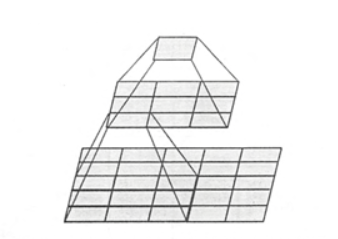
**VGGNet介绍**

**1 简要概括**

VGGNet由牛津大学计算机视觉组合和Google DeepMind公司研究员一起研发的深度卷积神经网络。它探索了卷积神经网络的深度和其性能之间的关系，通过反复的堆叠3\*3的小型卷积核和2\*2的最大池化层，成功的构建了16~19层深的卷积神经网络。VGGNet获得了ILSVRC 2014年比赛的亚军和定位项目的冠军，在top5上的错误率为7.5%。目前为止，VGGNet依然被用来提取图像的特征。

**2 创新点**

VGGNet全部使用3\*3的卷积核和2\*2的池化核，通过不断加深网络结构来提升性能。网络层数的增长并不会带来参数量上的爆炸，因为参数量主要集中在最后三个全连接层中。同时，两个3\*3卷积层的串联相当于1个5\*5的卷积层，3个3\*3的卷积层串联相当于1个7\*7的卷积层，即3个3\*3卷积层的感受野大小相当于1个7\*7的卷积层。但是3个3\*3的卷积层参数量只有7\*7的一半左右，同时前者可以有3个非线性操作，而后者只有1个非线性操作，这样使得前者对于特征的学习能力更强。

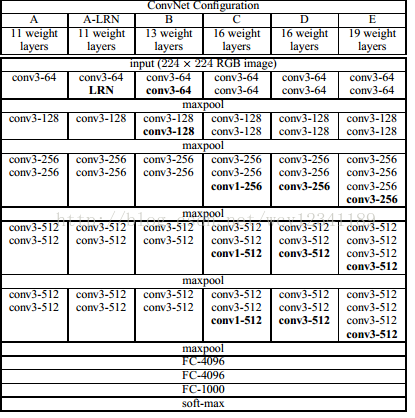


两个串联3x3的卷积层功能类似于一个5x5的卷积层

使用1\*1的卷积层来增加线性变换，输出的通道数量上并没有发生改变。这里提一下1\*1卷积层的其他用法，1\*1的卷积层常被用来提炼特征，即多通道的特征组合在一起，凝练成较大通道或者较小通道的输出，而每张图片的大小不变。有时1\*1的卷积神经网络还可以用来替代全连接层。 其他小技巧。VGGNet在训练的时候先训级别A的简单网络，再复用A网络的权重来初始化后面的几个复杂模型，这样收敛速度更快。VGGNet作者总结出LRN层作用不大，越深的网络效果越好，1\*1的卷积也是很有效的，但是没有3\*3的卷积效果好，因为3\*3的网络可以学习到更大的空间特征。3 网络结构 VGGNet的网络结构如下图所示。VGGNet包含很多级别的网络，深度从11层到19层不等，比较常用的是VGGNet-16和VGGNet-19。VGGNet把网络分成了5段，每段都把多个3\*3的卷积网络串联在一起，每段卷积后面接一个最大池化层，最后面是3个全连接层和一个softmax层。

**3 网络结构**

VGGNet的网络结构如下图所示。VGGNet包含很多级别的网络，深度从11层到19层不等，比较常用的是VGGNet-16和VGGNet-19。VGGNet把网络分成了5段，每段都把多个3\*3的卷积网络串联在一起，每段卷积后面接一个最大池化层，最后面是3个全连接层和一个softmax层。



**4 代码实现**

1. **读取数据使用多线程池化来缩短读取数据的时间**

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是 如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些 很耗资源的对象创建和销毁。如何利用已有对象来服务就是一个需要解决的关键问题，其实这就是一些"池化资源"技术产生的原因。

我理解为线程池是一个存放很多线程的单位，同时还有一个对应的任务队列。整个执行过程其实就是使用线程池中已有有限的线程把任务 队列中的任务做完。这样做的好处就是你不需要为每个任务都创建一个线程，因为当你创建第100个线程来执行第100个任务的时候，可能前面已经有50个线 程结束工作了。因此重复利用线程来执行任务，减少系统资源的开销。

一个不怎么恰当的比喻就是，有100台电脑主机箱需要从1楼搬到2楼，你不需要喊来100人帮忙搬，你只需要叫十个或者二十个人就足以，每个人分配十个或者五个甚至是谁搬的快谁就多搬。

1. **使用keras+tensorflow完成基于VGG模型的训练和预测**

**数据：**CIFAR-10数据集共有60000个32x32的彩色图像，有50000个训练图像和10000个测试图像。共有10个分类，它们是：飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车。每个分类有6000个图像。数据在工程目录的data文件目录下。

**数据处理**：首先利用多线程池化读取图片数据，其中有50000个训练数据和10000个测试数据。然后将数据正则化，为了使模型预测更精准。将lael进行OneHot编码当做输入的标签。

**搭建模型：**搭建VGG模型，其中包括卷积层，池化层，平坦层，全连接层，Dropout层，最后建立softmax输出层，输出层是10分类。采用的优化器方案是Adam。

**开始训练：**模型训练每一个epoch会打印出训练集的accuracy和loss,以及验证集的accuracy和loss，其中验证集的数据是取测试集的10%。

**画出训练集和验证集的准确率和误差：**

图片保存在工程中，可以查看训练过程中训练集和验证集的准确率和误差的变化，更好的理解训练过程。

**评估模型测试集的准确率：**利用训练好的模型，输入测试集的数据，打印输出测试集在模型上的准确率。

**进行预测：**利用测试集在训练好的模型上进行预测，画出预测集的前10项，其中展示的图片上方的文字代表实际结果和预测结果，相同代表预测正确，不同代表预测错误。

**并行计算：**利用tensorflow训练深度神经网络模型需要消耗很长时间，因为并行化计算就为提升运行速度提供了重要思路。Tensorflow提供了多种方法来使程序的并行运行，在使用这些方法时需要考虑的问题有：选取的计算设备是CPU还是GPU，每个CPU多少核的资源并行计算，构建图Graph时消耗资源如何分配等等问题。下面我们以Linux多核CPU的环境为例介绍几种常见方法来提升你的tensorflow程序的运行速度。

一、多核并行：CPU多核的运算和资源调用

在Tensorflow程序中，我们会经常看到”with tf.device("/cpu:0"): “ 这个语句。单独使用这个语句，而不做其他限制，实际上默认tensorflow程序占用所有可以使用的内存资源和CPU核，比如如果你的linux服务器是8核CPU，那么该程序会迅速占用可以使用的任意CPU，使用接近100%，最终结果就是影响整台服务器的其他程序。因此我们会想到需要限制使用的CPU核的个数和资源。在构建tf.Session() 变量时，可以通过传入tf.ConfigProto() 参数来改变一个tensorflow的session会话所使用的CPU核的个数以及线程数等等。多线程，设置Multi-threads在进行tf.ConfigProto()初始化时，我们也可以通过设置intra\_op\_parallelism\_threads参数和inter\_op\_parallelism\_threads参数，来控制每个操作符op并行计算的线程个数。二者的区别在于:intra\_op\_parallelism\_threads 控制运算符op内部的并行

当运算符op为单一运算符，并且内部可以实现并行时，如矩阵乘法，reduce\_sum之类的操作，可以通过设置intra\_op\_parallelism\_threads 参数来并行, intra代表内部。

inter\_op\_parallelism\_threads 控制多个运算符op之间的并行计算

当有多个运算符op，并且他们之间比较独立，运算符和运算符之间没有直接的路径Path相连。Tensorflow会尝试并行地计算他们，使用由inter\_op\_parallelism\_threads参数来控制数量的一个线程池。实例比较，下面我们比较线程数为2和4，平均每个batch的运行时间：

当参数为intra\_op\_parallelism\_threads = 2时, 每个step的平均运行时间从610ms降低到380ms。当参数为intra\_op\_parallelism\_threads = 4时, 每个step的平均运行时间从610ms降低到230ms。总结，在固定CPUcore的资源限制下，通过合理设置线程thread个数可以明显提升tensorflow程序运行速度。

**Keras并行计算：**

Keras在使用TensorFlow作为后端的时候可以进行分布式/多GPU的运算，Keras对多GPU和分布式的支持是通过TF完成的。注意，上例中由LSTM创建的变量不在GPU上：所有的TensorFlow变量总是在CPU上生存，而与它们在哪创建无关。各个设备上的变量转换TensorFlow会自动完成。

如果你想在不同的GPU上训练同一个模型的不同副本，但在不同的副本中共享权重，你应该首先在一个设备上实例化你的模型，然后在不同的设备上多次调用该对象。由于我的电脑只有CPU，所以训练过程并没有采用多GPU并行计算，而是在数据读取这里进行了多线程并行计算

**数据读取多线程并行计算：**

线程池在IBM文档库中这样的一段描写：“在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁。如何利用已有对象来服务就是一个需要解决的关键问题，其实这就是一些"池化资源"技术产生的原因。

诸如web服务器、数据库服务器、文件服务器和邮件服务器等许多服务器应用都面向处理来自某些远程来源的大量短小的任务。构建服务器应用程序的一个过于简单的模型是：每当一个请求到达就创建一个新的服务对象，然后在新的服务对象中为请求服务。但当有大量请求并发访问时，服务器不断的创建和销毁对象的开销很大。所以提高服务器效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这样就引入了“池”的概念，“池”的概念使得人们可以定制一定量的资源，然后对这些资源进行复用，而不是频繁的创建和销毁。

线程池是预先创建线程的一种技术。线程池在还没有任务到来之前，创建一定数量的线程，放入空闲队列中。这些线程都是处于睡眠状态，即均为启动，不消耗CPU，而只是占用较小的内存空间。当请求到来之后，缓冲池给这次请求分配一个空闲线程，把请求传入此线程中运行，进行处理。当预先创建的线程都处于运行状态，即预制线程不够，线程池可以自由创建一定数量的新线程，用于处理更多的请求。当系统比较闲的时候，也可以通过移除一部分一直处于停用状态的线程。

线程池的注意事项：

虽然线程池是构建多线程应用程序的强大机制，但使用它并不是没有风险的。在使用线程池时需注意线程池大小与性能的关系，注意并发风险、死锁、资源不足和线程泄漏等问题。

（1）线程池大小。多线程应用并非线程越多越好，需要根据系统运行的软硬件环境以及应用本身的特点决定线程池的大小。一般来说，如果代码结构合理的话，线程数目与CPU 数量相适合即可。如果线程运行时可能出现阻塞现象，可相应增加池的大小；如有必要可采用自适应算法来动态调整线程池的大小，以提高CPU 的有效利用率和系统的整体性能。

（2）并发错误。多线程应用要特别注意并发错误，要从逻辑上保证程序的正确性，注意避免死锁现象的发生。

（3）线程泄漏。这是线程池应用中一个严重的问题，当任务执行完毕而线程没能返回池中就简单线程池的设计

一个典型的线程池，应该包括如下几个部分：

1、线程池管理器（ThreadPool），用于启动、停用，管理线程池

2、工作线程（WorkThread），线程池中的线程

3、请求接口（WorkRequest），创建请求对象，以供工作线程调度任务的执行

4、请求队列（RequestQueue）,用于存放和提取请求

5、结果队列（ResultQueue）,用于存储请求执行后返回的结果

线程池管理器，通过添加请求的方法（putRequest）向请求队列（RequestQueue）添加请求，这些请求事先需要实现请求接口，即传递工作函数、参数、结果处理函数、以及异常处理函数。之后初始化一定数量的工作线程，这些线程通过轮询的方式不断查看请求队列（RequestQueue），只要有请求存在，则会提取出请求，进行执行。然后，线程池管理器调用方法（poll）查看结果队列（resultQueue）是否有值，如果有值，则取出，调用结果处理函数执行。通过以上讲述，不难发现，这个系统的核心资源在于请求队列和结果队列，工作线程通过轮询requestQueue获得人物，主线程通过查看结果队列，获得执行结果。因此，对这个队列的设计，要实现线程同步，以及一定阻塞和超时机制的设计，以防止因为不断轮询而导致的过多cpu开销。在本文中，将会用python语言实现，python的Queue，就是很好的实现了对线程同步机制。

