人工智能实验

Deep Learning Foundation

第 4 次实验

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 算法原理 **2**

1.1 神经网络 . . . 2
1.2 前向传播 . . . 2
1.3 反向传播 . . . 4
1.4 神经网络算法流程 . . . 6
1.5 激活函数 . . . 6

2 伪代码/流程图 **7**

3 关键代码 **8**

3.1 数据预处理 . . . 8
3.2 激活函数 . . . 8
3.3 前向传播/反向传播 . . . 9

4 实验过程与结果分析 **10**

4.1 划分数据集 . . . 10
4.2 三层神经网络的结果分析 . . . 10

5 实验优化与结果分析 **13**

5.1 数据预处理 . . . 13
5.2 不同的激活函数 . . . 13
5.3 多层深度神经网络 . . . 15
5.4 mini-batch . . . 16

6 思考题 **17**

6.1 其他激活函数的优缺点 . . . 17
6.2 实现在传递过程中不激活所有节点的方法 . . . 18
6.3 梯度消失和梯度爆炸的概念和解决方法 . . . 18

7 实验总结与感想 **18**

1 算法原理

1.1 神经网络

在机器学习中，神经网络是一种模仿动物神经网络行为特征，进行分布式并行信息处理的算法数学模型。这种网络依靠系统的复杂程度，通过调整内部大量节点之间相互连接的关系，从而达到处理信息的目的。

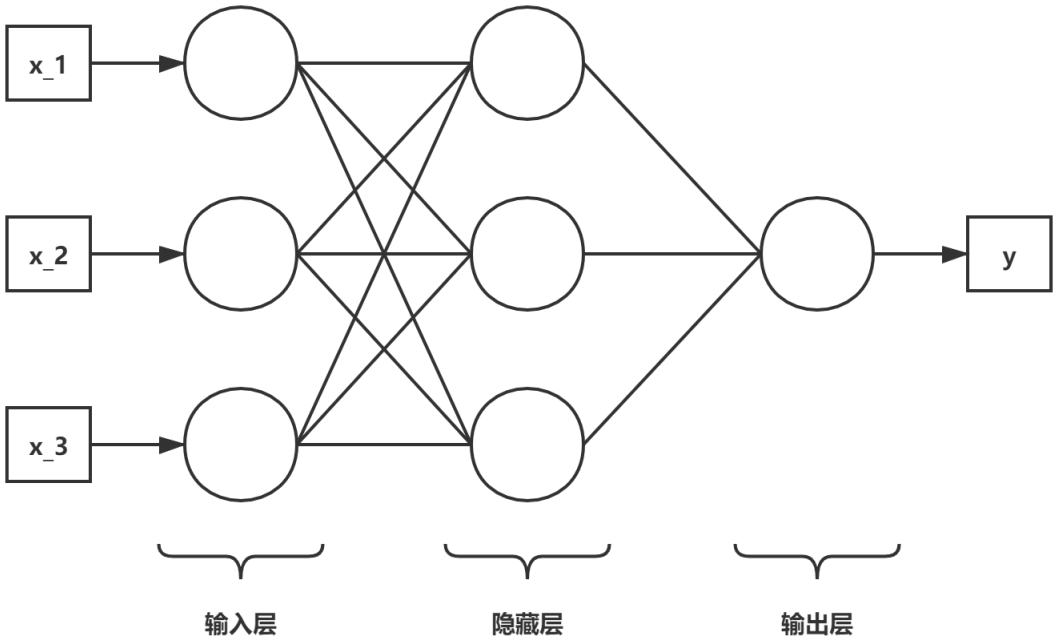

图 1: 一个三层神经网络

如图1所示，一个典型的神经网络分为三部分：输入层、隐藏层和输出层。输入层负责接受数据的输入，然后输入数据被传输到隐藏层。隐藏层的每个节点都有其对应的参数权重，在对数据进行一系列复杂的运算后，输出结果到输出层。输出层最终输出结果，这就是该神经网络预测的值。

对于一个神经网络，输入原始数据、经过从左到右的计算得到结果的过程就是前向传播；而当我们需要调整每个节点的参数权重、使得网络输出结果更符合实际值时，需要把误差从右到左传递给每个节点，这就是反向传播。

1.2 前向传播

我们首先介绍前向传播。假设我们有如图2所示的神三层经网络，每个节点的有关参数权重都已标记到图上。

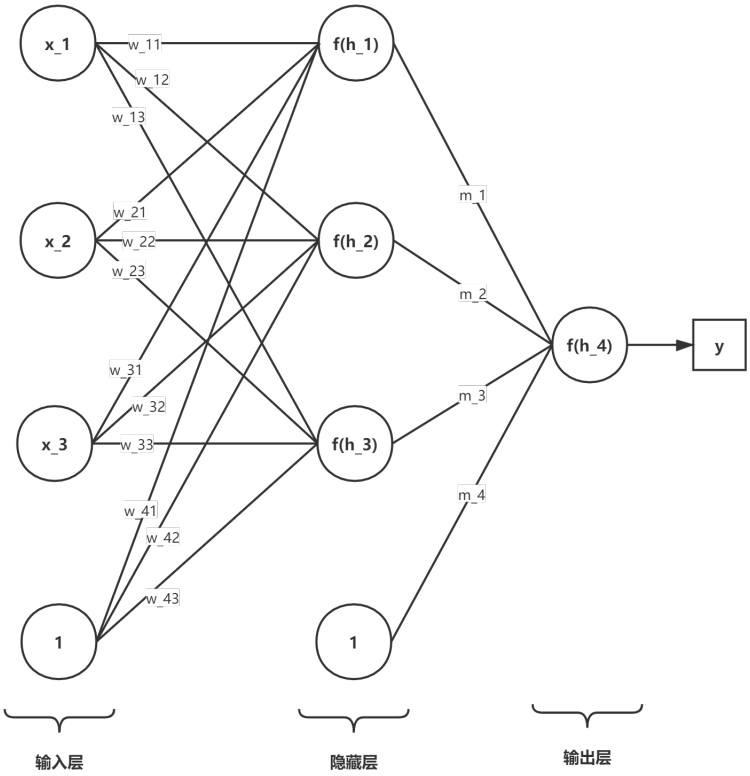

图 2: 三层神经网络

对于输入层，其不对输入数据进行任何处理。对于第 i 个输入层节点，其输入是样本数据的第 i 维数据 x_i，输出为 $y_{input(1)} = x_i$。

对于隐藏层，其每个节点的输入是上一层网络（输入层或上一层隐藏层）的输出。每个节点对所有输入数据加权求和后通过激活函数计算，输出最终结果。以图2中的隐藏层的第一个节点为例，其输入是输入层的输出 $\mathbf{X}_{hidden} = \begin{bmatrix} x_1 & x_2 & x_3 & 1 \end{bmatrix}$，而该节点的权重矩阵为 $\mathbf{W}_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \end{bmatrix}^T$，那么其首先进行加权求和：

$$h_1 = \mathbf{X}_{hidden}\mathbf{W}_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}$$

然后使用激活函数 $f()$ 处理，得到该节点的最终输出：

$$y_{hidden(1)} = f(h_1)$$

使用激活函数的原因十分简单，因为一层网络仅仅是对上一层的输出加权求和的话，那么无论网络深度多大，最终输出结果都是对原始数据的线性计算。因此我们需要激活函数将结果转换为非线性的数值。

在通过一系列隐藏层的计算后，最终的计算结果输入到输出层中。输出层与隐藏层的计算方法相同，根据任务类型确定激活函数和节点数量。如果任务是二分类，那么输出层只有一个节点，其对最后一层隐藏层的输出结果加权求和后使用 sigmoid 函数得到概率值。如果是多分类，那么输出层有多个节点，每个节点对应一种标签的概率值。每个节点对上一层的输出结果加权求和后，使用 softmax 函数得到对应标签的概率值。而本次实验是要实现回归，因此我们输出层只需要一个节点，激活函数使用 $f(x) = x$ 即可。

在如图2所示的神经网络中，输出层输入为 $\mathbf{X}_{output} = \begin{bmatrix} f(h_1) & f(h_2) & f(h_3) & 1 \end{bmatrix}$，权重矩阵为 $\mathbf{M} =$

$\begin{bmatrix} m_1 & m_{12} & m_{13} & m_{14} \end{bmatrix}^T$，我们最终的输出结果为

$$y_{output} = f(h_4) = h_4 = \mathbf{X}_{output}\mathbf{M} = m_1 f(h_1) + m_2 f(h_2) + m_3 f(h_3) + m_4$$

将上述过程扩展成矩阵计算。假设我们有大小为 $m \times n$ 的样本特征数据 $\mathbf{X}$，其中为 m 为样本数量，n 为特征维度数量（不包括偏置值 1）。

设输入层为第 0 层网络，隐藏层为第 1 层，输出层为最后一层。对于第 i 层网络 $(i > 0)$，设节点数量为 N_i（不包括偏置节点，$i = 0$ 时 $N_0 = n$），那么其权重矩阵 $\mathbf{W}_i$ 大小为 $(N_{i-1} + 1) \times N_i$；输入为矩阵 $\mathbf{X}_i$，则我们有

$$\mathbf{X}_i = \mathbf{Y}_{i-1} \tag{1}$$

其中 $\mathbf{X}_i$ 大小为 $m \times N_{i-1}$，我们为 $\mathbf{X}_i$ 的每行附上偏置 1，得到 $\widetilde{\mathbf{X}_i}$，大小为 $m \times (N_{i-1} + 1)$。此外，我们设该层的激活函数 $f_i()$ 为对矩阵中的每个元素分别计算，则有

$$\mathbf{H}_i = \widetilde{\mathbf{X}_i}\mathbf{W}_i \tag{2}$$

$$\mathbf{Y}_i = f(\widetilde{\mathbf{H}_i}) \tag{3}$$

其中 $\mathbf{H}_i$ 大小为 $m \times N_i$，而该层网络的最终输出 $\mathbf{Y}_i$ 是一个大小为 $m \times N_i$ 的矩阵。如果其为输出层，则该矩阵中每一行为对应样本的预测数据。

1.3 反向传播

接下来我们以图3回归神经网络中的某一隐藏层为例，说明反向传播。

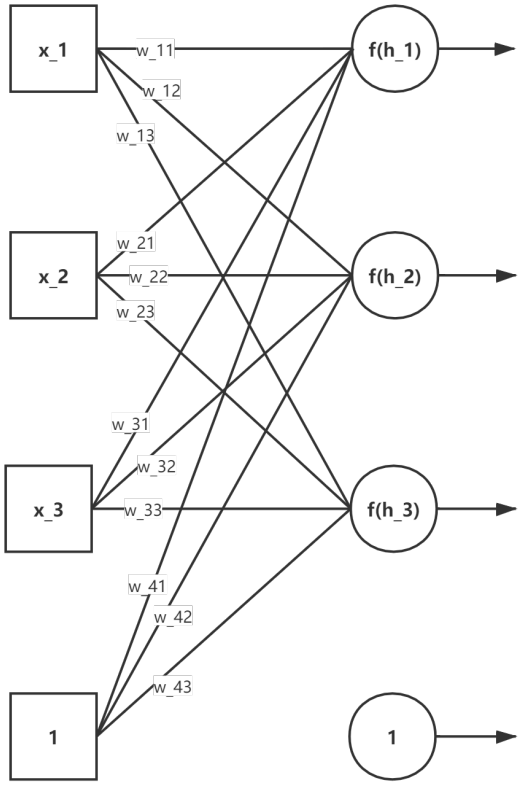

图 3: 某层神经网络

对于某个样本，在正向传播时，假设该层网络的输入为 $\mathbf{X}_{hidden} = \begin{bmatrix} x_1 & x_2 & x_3 & 1 \end{bmatrix}$，输出为 $\mathbf{y}_{hidden} = \begin{bmatrix} f(h_1) & f(h_2) & f(h_3) & 1 \end{bmatrix}$。我们设该网络最终输出值为 $\widehat{y}$，而真实值为 y，则误差为 $E = \frac{1}{2}(y - \widehat{y})^2$。当我们想求第 i 个输入在第 j 个节点的权重 w_{ij} 的梯度时，根据链式求导有：

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial f(h_j)} \frac{\partial f(h_j)}{\partial h_j} \frac{\partial h_j}{\partial w_{ij}} \tag{4}$$

其中，$\frac{\partial E}{\partial \widehat{y}} = -(y - \widehat{y})$，当前节点的误差梯度为 $\frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial f(h_j)}$。我们暂且跳过如何计算；$\frac{\partial f(h_j)}{\partial h_j}$ 是对激活函数进行求导，根据激活函数的不同，其值也不同；h_j 是线性函数，因此 $\frac{\partial h_j}{\partial w_{ij}} = x_i$。

在说明如何获得当前节点的误差梯度前，我们可以先讨论如何获得输出值为 x_i 的节点的误差梯度。我们有：

$$\begin{aligned} \frac{\partial E}{\partial x_i} &= \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial x_i} \\ &= \frac{\partial E}{\partial \widehat{y}} \sum_j \frac{\partial \widehat{y}}{\partial f(h_j)} \frac{\partial f(h_j)}{\partial h_j} \frac{\partial h_j}{\partial x_i} \\ &= \sum_j \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial f(h_j)} \frac{\partial f(h_j)}{\partial h_j} w_{ij} \end{aligned}$$

可见，每层节点的误差梯度都依赖于下一层节点的误差梯度。

我们使用所有样本，求得每个样本对应于一个权重的梯度值后，求得平均值，使用梯度下降法更新该权重，这样就视为完成一次迭代。

我们把上述过程转换成矩阵计算。假设对于第 i 层网络，节点数量为 N_i（不包括偏置节点）。我们已经有

(1) 输入 $\widetilde{\mathbf{X}_i}$，其大小为 $m \times (N_{i-1} + 1)$（包含偏置 1），m 为样本数量；

(2) 权重矩阵 $\mathbf{W}_i$，大小为 $(N_{i-1} + 1) \times N_i$；

(3) 输出 $\mathbf{Y}_i = f(\mathbf{H}_i)$，其中 $\mathbf{Y}_i$ 和 $\mathbf{H}_i$ 大小均为 $m \times N_i$；

(4) 该层网络的误差梯度 $d\mathbf{E}_i$，大小为 $m \times N_i$，每行是一个样本对应的该层网络节点误差梯度（偏置节点不需要误差梯度，故为 N_i 列而不是 $N_i + 1$ 列）；

那么根据式4，我们有：

$$d\mathbf{W}_i = \widetilde{\mathbf{X}_i}^T (f'(\mathbf{H}_i) * \mathbf{E}_i) \tag{5}$$

$f'(\mathbf{h}_i)$ 为对矩阵中每个元素进行求导，与 $\mathbf{E}_i$ 的元素一一对应相乘后得到矩阵大小为 $m \times N_i$。而 $d\mathbf{W}_i$ 大小为 $(N_{i-1} + 1) \times N_i$，正是权重矩阵的梯度。需要注意的是，这个梯度是所有样本梯度累加的和，需要除以 m 取得平均值。假设学习率为 η，则更新函数为

$$\mathbf{W}_i = \mathbf{W}_i - \frac{\eta}{m} d\mathbf{W}_i \tag{6}$$

此外，我们可以得到上一层网络的误差梯度：

$$\widetilde{d\mathbf{E}_{i-1}} = (f'(\mathbf{H}_i) * \mathbf{E}_i)\mathbf{W}_i^T \tag{7}$$

$\widetilde{d\mathbf{E}_{i-1}}$ 大小为 $m \times (N_{i-1} + 1)$。之所以不是 $m \times N_{i-1}$ 是因为最后一列是偏置节点的误差，需要去除，这样我们就得到了大小为 $m \times N_{i-1}$ 的误差梯度 $d\mathbf{E}_{i-1}$。

1.4 神经网络算法流程

(1) 对所有权重进行随机初始化；

(2) 前向传播：计算出每层网络的有关数据与最终输出结果；

(3) 计算误差 $\mathbf{E}$；

(4) 反向传播：计算每层网络的误差梯度 $d\mathbf{E}_i$ 和权重矩阵梯度 $d\mathbf{W}_i$；

(5) 更新所有权重矩阵：$\mathbf{W}_i = \mathbf{W}_i - \frac{\eta}{m} d\mathbf{W}_i$；

(6) 过程 (2) 至 (5) 重复迭代；

1.5 激活函数

在上述原理中，我们没有说明激活函数的具体形式。激活函数都是非线性函数，目前主要有以下几种激活函数：

(1) sigmoid 函数：$f(x) = \frac{1}{1+e^{-x}}$。$f'(x) = f(x)(1 - f(x))$；

(2) tanh 函数：$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$。$f'(x) = 1 - [f(x)]^2$；

(3) ReLU 函数：其原始公式是 $f(x) = max(0, x)$。如今还有许多该函数的修改版本，如 Leaky ReLU 函数：$f(x) = max(\alpha x, x)$，其中 $\alpha \in (0, 1)$。

2 伪代码/流程图

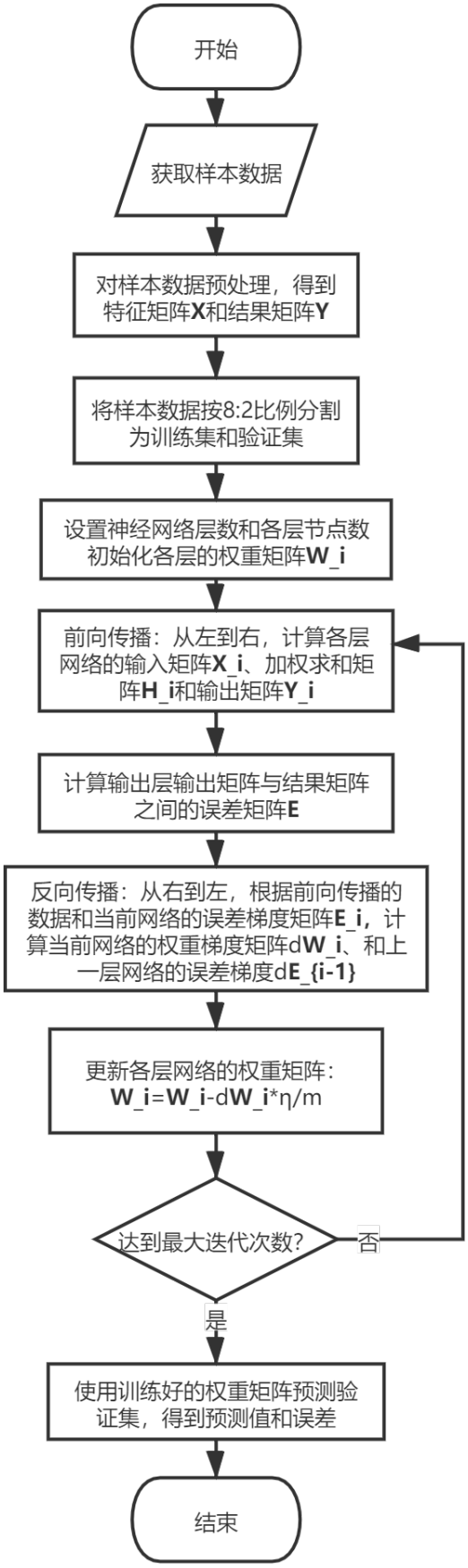

图 4: 神经网络算法流程图

3 关键代码

3.1 数据预处理

本次数据集的特征数据比较杂乱，我们将其读取进特征矩阵的同时需要对其进行预处理。我们对 season，mnth，hr，weekday 和 weathersit 进行了 one-hot 编码，对 temp、atemp、hum、windspeed 和结果数据标准化。代码如下：

```
with open('AIlab4/lab4_dataset/train.csv') as tf:
    FileText=tf.readlines()                 # 读取文本
samples_num=len(FileText)-1                 # 总样本数量
# 数据预处理
X=np.zeros((58,samples_num))                # 特征矩阵：特征数量*样本数量
Y=np.zeros((1,samples_num))                 # 结果矩阵：1*样本数量
for i in range(samples_num):                # 对每行样本文本处理
    temp=FileText[i+1].split(',')
    X[int(temp[2])-1,i]=1                   # season转换成one-hot编码
    X[4,i]=float(temp[3])                   # yr不做转换
    X[int(temp[4])+4,i]=1                   # mnth转换成one-hot编码
    X[int(temp[5])+17,i]=1                  # hr转换成one-hot编码
    X[41,i]=float(temp[6])                  # holiday不做转换
    X[int(temp[7])+42,i]=1                  # weekday转换成one-hot编码
    X[49,i]=float(temp[8])                  # workingday不做转换
    X[int(temp[9])+49,i]=1                  # weathersit转换成one-hot编码
    X[54,i]=float(temp[10])                 # temp不做转换
    X[55,i]=float(temp[11])                 # atemp不做转换
    X[56,i]=float(temp[12])                 # hum不做转换
    X[57,i]=float(temp[13])                 # windspeed不做转换
    Y[0,i]=float(temp[14])                  # 真实值直接读取
X=X.T                                       # 特征矩阵：样本数量*特征数量
Y=Y.T                                       # 结果数量：样本数量*1

# 对几个数值大的特征和结果标准化、进行缩放
X_mean=np.mean(X,axis=0)                    # 特征平均值
X_std=np.std(X,axis=0)                      # 特征方差
for i in range(54):                         # 有些特征不需要标准化
    X_mean[i]=0
    X_std[i]=1
Y_mean=np.mean(Y,axis=0)                    # 结果平均值
Y_std=np.std(Y,axis=0)                      # 结果方差
X=(X-X_mean)/X_std                          # 特征标准化
Y=(Y-Y_mean)/Y_std                          # 结果标准化
```

3.2 激活函数

本次实验我们涉及到三种激活函数：sigmoid、tanh 和 ReLU。代码如下所示：

```
def sigmoid(x):
```

```
    """
    sigmoid激活函数
    """
    if x>0:
        return 1/(1+np.exp(-x))
    else:
        return np.exp(x)/(1+np.exp(x))
```

```
def tanh(x):
    """
    tanh激活函数
    """
    return (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
```

```
def ReLU(x,a):
    """
    ReLU函数
    """
    return max(a*x,x)
```

3.3 前向传播/反向传播

以三层神经网络为例，以下是训练函数代码，涉及到了神经网络的前向传播和反向传播：

```
def train_network(X, Y, input_nodes, hidden_nodes, output_nodes, learning_rate, iterations
    ):
    """
    训练网络
    """
    np.random.seed(0)
    # 初始化权重矩阵
    W_input_hidden=np.random.normal(loc=0,scale=1,size=(input_nodes+1,hidden_nodes))
    W_hidden_output=np.random.normal(loc=0,scale=1,size=(hidden_nodes+1,output_nodes))
    m=np.size(X,axis=0)
    for iter in range(iterations):
    # 前向传播
        # 输入层
        input_X=X
        input_H=input_X
        input_F=input_H
        # 隐藏层
        addMatrix=np.ones((np.size(input_F,axis=0), 1))
        hidden_X=np.column_stack((input_F,addMatrix))
        hidden_H=np.dot(hidden_X, W_input_hidden)
        hidden_F=np.zeros(hidden_H.shape)
        for i in range(np.size(hidden_F,axis=0)):
            for j in range(np.size(hidden_F,axis=1)):
```

```
                hidden_F[i,j]=sigmoid(hidden_H[i,j])
        # 输出层
        addMatrix=np.ones((np.size(hidden_F,axis=0), 1))
        output_X=np.column_stack((hidden_F,addMatrix))
        output_H=np.dot(output_X, W_hidden_output)
        output_Y=output_H
    # 反向传播
        error=output_Y-Y
        output_error=error
        # 输出层
        dW_hidden_output=np.dot(output_X.T, output_error)
        hidden_error=np.dot(output_error, W_hidden_output.T)
        hidden_error=hidden_error[:,:-1]
        # 隐藏层
        dW_input_hidden=np.dot(hidden_X.T, hidden_F*(1-hidden_F)*hidden_error)
    # 更新权重
        W_input_hidden-=dW_input_hidden*learning_rate/m
        W_hidden_output-=dW_hidden_output*learning_rate/m
    # 返回训练好的权重矩阵
    return W_input_hidden, W_hidden_output
```

其中输入参数 *input_nodes*、*hidden_nodes* 和 *output_nodes* 分别是输入层、隐藏层和输出层的节点数量（均不包括偏置节点）。

4 实验过程与结果分析

4.1 划分数据集

本次实验的数据集共有 8619 个样本，数量较多，故我没有按照以前的 7:3 的比例划分训练集和验证集，而是使用 8:2 的比例，在保证验证集有适量数据检验的同时扩大训练集规模。

4.2 三层神经网络的结果分析

首先我实现三层神经网络，输入层节点数为 58（未计偏置节点），隐藏层节点数为 50（计入偏置节点），输出层节点数为 1。隐藏层激活函数是 sigmoid，输出层不采用激活函数、直接将隐藏层节点的输出值加权求和后输出。设置迭代次数为 10000，每隔 20 次迭代计算在训练集和验证集上的 loss。采用不同的学习率，最终结果如图5和6所示。

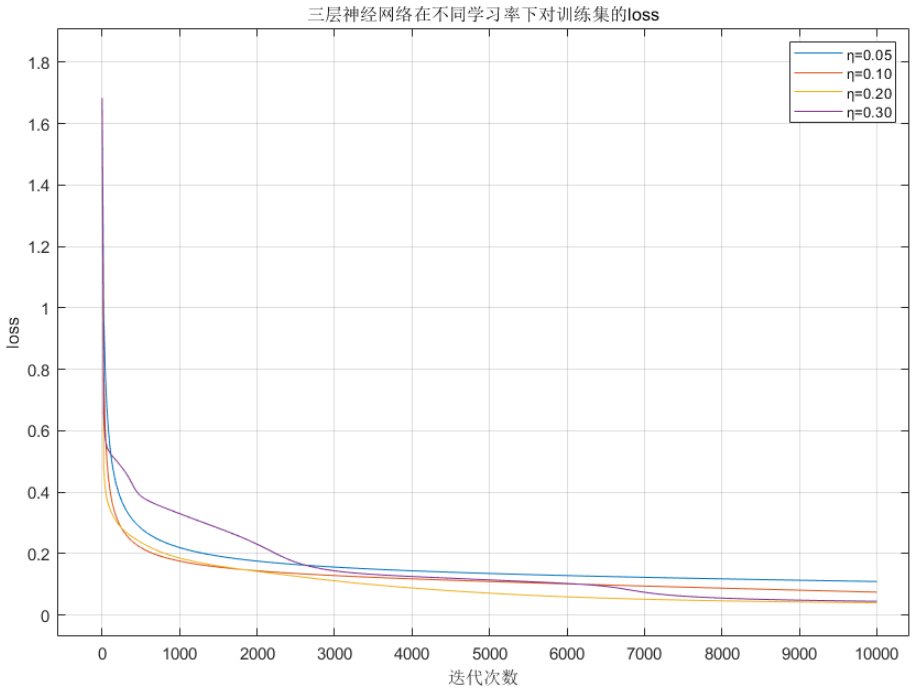

图 5: 不同学习率下的训练集 loss

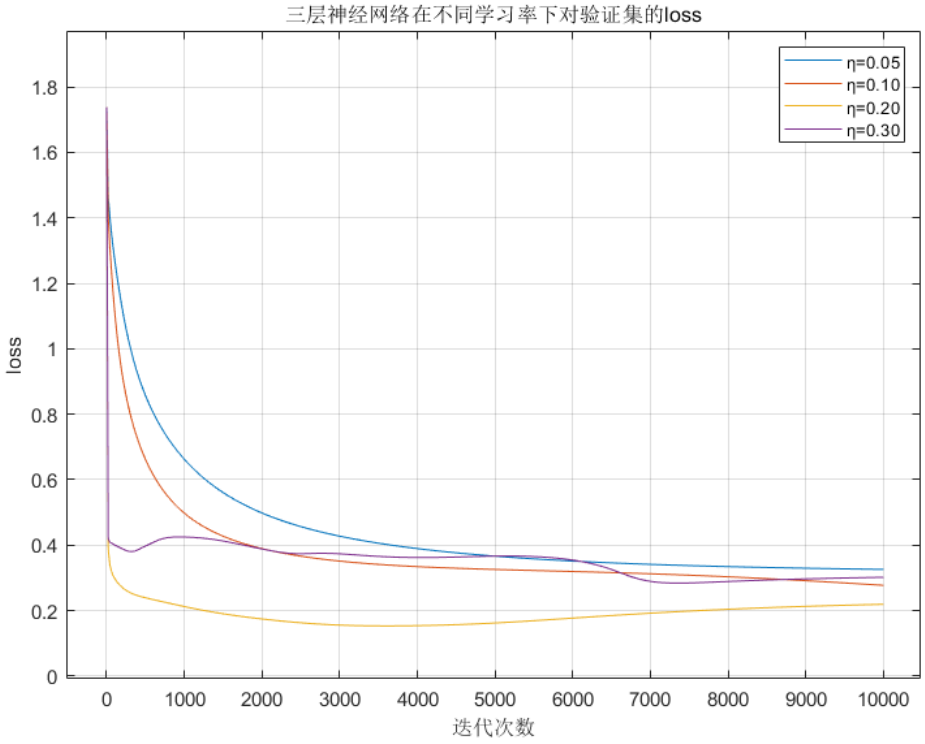

图 6: 不同学习率下的验证集 loss

可见，在训练集上模型的 loss 一直下降，不过因为学习率的不同而导致下降速度有所不同。在验证集上，当学习率为 0.2 和 0.3 时在迭代后期都出现了过拟合现象，loss 上升。表现的最好为学习率 0.2、迭代 3600 次后的网络，loss 为 0.15306。该模型对前五个验证集预测样本如表1所示。

序号	真实数据	预测数据
6896	107	111
6897	69	62
6898	26	36
6899	15	4
6900	4	-39

表 1: 真实数据与预测数据的对比

可见，第 6896 和 6897 个样本预测比较准确，而第 6898 和 6899 个预测出现一定偏移。第 6900 个样本预测出现负数情况。这说明该网络的预测还不够准确。

此外，我也设置了不同隐藏层节点数量的网络。分别设置隐藏层节点数量为 10、25 和 50（均计入偏置节点）；设置学习率为 0.2，迭代 10000 次，结果如图7和8所示。

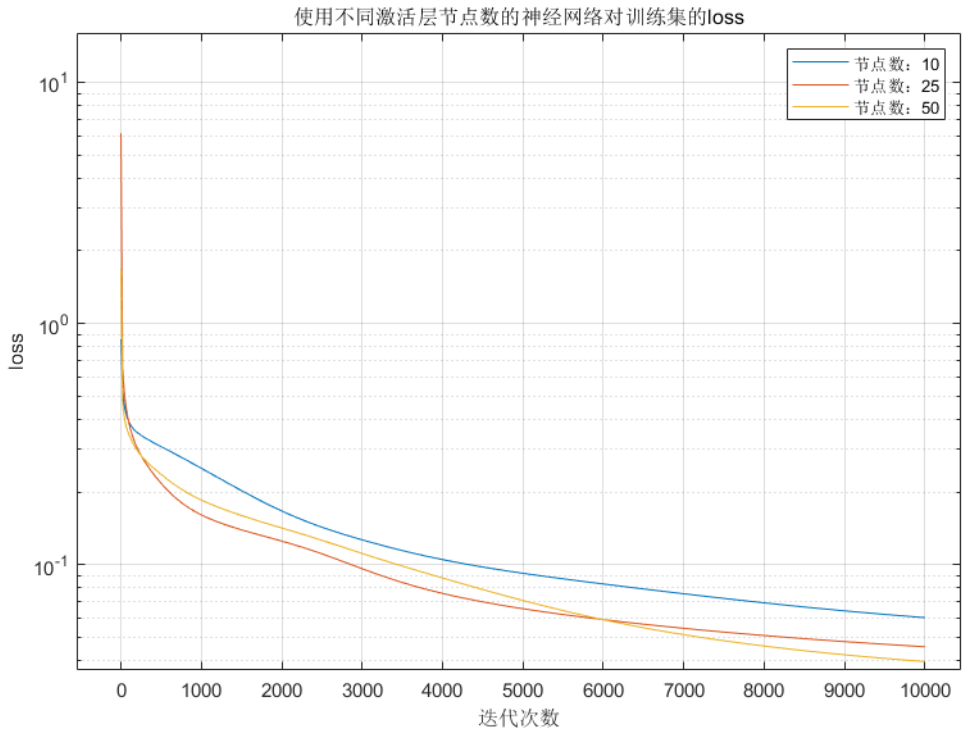

图 7: 不同隐藏层节点数量下的训练集 loss

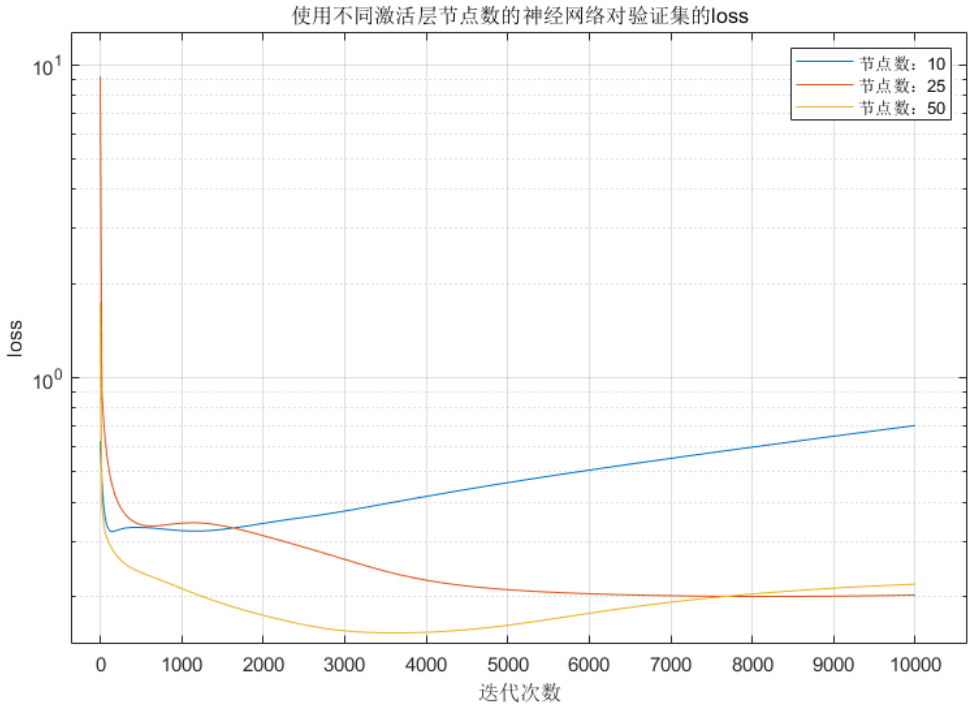

图 8: 不同隐藏层节点数量下的验证集 loss

可见，当节点数量为 10 时，网络在验证集上很快就出现了过拟合现象，而节点数量为 25 时出现了一定波动。在验证集上表现最好的是节点数量为 50 的网络，迭代 3600 次后 loss 为 0.15306。

5 实验优化与结果分析

5.1 数据预处理

本次实验，我在一开始就是用了数据预处理，方法分别是 one-hot 编码和数据标准化。

对于 season，mnth，hr，weekday 和 weathersit 等特征，其原始数据使用不同大小的值表示不同的情况。然而实际上，各个情况相互独立，使用不同大小的值表示不同情况将会导致模型不够准确。因此我们使用 one-hot 编码转换数据。以 weathersit 为例，其原始数值可能有 1、2、3 和 4，我们分别转换为 $[1,0,0,0]$，$[0,1,0,0]\cdots$ one-hot 具体实现代码见3.1。

此外，观察数据我们可以发现，除了 temp、atemp、hum、windspeed 和样本结果，其他特征值大多在 $[0,1]$ 范围内。这就需要我们将这几个数值波动范围较大的特征和样本结果进行标准化。假设某个特征/结果的平均值为 $\bar{x}$，方差为 $\sigma(x)$，则第 i 个数值标准化后为

$$x_i' = \frac{x_i - \overline{x}}{\sigma(x)} \tag{8}$$

同样，具体实现代码见3.1。

5.2 不同的激活函数

本次实验，在原先的三层神经网络上，我应用不同激活函数。输入层节点数为 58（未计入偏置节点），隐藏层节点数为 50（计入偏置节点），输出层节点数为 1，学习率为 0.05，结果如图9和10所示。

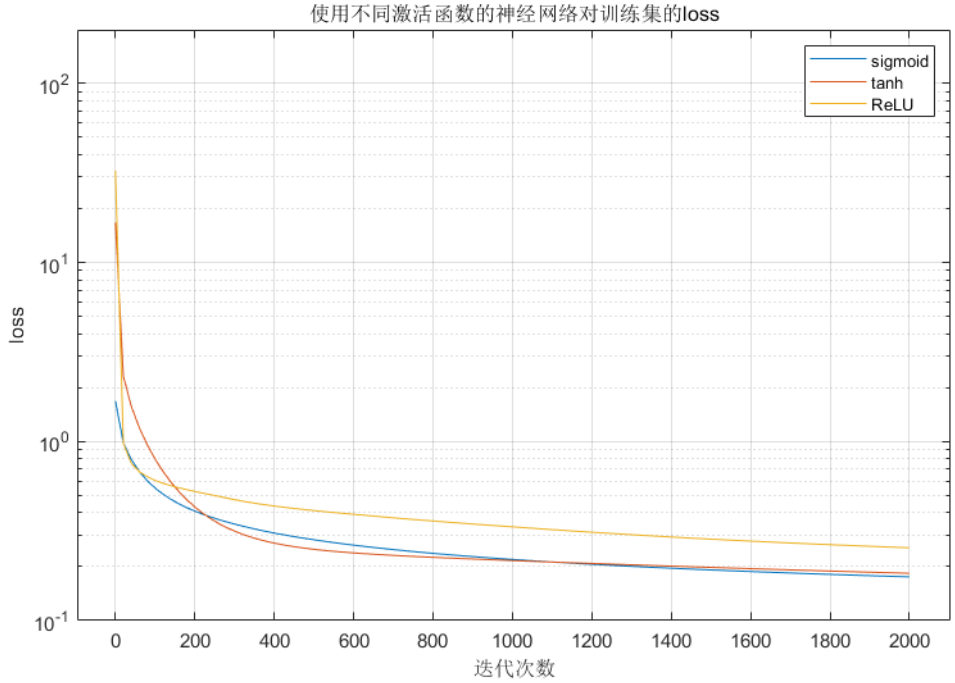

图 9: 不同激活函数下的训练集 loss

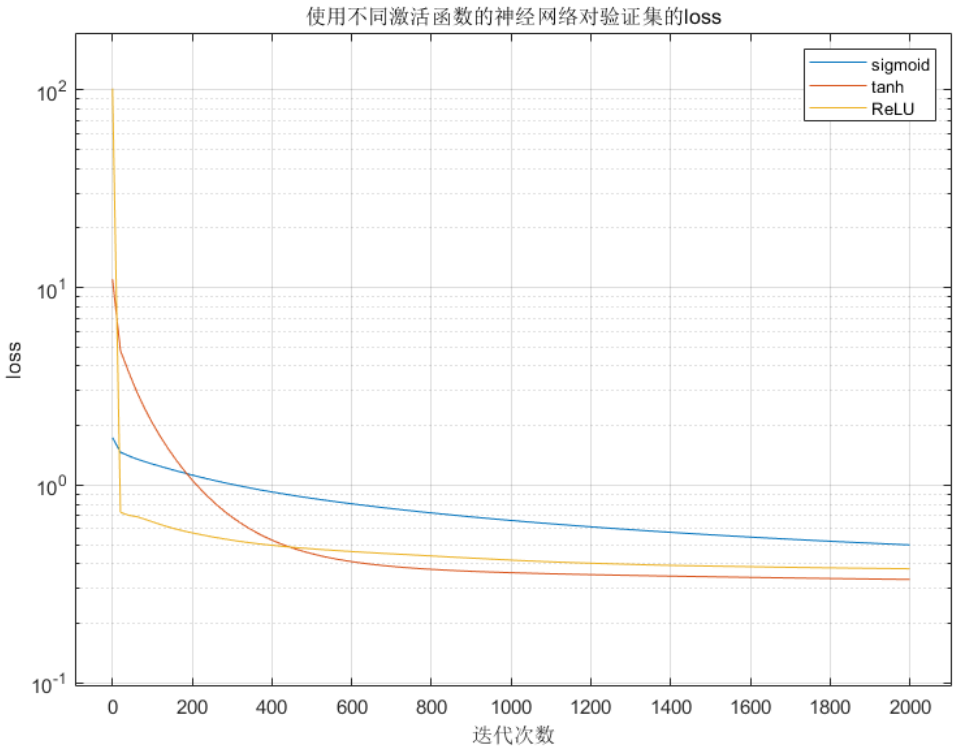

图 10: 不同激活函数下的验证集 loss

可见，在验证集上，这个网络中表现最好的是 tanh 函数，迭代 2000 次后 loss 为 0.33288；ReLU 表现次之，sigmoid 最差。

5.3 多层深度神经网络

除了原有的三层神经网络，我也尝试了四层和五层神经网络。其中四层神经网络的隐藏层第一、二层节点数均为 50（计入偏置节点）；五层神经网络的隐藏层第一层节点数为 50，第二层节点数为 25，第三层为 10。学习率为 0.2，激活函数为 sigmoid，结果如图11和12所示。

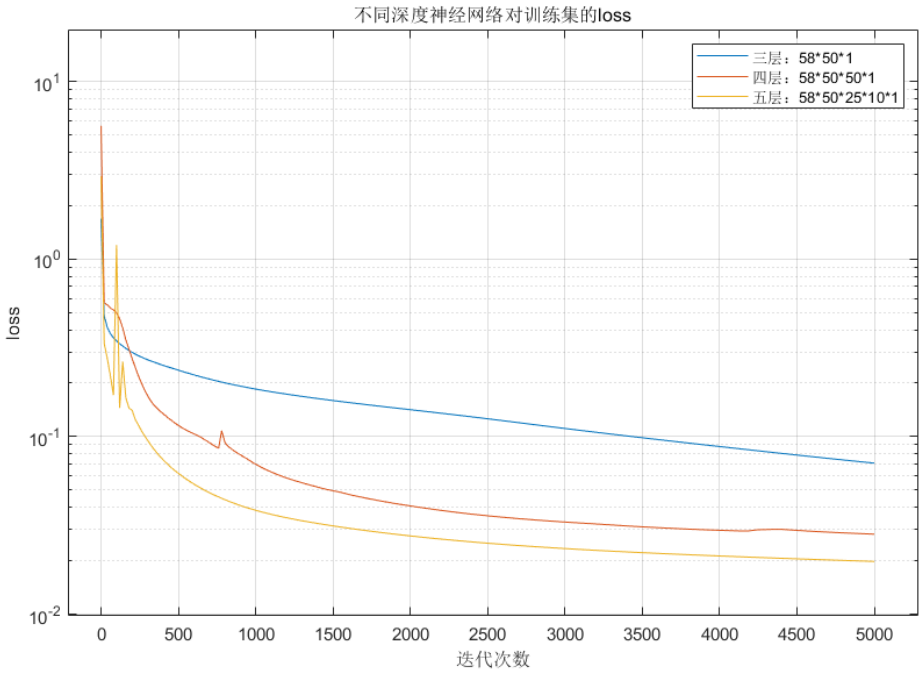

图 11: 不同深度下的训练集 loss

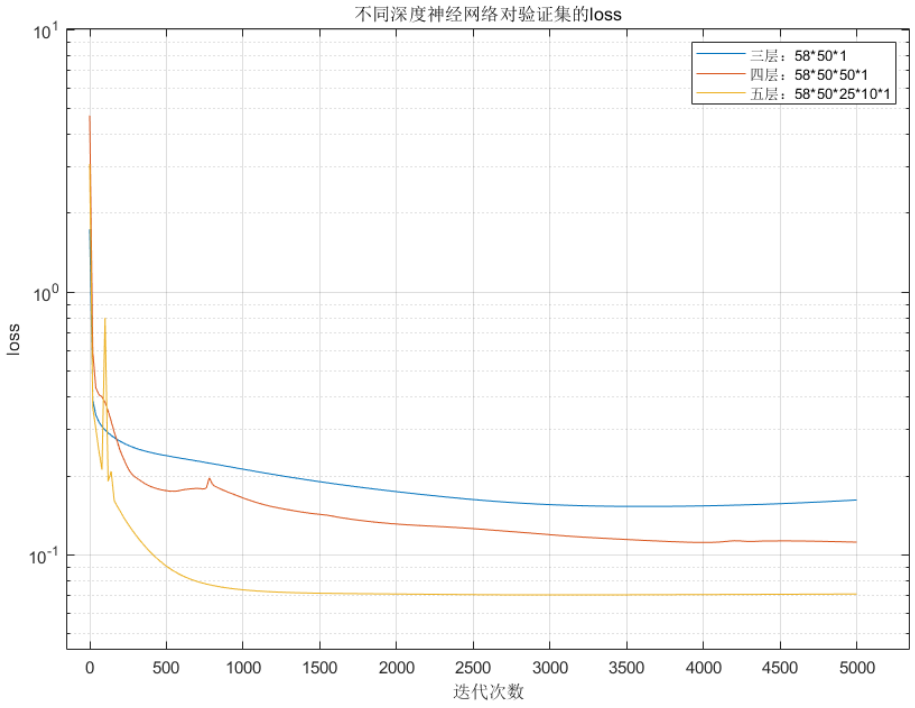

图 12: 不同深度下的验证集 loss

可见，三个神经网络在后期都出现了过拟合现象。在验证集上表现最好的是五层神经网络，迭代 3040

次后 loss 为 0.70439。

5.4 mini-batch

mini-batch 相比于 batch，主要内容是每次迭代只选取训练集的一部分、而不是加载全部训练集。我在原三层神经网络的基础上设置不同的 batch 大小，隐藏层节点数均为 50，学习率 0.2，激活函数为 sigmoid，结果如图13和14所示。

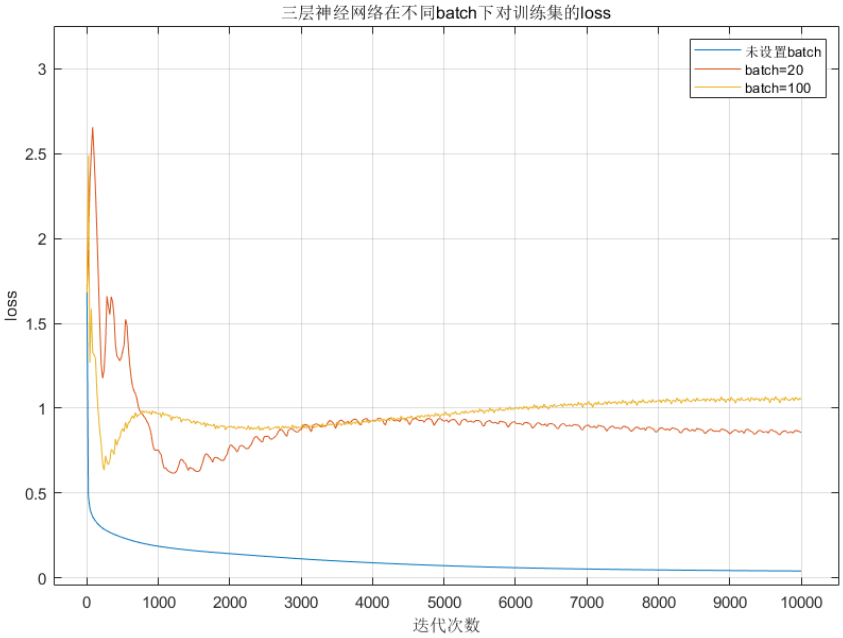

图 13: 不同 batch 下的训练集 loss

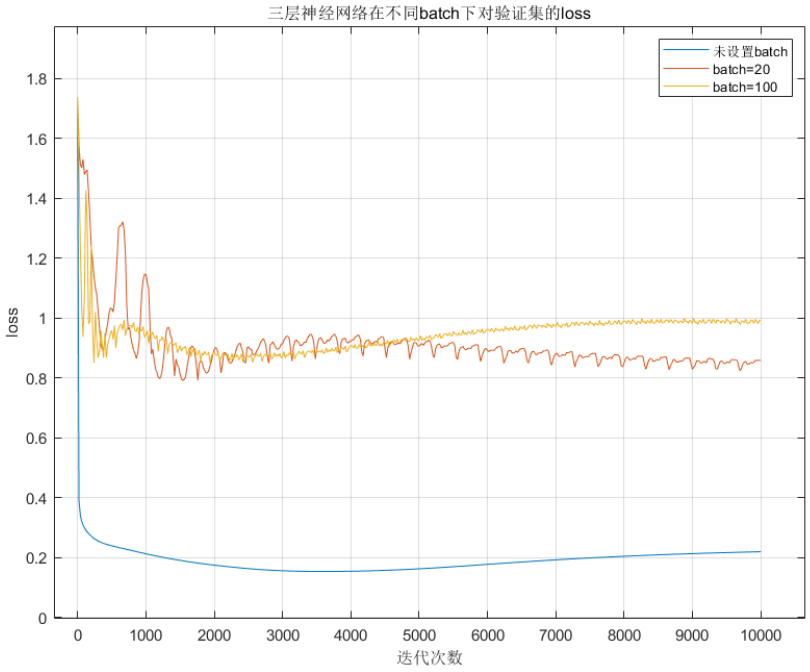

图 14: 不同 batch 下的验证集 loss

可见，batch 为 20 和 100 时 loss 都会出现波动。mini-batch 可以提高迭代速度，但在我的网络中，表现并不比不使用 mini-batch 的网络好。

6 思考题

6.1 其他激活函数的优缺点

常用的激活函数有 3 种：sigmoid 函数、tanh 函数和 ReLU 函数。其形式详见1.5。

sigmoid 函数是传统神经网络使用频率最高的函数。其主要优点是：

(1) 简单清晰，易于理解；

(2) 函数连续平滑，便于求导；

(3) 输出映射在 $(0,1)$ 之间，范围有限、优化稳定；

其主要缺点是：

(1) 幂运算带来了极高的计算成本；

(2) 因为导数总是小于 1，且 x 值过大或过小时导数趋近于 0，导致梯度消失问题；

(3) 函数输出值恒大于 0，不是以 0 为中心，导致收敛速度下降；

tanh 函数在一定程度上弥补了 sigmoid 函数的缺点，性能更好，主要优点如下：

(1) 解决了 sigmoid 函数的输出值恒大于 0 的问题；

(2) 导数范围有所增大，缓解了梯度消失问题，收敛速度更快；

其主要缺点是：

(1) 使用了更多的幂运算；

(2) 梯度消失问题依然存在；

后来，人们又提出了新的激活函数 ReLU：$f(x) = max(0, x)$，其主要优点如下：

(1) $x > 0$ 时导数恒定，不存在梯度消失问题；

(2) 不需要幂运算，计算速度快，复杂度低；

(3) 收敛速度快；

但其也存在着缺点：

(1) $x < 0$ 时输出恒为 0，可能导致相应参数永远不能更新，即所谓的 Dead ReLU 现象；

(2) 输出值恒大于 0；

(3) 对参数的初始化和学习率十分敏感；

为了改进 ReLU 的 Dead ReLU 问题，后来的学者对 ReLU 进行了许多改进，其中一种就是 Leaky ReLU：$f(x) = max(\alpha x, x)$。其具有 ReLU 的所有优点，同时改善了 ReLU 的部分缺陷。

6.2 实现在传递过程中不激活所有节点的方法

在训练神经网络时，让部分节点不工作/激活的方法称为 Dropout。该方法会按照一定概率将一些节点从网络中暂时丢弃，主要目的是防止深度神经网络过拟合。我们可以设置概率阈值 p，当进行新一轮迭代时，每个节点都生成一个大小在 0 与 1 之间的随机数，当随机数大于 p 时该节点正常参与迭代、输出计算结果，否则该节点的输出为 0。这样，我们就实现了不激活所有节点。

6.3 梯度消失和梯度爆炸的概念和解决方法

梯度消失和梯度爆炸的出现都与反向传播算法有关。由1.5的式5，6和7可知，在更新权重时权重梯度不仅与权重和输入值有关，还与后面网络的激活函数的导数有关，一个权重矩阵的更新梯度是后面网络中各层网络的激活函数的导数乘积。当激活函数的导数值一直小于 1 时，会导致更新梯度指数级衰减，反向传播到靠近输入层的网络时更新缓慢，出现梯度消失的现象；当激活函数的导数值一直大于 1 时，会导致更新梯度指数级增加，反向传播到靠近输入层的网络时波动剧烈，出现梯度爆炸的现象。

解决梯度消失和梯度爆炸的方法有以下几种：

(1) 梯度剪切，这方法主要用于解决梯度爆炸。其思想是设置一个阈值，当梯度超过这个阈值时，就将梯度强制限制在这个范围内，防止出现爆炸；

(2) 选择 LeRU 等不会导致梯度消失、爆炸的激活函数；

(3) 先对网络进行预训练，然后进行微调；

(4) 使用 BN 算法，过对每一层的输出规范为均值和方差一致的方法，消除了权重参数放大缩小带来的影响，进而解决梯度消失和爆炸的问题；

此外，还有很多其他方法可以缓解、解决梯度消失和爆炸，在此不再展开。

7 实验总结与感想

本次实验可以说是“炼丹”的前奏，我们开始接触并尝试简单的神经网络模型。在本次实验中，个人感觉最困难的部分就是反向传播算法。虽然理解、计算反向传播的梯度不是特别复杂，但如何将其表达成矩阵的计算花费了我好几天的时间去思考，所幸的是最后总算推导出了矩阵计算的格式（如果推错就尴尬了……）此外，本次实验中我也尝试进行数据的预处理、不同激活函数的使用和不同深度的网络模型，还有 mini-batch 的应用，也得到了不错的效果。

在此感谢老师和助教的帮助，希望我能顺利完成下一次实验。