Luozm +

Backpropagation

2017-03-24

Stanford CS231n课程笔记-2

原创文章，转载请注明：转自Luozm's Blog

之前对于反向传播算法（BP）的理解很浅，而且总认为一个模型需要从头到尾求导解决问题，事实上这根本无法实现，而且也很难计算。为了更好的理解BP，我抽空学习了CS231n的Lecture3&4，这给了我很大的启发，现在对于BP有了全新的认识。而且这门课的思路与TensorFlow特别相似，对于更好的掌握深度学习框架也有一定的帮助。

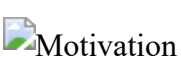
想要直接看课件的同学可以点这里：

- videos
- slides
- backprop notes

更多参考资料：

- Schedule and Syllabus
- A Neural Network in 11 lines of Python (Part 1)

1. BP–motivation



前面的梯度下降（SGD）让我们有了参数更新的方法，但是其中需要对每个参数求偏导，如何能够高效灵活的完成这个任务呢？这就要用到 **反向传播算法 (Backpropagation)** 了。

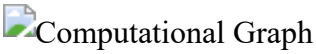
1.1 什么是反向传播算法?

反向传播算法是一种通过递推使用链式法则（chain rule）来计算表达式梯度（即偏导数组成的向量）的方法。它能够极大的提升神经网络的训练速度。除了深度学习，反向传

播算法在许多其他领域也是一种强大的计算工具，只是有着不同的名字。事实上，这种算法在不同领域至少被重复发明了十次以上，详情见Griewank (2010) 。它的普遍的，与应用无关的名字是 **reverse-mode differentiation** 。

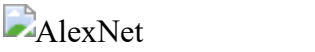
2. BP–Computational Graph

为了使用反向传播算法，我们需要首先引入 **计算图（Computational Graph）** 的概念。计算图是一种使用类似电路图和逻辑门的方式来表示一系列数学计算表达式的模型。大概就是下面这个样子：

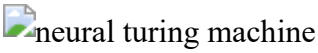

Computational Graph

这个计算图表示的就是简单线性分类器的模型。其中 x 是输入，W 是权重矩阵，f 是scores function，而loss function采用的是hinge loss，再加上正则化项 $R(W)$，最终计算得到损失L。

事实上，如果我们需要计算的模型都是这样的，那么也不需要反向传播了，也可以直接计算出 W 的梯度。但是，现在的模型往往都非常大，比如著名的AlexNet：


AlexNet

是不是很吓人了？还有更可怕的，比如神经图灵机模型：


neural turing machine

如果要人工直接计算这种模型的梯度解析解，恐怕要算到猴年马月了。因此我们需要对求解过程进行分解，这样就可以借助计算机的力量来解决问题。

2.1 链式法则

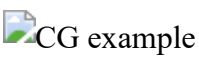
这里先简单的复习一下链式法则：例如我们想要计算 $f(x,y,z)=(x+y)z$的偏导数，可以引入一个中间变量 q，使其转化为两个表达式：$f=qz$ 和 $q=x+y$。

每个表达式的偏导数都非常好求：$\frac{\partial f}{\partial q}=z, \frac{\partial f}{\partial z}=q$ 以及 $\frac{\partial q}{\partial x}=1, \frac{\partial q}{\partial y}=1$。

但是我们并不关心中间变量 q 的导数，而只想知道 f 对于 x,y,z 的偏导数，这时链式法则告诉我们：$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$，因此只需要把上面的式子简单的相乘，就可以得到想要的结果了。

2.2 计算图的计算方法

继续使用上面的例子，把这个问题使用计算图来表示，就是下面的样子：


CG example

图中，每一个节点表示一次运算。输入的 x, y, z 已知，然后通过简单的运算得出了每次中间运算的结果，在图中每个节点的上面给出。接下来如何通过计算图来计算偏导数呢？

由链式法则可知，**最终的梯度（偏导数）可以由一系列局部梯度（local gradients）相乘得到**。让我们先从最后一项开始看起：


df

这最后一项的局部偏导数就是1，因为想要求的就是对 f 的偏导。现在我们知道了这项的局部偏导数，把它写到这个节点的下面，与这点的数值对应。


dz

继续往前推进，上面我们已经知道，这个节点的局部偏导数是 q，查看图中的数值发现 $q = 3$，因此这个点的局部偏导数就是3。根据链式法则：$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial z}$，因此 z 的偏导数就是 $3 \times 1 = 3$，也写到图上的相应位置。

同样的思路，也可以得到 q 的偏导数为-4，也写到图上。接下来我们考虑 y：


dy

一样的方法，先计算出 y 的局部偏导数 $\frac{\partial q}{\partial y} = 1$，然后与我们之前得到的 q 的偏导数相乘，得到 y 的偏导数为 $1 \times -4 = -4$。


dx

还是相同的方法，我们也可以很容易的计算出 x 的偏导数为-4。至此，所有的偏导数都求出来了。是不是很简单？

2.3 计算图的节点

上面的例子比较简单，使用计算图可能比直接计算还慢一些。让我们看一个抽象的例子：


activations

这是计算图中的任意一个节点，x, y 是输入，z 是输出，节点的运算是 $z = f(x, y)$。这个运算可以是上面的加法、乘法，或者也可以是一个简单的函数，甚至还可以表示一堆函数的组合。

对于一个节点来说，我们已知的是它的输入和函数 f 的计算方法。然后我们可以计算出它的输出 z，同时由于我们清楚这个节点的函数 f 的计算表达式，也可以算出这个节点的局部梯度（偏导数）。如果知道这个节点上游（输出 z）对于目标函数（L）的梯度，与局部梯度相乘，就可以计算出这个节点的输入（x, y）对于目标函数的梯度了。

总结一下：

1. 给定一个节点的运算，其局部梯度也同时已知（即，$f(x, y)$、$\frac{\partial z}{\partial x}$、$\frac{\partial z}{\partial y}$）；
2. 有输入进入节点，首先进行正向传播，得到输出 z；
3. 然后反向传播，根据传入的上游梯度与局部梯度相乘得到每个输入的梯度。

图中的每个节点都有这两个功能：**正向推断（Inference）**和**反向传播（Backpropagation）**。把这些节点按照一定的顺序连接起来，组成一张计算图，那么这张图也就可以具有这两个功能：由输入计算输出，再反向计算每个参数的梯度。这样的话，即使是复杂如之前的卷积神经网络，只要我们定义好几个基础节点的这两个功能，比如线性组合节点、激活函数节点、卷积节点、池化节点，那么由这些基础节点组合出来的复杂模型也可以就很容易的计算和训练了。

2.4 计算图练习

看了前两节的介绍，是不是对计算图的计算已经有所了解了？让我们来练练手吧，请看下面的例子：

 dense

这个计算图表示的就是神经网络模型中常用的一个全连接层，包括两个输入变量的线性组合加偏置，再加上Sigmoid激活函数。

练习：请按照前面所讲的方式在图上计算出每个参数（w_0, w_1, w_2）的偏导数。（注意：每次运算只保留两位小数）

Hint：图上已经标出每个节点正向计算出的数值，并给出可能用到的导数。

2.5 节点的组合（gate）

是不是很快就得到了结果呢？可以与下图对比一下看看是否算对了（**注：图中有误，**$x_0 = w_1 = 0.40$，$x_1 = 0.60$）。

 sigmoid gate

在我们前面讲的计算图中，每次运算都单独作为一个节点存在，但在实践中我们有时也会把一些常用的运算组合成一个运算门（gate）。图中蓝色圈出的运算合起来就是神经网络中常见的Sigmoid函数，为了写代码时方便起见，我们把它们合并起来成为一个单独的节点：sigmoid gate。这个节点的运算方式与之前的相同，只是把中间部分的运算当作黑箱，只由第一个节点和第四个节点作为接口与外界进行交互。

 patterns

上图展示了一些非常常用的gate：加法、最大值、乘法。为了更好的理解它们梯度的意思，我们在这里给它们起了一些名字。

- 加法运算（add gate）的导数恒为1，因此被称为梯度批发商（我自己翻译的，挺有意思 ^_^），意思是把上游传递过来的梯度给下游每个节点都相同的复制一份；
- 取最大值运算（max gate）的导数则是只有最大的节点为1，其他为0。这个运算使得上游的梯度只传递给最大的那个节点，因此被称为梯度路由器；
- 乘法运算（mul gate）则是梯度与数值互换，可以看成是梯度交换机。

这里要补充一下，上面的梯度传播可能会有一个小bug，如果反向传播时有多个节点同时指向一个节点怎么办呢？如下图所示：

 branches

其实很简单，把每个节点传递过来的梯度相加求和就可以了 ~

2.6 向量版本

 vectorized

还是之前讲过的单节点模型，但是前面输入和输出都是一个标量，现在我们考虑向量版本。如果 x, y, z 现在都是向量了，那么这个反向传播应该怎么算呢？

让我们看一个简单的例子：假设输入 $x = (x_1, x_2), y = (y_1, y_2)$，输出 $z = (z_1, z_2)$，节点运算为 $z = f(x, y) = x + y = (x_1 + y_1, x_2 + y_2)$。那么分别计算 z 对于 x 各个元素的局部梯度就是：

$$\frac{\partial z_1}{\partial x_1} = 1, \frac{\partial z_1}{\partial x_2} = 0, \frac{\partial z_2}{\partial x_1} = 0, \frac{\partial z_2}{\partial x_2} = 1$$

方便起见，把它们写到一起，就变成了一个矩阵：

$$\frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} \\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

y 也是同理，这个矩阵就是著名的雅可比矩阵（Jacobian matrix）。有了雅可比矩阵，对于向量化的输入和输出，只需要在反向传播时乘上这个矩阵就可以计算梯度了。

3. BP–Implementation

前面讲的是理论部分，下面来看一下如何编程实现。

注：文中使用的代码为Python

3.1 计算图的伪代码

对于每个运算节点，具体编程中是如何定义的呢？来看下面这个类：


gate api

这是一个乘法门的实例，其中的输入和输出都是标量。

1. 首先定义运算节点为一个类（class），它拥有两个函数：forward和backward，分别对应正向推断输出和反向传播梯度；
2. 在正向函数中，它可以根据输入来计算输出，**在类中保存该输入值**（非常重要，用来计算反向传播梯度），并返回输出值；
3. 在反向函数中，它已经定义好了局部梯度的计算公式，利用之前保存的输入值以及从高层传播下来的梯度值计算输入的梯度，并返回各个输入的梯度。

每个具体的运算节点都是类似的定义方法。那么对于整个计算图来说，又是如何定义的呢？


API

这是一段计算图的伪代码，描述了一张组合好的计算图（或者网络）的大概功能和实现方法。这里假设计算图中需要用到的每个节点都已经定义好了。

1. **class：** 首先定义计算图为一个类，它实际上就是计算图中各个运算节点的简单封装，拥有两个主要函数功能：forward和backward，分别对应正向推断输出和反向传播梯度，与单个节点类似；
2. **forward：** 在正向函数中，先遍历计算图中的所有运算节点（gate），然后按照计算顺序对其进行排序，接着根据顺序依次计算每个节点的输出并作为下一个节点的输入向前传播，直到传递到损失函数，最后返回Loss值；
3. **backward：** 在反向传播中，逆向使用之前的排序，按照这个顺序依次计算梯度，直到传播到第一层，记录并返回所有所需参数的梯度。

3.2 计算图的实例

看到这里，相信你已经学会如何来自己编程搭建一张计算图用于神经网络模型的训练了。下面来看一些深度学习框架中的实例，看看它们是不是也是这样编写的呢？


torch

第一个例子是Torch，这是Facebook开源的基于Lua语言的深度学习框架（Lua我也不会，但是不影响我们理解）。可以看到在它的库中定义好了大量的层（常用运算节点的组合），并且每一个层都有向前和向后的两个函数，就像搭积木一样，我们需要做的只是在其中挑选我们想要的模块，然后按顺序把它们拼接起来，一个可以训练的神经网络模型就这样建立完成了，是不是很简单？


torch mulconstant

让我们具体来看一个模块，这是一个简单的数乘运算节点。它有三个函数，updateOutput（），也就是之前讲过的forward函数。updateGradInput（），也就是backward函数。再加上一个用于初始化的init（）函数。

在初始化函数中，记录要乘的数，保存在类中。**这里有一个inplace标签，指的是是否直接使用输出覆盖输入值**（好处是可以节省内存，但是只能用于局部求导与输入无关的情况）。

在forward和backward的函数中实现了我们之前所说的功能，大家可以自行对照一下。

第二个例子是Caffe，这是贾扬清使用C++编写的深度学习框架，在图像领域被广泛使用。尽管语言不同，但是与Torch类似，Caffe也定义了大量的运算模块，下面具体来看一个层：


caffe

这是使用CPU的Sigmoid层的代码，可以看到与前面的MulConstant非常相似，也是三个函数：内联函数定义Sigmoid运算（与init相同，只是不需要传入外界参数），Forward_cpu()函数以及Backward_cpu()函数分别使用CPU来进行输出和反向传播梯度的运算。

3.3 向量实现

之前我们介绍过，如果输入和输出都是向量的话，那么中间的局部梯度就需要使用一个雅可比矩阵来表示。但是这在实际编程中可能会遇到严重的问题，来看下面的例子：


imp-verctorized

假设我们的输入是一个4096维的向量，然后在这个节点对其进行一个简单的ReLU（就是只保留正数部分，负数全部置0，这是现在图像模型中最常用的激活函数）运算，输出也是一个4096维的向量。

如果我们使用前面讲的理论方法，这个运算的局部梯度需要使用一个多大的雅可比矩阵来表示呢？没错，需要4096×4096！那么这个矩阵长什么样子呢？大概就是下面这样：

$$\begin{bmatrix} 1 & & & & & \\ & 0 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix}$$

这个矩阵类似于一个单位对角矩阵，只有对角线上有元素0或1，其中输入为正数的位置为1，其他部分为0。对于这个运算来说，不管是正向还是反向，只需要最多进行4096次0或1的数乘就完全可以完成任务了，与4096×4096矩阵乘法的内存消耗以及运算量完全不在一个数量级。

 imp-vectorized-2

在实际的训练中，我们常常采用batch训练的方式，即一次同时将一批（比如100个）输入向量送入模型进行训练，那么这时所需要的雅可比矩阵就更加可怕了，达到了409600×409600！这完全是个灾难！

因此在实际操作中，我们通常不需要完整给出雅可比矩阵，只需要人工简化到能够输出对应的输出值和梯度值即可。这里可以采用少量循环的方式来计算，或者一定要使用矩阵通常也采用稀疏矩阵的方式来进行储存和运算，这可以大大的简化内存占用量。

4. Summary

 summary

总结一下，由于模型过于庞大，直接写出所有参数的梯度公式是不可能做到的。因此我们引入了反向传播算法，通过递推使用求导链式法则沿着计算图来计算所有需要的参数梯度。

在编程实现中，我们需要定义每个常用节点的forward和backward接口API，并封装在计算图结构中。

最后介绍一个有趣的代码，仅用11行Python代码就完成了一个神经网络模型的定义以及训练：

A Neural Network in 11 lines of Python (Part 1)

```
1. X = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ])
2. y = np.array([[0,1,1,0]]).T
3. syn0 = 2*np.random.random((3,4)) - 1
4. syn1 = 2*np.random.random((4,1)) - 1
5. for j in xrange(60000):
6.     l1 = 1/(1+np.exp(-(np.dot(X,syn0))))
7.     l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))
8.     l2_delta = (y - l2)*(l2*(1-l2))
9.     l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1))
10.    syn1 += l1.T.dot(l2_delta)
11.    syn0 += X.T.dot(l1_delta)
```

这段代码也遵循了我们之前所讲的规则来编写，只是省略了不必要的部分。有兴趣的同学可以自己分析一下，然后看看这份教程，它主要是从代码层面来分析反向传播算法。

重要勘误说明： 上面的代码我认为有误：在课中老师说这份代码采用了逻辑回归损失作为损失函数，但是经推导发现，如果采用逻辑回归损失，即如下公式：

$$\text{Loss}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y) \log(1-h_\theta(x))$$

那么`l2_delta = (y - l2)*(l2*(1-l2))`这行中的`(l2*(1-l2))`是没有必要的，应该修改为`l2_delta = (y - l2)`。

经过与学长讨论，发现如果加上后面多余的部分，则恰好满足 **指数损失函数** 的梯度公式，因此相当于采用了指数损失作为损失函数，也一样可以训练出收敛的模型。

为了验证两种损失函数的效果，我分别尝试了使用两种代码来测试这个模型（代码采用MATLAB编写）：

```
1. x=[0,0,1;0,1,1;1,0,1;1,1,1];
2. y=[0,1,1,0]';
3.
4. % 初始化参数
5. theta=2*rand(3,5)-1;   % 第一层的权重
6. theta2=2*rand(5,1)-1;   % 第二层的权重
7.
8. for i=1:1000   % 训练1000次
```

```
    % 正向计算输出
    l1=x*theta;
    h1=1./(1+exp(-l1));
    l2=h1*theta2;
    h2=1./(1+exp(-l2));

    % 计算梯度值
    %l2_e=y-h2;  % 采用逻辑回归损失
    l2_e=(y-h2).*h2.*(1-h2);  % 采用指数损失
    l1_e=l2_e*theta2'.*h1.*(1-h1);
    l2_d=h1'*l2_e;
    l1_d=x'*l1_e;

    % 更新参数
    theta2=theta2+l2_d;
    theta=theta+l1_d;
end
```

经实验发现，两种损失函数确实都可以达到很好的拟合效果，但是逻辑回归损失的效果要更好一些。

由于我现在对指数损失还不太熟悉，因此这段勘误说明也可能有误，如果大家在阅读时发现错误，希望能够联系我，我会立刻改正，谢谢大家！