中山大学数据科学与计算机院本科生实验报告

(2020 秋季学期)

课程名称：人工智能 **任课老师：**饶洋辉

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.10.20

实验四 BPNN

1 实验原理

1.1 神经元模型

M-P 神经元模型抽象人类神经网络的神经元模型，其结构如图 1。在模型中，神经元接收来自 n 个其他神经元传递过来的输入信号（或原始输入信号），这些输入信号通过带权重 w_i 的连接进行传递，神经元接收到的总输入值将与神经元的阈值 θ 进行比较，然后通过“激活函数”处理以产生神经元的输出 $y = f(\sum_{i=1}^{n} w_i x_i - \theta)$，。

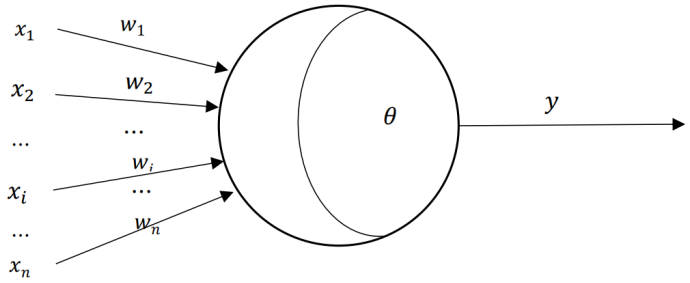


图 1: M-P 神经元模型

1.2 BPNN

BPNN 是由神经元模型组成的神经网络，其输出结果采用前向传播，误差采用反向传播方式进行。输入从输入层输入，经隐藏层处理以后，传向输出层。如果输出层的实际输出和期望输出不符合，就进入误差的反向传播阶段。误差反向传播是将输出误差以某种形式通过隐藏层向输入层反向传播，并将误差分摊给各层的所有单元，从而获得各层单元的误差信

号，这个误差信号就作为修正个单元权值的依据。知道输出的误差满足一定条件或者迭代次数达到一定次数。

BPNN 层与层之间为全连接，同一层之间没有连接。结构模型如图 2，给定训练集 $D = (x_1, y_1), \ldots, (x_m, y_m), x_i \in R^N, y_i \in R^l$，图中神经网络拥有 N 个输入神经元、M 个隐藏神经元、l 个输出神经元，输入层第 j 个神经元与隐藏层第 i 个神经元之间的连接权为 w_{ij}, 隐藏层与输出神经元之间的连接权重为 W_{ik}，隐藏层第 j 个神经元的偏置为 b_i，输出层第 k 个神经元的偏置为 B_k。

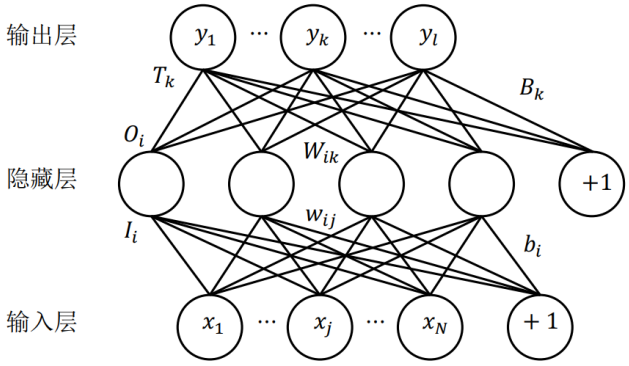


图 2: BPNN

则隐藏层的输入 $I_i = \sum_{j=1}^{N} w_{ij}x_j + b_i$, 输出 $O_i = f(I_i)$,输出层的输入 $T_k = \sum_{i=1}^{M} W_i O_i + B_k$。

1.2.1 前向传递

根据神经元模型，每个神经元在输入后，经过激活函数得到输出，而输出结果作为下一层的输入，层之间的神经元没有联系，因此前向传递为单向的。一般采用的激活函数为 **Sigmoid** 函数。

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

对于每一个训练样例 (x_f, y_f)，可以得到神经网络的输出 $\tilde{\mathbf{y}}_f = (\tilde{y}_1^f, \tilde{y}_2^f, \tilde{y}_3^f, \ldots, \tilde{y}_l^f)$, 即

$$\tilde{y}_k^f = f(\sum_{i=1}^{M} O_i W_{ik} + B_k),$$

则网络在 (x_f, y_f) 上的均方误差为

$$E_f = \frac{1}{2} \sum_{k=1}^{l} (\tilde{y}_k^f - y_k^f)^2$$

1.2.2 反向误差传播

由前向传递得到了每一个训练样例 (x_f, y_f) 的误差，图 2中有 W_{ik}, w_{ij}, b_i, B_k 一共 $(N*M+M+M*L+L)$ 个参数需要进行更新。在反向误差传播过程中，每一轮迭代采用感知机的学习规则基于梯度下降法对参数进行更新估计，以目标的负梯度对参数进行调整，缩小误差。以 w_ij 的更新为例，w_{ij} 首先影响隐藏层的输入 I_i，再通过输出层的输入 T_k 对误差 E_k 进行影响，有

$$\frac{\partial E_k}{\partial w_{ij}} = \frac{\partial E_k}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial T_k}\frac{\partial T_k}{\partial O_i}\frac{\partial O_i}{\partial I_i}\frac{\partial I_i}{\partial w_{ij}},$$

即对于误差 E_k, 给定学习率，可通过

$$\triangle w_{ij} = -\eta\frac{\partial E_k}{\partial w_{ij}}, w_{ij} \leftarrow w_{ij} + \triangle w_{ij}$$

对 w_{ij} 进行更新，类似可得

$$\triangle W_{ik} = -\eta\frac{\partial E_k}{\partial W_{ik}}, W_{ik} \leftarrow W_{ik} + \triangle W_{ik},$$

$$\triangle B_k = -\eta\frac{\partial E_k}{\partial B_k}, B_k \leftarrow B_k + \triangle B_k,$$

$$\triangle b_i = -\eta\frac{\partial E_k}{\partial b_i}, b_i \leftarrow b_i + \triangle b_i.$$

反向误差传播通过计算输出层与期望值之间的误差来调整网络参数，基于链式法则，找到这个参数影响最终结果的关系，采用梯度下降法减少梯度。

1.2.3 BP 算法

对每个训练样例，BP 算法先将输入实例提供给输入层神经元，然后逐层将信号前传，直到产生输出层的结果；然后计算输出层的误差，再将误差逆向传播到隐藏层神经元，最后根据隐藏层神经元的误差来对连接权和偏置进行调整。循坏迭代该过程，直到满足某些停止条件（如：达最高迭代次数，误差小于一定值），伪代码如下

Algorithm 1 BP 算法

输入: 训练集 $D = \{(\mathbf{x}_f, \mathbf{y}_f)\}_{f=1}^{m}$; 学习率 η

输出: 连接权值和偏置确定的多层前馈神经网络

1: **function** $BP(D, \eta)$
2: 在 $(0, 1)$ 范围内随机初始化网络中的所有连接权值和偏置
3: **repeat**
4: **for all**$(\mathbf{x}_f, \mathbf{y}_f) \in D$ **do**
5: 计算当前样本每个输出神经元的输出 $\hat{\mathbf{y}}_f = f(\sum_{i=1}^{M} O_i W_{ik} + B_k)$

6: 计算误差

$$E_f = \frac{1}{2}\sum_{k=1}^{l}(\tilde{y}_k^f - y_k^f)^2$$

7: 计算输出神经元的权重与偏置梯度

$$\frac{\partial E_k}{\partial W_{ik}}, \frac{\partial E_k}{\partial B_k}$$

8: 计算隐藏层神经元的权重与偏置梯度

$$\frac{\partial E_k}{\partial w_{ij}}, \frac{\partial E_k}{\partial b_i}$$

9: 更新权值

$$w_{ij} = w_{ij} - \eta\frac{\partial E_k}{\partial w_{ij}}$$

$$W_{ik} = W_{ik} - \eta\frac{\partial E_k}{\partial W_{ik}}$$

10: 更新偏置

$$B_k = B_k - \eta\frac{\partial E_k}{\partial B_k}$$

$$b_i = b_i - \eta\frac{\partial E_k}{\partial b_i}$$

11: **end for**

12: **until** 达到停止条件

13: **end function**

2 实验过程

2.1 梯度推导

本次实验使用 **BPNN** 进行回归任务，实现的神经网络如图 3，对于每个样例输入 $(x_1, \ldots, x_j, \ldots, x_N)$, 只有一个输出神经元输出最终回归结果，$R$ 为该输出神经元的输出的结果, 样例实际值为 Y, 隐藏层神经元采用 **Sigmoid** 函数作为激活函数，则该神经网络满足

$$R = \sum_{i=1}^{M} W_i O_i + B,$$

$$E_k = \frac{1}{2}(R - Y)^2,$$

$$I_i = \sum_{j=1}^{N} w_{ij} x_j + b_i,$$

$$O_i = \frac{1}{1 + e^{-I_i}}.$$

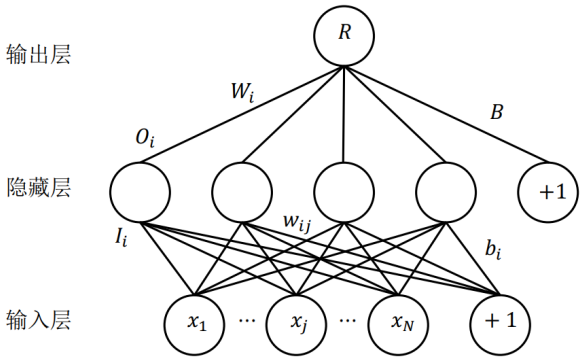


图 3: 实验实现的神经网络

根据链式法则，基于误差对需要更新的权重与偏置进行求导，得到结果为

$$\frac{\partial E_k}{\partial W_i} = \frac{\partial E_k}{\partial R} \times \frac{\partial R}{\partial W_i} = (R - Y)O_i,$$
$$\frac{\partial E_k}{\partial B} = \frac{\partial E_k}{\partial R} \times \frac{\partial R}{\partial B} = (R - Y),$$
$$\frac{\partial E_k}{\partial w_{ij}} = \frac{\partial E_k}{\partial R} \times \frac{\partial R}{\partial O_i} \times \frac{\partial O_i}{\partial I_i} \times \frac{\partial I_i}{\partial w_{ij}} = (R - Y)W_i O_i(1 - O_i)x_j,$$
$$\frac{\partial E_k}{\partial b_i} = \frac{\partial E_k}{\partial R} \times \frac{\partial R}{\partial O_i} \times \frac{\partial O_i}{\partial I_i} \times \frac{\partial I_i}{\partial b_i} = (R - Y)W_i O_i(1 - O_i),$$

使用梯度下降法对权重与偏置进行更新，得到更新过程为

$$W_i = W_i - \eta \frac{\partial E_k}{\partial W_i},$$
$$B = B - \eta \frac{\partial E_k}{\partial B},$$
$$w_{ij} = w_{ij} - \eta \frac{\partial E_k}{\partial w_{ij}},$$
$$b_i = b_i - \eta \frac{\partial E_k}{\partial b_i}.$$

2.2 关键代码

本次实验采用 python 实现 BPNN，关键代码分数据预处理与 BP 算法两部分。

2.2.1 数据预处理

实验所给的数据集中，有 **dteday、season、yr、mnth、hr、holiday、weekday** 七种离散的特征，其中 dteday 与其他特征取值有重复，选择保留日期作为特征取值，而 yr 特

征取值唯一，则将其除去。

离散的特征选择使用 One-Hot 进行编码，如 season 有 4 种取值，那么就用 4 个特征位来表示，原特征取值对应的特征位为 1，其余为 0。One-Hot 实现代码如下

```
def OneHot(data_set, index, attribute_set):
    # 特征值集合
    attribute_value = list(set([line[index] for line in data_set]))
    attribute_value = sorted(attribute_value)
    # 特征取值只有 0 与 1 则无需重新编码
    if(len(attribute_value) == 2):
        for i in range(0, len(data_set)):
            data_set[i] = data_set[i][0:index] + \
                data_set[i][index+1:] + [data_set[i][index]]
        attribute_set.append(attribute_set[index])
    # 特征取值唯一则删除
    elif(len(attribute_value) == 1):
        for i in range(0, len(data_set)):
            data_set[i] = data_set[i][0:index]+data_set[i][index+1:]
    # One-Hot
    else:
        # 将编码后的新特征放到数据集后
        for i in range(0, len(data_set)):
            one_hot = ['0' for temp in range(0, len(attribute_value))]
            # 对应特征位为 1
            one_hot[attribute_value.index(data_set[i][index])] = '1'
            data_set[i] = data_set[i][0:index]+data_set[i][index+1:]+one_hot
        # 新特征名为 原特征名 + 特征取值
        for i in range(0, len(attribute_value)):
            attribute_set.append(attribute_set[index]+attribute_value[i])
    attribute_set.pop(index)
    return
```

连续的特征选择使用 0 均值标准化，归一化公式如下，x 为特征取值，μ 为所有样例该特征取值的平均值，σ 为所有样例该特征取值的方差

$$x_{new} = \frac{x - \mu}{\sigma}.$$

归一化代码如下

```python
def Normalization(data_set, index):
    X = [line[index] for line in data_set]
    # 平均值
    X_avg = np.mean(X, axis=0)
    # 方差
    X_std = np.std(X, axis=0)
    # 归一化
    for i in range(0, len(data_set)):
        data_set[i][index] -= X_avg
        data_set[i][index] /= X_std
    return
```

使用以上两个函数对数据集进行预处理，具体处理代码较为简单，详情可见源码。本次实验训练集与测试集采用随机抽样的方式，按照一定比例进行随机划分。

2.2.2 BP 算法

根据 2.1推导出的梯度结果与 BP 算法伪代码进行代码编写，每次迭代分为两个过程：前向传递与反向传播，先在划分的训练集上进行训练，当迭代次数超过阈值后则停止迭代，再在划分的测试集上测试，得到测试集的均方误差，实现代码如下

```python
def BPNN(train_set, test_set, learning_rate, y_t, y_e, M, N, max_iter):
    # 对权重与偏置随机初始化
    w_MN = np.random.rand(M, N)
    b_M = np.random.rand(M)
    w_M = np.random.rand(M)
    b = random.random()

    count = 0
    x_in = np.array(train_set)
    y_r = np.array(y_t)
    # 开始迭代
    out = []
    while max_iter > count:
        # 初始化梯度
        delta_w_M = np.zeros((M))
```

```
        delta_b = 0.0
        delta_w_MN = np.zeros((M, N))
        delta_b_M = np.zeros(M)
        # 隐藏层输出
        O = (np.exp(2*(np.dot(x_in, w_MN.T)+b_M))-1)/(np.exp(2*(np.dot(x_in, w_MN.T)+b_M))+1)# Sigmoid 函数
        # O = 1/(1+np.exp(-np.dot(x_in, w_MN.T)-b_M))
        # O = np.maximum(0.01*(np.dot(x_in, w_MN.T)+b_M),np.dot(x_in, w_MN.T)+b_M)# LRelu 函数
        # 输出层输出
        R = np.dot(O, w_M)+b
        # 误差
        E = 1/2*(R-y_r)**2
        # 计算隐藏层到输出层权重的梯度
        delta_w_M = learning_rate*O*(R-y_r).reshape((-1, 1))
        # 所有样例梯度累加
        delta_w_M = delta_w_M.sum(axis=0)
        # 计算隐藏层到输出层偏置的梯度
        delta_b = learning_rate*(R-y_r)
        # 所有样例梯度累加
        delta_b = delta_b.sum(axis=0)
        # 计算输入层到隐藏层偏置的梯度
        delta_b_M = learning_rate*(R-y_r).reshape((-1, 1))*w_M*(1-O*O)
        # delta_b_M = learning_rate*(R-y_r).reshape((-1, 1))*w_M*np.minimum(np.maximum(O,0.01),1)# LRelu 函数
        # 计算输入层到隐藏层权重的梯度
        delta_w_MN = np.dot(delta_b_M.T, x_in)
        delta_b_M = delta_b_M.sum(axis=0)

        # 更新权重与偏置
        w_MN = w_MN-delta_w_MN/len(train_set)
        w_M = w_M-delta_w_M/len(train_set)
        b = b-delta_b/len(train_set)
        b_M = b_M-delta_b_M/len(train_set)
        count += 1
```

```
48          # 每次迭代结束后的均方误差
49          print(count)
50          print(E.sum()/len(train_set))
51          out.append(E.sum()/len(train_set))
52
53
54      temp_in = np.array(test_set)
55      y_temp = np.array(y_e)
56
57      O = (np.exp(2*(np.dot(temp_in,
        ↪  w_MN.T)+b_M))-1)/(np.exp(2*(np.dot(temp_in, w_MN.T)+b_M))+1)# tanh
        ↪  函数
58      # O = np.maximum(0.01*(np.dot(temp_in, w_MN.T)+b_M),np.dot(temp_in,
        ↪  w_MN.T)+b_M)# LRelu 函数
59      # O = 1/(1+np.exp(-np.dot(temp_in, w_MN.T)-b_M))# Sigmoid 函数
60      # 输出层输出
61      R = np.dot(O, w_M)+b
62          # 误差
63      E = 1/2*(R-y_temp)**2
64      print(" 测试集误差")
65      print(E.sum()/len(test_set))
66      return
```

3 实验结果及分析

3.1 参数调整

调整学习率，训练集的均方误差在不同学习率下随迭代的变化如图 4。当最大迭代次数固定为 1000，隐藏层节点数固定，在学习率为 0.1 时，均方误差稳定在 7000 以上，下降较为缓慢；学习率 0.05 会出现均方误差会出现小幅度的振荡，稳定在 2000 左右；学习率为 0.005 时下降较为平缓，但下降速度相比于学习率为 0.01 时较慢。综上，选择 0.1 作为最终的学习率。

由于输入层神经元数为 88，初步选择 50 到 88 作为隐藏层的神经元数目。调整隐藏层神经元个数，固定学习率为 0.01，训练集的均方误差在不同隐藏层神经元数目下随迭代的变化如图 5。当最大迭代次数固定为 1000，均方误差基本随着隐藏层节点数增加而减少，在隐藏层神经元数目达到 80 左右时，均方误差基本不再减少。考虑若隐层节点数太少，神经

网络性能可能较差；若隐层节点数太多，虽然可使网络的系统误差减小，但一方面使网络训练时间延长，另一方面，训练容易陷入局部极小点而得不到最优点，也是训练时出现“过拟合”的内在原因。综上选择 65 作为最终的隐藏层数据元数量，以提高泛化性。

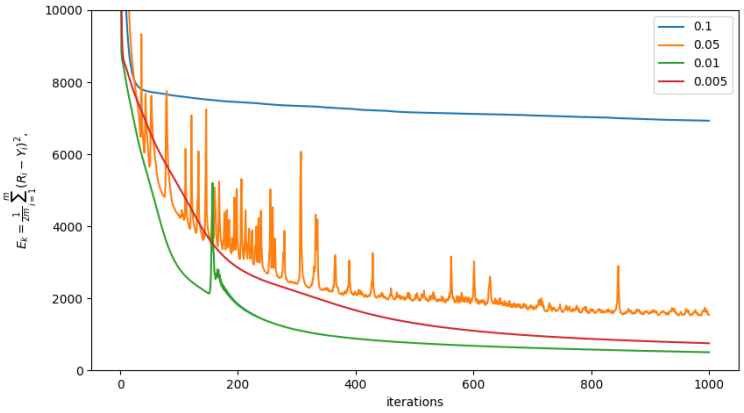


图 4: 训练集均方误差在不同学习率下随迭代次数的变化

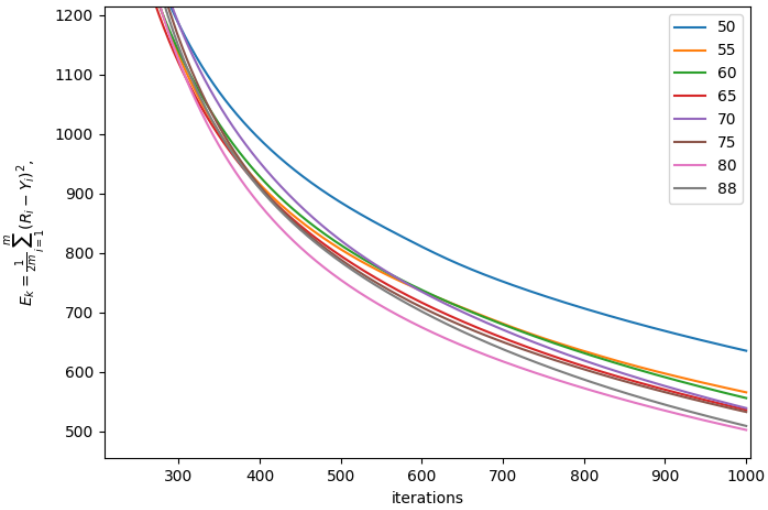


图 5: 训练集均方误差在不同隐藏层神经元数目下随迭代的变化

3.2 优化实现

在以上设置下，取训练集 200 个样本的预测结果进行观察，如图 6，可见训练到一定程度后，训练集在数值较大的样本训练效果较差。同样取测试集 200 个样本的预测结果进行

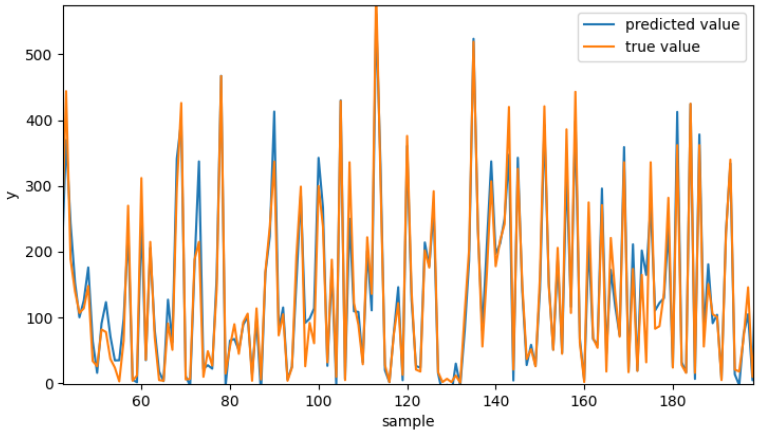


图 6: 训练集 200 个样本的预测结果比较

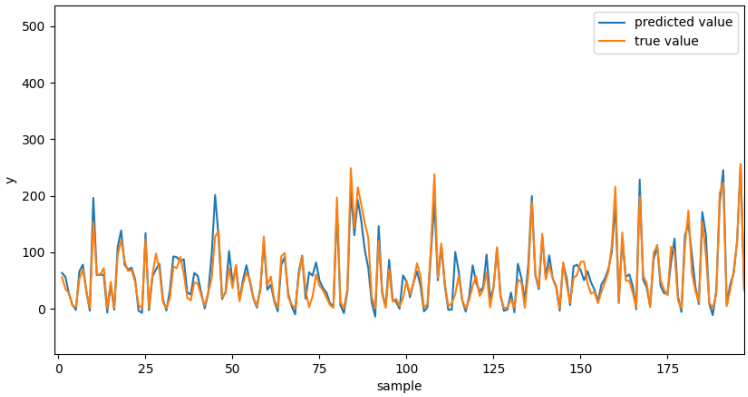


图 7: 测试集 200 个样本的预测结果比较

观察，如图 7，发现在测试集上有同样问题。

选择使用 **tanh 函数**作为激活函数，修改隐藏层的输出函数，训练集的均方误差在不同激活函数下随迭代的变化如图 8。可知在前 100 次迭代中，**tanh 函数**的下降速度明显优于 **sigmoid 函数**，而迭代到 500 步以后，两者的误差大致相等。取测试集 200 个样本的预测结果进行观察，如图 9，可见测试集上对某些数值较大的样本预测效果较好，而在一些数值适中的样本预测结果较差，这与 **tanh 函数**的中间区，即兴奋态变化较大有关系，隐藏层神经元输出容易趋向 0 或 1 导致变化较大。

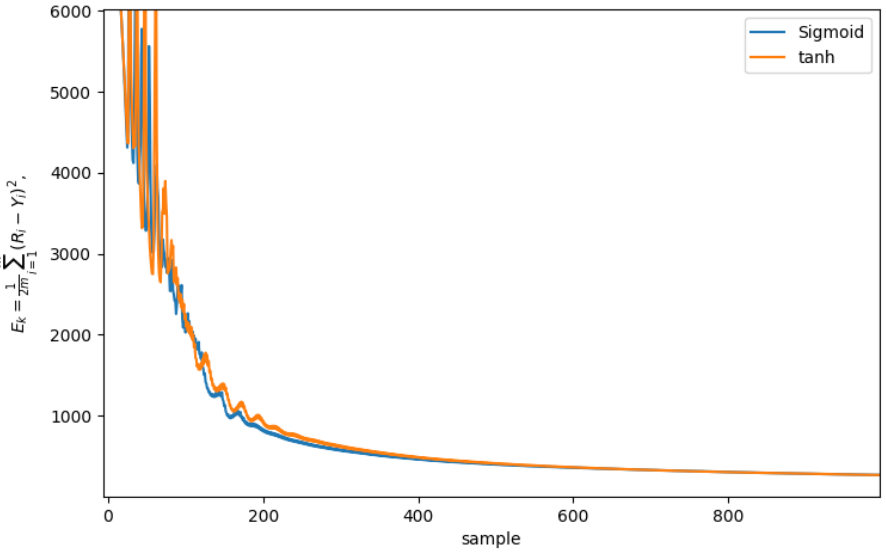


图 8: Sigmoid 函数与 tanh 函数的误差比较

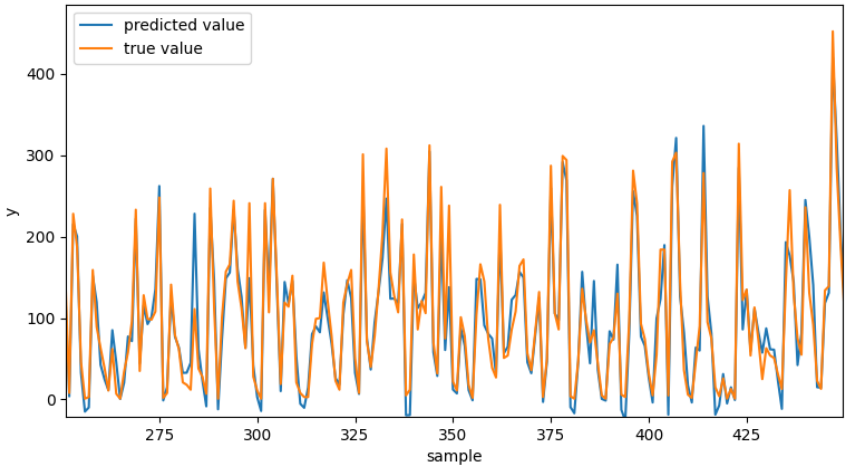


图 9: tanh 函数的预测结果比较

选择使用 **Leaky Relu** 作为激活函数，采用函数为

$$f(x) = \begin{cases} x, x \geq 0 \\ \alpha x, x < 0 \end{cases}$$

其中 $\alpha = 0.01$，修改隐藏层的输出函数，训练集的均方误差在 Leaky Relu 激活函数下随迭代的变化如图 10。可知在前 500 次迭代中，**Leaky Relu 函数**的下降速度较快，而迭代到 500 步以后，下降速度较缓慢。而 1000 步之后基本不下降，最终在 3000 左右稳定，拟合效果并不好，很大原因是由于连续的特征选择使用 0 均值标准化，因此有许多小于 0 的维度取值，导致更新的梯度较小，误差下降较慢。

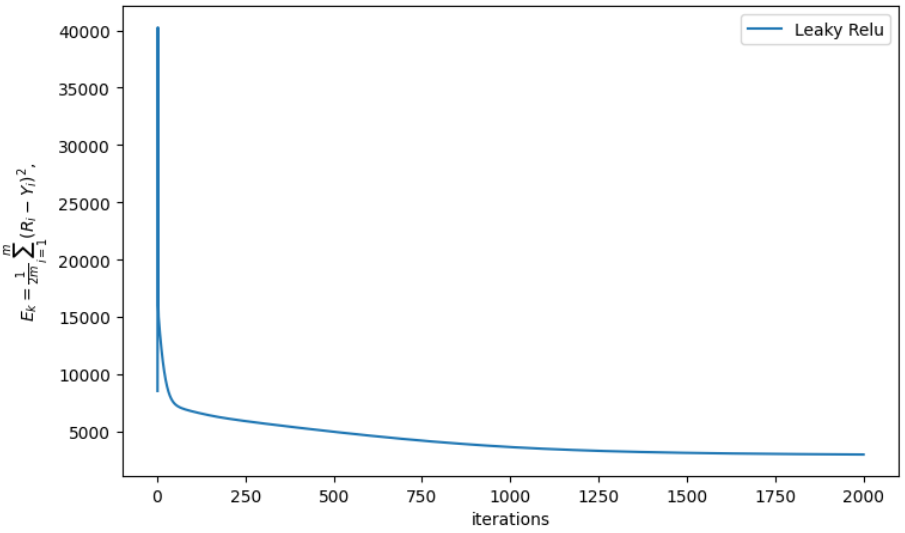


图 10: 训练集的均方误差在 Leaky Relu 激活函数下随迭代的变化

最终激活函数采用 **tanh 函数**。

4 实验思考

- **尝试说明下其他激活函数的优缺点。**

  1. Sigmoid 函数，从 $(-\infty, +\infty)$ 映射到 $(0, 1)$

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

     具备可求导的性质

     $$f'(x) = f(x)(1 - f(x))$$

优点：Sigmoid 函数对中央区的信号增益较大，对两侧区的信号增益小，在信号的特征空间映射上，有很好的效果。从神经科学上来看，中央区神经元的兴奋态，两侧区神经元的抑制态，因而在神经网络中可以将重点特征推向中央区，将非重点特征推向两侧区，而函数对输入超过一定范围就会不敏感。

缺点：存在饱和问题当输入非常大或者非常小的时候，神经元的梯度就接近于 0。在深度神经网络中梯度反向传递时导致梯度爆炸和梯度消失；sigmoid 的输出不是零中心的，即均值不为 0，这会导致传递到后面的数据均值不为零，导致梯度下降时的晃动，如果数据进入神经元的时候是正的，那么计算出的梯度也会始终都是正的。

2. Tanh 函数，从 $(-\infty,+\infty)$ 映射到 $(-1,1)$

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

具备可求导的性质

$$f'(x)=1-f(x)^2$$

优点：Tanh 函数输出均值为 0，解决了 Sigmoid 函数输出均值不为 0 的问题。Tanh 函数的变化敏感区间较宽，导数值渐进于 0、1，符合人脑神经饱和的规律，比 Sigmoid 函数延迟了饱和期；

缺点：同样存在饱和的问题，计算较复杂。

3. 修正线性单元 ReLU：整流线性单元，激活部分神经元，增加稀疏性。

$$f(x)=max(0,x)$$

求导更为简单

$$f'(x)=\begin{cases}1, x\geq 0\\ 0, x<0\end{cases}$$

优点：ReLU 函数非饱和；在梯度下降上有更快的收敛速度；没有指数运算，只是简单的设置一个阈值，比 Sigmoid/Tanh 函数操作开销小；

缺点：Relu 的输入值为负的时候，输出始终为 0，其一阶导数也始终为 0，无法再进行训练。该缺点可使用 Leaky Relu 进行修正。

- **有什么方法可以实现传递过程中不激活所有节点？**

Dropout 方法: 按照一定的概率随机丢弃一部分神经元，如图 14所示，这样每次都会有一种新的组合，假设有某一层有 N 个神经元，就有 2 的 N 次方个组合（子网络），最后子网络的输出均值就是最终的结果。Dropout 方法训练阶段在每个 mini-batch 中，依概率 P 随机屏蔽掉一部分神经元，只训练保留下来的神经元对应的参数，屏蔽掉的神经元梯度为 0，参数不参数与更新。

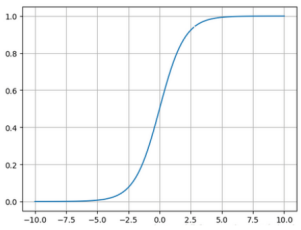


图 11: Sigmoid 函数

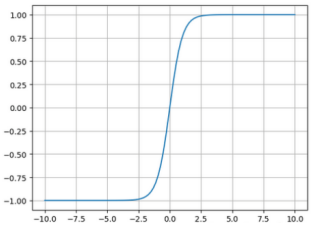


图 12: Tanh 函数

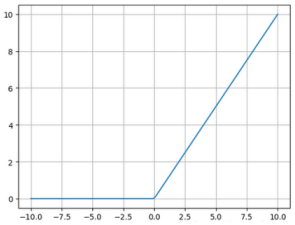


图 13: Relu

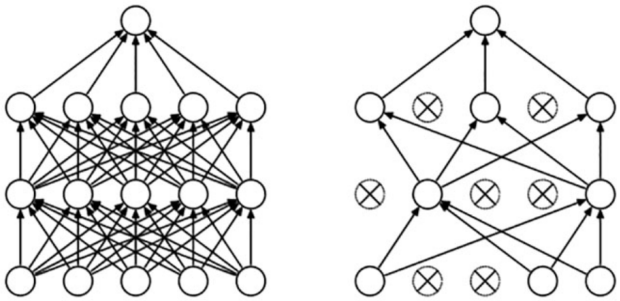
标准神经网络计算公式为

$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)}\mathbf{y}^l + b_i^{(l+1)},$$
$$y_i^{(l+1)} = f(z_i^{(l+1)}),$$

其中 f 为激活函数。Dropout 对公式进行修改

$$r_j^{(l)} \sim Bernoulli(p),$$
$$\tilde{\mathbf{y}}^{(l)} = \mathbf{r}^{(l)}\mathbf{y}^{(l)},$$
$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)}\tilde{\mathbf{y}}^l + b_i^{(l+1)},$$
$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)}\mathbf{y}^l + b_i^{(l+1)}.$$

其中 r 是依伯努利概率分布（0-1 分布）随机产生的向量，向量元素取值 0 或 1，取 1 的概率为 P，取 0 的概率为 1-P ，向量维度与某一层的输入神经元维度一致。r 向量与神经元对应元素相乘，r 中元素为 1 的被保留，为 0 的则置 0，那么只有被保留的神经元对应的参数得到训练，传递过程中不激活所有节点（强迫神经元和其他随机挑选出来的神经元共同工作，减弱了神经元节点间的联合适应性）。



(a) Standard Neural Net (b) After applying dropout.

图 14: Dropout

- **梯度消失和梯度爆炸是什么？可以怎么解决？**

  梯度消失：sigmoid 函数的导数曲线如图 15。在深度神经网络中梯度反向传递时前面的层比后面的层梯度变化更小，故变化更慢，从而引起了梯度消失问题。

  梯度爆炸：当权重初始化为一个较大的值时，那么从输出层到输入层每一层都会有一个权重的增倍，当神经网络很深时，梯度呈指数级增长，最后到输入时，得到一个非常大的权重更新，引发梯度爆炸问题，在循环神经网络中最为常见。

  解决方案：好的参数初始化方式，如 He 初始化，非饱和的激活函数（如 ReLU），批量规范化，梯度截断。

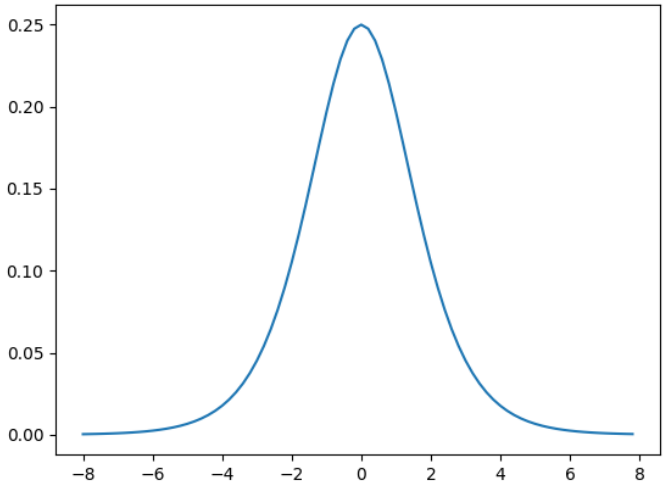


图 15: Sigmoid 函数导数