

Lecture 9

红色标注的语句，为重点。

蓝色下划线标注的语句，说明给出了参考阅读链接，可依兴趣阅读。

紫色加粗，表示参看附件。

写在Lecture9之前

本次分享的主题是Python与深度学习，主要分为两部分：1.以卷积神经网络CNN为例，大致介绍神经网络的原理。并推荐进一步了解的材料。2.以CNN进行验证码识别为例，介绍Python生态下深度学习的常用工具PyTorch使用方法。

需要注意的是，深度学习和机器学习一样，正在被PyTorch、TensorFlow等工具变得大众化、普及化。虽然背后的原理还是较为复杂的，但其本身的使用门槛已经急剧降低，如果大家仅仅是使用它，而非精通它，需要投入的精力其实不多，甚至可以现学现用。具体该投入多少时间去学习，还是要依各位的方向来平衡。我也会列出一些适合现学现用的资料，帮助大家今后需要时快速入门。

1. 卷积神经网络简介

卷积神经网络（Convolutional Neural Networks, CNN）是深度学习的代表性算法，其理论也是最成熟的。CNN被广泛应用于计算机视觉、自然语言处理等领域，助推了图像识别领域的高速发展，今天大家使用的人脸识别、自动驾驶等技术，早期均脱胎于CNN。

卷积神经网络中的“卷积”一词，是其思想的灵魂所在。卷积，可以简单地理解为（注意，这不是实际上的定义）对信息的浓缩和抽象，将多个输入处理成一个输出。一个卷积神经网络会经历多次卷积，这样，复杂的原始信息被一层层地抽象成一层层更高级的概念。例如，在图像处理过程中，第一层卷积可能只会识别出直线和拐弯，第二层卷积会在前面的基础上识别出物体的边界，第三层卷积会判断边界是生物还是物体，第四层会判断是猫还是狗……可以说，卷积赋予了计算机由浅到深，认知数据的能力。

卷积神经网络中的“神经网络”一词，是其实现的物理基础。神经网络在形态上是对于生物神经元网络的模拟，即通过一个个神经元之间的连接（神经），对数据（生物外界刺激）进行一层层的加权处理，最终得到一个输出（生物电信号），但实际上，神经网络和生物并没有太大的关系，它的本质是建立在坚实的数学基础上的：通过神经网络这种形式，我们可以实现对于任意函数的逼近，从而在理论上拟合任意数据的概率分布。

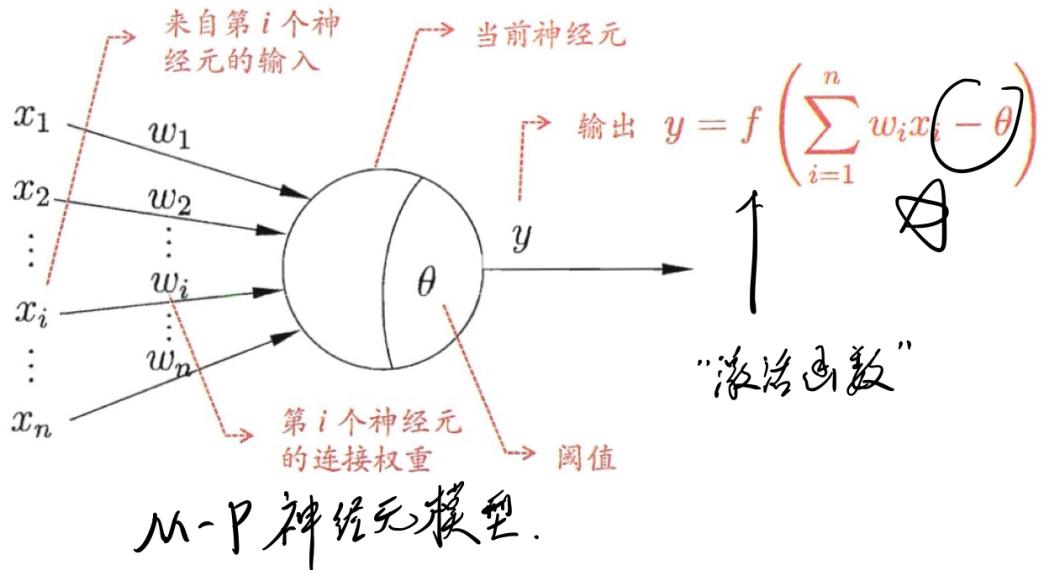
（以上言论均为方便大家理解，切勿先入为主）

1.1 前馈神经网络

卷积神经网络是前馈神经网络的一种，因此我们先介绍前馈神经网络，这是最简单的神经网络模型。

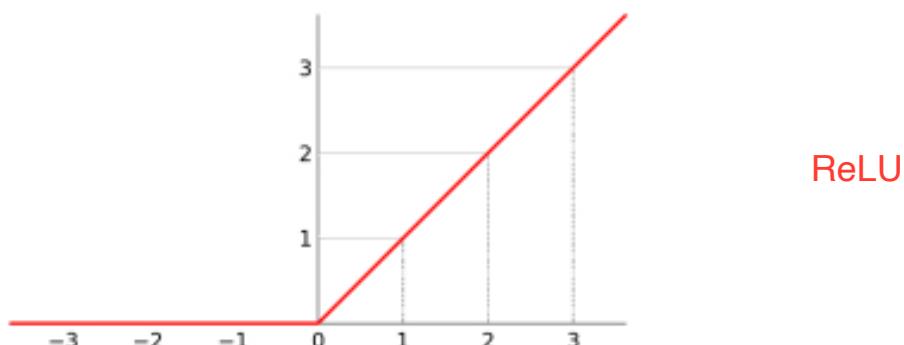
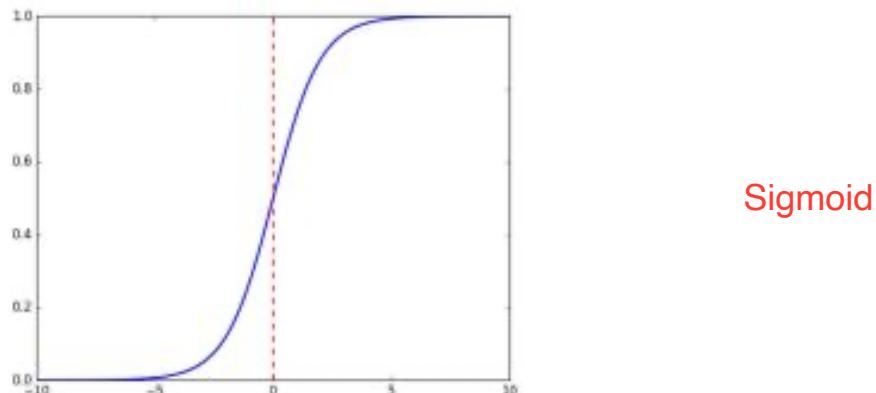
前馈神经网络的最基本单位是神经元。

Lecture 9



如图，一个神经元的基本组成部分为输入、权值、阈值、激活函数。在一个神经网络中，神经元接受一定个数的输入，将这些输入分别乘一个对应的权值进行加总，如果加总结果小于阈值，该神经元不会激活，如果大于阈值，则将加总结果减去阈值，作为激活函数（一般情况下，约定俗成有若干种激活函数可挑选）的输入，激活函数的输出值即为该神经元的输出值。

常见的激活函数有Sigmoid函数、ReLU函数等。



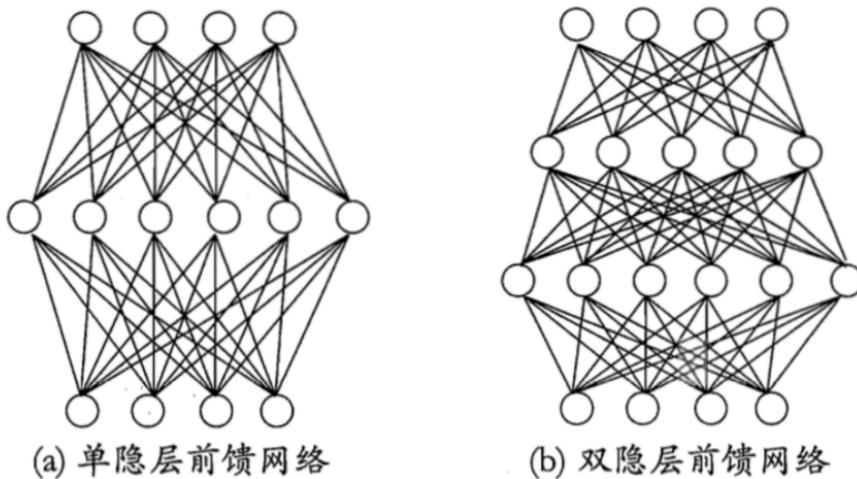
Lecture 9

也就是说，每个神经元接受一系列输入，通过权值、阈值和激活函数进行处理，最终得到一个输出。这一过程很像感知外界信号，处理后输出新信号，因此神经元又被称为感知机。

数学上可以证明，一个神经元（感知机）可以处理线性二分类问题，因此可以进行与运算、或运算这样的基本布尔运算，但无法解决“异或”这样稍复杂些的运算。

既然单个神经元对数据的处理效果有限，人们自然想到，如果我们将多个神经元，按照层叠的方式组合起来，是否就可以处理更加复杂的问题。

事实上，这样我们就得到了一个前馈神经网络。



前馈神经网络由一层层神经元组成，一般分为输入层、隐藏层、输出层三个部分：第一层即为输入层，用来接受外界输入，这一层的神经元个数等于外界输入数据的维度；中间的若干层为隐藏层，用来对数据进行复杂的处理（因为前面说过，只有一层的话，只能拟合较简单的函数，做线性二分类，因此想要拟合复杂的函数就需要增加隐藏层的数量），一般来说，隐藏层的神经元个数会有所变化；最后一层是输出层，用来将隐藏层输出的结果再经过一次处理，成为我们想要的输出，因此，输出层神经元的个数等于我们想要的输出数据的维度。

那么问题来了，我们怎么保证这样的前馈神经网络，能够对数据做出我们想要的处理呢？例如，输入一张黑白图片的三维数据（ x 坐标、 y 坐标、灰度值），输出这张照片是猫、狗的可能性（二维数据）。

这就需要我们对网络进行训练，训练的目标主要是神经元之间连接的权值，最经典的算法是反向传播算法（Backpropagation algorithm, BP算法），BP算法分为正向传播和反向传播两个传播步骤，首先，将一条训练数据（接上例，是一张黑白照片的三维数据）作为网络的输入，网络会通过一层层处理得到一个二维输出（分别为猫、狗的可能性），我们将这个输出与真实值比对，就得到了误差（如均方误差），这一步叫做正向传播。紧接着，由于输入值先影响隐藏层、再影响输出层、再影响误差，对于每一层，我们都计算这一层的误差，按照“按劳分配”的原则，用输出层误差计算隐藏层误差，用隐藏层误差计算输入层误差，这一步叫做反向传播。最后，我们得到了每一层的误差，就可以使用梯度下降的方法，用误差对每一层中神经元的权重进行调节。这样，经过一轮

Lecture 9

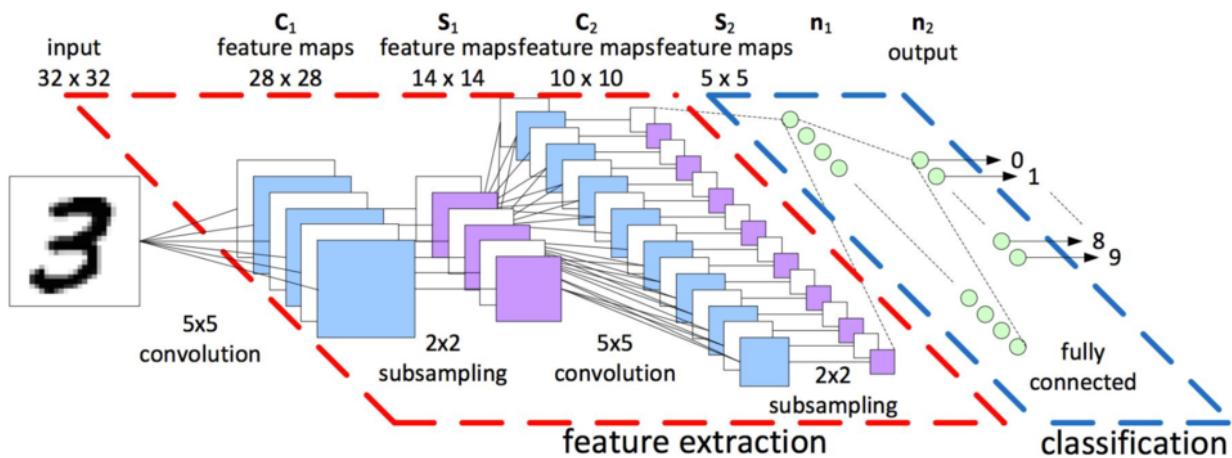
一轮地输入、调整网络，在不陷入局部最优的情况下，我们总能找到最优的参数，使整个训练数据的总体误差最小。

BP算法具体见：<https://www.jianshu.com/p/964345dddb70>

1.2 卷积神经网络

卷积神经网络的结构和前面介绍的前馈神经网络没有太大区别，主要的差别在于隐藏层的构建方式。

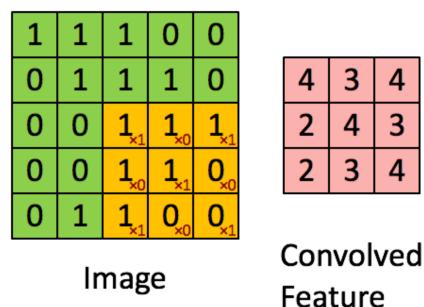
还是以之前的猫狗识别为例，卷积神经网络的常见结构如下：



卷积神经网络中，最重要的是卷积（Convolution）和池化（Pooling）这两个步骤，其中池化又称下采样（Subsampling），在上图中可以看到，卷积和池化这两个步骤是交替进行的。

什么是卷积呢？

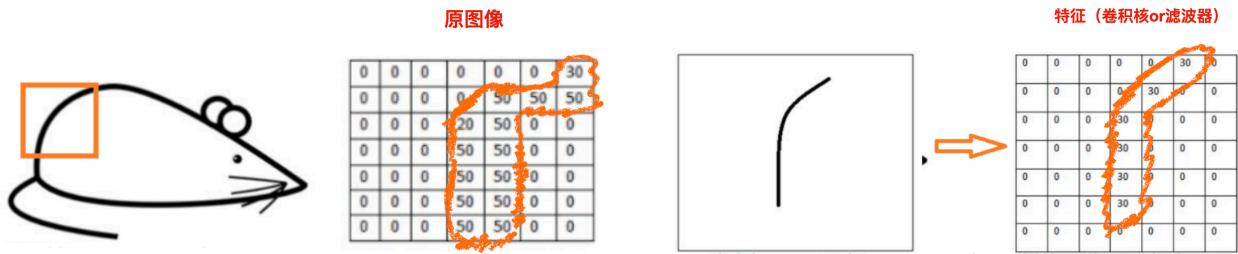
前面已经提过，卷积的目的是为了将数据进行一层层的抽象，这个抽象过程是由卷积核（Convolution Kernel）完成的，卷积核工作的方式如下（以图像识别为例）：



可以看到，卷积核的本质是一个小矩阵，我们将卷积核盖在原始数据上（在这里原始数据是一个三维数据：x, y, 该像素点上是否被占据），计算卷积核和原始数据这一部分每个元素的乘积和（注意：不是矩阵乘法），这样，一个3*3矩阵的信息就被一个数字代替了。紧接着，卷积核向右移动（移动的格数可自定义），逐层逐行地对原矩阵进行扫描，这样就得到了一个卷积后的结果。

Lecture 9

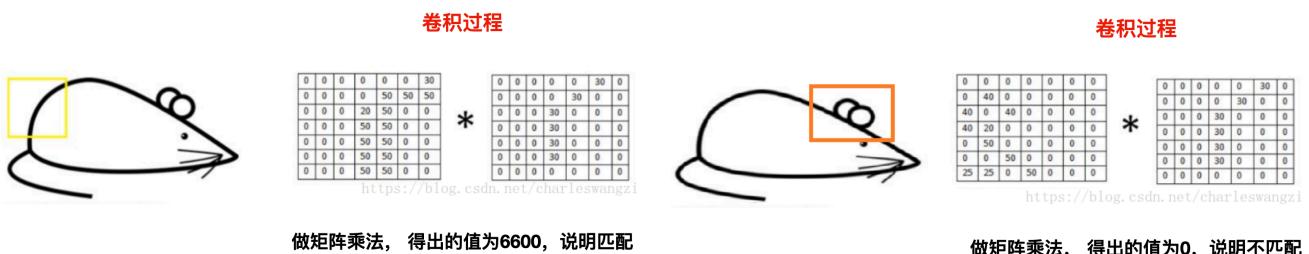
那么大家可能会好奇，这样的卷积操作真的能够抽象出图像的信息嘛？



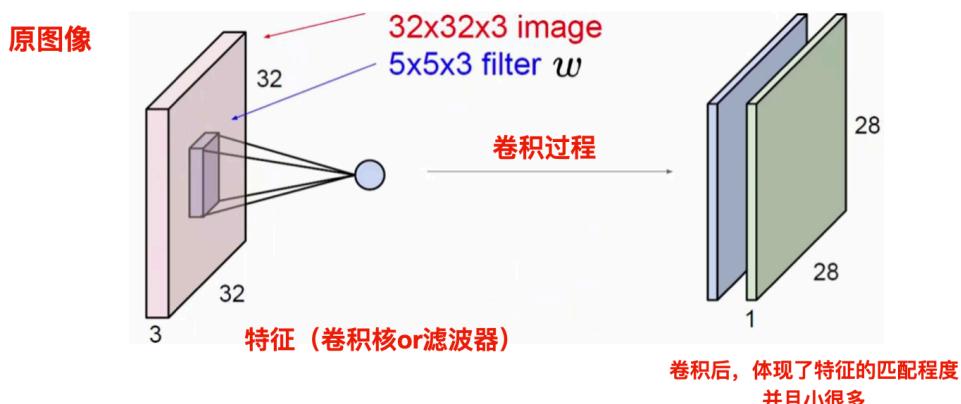
举一个例子可以方便我们理解卷积的过程，如上图，老鼠的屁股部分，正方形区域用矩阵表示的值为左图，可以看到矩阵上的数字组成了老鼠屁股的轮廓。如果我们希望我们的神经网络中，某一个隐藏层就是专门用来匹配老鼠屁股的信息的，就会定义如右图的一个卷积核，可以看到它的特征和我们想要匹配的左图部分是比较相似的。

我们再仔细回顾下卷积核的卷积方式：将卷积核矩阵中的每一个元素和被卷积部分对应位置的元素相乘，并求和。这说明，如果卷积核和被卷积部分越相似，它们相同位置上元素不为零的概率越高，既然卷积的值是对应位置上元素的乘积，那么卷积值就越大。我们就获得了这样的结论：越和卷积核相似的部分，卷积值越大，就越是希望卷积神经网络学习到的规律。

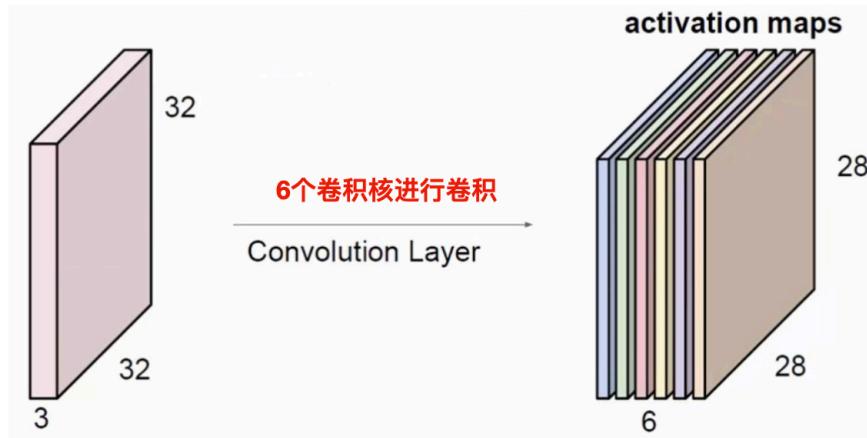
下图以例子证实了这样的逻辑：



因此，每一个卷积核都是为了匹配一种特征，那么在实际操作中，我们的卷积核显然不能只会匹配如此简单的特征，而且也不会只有一个，实际的网络中卷积核的使用更接近下面两张图：



Lecture 9



第一张图描述了更复杂的卷积核，该卷积核是一个 $5 \times 5 \times 3$ 的核，注意到，原始数据也是 $32 \times 32 \times 3$ ，前两个数字都好理解，是图像的尺寸，但为什么图像数据都是会有3呢？这是因为对于彩色图像，我们采用了RGB（红绿蓝）三通道数据来表现单个像素点的色彩。

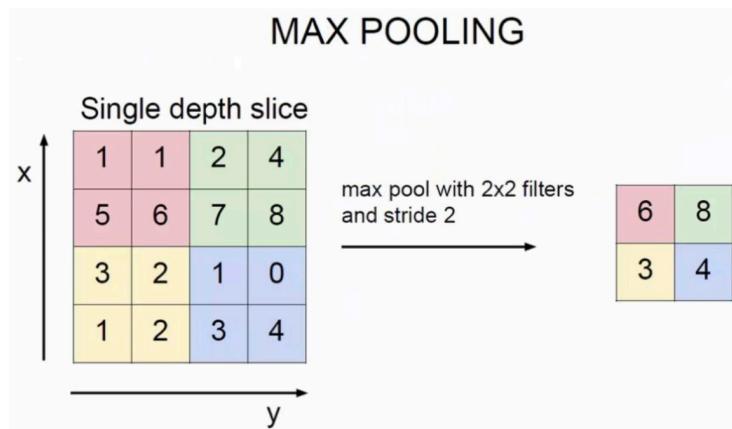
第二张图描述了多个卷积核的情况。

进行完卷积后，下一个重要的步骤是池化（Pooling）。

池化的思想非常简单，因为卷积是逐像素分析的，虽然卷积本身会让原数据矩阵大小一步步缩小，但这样的速度还不够，我们要尽可能减小计算开销，提高网络效率，池化就是再砍一刀，把数据人为再次提炼，保留整体信息，去除细节信息，方便下一步的卷积提取更高层次的特征。

如果我们不对数据进行池化，一是会遇到计算开销很大的问题，二是会“不识庐山真面目，自缘身在此山中”，网络无法从更高视角来看数据。

以Max Pooling为例，池化过程如下：



如图，取区域窗格内的最大值作为该区域的池化结果，拼接起来就得到了整体的池化结果。

卷积神经网络的核心，就是在卷积和池化的交替中，不断捕捉数据的高层次抽象信息。

Lecture 9

2. 使用Python搭建和训练卷积神经网络

2.1 环境搭建

我们建议使用PyTorch作为Python深度学习平台的搭建工具，PyTorch相比于另一大工具：Google推出的TensorFlow，具有更加简洁、更新稳定、易于学习的特点。TensorFlow经过大改版后，原有的语法大量被修改，会导致有时查阅的较早资料无法使用，因此不做推荐（虽然CS学术界更偏爱TensorFlow，其可定制性较强，但我们用不上）。

PyTorch的安装参照：<https://pytorch.org/get-started/locally/>

基本语法，我推荐官方文档的中文开源：<https://pytorch.apachecn.org/docs/1.4/4.html>

PyTorch仅能帮我们搭建网络，定义训练方法。如果大家自己在电脑上执行PyTorch，会默认用CPU进行计算。但深度学习的训练靠CPU是远远不够的，需要调用GPU进行并行计算，主流的计算平台为Nvidia，为了让代码跑在本地的GPU上，还需要布置Nvidia开发的CUDA环境。

（鉴于好像没看到大家用配备高性能GPU的电脑，以及上次组会时我看了下实验室的台式电脑发现并没有配备独立GPU，说明可能大家没有这方面的需求，这里就不介绍CUDA了）

但如果我们突然要用GPU进行深度学习任务，手头的轻薄笔记本执行不了这个任务该怎么办呢？

Google提供了Colab来解决这一问题，Colab免费为用户提供了云端的算力，基础配置为16g的RAM和NVIDIA Tesla K80显卡，足以解决大部分计算任务。此外，Colab的使用方法和大家熟悉的Jupyter Notebook很类似，各位可以很方便地上手。

Colab教程（需要翻墙）：<https://blog.csdn.net/zhouchen1998/article/details/103213108>

2.2 用Python搭建卷积神经网络识别验证码

本部分内容参考：https://github.com/fowill/captcha_torch

材料我也以附件的形式发送给大家，见Lecture9-附件。

（注意⚠！本部分的代码不建议在本地运行，条件允许的话请在colab环境下运行）

卷积神经网络最佳的应用场景是图像识别，这里以日常生活中常见的验证码为例。在creat **captchas.ipynb**中，我们使用代码生成了10000张验证码，其中8000张作为神经网络的训练集，2000张作为测试集。

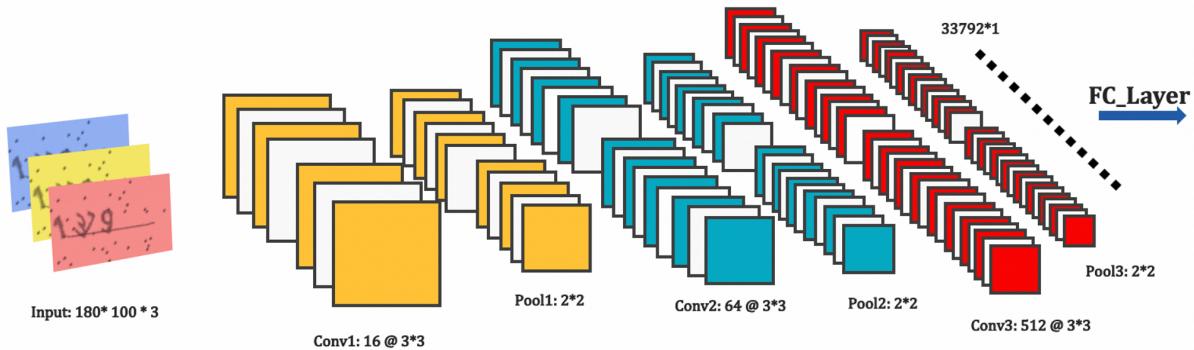
生成的验证码大致如下：



Lecture 9

可以看到，图像中具有较多的点线进行干扰，传统的SVM等方法并不能很好地处理这些干扰，甚至基本的前馈神经网络也无法取得较好效果，卷积神经网络抽象提取信息的特质在这里就派上了用场。

我们先不关注代码，而是思考卷积神经网络的结构设计，我们设计如下图的网络：



复杂版：该网络的输入为 $180*100*3$ 的三通道数据，其中 $180*100$ 为验证码的尺寸，3对应RGB三色通道。网络的主体部分由三组卷积、池化层连接而成。每个卷积层的channel数（每一个卷积层中卷积核的数量）如图所示依次为16、64、512，kernel_size（卷积核的尺寸）均为 $2*2$ 。池化层的kernel_size（在多大的尺寸里挑最大的数做池化）均为 $2*2$ ，在每一个池化层后，为了加速收敛和稳定性，我们还加入了BatchNorm计算，随后为了让模型能学习非线性的划分，我们加入ReLU层作为激活函数。在三组卷积、池化层后，接入一层全连接层。网络的输出结果为四位验证码对应的one-hot编码（例如1111000000表示验证码为0123，每一位上1表示有，0表示无）。

简单版：第一个卷积层用了16个卷积核，第二个用了4个卷积核，第三个用了8个卷积核，对于每一次卷积后，都做了一次 $2*2$ 的池化、一次BatchNorm加速、一层ReLU函数充当激活函数。最后用一个全联接层把已经很复杂的数据（ $512*11*6$ ）转换成我们想要的验证码结果（one-hot形式）

```
In [0]: #Step2 Create a CNN
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self, num_class=10, num_char=4):
        super(CNN, self).__init__()
        self.num_class = num_class
        self.num_char = num_char
        self.conv = nn.Sequential(
            #batch*3*180*100
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=(1, 1)),
            nn.MaxPool2d(2, 2), #size=2,stride=2,which means no overlap between pools
            nn.BatchNorm2d(num_features=16), #use batchnorm to accelerate the training
            nn.ReLU(),#use ReLU to add non-linear factor to cnn model
            #batch*16*90*50
            nn.Conv2d(in_channels=16, out_channels=64, kernel_size=3, padding=(1, 1)),
            nn.MaxPool2d(2, 2),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            #batch*64*45*25
            nn.Conv2d(in_channels=64, out_channels=512, kernel_size=3, padding=(1, 1)),
            nn.MaxPool2d(2, 2),
            nn.BatchNorm2d(num_features=512),
            nn.ReLU(),
            #batch*512*22*12
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=(1, 1)),
            nn.MaxPool2d(2, 2),
            nn.BatchNorm2d(num_features=512),
            nn.ReLU(),
            #batch*512*11*6
        )
        self.fc = nn.Linear(in_features=512*11*6, out_features=self.num_class*self.num_char)

    def forward(self, x):
        x = self.conv(x)
        x = x.view(-1, 512*11*6)
        x = self.fc(x)
        return x
```

Lecture 9

了解了网络设计架构后，相信这时候来看PyTorch的代码就简单不少了：

能看到，PyTorch搭神经网络的过程就像是搭积木，我们只要在nn.Sequential中告诉程序每一层都是什么样的东西（定义过程也简单的离谱），就可以直接完成了。

设计完网络后，我们还需要关注网络训练过程中的参数：

首先，我们设定每次训练读入的数据数目batch_size，这是因为在神经网络训练中，每次的梯度下降固然可以在读入每一个样本后进行，但这样会大大增长训练时间，也会导致收敛缓慢。因此我们选择同时读入一批数据后再进行梯度下降。根据前人经验，这里将batch_size设置为128，即一次性读入128张验证码。

我们随后设定学习率lr，这是用来调整网络参数每次变化的幅度的，如果lr过高，参数变化过快容易学习不到规则，如果lr过低，则会导致训练时间过长。依然是遵循经验，我们将lr设置为0.001。

最后，我们将Loss函数设定为MultiLabelSoftMarginLoss，这是在验证码识别中常用的损失函数。对于多目标分类问题，在预测值为x，真实值为y，x的shape为(N,C)时（其中N代表batch_size，C代表分类数），该损失函数计算公式如下：

$$loss(x, y) = -\frac{1}{C} * \sum_i y[i] * \log((1 + \exp(-x[i]))^{-1}) + (1 - y[i]) * \log(\frac{\exp(-x[i])}{1 + \exp(-x[i])})$$

这些参数在PyTorch中也可以方便地设定（截图没截全）

```
In [0]: #Step3 Training
batch_size = 128
base_lr = 0.001
max_epoch = 40
model_path = './checkpoints/model.pth'
restor = False

if not os.path.exists('./checkpoints'):
    os.mkdir('./checkpoints')

def calculat_acc(output, target):
    output, target = output.view(-1, 10), target.view(-1, 10)
    output = nn.functional.softmax(output, dim=1)
    output = torch.argmax(output, dim=1) #torch.argmax() returns the index of the max element of a iterable object
    target = torch.argmax(target, dim=1)
    output, target = output.view(-1, 4), target.view(-1, 4)
    correct_list = []
    for i, j in zip(target, output):
        if torch.equal(i, j):
            correct_list.append(1)
        else:
            correct_list.append(0)
    acc = sum(correct_list) / len(correct_list)
    return acc

def train():
    transforms = Compose([ToTensor()])
    train_dataset = CaptchaData('./captcha_torch/data/train', transform=transforms)
    train_data_loader = DataLoader(train_dataset, batch_size=batch_size, num_workers=0,
                                   shuffle=True, drop_last=True)
    test_data = CaptchaData('./captcha_torch/data/test', transform=transforms)
    test_data_loader = DataLoader(test_data, batch_size=batch_size,
                                  num_workers=0, shuffle=True, drop_last=True)
    cnn = CNN()
    if torch.cuda.is_available():
        cnn.cuda() #moves all model parameters and buffers to the GPU
    if restor:
        cnn.load_state_dict(torch.load(model_path))

    optimizer = torch.optim.Adam(cnn.parameters(), lr=base_lr)
    criterion = nn.MultiLabelSoftMarginLoss()
```

Lecture 9

可以看到在倒数第二行，我们指定优化器为Adam时，就附带指定了学习率。

说实话，我可以解释一些常见的网络参数，但我也没能力解释清楚每一个深度学习中的参数，很多参数对我们来说意义依然是不明的状态，大部分参数的调整是要靠经验的，这个过程会显得比较无聊，所以大家常称为“炼丹”。

好在最新的技术已经能够让超参数本身都称为训练对象，所有调参过程自动完成。所以也让我们一起期待深度学习走向小学课堂的那天（？）

3. 进阶路线推荐

下面列一些深度学习推荐的材料：

首先依然是PyTorch的官方教程：<https://pytorch.org/tutorials/>

建议看到不懂的函数查阅官方文档：<https://pytorch.org/docs/stable/index.html>

其他资源已经有热心国人帮忙整理到Github上了，有兴趣的同学可以Fork一下：

<https://github.com/INTERMT/Awesome-PyTorch-Chinese>