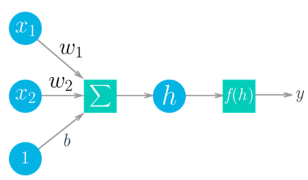
人工智能lab4 实验报告

> 学号：
>
> 姓名：TRY
>
> 专业：计算机科学与技术
>
> 时间：2020/10/21

一、算法原理

1.1 BPNN introduction

- **深度学习**的基本原理是基于人工神经网络的，而**BPNN**（反向传播神经网络）就是其中的典型模型。
- ### BPNN：受生物神经网络启发的计算系统。
  - 整体过程为：**前向传播+ 后向传播**。前者输出结果，后者计算误差，更新权重。
  - **整体思路**为：随机初始化权重向量`w`和偏置`b`。信号用`w`和`b`经过线性组合，从一个神经元进入，经过非线性的激活函数，传入到下一层神经元作为输入；再经过该层神经元的激活，继续往下传递，如此循环往复，直到输出层，这个过程叫做"**前向传播（forward_pass）**"。然后，再通过输出值与真实值，计算输出层和隐藏层的误差取值，通过**梯度下降法**从后往前更新各层的`w`和`b`的取值，直到输入层，这个过程叫做"**后向传播**"。
  - 以**三层神经网络**为例，BP神经网络含输入层、隐含层、输出层三层结构。输入层接收数据，输出层输出数据，前一层神经元连接到下一层神经元，收集上一层神经元传递来的信息，经过"激活"把值传递给下一层。
- ### 前向传播：
  - 

  - 以上图为例，假设$w=(w_1,w_2)$是连接前后两层神经元的权重。假设上一层神经元的输出为$x(x_1,x_2)$，则这一层神经元的输入h就是上一层神经元输出的加权值和（加上偏置），然后输出就是对于输入h的激活$f(h)$：

$$h = w_1 * x_1 + w_2 * x_2 + b$$
$$y' = f(h), f(h)\text{为激活函数}$$

  - **激活函数**有很多种，如

$$f(h) = Sigmoid(h) = \frac{1}{1+e^{-h}}$$
$$f(h) = Tanh(h) = \frac{e^h - e^{-h}}{e^h + e^{-h}}$$
$$f(h) = Leaky\_Relu(h) = max(0.01h, h) = \begin{cases} h,\ h >= 0 \\ 0.01h,\ h < 0 \end{cases}$$

- ### 后向传播：

- 在反向传播阶段， 权重更新方程应用于相反的方向。也就是说， 第$(l+1)$层的权重在更新第 l 层的权重之前被更新，这允许我们使用第$(l+1)$层神经元的误差来估计第 l 层神经元的误差。
- 使用"**梯度下降法**"不断迭代更新权重向量w和偏置 b。
- 下面以本实验要求的**三层神经网络**，且输出层只有一个节点为例分析。
- **损失函数**：$E=\frac{1}{2}(y-y')^2$，目的将误差最小化。
- 对于**输出层**中的单元`k`，误差Err_k 由下式计算：

  $$Err_k=(y-y')f'(h)$$

  其中h 为输出节点的输入，y'为输出层预测值，y为输出层真实值。
- 对于**隐藏层**单元`j`，误差Err_j为：

  $$Err_j=Err_k\ w_{jk}\ f'(h_j)$$

  其中，Err_k为输出层误差，w_{jk}为连接隐藏层和输出层之间的权重向量，$f'(h_j)$ 为隐藏层激活函数的导数，h_j为隐藏层输入。
- 每个数据点更新权重**步长（梯度）**：

  $$\begin{gathered}\triangle W_{jk}=\triangle W_{jk}+Err_k*O_j\\ \triangle W_{ij}=\triangle W_{ij}+Err_j*O_i\\ \triangle b_j=\triangle b_j+Err_k\\ \triangle b_i=\triangle b_i+Err_j\end{gathered}$$

  其中，`i`表示输入层，`j`表示隐藏层，`k`表示输出层，O_x表示对应层的输出。
- 当所有数据点都计算更新完之后，求均值，更新权重向量：

  $$\begin{gathered}W_{jk}=W_{jk}+\eta\ \frac{\triangle W_{jk}}{m}\\ W_{ij}=W_{ij}+\eta\ \frac{\triangle W_{ij}}{m}\\ b_i=b_i+\ \eta\ \frac{\triangle b_i}{m}\\ b_j=b_j+\ \eta\ \frac{\triangle b_j}{m}\end{gathered}$$

  其中，m为数据点的个数，η为学习率。

1.2 数据预处理

1.2.1 对类型变量的处理：one-hot编码

- 在我们这个案例中，`season`,`mnth`,`weathersit`,`hr`,`weekday`这几个变量都是我们为了日常生活方便而约定的，属于**类型变量**。他们的数字大小没有实际的意义，比如星期二并不代表它比星期一的值大，直接把大小输入模型，会造成神经网络的"**误解**"：取值越大，会更强烈地影响网络内部的变化。所以我们需要对这些类型变量做二进制处理。
- 拿`weekday`举例，我们需要把星期的标识，全部转化为二进制编码，哪一位为1，就代表对应的类型（其实就相当于会激活与之相关的神经元）

weekday	类型变量	类型编码
Sunday	0	1000000
Monday	1	0100000
Tuesday	2	0010000
Wednesday	3	0001000
Thursday	4	0000100
Friday	5	0000010
Saturday	6	0000001

- 具体代码可用`pandas`中的`get_dummies`函数来进行生成。

1.2.2 数值归一化

- 通常，原始数据中不同单位的变量，他们的数值差异有可能会非常大。
  - 尽管在本次实验中，`temp`,`hum`,`windspeed`三个变量已经做过初步的归一化处理（除了一个数），但是为了**提高训练速度和训练效果**，使得他们的分布更均匀，加快权重学习效率，仍旧再次进行归一化操作。
- 而我采用的是最基础的归一化操作，将数值转化到`[0, 1]`的区间：

$$新数据 = (原数据 - 最小值)/(最大值 - 最小值)$$

- **针对`cnt`要不要归一化的问题：**
  - 实际上，也可以对`cnt`变量进行归一化，然后再对归一化后的`cnt`进行误差计算。但由于需要输出共享单车的预测值，且归一化后的准确率的`threshold`难以确定，所以**不**对`cnt`进行**归一化**操作。
  - 也正是这个原因，**输出层的激活函数应该设置为**$y = x$，而不用其他特殊的激活函数。而**隐藏层**则可使用如**sigmoid**等的激活函数。
    - 预测值会往真实的整数值不断靠近。

1.2.3 特征选取

- 在数据集中，可以能看到一些**冗余的特征**，如`dteday`实际上已经通过`season`、`yr`、`mnth`、`weekday`等进行了分类。因此，数据集的`dteday`特征是可以丢弃的。
- 同理，由于`temp`和`atemp`一个表示温度，一个表示体感温度，两者的含义实际上类似，所以我也只保留了一个`temp`来反应温度的指标，丢弃`atemp`。

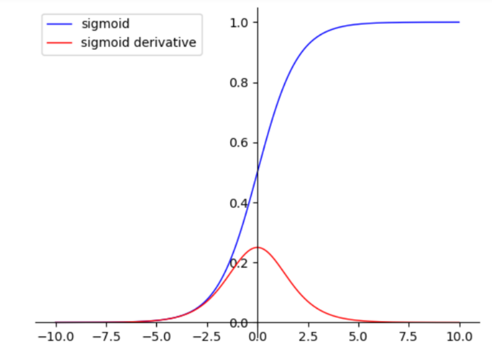
1.3 几种激活函数

- **非线性激活函数的必要性**
  - 如果使用**线性激活函数**，那么神经网络仅是将输入线性组合再输出，在这种情况下，多个隐藏层神经网络与只有一个隐藏层的神经网络没有任何区别，不如去掉多个隐藏层。 且实际上就是输入的线性组合，隐藏层没有起效。
- **$Sigmoid$函数**

- 公式：

$$\text{函数：} f(h) = Sigmoid(h) = \frac{1}{1+e^{-h}}$$
$$\text{导数：} f'(h) = f(h)(1-f(h))$$

- 图像：



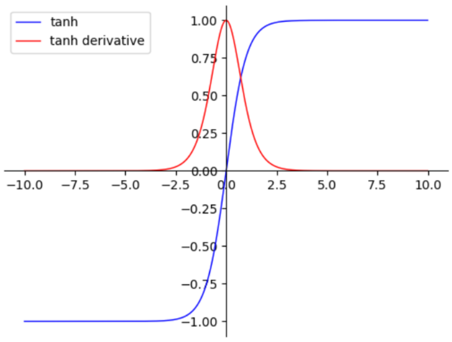
- 优缺点：详见思考题1

• $Tanh$函数

- 公式：

$$\text{函数：} f(h) = Tanh(h) = \frac{e^h - e^{-h}}{e^h + e^{-h}} = \frac{e^{2h}-1}{e^{2h}+1} = 2 * Sigmoid(2h) - 1$$
$$\text{导数：} f'(h) = 1 - Tanh^2(h)$$

- 图像：



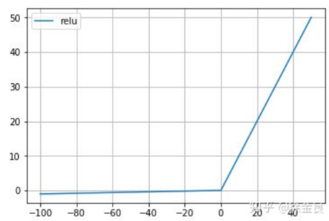
- 优缺点：详见思考题1

• $LeakyRelu$函数

- 公式：

$$\text{函数：} f(h) = Leaky\_Relu(h) = max(0.01h, h) = \begin{cases} h,\ h >= 0 \\ 0.01h,\ h < 0 \end{cases}$$
$$\text{导数：} f'(h) = \begin{cases} 1,\ h > 0 \\ undefined,\ h = 0 \\ 0.01,\ h < 0 \end{cases}$$

- 图像：



- 优点：详见思考题1

1.4 损失函数

常见的损失函数有以下几种：

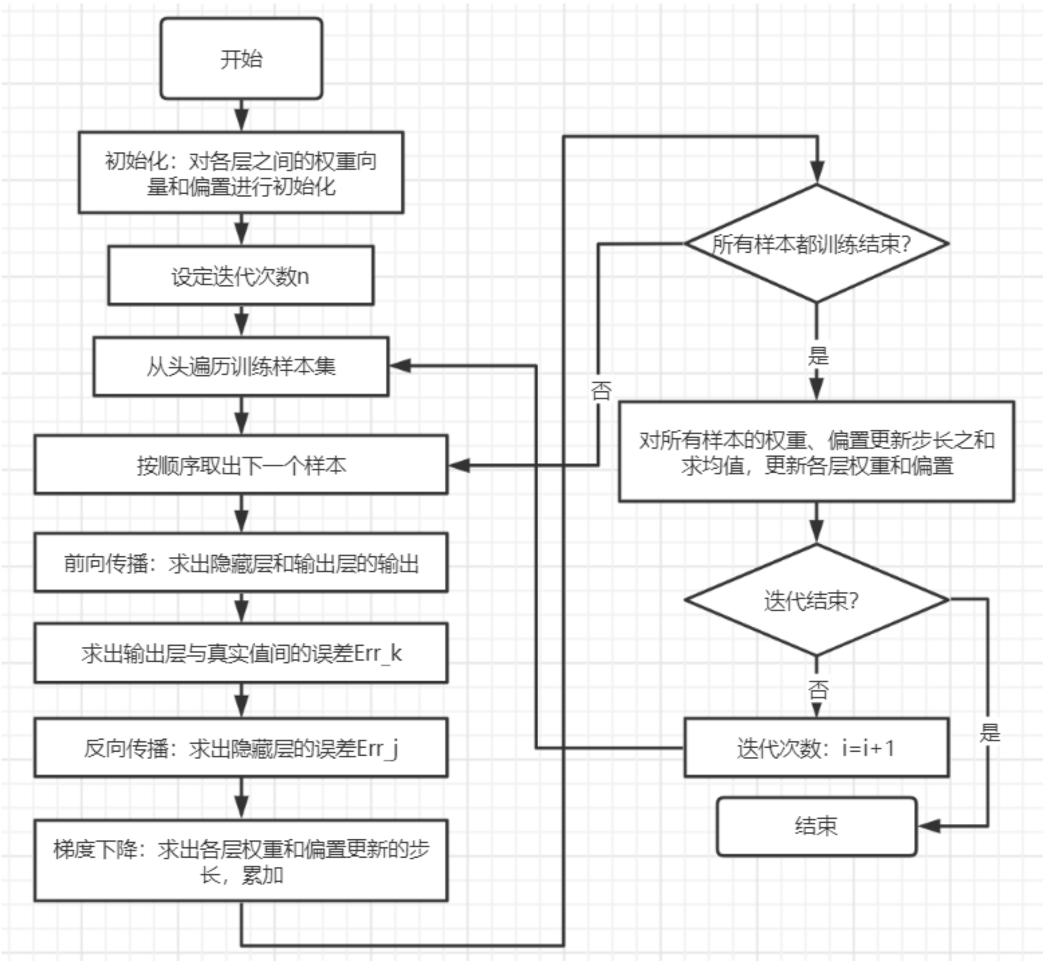
- **绝对值损失函数**：
$$L(Y,f(x)) = |Y - f(x)|$$
- **log对数损失函数**：LR使用
$$L(Y,P(Y|X)) = -logP(Y|X)$$
- **平方损失函数**：常应用于回归问题
$$L(Y|f(x)) = \sum_N (Y - f(x))^2$$
  在本实验中，应用了平方损失函数的变式：
$$L(Y|f(X)) = \frac{1}{2}\sum_N (Y - f(X))^2$$

1.5 如何划分数据集？

- 在本次实验中，考虑到数据并没有产生聚集的现象（即相同输入的样本聚集在一起），我直接将数据集按照`7:3`的比例进行划分，前面7份作为训练集，后面3份作为验证集来进行训练和验证。

二、流程图

- 以下展示BPNN算法的流程图：



三、代码截图

- one-hot函数：对类型变量进行处理

```
def one_hot(dataset):
    """
    利用pd.get_dummies函数，对season, mnth, weathersit, hr, weekday进行one-hot
编码。
    原因：如对于season来说，四季编码为0,1,2,3的数字大小没有实际意义，所以需要进行one-
hot编码
    :param dataset:数据集
    :return:返回one-hot编码完的数据集
    """
    # 需要进行one-hot处理的特征组合
    dummy_features = ['season', 'mnth', 'weathersit', 'hr', 'weekday']
    for feature in dummy_features:
        dummies = pd.get_dummies(dataset[feature], prefix=feature,
drop_first=False)    # prefix表示前缀
        # 合并进dataset中
        dataset = pd.concat([dataset, dummies], axis=1)     # 用concat来合
并，而不用join来合并。join是dataframe的函数
    # 把原来的列去掉（加上atemp，两个温度差不多）
    features_to_drop = ['season', 'mnth', 'weathersit', 'hr', 'weekday',
'atemp']
    dataset = dataset.drop(features_to_drop, axis=1)
    # 把cnt这一列移到最后
    cnt_temp = dataset['cnt'].values        # 取了values之后才可以用reshape函数
转成列向量
    cnt_temp = cnt_temp.reshape(len(cnt_temp), 1)
    dataset = dataset.drop('cnt', axis=1)
    dataset['cnt'] = cnt_temp    # 这样就可以插入到最后一列，且带标签名！！
    return dataset
```

- 归一化函数：对变量进行归一化处理

```
def normalization(dataset):
    """
    归一化函数：对temp,hum,windspeed,cnt进行归一化处理
    :param dataset: 数据集
    :return: 返回数据集，且去掉了第一行（之后不需要属性名了）
    """
    # 需要进行归一化处理的属性：
    nor_features = ['temp', 'hum', 'windspeed']
    data_num = dataset.shape[0]
    list = []
    for feature in nor_features:
        list.append(dataset[feature])    # 这里提取出来会变成行向量
    list = np.array(list)    # 转成numpy数组
    list = list.reshape(-1, len(nor_features))    # 转置
    mean = np.mean(list, axis=0)
    min = np.min(list, axis=0)       # axis=0表示0维消失
    max = np.max(list, axis=0)
```

```
    temp = max - min
    for i, feature in enumerate(nor_features):
        dataset.loc[:, feature] = (dataset[feature] - min[i]) / temp[i]
    dataset = dataset.values
    return dataset
```

- 初始化参数函数和生成矩阵函数：

```
def make_matrix(row_num, col_num):
    result = []
    for i in range(row_num):
        temp = []
        for j in range(col_num):
            temp.append(random.uniform(-1.0, 1.0))     # 产生一个随机数在
（-1,1）之间
        result.append(temp)
    return result


def init_para(dataset, hnode_num):
    feature_num = dataset.shape[1] - 1  # 第一维-1等于变量数
    w_hidden = np.array(make_matrix(feature_num, hnode_num))
    b_hidden = np.array(make_matrix(1, hnode_num))
    w_output = np.array(make_matrix(hnode_num, 1))
    b_output = random.uniform(0, 1.0)   # b_output初始化为[-1,1]之间的一个数
    return w_hidden, b_hidden, w_output, b_output
```

- 激活函数和激活函数求导：

```
def activation_func(func_type, x):
    if func_type == "sigmoid":
        return sigmoid(x)
    elif func_type == "tanh":
        return 2 * sigmoid(2*x) - 1
    elif func_type == "relu":
        return np.maximum(0.01*x, x)

def derivation(func_type, hidden_output):    # 激活函数的求导
    if func_type == "sigmoid":
        return hidden_output * (1 - hidden_output)
    elif func_type == "tanh":
        return 1 - np.square(hidden_output)
    elif func_type == "relu":
        temp = [[0.0] * len(hidden_output[0]) for i in
range(len(hidden_output))]      # 二维数组初始化
        for i, data in enumerate(hidden_output):
            for j, data_each in enumerate(data):
                if data_each < 0:
                    temp[i][j] = 0.01
                else:
                    temp[i][j] = 1
        result = np.array(temp)
        return result
```

- 前向传播函数：得到隐藏层和输出层的输出（通过`numpy`进行并行加速）

```python
def forward_pass(dataset, w_hidden, b_hidden, w_output, b_output, 
func_type):
    data_num = dataset.shape[0]      # 样本数
    # 隐藏层计算
    b_hidden1 = np.repeat(b_hidden, data_num, axis=0)    # 将b_hidden沿着行扩展
    dataset1 = dataset[:, 0:dataset.shape[1] - 1]  # 剔除掉最后一列的label
    hidden_input = np.dot(dataset1, w_hidden)     # 隐藏层节点
    hidden_input = hidden_input + b_hidden1    # 加上偏置theta
    hidden_output = activation_func(func_type, hidden_input)     # 激活函数
    # 输出层计算
    output = np.dot(hidden_output, w_output)     # 输出层预测输出
    b_output1 = np.array([b_output] * data_num)      # 这样可以形成数组，且各单元
值相同。因为这只是一个数，所以用按行repeat不会形成列向量；多个数就会
    b_output1 = b_output1.reshape(-1, 1)        # 转成列向量
    output = output + b_output1      # 加上偏置量
    return output, hidden_output
```

- 后向传播函数：计算误差，得到更新权重的步长，求均值，更新权重和偏置

```python
def backward_pass(dataset, output, hidden_output, learning_rate, w_hidden, 
b_hidden, w_output, b_output, func_type):
    feature_num = dataset.shape[1] - 1  # 第一维-1等于变量数
    hnode_num = hidden_output.shape[1]  # 隐藏层节点数
    data_num = dataset.shape[0]  # 样本数
    dataset1 = dataset[:, 0:dataset.shape[1] - 1]  # 剔除掉最后一列的label
    label = (dataset[:, dataset.shape[1]-1]).reshape(data_num, 1)   # 取出最
后一列，并转成列向量
    # 计算输出层误差err_output
    err_output = label - output      # 此时激活函数 y=x的导数为1
    # 计算err_output * w_output
    temp = np.repeat(err_output, hnode_num, axis=1)  # 沿着列扩展err_k
    temp_w_output = np.repeat(w_output.reshape(1, hnode_num), data_num, 
axis=0)   # 沿着行扩展w_output，原来是（hnode_num,1）
    temp = temp * temp_w_output

    # 计算隐藏层误差 err_hidden
    err_hidden = derivation(func_type, hidden_output) * temp
    # 求出所有样本的平均误差：输出层平均
    err_output1 = np.repeat(err_output, hnode_num, axis=1)
    err_output1 = err_output1 * hidden_output    # 先乘法
    err_output_mean_w = np.mean(err_output1, axis=0)     # 更新w时的误差*Oj的均
值
    err_output_mean_b = np.mean(err_output)      # 更新b时的误差均值
    # 更新输出层的w,b：
    for i in range(hnode_num):
        w_output[i] += learning_rate * err_output_mean_w[i]
    b_output += learning_rate * err_output_mean_b

    # 求出所有样本的平均误差：隐藏层平均
    err_hidden_mean_b = np.mean(err_hidden, axis=0)
    err_hidden_mean_w = []
    for i in range(hnode_num):
        err_hidden_each = (err_hidden[:, i]).reshape(-1, 1)  # 取出一个隐藏层
节点的误差
        err_hidden_each1 = np.repeat(err_hidden_each, feature_num, axis=1)
 # 沿着列扩展
        err_hidden_each1 = err_hidden_each1 * dataset1
```

```
        err_hidden_mean_w.append(np.mean(err_hidden_each1, axis=0))
    err_hidden_mean_w = np.array(err_hidden_mean_w)
    # 更新隐藏层的w,b:
    for j in range(hnode_num):
        for i in range(feature_num):
            w_hidden[i][j] += learning_rate * err_hidden_mean_w[j][i]
        b_hidden[0][j] += learning_rate * err_hidden_mean_b[j]      #
b_hidden是一个行向量，(1,hnode_num)
    return w_hidden, b_hidden, w_output, b_output
```

- BPNN函数：调用前向传播和后向传播

```
def bpnn(hnode_num, dataset, iteration, learning_rate, func_type):
    w_hidden, b_hidden, w_output, b_output = init_para(dataset, hnode_num)
    i = 0
    for i in range(iteration):
        # 前向传播：得到输出层和隐藏层的输出
        print("在迭代次数为%s时，预测值和真实值分别为：" % i)
        output, hidden_output = forward_pass(dataset, w_hidden, b_hidden,
w_output, b_output, func_type)
        # 后向传播：更新两组w,b
        w_hidden, b_hidden, w_output, b_output = backward_pass(dataset,
output, hidden_output, learning_rate,
                                                               w_hidden,
b_hidden, w_output, b_output, func_type)
    return w_hidden, b_hidden, w_output, b_output
```

- 验证函数：预测验证集的输出，并计算loss和准确率（设置阈值threshold）

```
def validation(dataset, func_type, w_hidden, b_hidden, w_output, b_output,
threshold):
    output, hidden_output = forward_pass(dataset, w_hidden, b_hidden,
w_output, b_output, func_type)
    label = dataset[:, dataset.shape[1] - 1].reshape(-1, 1)     # 取出最后一列
    diff = output - label
    loss = np.mean(np.square(diff)) / 2      # 计算loss
    cnt = 0
    for i in range(diff.shape[0]):
        if diff[i][0] < threshold:
            cnt += 1
    accuracy = cnt / diff.shape[0]
    return loss, accuracy
```

四、实验结果以及分析

1. 结果展示和分析

（1）$Sigmoid$函数

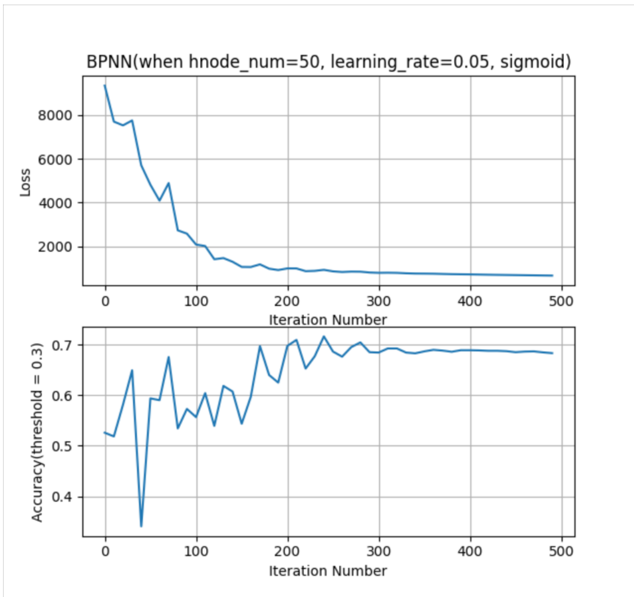
- 当迭代500次时，分别输出训练集、验证集的头5个样本的预测结果和真实值结果：

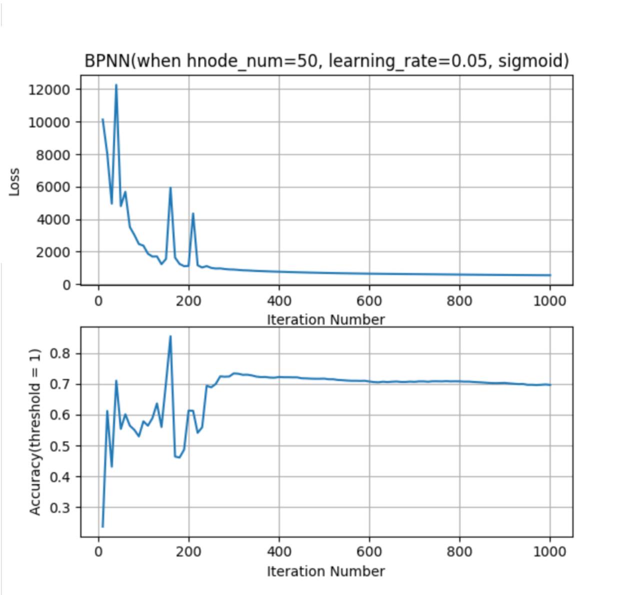
```
[[17.3520357 ]
 [33.94668467]
 [26.1222944 ]
 [12.07638984]
 [ 0.89094577]]
[16. 40. 32. 13.  1.]
```

```
[[568.29759865]
 [539.60515283]
 [384.78972472]
 [275.43549627]
 [204.41034617]]
[579. 539. 396. 272. 245.]
loss of valid_set is 1321.5123291642885, accuracy is 0.5621141975308642 (when threshold is 0.3)
```

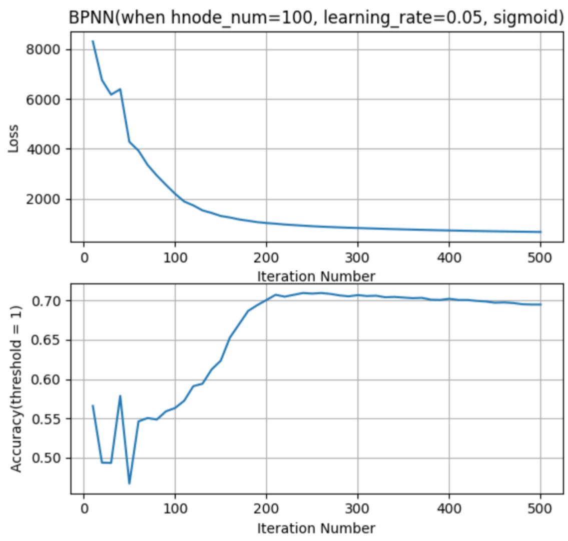
此时可以看出，对于训练集和验证集来说，预测值和真实值较为接近，因此可初步看出模型是正确的。

- 当使用**sigmoid**作为激活函数，隐藏层节点为50，学习率为0.05时，**迭代**500次和1000次的结果如下：

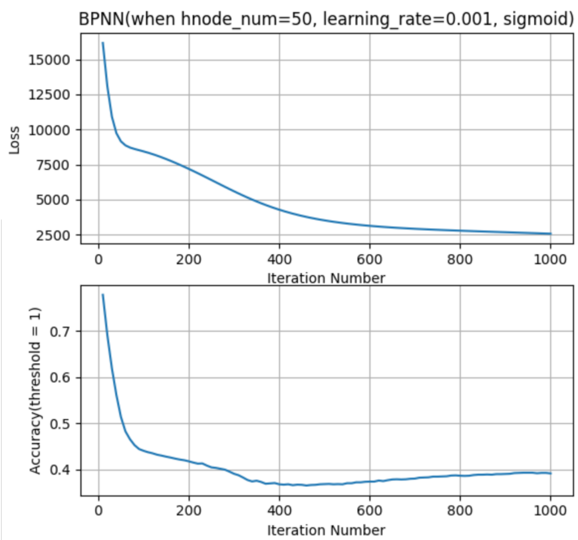




- 当使用**sigmoid**作为激活函数，**隐藏层节点**为100，学习率为0.05时，迭代500次的结果如下：



- 当使用**sigmoid**作为激活函数，隐藏层节点为50，**学习率**为0.001时，迭代1000次的结果如下：



- 在运行时，我通过遍历迭代次数从1到500/1000来查看该条件下**最小的loss**和**准确率**。此处的**准确率**是通过设置threshold，当$|\text{预测值}-\text{真实值}|<threshold$时，判断预测正确；否则判断预测错误。
  - 实际上，本回归问题应该通过loss来判断梯度收敛，用accuracy并不严谨（原因在于threshold的选择恰当与否），但当threshold一定的情况下，也可以作为参考指标。在本次实验中，我将threshold设置为1。
- **对于$sigmoid$来说：**
  - **整体曲线趋势**：从上面各图可以看出，训练集的**loss**整体是先下降后趋于平稳的，一开始可能会出现波动，但整体趋势不变，符合模型设定。对应地，准确率`accuracy`的整体趋势是先波动上升后趋于平稳。
  - **是否梯度收敛**：从以上图都可看出，`loss`值在到达一定迭代次数的时候，`loss`值都趋于稳定，因此可判断模型收敛，与预测相符。
  - **学习率的设置**：由上面学习率设置为`0.05`和`0.001`的比较可以看出，在`sigmoid`的情况下，学习率的设置**对收敛速度和收敛效果影响较大**。当学习率$\eta=0.05$时，`loss`在200的时候开始收敛，且收敛到676左右；而在学习率$\eta=0.001$时，`loss`在600的时候开始收敛，且收敛到2556左右，明显大于学习率较大的时候。且对应得，准确率也明显低于学习率较大的时候。

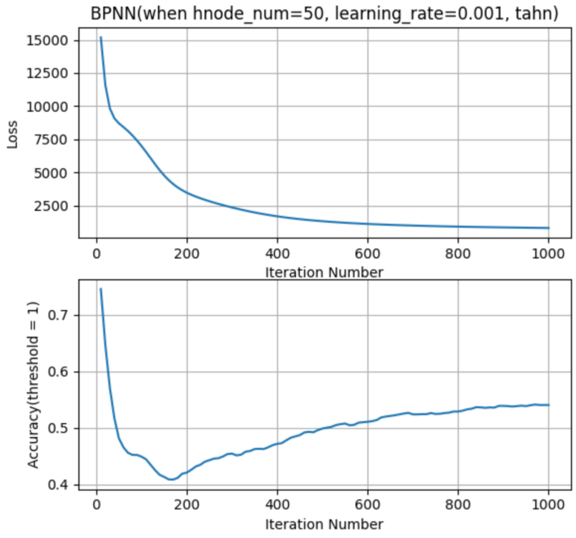
这说明**学习率的设置**在`sigmoid`的BPNN中是十分重要的，只有设置合适的学习率才会有较好的结果。

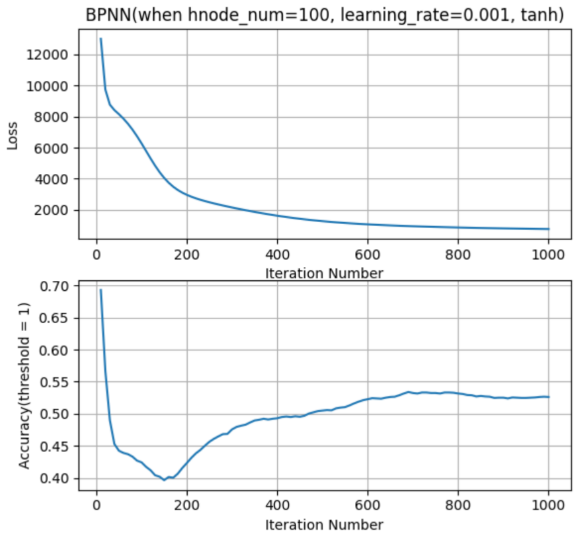
- **隐藏层节点的设置**：通过上图的比较，可以看到，当隐藏层节点个数设置为50和100的时候，模型的收敛速度类似，都在迭代次数为200左右收敛；且收敛到的`loss`值都位于650~750之间，较为接近。

  因此可知，隐藏层节点个数在单层隐藏层的`sigmoid`的BPNN中**不是关键因素**。

(2) $Tanh$函数

- 当使用$Tanh$作为激活函数，**隐藏层节点**为50和100，学习率为0.001时，迭代1000次的结果如下：





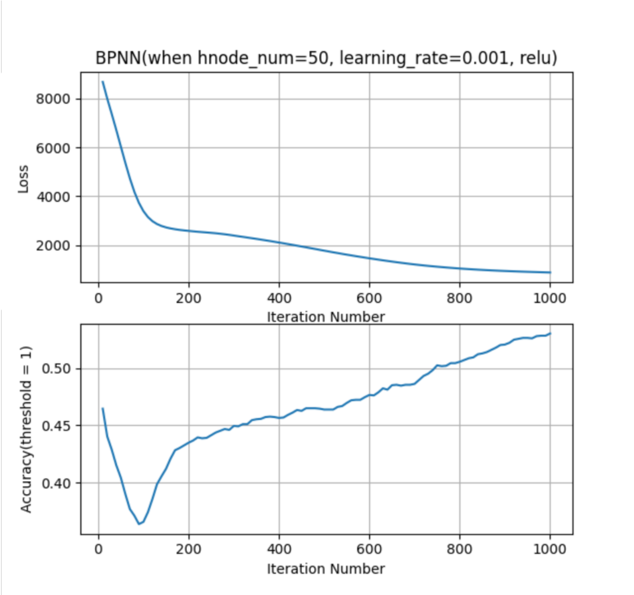
- **对于$tanh$来说：**
  - **整体曲线趋势**：训练集的**loss**整体是先下降后趋于平稳的，符合模型设定。对应地，准确率**accuracy**的整体趋势是先下降后上升，最后趋于平稳。
  - **是否梯度收敛**：从以上图都可看出，`loss`值在到达一定迭代次数的时候，`loss`值都趋于稳定，因此可判断模型收敛，与预测相符。
  - **学习率的设置**：由于$tanh$在学习率较大的时候会**溢出**，所以将学习率设置为0.001。

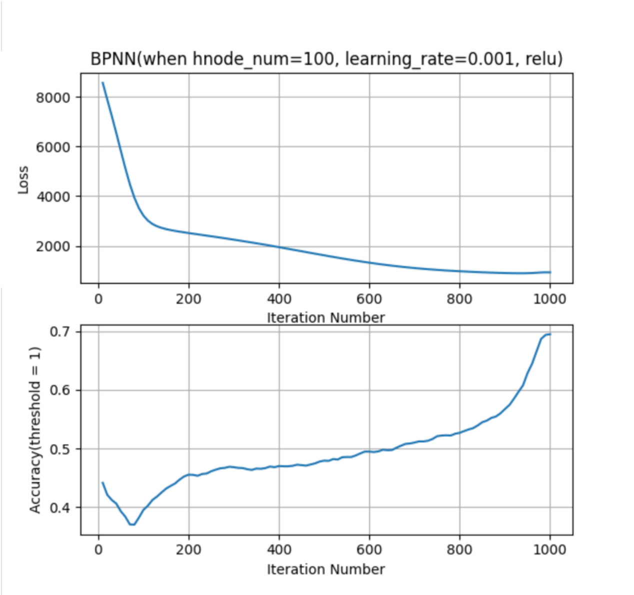
- **隐藏层节点的设置**：通过上图的比较可以看到，当隐藏层节点个数设置为50和100的时候，模型的收敛速度类似，都在迭代次数为600左右收敛；且收敛到的loss值都位于760~830之间，较为接近。同样的，对于准确率也是一样的结论，即两条曲线无论从趋势还是取值来说，基本比较类似。

  因此可知，隐藏层节点个数在单层隐藏层的$tanh$的BPNN中**不是关键因素**。

(3)$Leaky\ Relu$函数：

- 当使用$Leaky\ Relu$作为激活函数，，**隐藏层节点**为50和100，学习率为0.001时迭代1000次的结果如下：





- 对于$Leaky\ Relu$来说：
  - **整体曲线趋势**：训练集的**loss**整体是先下降后趋于平稳的，符合模型设定。对应地，准确率**accuracy**的整体趋势是先下降后上升，但由于迭代次数较少，因此还**未出现**明显平稳的趋势。
  - **是否梯度收敛**：从以上图都可看出，`loss`值在到达一定迭代次数的时候，`loss`值都趋于稳定，因此可判断模型**收敛**，与预测相符。
  - **学习率的设置**：由于$tanh$在学习率较大的时候会**溢出**，所以将学习率设置为0.001。

- **隐藏层节点的设置**：通过上图的比较可以看到，从`loss`的角度看，当隐藏层节点个数设置为50和100的时候，模型的收敛速度类似，都在迭代次数为800左右开始收敛；且收敛到的loss值都位于891~934之间，较为接近。

  然而，从**准确率**`accuracy`的角度看，两幅图的走势虽然类似，但两者最终在迭代次数为1000的时候到达的准确率却有**较大差异。**在隐藏层节点数为50的情况下，到达的准确率为53%，而在节点数为100时，准确率到达了69%。

  因此可知，隐藏层节点个数在单层隐藏层的$Leaky\ Relu$的BPNN中**是关键因素。**

（4）三种函数的比较

- 可以看出，其他条件（学习率η，隐藏层节点个数）相同的情况下，**收敛速度**从大到小依次为$Sigmoid > Tanh > Leaky\ Relu$，例如在隐藏层节点为50，学习率$\eta = 0.001$时，$Tanh$和$Leaky\ Relu$分别在迭代次数为500和800左右开始收敛，而$sigmoid$在迭代次数为200左右就收敛了。
- 但是，同样的条件下，**收敛效果**从好到坏依次为$Tanh > Leaky\ Relu > Sigmoid$，例如在隐藏层节点为50，学习率$\eta = 0.001$时，$tanh$会收敛到824左右，$Leaky\ Relu$会收敛到891左右，远小于$sigmoid$的2556。

2. 模型性能展示和分析

- 以下呈现关于“激活函数类型”，“学习率η”，“隐藏层节点个数”，”迭代次数“的**调参**，判断标准为训练集`loss`值和准确率`accuracy`（$threshold = 1$的情况）。其中，主要通过loss判断回归模型的效果，accuracy作为对应loss取值下的参考指标。

优化	激活函数	学习率	隐藏层节点数	最小loss	对应的准确率	对应的迭代次数

优化	激活函数	学习率	隐藏层节点数	最小loss	对应的准确率	对应的迭代次数
初始	Sigmoid	0.05	50	7501.46	35.76%	10
优化1	Sigmoid	0.05	50	676	68.33%	500
优化2	Sigmoid	0.05	50	546	69.60%	1000
优化3	Sigmoid	0.001	50	2556	39.16%	1000
优化4	Sigmoid	0.05	100	656	69.48%	500
优化5	Tanh	0.001	50	824	54.05%	1000
优化6	Tanh	0.001	100	764	52.62%	1000
优化7	Leaky Relu	0.001	50	891	53.01%	1000
优化8	Leaky Relu	0.001	100	934	69.48%	1000
最优	Sigmoid	0.05	50	546	69.60%	1000

分析：激活函数类型、学习率、迭代次数对于训练集的最小loss影响较大，而隐藏层节点个数对于$Sigmoid$的影响较大，对于$Tanh, Leaky\ Relu$的影响较小。

五、创新点

1. 在数据预处理中，对类型变量`season`,`mnth`,`weathersit`,`hr`,`weekday`进行了**"one-hot"处理**，消除了数字大小的意义，使得他们的取值平均，避免神经网络的"误解"（即取值越大，越强烈地影响网络内部的变化）。且效果会比不进行one-hot处理的好很多！
2. 在数据预处理中，对`temp`,`hum`,`windspeed`数据进行了**归一化**normalization处理，**提高训练速度和训练效果**，使得他们的分布更均匀，加快权重学习效率。
3. 使用了$numpy$操作，并行计算前向传播的输出和后向传播的权重、偏置更新，大大加快了计算运行的效率。
4. 使用了多种激活函数。

六、思考题

1. **尝试说明下其他激活函数的优缺点**

（1）$Sigmoid$函数

- **优点**：Sigmoid的取值范围在(0, 1)，而且是单调递增，比较容易优化； 求导比较容易，可以直接推导得出。
- **缺点**： Sigmoid函数**收敛比较缓慢**；Sigmoid函数并不是以（0,0）为中心点； 由于Sigmoid是软饱和，容易产生梯度消失，对于深度网络训练不太适合（多层隐藏层时）。

（2）$Tanh$函数：

- **优点：** 函数输出以（0,0）为中心，收敛速度相对于`Sigmoid`更快。
- **缺点**： `tanh`并没有解决`sigmoid`梯度消失的问题

（3）$Relu$函数：

- **优点：** 相比`Sigmoid`/`Tanh`函数，使用梯度下降法时，收敛速度更快； 相比`sigmoid`/`tanh`函数， `Relu`只需要一个门限值，即可以得到激活值，计算速度更快
- **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，这样会导致神经元不能更新参数，也就是神经元不学习了，这种现象叫做“Dead Neuron死神经元” 。

（4）$Leaky\ Relu$函数：

- **优点**： 收敛速度要比`Sigmoid`和`tanh`快很多 ，有效的缓解了梯度消失问题 ；且**与**`Relu`**相比**，对于小于0的值，梯度也不会永远为0，使得负值的信息不回全部丢失。 解决了`Relu`函数进入负区间后，导致神经元不学习的问题。
- **缺点**：在基于梯度的学习会比较慢。

2. 有什么方法可以实现传递过程中不激活所有节点?

- 可以将对应的节点的权重设置为0，这样节点就不会被激活。
- 也可以使用$Relu$函数或者$LeakyRelu$函数作为激活函数，使得输入$x<0$时不被激活或者激活非常小。

3. 梯度消失和梯度爆炸是什么？可以怎么解决?

- 目前优化神经网络的方法都是基于BPNN的，即根据损失函数计算的误差通过**梯度反向传播**的方式，指导深度网络权值的更新优化。其中将误差从末层往前传递的过程需要**链式法则**的帮助，因此反向传播算法可以说是梯度下降在链式法则中的应用。而链式法则是一个**连乘的形式**，所以当层数越深的时候，梯度将以**指数**形式传播。
- **梯度消失**：在通过梯度**反向传播**的方式对权值进行更新时，得到的**梯度值接近0**，发生**梯度消失。**当梯度消失发生时，接近于输出层的隐藏层由于其梯度相对正常，所以权值更新时也就相对正常，但是当越靠近输入层时，由于梯度消失现象，会导致**靠近输入层**的隐藏层权值更新缓慢或者更新停滞。这就导致在训练时，只等价于后面几层的浅层网络的学习。
- **梯度爆炸**： 一般出现在**深层网络**和**权值初始化值太大**的情况下。在深层神经网络或循环神经网络中，**误差的梯度可在更新中累积相乘。**如果网络层之间的**梯度值大于 1.0**，那么**重复相乘会导致梯度呈指数级增长**，梯度变的非常大，然后导致网络权重的大幅更新，并因此使网络变得不稳定。
- **解决方法**：
  - **重新设计网络模型**：梯度爆炸可以通过重新设计更少层数的网络来解决，使用更小的尺寸对网络训练也有好处；另外也许是学习率过大导致的问题，减少学习率也可以有助解决。

- **梯度剪切**：对于梯度爆炸问题，可以设定一个**剪切阈值**。更新梯度的时候，如果梯度超过这个阈值，那么就将其强制限制在这个范围之内。
- **权重正则化**：主要用于限制过拟合。如果发生梯度爆炸，权值会变的非常大，若用正则化项来限制权重的大小，可以在一定程度上防止梯度爆炸。比较常见的是 **L1 正则**和 **L2 正则**。
- **选择`Relu`等激活函数**：`Relu`函数的导数在正数部分是恒等于1的，因此在深层网络中使用`Relu`激活函数就不会导致梯度消失和爆炸的问题。
- **批规范化（Batch normalization）**：通过批规范化操作将输出信号`x`规范化到均值为`0`，方差为`1`，保证网络的稳定性。这样消除了权重参数`w`放大缩小带来的影响，进而解决梯度消失和爆炸的问题。