrix} \quad b_2 = [-5]$$

且激活函数均为 sigmoid.

当 $x_1x_2x_3 = 000 \sim 111$ 时，验算均正确。

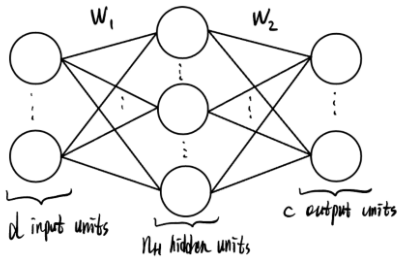
4.2

答：如果隐层的激活函数是线性的，那么即使有多个隐层，输入经过这些层后得到的最终输出是经过多个线性变换的组合得到的结果，等价于只经过一次线性变换得到的结果，因此三层网络本质上相当于两层的感知机，还是无法解决XOR问题。

4.3

(a) 答：权重共有 $d \cdot n_H + n_H \cdot c$ 个

(b)



记输入层－隐层间的权重矩阵为 W_1，隐层－输出层间的为 W_2

则 W_1 尺寸为 $n_H \times d$，W_2 尺寸为 $c \times n_H$

∴ number of weights = W_1 元素个数 + W_2 元素个数

$= n_H \times d + c \times n_H$

$= n_H \cdot (d + c)$

4.4

解：$f(x) = \frac{1}{1+e^{-ax}}$

$$\frac{df(x)}{dx} = \frac{-(-ae^{-ax})}{(1+e^{-ax})^2} = \frac{ae^{-ax}}{(1+e^{-ax})^2} = \frac{a}{1+e^{-ax}} \cdot \left(1 - \frac{1}{1+e^{-ax}}\right) = af(x)(1-f(x))$$

4.6

MLP实现代码如下：

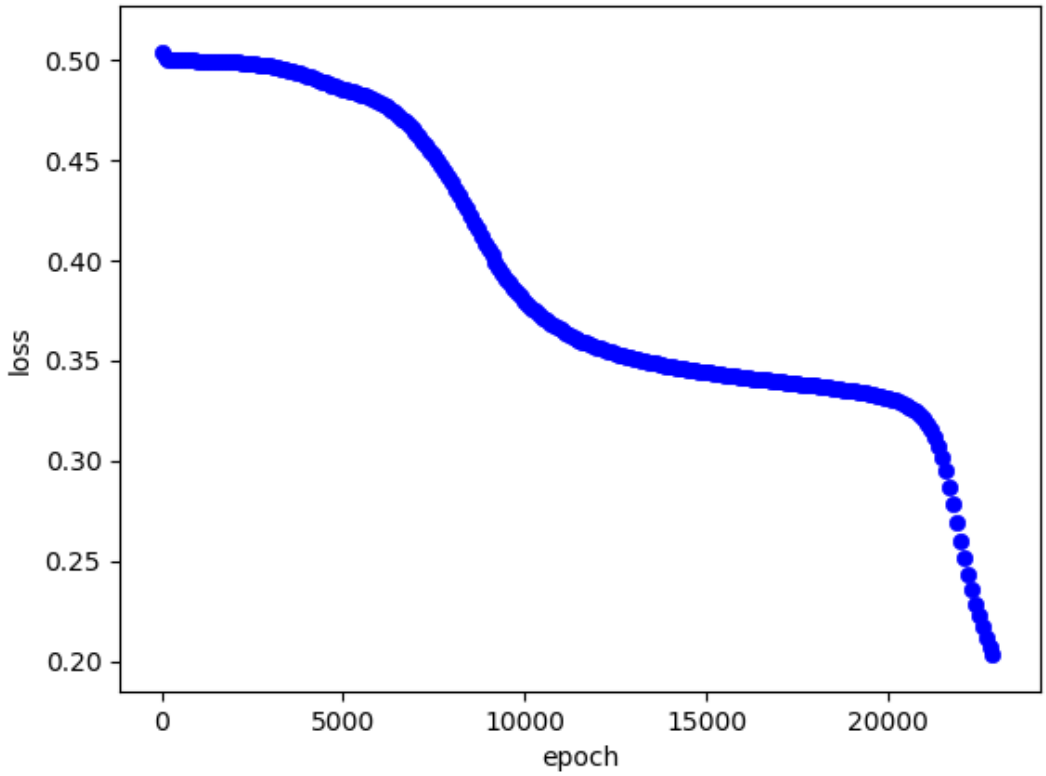
```
import numpy as np
from matplotlib import pyplot as plt

lr = 0.1 #学习率

class MLP():

    def __init__(self, inputNum, hiddenNum, outputNum):

        # 初始化权重及偏置参数
        self.W_1 = np.random.randn(hiddenNum, inputNum)
        self.W_2 = np.random.randn(outputNum, hiddenNum)
        self.b_1 = np.random.randn(hiddenNum, 1)
        self.b_2 = np.random.randn(outputNum, 1)

    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    # sigmoid的导数
    def d_s(x):
        return x * (1 - x)

    # 前向计算
    def forward(self, input):
        h_out = MLP.sigmoid(np.dot(self.W_1, input) + self.b_1)
        out = MLP.sigmoid(np.dot(self.W_2, h_out) + self.b_2)
        return h_out, out

    # 反向传播
    def backpropagation(self, input, h_out, out, y_true, lr = lr):
        # d_2是误差对输出层输出的偏导乘上输出层输出对输出层输入的偏导，尺寸为1x4
        d_2=(y_true - out) * MLP.d_s(out)
        # d_1是误差对隐层输入的偏导，尺寸为2x4
        d_1 = np.dot(self.W_2.T, d_2) * MLP.d_s(h_out)
        # d_W2是误差对W2矩阵的偏导，尺寸为1x2
        d_W2 = np.dot(d_2, h_out.T)
        # d_W1是误差对W1矩阵的偏导，尺寸为2x2
        d_W1 = np.dot(input, d_1.T)
        # 更新矩阵参数
        self.W_2 += lr * d_W2
        self.W_1 += lr * d_W1
        # 误差对偏置的偏导（4个输入的累加影响）
        d_b2 = float(sum(sum(d_2)))
        d_b1 = np.sum(d_1, axis=1).reshape(2,1)
        # 更新偏置参数
        self.b_2 += lr * d_b2
        self.b_1 += lr * d_b1


if __name__ == '__main__':

    # 实例化一个mlp
    mlp = MLP(2, 2, 1)
```

```
    # 给出输入及真实标签
    x_data = np.array([[0, 0, 1, 1],
                       [0, 1, 0, 1]])
    y_true = np.array([[0, 1, 1, 0]])
    # 不断迭代，当误差小于0.2时退出循环
    loss = 1.0
    i = 0
    losses = []
    while(loss > 0.20):
        i += 1
        # 前向传播，计算输出
        h_out, out = mlp.forward(x_data)
        # 反向传播，更新参数
        mlp.backpropagation(x_data, h_out, out, y_true)
        # 保存当前loss
        loss = np.mean(np.abs(out - y_true))
        losses.append(loss)
    # 可视化loss下降曲线
    for i in range(len(losses)):
        if i % 100 == 0:
            plt.scatter(i, losses[i], linewidths = 0.5, color = 'blue')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()
```

结果分析：

loss下降曲线如下。

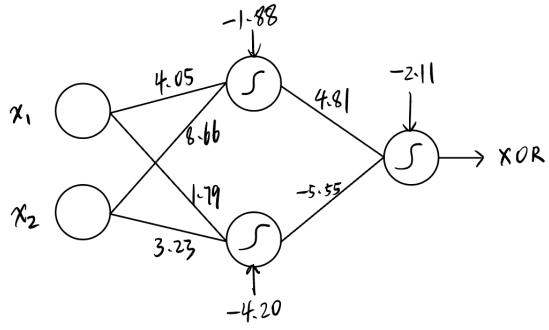
在实际训练MLP的过程中，可以发现在loss=0.34左右的地方存在局部最优点，如果学习率设置得太小，则一方面可能走不出该局部最优，另一方面可能需要过多的epoch才能走出局部最优；如果学习率设置得太大，则容易导致loss异常增大后再减小的情况（即先走出局部最优然后逐渐收敛到全局最优）或导致误差不收敛。因此要注意将学习率设定为一个合适的值。但是在实际运行程序的过程中仍然有可能因陷入局部最优无法出去而使程序卡死。



各层的间的权重矩阵及各层的偏置向量如下：

```
PS C:\Users\HP> & D:/Python3.6.6/python.exe c:/Users/HP/Desktop/MLP.py
输入层-隐层权值：
 [[4.05343331 8.66260454]
 [1.79145926 3.22758575]]
隐层-输出层权值：
 [[ 4.80787087 -5.54709488]]
隐层偏置：
 [[-1.88145423]
 [-4.19531846]]
输出层偏置：
 [[-2.11441529]]
预测结果：
 [[0.17359672 0.76139777 0.85083118 0.23837426]]
阈值过滤后的最终输出：
0
1
1
0
```

分析隐层神经元的函数：



隐层第1个神经元 { 输入 = $4.05x_1 + 8.66x_2 - 1.88$; 输出 = $\sigma(4.05x_1 + 8.66x_2 - 1.88)$

隐层第2个神经元 { 输入 = $1.79x_1 + 3.23x_2 - 4.20$; 输出 = $\sigma(1.79x_1 + 3.23x_2 - 4.20)$

(其中 $\sigma(x)$ 表示 sigmoid 函数)