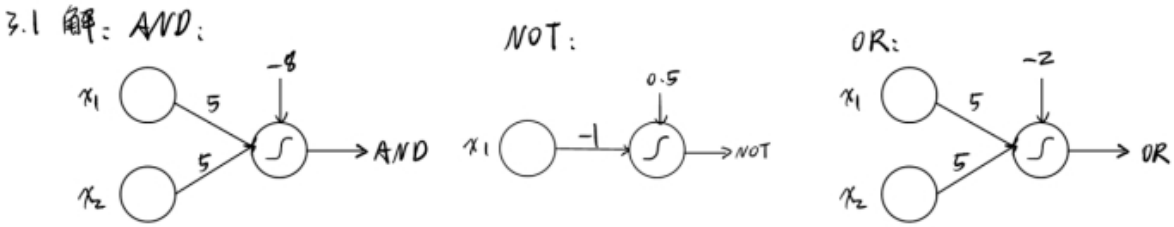
第五章 Perceptron

3-1 请考察二元逻辑函数AND、NOT、OR和XOR，(1) 是否它们是线性可分的？(2) 对于其中的线性可分的逻辑函数，请设计perceptron实现这些逻辑函数。

答：(1) AND、NOT、OR线性可分，XOR线性不可分



3-2 对于上述线性可分的逻辑函数，通过编程perceptron学习算法训练获得perceptron来实现它们，并与3-1得到的结果进行比较。

程序如下：

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

lr = 0.1
epoch = 10000

class Perceptron():

    def __init__(self, inputNum, outputNum):

        # 初始化权重及偏置参数
        self.W = np.random.randn(outputNum, inputNum)
        self.b = np.random.randn(outputNum, 1)

    # sigmoid激活函数
    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):
        out = Perceptron.sigmoid(np.dot(self.W, input) + self.b)
        return out

    # 反向传播
    def backpropagation(self, input, out, y_true, lr = lr):
        # d_W是误差对W矩阵的偏导，尺寸为1x2
        d_W = np.dot((y_true - out) * Perceptron.d_s(out), input.T)
        # 更新矩阵参数
        self.W += lr * d_W
```

```
        # 误差对偏置的偏导（4个输入的累加影响）
        d_b = float(sum(sum((y_true - out) * Perceptron.d_s(out))))
        # 更新偏置参数
        self.b += lr * d_b


if __name__ == '__main__':

    # 实例化一个perceptron
    perceptron = Perceptron(2,1)    # 与 或
    # perceptron = Perceptron(1,1)  # 非
    # 给出输入及真实标签
    x_data = np.array([[0, 0, 1, 1], # 与 或
                       [0, 1, 0, 1]])
    # x_data = np.array([[0, 1]])  # 非
    # 逻辑与
    y_data = np.array([[0, 0, 0, 1]])
    # 逻辑或
    # y_data = np.array([[0, 1, 1, 1]])
    # 逻辑非
    # y_data = np.array([[1, 0]])

    # 迭代
    losses = []
    for i in range(epoch):
        # 前向传播，计算输出
        out = perceptron.forward(x_data)
        # 反向传播，更新参数
        perceptron.backpropagation(x_data, out, y_data)
        loss = np.mean(np.abs(y_data - out))
        losses.append(loss)
    #绘制loss曲线图
    for i in range(epoch):
        if i % 100 == 0:
            plt.scatter(i, losses[i], color='blue')
    plt.xlabel('eopch')
    plt.ylabel('loss')
    plt.show()
    print('输入层到输出层权值:\n', perceptron.w)
    print('输出层偏置: \n', perceptron.b)
    print('最终结果:\n',out)
    print('忽略误差的近似输出:')
    # 设定sigmoid激活后的阈值为0.5
    for i in sum(out):
        if i < 0.5:
            print(0)
        else:
            print(1)
```

运行结果如下：

逻辑与：

```
输入层到输出层权值:
 [[5.46824022 5.46824021]]
输出层偏置:
 [[-8.29510229]]
最终结果:
 [[2.49718748e-04 5.58928338e-02 5.58928343e-02 9.33473892e-01]]
忽略误差的近似输出:
0
0
0
1
```

我在3-1中的设计的逻辑与感知机输入层到隐层的权值为5，5；输出层偏置为-8，和程序运行结果接近，这验证了我设计的正确性。

逻辑或:

```
输入层到输出层权值:
 [[6.15421828 6.15419649]]
输出层偏置:
 [[-2.83071127]]
最终结果:
 [[0.05569011 0.96522384 0.96522457 0.99992345]]
忽略误差的近似输出:
0
1
1
1
```

我在3-1中的设计的逻辑或感知机输入层到隐层的权值为5，5；输出层偏置为-2，和程序运行结果接近，这验证了我设计的正确性。

逻辑非:

```
输入层到输出层权值:
 [[-6.47672839]]
输出层偏置:
 [[3.13029432]]
最终结果:
 [[0.95812295 0.03401395]]
忽略误差的近似输出:
1
0
```

我在3-1中的设计的逻辑非感知机输入层到隐层的权值为-1；输出层偏置为0.5，和程序运行结果接近，只是相比而言在数值上缩小了6倍，这验证了我设计的正确性。

3-3 分类手写数字0，1，2，...，10，(1)如果采用一对多策略，至少需要用多少个二分类器来完成? (2)所对应的编码矩阵是怎样的？(3)对于一个测试样本，怎样通过这些二分类器的判别结果，给出这个样本属于哪一类的决策?

答：(1) 10个

(2)

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0
9	0	0	0	0	0	0	0	0	0	1	0
10	0	0	0	0	0	0	0	0	0	0	1

(3) 将所有二分类器的分类结果构成一个向量，求该向量与编码矩阵每一行构成的向量的距离，和哪一行的距离最小，就根据该行得出这个样本属于哪一类。

3-4 两元逻辑函数XOR不是一个线性可分问题，因此该函数无法用Perceptron实现。请编程Perceptron学习算法，学习出这个XOR函数的广义Perceptron实现。

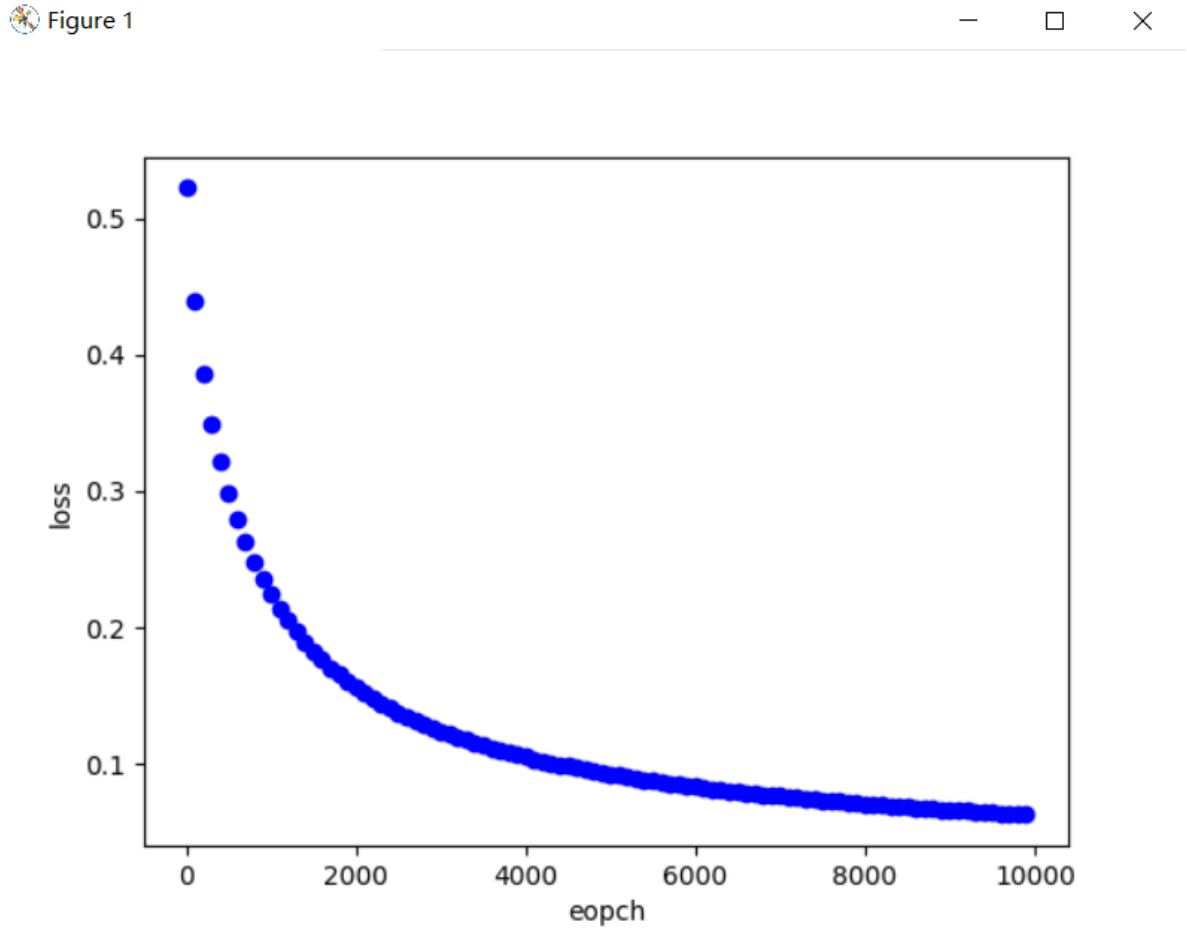
广义Perceptron实现代码如下，和3-2题相比，主体perceptron的类不变，只是将输入的二维向量x_1, x_2扩增为了三维空间中的x_1, x_2, x_1x_2，使得在三维空间中能够用perceptron解决XOR问题。

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

lr = 0.1
epoch = 10000

class Perceptron():

    def __init__(self, inputNum, outputNum):

        # 初始化权重及偏置参数
        self.W = np.random.randn(outputNum, inputNum)
        self.b = np.random.randn(outputNum, 1)

    # sigmoid激活函数
    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):
        out = Perceptron.sigmoid(np.dot(self.W, input) + self.b)
        return out

    # 反向传播
    def backpropagation(self, input, out, y_true, lr = lr):
        # d_W是误差对W矩阵的偏导，尺寸为1x2
```

```
        d_W = np.dot((y_true - out) * Perceptron.d_s(out), input.T)
        # 更新矩阵参数
        self.W += lr * d_W
        # 误差对偏置的偏导（4个输入的累加影响）
        d_b = float(sum(sum((y_true - out) * Perceptron.d_s(out))))
        # 更新偏置参数
        self.b += lr * d_b


if __name__ == '__main__':

    # 实例化一个perceptron
    perceptron = Perceptron(3,1)
    # 给出输入及真实标签
    x_data = np.array([[0, 0, 1, 1],
                       [0, 1, 0, 1],
                       [0, 0, 0, 1]])
    # 异或
    y_data = np.array([[0, 1, 1, 0]])

    # 迭代
    losses = []
    for i in range(epoch):
        # 前向传播，计算输出
        out = perceptron.forward(x_data)
        # 反向传播，更新参数
        perceptron.backpropagation(x_data, out, y_data)
        loss = np.mean(np.abs(y_data - out))
        losses.append(loss)
    #绘制loss曲线图
    for i in range(epoch):
        if i % 100 == 0:
            plt.scatter(i, losses[i], color='blue')
    plt.xlabel('eopch')
    plt.ylabel('loss')
    plt.show()
    print('输入层到输出层权值:\n', perceptron.W)
    print('输出层偏置: \n', perceptron.b)
    print('最终结果:\n',out)
    print('忽略误差的近似输出:')
    # 设定sigmoid激活后的阈值为0.5
    for i in sum(out):
        if i < 0.5:
            print(0)
        else:
            print(1)
```

程序运行结果如下：

训练loss曲线：

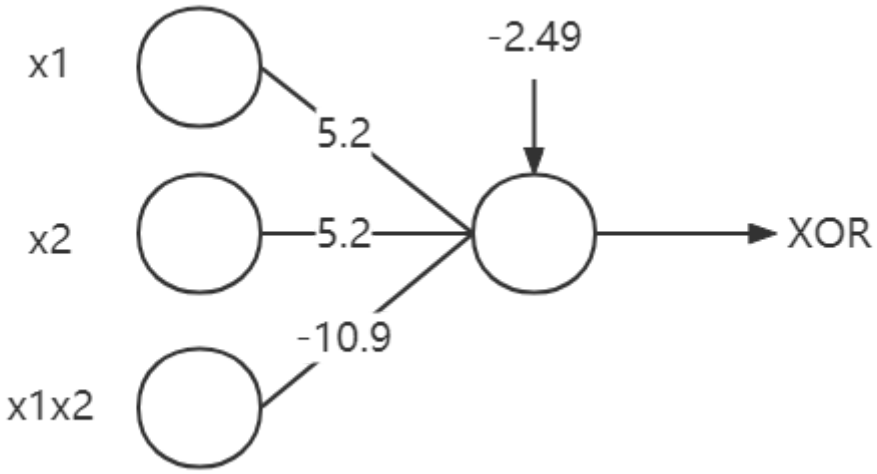


各层权值及偏置项如下：

```
输入层到输出层权值：
 [[  5.21048445   5.21048445 -10.87288501]]
输出层偏置：
 [[-2.49101236]]
最终结果：
 [[0.07649497 0.93816247 0.93816247 0.05007457]]
忽略误差的近似输出：
0
1
1
0
```

通过训练完成后perceptron的预测输出我们可以发现，当把原始问题维数扩增到三维(x_1, x_2, x_1x_2)后，已经能够使用perceptron对XOR问题进行正确的解决。

最终的perceptron模型示意图如下：



最终输出函数为：$y = s(5.2x_1 + 5.2x_2 - 10.9x_1x_2 - 2.49)$，其中$s(x)$为sigmoid函数。当$x_1, x_2$分别取0 0，0 1，1 0，1 1时，手工验证输出结果为0 1 1 0，结果正确，说明训练好的广义perceptron能够解决XOR问题。