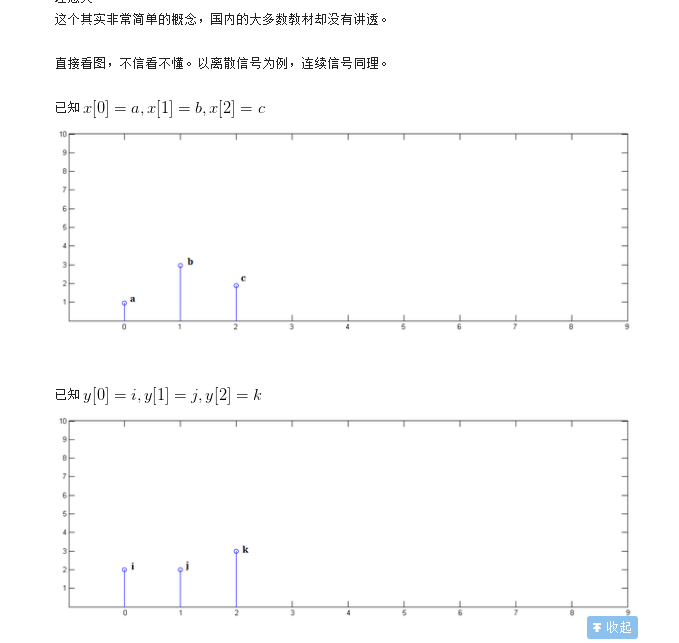
卷积神经网络

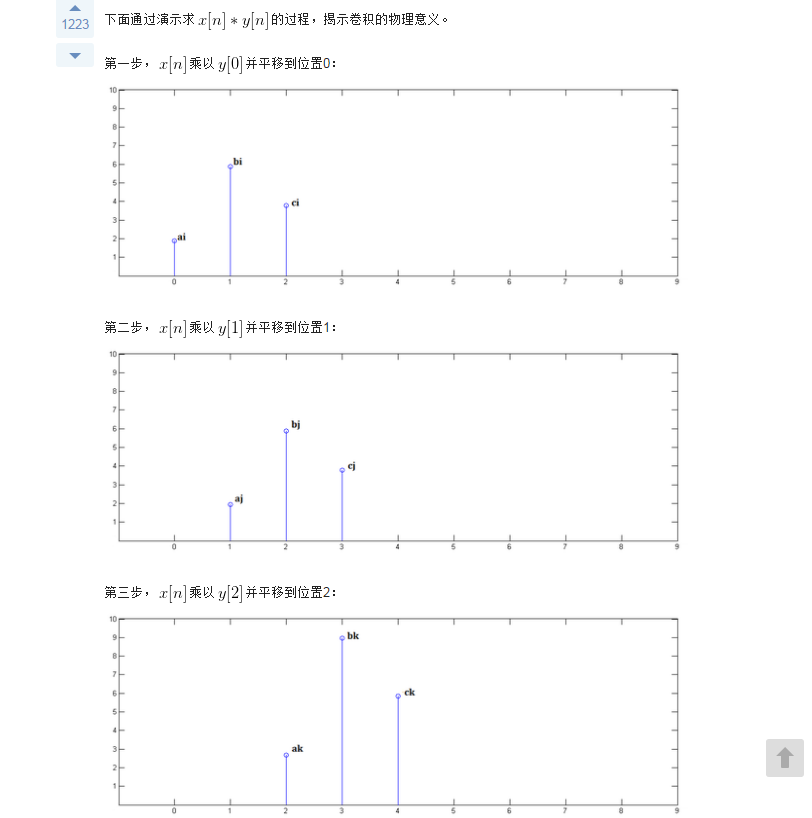
**卷积：设:f(x),g(x)是R1上的两个可积函数，作积分：**

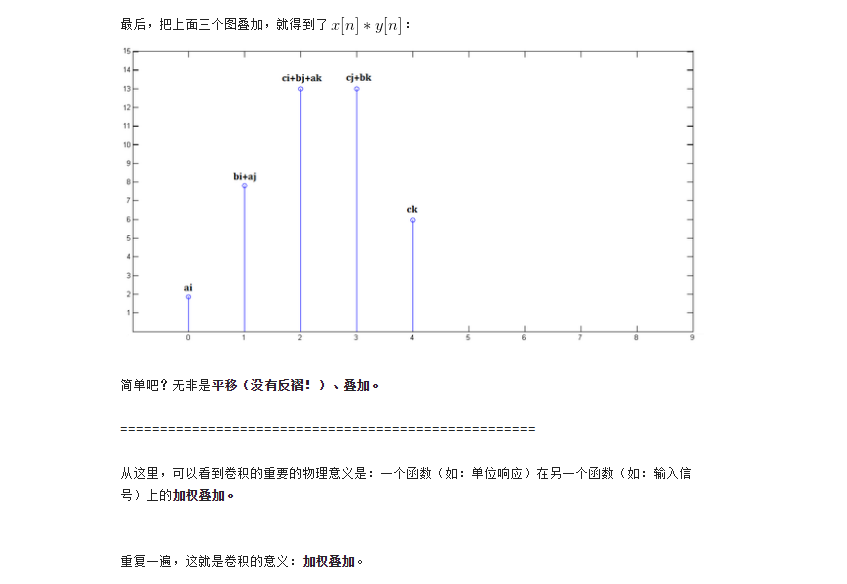
http://c.hiphotos.baidu.com/baike/s%3D143/sign=ab6317f1b251f819f525074ee9b54a76/54fbb2fb43166d22dd3b844e442309f79152d298.jpg

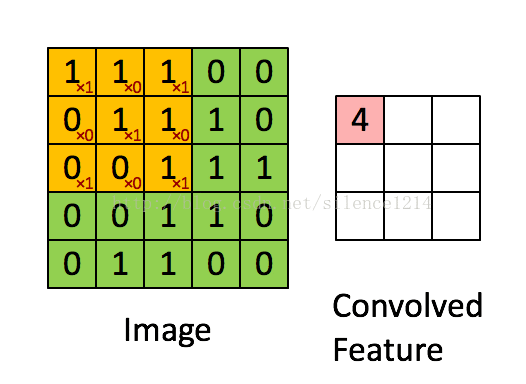
**可以证明，关于几乎所有的实数x，上述积分是存在的。这样，随着x的不同取值，这个积分就定义了一个新函数h(x)，称为函数f与g的卷积，记为h(x)=(f\*g)(x)。**

**一个生动的解释：**

****

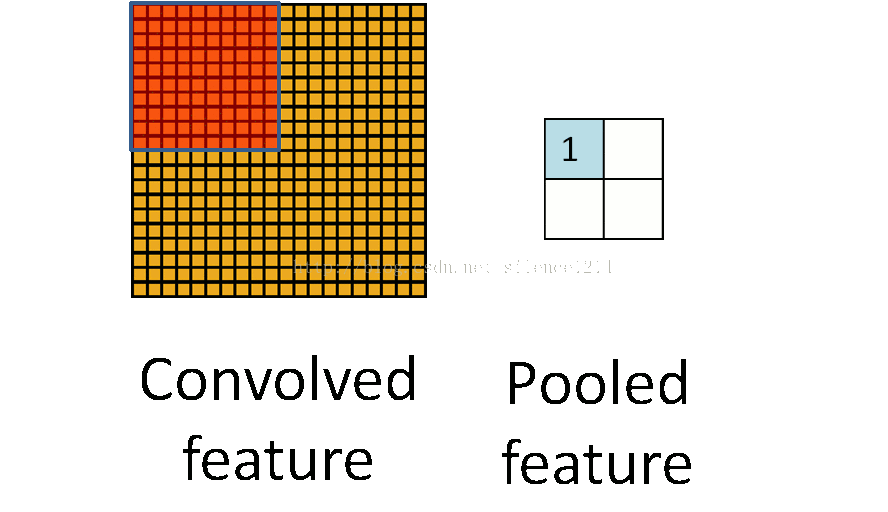
****

****

****

这幅图是对一个5\*5的矩阵A进行3\*3的矩阵B的卷积，那么就从最上角到右下角，生成卷积之后的矩阵的大小是(5-3+1)\*(5-3+1)的矩阵，生成之后的矩阵的元素值，是之前的两个矩阵对应元素的乘积之和(动图见附件)

**池化：**

****

比如上方左侧矩阵A是20\*20的矩阵要进行大小为10\*10的池化，那么左侧图中的红色就是10\*10的大小，对应到右侧的矩阵，右侧每个元素的值，是左侧红色矩阵每个元素的值得和再处于红色矩阵的元素个数，也就是平均值形式的池化。(动图见附件)

**卷积神经网络(CNN)：**

1.CNN介绍

CNN是应用在图像识别领域较多的一种机器学习算法，它与普通的神经网络有点类似，通过输入层然后乘以权重加上偏置，并且通过一个非线性的激活函数，即可得到输出值。CNN要做的是给定一幅图，对该图像进行编码最终即可将该图进行分类，而此过程中参数相比普通神经网络的参数要少，并且更加高效。

CNN通常由三个部分组成，分别是Conv layer、Pooling layer、Full-connected layer，它们组成了一个完整的CNN结构。通常输入是一幅图片，w\*h\*c，分别为图像的宽度高度以及频道数目(RGB)，conv layer将计算输入层与权重矩阵的卷积，然后将该结果通过ReLU函数，Pooling layer用于对卷积后的矩阵进行抽样降维，例如按2\*2矩形大小对原卷积矩阵分成若干个2\*2大小的块，然后对于每个块只取其中的最大值，这样就可以使得维度降低为原来的1/4，而这就是max pooling方法，该方法的优点是减少了计算过程中参数的个数以及防止过拟合的产生。最后，经过的是MLP，方法类似，可以参考以前的博文（逻辑回归及感知器）。

总结一下，CNN就是将一幅图像经过一层又一层的编码过后，可以得到一个结论值，该值是对该图像的分类或描述。

2.CNN基本原理

当输入数据的维度很大时，如图像，就不能简单地每个输入值连接一个神经元，这时候每一个神经元要负责一片感知场，例如若输入数据为32\*32\*3，而感知场的个数为5\*5，则此时conv layer的权重为5\*5\*3，因此共有75个权重。这样就可以有效减少参数的数目。

一般情况下，CNN的结构形式是:输入层--->Conv层--->Pooling层--->(重复Conv、Pooling层)...--->FC层--->输出结果。具体说来，通常还有以下几种形式：

    Input--->((Conv--->ReLU--->Conv--->ReLU)--->Pooling)\*3--->(FC--->ReLU)\*2--->Output

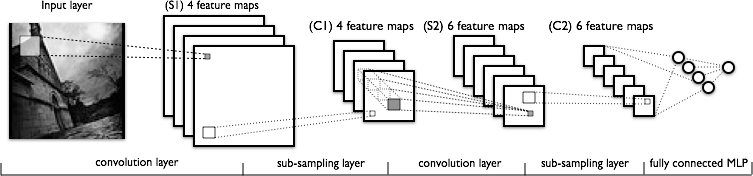
    Input--->(Conv--->ReLU--->Pooling)\*2--->FC--->ReLU--->Output

通常输入层大小一般为2的整数倍，如32，64，96，224，384等。

通常卷积层使用较小的filter，如3\*3，最多也就5\*5，并且使用stride=1（stride表示卷积时对input进行每次移动一步，上下左右都是），并且对原来的input进行padding（使用0元素填充）。

Pooling层用于对卷积结果进行降低维度，例如选择2\*2的区域对卷积层进行降低维度，则选择2\*2区域的最大值作为输出，这样卷积层的维度就降为之前一半。

卷积层和Pooling层是CNN的核心所在，下图显示了CNN的结构图：



较低层是用来卷积以及抽样的，较高的层用来进行逻辑回归。

**一篇好的CNN文章：**

# [Deep Learning模型之：CNN卷积神经网络（一）深度解析CNN](http://www.cnblogs.com/nsnow/p/4562363.html)

http://m.blog.csdn.net/blog/wu010555688/24487301

本文整理了网上几位大牛的博客，详细地讲解了CNN的基础结构与核心思想，欢迎交流。

### 

[**[1]**Deep learning简介](http://blog.csdn.net/u010555688/article/details/24438059)

[**[2]**Deep Learning训练过程](http://blog.csdn.net/u010555688/article/details/24438251)

[**[3]**Deep Learning模型之：CNN卷积神经网络推导和实现](http://blog.csdn.net/u010555688/article/details/38780767)

[**[4]**Deep Learning模型之：CNN的反向求导及练习](http://blog.csdn.net/u010555688/article/details/38780807)

[[5]](http://blog.csdn.net/u010555688/article/details/38780807)[Deep Learning模型之：CNN卷积神经网络（一）深度解析CNN](http://blog.csdn.net/u010555688/article/details/24487301)

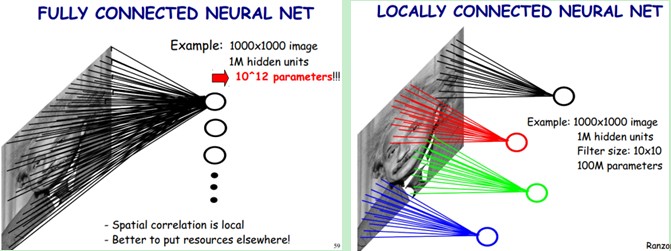
[6][Deep Learning模型之：CNN卷积神经网络（二）文字识别系统LeNet-5](http://blog.csdn.net/u010555688/article/details/24848367)

[7][Deep Learning模型之：CNN卷积神经网络（三）CNN常见问题总结](http://blog.csdn.net/u010555688/article/details/26353333)

**1. 概述**

   卷积神经网络是一种特殊的深层的神经网络模型，它的特殊性体现在两个方面，一方面它的神经元间的连接是非全连接的， 另一方面同一层中某些神经元之间的连接的权重是共享的（即相同的）。它的非全连接和权值共享的网络结构使之更类似于生物 神经网络，降低了网络模型的复杂度（对于很难学习的深层结构来说，这是非常重要的），减少了权值的数量。

     回想一下BP神经网络。BP网络每一层节点是一个线性的一维排列状态，层与层的网络节点之间是全连接的。这样设想一下，如果BP网络中层与层之间的节点连接不再是全连接，而是局部连接的。这样，就是一种最简单的一维卷积网络。如果我们把上述这个思路扩展到二维，这就是我们在大多数参考资料上看到的卷积神经网络。具体参看下图：



       上图左：全连接网络。如果我们有1000x1000像素的图像，有1百万个隐层神经元，每个隐层神经元都连接图像的每一个像素点，就有1000x1000x1000000=10^12个连接，也就是10^12个权值参数。

       上图右：局部连接网络，每一个节点与上层节点同位置附件10x10的窗口相连接，则1百万个隐层神经元就只有100w乘以100，即10^8个参数。其权值连接个数比原来减少了四个数量级。

      根据BP网络信号前向传递过程，我们可以很容易计算网络节点的输出。例如，对于上图中被标注为红色节点的净输入，就等于所有与红线相连接的上一层神经元节点值与红色线表示的权值之积的累加。这样的计算过程，很多书上称其为卷积。

       事实上，对于数字滤波而言，其滤波器的系数通常是对称的。否则，卷积的计算需要先反向对折，然后进行乘累加的计算。上述神经网络权值满足对称吗？我想答案是否定的！所以，上述称其为卷积运算，显然是有失偏颇的。但这并不重要，仅仅是一个名词称谓而已。只是，搞信号处理的人，在初次接触卷积神经网络的时候，带来了一些理解上的误区。

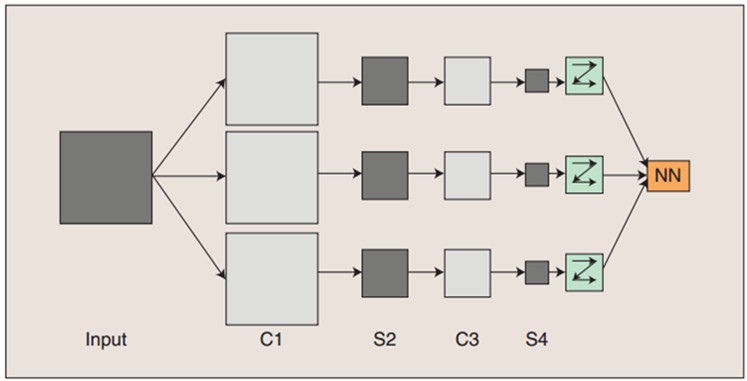
        卷积神经网络另外一个特性是权值共享。例如，就上面右边那幅图来说，权值共享，也就是说所有的红色线标注的连接权值相同。这一点，初学者容易产生误解。

        上面描述的只是单层网络结构，前A&T Shannon Lab   的  Yann LeCun等人据此提出了基于卷积神经网络的一个文字识别系统 LeNet-5。该系统90年代就被用于银行手写数字的识别。

**2、 CNN的结构**

卷积网络是为识别二维形状而特殊设计的一个多层感知器，这种网络结构对平移、比例缩放、倾斜或者共他形式的变形具有高度不变性。 这些良好的性能是网络在有监督方式下学会的，网络的结构主要有稀疏连接和权值共享两个特点，包括如下形式的约束：  
1、 [特征提取](http://deeplearning.stanford.edu/wiki/index.php/%E5%8D%B7%E7%A7%AF%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96)。每一个神经元从上一层的局部接受域得到突触输人，因而迫使它提取局部特征。一旦一个特征被提取出来， 只要它相对于其他特征的位置被近似地保留下来，它的精确位置就变得没有那么重要了。  
2 、特征映射。网络的每一个计算层都是由多个特征映射组成的，每个特征映射都是平面形式的。平面中单独的神经元在约束下共享 相同的突触权值集，这种结构形式具有如下的有益效果：a.平移不变性。b.自由参数数量的缩减(通过权值共享实现)。  
3、[子抽样](http://deeplearning.stanford.edu/wiki/index.php/%E6%B1%A0%E5%8C%96)。每个卷积层后面跟着一个实现局部平均和子抽样的计算层，由此特征映射的分辨率降低。这种操作具有使特征映射的输出对平移和其他 形式的变形的敏感度下降的作用。

卷积神经网络是一个多层的神经网络，每层由多个二维平面组成，而每个平面由多个独立神经元组成。



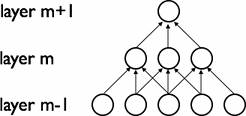
       图：卷积神经网络的概念示范：输入图像通过和三个可训练的滤波器和可加偏置进行卷积，卷积后在C1层产生三个特征映射图，然后特征映射图中每组的四个像素再进行求和，加权值，加偏置，通过一个Sigmoid函数得到三个S2层的特征映射图。这些映射图再进过滤波得到C3层。这个层级结构再和S2一样产生S4。最终，这些像素值被光栅化，并连接成一个向量输入到传统的神经网络，得到输出。

       一般地，C层为特征提取层，每个神经元的输入与前一层的局部感受野相连，并提取该局部的特征，一旦该局部特征被提取后，它与其他特征间的位置关系也随之确定下来；S层是特征映射层，网络的每个计算层由多个特征映射组成，每个特征映射为一个平面，平面上所有神经元的权值相等。特征映射结构采用影响函数核小的sigmoid函数作为卷积网络的激活函数，使得特征映射具有位移不变性。

       此外，由于一个映射面上的神经元共享权值，因而减少了网络自由参数的个数，降低了网络参数选择的复杂度。卷积神经网络中的每一个特征提取层（C-层）都紧跟着一个用来求局部平均与二次提取的计算层（S-层），这种特有的两次特征提取结构使网络在识别时对输入样本有较高的畸变容忍能力。

2.1 稀疏连接(Sparse Connectivity)

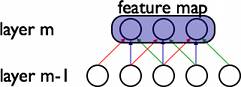
卷积网络通过在相邻两层之间强制使用局部连接模式来利用图像的空间局部特性，在第m层的隐层单元只与第m-1层的输入单元的局部区域有连接，第m-1层的这些局部 区域被称为空间连续的接受域。我们可以将这种结构描述如下：  
设第m-1层为视网膜输入层，第m层的接受域的宽度为3，也就是说该层的每个单元与且仅与输入层的3个相邻的神经元相连，第m层与第m+1层具有类似的链接规则，如下图所示。



    可以看到m+1层的神经元相对于第m层的接受域的宽度也为3，但相对于输入层的接受域为5，这种结构将学习到的过滤器（对应于输入信号中被最大激活的单元）限制在局部空间 模式（因为每个单元对它接受域外的variation不做反应）。从上图也可以看出，多个这样的层堆叠起来后，会使得过滤器（不再是线性的）逐渐成为全局的（也就是覆盖到了更 大的视觉区域）。例如上图中第m+1层的神经元可以对宽度为5的输入进行一个非线性的特征编码。

2.2 权值共享(Shared Weights)

在卷积网络中，每个稀疏过滤器hi通过共享权值都会覆盖整个可视域，这些共享权值的单元构成一个特征映射，如下图所示。

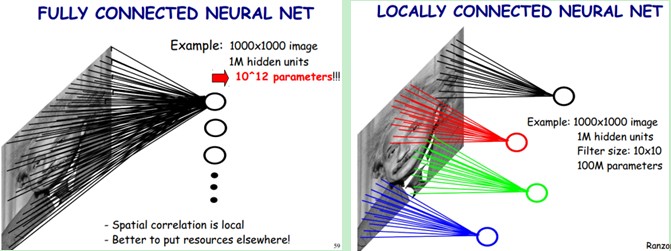


    在图中，有3个隐层单元，他们属于同一个特征映射。同种颜色的链接的权值是相同的，我们仍然可以使用梯度下降的方法来学习这些权值，只需要对原始算法做一些小的改动， 这里共享权值的梯度是所有共享参数的梯度的总和。我们不禁会问为什么要权重共享呢？一方面，重复单元能够对特征进行识别，而不考虑它在可视域中的位置。另一方面，权值 共享使得我们能更有效的进行特征抽取，因为它极大的减少了需要学习的自由变量的个数。通过控制模型的规模，卷积网络对视觉问题可以具有很好的泛化能力。

举例讲解：

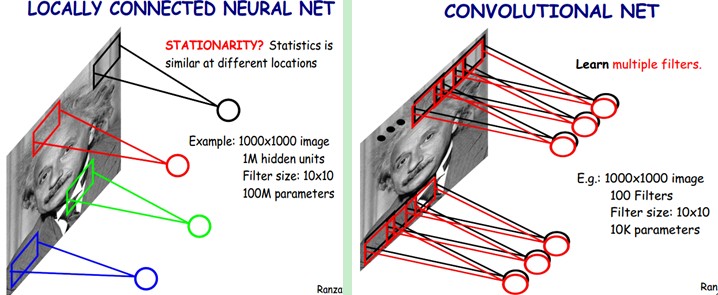
     上面聊到，好像CNN一个牛逼的地方就在于通过感受野和权值共享减少了神经网络需要训练的参数的个数。那究竟是啥的呢？

     下图左：如果我们有1000x1000像素的图像，有1百万个隐层神经元，那么他们全连接的话（每个隐层神经元都连接图像的每一个像素点），就有1000x1000x1000000=10^12个连接，也就是10^12个权值参数。然而图像的空间联系是局部的，就像人是通过一个局部的感受野去感受外界图像一样，每一个神经元都不需要对全局图像做感受，每个神经元只感受局部的图像区域，然后在更高层，将这些感受不同局部的神经元综合起来就可以得到全局的信息了。这样，我们就可以减少连接的数目，也就是减少神经网络需要训练的权值参数的个数了。如下图右：假如局部感受野是10x10，隐层每个感受野只需要和这10x10的局部图像相连接，所以1百万个隐层神经元就只有一亿个连接，即10^8个参数。比原来减少了四个0（数量级），这样训练起来就没那么费力了，但还是感觉很多的啊，那还有啥办法没？

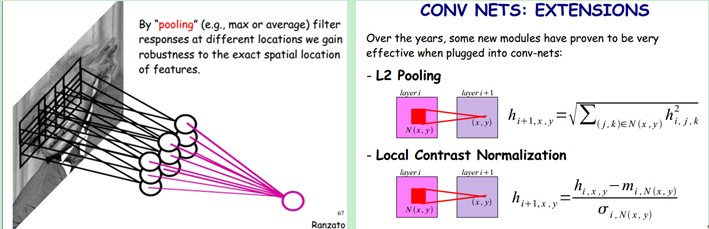


       我们知道，隐含层的每一个神经元都连接10x10个图像区域，也就是说每一个神经元存在10x10=100个连接权值参数。那如果我们每个神经元这100个参数是相同的呢？也就是说每个神经元用的是同一个卷积核去卷积图像。这样我们就只有多少个参数？？只有100个参数啊！！！亲！不管你隐层的神经元个数有多少，两层间的连接我只有100个参数啊！亲！这就是权值共享啊！亲！这就是卷积神经网络的主打卖点啊！亲！（有点烦了，呵呵）也许你会问，这样做靠谱吗？为什么可行呢？这个……共同学习。

       好了，你就会想，这样提取特征也忒不靠谱吧，这样你只提取了一种特征啊？对了，真聪明，我们需要提取多种特征对不？假如一种滤波器，也就是一种卷积核就是提出图像的一种特征，例如某个方向的边缘。那么我们需要提取不同的特征，怎么办，加多几种滤波器不就行了吗？对了。所以假设我们加到100种滤波器，每种滤波器的参数不一样，表示它提出输入图像的不同特征，例如不同的边缘。这样每种滤波器去卷积图像就得到对图像的不同特征的放映，我们称之为Feature Map。所以100种卷积核就有100个Feature Map。这100个Feature Map就组成了一层神经元。到这个时候明了了吧。我们这一层有多少个参数了？100种卷积核x每种卷积核共享100个参数=100x100=10K，也就是1万个参数。才1万个参数啊！亲！（又来了，受不了了！）见下图右：不同的颜色表达不同的滤波器。



       嘿哟，遗漏一个问题了。刚才说隐层的参数个数和隐层的神经元个数无关，只和滤波器的大小和滤波器种类的多少有关。那么隐层的神经元个数怎么确定呢？它和原图像，也就是输入的大小（神经元个数）、滤波器的大小和滤波器在图像中的滑动步长都有关！例如，我的图像是1000x1000像素，而滤波器大小是10x10，假设滤波器没有重叠，也就是步长为10，这样隐层的神经元个数就是(1000x1000 )/ (10x10)=100x100个神经元了，假设步长是8，也就是卷积核会重叠两个像素，那么……我就不算了，思想懂了就好。注意了，这只是一种滤波器，也就是一个Feature Map的神经元个数哦，如果100个Feature Map就是100倍了。由此可见，图像越大，神经元个数和需要训练的权值参数个数的贫富差距就越大。



      需要注意的一点是，上面的讨论都没有考虑每个神经元的偏置部分。所以权值个数需要加1 。这个也是同一种滤波器共享的。

      总之，卷积网络的核心思想是将：局部感受野、权值共享（或者权值复制）以及时间或空间亚采样这三种结构思想结合起来获得了某种程度的位移、尺度、形变不变性。

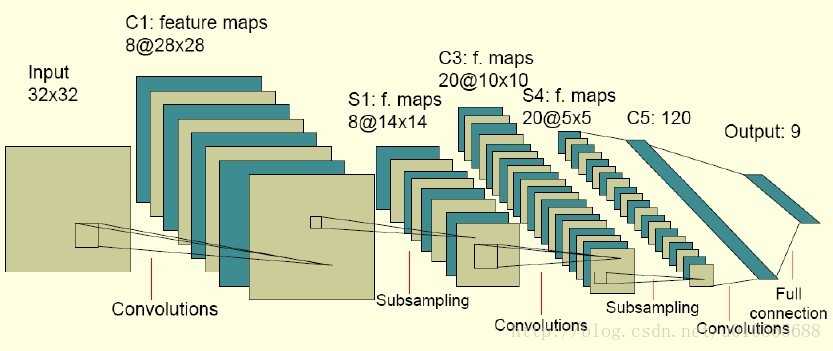
**2.3 The Full Model**

        卷积神经网络是一个多层的神经网络，每层由多个二维平面组成，而每个平面由多个独立神经元组成。网络中包含一些简单元和复杂元，分别记为S-元 和C-元。S-元聚合在一起组成S-面，S-面聚合在一起组成S-层，用Us表示。C-元、C-面和C-层(Us)之间存在类似的关系。网络的任一中间级由S-层与C-层 串接而成，而输入级只含一层，它直接接受二维视觉模式，样本特征提取步骤已嵌入到卷积神经网络模型的互联结构中。

一般地，Us为特征提取层(子采样层)，每个神经元的输入与前一层的局部感受野相连，并提取该局部的特征，一旦该局部特征被提取后，它与其他特征间的位置关系 也随之确定下来；

Uc是特征映射层(卷积层)，网络的每个计算层由多个特征映射组成，每个特征映射为一个平面，平面上所有神经元的权值相等。特征映射结构采用 影响函数核小的sigmoid函数作为卷积网络的激活函数，使得特征映射具有位移不变性。此外，由于 一个映射面上的神经元共享权值，因而减少了网络自由参数的个数，降低了网络参数选择的复杂度。卷积神经网络中的每一个特征提取层(S-层)都紧跟着一个 用来求局部平均与二次提取的计算层(C-层)，这种特有的两次特征提取结构使网络在识别时对输入样本有较高的畸变容忍能力。

下图是一个卷积网络的实例，在博文”[Deep Learning模型之：CNN卷积神经网络（二） 文字识别系统LeNet-5](http://blog.csdn.net/u010555688/article/details/24848367)“中有详细讲解：



 图中的卷积网络工作流程如下，输入层由32×32个感知节点组成，接收原始图像。然后，计算流程在卷积和子抽样之间交替进行，如下所述：

    第一隐藏层进行卷积，它由8个特征映射组成，每个特征映射由28×28个神经元组成，每个神经元指定一个 5×5 的接受域；

    第二隐藏层实现子 抽样和局部平均，它同样由 8 个特征映射组成，但其每个特征映射由14×14 个神经元组成。每个神经元具有一个 2×2 的接受域，一个可训练 系数，一个可训练偏置和一个 sigmoid 激活函数。可训练系数和偏置控制神经元的操作点。

    第三隐藏层进行第二次卷积，它由 20 个特征映射组 成，每个特征映射由 10×10 个神经元组成。该隐藏层中的每个神经元可能具有和下一个隐藏层几个特征映射相连的突触连接，它以与第一个卷积 层相似的方式操作。

    第四个隐藏层进行第二次子抽样和局部平均汁算。它由 20 个特征映射组成，但每个特征映射由 5×5 个神经元组成，它以 与第一次抽样相似的方式操作。

    第五个隐藏层实现卷积的最后阶段，它由 120 个神经元组成，每个神经元指定一个 5×5 的接受域。

    最后是个全 连接层，得到输出向量。

    相继的计算层在卷积和抽样之间的连续交替，我们得到一个“双尖塔”的效果，也就是在每个卷积或抽样层，随着空 间分辨率下降，与相应的前一层相比特征映射的数量增加。卷积之后进行子抽样的思想是受到动物视觉系统中的“简单的”细胞后面跟着“复杂的”细胞的想法的启发而产生的。

图中所示的多层感知器包含近似 100000 个突触连接，但只有大约2600 个自由参数(每个特征映射为一个平面，平面上所有神经元的权值相等)。自由参数在数量上显著地减少是通过权值共享获得 的，学习机器的能力（以 VC 维的形式度量）因而下降，这又提高它的泛化能力。而且它对自由参数的调整通过反向传播学习的随机形式来实 现。另一个显著的特点是使用权值共享使得以并行形式实现卷积网络变得可能。这是卷积网络对全连接的多层感知器而言的另一个优点。

3、 CNN的训练

        神经网络用于模式识别的主流是有指导学习网络，无指导学习网络更多的是用于聚类分析。对于有指导的模式识别，由于任一样本的类别是已知的，样本在空间的分布不再是依据其自然分布倾向来划分，而是要根据同类样本在空间的分布及不同类样本之间的分离程度找一种适当的空间划分方法，或者找到一个分类边界，使得不同类样本分别位于不同的区域内。这就需要一个长时间且复杂的学习过程，不断调整用以划分样本空间的分类边界的位置，使尽可能少的样本被划分到非同类区域中。

       卷积网络在本质上是一种输入到输出的映射，它能够学习大量的输入与输出之间的映射关系，而不需要任何输入和输出之间的精确的数学表达式，只要用已知的模式对卷积网络加以训练，网络就具有输入输出对之间的映射能力。卷积网络执行的是有导师训练，所以其样本集是由形如：（输入向量，理想输出向量）的向量对构成的。所有这些向量对，都应该是来源于网络即将模拟的系统的实际“运行”结果。它们可以是从实际运行系统中采集来的。在开始训练前，所有的权都应该用一些不同的小随机数进行初始化。“小随机数”用来保证网络不会因权值过大而进入饱和状态，从而导致训练失败；“不同”用来保证网络可以正常地学习。实际上，如果用相同的数去初始化权矩阵，则网络无能力学习。

       训练算法与传统的BP算法差不多。主要包括4步，这4步被分为两个阶段：

第一阶段，向前传播阶段：

a）从样本集中取一个样本(X,Yp)，将X输入网络；

b）计算相应的实际输出Op。

      在此阶段，信息从输入层经过逐级的变换，传送到输出层。这个过程也是网络在完成训练后正常运行时执行的过程。在此过程中，网络执行的是计算（实际上就是输入与每层的权值矩阵相点乘，得到最后的输出结果）：

          Op=Fn（…（F2（F1（XpW（1））W（2））…）W（n））

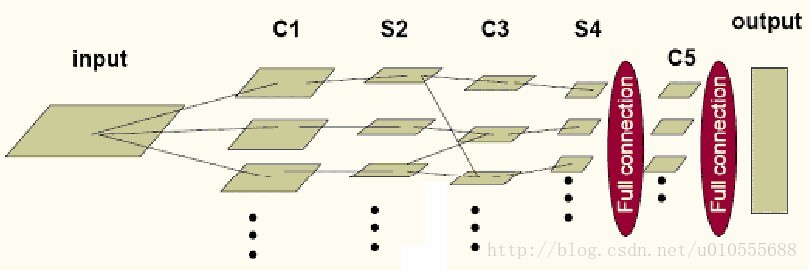
第二阶段，向后传播阶段

a）算实际输出Op与相应的理想输出Yp的差；

b）按极小化误差的方法反向传播调整权矩阵。

4、 CNN的学习

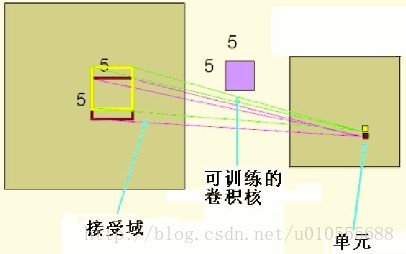
总体而言，卷积网络可以简化为下图所示模型：



    其中，input 到C1、S4到C5、C5到output是全连接，C1到S2、C3到S4是一一对应的连接，S2到C3为了消除网络对称性，去掉了一部分连接， 可以让特征映射更具多样性。需要注意的是 C5 卷积核的尺寸要和 S4 的输出相同，只有这样才能保证输出是一维向量。

4.1 卷积层的学习

卷积层的典型结构如下图所示：



**卷积层的前馈运算是通过如下算法实现的：**

卷积层的输出= Sigmoid( Sum(卷积) +偏移量)

**其中卷积核和偏移量都是可训练的。下面是其核心代码：**

ConvolutionLayer::fprop(input,output) {

//取得卷积核的个数

int n=kernel.GetDim(0);

for (int i=0;i<n;i++) {

//第i个卷积核对应输入层第a个特征映射，输出层的第b个特征映射

//这个卷积核可以形象的看作是从输入层第a个特征映射到输出层的第b个特征映射的一个链接

int a=table[i][0], b=table[i][1];

//用第i个卷积核和输入层第a个特征映射做卷积

convolution = Conv(input[a],kernel[i]);

//把卷积结果求和

sum[b] +=convolution;

}

for (i=0;i<(int)bias.size();i++) {

//加上偏移量

sum[i] += bias[i];

}

//调用Sigmoid函数

output = Sigmoid(sum);

}

    其中，input是 n1×n2×n3 的矩阵，n1是输入层特征映射的个数，n2是输入层特征映射的宽度，n3是输入层特征映射的高度。output, sum, convolution, bias是n1×(n2-kw+1)×(n3-kh+1)的矩阵，kw,kh是卷积核的宽度高度(图中是5×5)。kernel是卷积核矩阵。table是连接表，即如果第a输入和第b个输出之间 有连接，table里就会有[a,b]这一项，而且每个连接都对应一个卷积核。

**卷积层的反馈运算的核心代码如下：**

ConvolutionLayer::bprop(input,output,in\_dx,out\_dx) {

//梯度通过DSigmoid反传

sum\_dx = DSigmoid(out\_dx);

//计算bias的梯度

for (i=0;i<bias.size();i++) {

bias\_dx[i] = sum\_dx[i];

}

//取得卷积核的个数

int n=kernel.GetDim(0);

for (int i=0;i<n;i++)

{

int a=table[i][0],b=table[i][1];

//用第i个卷积核和第b个输出层反向卷积（即输出层的一点乘卷积模板返回给输入层），并把结果累加到第a个输入层

input\_dx[a] += DConv(sum\_dx[b],kernel[i]);

//用同样的方法计算卷积模板的梯度

kernel\_dx[i] += DConv(sum\_dx[b],input[a]);

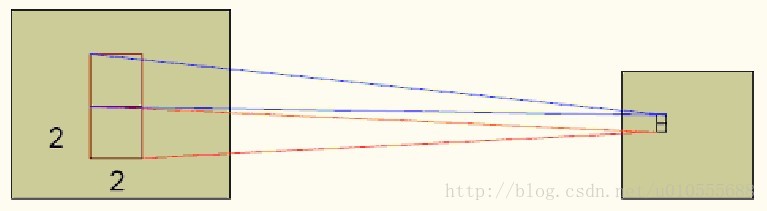
}

}

其中in\_dx,out\_dx 的结构和 input,output 相同，代表的是相应点的梯度。

4.2 子采样层的学习

**子采样层的典型结构如下图所示：**



**类似的子采样层的输出的计算式为：**

**输出= Sigmoid( 采样\*权重 +偏移量)**

**其核心代码如下：**

SubSamplingLayer::fprop(input,output) {

int n1= input.GetDim(0);

int n2= input.GetDim(1);

int n3= input.GetDim(2);

for (int i=0;i<n1;i++) {

for (int j=0;j<n2;j++) {

for (int k=0;k<n3;k++) {

//coeff 是可训练的权重，sw 、sh 是采样窗口的尺寸。

sub[i][j/sw][k/sh] += input[i][j][k]\*coeff[i];

}

}

}

for (i=0;i<n1;i++) {

//加上偏移量

sum[i] = sub[i] + bias[i];

}

output = Sigmoid(sum);

}

子采样层的反馈运算的核心代码如下：

SubSamplingLayer::bprop(input,output,in\_dx,out\_dx) {

//梯度通过DSigmoid反传

sum\_dx = DSigmoid(out\_dx);

//计算bias和coeff的梯度

for (i=0;i<n1;i++) {

coeff\_dx[i] = 0;

bias\_dx[i] = 0;

for (j=0;j<n2/sw;j++)

for (k=0;k<n3/sh;k++) {

coeff\_dx[i] += sub[j][k]\*sum\_dx[i][j][k];

bias\_dx[i] += sum\_dx[i][j][k]);

}

}

for (i=0;i<n1;i++) {

for (j=0;j<n2;j++)

for (k=0;k<n3;k++) {

in\_dx[i][j][k] = coeff[i]\*sum\_dx[i][j/sw][k/sh];

}

}

}

**5、CNN的优点**

        卷积神经网络CNN主要用来识别位移、缩放及其他形式扭曲不变性的二维图形。由于CNN的特征检测层通过训练数据进行学习，所以在使用CNN时，避免了显式的特征抽取，而隐式地从训练数据中进行学习；再者由于同一特征映射面上的神经元权值相同，所以网络可以并行学习，这也是卷积网络相对于神经元彼此相连网络的一大优势。卷积神经网络以其局部权值共享的特殊结构在语音识别和图像处理方面有着独特的优越性，其布局更接近于实际的生物神经网络，权值共享降低了网络的复杂性，特别是多维输入向量的图像可以直接输入网络这一特点避免了特征提取和分类过程中数据重建的复杂度。

        流的分类方式几乎都是基于统计特征的，这就意味着在进行分辨前必须提取某些特征。然而，显式的特征提取并不容易，在一些应用问题中也并非总是可靠的。卷积神经网络，它避免了显式的特征取样，隐式地从训练数据中进行学习。这使得卷积神经网络明显有别于其他基于神经网络的分类器，通过结构重组和减少权值将特征提取功能融合进多层感知器。它可以直接处理灰度图片，能够直接用于处理基于图像的分类。

       卷积网络较一般神经网络在图像处理方面有如下优点： a）输入图像和网络的拓扑结构能很好的吻合；b）特征提取和模式分类同时进行，并同时在训练中产生；c）权重共享可以减少网络的训练参数，使神经网络结构变得更简单，适应性更强。

**6、CNN的实现问题**

       CNNs中这种层间联系和空域信息的紧密关系，使其适于图像处理和理解。而且，其在自动提取图像的显著特征方面还表现出了比较优的性能。在一些例子当中，Gabor滤波器已经被使用在一个初始化预处理的步骤中，以达到模拟人类视觉系统对视觉刺激的响应。在目前大部分的工作中，研究者将CNNs应用到了多种机器学习问题中，包括人脸识别，文档分析和语言检测等。为了达到寻找视频中帧与帧之间的相干性的目的，目前CNNs通过一个时间相干性去训练，但这个不是CNNs特有的。

      由于卷积神经网络采用BP网络相同的算法。所以，采用现有BP网络就可以实现。开源的神经网络代码FAAN可以利用。这个开源的实现采用了一些代码优化技术，有双精度，单精度，定点运算三个不同的版本。

      由于经典的BP网络是一个一维节点分布排列，而卷积神经网络是二维网络结构。所以，要把卷积神经网络的每一层，按照一定的顺序和规则映射为一维节点分布，然后，按照这个分布创建一个多层反向传播算法的网络结构，就可以按照一般的BP训练算法去学习网络参数。对于实际环境中新样本的预测，也采用BP算法中相同信号前向传递算法进行。具体细节也可以参考网上的一个开源代码，链接如下：

<http://www.codeproject.com/Articles/16650/Neural-Network-for-Recognition-of-Handwritten-Digi>

 注：这个代码在创建CNN的时候有个明显的BUG，如果你看明白了我上面对简化的LeNet-5的结构描述，一眼就会找出问题所在。

**Tensorflow下的CNN**

**快速安装python2和python3的方法：**分别安装Anaconda2和Anaconda3。(Anaconda一个开源的 [Python](http://baike.baidu.com/view/21087.htm) 发行版本，包含了conda、Python 等180 多个科学包及其依赖项)

一个快速的源：https://mirror.tuna.tsinghua.edu.cn/anaconda/archive/

一、win7-64安装tensorflow

## 相关背景

网上现有的方法都是教如何在windows下安装虚拟机或者安装docker来完成TensorFlow的安装。得益于google目前已经支持直接在windows直接安装，本文记录下在windows下通过Anaconda安装TensorFlow的过程。

## Why Anaconda

因为Anaconda能够提供强大的包管理功能及虚拟环境，为TensorFlow开发提供便利。Anaconda利用工具/命令conda来进行package和environment的管理，并且已经包含了Python和相关的配套工具。

## 详细步骤

**1、安装Anaconda**   
直接在官网下载并安装Anaconda，这里选择64版本。   
（<https://www.continuum.io/downloads#windows>）太慢了，我们用一个快速的源：https://mirror.tuna.tsinghua.edu.cn/anaconda/archive/

注意，windows下安装TensorFlow，要求python版本是3.5，64位。   
安装完Anaconda，也就安装了python3.5等相关工具

**2、安装完成后，打开Anaconda Prompt，创建TensorFlow虚拟环境**   
在Prompt中输入：

>>> conda create -n tensorflow python=3.5

**3、进入TensorFlow环境，输入**

>>> activate tensorflow

在命令行前，你就可以看到在输入提示符前加了(tensorflow)   
变成了这样：

（tensorflow）...>>>

如果要退出该虚拟环境，输入 deactive tensorflow 即可

**4、安装tensorflow**   
在activate tensorflow状态下直接输入：

>>> pip install tensorflow (可能需要更新pip版本，按提示命令更新即可)

**5、测试tensorflow是否安装成功**   
按照以下步骤验证即可：

$ >>> python

>>> import tensorflow as tf

>>> hello = tf.constant('Hello, TensorFlow!')

>>> sess = tf.Session()

>>> print (sess.run(hello)) #py3 print加括号

Hello, TensorFlow!

>>> a = tf.constant(10)

>>> b = tf.constant(32)

>>> print (sess.run(a+b))

42

可以开始使用了。方法：

1）打开Anaconda Prompt，输入activate tensorflow，激活环境

2) 输入python进行编程即可

或者 用文本编辑器写好.py 直接在2)运行python /xx/xx/xxx.py即可

**可视化Tensorflow:**

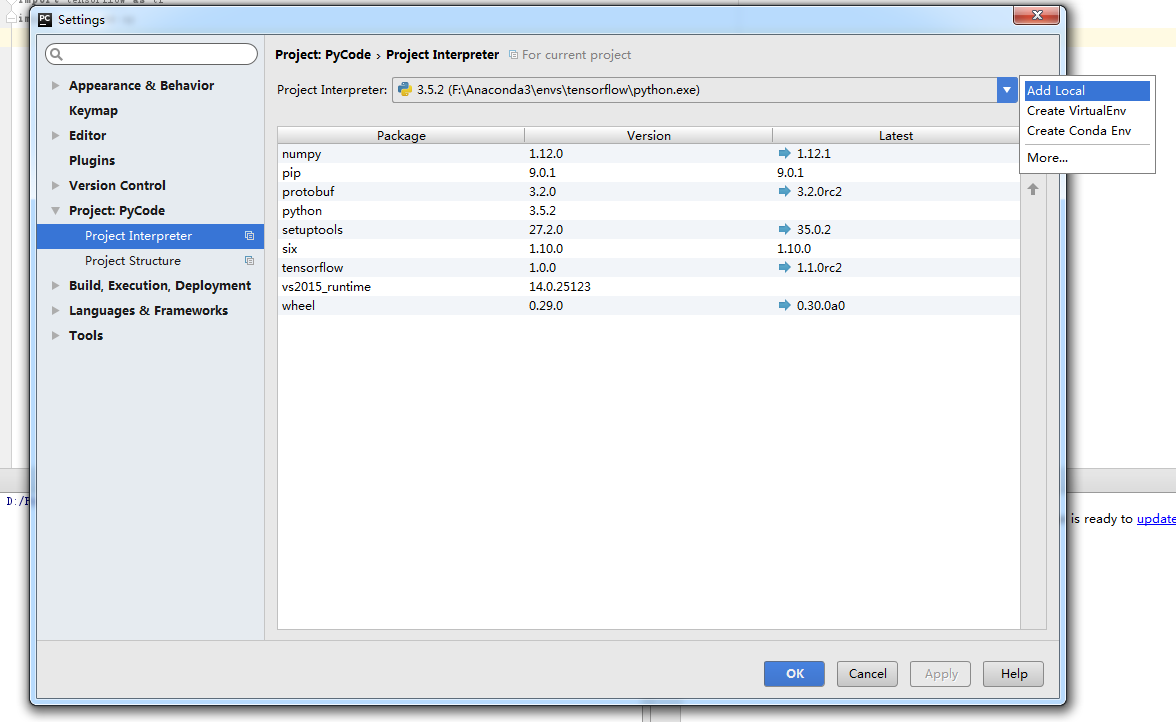
**http://wiki.jikexueyuan.com/project/tensorflow-zh/how\_tos/summaries\_and\_tensorboard.html**

**pycharm下使用tensorflow**

**需要做更新python解释器的操作，具体如下：**

**1.按照上述步骤装好aconda3 下的tensorflow**

**2.在pycharm中找到下图：**

****

**3.在add local中添加本地的ancoda3->envs->tensorflow->python.exe 解释器，点击ok，此时python解释器变被更新为tensorflow可用的版本，即可在pycharm中编写程序了。**

# 二、TensorFlow基本使用方法

使用 TensorFlow, 你必须明白 TensorFlow:

* 使用图 (graph) 来表示计算任务.
* 在被称之为 会话 (Session) 的上下文 (context) 中执行图.
* 使用 tensor 表示数据.
* 通过 变量 (Variable) 维护状态.
* 使用 feed 和 fetch 可以为任意的操作(arbitrary operation) 赋值或者从其中获取数据.

## 综述

TensorFlow 是一个编程系统, 使用图来表示计算任务. 图中的节点被称之为 op (operation 的缩写). 一个 op 获得 0 个或多个 Tensor, 执行计算, 产生 0 个或多个 Tensor. 每个 Tensor 是一个类型化的多维数组. 例如, 你可以将一小组图像集表示为一个四维浮点数数组, 这四个维度分别是 [batch, height, width, channels].

一个 TensorFlow 图描述了计算的过程. 为了进行计算, 图必须在 会话 里被启动. 会话 将图的 op 分发到诸如 CPU 或 GPU 之类的 设备 上, 同时提供执行 op 的方法. 这些方法执行后, 将产生的 tensor 返回. 在 Python 语言中, 返回的 tensor 是 [numpy](http://www.numpy.org/) ndarray 对象; 在 C 和 C++ 语言中, 返回的 tensor 是tensorflow::Tensor 实例.

## 计算图

TensorFlow 程序通常被组织成一个构建阶段和一个执行阶段. 在构建阶段, op 的执行步骤 被描述成一个图. 在执行阶段, 使用会话执行执行图中的 op.

例如, 通常在构建阶段创建一个图来表示和训练神经网络, 然后在执行阶段反复执行图中的训练 op.

TensorFlow 支持 C, C++, Python 编程语言. 目前, TensorFlow 的 Python 库更加易用, 它提供了大量的辅助函数来简化构建图的工作, 这些函数尚未被 C 和 C++ 库支持.

三种语言的会话库 (session libraries) 是一致的.

### 构建图

构建图的第一步, 是创建源 op (source op). 源 op 不需要任何输入, 例如 常量 (Constant). 源 op 的输出被传递给其它 op 做运算.

Python 库中, op 构造器的返回值代表被构造出的 op 的输出, 这些返回值可以传递给其它 op 构造器作为输入.

TensorFlow Python 库有一个默认图 (default graph), op 构造器可以为其增加节点. 这个默认图对 许多程序来说已经足够用了. 阅读 [Graph 类](http://www.tensorfly.cn/tfdoc/api_docs/python/framework.html#Graph) 文档 来了解如何管理多个图.

import tensorflow as tf

# 创建一个常量 op, 产生一个 1x2 矩阵. 这个 op 被作为一个节点

# 加到默认图中.

#

# 构造器的返回值代表该常量 op 的返回值.

matrix1 = tf.constant([[3., 3.]])

# 创建另外一个常量 op, 产生一个 2x1 矩阵.

matrix2 = tf.constant([[2.],[2.]])

# 创建一个矩阵乘法 matmul op , 把 'matrix1' 和 'matrix2' 作为输入.

# 返回值 'product' 代表矩阵乘法的结果.

product = tf.matmul(matrix1, matrix2)

默认图现在有三个节点, 两个 constant() op, 和一个matmul() op. 为了真正进行矩阵相乘运算, 并得到矩阵乘法的 结果, 你必须在会话里启动这个图.

### 在一个会话中启动图

构造阶段完成后, 才能启动图. 启动图的第一步是创建一个 Session 对象, 如果无任何创建参数, 会话构造器将启动默认图.

欲了解完整的会话 API, 请阅读[Session 类](http://www.tensorfly.cn/tfdoc/api_docs/python/client.html#session-management).

# 启动默认图.

sess = tf.Session()

# 调用 sess 的 'run()' 方法来执行矩阵乘法 op, 传入 'product' 作为该方法的参数.

# 上面提到, 'product' 代表了矩阵乘法 op 的输出, 传入它是向方法表明, 我们希望取回

# 矩阵乘法 op 的输出.

#

# 整个执行过程是自动化的, 会话负责传递 op 所需的全部输入. op 通常是并发执行的.

#

# 函数调用 'run(product)' 触发了图中三个 op (两个常量 op 和一个矩阵乘法 op) 的执行.

#

# 返回值 'result' 是一个 numpy `ndarray` 对象.

result = sess.run(product)

print result

# ==> [[ 12.]]

# 任务完成, 关闭会话.

sess.close()

Session 对象在使用完后需要关闭以释放资源. 除了显式调用 close 外, 也可以使用 "with" 代码块 来自动完成关闭动作.

with tf.Session() as sess:

result = sess.run([product])

print result

在实现上, TensorFlow 将图形定义转换成分布式执行的操作, 以充分利用可用的计算资源(如 CPU 或 GPU). 一般你不需要显式指定使用 CPU 还是 GPU, TensorFlow 能自动检测. 如果检测到 GPU, TensorFlow 会尽可能地利用找到的第一个 GPU 来执行操作.

如果机器上有超过一个可用的 GPU, 除第一个外的其它 GPU 默认是不参与计算的. 为了让 TensorFlow 使用这些 GPU, 你必须将 op 明确指派给它们执行. with...Device 语句用来指派特定的 CPU 或 GPU 执行操作:

with tf.Session() as sess:

with tf.device("/gpu:1"):

matrix1 = tf.constant([[3., 3.]])

matrix2 = tf.constant([[2.],[2.]])

product = tf.matmul(matrix1, matrix2)

...

设备用字符串进行标识. 目前支持的设备包括:

* "/cpu:0": 机器的 CPU.
* "/gpu:0": 机器的第一个 GPU, 如果有的话.
* "/gpu:1": 机器的第二个 GPU, 以此类推.

阅读[使用GPU](http://www.tensorfly.cn/tfdoc/how_tos/using_gpu.html)章节, 了解 TensorFlow GPU 使用的更多信息.

## 交互式使用

文档中的 Python 示例使用一个会话 [Session](http://www.tensorfly.cn/tfdoc/api_docs/python/client.html#Session) 来 启动图, 并调用 [Session.run()](http://www.tensorfly.cn/tfdoc/api_docs/python/client.html#Session.run) 方法执行操作.

为了便于使用诸如 [IPython](http://ipython.org/) 之类的 Python 交互环境, 可以使用 [InteractiveSession](http://www.tensorfly.cn/tfdoc/api_docs/python/client.html#InteractiveSession) 代替 Session 类, 使用 [Tensor.eval()](http://www.tensorfly.cn/tfdoc/api_docs/python/framework.html#Tensor.eval) 和 [Operation.run()](http://www.tensorfly.cn/tfdoc/api_docs/python/framework.html#Operation.run) 方法代替 Session.run(). 这样可以避免使用一个变量来持有会话.

# 进入一个交互式 TensorFlow 会话.

import tensorflow as tf

sess = tf.InteractiveSession()

x = tf.Variable([1.0, 2.0])

a = tf.constant([3.0, 3.0])

# 使用初始化器 initializer op 的 run() 方法初始化 'x'

x.initializer.run()

# 增加一个减法 sub op, 从 'x' 减去 'a'. 运行减法 op, 输出结果

sub = tf.sub(x, a)

print sub.eval()

# ==> [-2. -1.]

## Tensor

TensorFlow 程序使用 tensor 数据结构来代表所有的数据, 计算图中, 操作间传递的数据都是 tensor. 你可以把 TensorFlow tensor 看作是一个 n 维的数组或列表. 一个 tensor 包含一个静态类型 rank, 和 一个 shape. 想了解 TensorFlow 是如何处理这些概念的, 参见 [Rank, Shape, 和 Type](http://www.tensorfly.cn/tfdoc/resources/dims_types.html).

## 变量

[Variables](http://www.tensorfly.cn/tfdoc/how_tos/variables.html) for more details. 变量维护图执行过程中的状态信息. 下面的例子演示了如何使用变量实现一个简单的计数器. 参见 [变量](http://www.tensorfly.cn/tfdoc/how_tos/variables.html) 章节了解更多细节.

# 创建一个变量, 初始化为标量 0.

state = tf.Variable(0, name="counter")

# 创建一个 op, 其作用是使 state 增加 1

one = tf.constant(1)

new\_value = tf.add(state, one)

update = tf.assign(state, new\_value)

# 启动图后, 变量必须先经过`初始化` (init) op 初始化,

# 首先必须增加一个`初始化` op 到图中.

init\_op = tf.initialize\_all\_variables()

# 启动图, 运行 op

with tf.Session() as sess:

# 运行 'init' op

sess.run(init\_op)

# 打印 'state' 的初始值

print sess.run(state)

# 运行 op, 更新 'state', 并打印 'state'

for \_ in range(3):

sess.run(update)

print sess.run(state)

# 输出:

# 0

# 1

# 2

# 3

代码中 assign() 操作是图所描绘的表达式的一部分, 正如 add() 操作一样. 所以在调用 run() 执行表达式之前, 它并不会真正执行赋值操作.

通常会将一个统计模型中的参数表示为一组变量. 例如, 你可以将一个神经网络的权重作为某个变量存储在一个 tensor 中. 在训练过程中, 通过重复运行训练图, 更新这个 tensor.

## Fetch

为了取回操作的输出内容, 可以在使用 Session 对象的 run() 调用 执行图时, 传入一些 tensor, 这些 tensor 会帮助你取回结果. 在之前的例子里, 我们只取回了单个节点 state, 但是你也可以取回多个 tensor:

input1 = tf.constant(3.0)

input2 = tf.constant(2.0)

input3 = tf.constant(5.0)

intermed = tf.add(input2, input3)

mul = tf.mul(input1, intermed)

with tf.Session():

result = sess.run([mul, intermed])

print result

# 输出:

# [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]

需要获取的多个 tensor 值，在 op 的一次运行中一起获得（而不是逐个去获取 tensor）。

## Feed

上述示例在计算图中引入了 tensor, 以常量或变量的形式存储. TensorFlow 还提供了 feed 机制, 该机制 可以临时替代图中的任意操作中的 tensor 可以对图中任何操作提交补丁, 直接插入一个 tensor.

feed 使用一个 tensor 值临时替换一个操作的输出结果. 你可以提供 feed 数据作为 run() 调用的参数. feed 只在调用它的方法内有效, 方法结束, feed 就会消失. 最常见的用例是将某些特殊的操作指定为 "feed" 操作, 标记的方法是使用 tf.placeholder() 为这些操作创建占位符.

input1 = tf.placeholder(tf.types.float32)

input2 = tf.placeholder(tf.types.float32)

output = tf.mul(input1, input2)

with tf.Session() as sess:

print sess.run([output], feed\_dict={input1:[7.], input2:[2.]})

# 输出:

# [array([ 14.], dtype=float32)]

for a larger-scale example of feeds. 如果没有正确提供 feed, placeholder() 操作将会产生错误. [MNIST 全连通 feed 教程](http://www.tensorfly.cn/tfdoc/tutorials/mnist_tf.html) ([source code](https://tensorflow.googlesource.com/tensorflow/+/master/tensorflow/g3doc/tutorials/mnist/fully_connected_feed.py)) 给出了一个更大规模的使用 feed 的例子.

三、经典实例MNIST机器学习入门

入门网站：http://www.tensorfly.cn/tfdoc/tutorials/overview.html

进阶网站：http://www.tensorfly.cn/tfdoc/get\_started/basic\_usage.html

四、基于Tensoflow的CNN实现

**见附件MNIST**