山东大学

崇新学堂

2023－2024 学年第一学期

实 验 报 告

课程名称：信息基础 II

专 业 班 级 崇新学堂 21 级

学 生 姓 名 刘浩

个 人 学 号 202120120312

实验一：常规神经网络函数逼近实验

一、 实验要求

函数逼近问题是指在不给出函数关系式的前提下，仅通过大量的函数值对应实例对神经网络进行训练，使得神经网络可以根据一个未知的自变量预测对应的应变量值，在本实验中我采用手动搭建 BP 神经网络，使用三层神经网络进行函数逼近，分别对 XOR 和 $y = \frac{1}{\sin x} + \frac{1}{\cos x}$ 进行函数逼近操作

二、 实验原理

此部分主要参考了知乎和 CSDN 的资料[1]，对 BP 神经网络加以理解，下面是我关于本实验中 BP 神经网络的理解和思考过程，以我做第一个 XOR 函数逼近的神经网络为例，我采取输入层两个神经元，隐藏层四个神经元，输出层一个神经元，如图 1 所示：

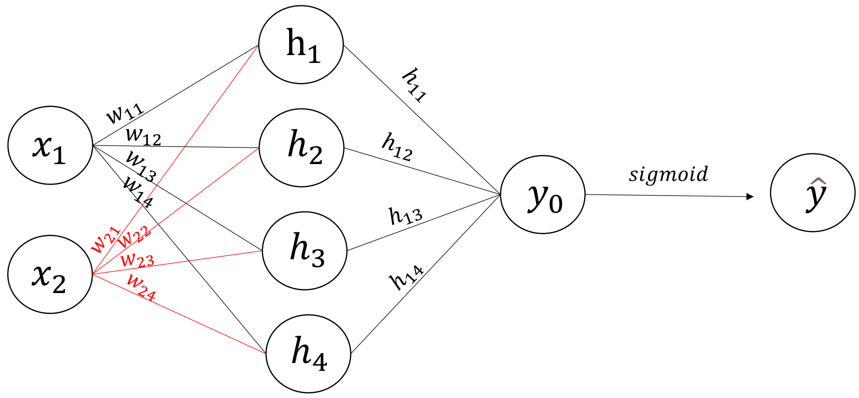

图 1 XOR 逼近使用的神经网络

因此第一层的权重矩阵可以表示为：

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \tag{1}$$

在 XOR 拟合中输入为二维的四种组合(0,0), (0,1), (1,0), (1,1),输入矩阵可以表示为：

$$[x_1 \quad x_2] \tag{2}$$

为此从输入层到隐藏层可以表示为：

$$[x_1 \quad x_2]\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \tag{3}$$

再加上隐藏层的偏置项可以表示为：

$$[x_1 \quad x_2]\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} + [b_1 \quad b_2 \quad b_3 \quad b_4] \tag{4}$$

为此隐藏层的输出只需要经过激活函数即：

$$[u_1 \quad u_2 \quad u_3 \quad u_4] = sigmoid\left([x_1 \quad x_2]\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} + [b_1 \quad b_2 \quad b_3 \quad b_4]\right) \tag{5}$$

同理，从隐藏层到输出层：

$$[u_1 \quad u_2 \quad u_3 \quad u_4]\begin{bmatrix} h_{11} \\ h_{21} \\ h_{31} \\ h_{41} \end{bmatrix} + b_2 \tag{6}$$

那么最终的输出为：

$$\hat{y} = sigmoid\left([u_1 \quad u_2 \quad u_3 \quad u_4]\begin{bmatrix} h_{11} \\ h_{21} \\ h_{31} \\ h_{41} \end{bmatrix} + b_2\right) \tag{7}$$

至此，前向传播过程结束，下面进行反向传播的过程：

首先是损失函数的定义，采取 MSE 损失函数，其计算公式如公式(8)所示：

$$L = \frac{1}{2}(y - \hat{y})^2 \tag{8}$$

那么为了逐渐向着目标靠近，我们只需要让 L 尽可能的小即可所以我们利用梯度下降原理，采取求偏导，然后结合学习率进行训练，在代码中我采用链式法则(分步求偏导的方式)进行，以简化手工计算，我手工推算的过程如下(也是我代码中 backword 函数写法的来源)：

首先是L对$\hat{y}$的偏导：

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y \tag{9}$$

然后是L对b_2的偏导(其中$dsigmoid$表示$sigmoid$函数的导数)：

- 注：此处最开始推导的时候我采取了一种比较**笨重的**计算导数的方式，在实验感想中，对比了两种求导方式训练时间的大小，发现采取公式(10)计算导数是最**方便快捷**的！

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} dsigmoid(\hat{y}) \tag{10}$$

L 对W_2(隐藏层到输出层权重矩阵)的偏导，不妨记隐藏层的输出为 H：

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial b_2} H^T \tag{11}$$

L 对b_1(输入层到隐藏层的偏置矩阵)的偏导：

$$\frac{\partial L}{\partial b_1} = W_2 \frac{\partial L}{\partial b_2} dsigmoid(H) \tag{12}$$

L 对W_1(输入层到隐藏层的权重矩阵)的偏导：

$$\frac{\partial L}{\partial W_1} = X \frac{\partial L}{\partial b_1} \tag{13}$$

再结合学习率进行更新(其中 lr 是学习率)：

$$\begin{cases} W_1 = W_1 - lr * \dfrac{\partial L}{\partial W_1} \\ W_2 = W_2 - lr * \dfrac{\partial L}{\partial W_2} \\ b_1 = b_1 - lr * \dfrac{\partial L}{\partial b_1} \\ b_2 = b_2 - lr * \dfrac{\partial L}{\partial b_2} \end{cases} \tag{14}$$

至此反向传播过程结束，在这个过程中权重矩阵和偏置矩阵不断更新，最终的结果是 loss 不断减小，输出的结果也越来越准确；

在训练过程中激活函数的选取也非常重要，例如在逼近$y = \dfrac{1}{\sin x} + \dfrac{1}{\cos x}$的时候，我对隐藏层采取的是tanh 函数作为激活函数，而输出层则采取不使用激活函数的形式。

另外对于隐藏层神经元数量的选择，对于 XOR 函数数据集较少，我隐藏层只采用了四个神经元；而对于第二个连续函数，数据集较大而且拟合的函数比较复杂，我隐藏层采用了 100 个神经元。

三、 实验步骤

3.1 XOR 函数的逼近：

我按照我的实验原理中的思路我完成了这部分的代码：

定义激活函数$sigmoid$和其导数：

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


def dsigmoid(z):
    return z * (1 - z)
```

然后我定义了一个 BP 类，用来搭建我的神经网络：

```
class BP:
    def __init__(self, input_layer, hidden_layer, epoch, lr):
        self.input_layer = input_layer
        self.hidden_layer = hidden_layer
        self.epoch = epoch
        self.lr = lr

        self.weight1 = np.random.uniform(-0.5, 0.5, (input_layer, hidden_layer))
        self.weight2 = np.random.uniform(-0.5, 0.5, (hidden_layer, 1))

        self.bias1 = np.zeros((1, hidden_layer))
        self.bias2 = np.zeros((1, 1))
```

```
    def lossfunc(self, y, y_hat):
        loss = np.mean(0.5 * (y - y_hat) ** 2)
        return loss

    def forward(self, X):
        H = np.dot(X.T, self.weight1) + self.bias1
        hidden_out = sigmoid(H)
        out = np.dot(hidden_out, self.weight2) + self.bias2
        y_hat = sigmoid(out)
        return y_hat, hidden_out

    def backword(self, x, y, y_hat, hidden_out):
        pianLy_hat = y_hat - y
        pianLb2 = pianLy_hat * dsigmoid(y_hat)
        pianLW2 = np.dot(hidden_out.T, pianLb2)
        pianLb1 = np.dot(pianLb2, self.weight2.T) * dsigmoid(hidden_out)
        pianLW1 = np.dot(x, pianLb1)

        self.weight1 -= self.lr * pianLW1
        self.weight2 -= self.lr * pianLW2
        self.bias1 -= self.lr * pianLb1
        self.bias2 -= self.lr * pianLb2

    def train(self, inp, out):
        loss = []
        for i in range(self.epoch):
            epoch_loss = 0.0
            for (x, target) in zip(inp, out):
                x = x.reshape(-1, 1)
                y_hat, hidden_out = self.forward(x)
                self.backword(x, target, y_hat, hidden_out)
                epoch_loss += self.lossfunc(target, y_hat)

            loss.append(epoch_loss / len(inp))

            if i % 1000 == 0:
                print(f"Epoch: {str(i).ljust(5)} | Loss: {str(loss[1]).ljust(18)}")

        return loss


    def test(self, inp):
        result = []
```

```
        for x in inp:
            x = x.reshape(-1, 1)
            y_hat, hidden_out = self.forward(x)
            result.append(y_hat)
        return np.round(result)
```

前向传播过程和后向传播过程均利用我在实验原理推导的公式。

训练过程代码的思路主要是：每次训练，我们读取训练数据，然后前向传播和反向传播来调整参数，我采取一个中间变量 epoch_loss 存储每个训练数据集的 loss，然后在数据集全部训练完后得到累加的 epoch_loss，最后这个 epoch 训练完，将其归一化添加到 loss 列表中作为这次训练的结果，然后为了可以看到输出过程，我每隔 1000 次训练输出一次 loss。

测试过程的代码思路主要是：读取测试集数据，再次进行前向传播，得到预测值，将预测值添加到 result 列表中，由于 XOR 只有四个数据，最后我们打印 result 出来即可。

代码运行结果如下：

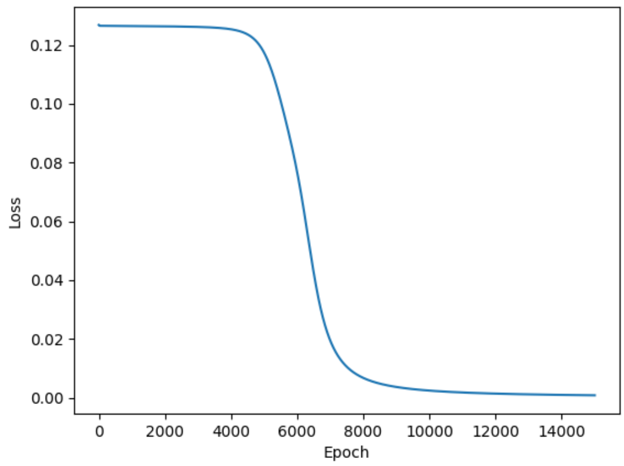

图 2 XOR 拟合过程损失函数变化

且预测结果：

```
Predictions:
[[[0.]]

 [[1.]]

 [[1.]]

 [[0.]]]
```

可以看到准确率高达 100%

3.2 函数 $y = \dfrac{1}{\sin x} + \dfrac{1}{\cos x}$ 逼近：

和 XOR 函数类似，只不过这次激活函数发生变化，我采用了 tanh 作为隐藏层的激活函数，而输出层不采用激活函数，新的激活函数如下：

```python
def tanh(z):
    return np.tanh(z)
def dtanh(z):
    return 1 - z ** 2
```

对于训练集数据的读取我采用在$\left(-\dfrac{\pi}{2},\dfrac{\pi}{2}\right)$上等间隔取 100 个点，再计算它们的 y 值即可

```python
# 生成训练数据
x = np.linspace(-np.pi / 2, np.pi / 2, 100)
train_x = x.reshape(-1, 1)
train_y = np.clip(1 / np.sin(x) + 1 / np.cos(x), -50, 50)
```

将训练数据可视化得到结果如图 3 所示，利用 Desmos 可视化该连续函数如图图 4 图 4 Desmos 可视化所示

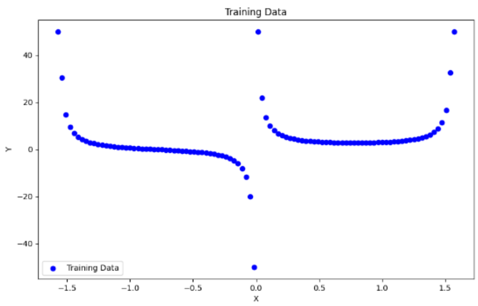

图 3 训练集数据可视化

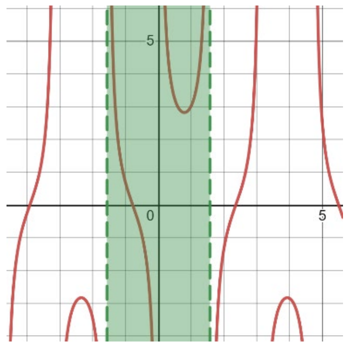

图 4 Desmos 可视化

可以验证，训练集所选数据是**正确**的。

除此以外，由于激活函数发生改变，反向传播过程也发生了变化，原来由于输出层也存在激活函数，所以输出层对隐藏层的参数求偏导的时候，需要乘以激活函数的导数，此时不再需要，新的反向传播过程如下：

```python
def backward(self, x, y, y_hat, hidden_out):
    pianLy_hat = y_hat - y
    pianLb2 = pianLy_hat
    pianLW2 = np.dot(hidden_out.T, pianLb2)
    pianLb1 = np.dot(pianLb2, self.weight2.T) * dtanh(hidden_out)
    pianLW1 = np.dot(x, pianLb1)

    self.weight1 -= self.lr * pianLW1
    self.weight2 -= self.lr * pianLW2
    self.bias1 -= self.lr * pianLb1
    self.bias2 -= self.lr * pianLb2
```

新的前向传播过程如下：

```python
def forward(self, X):
    H = np.dot(X.T, self.weight1) + self.bias1
    hidden_out = tanh(H)
    out = np.dot(hidden_out, self.weight2) + self.bias2
    y_hat = out
    return y_hat, hidden_out
```

训练过程和测试过程和 XOR 思路相同，此处不再赘述。

我测试集数据的选取是在$\left(-\dfrac{\pi}{2},\dfrac{\pi}{2}\right)$上，等间隔选了 200 个点，这样测试集中就存

在和之前训练集中不同的点，达到测试的效果；

代码运行结果如下：

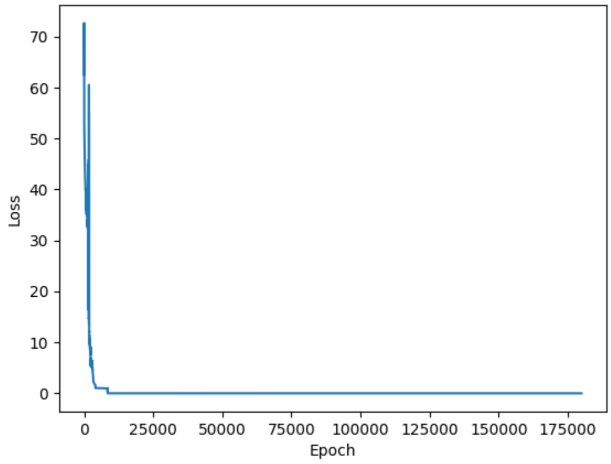

图 5 连续函数逼近的 Loss 变化

测试集测试结果可视化：

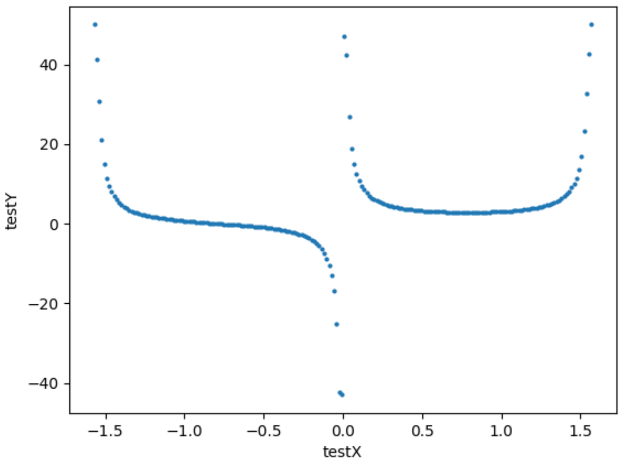

图 6 测试集结果可视化

四、 实验结果

实验结果一览：

4.1 XOR 逼近

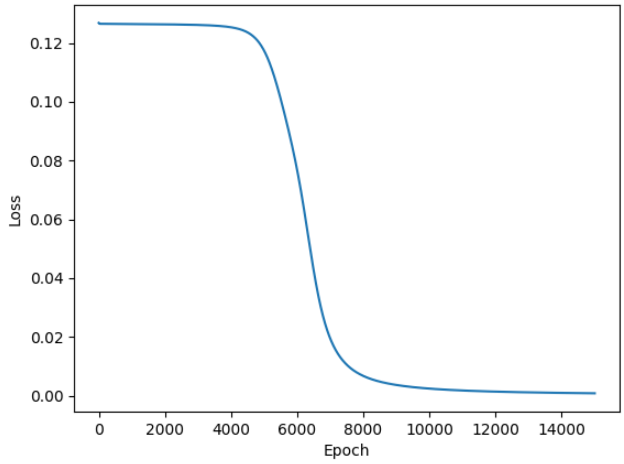

图 7 XOR 逼近实验结果

4.2 函数 $y = \dfrac{1}{\sin x} + \dfrac{1}{\cos x}$ 逼近

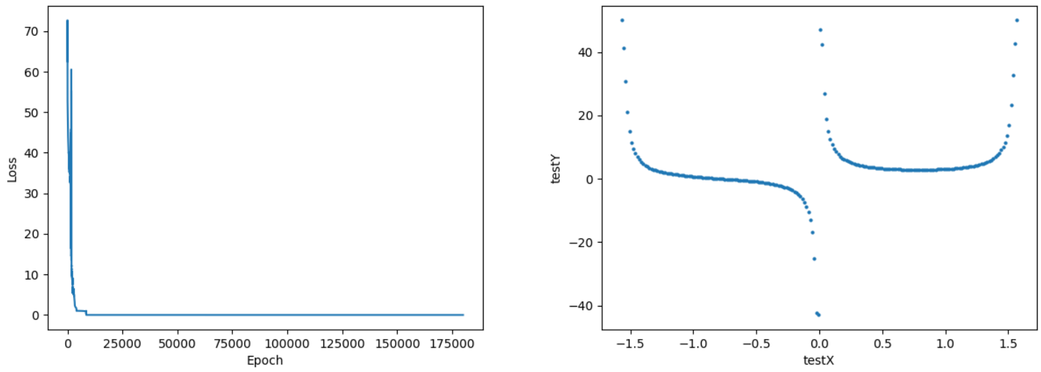

图 8 连续函数逼近实验结果

五、 实验探究

5.1 学习率对收敛趋势的影响

对于 XOR 逼近过程，在同样训练 15000 次时，我将学习率从 0.1 以步长 0.1 遍历到 0.8 以观察 Loss 的变化，得到的结果如下：

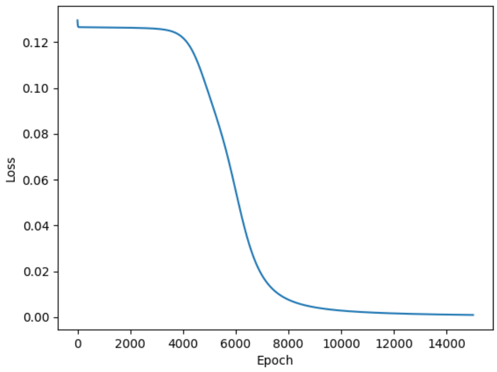

图 9 lr=0.1

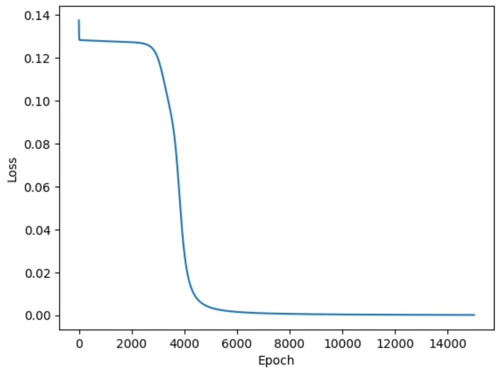

图 10 lr=0.2

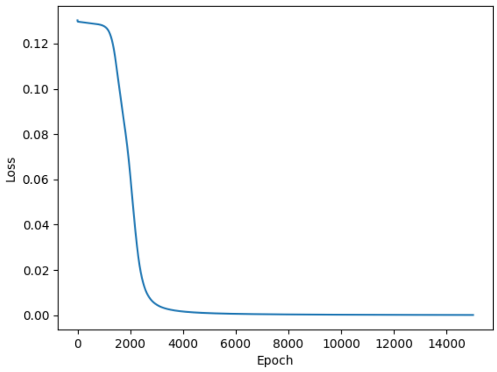

图 11 lr=0.3

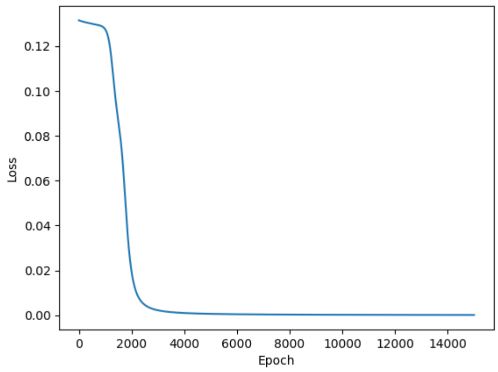

图 12 lr=0.4

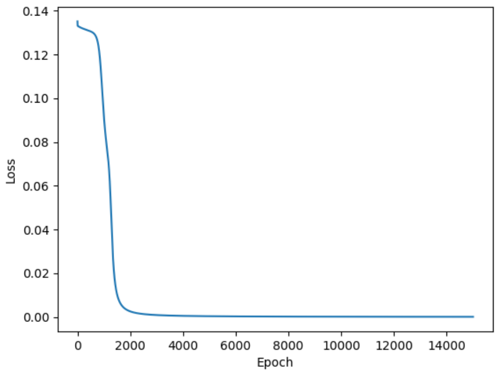

图 13 lr=0.5

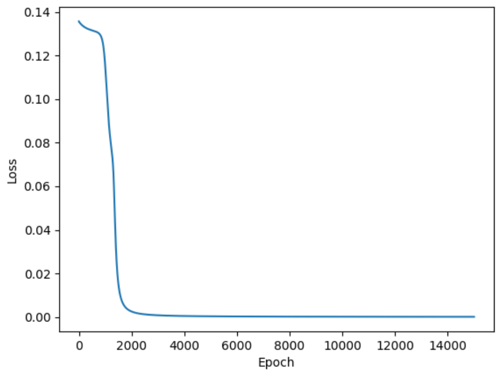

图 14 lr=0.6

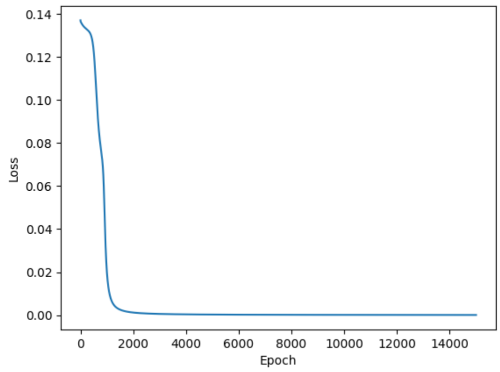

图 15 lr = 0.7

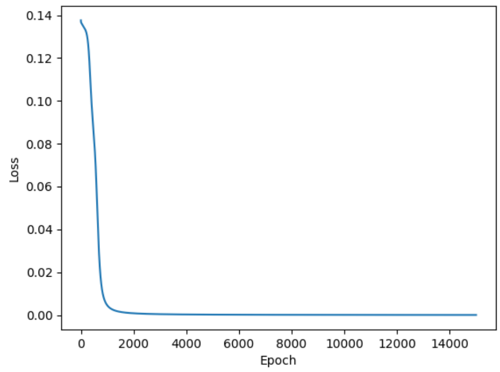

图 16 lr=0.8

通过上面八个图，我发现对于 XOR 逼近神经网络，在训练次数相同的情况下，随着 lr 的增加，Loss 收敛速度加快，由原来的 8000 次收敛到现在的 1000 次收敛，收敛速度大大加快，所以学习率的选取是神经网络训练中的一个重要的因素。

对于本实验 XOR 由于需要逼近的函数比较简单，所以得到的 lr 与 Loss 之间的关系也不是一般性的，一般而言：

如果**学习因子设置得过大**，神经网络的权重和偏置在每次迭代中会发生较大的变化，这可能导致算法不稳定，甚至无法收敛。在这种情况下，误差可能会震荡或发散，网络无法学习到有效的模型。

如果**学习因子设置得过小**，权重和偏置的更新会非常缓慢，这会导致算法收敛速度非常慢，甚至可能在有限的时间内无法收敛到一个满意的解。此时，需要更多的迭代次数来使网络收敛。

5.2 修改部分连接权值，看测试结果变化

同样基于 XOR 逼近神经网络进行，我修改训练后的权重矩阵 1 的部分权重值：

```
# 修改部分连接权值
Net.weight1[0][1] = 0
Net.weight1[0][2] = 0
```

再次预测 XOR 函数逼近得到的结果如下：

```
Predictions:
[[[0.]]

 [[1.]]

 [[1.]]

 [[1.]]]
```

发现预测结果发生了错误，代表神经网络有分布存储的特点，正是这些权重值体现了不同神经网络的特征。

六、 实验感想

首先是在实验原理中提到的，我最开始计算梯度的时候，采取的笨重方法：

我定义的$sigmoid$的导数是下面这样的：

```python
def dsigmoid(z):
    return sigmoid(z) * (1 - sigmoid(z))
```

按照我这个想法，在后向传播的时，例如计算L对b_2的偏导就得像公式 15 这样计算

$$\frac{\partial L}{\partial b_2}=\frac{\partial L}{\partial \hat{y}}dsigmoid\left(\begin{bmatrix}u_1 & u_2 & u_3 & u_4\end{bmatrix}\begin{bmatrix}h_{11}\\h_{21}\\h_{31}\\h_{41}\end{bmatrix}+b_2\right) \tag{15}$$

为此我最初的前向传播多返回了几个参数：

```python
def forward(self, X):
    H = np.dot(X.T, self.weight1) + self.bias1
    hidden_out = sigmoid(H)
    out_temp = np.dot(hidden_out, self.weight2) + self.bias2
    y_hat = sigmoid(out_temp)
    return y_hat, out_temp, hidden_out, H
```

最初后向传播也多传入了几个参数是这样的：

```python
def backword(self, x, y, y_hat, out, hidden_out, H):
    pianLy_hat = y_hat - y
    pianLb2 = pianLy_hat * dsigmoid(out)
    pianLW2 = np.dot(hidden_out.T, pianLb2)  # 隐藏层到输出层矩阵梯度
    pianLb1 = np.dot(pianLb2, self.weight2.T) * dsigmoid(H)
    pianLW1 = np.dot(x, pianLb1)

    self.weight1 = self.weight1 - self.lr * pianLW1
    self.weight2 = self.weight2 - self.lr * pianLW2
    self.bias1 = self.bias1 - self.lr * pianLb1
    self.bias2 = self.bias2 - self.lr * pianLb2
```

后来我发现括号内的东西可以直接用$\hat{y}$代替！(如我在实验原理中的计算方法)只需要改变一下 dsigmoid 的形式即可，大大简化了运算量，我对比了两种算法：

	训练时间(单位：秒)
未优化	3.28
优化后	2.54

节省了大约 22%的训练时间。

我的这个改法对很多人说可能是不值一提的，也许很多人从一开始求导的时候就是用的后者；不过我认为在这次实验后，我更加深刻的理解了 BP 神经网络的基本原理、前向、后向传播的过程，在自己手推一遍公式，写一遍代码后，这些看起来高大上的知识也不再那么陌生了。

七、 参考资料

[1]. https://zhuanlan.zhihu.com/p/485348369
[2]. http://t.csdn.cn/Fl4k3
[3]. https://zhuanlan.zhihu.com/p/353594833?utm_id=0

[4]. https://zhuanlan.zhihu.com/p/220447051