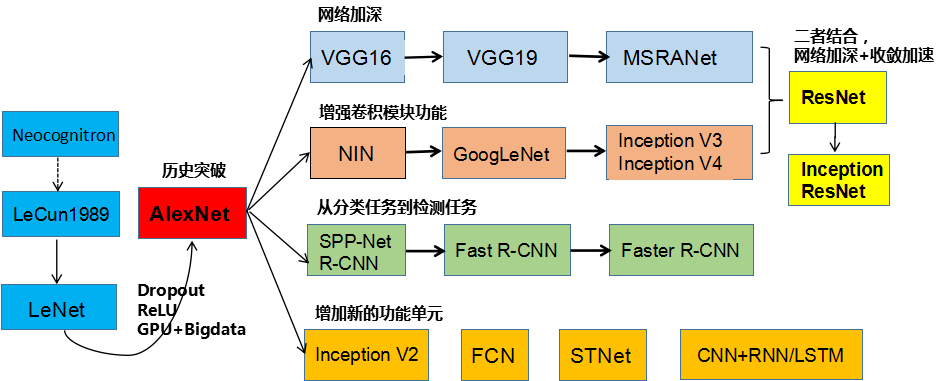
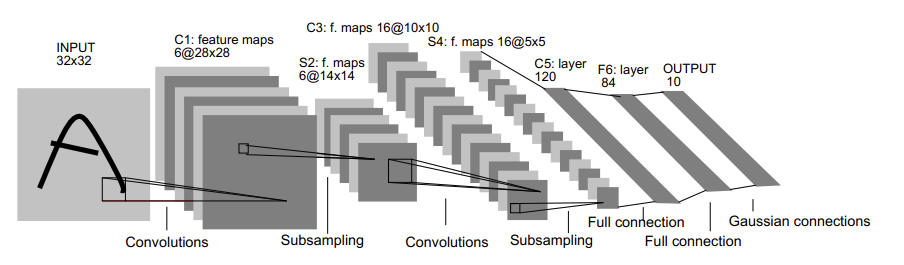
# 常用的深度学习网络架构



## 1.LeNet5

LeNet5诞生于1994年，是最早的卷积神经网络之一， 并且推动了深度学习领域的发展。自从1988年开始，在许多次成功的迭代后，这项由Yann LeCun完成的开拓性成果被命名为LeNet5。

下图为LeNet5网络架构图：



输入原始图像的大小是32×32，卷积层用 Cx 表示，子采样层（pooling）用Sx表示，全连接层用Fx表示，x 代表层数。

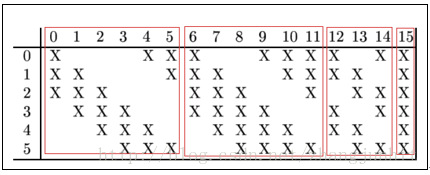
**(1)C1层是卷积层**，单通道下用了6个卷积核，用每个卷积核与原始的输入图像进行卷积，这样feature map的大小为(32-5+1)×(32-5+1)= 28×28。

我们再来看看需要多少个参数，卷积核的大小为5\*5，总共就有6\*（5\*5+1）=156个参数，其中+1是表示一个核有一个bias。对于卷积层C1，C1内的每个像素都与输入图像中的5\*5个像素和1个bias有连接，所以总共有156\*28\*28=122304个连接（connection）。有122304个连接，但是我们只需要学习156个参数，主要是通过权值共享实现的。

**(2)S2层为 pooling 层**，也可以说是池化或者特征映射的过程，拥有6个 feature map，每个feature map的大小为14\*14。S2这个pooling层是对C1中的2\*2区域内的像素求和乘以一个权值系数再加上一个偏置，然后将这个结果再做一次映射。于是每个池化核有两个训练参数，所以共有2x6=12个训练参数。但是有(2\*2+1)x14x14x6=5880个连接。

**(3)C3层也是一个卷积层**，16个卷积核，卷积模板的大小为5\*5，因此具有16个feature maps，每个feature map的大小为(14-5+1)×(14-5+1）= 10×10。

前6个feature map（对应上图第一个红框的6列）与S2层相连的3个feature map相连接（上图第一个红框），后面6个feature map与S2层相连的4个feature map相连接（上图第二个红框），，后面3个feature map与S2层部分不相连的4个feature map相连接，最后一个与S2层的所有feature map相连。卷积核大小依然为5\*5，所以总共有6\*（3\*5\*5+1）+6\*（4\*5\*5+1）+3\*（4\*5\*5+1）+1\*（6\*5\*5+1）=1516个参数。而图像大小为10\*10，所以共有151600个连接。



为什么采用上述这样的组合了？论文中说有两个原因：1）减少参数，2）这种不对称的组合连接的方式有利于提取多种组合特征。

**(4) S4是pooling层，**窗口大小仍然是2\*2，共计16个feature map，C3层的16个10x10的图分别进行以2x2为单位的池化得到16个5x5的特征图。这一层有2x16共32个训练参数，5x5x5x16=2000个连接。连接的方式与S2层类似。

**(5) C5层是一个卷积层**。由于S4层的16个图的大小为5x5，与卷积核的大小相同，所以卷积后形成的图的大小为1x1。这里形成120个卷积结果。每个都与上一层的16个图相连。所以共有(5x5x16+1)x120 = 48120个参数，同样有48120个连接。

**(6)F6层是全连接层。**F6层有84个节点，对应于一个7x12的比特图，-1表示白色，1表示黑色，这样每个符号的比特图的黑白色就对应于一个编码。该层的训练参数和连接数是(120 + 1)x84=10164。ASCII编码图如下：



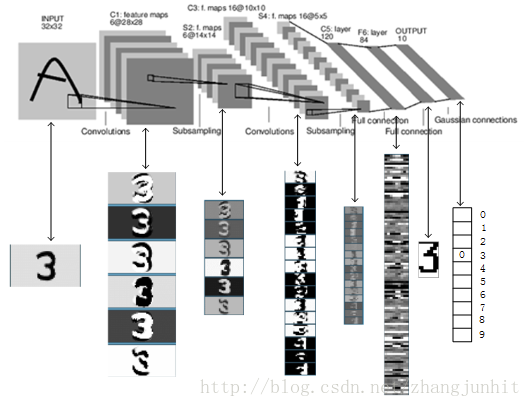
**為什麼F6神經元是84維 ？**

原文是用於手寫字識別，手寫字範圍為0-9共有10個字符，因此輸出OUTPUT層的節點共10個，可以把每個字符格式化為12\*7的圖像，如上圖，共有84個像元，像元的參數成分（即像素值）可以設為1和-1，此時C5到F6是通過sigmod函數計算的，因此F6節點的值範圍為-1到1，因此可以根據F6節點的值和格式化圖像（參數向量）的值距離，來判斷輸入圖像的類別，即通過徑向基函數。

**(7)Output层：**Output层也是全连接层，共有10个节点，分别代表数字0到9，且如果节点i的值为0，则网络识别的结果是数字i。采用的是径向基函数（RBF）的网络连接方式。假设x是上一层的输入，y是RBF的输出，则RBF输出的计算方式是：



上式w\_ij 的值由i的比特图编码确定，i从0到9，j取值从0到7\*12-1。RBF输出的值越接近于0，则越接近于i，即越接近于i的ASCII编码图，表示当前网络输入的识别结果是字符i。该层有84x10=840个参数和连接。



## 2.AlexNet

### 2.1 LeNet的局限性

早在1989年，Yann LeCun (現紐約大學教授) 和他的同事們就發表了卷積神經網絡（Convolution NeuralNetworks， 簡稱CNN）的工作。在很長時間裡，CNN雖然在小規模的問題上，如手寫數字，取得過當時世界最好結果，但一直沒有取得巨大成功。這主要原因是，CNN在大規模圖像上效果不好，比如像素很多的自然圖片內容理解，所以沒有得到計算機視覺領域的足夠重視。

這個驚人的結果為什麼在之前沒有發生？ 原因當然包括算法的提升，比如dropout等防止過擬合技術，但最重要的是，GPU帶來的計算能力提升和更多的訓練數據。

### 2.2 AlexNet简介

AlexNet 可以说是具有历史意义的一个网络结构，可以说在AlexNet之前，深度学习已经沉寂了很久。历史的转折在2012年到来，AlexNet 在当年的ImageNet图像分类竞赛中，top-5错误率比上一年的冠军下降了十个百分点，而且远远超过当年的第二名。

AlexNet 之所以能够成功，深度学习之所以能够重回历史舞台，原因在于：

非线性激活函数：ReLU

防止过拟合的方法：Dropout，Data augmentation

大数据训练：百万级ImageNet图像数据

其他：GPU实现，LRN归一化层的使用

下面简单介绍一下AlexNet的一些细节：

**数据增强：**

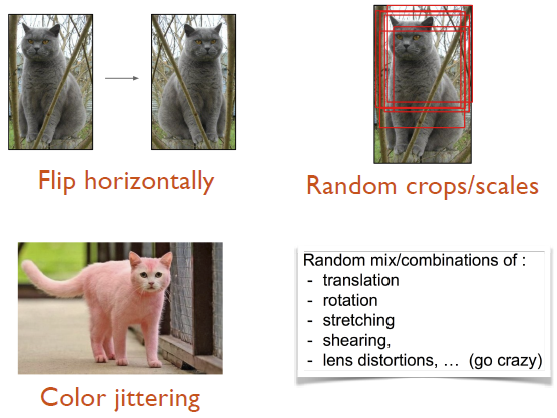
有一种观点认为神经网络是靠数据喂出来的，若增加训练数据，则能够提升算法的准确率，因为这样可以避免过拟合，而避免了过拟合你就可以增大你的网络结构了。当训练数据有限的时候，可以通过一些变换来从已有的训练数据集中生成一些新的数据，来扩大训练数据的size。

其中，最简单、通用的图像数据变形的方式:

从原始图像（256**,**256）中，随机的crop出一些图像（224**,**224）。【平移变换，crop】

水平翻转图像。【反射变换，flip】

给图像增加一些随机的光照。【光照、彩色变换，color jittering】



AlexNet 训练的时候，在data augmentation上处理的很好：

随机crop。训练时候，对于256＊256的图片进行随机crop到224＊224，然后允许水平翻转，那么相当与将样本倍增到((256-224)^2)\*2=2048。

测试时候，对左上、右上、左下、右下、中间做了5次crop，然后翻转，共10个crop，之后对结果求平均。作者说，不做随机crop，大网络基本都过拟合(under substantial overfitting)。

对RGB空间做PCA，然后对主成分做一个(0, 0.1)的高斯扰动。结果让错误率又下降了1%。

**ReLU 激活函数：**

Sigmoid是常用的非线性的激活函数，它能够把输入的连续实值“压缩”到0和1之间。特别的，如果是非常大的负数，那么输出就是0；如果是非常大的正数，输出就是1。但是它有一些致命的 缺点：

Sigmoids saturate and kill gradients. sigmoid有一个非常致命的缺点，当输入非常大或者非常小的时候，会有饱和现象，这些神经元的梯度是接近于0的。如果你的初始值很大的话，梯度在反向传播的时候因为需要乘上一个sigmoid 的导数，所以会使得梯度越来越小，这会导致网络变的很难学习。

Sigmoid 的 output 不是0均值. 这是不可取的，因为这会导致后一层的神经元将得到上一层输出的非0均值的信号作为输入。

产生的一个结果就是：如果数据进入神经元的时候是正的(e.g. x>0 elementwise in f=wTx+b)，那么 w 计算出的梯度也会始终都是正的。

当然了，如果你是按batch去训练，那么那个batch可能得到不同的信号，所以这个问题还是可以缓解一下的。因此，非0均值这个问题虽然会产生一些不好的影响，不过跟上面提到的 kill gradients 问题相比还是要好很多的。

Alex用ReLU代替了Sigmoid，发现使用 ReLU 得到的SGD的收敛速度会比 sigmoid/tanh快很多。

主要是因为它是linear，而且 non-saturating（因为ReLU的导数始终是1），相比于 sigmoid/tanh，ReLU 只需要一个阈值就可以得到激活值，而不用去算一大堆复杂的运算。

**Dropout：**

结合预先训练好的许多不同模型，来进行预测是一种非常成功的减少测试误差的方式（Ensemble）。但因为每个模型的训练都需要花了好几天时间，因此这种做法对于大型神经网络来说太过昂贵。

然而，AlexNet 提出了一个非常有效的模型组合版本，它在训练中只需要花费两倍于单模型的时间。这种技术叫做Dropout，它做的就是以0.5的概率，将每个隐层神经元的输出设置为零。以这种方式“dropped out”的神经元既不参与前向传播，也不参与反向传播。

所以每次输入一个样本，就相当于该神经网络就尝试了一个新的结构，但是所有这些结构之间共享权重。因为神经元不能依赖于其他特定神经元而存在，所以这种技术降低了神经元复杂的互适应关系。

正因如此，网络需要被迫学习更为鲁棒的特征，这些特征在结合其他神经元的一些不同随机子集时有用。在测试时，我们将所有神经元的输出都仅仅只乘以0.5，对于获取指数级dropout网络产生的预测分布的几何平均值，这是一个合理的近似方法。

**多gpu训练：**

单个GTX 580 GPU只有3GB内存，这限制了在其上训练的网络的最大规模。因此他们将网络分布在两个GPU上。

目前的GPU特别适合跨GPU并行化，因为它们能够直接从另一个GPU的内存中读出和写入，不需要通过主机内存。

他们采用的并行方案是：在每个GPU中放置一半核（或神经元），还有一个额外的技巧：GPU间的通讯只在某些层进行。

**LRN：**

一句话概括：本质上，这个层也是为了防止激活函数的饱和的。

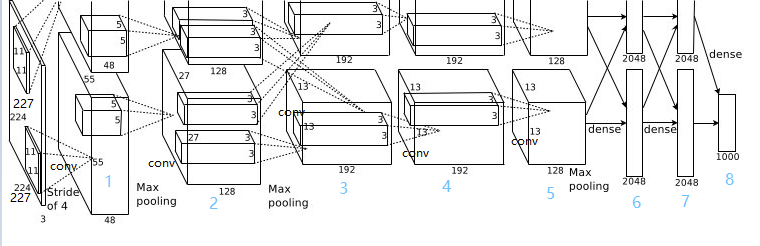
个人理解原理是通过正则化让激活函数的输入靠近“碗”的中间(避免饱和)，从而获得比较大的导数值。

所以从功能上说，跟ReLU是重复的。

不过作者说，从试验结果看，LRN操作可以提高网络的泛化能力，将错误率降低了大约1个百分点。

**AlexNet 优势在于：**网络增大（5个卷积层+3个全连接层+1个softmax层），同时解决过拟合（dropout，data augmentation，LRN），并且利用多GPU加速计算。

### 2.3 AlexNet的网络结构



该网络一共8层。

**第一层：**

第一层卷积层，输入层227×227×3，使用96个(11,11,3)的卷积核，可以生成96个卷积后的55×55个块，将他们叠起来，大小就是55×55×96（示意图中采样两个GPU，所以是两个55×55×48的块），然后应用Relu激励函数，来确保特征图的值范围在合理范围之内，比如{0,1}，{0,255}。再经过池化层，池化大小为(3,3)，滑动步长为2，所以得到27×27×96的特征，最后进行LRN处理。

**第二层：**

第二层卷积层使用256个，做pad\_size(2,2)的处理，以1个pixel为单位移动，能够产生27\*27个卷积后的矩阵框，做LRN处理，然后pooled，池化以3\*3矩形框，2个pixel为步长，得到256个13\*13个features。

**第三层：**

第三层卷积层，使用384个(3×3×256)kernels，步长为1，再经过padding，pad\_size=(1,1)，得到13×13×384（也就是两个(13×13×192)）的特征值，应用Relu激励函数，第三层没有池化层。

**第四层：**

第四层卷积层，使用384个大小为(3,3,384)的kernels进行卷积，步长为1，pad\_size为(1,1)，得到13×13×384个特征，然后应用Relu激励函数。

**第五层：**

第五层卷积层，使用256个大小为(3,3,384)kernels进行卷积， 步长为1，pad\_size为(1,1)，得到13×13×256个特征，接着应用Relu激励函数，再经过pool\_size为(3,3)，步长为2pixels的池化过程，得到6×6×256个特征。

**第六层：**

第六层全连接层，输入为上一层的6×6×256个特征，经过这一层的4096个神经元得到4096个特征，再应用Relu，接着进行Dropout。

**第七层：**

第七层全连接层，同样有4096个神经元，应用Relu，再进行Dropout。

**第八层：**

第八层全连接层，在最后的Softmax层得到1000维特征。

### 2.4 使用AlexNet实现mnist手写识别

# Import MINST data

from tensorflow**.**examples**.**tutorials**.**mnist import input\_data

mnist **=** input\_data**.**read\_data\_sets**(**"/tmp/data"**,** one\_hot**=**True**)**

import tensorflow as tf

# Parameters

learning\_rate **=** 0.001

training\_iters **=** 200000

batch\_size **=** 64

display\_step **=** 20

# Network Parameters

n\_input **=** 784 # MNIST data input **(**img shape**:** 28**\***28**)**

n\_classes **=** 10 # MNIST total classes **(**0**-**9 digits**)**

dropout **=** 1 # Dropout**,** probability to keep units

# tf Graph input

x **=** tf**.**placeholder**(**tf**.**float32**,** **[**None**,** n\_input**])**

y **=** tf**.**placeholder**(**tf**.**float32**,** **[**None**,** n\_classes**])**

keep\_prob **=** tf**.**placeholder**(**tf**.**float32**)** # dropout **(**keep probability**)**

# Create AlexNet model

def conv2d**(**name**,** l\_input**,** w**,** b**):**

**return** tf**.**nn**.**relu**(**tf**.**nn**.**bias\_add**(**tf**.**nn**.**conv2d**(**l\_input**,** w**,** strides**=[**1**,** 1**,** 1**,** 1**],** padding**=**'SAME'**),**b**),** name**=**name**)**

def max\_pool**(**name**,** l\_input**,** k**):**

**return** tf**.**nn**.**max\_pool**(**l\_input**,** ksize**=[**1**,** k**,** k**,** 1**],** strides**=[**1**,** k**,** k**,** 1**],** padding**=**'SAME'**,** name**=**name**)**

def norm**(**name**,** l\_input**,** lsize**=**4**):**

**return** tf**.**nn**.**lrn**(**l\_input**,** lsize**,** bias**=**1.0**,** alpha**=**0.001 **/** 9.0**,** beta**=**0.75**,** name**=**name**)**

def alex\_net**(**\_X**,** \_weights**,** \_biases**,** \_dropout**):**

# Reshape input picture

\_X **=** tf**.**reshape**(**\_X**,** shape**=[-**1**,** 28**,** 28**,** 1**])**

# Convolution Layer

conv1 **=** conv2d**(**'conv1'**,** \_X**,** \_weights**[**'wc1'**],** \_biases**[**'bc1'**])**

# Max Pooling (down-sampling)

pool1 **=** max\_pool**(**'pool1'**,** conv1**,** k**=**2**)**

# Apply Normalization

norm1 **=** norm**(**'norm1'**,** pool1**,** lsize**=**4**)**

# Apply Dropout

norm1 **=** tf**.**nn**.**dropout**(**norm1**,** \_dropout**)**

# Convolution Layer

conv2 **=** conv2d**(**'conv2'**,** norm1**,** \_weights**[**'wc2'**],** \_biases**[**'bc2'**])**

# Max Pooling (down-sampling)

pool2 **=** max\_pool**(**'pool2'**,** conv2**,** k**=**2**)**

# Apply Normalization

norm2 **=** norm**(**'norm2'**,** pool2**,** lsize**=**4**)**

# Apply Dropout

norm2 **=** tf**.**nn**.**dropout**(**norm2**,** \_dropout**)**

# Convolution Layer

conv3 **=** conv2d**(**'conv3'**,** norm2**,** \_weights**[**'wc3'**],** \_biases**[**'bc3'**])**

# Max Pooling (down-sampling)

pool3 **=** max\_pool**(**'pool3'**,** conv3**,** k**=**2**)**

# Apply Normalization

norm3 **=** norm**(**'norm3'**,** pool3**,** lsize**=**4**)**

# Apply Dropout

norm3 **=** tf**.**nn**.**dropout**(**norm3**,** \_dropout**)**

# Fully connected layer

dense1 **=** tf**.**reshape**(**norm3**,** **[-**1**,** \_weights**[**'wd1'**].**get\_shape**().**as\_list**()[**0**]])** # Reshape conv3 output to fit dense layer input

dense1 **=** tf**.**nn**.**relu**(**tf**.**matmul**(**dense1**,** \_weights**[**'wd1'**])** **+** \_biases**[**'bd1'**],** name**=**'fc1'**)** # Relu activation

dense2 **=** tf**.**nn**.**relu**(**tf**.**matmul**(**dense1**,** \_weights**[**'wd2'**])** **+** \_biases**[**'bd2'**],** name**=**'fc2'**)** # Relu activation

# Output, class prediction

out **=** tf**.**matmul**(**dense2**,** \_weights**[**'out'**])** **+** \_biases**[**'out'**]**

**return** out

# Store layers weight & bias

weights **=** **{**

'wc1'**:** tf**.**Variable**(**tf**.**random\_normal**([**3**,** 3**,** 1**,** 64**])),**

'wc2'**:** tf**.**Variable**(**tf**.**random\_normal**([**3**,** 3**,** 64**,** 128**])),**

'wc3'**:** tf**.**Variable**(**tf**.**random\_normal**([**3**,** 3**,** 128**,** 256**])),**

'wd1'**:** tf**.**Variable**(**tf**.**random\_normal**([**4**\***4**\***256**,** 1024**])),**

'wd2'**:** tf**.**Variable**(**tf**.**random\_normal**([**1024**,** 1024**])),**

'out'**:** tf**.**Variable**(**tf**.**random\_normal**([**1024**,** 10**]))**

**}**

biases **=** **{**

'bc1'**:** tf**.**Variable**(**tf**.**random\_normal**([**64**])),**

'bc2'**:** tf**.**Variable**(**tf**.**random\_normal**([**128**])),**

'bc3'**:** tf**.**Variable**(**tf**.**random\_normal**([**256**])),**

'bd1'**:** tf**.**Variable**(**tf**.**random\_normal**([**1024**])),**

'bd2'**:** tf**.**Variable**(**tf**.**random\_normal**([**1024**])),**

'out'**:** tf**.**Variable**(**tf**.**random\_normal**([**n\_classes**]))**

**}**

# Construct model

pred **=** alex\_net**(**x**,** weights**,** biases**,** keep\_prob**)**

# Define loss and optimizer

cost **=** tf**.**reduce\_mean**(**tf**.**nn**.**softmax\_cross\_entropy\_with\_logits**(**logits**=**pred**,** labels**=**y**))**

optimizer **=** tf**.**train**.**AdamOptimizer**(**learning\_rate**=**learning\_rate**).**minimize**(**cost**)**

# Evaluate model

correct\_pred **=** tf**.**equal**(**tf**.**argmax**(**pred**,**1**),** tf**.**argmax**(**y**,**1**))**

accuracy **=** tf**.**reduce\_mean**(**tf**.**cast**(**correct\_pred**,** tf**.**float32**))**

# Initializing the variables

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

# Launch the graph

with tf**.**Session**()** as sess**:**

sess**.**run**(**init**)**

step **=** 1

# Keep training until reach max iterations

**while** step **\*** batch\_size **<** training\_iters**:**

batch\_xs**,** batch\_ys **=** mnist**.**train**.**next\_batch**(**batch\_size**)**

# Fit training using batch data

sess**.**run**(**optimizer**,** feed\_dict**={**x**:** batch\_xs**,** y**:** batch\_ys**,** keep\_prob**:** dropout**})**

**if** step **%** display\_step **==** 0**:**

# Calculate batch accuracy

acc **=** sess**.**run**(**accuracy**,** feed\_dict**={**x**:** batch\_xs**,** y**:** batch\_ys**,** keep\_prob**:** 1.**})**

# Calculate batch loss

loss **=** sess**.**run**(**cost**,** feed\_dict**={**x**:** batch\_xs**,** y**:** batch\_ys**,** keep\_prob**:** 1.**})**

print **(**"Iter " **+** str**(**step**\***batch\_size**)** **+** ", Minibatch Loss= " **+** "{:.6f}"**.**format**(**loss**)** **+** ", Training Accuracy= " **+** "{:.5f}"**.**format**(**acc**))**

step **+=** 1

print **(**"Optimization Finished!"**)**

# Calculate accuracy for 256 mnist test images

print **(**"Testing Accuracy:"**,** sess**.**run**(**accuracy**,** feed\_dict**={**x**:** mnist**.**test**.**images**[:**256**],** y**:** mnist**.**test**.**labels**[:**256**],** keep\_prob**:** 1.**}))**

### 2.5 AlexNet benchmark

AlexNet benchmark用于模拟AlexNet前向和反向传播的计算速度，可以评估网络性能。

# Copyright 2015 The TensorFlow Authors. All Rights Reserved.

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import argparse

from datetime import datetime

import math

import sys

import time

from six**.**moves import xrange

import tensorflow as tf

FLAGS **=** None

#输出当前操作的名称以及当前矩阵的尺寸

def print\_activations**(**t**):**

print**(**t**.**op**.**name**,** ' '**,** t**.**get\_shape**().**as\_list**())**

#alexnet网络定义

def inference**(**images**):**

parameters **=** **[]**

# conv1

with tf**.**name\_scope**(**'conv1'**)** as scope**:**

kernel **=** tf**.**Variable**(**tf**.**truncated\_normal**([**11**,** 11**,** 3**,** 64**],** dtype**=**tf**.**float32**,**stddev**=**1e-1**),** name**=**'weights'**)**

conv **=** tf**.**nn**.**conv2d**(**images**,** kernel**,** **[**1**,** 4**,** 4**,** 1**],** padding**=**'SAME'**)**

biases **=** tf**.**Variable**(**tf**.**constant**(**0.0**,** shape**=[**64**],** dtype**=**tf**.**float32**),** trainable**=**True**,** name**=**'biases'**)**

bias **=** tf**.**nn**.**bias\_add**(**conv**,** biases**)**

conv1 **=** tf**.**nn**.**relu**(**bias**,** name**=**scope**)**

print\_activations**(**conv1**)**

parameters **+=** **[**kernel**,** biases**]**

# lrn1

with tf**.**name\_scope**(**'lrn1'**)** as scope**:**

lrn1 **=** tf**.**nn**.**local\_response\_normalization**(**conv1**,**

alpha**=**1e-4**,**

beta**=**0.75**,**

depth\_radius**=**2**,**

bias**=**2.0**)**

# pool1

pool1 **=** tf**.**nn**.**max\_pool**(**lrn1**,**

ksize**=[**1**,** 3**,** 3**,** 1**],**

strides**=[**1**,** 2**,** 2**,** 1**],**

padding**=**'VALID'**,**

name**=**'pool1'**)**

print\_activations**(**pool1**)**

# conv2

with tf**.**name\_scope**(**'conv2'**)** as scope**:**

kernel **=** tf**.**Variable**(**tf**.**truncated\_normal**([**5**,** 5**,** 64**,** 192**],** dtype**=**tf**.**float32**,** stddev**=**1e-1**),** name**=**'weights'**)**

conv **=** tf**.**nn**.**conv2d**(**pool1**,** kernel**,** **[**1**,** 1**,** 1**,** 1**],** padding**=**'SAME'**)**

biases **=** tf**.**Variable**(**tf**.**constant**(**0.0**,** shape**=[**192**],** dtype**=**tf**.**float32**),**trainable**=**True**,** name**=**'biases'**)**

bias **=** tf**.**nn**.**bias\_add**(**conv**,** biases**)**

conv2 **=** tf**.**nn**.**relu**(**bias**,** name**=**scope**)**

parameters **+=** **[**kernel**,** biases**]**

print\_activations**(**conv2**)**

# lrn2

with tf**.**name\_scope**(**'lrn2'**)** as scope**:**

lrn2 **=** tf**.**nn**.**local\_response\_normalization**(**conv2**,**

alpha**=**1e-4**,**

beta**=**0.75**,**

depth\_radius**=**2**,**

bias**=**2.0**)**

# pool2

pool2 **=** tf**.**nn**.**max\_pool**(**lrn2**,**

ksize**=[**1**,** 3**,** 3**,** 1**],**

strides**=[**1**,** 2**,** 2**,** 1**],**

padding**=**'VALID'**,**

name**=**'pool2'**)**

print\_activations**(**pool2**)**

# conv3

with tf**.**name\_scope**(**'conv3'**)** as scope**:**

kernel **=** tf**.**Variable**(**tf**.**truncated\_normal**([**3**,** 3**,** 192**,** 384**],** dtype**=**tf**.**float32**,** stddev**=**1e-1**),** name**=**'weights'**)**

conv **=** tf**.**nn**.**conv2d**(**pool2**,** kernel**,** **[**1**,** 1**,** 1**,** 1**],** padding**=**'SAME'**)**

biases **=** tf**.**Variable**(**tf**.**constant**(**0.0**,** shape**=[**384**],** dtype**=**tf**.**float32**),** trainable**=**True**,** name**=**'biases'**)**

bias **=** tf**.**nn**.**bias\_add**(**conv**,** biases**)**

conv3 **=** tf**.**nn**.**relu**(**bias**,** name**=**scope**)**

parameters **+=** **[**kernel**,** biases**]**

print\_activations**(**conv3**)**

# conv4

with tf**.**name\_scope**(**'conv4'**)** as scope**:**

kernel **=** tf**.**Variable**(**tf**.**truncated\_normal**([**3**,** 3**,** 384**,** 256**],** dtype**=**tf**.**float32**,** stddev**=**1e-1**),** name**=**'weights'**)**

conv **=** tf**.**nn**.**conv2d**(**conv3**,** kernel**,** **[**1**,** 1**,** 1**,** 1**],** padding**=**'SAME'**)**

biases **=** tf**.**Variable**(**tf**.**constant**(**0.0**,** shape**=[**256**],** dtype**=**tf**.**float32**),**trainable**=**True**,** name**=**'biases'**)**

bias **=** tf**.**nn**.**bias\_add**(**conv**,** biases**)**

conv4 **=** tf**.**nn**.**relu**(**bias**,** name**=**scope**)**

parameters **+=** **[**kernel**,** biases**]**

print\_activations**(**conv4**)**

# conv5

with tf**.**name\_scope**(**'conv5'**)** as scope**:**

kernel **=** tf**.**Variable**(**tf**.**truncated\_normal**([**3**,** 3**,** 256**,** 256**],**dtype**=**tf**.**float32**,**stddev**=**1e-1**),** name**=**'weights'**)**

conv **=** tf**.**nn**.**conv2d**(**conv4**,** kernel**,** **[**1**,** 1**,** 1**,** 1**],** padding**=**'SAME'**)**

biases **=** tf**.**Variable**(**tf**.**constant**(**0.0**,** shape**=[**256**],** dtype**=**tf**.**float32**),**trainable**=**True**,** name**=**'biases'**)**

bias **=** tf**.**nn**.**bias\_add**(**conv**,** biases**)**

conv5 **=** tf**.**nn**.**relu**(**bias**,** name**=**scope**)**

parameters **+=** **[**kernel**,** biases**]**

print\_activations**(**conv5**)**

# pool5

pool5 **=** tf**.**nn**.**max\_pool**(**conv5**,**

ksize**=[**1**,** 3**,** 3**,** 1**],**

strides**=[**1**,** 2**,** 2**,** 1**],**

padding**=**'VALID'**,**

name**=**'pool5'**)**

print\_activations**(**pool5**)**

**return** pool5**,** parameters

def time\_tensorflow\_run**(**session**,** target**,** info\_string**):**

num\_steps\_burn\_in **=** 10

total\_duration **=** 0.0

total\_duration\_squared **=** 0.0

#num\_batches默认为100 所以i为0~109

**for** i in xrange**(**FLAGS**.**num\_batches **+** num\_steps\_burn\_in**):**

#起始时间

start\_time **=** time**.**time**()**

#进行计算，输入为Batch size张图片

\_ **=** session**.**run**(**target**)**

#运行时间

duration **=** time**.**time**()** **-** start\_time

**if** i **>=** num\_steps\_burn\_in**:**

#每隔10次打印一次

**if** **not** i **%** 10**:**

print **(**'%s: step %d, duration = %.3f' **%** **(**datetime**.**now**(),** i **-** num\_steps\_burn\_in**,** duration**))**

total\_duration **+=** duration

total\_duration\_squared **+=** duration **\*** duration

#统计总时间

mn **=** total\_duration **/** FLAGS**.**num\_batches

vr **=** total\_duration\_squared **/** FLAGS**.**num\_batches **-** mn **\*** mn

sd **=** math**.**sqrt**(**vr**)**

print **(**'%s: %s across %d steps, %.3f +/- %.3f sec / batch' **%** **(**datetime**.**now**(),** info\_string**,** FLAGS**.**num\_batches**,** mn**,** sd**))**

def run\_benchmark**():**

"""Run the benchmark on AlexNet."""

with tf**.**Graph**().**as\_default**():**

#产生空图片

image\_size **=** 224

#大小为224\*224 rgb 有batch\_size张

images **=** tf**.**Variable**(**tf**.**random\_normal**([**FLAGS**.**batch\_size**,**

image\_size**,**

image\_size**,** 3**],**

dtype**=**tf**.**float32**,**

stddev**=**1e-1**))**

pool5**,** parameters **=** inference**(**images**)**

init **=** tf**.**global\_variables\_initializer**()**

config **=** tf**.**ConfigProto**()**

config**.**gpu\_options**.**allocator\_type **=** 'BFC' # 防止 out of memory

sess **=** tf**.**Session**(**config**=**config**)**

sess**.**run**(**init**)**

#模拟前向传播

time\_tensorflow\_run**(**sess**,** pool5**,** "Forward"**)**

#定义一个损失函数

objective **=** tf**.**nn**.**l2\_loss**(**pool5**)**

#梯度下降

grad **=** tf**.**gradients**(**objective**,** parameters**)**

#模拟前向和反向传播

time\_tensorflow\_run**(**sess**,** grad**,** "Forward-backward"**)**

def main**(**\_**):**

run\_benchmark**()**

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

parser **=** argparse**.**ArgumentParser**()**

parser**.**add\_argument**(**

'--batch\_size'**,**

type**=**int**,**

**default=**128**,**

help**=**'Batch size.'

**)**

parser**.**add\_argument**(**

'--num\_batches'**,**

type**=**int**,**

**default=**100**,**

help**=**'Number of batches to run.'

**)**

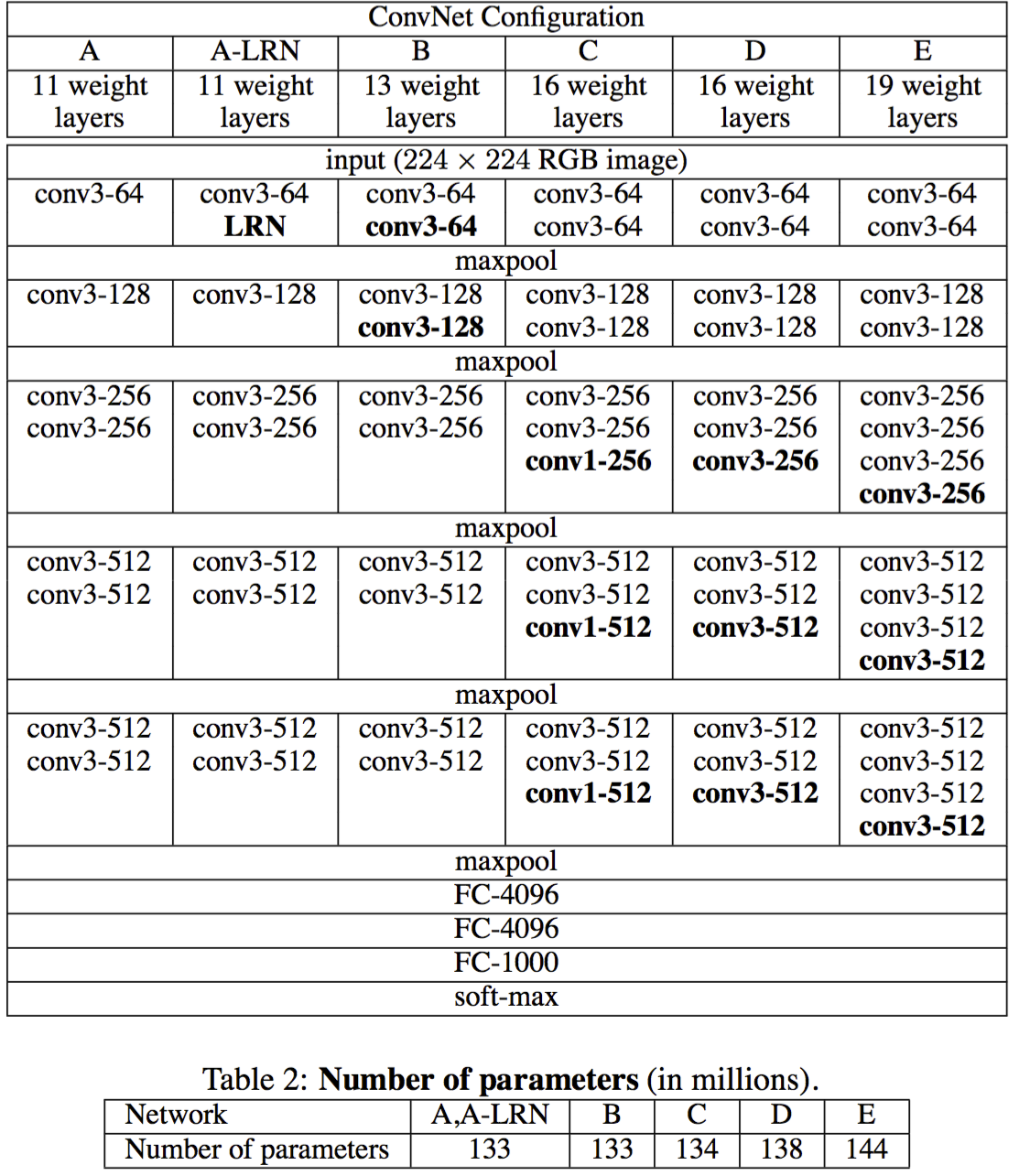
FLAGS**,** unparsed **=** parser**.**parse\_known\_args**()**

tf**.**app**.**run**(**main**=**main**,** argv**=[**sys**.**argv**[**0**]]** **+** unparsed**)**

## 3. VGG

### 3.1 VGG简介

VGGnet是Oxford的Visual Geometry Group的team，在ILSVRC 2014上的相关工作，主要工作是证明了增加网络的深度能够在一定程度上影响网络最终的性能，如下图，文章通过逐步增加网络深度来提高性能，虽然看起来有一点小暴力，没有特别多取巧的，但是确实有效，很多pretrained的方法就是使用VGG的model（主要是16和19），VGG相对其他的方法，参数空间很大，最终的model有500多m，alnext只有200m，googlenet更少，所以train一个vgg模型通常要花费更长的时间，所幸有公开的pretrained model让我们很方便的使用，paper中的几种模型如下：



可以从图中看出，从A到最后的E，他们增加的是每一个卷积组中的卷积层数，最后D，E是我们常见的VGG-16，VGG-19模型，C中作者说明，在引入1\*1是考虑做线性变换（这里channel一致， 不做降维），后面在最终数据的分析上来看C相对于B确实有一定程度的提升，但不如D、VGG主要得优势在于：

(1)尝试以更小的核以及提高核的数量来提高网络的性能，比如alex-net的核的大小为11×11×96不等，vgg网络一般都是用3×3的核，但是她核的数量提高了很多，有3×3×256不等，来提高性能。

(2)去掉了LRN，减少了内存的小消耗和计算时间。

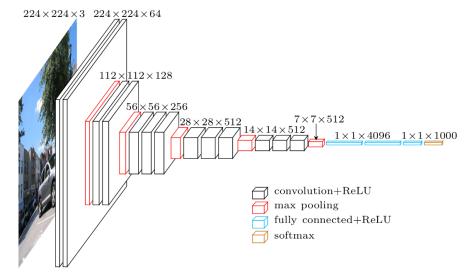
VGGNet论文作者给出的总结：

(1)LRN层作用不大。

(2)越深的网络效果越好。

(3)1\*1的卷积也是很有效的，但是没有3\*3的卷积好，大一些的卷积核可以学习更大的空间特征。

### 3.2 VGG-16 benchmark



图像识别 cifar10 imagenet

tflearn https://github.com/aymericdamien/TensorFlow-Examples

slim

tf例子：<http://blog.csdn.net/u014365862/article/details/53978811>

http://blog.csdn.net/u014365862/article/details/53978811

<https://github.com/wolfib/image-classification-CIFAR10-tf>

<https://www.leiphone.com/news/201701/Y4uyEktkkwb5YhJM.html>

<http://www.csdn.net/article/2015-12-16/2826496>

<https://github.com/tensorflow/models/tree/master/tutorials/image>

<http://shartoo.github.io/tensorflow-CIFAR10/>