Pytorch神经网络

神经网络简介

神经元

在深度学习中，必须要说的就是神经网络，或者说是人工神经网络（artificial neural network）。神经网络是一种人类受到生物神经细胞结构启发而研究出的算法体系。

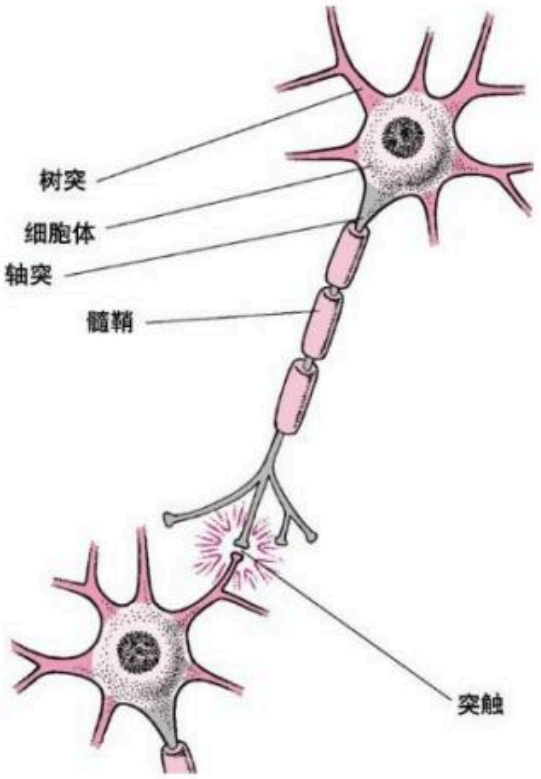


图 1 生物神经元结构图

如图 1 所示是一幅生物神经元的结构图，神经细胞使用的是化学信号传递，但是有机化学分子比较复杂，目前为止，人类并不了解这些化学分子具体承载的信息，然而人类通过从这种神经细胞之间的刺激来传递信息的方式中获得启发，从而设计了网络状的处理单元。

神经网络这个名字容易让人觉得特别神秘，不像我们接触过的程序算法那样直观，在编程的时候我们常用到的都是一些加减乘除、循环、分支、比大小、读写等等，使用这些基本步骤就能够完成一个明确的目标任务，然而神经网络和这种直观的方式还真有些不同。

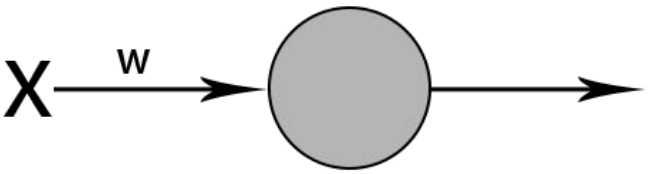


图 2 最简单的神经元

图 2 就是一个最简单的神经元，有一个输入，一个输出，中间是一个参数，然后节点表示进行某种运算，所以它表达的含义跟一个普通函数没什么区别。不过需要注意的是，我们目前使用的神经元内部的运算通常有两个部分组成，第一部分是所谓的“线性模型”，可以把它理解为一

个简单的包含加减乘除的函数，假设为 $f(x) = 2x + 1$，它是图 2 中圆圈内的第一个部分。图 2 中圆圈内运算的另外一部分是“激活函数”，这一部分也不是很复杂，就是将第一部分线性模型的结果再通过一个“激活函数”计算，这个函数通常是非线性的，例如 $f(x) = \frac{1}{1+e^{-2x+1}}$,其中 $2x + 1$ 是第一部分的线性函数。我们可以看出它虽然和普通的函数没什么区别，但这种方式的确是神经网络工作的原理。

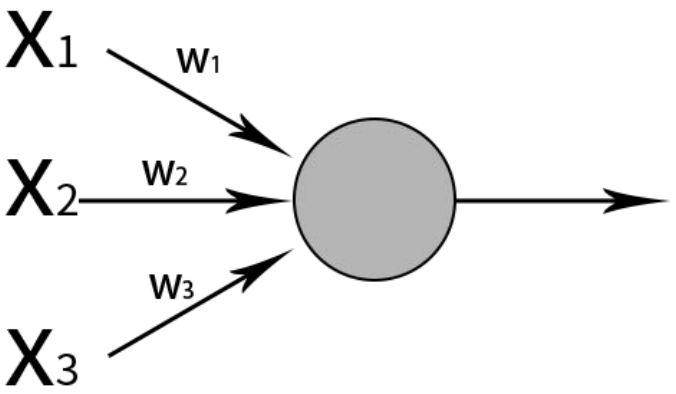


图 3 n 维的函数 f(x)

我们接着看图 3，输入 x 可以是一个一维向量，也可以是三维向量，当然也可以是一个更多的 n 维向量，对于 n 维向量输入项的神经元 $f(x)$ 来说，这个函数就是：

$$f(x) = wx + b$$

其中 x 是一个 $n \times 1$ 的矩阵也就是 n 维列向量，而 w 是一个 $1 \times n$ 的权重矩阵，b 是偏置项。这个过程是在计算什么？权重矩阵和偏执项到底是什么？我们举一个例子来说明，假设 x 是一个这样的矩阵：

$$\begin{pmatrix} 1 \\ 180 \\ 70 \end{pmatrix}$$

这个特征向量有三个维度，比如说第一行代表性别，数值 1 代表男生，数值 0 代表女生，第二行代表身高 180cm，第三行代表体重 70kg，这是一个多维度的描述，可能是描述人的体重情况。然后根据不同的人有不同的特征向量，w 是一个 $1 \times n$ 的矩阵，它表示权重的概念，就是表示每一项的重要程度，例如 [0.1 0.03 0.06]，b 是偏置项，它是一个实数值，假设在这里 b 就是 1×1 的矩阵 0，整个函数则表示为 w 与 x 进行内积操作，乘出来是一个实数，然后加上 b，即

最终的结果是 9.7，那么我们在这定义一个区间(6,13)，只要结果在这个区间内则体重正常。不在区间的话就通过一些算法计算，进而调整 w 与 b，使得计算结果尽量落在这个区间，这样做的前提是我们给出的数据都是正常体重的数据，经过大量数据的演算推导，我们就能够获得精确度极高的 w 和 b 的值，从而确定 $f(x)$，如此一来，我们就可以使用这个确定的函数 $f(x)$ 来检验新的体重是否标准了。

在以后训练神经网络的时候，这些 w 与 b 参数，是我们最终想要得到的最有价值的东西，有这些参数之后我们就可以拿新的输入 x，通过计算来预测结果。

通常，我们在训练神经网络的时候，会需要大量的数据样本，这些数据样本里面我们已知的有输入 x，然后还会有已经知道的输出 $f(x)$ 或 y ,这个输出通常称为标签，例如一张猫的图片，输入就是这张图片的像素矩阵，输出就是图片的名称-猫（当然可以拿数字来代替，计算机并不清楚文字猫和数字 0 的区别）。

这里我们引入一个新的概念，叫做损失函数，我们简单解释一下，在训练神经网络的时候，一开始，我们会初始化 w 与 b 参数，假设定义它们的矩阵里面全是接近 0 的浮点小数，那么输入与其计算后会得到结果 $f'(x)$，但是这个数据真实的标签却是 $f(x)$，我们会定义一个损失函数，它代表了计算输出的标签与真实数据的标签的差距，通常把这个函数写作为 Loss：

$$Loss = \sum_{i=1}^{n} |wx_i + b - y_i|$$

上式举了一个损失函数的例子，$wx_i + b$ 为输入通过线性模型计算得出的 $f'(x)$，

$|wx_i + b - y_i|$ 代表求 $f'(x)$ 与 $f(x)$ 之间差的绝对值，$\sum_{i=1}^{n} |wx_i + b - y_i|$ 代表将从 1 开始到 n 所有差的绝对值加起来，那么这个函数有什么用途呢？我们训练神经网络的最终目的是得到合适的 w 与 b，那么这个损失函数足够小的情况下，就代表神经网络计算出的结果与真实的结果差距足够小，也就说明我们的 w 与 b 越可靠，至于怎么通过这个函数去不断的调整 w 与 b 呢？我们后面部分会讲解。

激活函数

激活函数(activation function)，也叫作激励函数，在第一小节神经元中，我们提到过这个名词，也大概说明了一下什么是激活函数，它是在神经元中跟随 $f(x) = wx + b$ 之后加入的非线性因素，激活函数在神经元线性模型之后，如下图 4 红色部分：

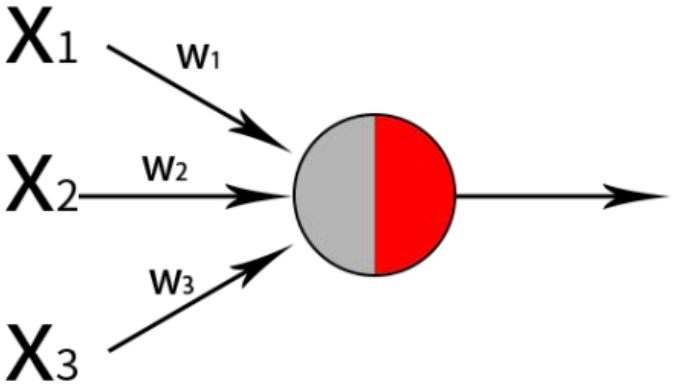
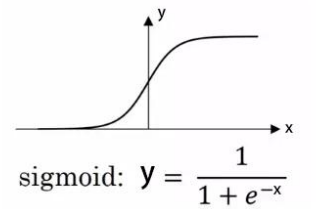
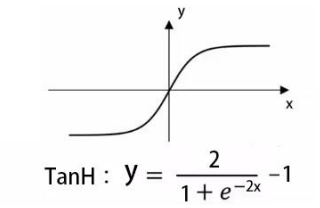
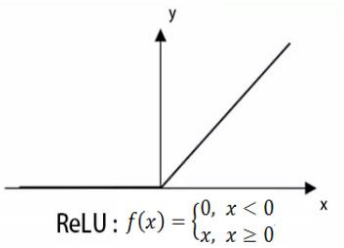


图 4 激活函数

那么解释一下什么是非线性，生活中的各种事物抽象为数学模型后几乎都是非线性，举个例子，理想情况下房子越大，价格越贵，这里的面积与价格可以视作线性关系，但是真实情况下，房价不仅受到面积的影响因素，还会受到地理位置、时间、楼层等等因素的影响，那么这几种因素与房价就不是线性关系了。通常神经元的串联和并联叠加构成了神经网络，如果都是线性模型的叠加，那最终整个网络也是线性的，也就是矩阵相乘的关系，但是其中加入了激活函数，那么叠加之后的神经网络理论上就可以构成任意复杂的函数从而解决一些复杂问题。下面我们给出神经网络中常用到的三种激活函数：







可以看到第一种 sigmoid 函数，是将线性模型的计算结果投射到 0 到 1 之间，第二个 TanH 函数是将线性模型的计算结果投射到-1 到 1 之间，最后一个 ReLU 是将线性模型计算结果小于 0 的部分投射为 0，大于等于 0 的部分投射为计算结果本身。

1.3 神经网络

学习完前两小节之后，我们大概能够想象出来一个神经网络的样子，通俗的说，就是神经元首尾相接形成一个类似网络的结构来协同计算，这个算法体系被称为神经网络，见图 5.

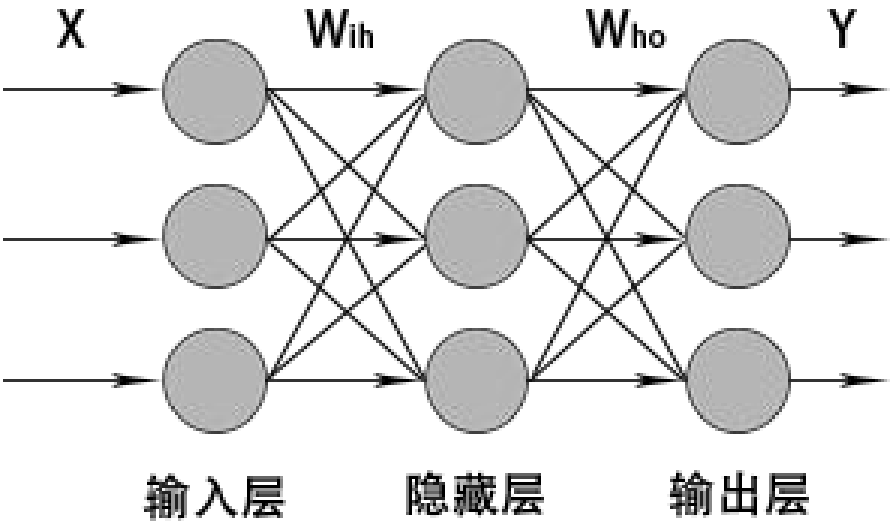


图 5 神经网络结构图

图 5 是一个非常简单的神经网络结构，在神经网络中通常会分为：输入层、隐藏层、输出层。输入层在整个网络最前端，图中为最左侧（也有自下而上的结构示意图），它直接接受输入的向量，输入层通常不计入层数，隐藏层这一层可以有多层，在比较深的网络中，隐藏层能达到数十上百层，输出层是最后一层，用来输出整个网络计算后的结果，这一层可能是比较复杂的类型的值或者向量，根据不同的需求输出层的构造也是不同的。

神经元就是通过这种结构进行数据传递，数据经过前一个神经元的计算输出给下一层的神经元当做输入，因为前一层的神经元节点连接了下一层的所有节点，因此这种前后层相互连接的网络也叫作全连接神经网络，这是一种非常常见的网络结构。

神经网络的工作过程

前向传播(forward)

在前面的学习中，我们介绍了神经网络的基本结构，还有神经元的计算方式，本节开始我们继续深入讲解神经网络的工作过程。在前面我们接触过了一种简单的神经网络结构，叫做全连接神经网络，同时，这种神经元从输入层开始，接受前一级输入，并输出到后一级，直至最终输出层，由于数据是一层一层从输入至输出传播的，也叫作前馈神经网络。

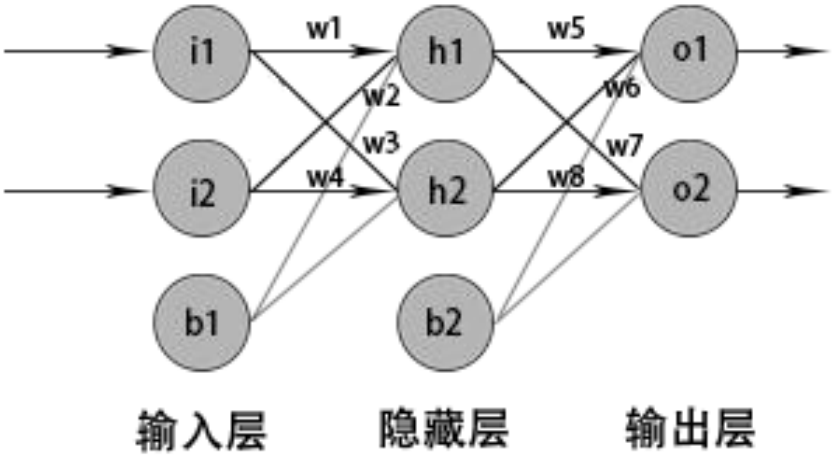


图 6 某神经网络例子

接下来我们看图 6 所示的神经网络，这是一个简单的神经网络结构，$i1$ 与 $i2$ 分别是两个输入，隐藏层有两个神经元节点 $h1$ 与 $h2$，偏置项 $b1$ 与 $b2$，输出层也有 2 个神经元节点 $o1$ 与 $o2$，那么前向传播过程中，隐藏层神经元节点线性部分计算有、

D

假设激活函数为 sigmoid，那么第一个隐藏层节点输出为

$$out_{h1} = \frac{1}{1 + e^{-l_{h1}}}$$

同理

$$out_{h2} = \frac{1}{1 + e^{-l_{h2}}}$$

输出层 o1 节点假设激活函数为 sigmoid（这里使用激活函数的作用往往是根据特定的需求，比如多分类问题需要用到 softmax 函数），那么、

D

同理、

D

这样整个网络的前向传播输出

$$out_{o1} = \frac{1}{1+e^{-l_{o1}}}$$
$$out_{o2} = \frac{1}{1+e^{-l_{o2}}}$$

这时如果将我们的参数都赋值常数带入进去，然后设定一组真实的数据（其中包含输入 $i1$ 与 $i2$，也包含一个与之对应的输出 out），输入之后会计算得到一个计算结果，然后将计算结果与我们设定的真实结果 out 做比较，会出现差距，那么损失函数我们上一章已经提到了，假设损失函数是

$$E_{total} = \frac{1}{n}\sum_{i=1}^{n}(target - output)^2$$

其中 target 代表数据样本的标签,是理应输出的值,output 代表神经网络计算的输出,那么这两者之间的差值就能代表当前神经网络计算的误差,这里取其平方再加和除 n 的意义是，实际输出与我们期望的输出差距越大，代价越高，然后取其均方差，则有

$$E_{o1} = \frac{1}{2}(target - out_{o1})^2$$
$$E_{o2} = \frac{1}{2}(target - out_{o2})^2$$
$$E_{total} = E_{o1} + E_{o2}$$

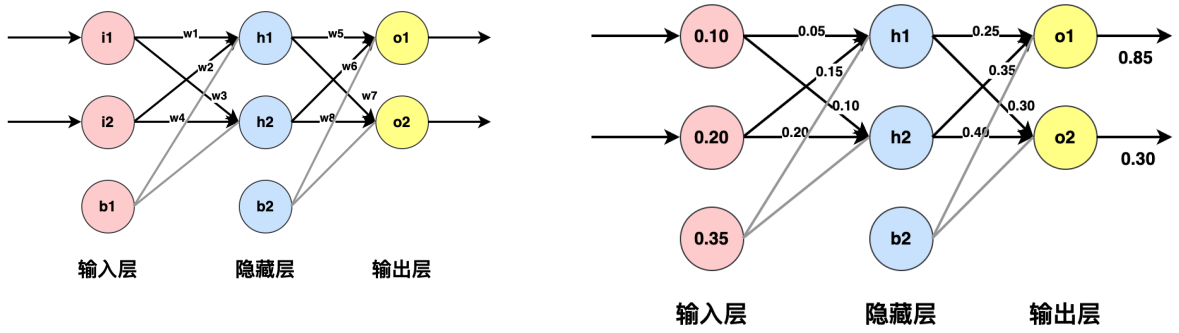
有了这个损失函数我们就可以学习接下来的反向传播了！

反向传播(backward)

现在我们来讲一下反向传播，顾名思义，反向传播算法是利用损失函数进而从输出到输入方向传播达到调整参数的目的，它的存在主要是解决深层（多个隐藏层）神经网络的参数更新问题，反向传播算法就是梯度下降应用了链式法则，梯度下降这一概念我
们下一小节会讲到，链式法则是微积分中的求导法则，用于复合函数的求导，在神经网络中只要有了隐藏层，那么隐藏层的权重参数与损失函数会构成复合函数，因此使用链式法则解决复合函数求导问题达到调整权重参数的目的。

具体链式法则的计算方式大家可以查阅高数资料了解。可能在这里出现了一些难理解的专有名词，那么举个例子来帮助我们更直观的理解，例如输入是**小红**，隐藏层是**小蓝**，输出层是**小黄**，根据神经网络的计算方式，最终会产生损失函数，那么利用梯度下
降算法，可以将损失函数的反馈直接反馈到输出层
小黄，尽而调整输出层**小黄**前面的参数矩阵，但是隐藏层**小蓝**不能得直接反馈而不能优化前面的参数矩阵，这时候利用链式法则，将**小蓝**与损失函数形成复合函数从而让误差反馈也能调整隐藏层**小蓝**前面的权重矩阵。最后迭代几次后最终误差会降低到最小。梯度下降具体是做什么的我们先不用管，知道它是利用损失函数进而调整权重参数最终使损失函数尽量小就可以了。

接下来我们将上一小节中的神经网络拿过来,并将输入、输出、权值、偏置都赋予实数,来演示一下反向传播的过程,具体实数赋值如下:



赋值后的神经网络

上图中，左侧是原来我们搭建的神经网络，右侧是将输入赋值 0.10 与 0.20，隐藏层权重为 0.05、0.15、0.10、0.20，偏置为 0.35，输出层权重为 0.25、0.35、0.30、0.40，偏置为 0.50，最终赋值输出为 0.85、0.30，注意这个输出是我们预设好的标签，
就是我们希望输入 0.10 与 0.20 后，通过神经网络计算，最终输出 0.85 与 0.30。 通过上一节的前向传播，我们得到了前向传播中各个节点的计算公式，将实数带入公式我们可以得到：

隐藏层第一个节点线性部分：$l_{h1} = 0.05 \times 0.10 + 0.15 \times 0.20 + 0.35 = 0.385$

隐藏层第二个节点线性部分：$l_{h2} = 0.10 \times 0.10 + 0.20 \times 0.20 + 0.35 = 0.400$

隐藏层第一个节点输出：$out_{h1} = \frac{1}{1+e^{-0.385}} = 0.595078474$

隐藏层第二个节点输出：$out_{h2} = \frac{1}{1+e^{-0.4}} = 0.598687660$

同理，输出层线性部分：

$l_{o1} = 0.25 \times 0.595078474 + 0.35 \times 0.598687660 + 0.50 = 0.8583102995$
$l_{o2} = 0.30 \times 0.595078474 + 0.40 \times 0.598687660 + 0.50 = 0.9179986062$

输出层输出：

D

损失函数计算结果：

$$E_{o1} = \sum \frac{1}{2}(0.85 - 0.702307507)^2 = 0.010906536$$
$$E_{o2} = \sum \frac{1}{2}(0.30 - 0.714634132)^2 = 0.085960731$$
$$E_{total} = 0.010906536 + 0.085960731 = 0.096867268$$

到这之后可以看到我们的总误差的数值，那么接下来进行反向传播算法，在反向传播算法里面应用了很多导数的知识，现在我们重点放在过程上，在下一节中我们再详细讲解梯度下降。

反向传播：

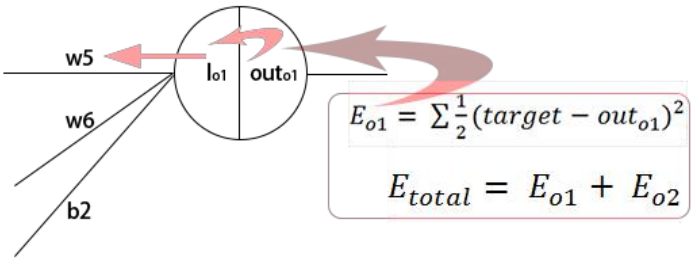


图 8 反向传播示意图

图 8 是反向传播示意图，对于权重参数 $w5$ 来说，我们想知道它的变化对损失函数的影响有多大，是根据最后的总损失函数反馈到 out_{o1}，然后由 out_{o1} 反馈到 l_{o1}，最后由 l_{o1} 反馈到 $w5$，那么我们对其求偏导并且根据链式法则有：

$$\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial l_{o1}} \times \frac{\partial l_{o1}}{\partial w5}$$

那么：

$$E_{total} = \frac{1}{2}(0.85 - out_{o1})^2 + \frac{1}{2}(0.3 - out_{o2})^2$$

对上式 out_{o1} 求偏导，根据求导公式得：

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 \times \frac{1}{2}(0.85 - out_{o1}) \times -1 = 0.702307507 - 0.85 = -0.147692493$$

接着计算 $\frac{\partial out_{o1}}{\partial l_{o1}}$：

$$out_{o1} = \frac{1}{1 + e^{-l_{o1}}}$$

这儿是对sigmod函数进行求导,我们具体可以通过配项推导一下,此处省略推导过程（如果感兴趣可以去找一下相关资料手动推一次）：

$$\frac{\partial out_{o1}}{\partial l_{o1}} = out_{o1} \times (1 - out_{o1}) = 0.702307507 \times (1 - 0.702307507) = 0.209071672$$

接着往下计算 $\frac{\partial l_{o1}}{\partial w5}$ ：

$$l_{o1} = w5 \times out_{h1} + w6 \times out_{h2} + b2$$

$$\frac{\partial l_{o1}}{\partial w5} = 1 \times out_{h1} \times w5^{1-1} + 0 + 0 = 0.595078474$$

最后三项相乘得出 $\frac{\partial E_{total}}{\partial w5}$

$$\frac{\partial E_{total}}{\partial w5} = -0.147692493 \times 0.209071672 \times 0.595078474 = -0.018375021$$

到这之后我们可以根据梯度下降的算法来更新 w5 权重了：

$$w_5^+ = w_5 - \eta \frac{\partial E_{total}}{\partial w5} = 0.25 - 1 \times (-0.018375021) = 0.268375021$$

上述的式子是梯度下降的算法，η 是学习率，就是说梯度下降的步幅，是由工程师凭借经验或者测试而确定的数值，在这我们先认识一下这些名词，那么 w_5 就被更新为 w_5^+。同理我们也可以将 $w6$，$w7$，$w8$ 更新完成。当我们使用更新完之后的参数再带入神经网络去计算时，会发现最终的输出与真实的输出已经更接近了。这就是梯度下降算法的作用，是不是迫不及待的想了解梯度下降了？让我们先把隐藏层的权重是如何通过反向传播算法更新的理解清楚。

对于隐藏层更新权重时，例如更新 $w1$，使用的方法基本与前文更新 $w5$ 是一致的，从 $out_{o1}, l_{o1}, w5$ 三处分别求偏导再求其乘积，但有区别的是，更新 w1 时候，损失函数传递方式变为下图 9 所示：

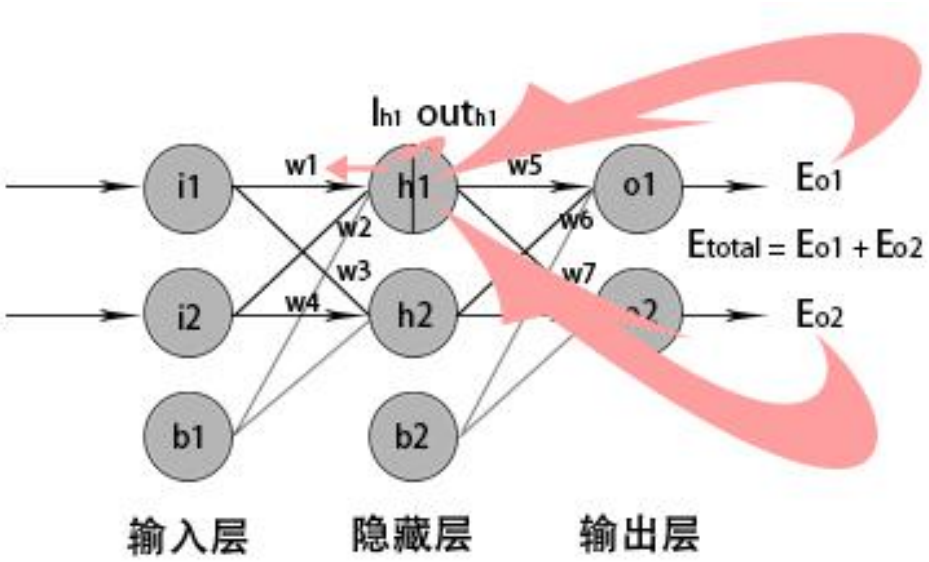


图 9 损失函数传递方式变化

那么就有损失函数对 $w1$ 权重求偏导：

$$\frac{\partial E_{total}}{\partial w1} = \frac{\partial E_{total}}{\partial out_{h1}} \times \frac{\partial out_{h1}}{\partial l_{h1}} \times \frac{\partial l_{h1}}{\partial w1}$$

由上图中的传递方式可以看到，out_{h1} 会接受有E_{o1} ,与 E_{o2} 两个地方传来的误差损失。
那么
$\frac{\partial E_{total}}{\partial out_{h1}}$ 就可以拆为：

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

继续往下计算 $\frac{\partial E_{o1}}{\partial out_{h1}}$，主要是利用链式法则将隐藏层与输出层链接起来计算：

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial l_{o1}} \times \frac{\partial l_{lo1}}{\partial out_{h1}}$$

那么可以利用输出层的计算结果($\frac{\partial E_{o1}}{\partial out_{o1}}$ 与 $\frac{\partial E_{total}}{\partial out_{o1}}$ 结果相等)：

$$\frac{\partial E_{o1}}{\partial l_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial l_{o1}} = -0.147692493 \times 0.209071672 = -0.030878316$$

接着：

$$l_{o1} = w5 \times out_{h1} + w6 \times out_{h2} + b2$$

$$\frac{\partial l_{o1}}{\partial out_{h1}} = w5 = 0.25$$

最终：

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial l_{o1}} \times \frac{\partial l_{lo1}}{\partial out_{h1}} = -0.030878316 \times 0.25 = -0.007719579$$

同样我们可以计算出：

$$\frac{\partial E_{o2}}{\partial out_{h1}} = 0.025367174$$

那么：

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = -0.007719579 + 0.025367174 = 0.017647595$$

在这之后我们计算第二部分 $\frac{\partial out_{h1}}{\partial l_{h1}}$：

$$out_{h1} = \frac{1}{1 + e^{-l_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial l_{h1}} = out_{h1}(1 - out_{h1}) = 0.595078474 \times (1 - 0.595078474) = 0.240960084$$

然后计算第三部分 $\frac{\partial l_{h1}}{\partial w1}$：

$$l_{h1} = w1 \times i1 + w2 \times i2 + b1$$

$$\frac{\partial l_{h1}}{\partial w1} = i1 = 0.1$$

那么三项相乘：

$$\frac{\partial E_{total}}{\partial w1} = \frac{\partial E_{total}}{\partial out_{h1}} \times \frac{\partial out_{h1}}{\partial l_{h1}} \times \frac{\partial l_{h1}}{\partial w1}$$

$$\frac{\partial E_{total}}{\partial w1} = 0.017647595 \times 0.240960084 \times 0.1 = 0.000425237$$

最后梯度下降算法更新 $w1$：

$$w_1^+ = w_1 - \eta \frac{\partial E_{total}}{\partial w1} = 0.05 - 1 \times 0.000425237 = 0.049574763$$

同理也可以将 $w2$、$w3$、$w4$ 更新完成，最后将更新完的所有参数全部带入重新计算，得出新的输出，然后再根据损失函数、反向传播、梯度下降不停的去迭代更新参数，最终理论上输出的结果会无限接近真实结果，那么这一组参数就会变得非常有价值，上文中的理论推导都是基于简单的实数，并且网络结构简单，目的是让大家理解神经网路的工作过程，真实的环境中都

是使用矩阵批量计算的，并且已经有很多深度学习工具帮我们实现了计算过程，我们学会调用对应的函数就能便捷的实现计算过程。

训练神经网络

训练神经网络，也是使用**梯度下降方法**。神经网络模型的参数，与其隐藏层数量和神经元数量相关，学习率可以在经验值 1e-3 范围内进行探索。 一个合适的学习率能够让我们的损失函数在合适的时间内收敛到局部最小值。

损失函数（loss function）是用来评估模型的预测结果与真实结果不一致程度的函数。学完前面内容之后，我们已经知道了神经网络的工作过程，在前向传播计算完之后会得出一个输出，这个输出就是模型的预测值，真实值是在准备数据时就定义好的。

再来复习一遍，比如说 1000 张猫的图片，那么图片像素矩阵就是我们的输入，我们命名这些图为猫 1、猫 2...，这个就是手动打上去的标签，但是计算机是不能理解的，这样就需要将名字为猫的图片也转化为一个输出值矩阵，矩阵中以猫为名字的图片我们定义为 1，我们就知道将猫图片的像素矩阵输入之后经过计算得出的结果我们去和 1 比较，如果计算的预测值和真实值 1 不同，接着进行反向传播、梯度下降优化等等训练神经网络，这样我们最终会得到一个猫图片像素矩阵与数字 1 的映射关系。

这个过程中，每次计算的值是预测值，真实值是概率值，通常取值为1（100%）。预测值通常需要softmax函数转换为概率值。然后才是通过损失函数计算预测和真实之间的误差。

损失函数一个非负实值函数，通常会经过绝对值或者平方的方式让其变为非负，因为如果第一次计算差距是负数，第二次是正数，那么就会抵消了，变为没有误差，那肯定是不合适的。损失函数越小，模型就越健壮。

Pytorch搭建并训练神经网络

我们从PyTorch中经典的入门示例开始，从中学习神经网络构建和训练过程。

其中要学到并熟练掌握的是如下这些流程：

- 数据预处理
- 构建模型
- 定制模型损失函数和优化器
- 训练并观察超参数

下面我们就一步步分解这个过程，其中也会学习认识到一些pytorch为我们提供的框架内置对象和函数。初次接触可能还不是很适应，所以以先完成完整的模型训练流程为重。后续再根据任务需求，一步步的扩展pytorch的认知版图

数据预处理

模型训练用的样本，大部分都来自于外部文件系统。本次训练用的数据来自框架内置的数据集，所以代码没有泛化性。涉及到具体数据再做补充。

PyTorch 有两个用于处理数据的工具：`torch.utils.data.DataLoader` 和 `torch.utils.data.Dataset`。 `Dataset` 存储的是数据样本和对应的标签，`DataLoader` 把Dataset包装成一个可迭代的对象。

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt
```

本次的数据样本来自于Pytorch的TorchVision 数据集。

```
# 下载的数据集文件会保存到当前用户工作目录的data子目录中。
# 如果不想下载后就找不到了，建议修改root参数的值。
# 例如"D:\\datasets\\fashionMNIST\\"一类的绝对路径
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# 测试集也需要下载，代码和上面一样。
# 但参数train=False代表不是训练集(逻辑取反,就是测试集)
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

下一步就是对已加载数据集的封装，把 `Dataset` 作为参数传递给 `DataLoader`。这样，就在我们的数据集上包装了一个迭代器(iterator)，这个迭代器还支持**自动批处理、采样、打乱顺序和多进程数据加载等**这些强大的功能。这里我们定义了模型训练期间，每个批次的数据样本量大小为 64，即数据加载器在迭代中，每次返回一批 64 个数据特征和标签。

```
batch_size = 64

# 创建数据加载器
train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

# 测试数据加载器输出
for X, y in test_dataloader:
    print("Shape of X [N, C, H, W]: ", X.shape)
    print("Shape of y: ", y.shape, y.dtype)
    break
```

Shape of X [N, C, H, W]: torch.Size([64, 1, 28, 28])
Shape of y: torch.Size([64]) torch.int64

构建模型

为了在 PyTorch 中定义神经网络，我们创建了一个继承自 `nn.Module` 的类。我们在 __init__ 函数中定义网络层，并在 forward 函数中指定数据将如何通过网络。为了加速神经网络中的操作，我们将其移至 GPU（如果可用）。

```python
# 检验可以使用的设备
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"使用 {device} 设备")

# 定义神经网络模型
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
                # wx + b = [64,1,784] * [784,512] = 64,1,512
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

```
使用 cuda 设备
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

pytorch是通过继承 nn.Module 父类来实现自定义的网络模型。

在 `__init__` 中初始化神经网络层。

在 `forward` 方法中实现对输入数据的操作。

模型层分解

为了方便分解 FashionMNIST 模型中的各个层进行说明，我们取一个由 3 张大小为 28x28 的图像组成的小批量样本，看看当它们通过网络传递时会发生什么。

```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

torch.Size([3, 28, 28])

nn.Flatten

我们初始化 `nn.Flatten` 层，将每个28x28 大小的二维图像转换为 784 个像素值的连续数组（保持小批量维度（dim=0））。

```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

torch.Size([3, 784])

nn.Linear

linear layer线性层是一个模块，它使用其存储的权重和偏置对输入应用线性变换。

```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

torch.Size([3, 20])

nn.ReLU

为了在模型的输入和输出之间创建复杂映射，我们使用非线性激活。激活函数在线性变换之后被调用，以便把结果值转为非线性，帮助神经网络学习到各种各样的关键特征值。 在这个模型中，线性层之间使用了 `nn.ReLU`，其实还有很多激活函数可以在模型中引入非线性。

```
print(f"ReLU 之前的数据: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"ReLU 之后的数据: {hidden1}")
```

ReLU 之前的数据:

```
tensor([[-0.2496,  0.4432, -0.1536,  0.4439,  0.3256,  0.6594, -0.2536,  0.2348,
-0.1113,  0.0732, -0.0658,  0.6014, -0.6135, -0.4709,  0.3016,  0.1500,
```

```
0.0801,  0.3644, -0.6113,  0.4129],
[-0.7556,  0.1309, -0.2760,  0.3292,  0.5749,  0.6503, -0.1372,  0.3096,
0.1499, -0.0446, -0.1845,  0.2553, -0.6012, -0.3562, -0.0291,  0.1380,
0.2641,  0.2835, -0.5634,  0.1305],
[-0.3772, -0.0354, -0.3879,  0.1846,  0.5425,  0.5019,  0.3323,  0.3478,
0.1171,  0.1153, -0.3414,  0.1688, -0.4068,  0.0950, -0.0322,  0.1272,
0.1653,  0.1538, -0.8849,  0.1446]], grad_fn=<AddmmBackward0>)
```

ReLU 之后的数据:

```
tensor([[0.0000, 0.4432, 0.0000, 0.4439, 0.3256, 0.6594, 0.0000, 0.2348, 0.0000,
0.0732, 0.0000, 0.6014, 0.0000, 0.0000, 0.3016, 0.1500, 0.0801, 0.3644,
0.0000, 0.4129],
[0.0000, 0.1309, 0.0000, 0.3292, 0.5749, 0.6503, 0.0000, 0.3096, 0.1499,
0.0000, 0.0000, 0.2553, 0.0000, 0.0000, 0.0000, 0.1380, 0.2641, 0.2835,
0.0000, 0.1305],
[0.0000, 0.0000, 0.0000, 0.1846, 0.5425, 0.5019, 0.3323, 0.3478, 0.1171,
0.1153, 0.0000, 0.1688, 0.0000, 0.0950, 0.0000, 0.1272, 0.1653, 0.1538,
0.0000, 0.1446]], grad_fn=<ReluBackward0>)
```

nn.Sequential

`nn.Sequential` 是一个有序的模块容器。数据按照容器中定义的顺序通过所有模块。我们可以使用顺序容器来组合一个像 seq_modules 这样的快速处理网络。

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

nn.Softmax

神经网络的最后一个线性层返回的是 logits类型的值，它们的取值是[-∞, ∞]。 把这些值传递给 `nn.Softmax` 模块。 logits的值将会被缩放到 [0, 1] 的取值区间，代表模型对每个类别的预测概率。 dim 参数指示我们在向量的哪个维度中计算softmax的值(和为1)。

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

模型参数

神经网络内的许多层都是包含可训练参数的，即具有在训练期间可以优化的相关权重(**weight**)和偏置(**bias**)。子类 `nn.Module` 可以自动跟踪模型对象中定义的所有参数字段。使用模型的 `parameters()` 或 `named_parameters()` 方法可以访问模型中所有的参数。 下面的代码可以迭代模型中的每一个参数，并打印出它们的大小和它们的值。

```
print("Model structure: ", model, "\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

```
Model structure:  NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)

Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[-0.018
[-0.0041,  0.0305, -0.0148,  ...,  0.0036, -0.0100,  0.0239]],
device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([-0.0168, -0.00

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[ 0.005
[ 0.0252,  0.0025, -0.0126,  ..., -0.0261,  0.0020, -0.0217]],
device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([-0.0202, -0.02

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[ 0.0142,
[ 0.0070, -0.0086, -0.0114,  ..., -0.0193,  0.0310,  0.0325]],
```

```
device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([-0.0353, -0.011
```

定制模型损失函数和优化器

训练模型之前，我们需要为模型定制一个损失函数 `loss function` 和一个优化器 `optimizer`。

```
loss_fn = nn.CrossEntropyLoss()  # 交叉熵损失函数
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)  # 使用随机梯度下降方法的优化器
```

训练并观察超参数

在单个训练循环中，模型对训练数据集进行预测（分批输入），并反向传播预测误差以调整模型参数。

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)  # 训练数据样本总量
    model.train() # 设置模型为训练模式
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)  # 张量加载到设备

        # 计算预测的误差
        pred = model(X)  # 调用模型获得结果(forward时被自动调用)
        loss = loss_fn(pred, y) # 计算损失

        # 反向传播 Backpropagation
        model.zero_grad() # 重置模型中参数的梯度值为0
        loss.backward() # 计算梯度
        optimizer.step() # 更新模型中参数的梯度值

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

我们还要依赖测试数据集来检查模型的性能，以确保它的学习优化效果。

```python
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()  # 模型设置为评估模式，代码等效于 model.train(False)
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

默认情况下，所有 `requires_grad=True` 属性值的张量都会被跟踪，以便于根据上一次的值来支持对梯度计算。但是，在某些情况下我们并不需要这样做，例如，当我们训练了模型但只想将其应用于某些输入数据的时。或者说白了就是我们只想通过网络进行前向计算时。我们可以把所有计算代码写在 `torch.no_grad()` 下面来停止跟踪计算。

训练过程在多轮迭代（epochs）中进行。在每个epoch中，模型通过学习更新内置的参数，以期做出更好的预测。我们在每个epochs打印模型的准确率和损失值；当然，最希望看到的，就是每个 epoch 过程中准确率的增加而损失函数值的减小。

```python
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("训练完成!")
```

总结

到此我们基本已经对深度学习中的神经网络有了一个初步的认识。从模型结构，到训练的整体流程。包括pytorch中相对应的优化器和损失函数。关于更多的模型结构，优化器和损失函数，我们下次课再进一步补充。

在深度学习领域中要注重打好基础，才能以不变应万变。

本次学习的内容是从源生 python 到机器学习再到深度学习的一个过程，目的是让大家对深度学习中的常用概念术语有一个初步的了解，对一些名词有一个初步的认识。切勿将思维沉浸深度学习当中，深度学习仅是机器学习的一个分支，当我们学习的内容足够多的时候，就会发现里面知识的微妙变化。可以理解现在初入这个领域的学习者，脑子中肯定如一锅粥一般，不知道体系结构，不知道要学多少，那么请一定要稳扎稳打的把基础打牢，过不了多久大脑就会清晰起来。一定要相信时间，也相信自己！