卷积神经网络详解

一、实验介绍

1.1 实验内容

Prisma 是最近很火的一款APP，它能够将一张普通的图像转换成各种艺术风格的图像。本课程基于卷积神经网络，使用Caffe框架，探讨图片风格迁移背后的算法原理，手把手教你实现和Prisma一样的功能，让电脑学习梵高作画。

本节课我们将学习CNN(卷积神经网络)的相关知识，学习本门课程之前，你需要先去学习814 使用python实现深度神经网络，也许你已经学过820 使用卷积神经网络进行图片分类中的卷积神经网络，对CNN有了一定的了解，但我依然强烈建议你学习本节课，本节课所讲解的卷积神经网络对于图像风格迁移算法原理的理解至关重要。

1.2 实验知识点

- 为什么要引入CNN
- 卷积神经网络的层次结构

1.3 实验环境

- Xfce终端

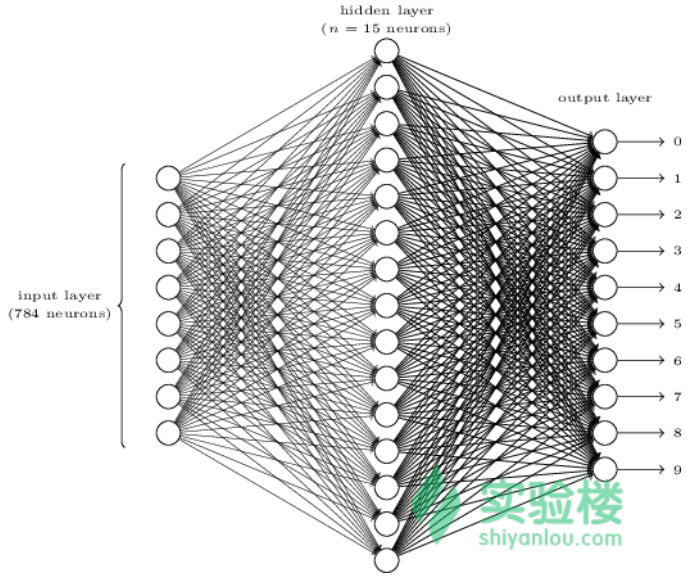
1.4 先修课程

- 814 使用python实现深度神经网络

二、卷积神经网络

2.1 CNN的引入

在人工的全连接神经网络中，每相邻两层之间的每个神经元之间都是有边相连的。当输入层的特征维度变得很高时，这时全连接网络需要训练的参数就会增大很多，计算速度就会变得很慢，例如一张黑白的 `28×28` 的手写数字图片，输入层的神经元就有784(28×28)个，如下图所示：



若在中间只使用一层隐藏层，参数 `w` 就有 `784×15=11760` 多个；若输入的是 `28×28` 带有颜色的RGB格式的手写数字图片，输入神经元就有28×28×3=2352（RGB有3个颜色通道）个。这很容易看出使用全连接神经网络处理图像中的需要训练参数过多的问题。

而在卷积神经网络（Convolutional Neural Network，CNN）中，卷积层的神经元只与前一层的部分神经元节点相连，即它的神经元间的连接是非全连接的，且同一层中某些神经元之间的连接的权重 `w` 和偏移 `b` 是共享的（即相同的），这样大量地减少了需要训练参数的数量。

卷积神经网络CNN的结构一般包含这几个层：

- 输入层：用于数据的输入
- 卷积层：使用卷积核进行特征提取和特征映射
- 激励层：由于卷积也是一种线性运算，因此需要增加非线性映射
- 池化层：进行下采样，对特征图稀疏处理，减少数据运算量。

- 全连接层：通常在CNN的尾部进行重新拟合，减少特征信息的损失
- 输出层：用于输出结果

当然中间还可以使用一些其他的功能层：

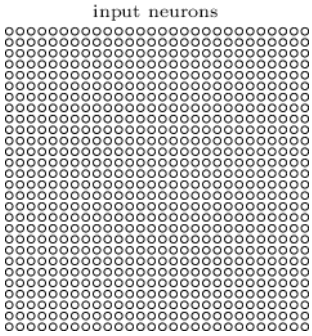
- 归一化层（Batch Normalization）：在CNN中对特征的归一化
- 切分层：对某些（图片）数据的进行分区域的单独学习
- 融合层：对独立进行特征学习的分支进行融合

2.2 CNN的层次结构

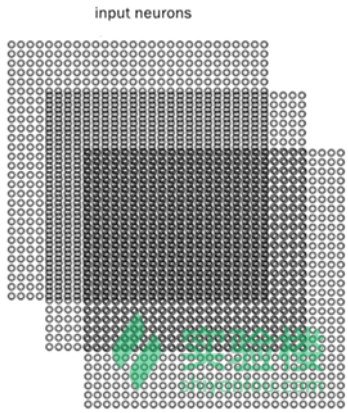
2.2.1 输入层 Input Layer

在CNN的输入层中，图片数据输入的格式与全连接神经网络的输入格式一维向量不太一样。**CNN的输入层的输入格式保留了图片本身的结构。**

对于黑白的 28×28 的图片，CNN的输入是一个 28×28 的的二维神经元，如下图所示：



而对于RGB格式的28×28图片，CNN的输入则是一个3×28×28的三维神经元（RGB中的每一个颜色通道都有一个28×28的矩阵），如下图所示：

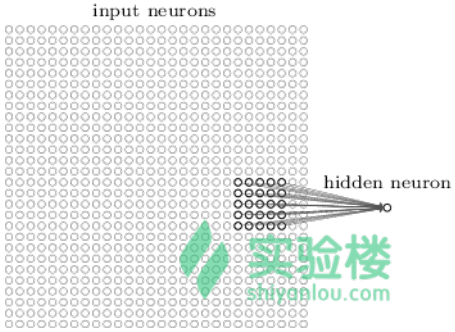


2.2.2 卷积层 Convolutional Layers

在卷积层中有几个重要的概念：

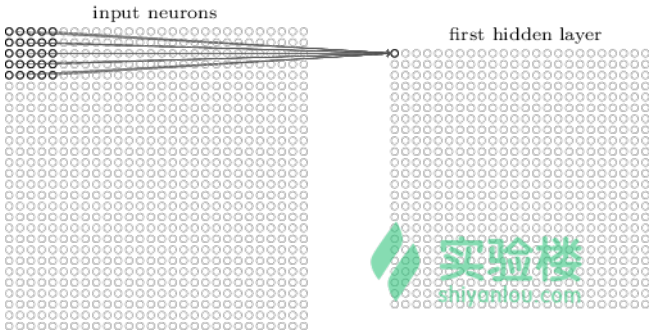
- 感知视野（local receptive fields）
- 共享权值（shared weights）

假设输入的是一个 28×28 的二维神经元，我们定义 5×5 的一个local receptive fields（感知视野），即隐藏层的神经元与输入层的 5×5 个神经元相连，这个 5×5 的区域就称之为 Local Receptive Fields，如下图所示：



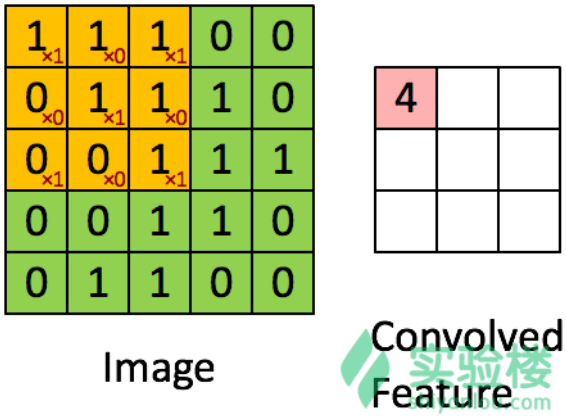
可类似看作：隐藏层中的神经元具有一个固定大小的感知视野去感受上一层的部分特征。在全连接神经网络中，隐藏层中的神经元的感知视野足够大乃至可以看到上一层的所有特征。

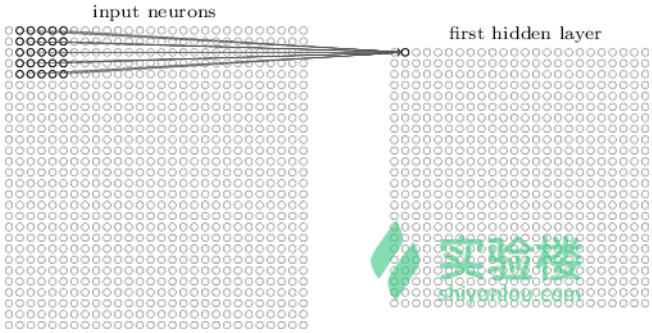
而在卷积神经网络中，隐藏层中的神经元的感受视野比较小，只能看到上一次的部分特征，上一层的其他特征可以通过平移感知视野来得到同一层的其他神经元，由同一层其他神经元来看：



设移动的步长为 1：从左到右扫描，每次移动 1 格，扫描完之后，再向下移动一格，再次从左到右扫描。

具体过程如动图所示：





可看出卷积层的神经元是只与前一层的部分神经元节点相连，每一条相连的线对应一个权重 w 。

一个感知视野带有一个 卷积核（或滤波器 filter），我们将感知视野中的权重 w 矩阵称为卷积核；将感受视野对输入的扫描间隔称为步长（stride，步长在长和宽上的移动保持一致）；当步长比较大时（stride>1），为了扫描到边缘的一些特征，感受视野可能会“出界”，这时需要对边界扩充(pad)，边界扩充可以设为0 或其他值。步长和边界扩充值的大小由用户来定义。

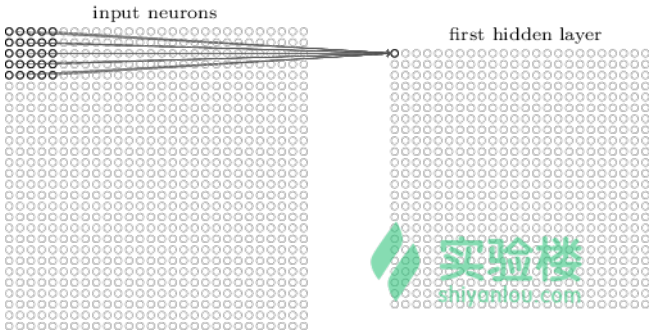
卷积核 的大小由用户来定义，即定义的感知视野的大小；卷积核的权重矩阵的值，便是卷积神经网络的参数，为了有一个偏移项，卷积核可附带一个偏移项 b，它们的初值可以随机来生成，可通过训练进行变化。 在神经网络中，我们经常加入偏移项来引入非线性因素，如果这里你不理解什么是非线性因素，后文中介绍激励层时将具体叙述。

动图演示的是 3*3 卷积核对图像卷积的过程，步长为 1，偏移项0。因此，当我们使用 5*5 卷积核操作时，感知视野扫描后可以计算出下一层神经元的值为：

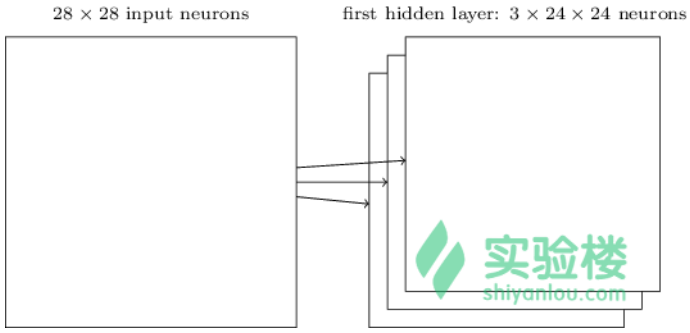
$$b+\sum_{i=0}^{4}\sum_{j=0}^{4}w_{ij}x_{ij}$$

对下一层的所有神经元来说，它们从不同的位置去探测了上一层神经元的特征。

我们将通过一个带有卷积核的感知视野扫描生成的下一层神经元矩阵称为一个 feature map（特征映射图），如下图的右边便是一个feature map：

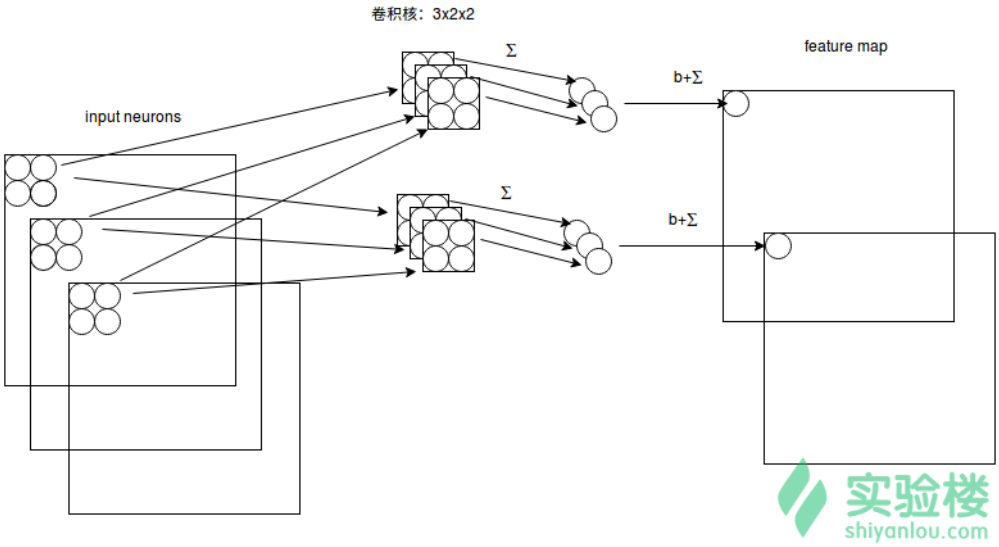


因此在同一个 `feature map` 上的神经元使用的卷积核是相同的，因此这些神经元共享权值 `shared weights`，共享卷积核中的权值和附带的偏移。一个 `feature map` 对应一个卷积核，若我们使用`3` 个不同的卷积核，可以输出3个 `feature map`：（感知视野： `5×5`，布长stride：`1`）



因此在CNN的卷积层，我们需要训练的参数大大地减少到了 `(5×5+1)×3=78` （一个卷积核中包括5*5=25个未定参数，随机偏移项1个参数）个。

假设输入的是 `28×28` 的RGB图片，即输入的是一个 `3×28×28` 的的二维神经元，这时卷积核的大小不只用长和宽来表示，还有深度，感知视野也对应的有了深度，如下图所示：



由图可知，感知视野： `3×2×2` ； 卷积核： `3×2×2` ，深度为 `3`；下一层的神经元的值为：

$$b+\sum_{d=0}^{2}\sum_{i=0}^{1}\sum_{j=0}^{1}w_{dij}x_{dij}$$

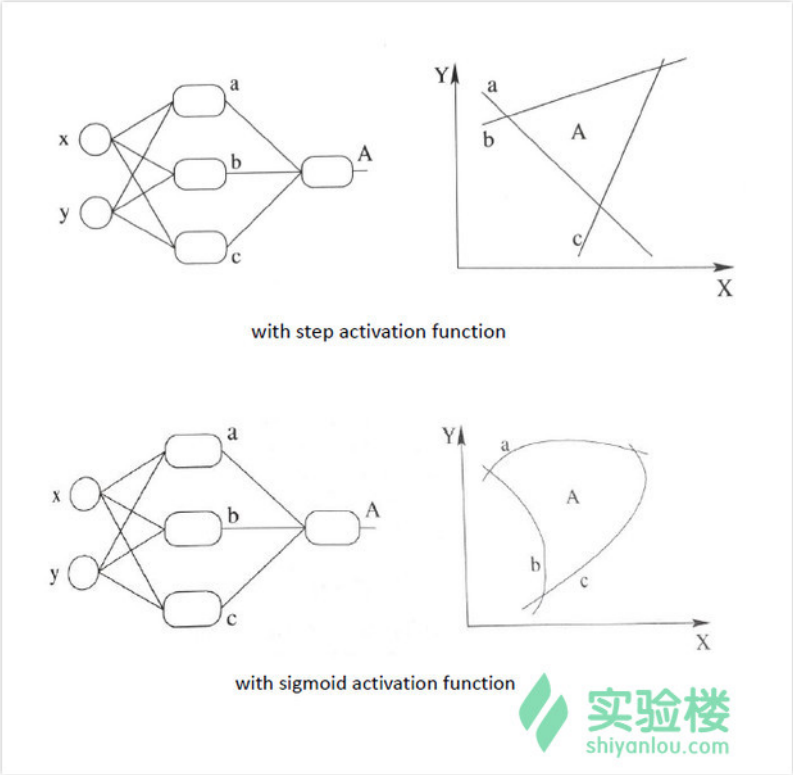
此处输入图片的描述

卷积核的深度和感知视野的深度相同，都由输入数据来决定，长宽可由自己来设定，数目也可以由自己来设定，一个卷积核依然对应一个 `feature map`。

2.2.3 激励层 `Activiation Layers`

关于神经网络激励函数的作用，常听到的解释是：不使用激励函数的话，神经网络的每层都只是做线性变换，多层输入叠加后也还是线性变换。因为线性模型的表达能力不够，激励函数可以引入非线性因素。 其实很多时候我们更想直观的了解激励函数的是如何引入非线性因素的。

我们使用神经网络来分割平面空间作为例子：



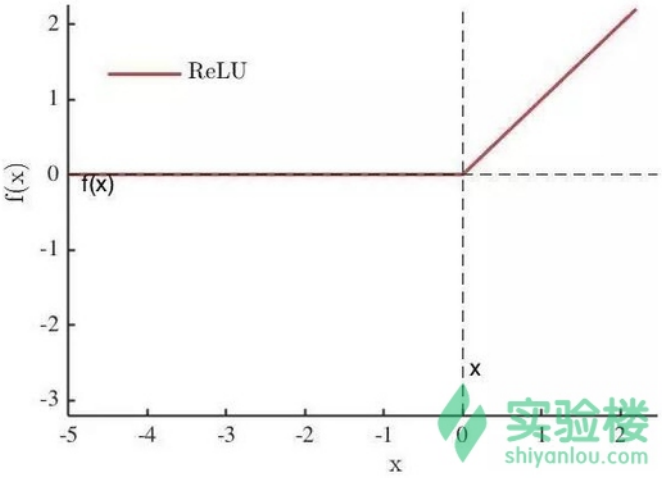
此处输入图片的描述

图的上半部分是没有激励函数的神经网络，其输出实际是线性方程，然后用复杂的线性组合来逼近曲线，得到最终的 `Positive Region A` ；

加入非线性激励函数后，神经网络就有可能学习到平滑的曲线来分割平面，而不是用复杂的线性组合逼近平滑曲线来分割平面，如图的下半部分所示，这就是为什么我们要有非线性的激活函数的原因。

常见的激活函数包括Sigmoid函数、Softmax函数、ReLu函数，深度神经网络使用的激励函数一般为 `ReLu(Rectified Linear Units)函数`（计算量小，效果显著）：

$$y = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$



卷积层和激励层通常合并在一起称为“卷积层”。

2.2.4 池化层 `Pooling Layers`

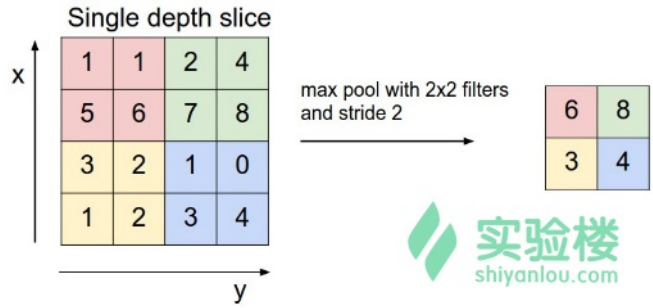
当输入经过卷积层时，若感受视野比较小，布长 `stride` 比较小，得到的 `feature map`（特征图）还是比较大，可以通过池化层来对每一个 `feature map` 进行降维操作，输出的深度还是不变的，依然为 `feature map` 的个数。

池化层也有一个“池化视野（ `pooling filter` ）”来对 `feature map` 矩阵进行扫描，对“池化视野”中的矩阵值进行计算，一般有两种计算方式：

- Max pooling：取“池化视野”矩阵中的最大值

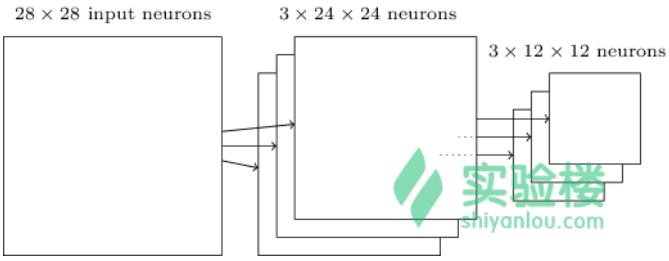
- Average pooling：取“池化视野”矩阵中的平均值

扫描的过程中同样地会涉及的扫描布长stride，扫描方式同卷积层一样，先从左到右扫描，结束则向下移动布长大小，再从左到右。如下图示例所示：



其中 池化卷积核： 2×2，布长stride：2。

最后可将 3 个 24×24 的 feature map 下采样得到 3 个 24×24的特征矩阵：



2.2.5 全连接层 Fully Connected Layers，FC

其实卷积层也可以看作是全连接的一种简化形式:不全连接+参数共享，但是卷积层还保留了空间位置信息，大大减少了参数并且使得训练变得可控。不同channel同一位置上的全连接等价与1x1的卷积核的卷积操作。N个节点的全连接可近似为N个模板卷积后的 均值池化(GAP)。

全连接本质是矩阵的乘法，相当于一个特征空间变换，可以把有用的信息提取整合。再加上激活函数的非线性映射，多层全连接层理论上可以模拟任何非线性变换；但缺点也很明显：无法保持空间结构，全连接的一个作用是维度变换，尤其是可以把高维变到低维，同时把有用的信息保留下来。全连接另一个作用是隐含语义的表达(embedding)，把原始特征映射到各个隐语义节点(hidden node)。对于最后一层全连接而言，就是分类的显示表达。

一般卷积神经网络模型中只有最后1~2层是全连接层，对于这一层一个不严谨的理解就是，经过卷积、池化等处理，我们可以得到了一系列高度抽象过的 feature map，将特征图输入全连接层，全连接的过程可以对这些特征进行投票，1，2，3 特征更像是猫的特征，2，4，5 更像是狗的特征，从而最终得到每个类别的概率。

但是需要注意的是，全连接层并不是必须的，由于全连接层输入参数过多，以及会丢失一些特征位置信息的缺点，现在FCN（Fully Convolutional Networks 全卷积网络）逐渐火了起来，但这不是本文的重点，不再详述。

三、实验总结

本节课我们学习了卷积神经网络的知识，概括为：

- 为什么要引入CNN
- 卷积神经网络的层次结构

下节课我们将学习经典卷积神经网络模型VGG，并用caffe提供的draw_net.py对网络模型进行可视化操作，之后我们还将讲解图像迁移算法原理。

VGG与风格迁移算法原理

一、实验介绍

1.1 实验内容

上节课我们学习了卷积神经网络的基本原理，本节实验我们将学习用于图像风格迁移的经典的卷积神经网络模型VGG，并用caffe提供的 draw_net.py 实现模型的可视化，本节实验我们也将学习图像风格转换的算法原理。

1.2 实验知识点

- 经典CNN模型 VGG16

- 图片风格算法原理

1.3 实验环境

- python 2.7
- caffe 实验楼环境内置
- Xfce终端

二、VGG

CNN的经典结构始于1998年的LeNet-5，成于2012年历史性的AlexNet，从此大盛于图像相关领域，主要包括：

- LeNet-5，1998年
- AlexNet，2012年
- ZF-net，2013年
- GoogleNet，2014年
- VGG，2014年
- ResNet，2015年
- Deep Residual Learning，2015年

vgg 和 googlenet 是2014年 ImageNet （图像分类大赛） 的双雄，这两类模型结构有一个共同特点是Go deeper。

VGG网络非常深，通常有 16/19层，称作 VGG16 和 VGG19 ，卷积核大小为 3 x 3 ，16和19层的区别主要在于后面三个卷积部分卷积层的数量。

本实验主要讲解VGG16。

2.1 可视化VGG16

启动终端，进入caffe下的python目录：

```
$ cd /opt/caffe/python
$ ls
```

确认是否有 draw_net.py 文件。

获得我们提供的VGG16的网格配置文件 vgg16_train_val.prototxt。

```
$ sudo wget http://labfile.oss.aliyuncs.com/courses/861/vgg16_train_val.prototxt
```

安装依赖包，这些都是基于python的画图工具包。

```
$ sudo apt-get install python-pydot
$ sudo apt-get install graphviz
```

安装完毕，准备可视化VGG16模型。

```
$ python draw_net.py /opt/caffe/python/vgg16_train_val.prototxt ~/Desktop/vgg16.png
```

程序运行完成之后，你可以通过以下命令查看vgg16.png，这个即为可视化后的VGG16。

```
$ cd ~/Desktop/
$ display vgg16.png
```

2.2 理解VGG16

下图是官方给出的VGG16的数据表格：

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

此处输入图片的描述

首先我们先学会看懂一些式子的表达：

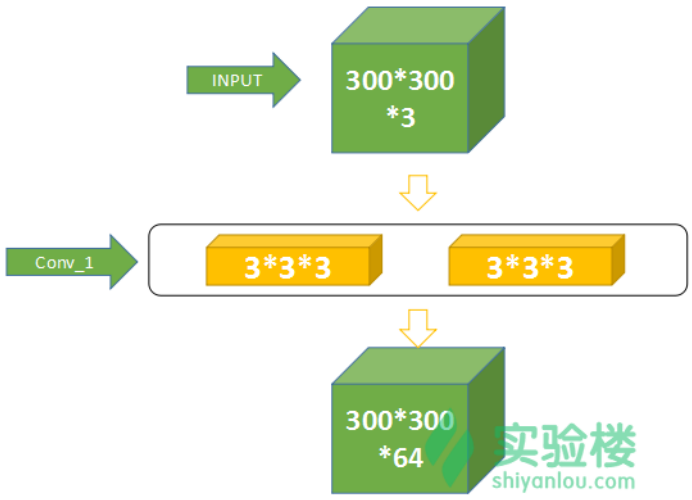
`Conv3-512` → 第三层卷积后维度变成512；

`Conv3_2 s=2` → 第三层卷积层里面的第二子层，滑动步长等于2（每次移动两个格子）

有了以上的知识可以开始剖析 `VGG16` 了。

由于篇幅的关系我们只推演从 `Input` 到 `Conv1`，再从 `Conv1` 过渡到 `Conv2` 的过程。

2.2.2 从 `Input` 到 `Conv1`



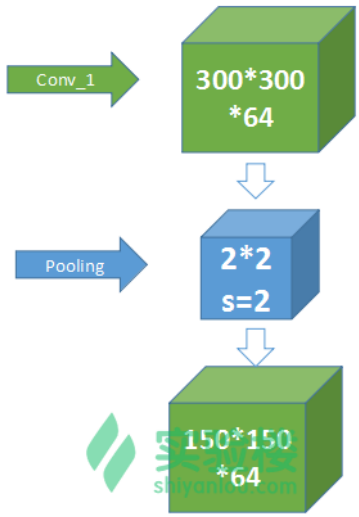
此处输入图片的描述

假如我们输入的是 `300x300`的RGB图像，那么输入层接受的输入就是 `300x300x3`个神经元。

图中的两个黄色表示卷积层，是VGG16网络结构十六层当中的第一层（Conv1_1）和第二层（Conv1_2），他们合称为Conv1。由上节的知识，我们可以知道卷积核大小应为 `3x3x3`。经过卷积层的运算我们可以得到一维的 `298x298x1` 的矩阵（因为卷积核也是三维所以结果是一维），为了在下一层中可以继续用 `3x3` 的卷积核做卷积，我们需要进行 `Padding` 操作，将图像扩充为 `300x300x1`，而 `VGG16` 在Conv1这层中使用了64个卷积核，那么生成了64张对应不同卷积核的 `feature map`，这样就得到了上图最终的结果

300x300x64。

2.2.2 从 Conv1 的 Conv2 过渡



此处输入图片的描述

这一步操作使用 Pooling ，使用的 池化卷积核为 2*2*64，步长为 2 ，那么滑动的矩阵本身没有重叠；刚好减半，第三维度 64不变。

其实之后的步骤都是类似的，根据我们刚刚画出的VGG16或者官方给出的表格，理解每一层有什么卷积核？多少个？池化的大小？步长多少？是否需要Padding？一步步思考，你就可以完全掌握VGG16的原理。

三、图像风格迁移 Image Style Transfer

以目前的深度学习技术，如果给定两张图像，完全有能力让计算机识别出图像具体内容。而图像的风格是一种很抽象的东西，人眼能够很有效地的辨别出不同画家不同流派绘画的风格，而在计算机的眼中，本质上就是一些像素，多层网络的实质其实就是找出更复杂、更内在的特性(features)，所以图像的风格理论上可以通过多层网络来提取图像里面可能含有的一些有意思的特征。

3.1 简介

此处输入图片的描述

将一幅图像的风格转移到另外一幅图像上被认为是一个图像纹理转移问题，传统上一般采用的是一些非参方法，通过一些专有的固定的方法（提取图像的亮度、低频颜色信息、高频纹理信息）来渲染。但是这些方法的问题在于只能提取底层特征而非高层抽象特征，而且往往一个程序只能做某一种风格或者某一个场景。随着CNN的日渐成熟，图像风格迁移技术也迎来了一次变革。需要注意的是，最近的很多应用型的研究成果都是将CNN渗透进各个领域，从而在普遍意义上完成一次技术的升级。

3.2 方法

实现图像的风格转换我们需要下面几个原料：

- 一个训练好的神经网络 VGG
- 一张风格图像，用来计算它的风格representation
- 一张内容图像，用来计算它的内容representation，
- 一张噪声图像，用来迭代优化
- loss函数，用来获得loss

给定一张风格图像 a 和一张普通图像 p，风格图像经过 VGG 的时候在每个卷积层会得到很多 feature maps，这些 feature maps 组成一个集合 A，同样的，普通图像 p 通过 VGG 的时候也会得到很多 feature maps，这些 feature maps 组成一个集合 F，然后生成一张随机噪声图像 x（在后面的实验中，其实就是普通图像 p），随机噪声图像 x 通过 VGG 的时候也会生成很多 feature maps，这些 feature maps 构成集合 G 和 F 分别对应集合 A 和 F，最终的优化函数是希望调整 x ，让随机噪声图像 x

最后看起来既保持普通图像 p 的内容，又有一定的风格图像 a 的风格。

可以进行风格转换的基础就是将内容和风格区分开来，接下来我们来看CNN如何做到这一点。

3.2.1 内容表示 Content Representation

给定一个图像 p，卷积神经网络每层使用滤波器(卷积核 filter)对图像进行编码。在 CNN 中，假设 layer l 有

N_l

此处输入图片的描述

个 filters，那么将会生成

N_l

此处输入图片的描述

个 feature maps，每个 feature map 的维度为

M_l

此处输入图片的描述

，

M_l

此处输入图片的描述

代表 feature map 的高与宽的乘积。所以每一层 feature maps 的集合可以表示为

$F^l \in R^{N_l \times M_l}$

此处输入图片的描述

，

F_{ij}^l

此处输入图片的描述

表示第 i 个 filter 在 position j 上的激活值。

所以，我们可以给出 Content 的 cost function，其中，p 是内容图像，x 是生成图像 ：

$$\mathcal{L}_{\text{content}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^l - P_{ij}^l\right)^2$$

此处输入图片的描述

对其求导后对应的激活函数为 ：

$$\frac{\partial \mathcal{L}_{\text{content}}}{\partial F_{ij}^l} = \begin{cases} \left(F^l - P^l\right)_{ij} & \text{if } F_{ij}^l > 0 \\ 0 & \text{if } F_{ij}^l < 0 \end{cases}$$

此处输入图片的描述

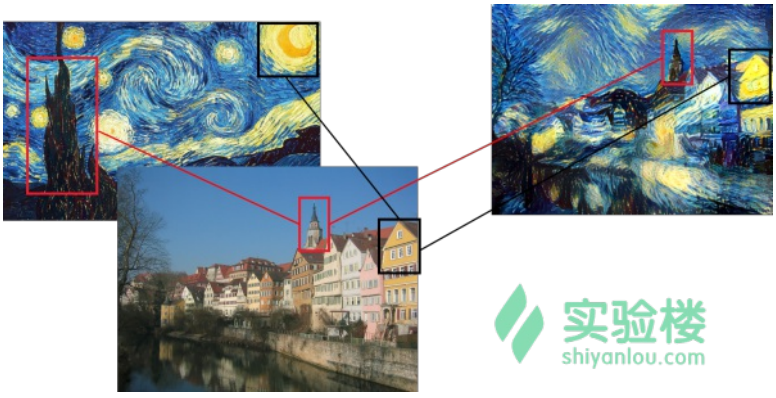
有了这个公式之后要怎么做呢？

使用现在公布的训练好的某些CNN网络，随机初始化一个输入图片大小的噪声图像x，然后保持CNN参数不变，将内容图片 p 和随机图像 x 输入进网络，然后对x求导，这样，x 就会在内容上越来越趋近于p。

3.2.2 风格表示 Style Representation

对于风格提取，我们需要用到 Gram matrix。

> Gram矩阵是计算每个通道i的feature map与每个通道j的feature map的内积。这个值可以看作代表i通道的feature map与j通道的feature map的互相关程度。而它又为什么能代表图片风格呢？



此处输入图片的描述

> 假设我们要对梵高的星空图做风格提取，在神经网络中某一层中有一个滤波器专门检测尖尖的塔顶这样的东西，另一个滤波器专门检测黑色。又有一个滤波器负责检测圆圆的东西，又有一个滤波器用来检测金黄色。
>
> 对梵高的原图做Gram矩阵，谁的相关性会比较大呢？如上图所示，“尖尖的”和“黑色”总是一起出现的，它们的相关性比较高。而“圆圆的”和“金黄色”都是一起出现的，他们的相关性比较高。
>
> 因此在风格转移的时候，其实也在内容图（待风格转换的图）里去寻找这种“匹配”，将尖尖的渲染为黑色（如塔尖），将圆

圆的渲染为金黄色（如近圆的房顶）。如果我们承认“图像的艺术风格就是其基本形状与色彩的组合方式”，这样一个假设，那么Gram矩阵能够表征艺术风格就是理所当然的事情了。

风格的抽取仍然是以层为单位的，对于风格图像 a，为了建立风格的 representation，我们先利用 Gram matrix 去表示每一层各个 feature maps 之间的关系，

$G^l \in R^{N_l \times N_l}$

此处输入图片的描述

，

G_{ij}^l

此处输入图片的描述

是 x 的 feature maps i,j 的内积，

A_{ij}^l

此处输入图片的描述

是 a 的feature maps i,j 的内积。

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$$

此处输入图片的描述

利用 Gram matrix，我们可以建立每一层的关于 style 的 cost function，

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - A_{ij}^l\right)^2$$

此处输入图片的描述

求偏导后对应的激励函数：

$$\frac{\partial E_l}{\partial F_{ij}^l} = \begin{cases} \frac{1}{N_l^2 M_l^2} \left((F^l)^{\mathrm{T}} \left(G^l - A^l\right)\right)_{ji} & \text{if } F_{ij}^l > 0 \\ 0 & \text{if } F_{ij}^l < 0 \end{cases}$$

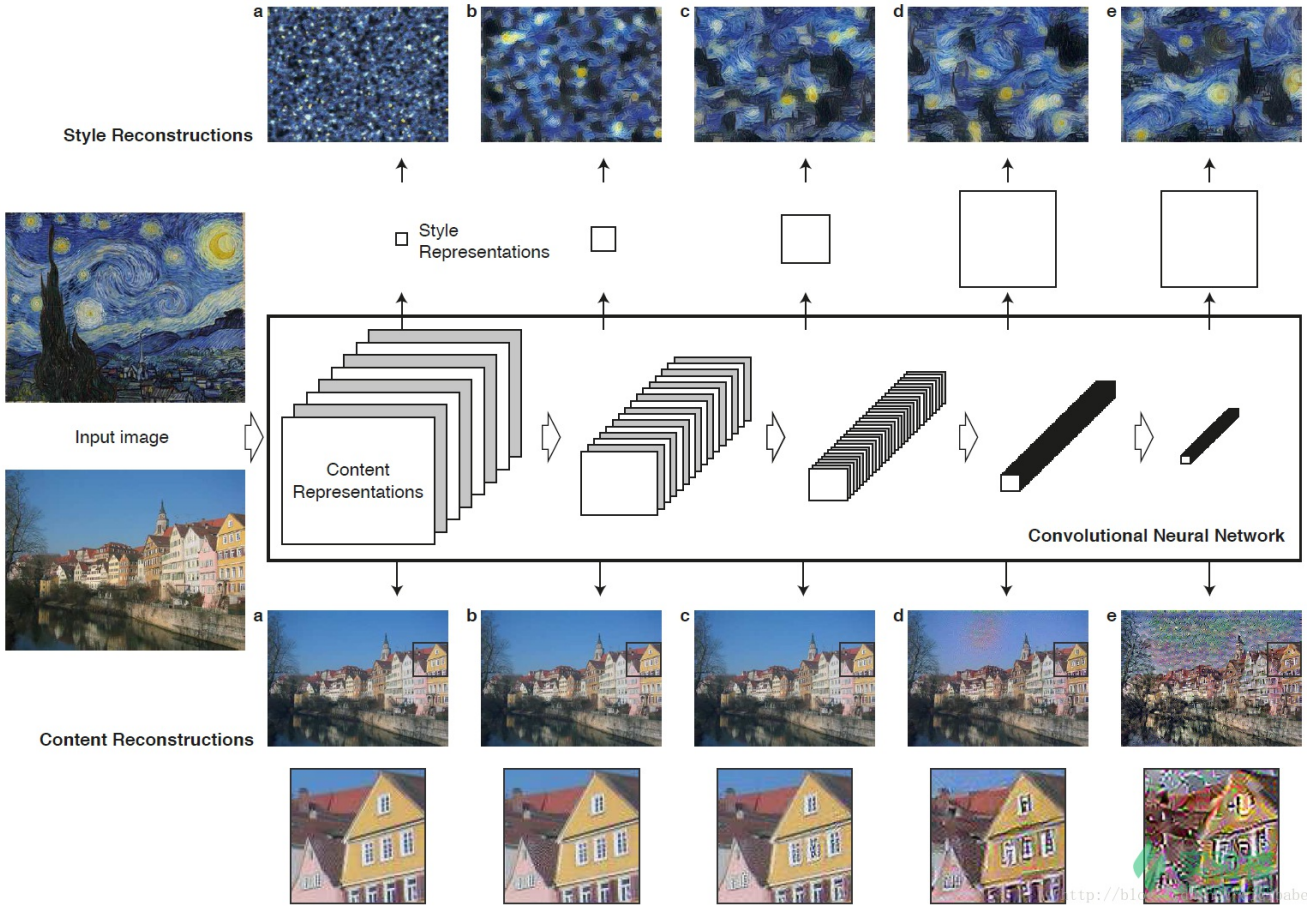
此处输入图片的描述

结合所有层，可以得到总的 cost ：

$$\mathcal{L}_{\text{style}}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

此处输入图片的描述

不考虑风格转换，只单独的考虑内容或者风格，可以看到如图所示：



此处输入图片的描述

上半部分是风格重建，由图可见，越用高层的特征，风格重建的就越粗粒度化。下半部分是内容重建，由图可见，越是底层的特征，重建的效果就越精细，越不容易变形。

最后将 content 和 style 的 cost 相结合，最终可以得到

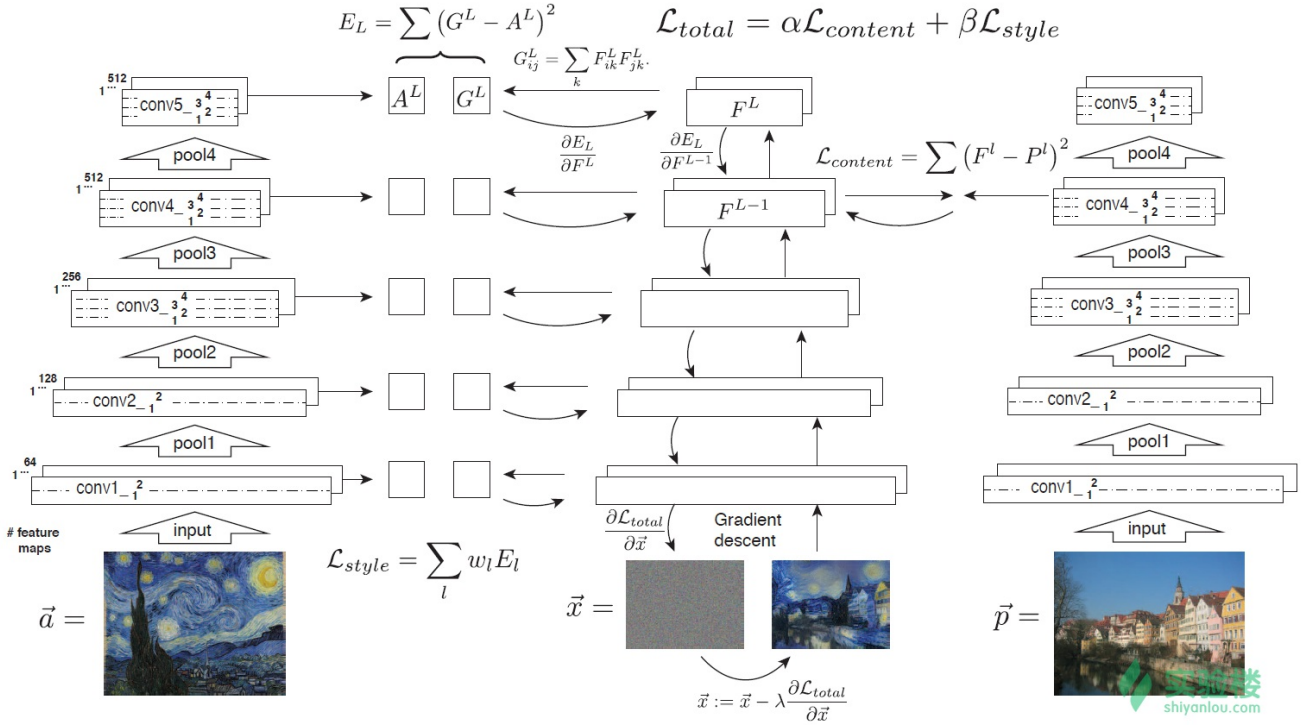
$$\mathcal{L}_{\text{total}}(\vec{p},\vec{a},\vec{x}) = \alpha\mathcal{L}_{\text{content}}(\vec{p},\vec{x}) + \beta\mathcal{L}_{\text{style}}(\vec{a},\vec{x})$$

此处输入图片的描述

3.2.3 风格转换

提取了内容和风格之后，我们就可以开始进行图片的风格迁移了。

具体过程如下图所示：



此处输入图片的描述

四、实验总结

本节实验，我们学习了

- 经典神经网络模型 VGG
- 图像风格迁移的算法原理基础

下节实验我们将亲自动手实现任意图片的风格转换。

五、课后习题

本节实验中，我们使用 draw_net.py 绘制出的结构图保存了参数信息，细节丰富，但是缺点是结构不是很清晰明了，这一点会在大型模型上的体现尤为明显，其实我们还可以用很多工具来实现网络模型的可视化操作，请使用在线工具QuickStart绘制googlenet的模型，并对照着可视化后的网络模型理解googlenet的操作原理。

实现图片的风格转换

一、实验介绍

1.1 实验内容

上一节课我们介绍了经典的CNN模型 VGG ，以及图像风格迁移算法的基本原理。本节课我们将使用另外一个经典的模型GoogLenet来实现我们的项目（这是由于环境的限制，用 googlenet可以更快的完成我们的风格转换），如果你完成了上节课的作业，那么你应该基本了解了 GoogleNet 的原理。在本节实验的算法展示中，我希望你可以将代码与上节课所讲解公式对照着看来帮助自己更好地理解，现在让我们开始本节实验，完成后你就可以亲自动手实现任意图片的风格转换。

1.2 实验知识点

- style-transfer
- 风格迁移算法代码详解
- 实现图片的任意风格转换

1.3 实验环境

- python 2.7、pip、numpy
- caffe
- Xfce终端

1.4 实验思路

我们可以先整理一下思路，从前面几节课的学习我们可以知道训练一个模型，我们需要准备：

- 一个训练好的神经网络官方参考的是Vgg16、Vgg19、Caffenet、Googlenet四类
- 一张风格图像style image，用来计算它的风格representation
- 一张内容图像content image，用来计算它的内容representation，
- 一张噪声图像result image，用来迭代优化一般都是拿Content内容图来做，Caffe里面默认也是拿内容图来作为底图
- 一个loss函数，用来获得loss
- 一个求反传梯度的计算图，用来依据loss获得梯度修改图片 Caffe有明确的forward和backward，会帮我们自动计算

二、实验步骤

2.1 style-transfer

2.1.1 简介

style-tansfer是 "A Neural Algorithm of Artistic Style"[2]的基于pyCaffe的实现。

> caffe的官方完美的支持Python语言的兼容，提供的接口就是pyCaffe

style-tansfer提供了将一个输入图像的艺术风格转移到另一个输入图像的方法，即图像风格迁移。

在这个算法中，神经网络操作由Caffe处理，使用numpy和scipy执行损耗最小化和其他杂项矩阵运算， L-BFGS用于最小化操作（关于L-BFGS）。

2.1.2 获取源码

我们可以从Github上 An implementation of "A Neural Algorithm of Artistic Style"[3]获取style-transfer源码

启动终端

```
$ cd /home/shiyanlou
$ wget http://labfile.oss.aliyuncs.com/courses/861/style-transfer-master.zip
$ mv style-transfer-master style-transfer
```

2.2 核心代码

在本实验中，所有的操作都在`style.py`中完成。

首先让我们来看一下`/home/shiyanlou/style-transfer/style.py`中的核心算法：

2.2.1 矩阵计算

```
def _compute_reprs(net_in, net, layers_style, layers_content, gram_scale=1):
    """
        首先正向传播计算出各层feature map，再将特征矩阵保存返回
        """
    (repr_s, repr_c) = ({}, {})
    net.blobs["data"].data[0] = net_in
    net.forward()

    for layer in set(layers_style)|set(layers_content):
        F = net.blobs[layer].data[0].copy()
        F.shape = (F.shape[0], -1)
        repr_c[layer] = F
        if layer in layers_style:
            repr_s[layer] = sgemm(gram_scale, F, F.T)

    return repr_s, repr_c
```

其中的网络权重的定义：

```
GOOGLENET_WEIGHTS = {"content": {"conv2/3x3": 2e-4,
                                 "inception_3a/output": 1-2e-4},
                     "style": {"conv1/7x7_s2": 0.2,
                               "conv2/3x3": 0.2,
                               "inception_3a/output": 0.2,
                               "inception_4a/output": 0.2,
                               "inception_5a/output": 0.2}}
```

内容用了2层，风格用了5层。我们可以用key()来获取层名：

```
layers_style = weights["style"].keys()
layers_content = weights["content"].keys()
```

2.2.2 梯度处理

```
def _compute_style_grad(F, G, G_style, layer):
    """
        完成风格梯度以及loss的计算
        """
    (Fl, Gl) = (F[layer], G[layer])
    c = Fl.shape[0]**-2 * Fl.shape[1]**-2
    El = Gl - G_style[layer]
    loss = c/4 * (El**2).sum()
    grad = c * sgemm(1.0, El, Fl) * (Fl>0)

    return loss, grad

    """
        完成内容梯度以及loss的计算
        """
def _compute_content_grad(F, F_content, layer):
    Fl = F[layer]
    El = Fl - F_content[layer]
    loss = (El**2).sum() / 2
    grad = El * (Fl>0)

    return loss, grad
```

2.2.3 噪声图像拟合

```
def _make_noise_input(self, init):
    """
        制造最开始的噪声输入，但是我们默认使用 `content` 作噪声图像。我们就是在这个上面进行不断的拟合
        """

    # 指定维度并在傅立叶域中创建网格
    dims = tuple(self.net.blobs["data"].data.shape[2:]) + \
           (self.net.blobs["data"].data.shape[1], )
    grid = np.mgrid[0:dims[0], 0:dims[1]]

    # 创建噪声的频率表示
    Sf = (grid[0] - (dims[0]-1)/2.0) ** 2 + \
```

```
            (grid[1] - (dims[1]-1)/2.0) ** 2
        Sf[np.where(Sf == 0)] = 1
        Sf = np.sqrt(Sf)
        Sf = np.dstack((Sf**int(init),)*dims[2])

        # 应用并使用 ifft 规范化
        ifft_kernel = np.cos(2*np.pi*np.random.randn(*dims)) + \
                      1j*np.sin(2*np.pi*np.random.randn(*dims))
        img_noise = np.abs(ifftn(Sf * ifft_kernel))
        img_noise -= img_noise.min()
        img_noise /= img_noise.max()

        # 预处理
        x0 = self.transformer.preprocess("data", img_noise)

        return x0
```

2.2.4 风格转换

```
def transfer_style(self, img_style, img_content, length=512, ratio=1e5,
                   n_iter=512, init="-1", verbose=False, callback=None):


        # 假设卷积层的输入是平方的
        orig_dim = min(self.net.blobs["data"].shape[2:])

        # 缩放图像
        scale = max(length / float(max(img_style.shape[:2])),
                    orig_dim / float(min(img_style.shape[:2])))
        img_style = rescale(img_style, STYLE_SCALE*scale)
        scale = max(length / float(max(img_content.shape[:2])),
                    orig_dim / float(min(img_content.shape[:2])))
        img_content = rescale(img_content, scale)

        # 计算表示风格的特征矩阵
        self._rescale_net(img_style)
        layers = self.weights["style"].keys()
        net_in = self.transformer.preprocess("data", img_style)
        gram_scale = float(img_content.size)/img_style.size
        G_style = _compute_reprs(net_in, self.net, layers, [],
                                 gram_scale=1)[0]

        # 计算表示内容的特征矩阵
        self._rescale_net(img_content)
        layers = self.weights["content"].keys()
        net_in = self.transformer.preprocess("data", img_content)
        F_content = _compute_reprs(net_in, self.net, [], layers)[1]

        # 噪声图像输入
        if isinstance(init, np.ndarray):
            img0 = self.transformer.preprocess("data", init)
        elif init == "content":
            img0 = self.transformer.preprocess("data", img_content)
        elif init == "mixed":
            img0 = 0.95*self.transformer.preprocess("data", img_content) + \
                   0.05*self.transformer.preprocess("data", img_style)
        else:
            img0 = self._make_noise_input(init)

        # 计算数据边界
        data_min = -self.transformer.mean["data"][:,0,0]
        data_max = data_min + self.transformer.raw_scale["data"]
        data_bounds = [(data_min[0], data_max[0])]*(img0.size/3) + \
                      [(data_min[1], data_max[1])]*(img0.size/3) + \
                      [(data_min[2], data_max[2])]*(img0.size/3)

        # 参数优化
        # 使用L-BFGS-B算法可以最小化 loss function 而且空间效率较高。
        grad_method = "L-BFGS-B"
        reprs = (G_style, F_content)
```

```
        minfn_args = {
            "args": (self.net, self.weights, self.layers, reprs, ratio),
            "method": grad_method, "jac": True, "bounds": data_bounds,
            "options": {"maxcor": 8, "maxiter": n_iter, "disp": verbose}
        }

        # 迭代优化
        self._callback = callback
        minfn_args["callback"] = self.callback
        if self.use_pbar and not verbose:
            self._create_pbar(n_iter)
            self.pbar.start()
            res = minimize(style_optfn, img0.flatten(), **minfn_args).nit
            self.pbar.finish()
        else:
            res = minimize(style_optfn, img0.flatten(), **minfn_args).nit

        return res
```

2.3 实现

2.3.1 pycaffe环境布置

因为github上的代码是基于pycaffe的，所以需要配置环境

```
$ cd /home/shiyanlou/style-transfer
$ sudo gedit style.py
```

在 `import caffe` 之前加入python和pycaffe的环境变量

```
sys.path.append("/opt/caffe/python")
sys.path.append("/opt/caffe/python/caffe")
```

当然，待会转换图片时我们需要一个进度条的显示，这里我们使用`python progressbar` 。

```
$ sudo pip install progressbar
```

2.3.2 下载模型

本实验用到的是googlenet，如果你完成了上节课的作业，那么你一定很清楚 googlenet 的结构了，其实和 VGG 一样，我们只需帮他们当作一个辅助工具，这是大牛们用巨大的数据集（ImageNet）帮我们训练好的可以准确提取图片特征的神经网络模型。下载完成之后需要将bvlc_googlenet.caffemodel放到指定路径下 `/style-transfer/models/googlenet`。

```
$ cd models/googlenet
$ wget http://labfile.oss.aliyuncs.com/courses/861/bvlc_googlenet.caffemodel
```

2.3.3 准备训练

一个可以使用的训练好的模型文件夹有三样东西 `style-transfer/models/googlenet`

- deploy.prototxt
- ilsvrc_2012_mean.npy
- bvlc_googlenet.caffemodel

1. `deploy.prototxt` 网格配置文件

你可以通过如下指令来观察网格的结构

```
#和第二节一样的操作
$ cd /opt/caffe/python
$ sudo apt-get install python-pydot
$ sudo apt-get insall graphviz
$ python draw_net.py /home/shiyanlou/style-transfer/models/googlenet/deploy.prototxt ~/Desktop/googlenet.png
$ cd ~/Desktop
$ display googlenet.png
```

1. `ilsvrc_2012_mean.npy` 均值文件（caffe中使用的均值数据格式是binaryproto,如果我们要使用python接口，或者我们要进行特征可视化，可能就要用到python格式的均值文件了）。

图片减去均值后，再进行训练和测试，会提高速度和精度。因此，一般在各种模型中都会有这个操作。那么这个均值怎么来的呢，实际上就是计算所有训练样本的平均值，计算出来后，保存为一个均值文件，在以后的测试中，就可以直接使用这个均值来相减，而不需要对测试图片重新计算。

2. `bvlc_googlenet.caffemodel` 训练好的神经网络模型。

2.3.4 参数解析

我们来看看 `/home/shiyanlou/style-transfer/style.py` 中关于参数的设定，具体代码在88行。

```
$ python style.py -S <style_image> -c <content_image> -m <model_name> -g 0
```

主要参数解析

- `-s`，风格图位置
- `-c`，内容图位置
- `-m`，模型位置
- `-g`，什么模式 `-1为CPU，0为单个GPU，1为两个GPU`。

其他默认或不必须参数

```
parser.add_argument("-r", "--ratio", default="1e4", type=str, required=False, help="style-to-content ratio")
```

非必要，转化比率α/β，有默认值

```
parser.add_argument("-n", "--num-iters", default=512, type=int, required=False, help="L-BFGS iterations")
```

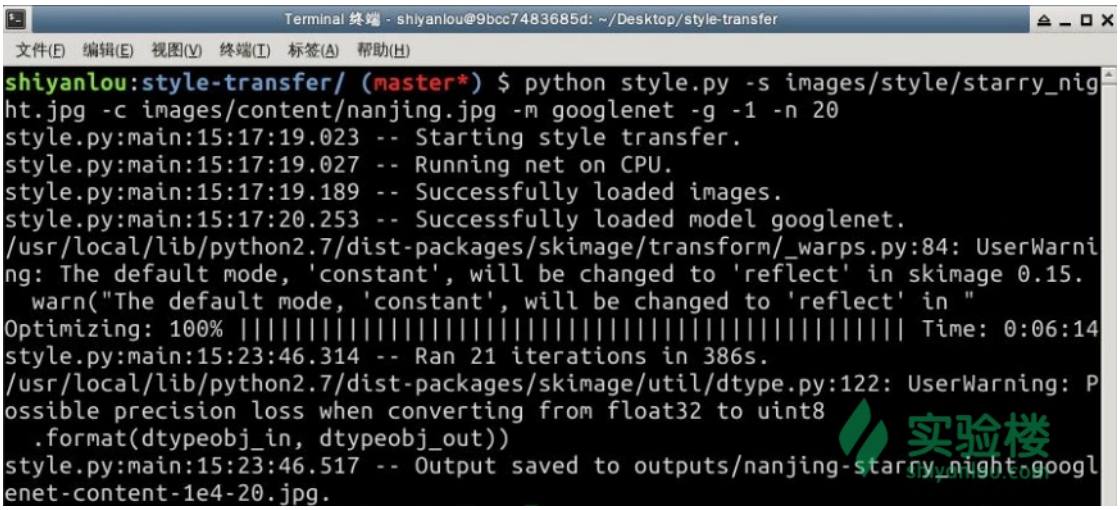
非必要，有默认值，表示迭代次数

2.3.5 开始转换

```
$ cd /home/shiyanlou/style-transfer
```

```
$ python style.py -s images/style/starry_night.jpg -c images/content/nanjing.jpg -m googlenet -g -1 -n 20
```

我们使用的是梵高“星空”的风格，需要转换风格的图片是nanjing.jpg，使用googlenet模型及CPU，训练20次的结果，如果操作正确的话，训练的过程应该如下图所示。



训练

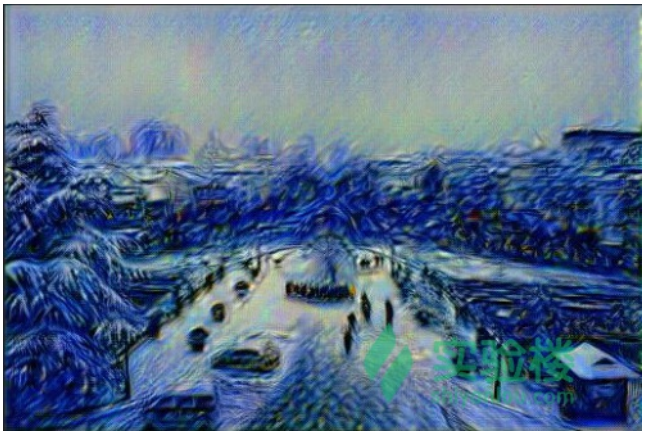
这里的警告其实我们不用在意，这只是提醒我们转换后的图片与原始图片的相对坐标系发生了变换。

2.3.6 生成风格转换图片

我们可以在 `/home/shiyanlou/style-transfer/images/style` 中查看风格图；

在 `/home/shiyanlou/style-transfer/images/content`中查看内容图。

在 `/home/shiyanlou/style-transfer/outputs/` 就可以看到我们训练完成得到的效果图，图片的名字包含我们使用的模型、转化比率、迭代次数。



效果图

我们可以通过更改 `style.py` 来设置图像输出的尺寸大小，例如你自己的照片图像大小是1024*500 ，更改输出length=1024,可以获得与原始图像一致的尺寸。不更改的话，程序中默认输出是512宽度，和输入原始图像一致的宽长比。

```
parser.add_argument("-l", "--length", default=1024, type=float, required=False, help="maximum image length")
```

```
def transfer_style(self, img_style, img_content, length=1024, ratio=1e4,
n_iter=512, init="-1", verbose=False, callback=None):
```

六、实验总结

至此，本门课程的学习结束，我们完成了一个很有趣的深度学习的项目。通过利用大牛训练的神经网络模型，还有我们自定义的loss函数，我们成功的让电脑学会了大师的绘画技巧，实现了以后是不是觉得其实并没有自己想象的这么难？

最近风格迁移的概念被炒的很火，Prisma也成了朋友圈的装逼利器，但是当我们真正理解了算法背后的原理，其实这也就是只纸老虎。最后，拿着自己画出来的图片去朋友圈装逼吧，记得屏蔽那些学过深度学习的人哦！