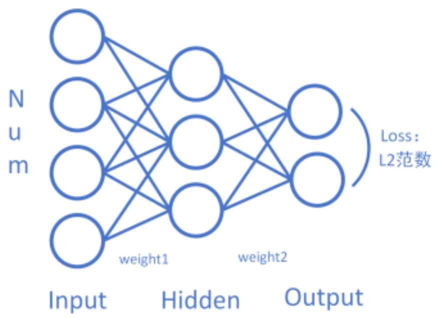
DNN解答

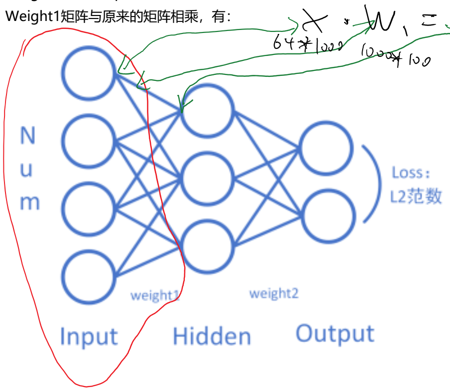
2020年11月5日　15:08



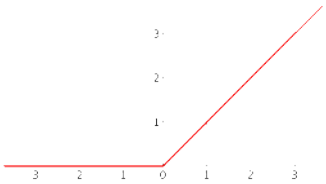
前向过程：

Weight1充当了Input输入特征矩阵跟Hidden隐含层特征矩阵之间全连接的权重矩阵，因此首先要将Weight1矩阵与原来的矩阵相乘，有：

$$\underset{64*1000}{X}\cdot\underset{1000*100}{W_1}=\underset{64*100}{h}\qquad(1)$$

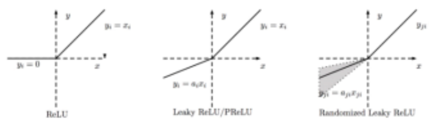


相乘之后需要经过relu激活函数的层，relu线性整流激活函数的函数表达式类似二极管的电压导通压降，为：



$$f(x)=\begin{cases}x, & x>0\\ 0, & x\leq 0\end{cases}$$

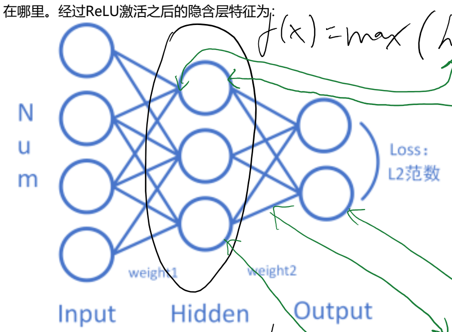
此外还有带泄露的Leaky ReLU函数，在负数时给予一个位于(0,1)区间很小的固定梯度a_i（通常为0.01，或者可以用反向传播计算得到），而random Leaky ReLU函数小的梯度a_i是取自均匀分布的随机变量，用来解决梯度消失的问题：



$$f(x)=\begin{cases}x, & x>0\\ a_i x, & x\leq 0\end{cases}$$

在ReLU函数出现之前，广泛使用Sigmod函数作为激活函数。可以自行思考ReLU相对于Sigmod的优点在哪里。经过ReLU激活之后的隐含层特征为：

$$f(x)=\max(h,0)=h\_relu\qquad(2)$$



接着同理，再对w2做矩阵乘法，有：

$$\underset{64*100}{h\_relu}\cdot\underset{100*8}{W_2}=\underset{\substack{64*8\\ \text{样本数 类别}}}{y\_pred}\qquad(3)$$

得到了预测的输出类别。这里的实际类别y是one-hot编码的，即假设属于类别1则为[1 0 0 0 0 0 0 0]的一维向量，为什么要这么做呢？分两部分看，从数学的角度来说是为了将不同的类别在标签空间上区分开来，实际上利用八个正交向量作为基构造了一个八维的label space，每个样本对应这个label space里的一个点，再对这些点的坐标进行优化使其尽可能地聚类（用词不太准确，聚类严格上应该是unsupervised的方法），而不会出现比如(1+3)/2=2的尴尬情况。

从信息论的角度讲，是为了便于与softmax和相对熵loss进行结合（尽管我们这个程序是用的L2 loss，但他与相对熵比是有缺点的 可以自行思考为什么），相对熵的公式为：

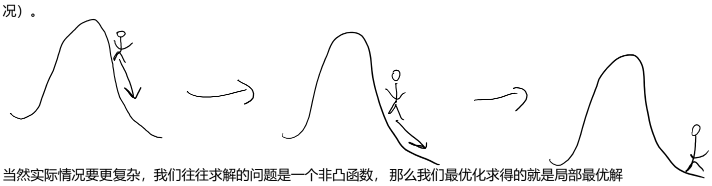
一个点，再对这些点的坐标进行优化使其尽可能地聚类（用词不太准确，聚类严格上应该是unsupervised的方法），而不会出现比如(1+3)/2=2的尴尬情况。
从信息论的角度讲，是为了便于与softmax和相对熵loss进行结合（尽管我们这个程序是用的L2 loss，但他与相对熵比是有缺点的 可以自行思考为什么）。相对熵的公式为:

$$\sum_{x\in X} p(x)\log\left(\frac{p(x)}{Q_2(x)}\right),\quad p(x)=\begin{cases}1，属于该类别\\0，不属于该类别\end{cases}$$

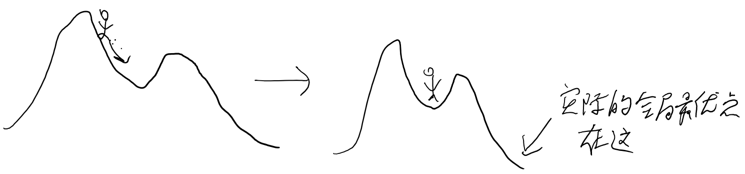
因此利用独热编码可以方便地只计算该类别真实值与预测值的误差，而不会把无关的类别计算进来。
其实最后这里也可以再加入一个激活函数，但为了运算简化，所以没有加入。接着计算预测值与实际类别的L2范数误差。loss通常作为模型分类性能的指标之一，loss越高误差越大，模型拟合数据的能力越差:

$$\sum(y\text{-}pred-y)^2=loss\quad(4)$$

前向传播过程完毕，接下来要反传梯度。为什么反传梯度就可以得到最优的解，涉及到凸函数的最优化问题，具体的数学可以参考WIKI，我在这里只举一个简单的例子方便理解：假设你在爬山过程中需要下山，怎样下山是最快的呢？就是沿着与梯度方向相反的方向（梯度方向为增长最快的方向，因此沿梯度方向是寻找局部极大值。绝大部分情况下，都是沿着负梯度方向进行搜索，除了某些在GAN上应用到的特殊情况）。



当然实际情况要更复杂，我们往往求解的问题是一个非凸函数， 那么我们最优化求得的就是局部最优解而不是全部最优解。因此，为了使我们的局部最优解尽可能靠近全局最优，在设计loss损失函数时要delicate design:



梯度传播:
既然知道了反向梯度传播的原理，我们可以通过链式求导轻松的写出反向传播的式子。预测值跟真实值的L2距离为:

$$(y_{pred}-y)^2=d\quad(5)$$

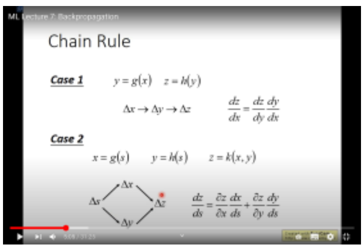
$$\frac{向量}{矩阵}\qquad \frac{\partial d}{\partial y_{pred}}=2(y_{pred}-y)\quad(6)$$

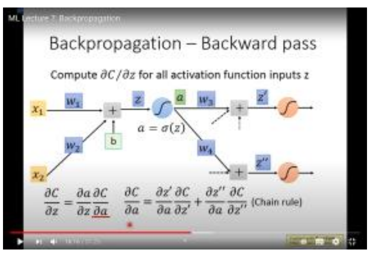
$$\frac{矩阵}{矩阵}\qquad \frac{\partial y_{pred}}{\partial W_2}=h_{relu}\quad(7)$$

更新的是权重矩阵而不是 feature map

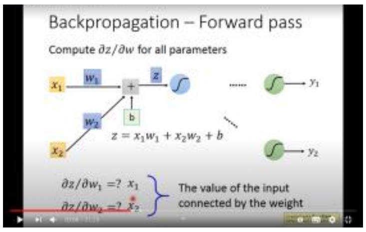
$$由(6)(7)有\ \frac{\partial d}{\partial W_2}=\frac{\partial d}{\partial y_{pred}}\cdot\frac{\partial y_{pred}}{\partial W_2}=\underset{100*64}{h_{relu}^{T}}\,\underset{64*8}{2(y_{pred}-y)}\quad(8)$$

为什么要转置并颠倒顺序?





涉及到梯度的计算有forward和backward两种，像TF pytorch这种AD框架基本都选用了backward也就是backpropagation的这种方式来进行计算，因为这样可以极大地提高我们的运算效率，这也是我们为什么要进行梯度反传的原因。关于这部分的知识推荐观看李宏毅的bacpropagation视频。



注意此处的矩阵相乘其实利用了bacpropagation的case 2，即将每个参数的梯度相乘再相加，**也就是说通过矩阵相乘等于元素相乘再相加的性质，替换了对$W2$矩阵的雅克比矩阵相乘时每个元素的单独计算。这也是为什么这里求了转置并颠倒顺序的原因。下面给出原本的梯度计算形式：**

向量

分母布局

$$\frac{\partial L}{\partial y_{pred}}\frac{\partial y_{pred}}{\partial W_2} = \begin{pmatrix} \frac{\partial L}{\partial y_{pred(1,1)}} & \cdots & \frac{\partial L}{\partial y_{pred(1,c)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial y_{pred(s,1)}} & \cdots & \frac{\partial L}{\partial y_{pred(s,c)}} \end{pmatrix} \begin{pmatrix} \frac{\partial y_{pred}}{\partial W_{2(1,1)}} & \cdots & \frac{\partial y_{pred}}{\partial W_{2(1,c)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{pred}}{\partial W_{2(hid,1)}} & \cdots & \frac{\partial y_{pred}}{\partial W_{2(hid,c)}} \end{pmatrix}$$

矩阵 矩阵

(s表样本数 c表示类别 hid表示隐层元素数量)

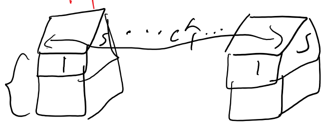
s * c   hid * c

$$\frac{\partial L}{\partial y_{pred(i,j)}} = \begin{pmatrix} \frac{\partial L_1}{\partial y_{pred(i,j)}} & \cdots & \frac{\partial L_s}{\partial y_{pred(i,j)}} \end{pmatrix}$$

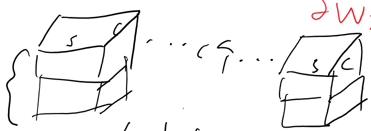
$$\frac{\partial y_{pred}}{\partial W_{2(i,j)}} = \begin{pmatrix} \frac{\partial y_{pred(1,1)}}{\partial W_{2(i,j)}} & \cdots & \frac{\partial y_{pred(1,c)}}{\partial W_{2(i,j)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{pred(s,1)}}{\partial W_{2(i,j)}} & \cdots & \frac{\partial y_{pred(s,c)}}{\partial W_{2(i,j)}} \end{pmatrix}$$

每个大矩阵下只有一个小矩阵

实际上是 s * c · 1 * s 的一个张量(Tensor)



张量运算：$A_{i,j} \cdot B_{j,k} = C_{i,k}$ （向量 · 向量 = 矩阵），与矩阵不同的是把标量推广为向量



hid * cs * c

可以看出，进行张量之间的运算过于复杂，效率低下且容易出错，而且进行张量本身的储存本身占用了太大的空间，容易爆内存，因此AD框架在实际计算时多采用一种叫做**Vector-Jacobian-Product(VJP)**的形式来进行计算，他利用了Jacobian矩阵是对角矩阵、矩阵相乘实际上等于向量相乘再相加的特质，从而**避免进行中间态Jacobian矩阵的运算，直接求出最后所需的梯度。**关于vector-jacobian-product的详细资料可以参考[CS231N(1)](),[(2)](),[CS421(1)](),[(2)]()。这里直接给出它的core idea：简单而言，他是通过先计算链式求导公式中靠前侧的单独一项梯度项（单独一项是比较好求的，比如y=wx求y对x、y对w的梯度是非常简单的），将计算后的结果转化成常数组成的向量，再计算该向量与靠后梯度项的向量积求得最终的梯度。

Recall that the error signal for each node is computed using the formula

$$\overline{\mathbf{v}_i} = \sum_{j \in \mathrm{Ch}(\mathbf{v}_i)} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i}^\top \overline{\mathbf{v}_j}.$$

This can be equivalently written as

$$\overline{\mathbf{v}_i}^\top = \sum_{j \in \mathrm{Ch}(\mathbf{v}_i)} \overline{\mathbf{v}_j}^\top \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i},$$

emphasizing that each piece of the computation involves multiplying a vector by the Jacobian. Hence the term VJP.

Note that we *do not* explicitly construct the Jacobian in order to compute a VJP. For instance, if a node represents a simple elementwise operation, e.g.

$$\mathbf{y} = \exp(\mathbf{z}),$$

then the Jacobian is a diagonal matrix:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix}.$$

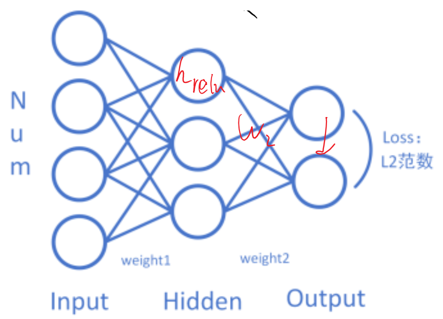
This matrix is size $D \times D$, and the explicit matrix-vector product would require $\mathcal{O}(D^2)$ operations. But the VJP itself can be implemented in linear time:

$$\overline{\mathbf{z}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}}^\top \overline{\mathbf{y}} = \exp(\mathbf{z}) \circ \overline{\mathbf{y}}.$$

对于我们这个作业本身，有：

$$\frac{\partial L}{\partial W_2} = \left(\frac{\partial y_{pred}}{\partial W_2}\right)^T \cdot \frac{\partial L}{\partial y_{pred}} = h_{relu}^T \cdot 2(y_{pred} - y)$$

先求这段



与博客园的这篇文章讲述的方法一致，只是这篇文章并没有给出严格的证明和这样做的原因，只是讲了具体的做法：

假设$\mathbf{x}, \mathbf{y}$分别是m, n维向量，那么$\frac{\partial z}{\partial \mathbf{x}}$的求导结果是一个$m \times 1$的向量，而$\frac{\partial z}{\partial \mathbf{y}}$是一个$n \times 1$的向量，$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$是一个$n \times m$的雅克比矩阵,右边的向量和矩阵是没法直接乘的。

但是假如我们把标量求导的部分都做一个转置，那么维度就可以相容了，也就是：

$$(\frac{\partial z}{\partial \mathbf{x}})^T = (\frac{\partial z}{\partial \mathbf{y}})^T \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

但是毕竟我们要求导的是$(\frac{\partial z}{\partial \mathbf{x}})$,而不是它的转置，因此两边转置我们可以得到标量对多个向量求导的链式法则：

$$\frac{\partial z}{\partial \mathbf{x}} = (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \frac{\partial z}{\partial \mathbf{y}}$$

如果是标量对更多的向量求导,比如$\mathbf{y_1} \to \mathbf{y_2} \to \ldots \to \mathbf{y_n} \to z$，则其链式求导表达式可以表示为：

$$\frac{\partial z}{\partial \mathbf{y_1}} = (\frac{\partial \mathbf{y_n}}{\partial \mathbf{y_{n-1}}} \frac{\partial \mathbf{y_{n-1}}}{\partial \mathbf{y_{n-2}}} \ldots \frac{\partial \mathbf{y_2}}{\partial \mathbf{y_1}})^T \frac{\partial z}{\partial \mathbf{y_n}}$$

知道了VJP这项强大的工具后，可以轻松的求出剩下的梯度：

$$\frac{\partial L}{\partial h_{relu}} = \left(\frac{\partial y_{pred}}{\partial h_{relu}}\right)^T \cdot \frac{\partial L}{\partial y_{pred}} = W_2^T \cdot 2(y_{pred} - y) \quad (4)$$

此时的W_2还未更新，在求出所有梯度后才更新

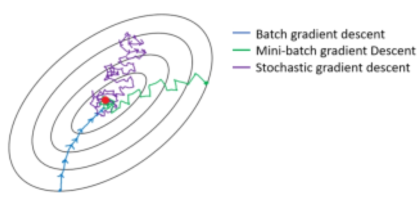
因为我们已经求出了对W_2的梯度并得以更新，因此需要反过来求对feature map的梯度将梯度继续反传回去，因为(3)式只与h_{relu}有关。

$$\frac{\partial L}{\partial h} = \left(\frac{\partial h_{relu}}{\partial h}\right)^T \frac{\partial L}{\partial h_{relu}} = \begin{cases} \frac{\partial L}{\partial h_{relu}}, & \frac{\partial L}{\partial h_{relu}} \geq 0 \\ 0, & \frac{\partial L}{\partial h_{relu}} < 0 \end{cases}$$

$$\frac{\partial L}{\partial W_1} = \left(\frac{\partial h}{\partial W_1}\right)^T \frac{\partial L}{\partial h} = \begin{cases} (X W_2)^T \cdot 2(y_{pred} - y), & \frac{\partial L}{\partial h_{relu}} \geq 0 \\ 0, & \frac{\partial L}{\partial h_{relu}} < 0 \end{cases}$$

数据更新

因为我们取的是整个数据集的样本数量来做这件事（即批量梯度下降），因此计算的是全局的负梯度方向，不存在随机性。目前神经网络为了节约计算开销，一般使用小批量梯度下降的方法，随机性相对较小，但也会因随机性产生振荡。



批量梯度下降->小批量梯度下降->随机梯度下降/online training

由于ReLU激活函数不像batch normalization一样有可学习的参数，因此我们只需要更新W1和W2权重矩阵即可。

设学习率超参为l，则：

$$W_1 \leftarrow W_1 - l \cdot \frac{\partial L}{\partial W_1}$$

$$W_2 \leftarrow W_2 - l \cdot \frac{\partial L}{\partial W_2}$$

这是批量随机梯度下降的情况。实际的问题因为样本数量太大，一般使用小批量梯度下降来做。关于梯度下降的不同方式跟不同的优化器，还是有必要了解一下的，详细的说明可以在D2L里找到：

$$W_1 \leftarrow W_1 - \frac{l}{|B|} \partial W_1 \sum_{i \in B} f(x_i, W)$$

$$W_2 \leftarrow W_2 - \frac{1}{|B|} \partial W_2 \cdot \sum_{i \in B_t} f(x_i, w)$$

~ 每个迭代 t Batch 都会改变