## 对反向传播的理解

我认为反向传播可以大致理解为用多个线性函数复合一个非线性函数去拟合一类问题的过程，根据这一类问题中一个样本的正向传播计算得到的结果，再根据预测的结果与实际结果的差异反向传播得到预测的结果对这个函数的影响，从而提高下一次结果预测的准确率。这个过程的原理有些类似先验概率的求法。

而关于反向传播的算法，我觉得最重要的一点就是链式求导的实现，导数的求法并不是最重要的，关键在于实现程序的过程能不能把“链式”给实现出来。很多时候会容易出错，想借此机会梳理一下此次反向传播的实现过程。

反向传播要注意层数的划分，必须明确输入层、隐藏层、输出层，还要主义输入层和隐藏层要“秘密”地在每一层新增一个神经元，用来调节 bias，如果输入值的个数是 1，那么就要两个输入层神经元，其中①号的神经元的值为这个输入值，②号神经元的值为 1，②号神经元与下一层的每个神经元之间的 weight 就相当于下一层每个神经元的 bias，bias = $weight_{2,i}$*1,这样就可以通过 bias[i]+= deltas[i]*learn 来调整 bias 了，同样的，对每个隐藏层都新增一个神经元也是这样的目的。

跟我的程序写的差不多，我觉得反向传播过程大致可以分为：

1. 初始化神经网络（设置输入层、隐藏层、输出层个数和初始值);
2. 向前传播得到预测结果，

Output[h]=fit_function((Σ$_{i=0}^{输入层神经元个数}$weight[i][h]*input[i])+bias[h])

3. 得到 deltas:

①输出层到最后一层隐藏层：self.output_deltas = ∂Error/∂(output_w)

output_deltas[o]=fit_function(output_cell[o])*(label[o]-output_cells[o])

②隐藏层之间的 deltas：self.hidden_deltases = ∂Error/∂(weight) weight 是指隐藏层 weight 中的元素。

hidden_deltases[k][o]=fit_function_deriv(hidden_result[k][o])*(deltas[i]*weight[o][i]

4. 更新 weight

Weight[i][o] += deltas[o]*input[i]*learn

5. 更新 bias

bias[i]+= deltas[i]*learn

以上基本上就是我理解的 bp 神经网络的过程和对它的一些看法。

## 对交叉熵和和损失函数的理解

在拟合 sinx 时使用的是损失函数进行拟合的，而在分类任务中使用的则是交叉熵函数。我认为损失函数和交叉熵函数的差别在与他们的输出值与实际结果的比较的衡量标准不同。损失函数要保证两者差别最小，而交叉熵函数则要保证概率最大。损失函数是 0.5*（label[i]-output[i]）^2,是用来拉近正确值与输出值的，也就是让正确值和输出值的差越小越好，而并不是一定要输出正确的结果，这种损失函数比较适合拟合函数，用来逼近输出值。但不适合使用分类任务，交叉熵是计算几个事物的概率误差，适合于 one-hot 编码的分类任务，也就是说，交叉熵其目的在于让“大者概率更大，小者概率更小”，而不是简单的趋向于结果值，所以它的正确输出可以是一个比 1 大很多的值和几个跟 1 差不多大的值组成，然后再进行 softmax 概率化。而损失函数却不能容忍这一点，这就是损失函数在分类任务中的局限性。而交叉熵使用 softmax 函数，使用 exp 指数化参数，很容易的就可以保证“大者概率更大，小者概率更小”这一目的。