Softmax vs. Softmax-Loss: Numerical Stability

http://freemind.pluskid.org/machine-learning/softmax-vs-softmax-loss-numerical-stability

posted on Free Mind on November 5, 2014
generated with pandoc on January 12, 2016
category: Machine Learning

tags: Deep Learning, Optimization, Julia

一切起源于我在 caffe 的网站上看到的关于 `SoftmaxLossLayer` 的描述：

> *The softmax loss layer computes the multinomial logistic loss of the softmax of its inputs. It's conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.*

caffe 是当下一个很常用的 C++/CUDA 的 Deep Convolutional Neural Networks (CNNs) 的库，由于清晰的代码结构和设计，并且速度也很快，所以不论是学术界还是工业界，要做一些扩展工作的时候经常会选用它。当然现在随着 Deep Learning 的流行，相关的计算库也繁荣起来，许多知名的库大都有各自的特色，具体要选用哪一个的话还得看自己的需求而定。

不过今天我们主要是要围绕上面引用的那一段话来闲聊一下。中心思想是：在数值计算（或者任何其他工程领域）里，知道一个东西的基本算法和写出一个能在实际中工作得很好的程序之间还是有一段不小的距离的。有很多可能看似无关紧要的小细节小 trick，可能会对结果带来很大的不同。当然这样的现象其实也很合理：因为理论上的工作之所以漂亮正是因为抓住了事物的主要矛盾，忽略 “无关” 的细节进行了简化和抽象，从而对比较 “干净” 的对象进行操作，在一系列的 “assumption” 下建立起理论体系。但是当要将理论应用到实践中的时候，又得将这些之前被忽略掉了的细节全部加回去，得到一团乱糟糟，在一系列的 “assumption” 都不再严格满足的条件下找出会出现哪些问题并通过一些所谓的 “engineering trick” 来让原来的理论能 “大致地” 继续有效，这些东西大概就主要是 Engineer 们所需要处理的事情了吧？这样说来 Engineer 其实也相当不容易。这样的话其实 Engineer 和 Scientist 的界线就又模糊了，就是工作在不同的抽象程度下的区别的样子。

接下来闲话就不多说了，今天的主角 Softmax 其实并不一定要和 Deep Learning 有什么关系，在 Logistic Regression 里就可以看到它[1]。具体来说，Softmax 函数 $\sigma(z) = (\sigma_1(z), \ldots, \sigma_m(z))$ 定义如下：

$$\sigma_i(z) = \frac{\exp(z_i)}{\sum_{j=1}^{m} \exp(z_j)}, \quad i = 1, \ldots, m$$

它在 Logistic Regression 里其到的作用是讲线性预测值转化为类别概率：假设 $z_i = w_i^T x + b_i$ 是第 i 个类别的线性预测结果，带入 Softmax 的结果其实就是先对每一个 z_i 取 exponential 变成非负，然后除以所有项之和进行归一化，现在每个 $o_i = \sigma_i(z)$ 就可以解释成观察到的数据 x 属于类别 i 的概率，或者称作似然 (Likelihood)。

[1] 当然反过来说经典的 Deep Neural Networks 顶层其实本来也就是一个 Logistic Regression 分类器。

然后 Logistic Regression 的目标函数是根据最大似然原则来建立的，假设数据 x 所对应的类别为 y，则根据我们刚才的计算最大似然就是要最大化 o_y 的值[2]。后面这个操作就是 caffe 文档里说的 Multinomial Logistic Loss，具体写出来是这个样子：

$$\ell(y, o) = -\log(o_y)$$

而 Softmax-Loss 其实就是把两者结合到一起，只要把 o_y 的定义展开即可

$$\tilde{\ell}(y, z) = -\log\left(\frac{e^{z_y}}{\sum_{j=1}^{m} e^{z_j}}\right) = \log\left(\sum_{j=1}^{m} e^{z_j}\right) - z_y$$

没有任何 fancy 的东西。比如如果我们要写一个 Logistic Regression 的 solver，那么因为要处理的就是这个东西，比较自然地就可以将整个东西合在一起来考虑，或者甚至将 $z_i = w_i^T x + b_i$ 的定义直接一起带进去然后对 w 和 b 进行求导来得到 Gradient Descent 的 update rule，例如我们之前介绍 Gradient Descent 的时候举的两类 Logistic Regression 的例子就是这样做的。

反过来，如果是在设计 Deep Neural Networks 的库，则可能会倾向于将两者分开来看待：因为 Deep Learning 的模型都是一层一层叠起来的结构，一个计算库的主要工作是提供各种各样的 layer，然后让用户可以选择通过不同的方式来对各种 layer 组合得到一个网络层级结构就可以了。比如用户可能最终目的就是得到各个类别的概率似然值，这个时候就只需要一个 Softmax Layer，而不一定要进行 Multinomial Logistic Loss 操作；或者是用户有通过其他什么方式已经得到了某种概率似然值，然后要做最大似然估计，此时则只需要后面的 Multinomial Logistic Loss 而不需要前面的 Softmax 操作。因此提供两个不同的 Layer 结构比只提供一个合在一起的 Softmax-Loss Layer 要灵活许多。从代码的角度来说也显得更加模块化。但是这里自然地就出现了一个问题：numerical stability。

首先我们来看一下在神经网络中进行 gradient descent 的时候所谓的 “Back Propagation” 是什么意思。例如图中所示的一个 3 层神经网络，除了最开始的数据层 L^0 之外，每一层都有输入节点和输出节点，我们用 I_2^1 表示第一层的第二个输入节点，O_3^1 表示第一层的第三个输出节点，每一层的输入和输出节点数量并不一定要一样多，但是通常情况下某一层的输入节点只是上一层的输出节点的 “复制”，比如 $I_3^2 = O_3^1$，因为所有的计算操作是发生在每一层结构内部的。对于普通的神经网络，通常每一层进行的计算都是一个线性映射再经过一个 sigmoid 的非线性操作 S，例如：

$$O_i^1 = \mathsf{S}\left(\sum_{j=1}^{3} w_{ij}^1 I_j^1 + b_i^1\right) = \mathsf{S}\left(\langle w_i^1, I^1\rangle + b_i^1\right)$$

[2] 通常是使用 negative log-likelihood 而不是 likelihood，也就是说最小化 $-\log(o_y)$ 的值，这两者结果在数学上是等价的。

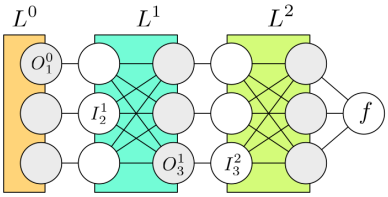

Figure 1: .....

后面是写成向量/矩阵的形式，一般会显得更简洁。现在如果要对参数 w_{ij}^1 进行求导以计算它的 gradient 来进行 gradient descent 一类的参数优化，利用微积分里的 Chain Rule 可以得到如下的式子：

$$\frac{\partial f}{\partial w_{ij}^1} = \sum_{k=1}^{3} \frac{\partial O_k^1}{\partial w_{ij}^1} \cdot \frac{\partial f}{\partial O_k^1}$$

注意到红色的部分是和第一层网络的内部结构相关的，只需要知道该层的局部结构就可以进行计算，而蓝色的部分，由于我们刚才说了输出节点其实等于下一层的输入节点，所以其实是 $\partial f / \partial O_k^1 = \partial f / \partial I_k^2$，这个是可以在 “上面” 的第二层进行计算的，而不需要知道任何关于第一层的信息。

因此整个网络的参数的 gradient 的计算方法是从顶层出发向后，在 L^p 层的时候，会拿到从 L^{p+1} 得到的 $\partial f / \partial I^{p+1}$ 也就是 $\partial f / \partial O^p$，然后需要做两个计算：首先是自己层内的参数的 gradient，比如如果是一个普通的全连通内积层，则会有参数 w^p 和 bias 参数 b^p，根据刚才的 Chain Rule 式子直接计算就可以了，如果 L^p 层没有参数[3]，这一步可以省略；其次就是 “向后传递” 的步骤，同样地，根据 Chain Rule 我们可以计算

$$\frac{\partial f}{\partial I_i^p} = \sum_{k=1}^{3} \frac{\partial O_k^p}{\partial I_i^p} \cdot \frac{\partial f}{\partial O_k^p}$$

计算出来的 $\partial f / \partial I^p$ 将会作为 $\partial f / \partial O^{p-1}$ 传递到 L^{p-1} 层，整个过程就叫做 Back Propagation，其实说白了就是 Chain Rule，只是涉及的符号有点多，容易搞混淆。

这里顺便可以跑题提一下在 Deep Learning 里经常会听到的 Vanishing Gradient 问题，因为 back propagation 是用 chain rule 将导数乘到一起，粗略地讲，如果每一层的导数都 “小于一” 的话，在层数较多的情况下很容易到后面乘着乘着就接近零了。反过来如果每一层的导数都 “大于一” 的话，gradient 乘到最后又会出现 blow up 的问题。人们发明了很多技术来处理这些问题，不过那不是今天的话题。

搞清楚 Back Propagation 之后让我们回到 Softmax-Loss 层，由于该层没有参数，我们只需要计算向后传递的导数就可以了，此外由于该层是最顶层，所以不用使用 chain rule 就可以直接计算对于最终输出（loss）的导数。回忆一下我们刚才的 notation，Softmax-Loss 层合在一起的时候我们用 $\tilde{\ell}(y,z)$ 来表示，它有两个输入，一个是 true label y，直接来自于最底部的数据层，并且我们不需要对数据层做任何的 gradient descent 参数更新，所以我们不需要像那个输入进行 back propagation，但是另外一个输入 z 则来自于下面的计算层，对于 Logistic Regression 或者普通的 DNNs 下面会是一个全连通的线性内积层，不过具体是什么我们也不需要关心，只要把 $\partial \tilde{\ell} / \partial z$ 计算出来丢给下面让他们自己去算后面的就好了。根据普通的微积分知识，我们很容易算出：

[3] 例如 Softmax 或者 Softmax-Loss 层就是没有参数的。

$$\frac{\partial \tilde{\ell}(y,z)}{\partial z_k} = \frac{\exp(z_k)}{\sum_{j=1}^{m}\exp(z_j)} - \delta_{ky} = \sigma_k(z) - \delta_{ky}$$

其中 $\sigma_k(z)$ 是 Softmax-Loss 的中间步骤 Softmax 在 Forward Pass 的计算结果，而

$$\delta_{ky} = \begin{cases} 1 & k = y \\ 0 & k \neq y \end{cases}$$

非常简单对吧？接下来看如果是 Softmax 层和 Multinomial Logistic Loss 层分成两层会是什么样的情况呢？继续回忆刚才的记号：我们把 Softmax 层的输出，也就是 Loss 层的输入记为 $o_i = \sigma_i(z)$，因此我们首先要计算顶层的

$$\frac{\partial \ell(y,o)}{\partial o_i} = -\frac{\delta_{iy}}{o_y}$$

然后我们把这个导数向下传递，现在到达 Softmax 层，在 apply chain rule 之前，首先计算层内的导数

$$\begin{aligned} \frac{\partial o_i}{\partial z_k} &= \frac{\delta_{ik}e^{z_i}\left(\sum_{j=1}^{m}e^{z_j}\right) - e^{z_i}e^{z_k}}{\left(\sum_{j=1}^{m}e^{z_j}\right)^2} \\ &= \delta_{ik}o_k - o_i o_k \end{aligned}$$

如果用 Chain Rule 带进去验算一下的话：

$$\sum_{i=1}^{m}\frac{\partial o_i}{\partial z_k}\cdot\frac{\partial \ell(y,o)}{\partial o_i} = o_k - \delta_{yk}\cdot\frac{o_k}{o_y} = o_k - \delta_{yk}$$

和刚才的结果一样的，看来我们求导没有求错。虽然最终结果是一样的，但是我们可以看出，如果分成两层计算的话，要多算好多步骤，除了计算量增大了一点，我们更关心的是数值上的稳定性。由于浮点数是有精度限制的，每多一次运算就会多累积一定的误差，注意到分成两步计算的时候我们需要计算 $-\delta_{iy}/o_y$ 这个量，如果碰巧这次预测非常不准，o_y 的值，也就是正确的类别所得到的概率非常小（接近零）的话，这里会有 overflow 的危险。下面我们来实际试验一下，首先定义好两种不同的计算函数：

```
function softmax(z)
  #z = z - maximum(z)
  o = exp(z)
```

```
  return o / sum(o)
end

function gradient_together(z, y)
  o = softmax(z)
  o[y] -= 1.0
  return o
end

function gradient_separated(z, y)
  o = softmax(z)
  ∂o_∂z = diagm(o) - o*o'
  ∂f_∂o = zeros(size(o))
  ∂f_∂o[y] = -1.0 / o[y]

  return ∂o_∂z * ∂f_∂o
end
```

然后由于 float (`Float32`) 比 double (`Float64`) 的精度要小很多，我们就以 double 的计算结果为近似的“正确值”，然后来比较两种情况下通过 float 来计算得到的结果和正确值之差。绘图代码如下：

```
using DataFrames
using Gadfly

M = 100
y = 1
zy = vec(10f0 .^ (-38:5:38)) # float range ~ [1.2*10^-38, 3.4*10^38]
zy = [-reverse(zy);zy]
srand(12345)

n_rep = 50
discrepancy_together = zeros(length(zy), n_rep)
discrepancy_separated = zeros(length(zy), n_rep)

for i = 1:n_rep
  z = rand(Float32, M)  # use float instead of double

  discrepancy_together[:,i] = [begin
    z[y] = x
    true_grad = gradient_together(convert(Array{Float64},z), y)
    got_grad = gradient_together(z, y)
    abs(true_grad[y] - got_grad[y])
```

```
  end for x in zy]
  discrepancy_separated[:,i] = [begin
    z[y] = x
    true_grad = gradient_together(convert(Array{Float64},z), y)
    got_grad = gradient_separated(z, y)
    abs(true_grad[y] - got_grad[y])
  end for x in zy]
end

df1 = DataFrame(x=zy, y=vec(mean(discrepancy_together,2)),
                label="together")
df2 = DataFrame(x=zy, y=vec(mean(discrepancy_separated,2)),
                label="separated")
df = vcat(df1, df2)

format_func(x) = @sprintf("%s10<sup>%d</sup>", x<0?"-":"",int(log10(abs(x))))
the_plot = plot(df, x="x", y="y", color="label",
                Geom.point, Geom.line, Geom.errorbar,
                Guide.xticks(ticks=int(linspace(1, length(zy), 10))),
                Scale.x_discrete(labels=format_func),
                Guide.xlabel("z[y]"), Guide.ylabel("discrepancy"))
```

这里我们做的事情是保持 z 的其他坐标不变，而改变 z_y 也就是对应于真是 label 的那个坐标的数值大小，我们刚才的推测是当 o_y 很接近零的时候会有 overflow 的危险，而 $o_y = \sigma_y(z)$，忽略掉 normalization 的话，正比于 $\exp(z_y)$，所以我们需要把 z_y 那个坐标设成绝对值很大的负数。在得到的图中我们可以看到以整个数值范围内的情况对比。图中横坐标是 z_y 的大小，纵坐标是分别用两种方法计算出来的结果和 “真实值” 之间的差距大小。

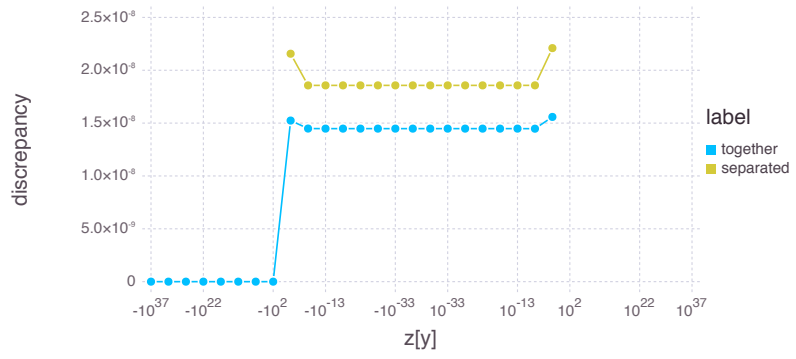

Figure 2: .....

首先可以看到的是单层直接计算确实比分成两层算要好一点，不过从纵坐标上也可以看到两者差距其实非常小。往左边看的话，会发现黄色

的点没有了，那是因为结果得到了 `NaN` 了，比如 o_y 由于求一个绝对值非常大的负数的 exponential，导致下溢超出 float 可以表示的小数点精度范围，直接变成 0 了，此时 $1/o_y$ 就是 `Inf`，当要乘以 o_y 进行 cancel 的时候得到 $0 \times \infty$，对于浮点数这个操作会直接得到 `NaN`，也就是 Not a Number。反过来看蓝线的话，好像有点奇怪的是越往左边好像反而变得更加精确了，其实是因为我们的 “真实值” 也 underflow 了，因为 double 虽然比 float 精度高很多，但是也是有限制的。根据 Wikipedia，float 的精度范围大致是 $10^{-38} \sim 10^{38}$，而 double 的精度范围大致是 $10^{-308} \sim 10^{308}$，大了很多，但是我们不妨来看一下图中的 -10^2 这个坐标点，注意到

$$e^x = 10^{x/\log 10}$$

所以 $\exp(-10^2) \approx 10^{-44}$，对于 float 来说已经下溢了，对于 double 来说还是可以表示的范围，但是和 0 的差别也已经如此小，在图上已经看不出区别来了。指数再移一格的话，$\exp(-10^3) \approx 10^{434}$，会直接导致 double 也 underflow，结果我们的 “真实值” 也会是零，所以 “误差” 直接变成零了。

比较有趣的是往右边的正数半轴看，发现到了 10^2 之后蓝线和黄线都没有了，说明他们都得到了 `NaN`，不过这里是另一个问题：对一个比较大的数求 exponential 非常容易发生 overflow。还是用刚才的式子可以看到 $\exp(10^2) \approx 10^{44}$，已经超过了 float 可以表达的最大上限，所以会变成 `Inf`，然后在 normalize 的一步会出现 `Inf/Inf` 这样的情况，于是就得到 `NaN` 了。

这个问题其实也是有解决办法的，我们刚才贴的代码里的 `softmax` 函数第一行有一行被注释掉的代码，就是在求 exponential 之前将 z 的每一个元素减去 z_i 的最大值。这样求 exponential 的时候会碰到的最大的数就是 0 了，不会发生 overflow 的问题，但是如果其他数原本是正常范围，现在全部被减去了一个非常大的数，于是都变成了绝对值非常大的负数，所以全部都会发生 underflow，但是 underflow 的时候得到的是 0，这其实是非常 meaningful 的近似值，而且后续的计算也不会出现奇怪的 `NaN`。

当然，总不能在计算的时候平白无故地减去一个什么数，但是在这个情况里是可以这么做的，因为最后的结果要做 normalization，很容易可以证明，这里对 z 的所有元素同时减去一个任意数都是不会改变最终结果的——当然这只是形式上，或者说 “数学上”，但是数值上我们已经看到了，会有很大的差别。

这就是本文的全部啦！如果想了解更多计算机浮点数值计算上会碰到的各种各样的问题，也许在这本书里会有更多的内容：《Numerical Computing with IEEE Floating Point Arithmetic: Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises》。

最后，julia 允许 unicode 的变量名，于是可以写各种带数学符号的变量名了！而且 vim 的 julia 插件有一个非常方便的功能就是能够将 LaTeX

的符号命令补全成对应的 unicode 符号，比如输入 `\nabla` 然后按 Tab，就会变成 ∇，非常方便。但是不得不念念碎一下的是很多地方还有待改进，比如现在加载一个像 Gadfly 绘图库这种规模的 package 简直就是要等到天荒地老，另外像这种没有事先编译的语言，哪里写错了不到运行到那里的时候都发现不了错误，可以想象我只是要写一个小的绘图 script 而已，写完运行，等个 N 秒终于把绘图库加载进来了然后碰到一个 typo，出错了，修掉 typo 再运行，又是一轮等待，然后又碰到一个 typo……当然我写代码比较粗心是我的问题，但是这种时候不得不就开始怀念编译型语言啊。不过听说现在正在开发的 0.4 版的一个重要内容就是静态预编译，这个功能实现之后各种库应该都能做到瞬间加载。

另外还想吐槽的是，julia 里轻量的 coroutine 和强大的宏虽然是两大卖点，但是却也还是要掂量着点用啊。最主要的问题就是错误报告，之前尝试过用 coroutine 来写一个东西，由于没有编译期错误检查，即使是很傻的代码错误也得等到运行期抛出异常来排查，结果用了 coroutine 之后，经常都在 stack trace 里找不到正确的出错行号。还有宏展开也是好像时不时会把错误汇报的行号搞乱掉。嘛，不过这些东西本来也就是双刃剑了，就像 C++ 的 template 用来做模板元编程如果出错的话也是会打印出几千页的编译错误的。