BP神经网络

> 现如今，各大媒体、企业乃至政府，都在大力发展AI技术，人工智能风起云涌，最火的深度学习，根本上就是神经网络技术。虽然现在有了Tensorflow、pytorch等方便搭建神经网络的工具，但是出于学习目的，初学者复现基本神经网络模型，仍然是一件十分有意义的工作。本文将以周志华《机器学习》为基础，复现一个简单的神经网络模型。

简介

学过逻辑回归的小伙伴们应该都知道逻辑回归的原理，通过一个线性函数计算输入，然后通过S型函数将其映射至0到1的连续区间上，这个输出可以被视为一种概率，所以根据这个概率可以推算样本所属的类别，完成一个分类任务。而训练这个分类器的过程，其实就是根据梯度调整我们的线性函数权值的过程。
吴恩达在机器学习课程中讲过，我们可以把神经网络看作一个个的逻辑回归的堆叠，自然而然，通过调整不同神经元的权值，就能够训练得到一个神经网络模型。通过改变神经元的激活函数（就像逻辑回归里面的S型函数），可以完成分类或者回归的任务。这样来理解神经网络，就要简单许多了。
神经网络需要两个过程，一个是前向传播，一个是误差反传（error backpropagation），后者是整个神经网络模型的核心，利用误差反传的思想，不断修正权值，从而达到更高的预测（分类）精度。
本文将以经典三层神经网络为例进行代码复现。

前向传播

首先，我们需要构建一个神经网络类，这个类是我们的三层神经网络的框架。为了更好更快的实现我们的算法，需要导入numpy库方便进行矩阵计算。
类中自然要有构造器，这个构造器需要传入输入层的节点数、隐层的节点数、输出层的节点数以及学习率。

```
import numpy as np
class NeuralNetwork:

    def __init__(self, inputNode, hiddenNode, outputNode, lr):
        """
        构造器，定义网络各层的节点数目与学习率，并初始化权重矩阵
        :param inputNode: 输入层的节点数
        :param hiddenNode: 隐层的节点数
        :param outputNode: 输出层的节点数
        :param lr: 学习率
        """
        self.inputNode = inputNode
        self.hiddenNode = hiddenNode
        self.outputNode = outputNode
        self.lr = lr
        #初始化权重矩阵，矩阵形状与各层节点个数有关
        self.weightsInToHidden = np.random.normal(0.0, self.hiddenNode**-0.5,
                                        ( self.hiddenNode, self.inputNode))

        self.weightsHiddentoOut = np.random.normal(0.0, self.outputNode**-0.5,
                                        (self.outputNode, self.hiddenNode))

        self.b1 = 0.5*np.random.rand(hiddenNode,1)-0.1
        self.b2 = 0.5*np.random.rand(outputNode,1)-0.1
```

在实例化对象的时候，需要初始化权重矩阵，这个权重矩阵的大小形状，需要由节点数来控制。这里b矩阵的初始化，按照吴恩达的教程来做的。
下面开始为前向传播做准备了。首先要定义一下我们的激活函数，这个激活函数我们就选择sigmoid函数吧。sigmoid函数的长这个样子： $\sigma = 1/(1+e^{-x})$

```
    def __activeFunction(self, arr):
        """
        激活函数，sigmoid函数
        :param arr: 输入向量
        :return: 返回计算结果
        """
        fun = 1/(1 + np.exp(-arr))
        return fun
```

下面正是进入正向传播，首先变化一下输入矩阵，让它变成每列是一个样本的形式，之后，按照权重与输入相乘，加上bias，并且直接调用激活函数计算。我这里是为了处理回归问题，所以输出层采用的是线性函数，如果想做分类，可以更换输出层的激活函数，变成sigmoid函数即可。

```
    def __feedForward(self, inputList):
        """
        前向传播算法
        :param inputList: 训练集列表，每个样本是一行
        :returns:
            inputs: 输入训练集矩阵
            hiddenOutputs: 隐层输出
            outOutput: 输出层的输出
        """
        inputs = np.array(inputList, ndmin=2).T #将输入训练集变成每一列为一个样本

        hiddenInputs = np.dot(self.weightsInToHidden, inputs).transpose() + self.b1.transpose() #隐层输入计算
        hiddenOutputs = self.__activeFunction(hiddenInputs).transpose() #隐层输出计算
        outInputs = (np.dot(self.weightsHiddentoOut, hiddenOutputs).transpose()\
                     + self.b2.transpose()).transpose() #计算输出层的输入
        outOutput = outInputs #计算输出层的输出

        return inputs, hiddenOutputs, outOutput
```

上面的代码，就完成了前向传播的全部过程，这里需要注意的是，矩阵计算需要注意形状大小，否则就很容易出现错误。

误差反向传播

这部分是整个BP神经网络的核心，也是它叫BP的原因所在(back propagation)。这里的公式不再细讲，所用到的知识，就是我们高等数学里的链式法则求导。使用梯度下降法实现loss函数最优。
这个loss函数是一个误差平方和的概念，也就是网络输出值与实际结果的误差平方和，我们的训练就是为了让我们的网络能够以一个极小的误差做出预测。
感兴趣的小伙伴可以下去自行推导一下，推导的过程中可能会有点后悔当初高数没有好好学哈哈哈哈。

```
    def __backPropagation(self, inputs, hiddenOutputs, outOutput, targetList):
        """
        BP误差反向传播算法
        :param targetList: 训练集的真实结果
        :param hiddenOutputs: 隐层输出结果
        :param outOutput: 输出层输出结果
        :return:
            outputError: 输出误差大小
        """
        n = len(targetList)
        targets = np.array(targetList, ndmin=2)


        #计算误差反向传播，由于是回归问题，因此把输出层激活函数设为x
        outputError = targets - outOutput
        hiddenError = np.dot(self.weightsHiddentoOut.transpose(), outputError) * hiddenOutputs*(1-hiddenOutputs)

        dw2 = np.dot(outputError, hiddenOutputs.transpose())
        db2 = np.dot(outputError, np.ones((n, 1)))

        dw1 = np.dot(hiddenError, inputs.transpose())
        db1 = np.dot(hiddenError, np.ones((n, 1)))


        #权值梯度下降
        self.weightsHiddentoOut += dw2 * self.lr
        self.weightsInToHidden += dw1 * self.lr

        self.b1 += db1*self.lr
        self.b2 += db2*self.lr

        return outputError
```

模型的训练与预测

上文已经定义了BP神经网络的两个核心，前向与反向传播，那么我们的训练函数，只需要对控制我们的最大轮次即可。先后调用我们的正向传播与反向传播，就完成了一个轮次的训练。这个训练的轮次，不宜过大也不宜过小，过大过小都会影响我们的模型优劣，使误差平方和无法达到最优。
这里我使用了tqdm模块，做一个进度条，来观察我们训练的进度，同时也设置了一个每间隔多少轮show一次误差平方和的参数，这些都是为方便我们调整模型。

```
    def train(self, inputList, targetList, maxEpcho, showTime = -1):
        """
        按照最大迭代次数训练网络
        :param inputList: 训练集列表，每个样本是一行
        :param targetList: 训练集的真实结果
        :param maxEpcho: 最大迭代次数
        :param showTime: 间隔几轮显示一次均方误差
        :return:
            historySSE: 每轮训练误差平方和列表
        """
        historySSE = [] #记录每一伦的误差平方和
        for epcho in tqdm(range(maxEpcho)):

            inputs, hiddenOutputs, outOutput = self.__feedForward(inputList) #正向传播
            outputError = self.__backPropagation(inputs, hiddenOutputs, outOutput, targetList) #误差反向传播
            SSE = sum(sum(outputError**2)) #计算误差平方和
            historySSE.append(SSE)

            if showTime < 0:
                pass
            elif epcho % showTime == 0:
                print("epcho %d   SSE="%epcho, SSE)

        return historySSE
```

训练完成后，我们就可以进行预测了，这个函数异常简单，就是一个输入输出的过程。

```
    def predict(self, inputList):
        """
        预测输出结果
        :param inputList: 训练集列表
        :return:
            outOutput: 预测结果
        """
        inputs, hiddenOutputs, outOutput = self.__feedForward(inputList)
        return outOutput
```

波士顿房价数据集的实验

上文已经构建了我们的模型，那么我们下面只需要在主程序中使用上文的模型即可。这里采用UCI标准数据集中波士顿房价数据集。

```
if __name__ == '__main__':
    data = load_boston()['data']
    target = load_boston()['target']

    #划分数据集
    trainData, testData, trainTarget, testTarget = train_test_split(data, target, test_size=0.33, random_state=42)

    #归一化数据集
    scaler = MinMaxScaler( )
    scaler.fit(trainData)
    trainData = scaler.transform(trainData)
    trainTarget = (trainTarget - trainTarget.mean()) / (trainTarget.max() - trainTarget.min())

    scaler = MinMaxScaler( )
    scaler.fit(testData)
    testData = scaler.transform(testData)
    testTarget = (testTarget - testTarget.mean()) / (testTarget.max() - testTarget.min())

    #开始训练
    nn = NeuralNetwork(13, 20, 1, 0.0032)
    historySSE = nn.train(trainData, trainTarget, 60000, showTime = 10000)

    #绘制出SSE变化图
    plt.figure(figsize=(50, 10))
    plt.plot(range(1, 60001), historySSE)
    plt.show()

    #进行预测
    predict = nn.predict(testData)

    #绘制预测与真实价格折线图
    plt.figure(figsize=(50, 10))
    plt.plot(range(167), predict.T)
    plt.plot(range(167), testTarget)
    plt.show()
```

最后，让我们看一下效果吧！