学习目标：学习CNN基础和原理；使用Pytorch框架构建CNN模型，并完成训练

1 卷积神经网络CNN

1.1 CNN简介

卷积神经网络（简称CNN）是是一类特殊的人工神经网络，是深度学习中重要的一个分支。它专门用来处理具有类似网格结构的数据，比如说时间序列数据（可以认为是在时间轴上有规律地采样形成的一维网格）以及图像数据（二维像素网格）。

CNN在很多领域都表现优异，精度和速度比传统计算学习算法高很多。特别是在计算机视觉领域，CNN是解决图像分类、人脸识别、图像检索、物体检测和语义分割的主流模型。近年来卷积神经网络也广泛地应用到自然语言处理、推荐系统等领域。

1.2 CNN主要原理

1.2.1 使用全链接前馈网络处理图像数据会遇到的问题

卷积神经网络最早是用来处理图像信息，在用全连接前馈网络来处理图像时会存在以下两个问题：

- **参数太多**

  如果输入的图像大小为100 x 100，那么在全连接情况下，输入层到第一层隐藏层需要训练的参数有(100 x 100) x (100 x 100) = 100000000个（假设隐藏层的神经个数和输入层个数相同），很明显参数的规模过于庞大，可能会导致整个神经网络的训练效率很低，而且很容易出现过拟合。

- **局部不变性特征**

  自然图像中的物体都具有局部不变性特征，比如说尺寸缩放、平移、旋转等操作不影响其语义信息。而全连接前馈网络很难提取这些局部不变性特征，一般需要进行数据增强来提高性能。

1.2.2 CNN的四个法宝

第一法宝：局部感知

针对上面参数过多的问题，我们可以进行以下优化，将隐藏层中的每个神经元只与输入层的部分神经元相连接。同样，假设输入的图像大小为100 x 100，全连接情况下，输入层到第一层隐藏层需要训练的参数有(100 x 100) x (100 x 100) = 100000000个；而在局部感知的情况下，我们假设隐藏层中的每个神经元只与输入层的10 x 10个输入（即10 x 10个像素点）相连，则需要训练的参数为(100 x 100) x (10 x 10) = 1000000个，比原来降低了100倍。

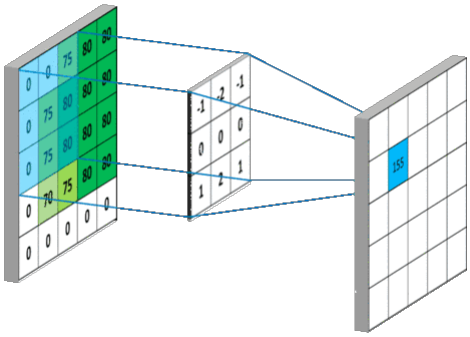
第二法宝：参数共享（以及卷积过程）

其实上面的参数个数还是很多，我们需要进行进一步优化。我们可以假设输入层中每10 x 10个输入连接到一个隐藏层中的参数是一样的。那么我们需要训练的参数数量瞬间降低到了100，是一个可以轻易进行训练的数量级。

那么你可能会问，这样的假设合理么？

答案是：从理论和思想上来说，是合理的，从实践结果的证明来看，是非常好的。这个假设意味着什么呢？这100个参数（就是卷积操作）是一种特征提取，该方式**与位置无关**。这其中隐含的原理则是：图像的一部分的**统计特性**与其他部分是一样的。这也意味着我们在这一部分学习的特征也能用在另一部分上，所以对于这个图像上的所有位置，我们都能使用同样的学习特征。再说白一点儿，一张图片，左半边和右半边的特点，风格一般情况下是一样的，哪怕毕加索这种抽象派大师，喜欢画左右脸不一样的抽象人物，其左右脸的风格也是相同的，找一个毕加索的画作，让你看，你可能一眼就看出来作者十有八九是毕加索，遮住图片的左侧，你可能也比较确认是毕加索的画作，遮住右边也是一样的，这个说法不太严谨，这个“风格”其实就是图像中的各种统计特征（图像编程了数值，那么任何位置都会有一些统计指标，比如极值，均值等）。

这个10 x 10的参数矩阵，就是所谓的卷积核！！！而卷积过程就如下图所示（卷积核为3x3）。



http://blog.csdn.net/weixin_36604953

第三法宝：多核卷积

上面所述只有一个10 x 10的卷积核，有100个参数，显然，特征提取是不充分的，我们可以添加多个卷积核，比如32个卷积核，可以学习32种特征。这样，通过多个卷积核的操作，对图像的特征提取就更加充分了。

第四法宝：池化

有时图像太大，即使我们参数不太多，但图像的像素实在太多，导致卷积操作后，我们得到的结果仍然过大。我们需要减少训练参数的数量，它被要求在随后的卷积层之间周期性地引进池化层。**池化的一个目的是减少图像的空间大小**。池化在每一个纵深维度上独自完成，因此图像的纵深保持不变。池化的主要形式有最大池化和平均池化。

池化还有一个目的是保持**平移不变性**。卷积对输入有平移不变性，池化对特征有平移不变性。平移不变性是什么呢？因为卷积核是在输入图或者feature maps上滑动，或者说平移，每次平移时，因此假设使用max pooling，会过滤掉那些不明显、未被激活的特征。

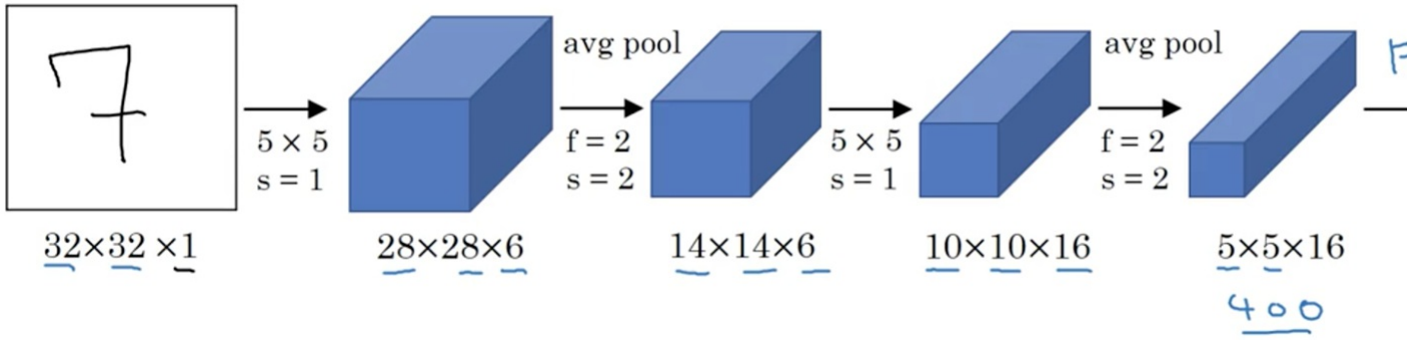
下图是最大池化的一个例子。在这里，我们把步幅定为 2，池化尺寸也为 2。也就是对下图左侧4 x 4的矩阵，用一个2 x 2的窗口去以2为步长去遍历，再直观的说，我们按照横向和纵向两条中轴线将他切成4个2 x 2的矩阵，然后取每个矩阵的最大值，作为池化后的结果，就得到了下图右侧的池化结果。最大化执行也应用在每个卷机输出的深度尺寸中。正如你所看到的，最大池化操作后，4 x 4 卷积的输出变成了 2 x 2。

429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657

792	856
913	851

2 经典的CNN网络

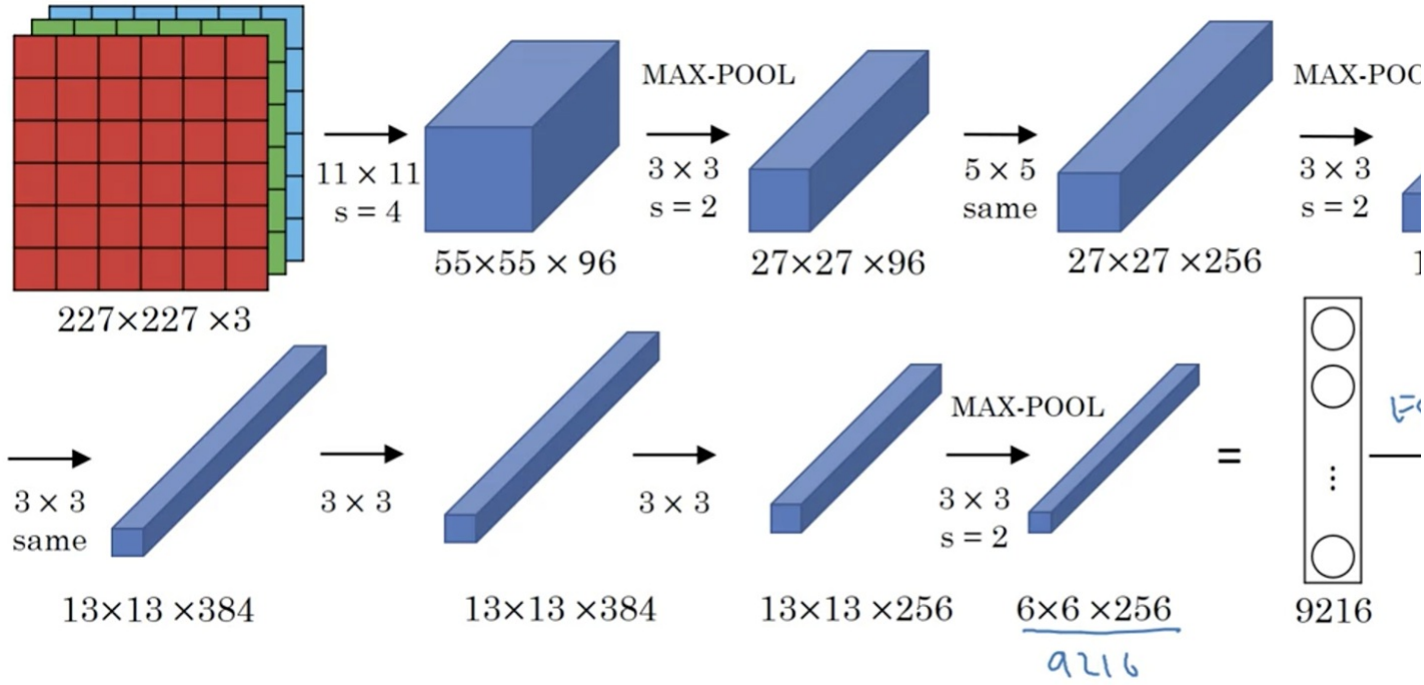
1 LeNet



LeNet-5 一些性质：

- 如果输入层不算神经网络的层数，那么 LeNet-5 是一个 7 层的网络。（有些地方也可能把 卷积和池化 当作一个 layer）（LeNet-5 名字中的“5”也可以理解为整个网络中含可训练参数的层数为 5。）
- LeNet-5 大约有 60,000 个参数。
- 随着网络越来越深，图像的高度和宽度在缩小，与此同时，图像的 channel 数量一直在增加。
- 现在常用的 LeNet-5 结构和 Yann LeCun 教授在 1998 年论文中提出的结构在某些地方有区别，比如激活函数的使用，现在一般使用 ReLU 作为激活函数，输出层一般选择 softmax。

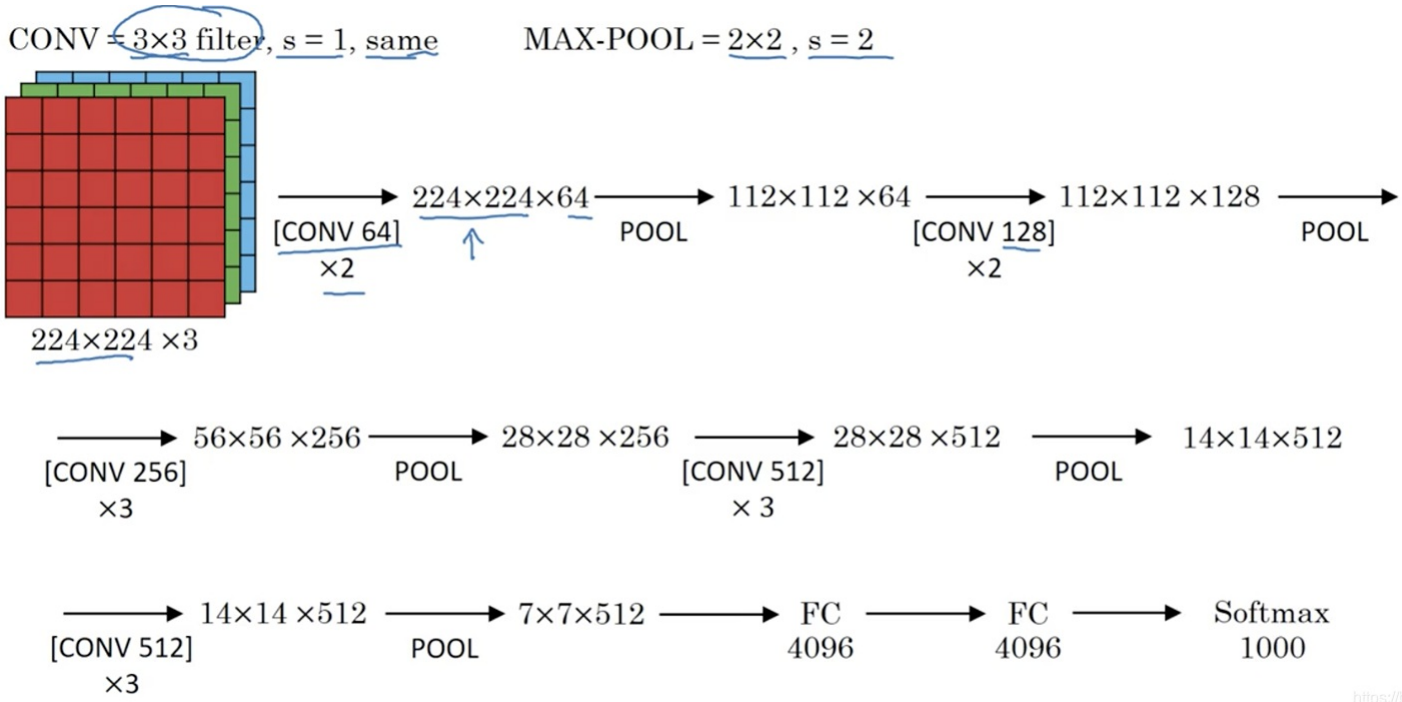
2 AlexNet



AlexNet 一些性质：

- 大约 60million 个参数；
- 使用 ReLU 作为激活函数。

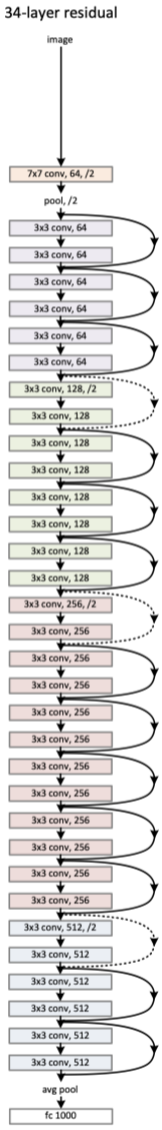
3 VGG-16



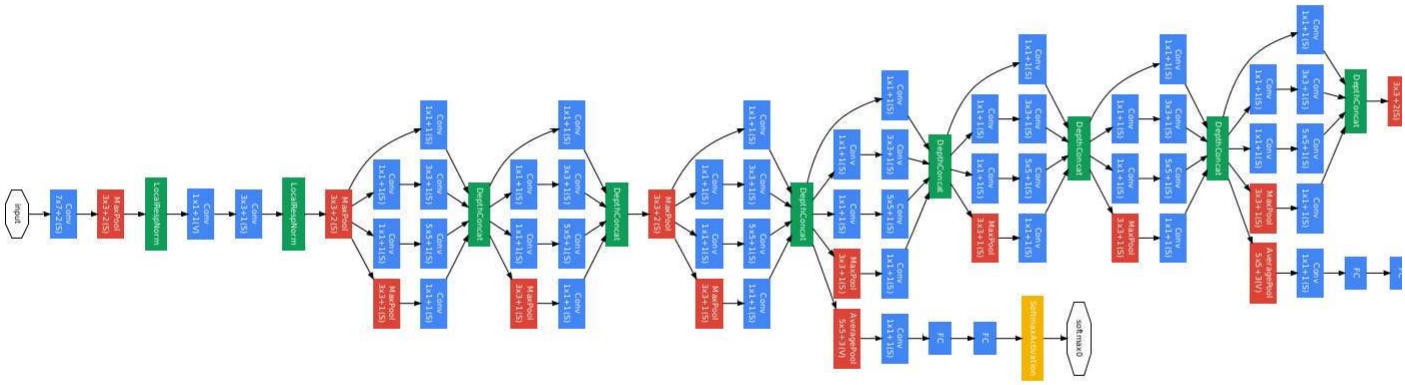
VGG-16 一些性质：

- VGG-16 中的 16 表示整个网络中有 trainable 参数的层数为 16 层。（trainable 参数指的是可以通过 back-propagation 更新的参数）
- VGG-16 大约有 138million 个参数。
- VGG-16 中所有卷积层 filter 宽和高都是 3，步长为 1，padding 都使用 same convolution；所有池化层的 filter 宽和高都是 2，步长都是 2。

4 ResNet



5 Inception



3 Pytorch构建CNN模型

在Pytorch中构建CNN模型非常简单，只需要定义好模型的参数和正向传播即可，Pytorch会根据正向传播自动计算反向传播。

下面的代码中构建了一个非常简单的CNN模型，并进行了训练。这个CNN模型包括两个卷积层，最后并联6个全连接层进行分类。

```
import torch
torch.manual_seed(0)
torch.backends.cudnn.deterministic = False
torch.backends.cudnn.benchmark = True

import torchvision.models as models
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torch.utils.data.dataset import Dataset

# 定义模型
class SVHN_Model1(nn.Module):
    def __init__(self):
        super(SVHN_Model1, self).__init__()
        # CNN提取特征模块
        self.cnn = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2)),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2)),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        #
        self.fc1 = nn.Linear(32*3*7, 11)
        self.fc2 = nn.Linear(32*3*7, 11)
        self.fc3 = nn.Linear(32*3*7, 11)
        self.fc4 = nn.Linear(32*3*7, 11)
        self.fc5 = nn.Linear(32*3*7, 11)
        self.fc6 = nn.Linear(32*3*7, 11)

    def forward(self, img):
        feat = self.cnn(img)
        feat = feat.view(feat.shape[0], -1)
        c1 = self.fc1(feat)
        c2 = self.fc2(feat)
        c3 = self.fc3(feat)
        c4 = self.fc4(feat)
        c5 = self.fc5(feat)
        c6 = self.fc6(feat)
        return c1, c2, c3, c4, c5, c6

model = SVHN_Model1()
```

下面是训练代码：

```
# 损失函数
criterion = nn.CrossEntropyLoss()
# 优化器
optimizer = torch.optim.Adam(model.parameters(), 0.005)

loss_plot, c0_plot = [], []
# 迭代10个Epoch
for epoch in range(10):
    for data in train_loader:
        c0, c1, c2, c3, c4, c5 = model(data[0])
        loss = criterion(c0, data[1][:, 0]) + \
                criterion(c1, data[1][:, 1]) + \
                criterion(c2, data[1][:, 2]) + \
                criterion(c3, data[1][:, 3]) + \
                criterion(c4, data[1][:, 4]) + \
                criterion(c5, data[1][:, 5])
        loss /= 6
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        loss_plot.append(loss.item())
        c0_plot.append((c0.argmax(1) == data[1][:, 0]).sum().item()*1.0 / c0.shape[0])

    print(epoch)
```

当然为了追求精度，也可以使用在ImageNet数据集上的预训练模型，具体方法如下：

```
class SVHN_Model2(nn.Module):
    def __init__(self):
        super(SVHN_Model1, self).__init__()

        model_conv = models.resnet18(pretrained=True)
        model_conv.avgpool = nn.AdaptiveAvgPool2d(1)
        model_conv = nn.Sequential(*list(model_conv.children())[:-1])
```

```
        self.cnn = model_conv

        self.fc1 = nn.Linear(512, 11)
        self.fc2 = nn.Linear(512, 11)
        self.fc3 = nn.Linear(512, 11)
        self.fc4 = nn.Linear(512, 11)
        self.fc5 = nn.Linear(512, 11)

    def forward(self, img):
        feat = self.cnn(img)
        # print(feat.shape)
        feat = feat.view(feat.shape[0], -1)
        c1 = self.fc1(feat)
        c2 = self.fc2(feat)
        c3 = self.fc3(feat)
        c4 = self.fc4(feat)
        c5 = self.fc5(feat)
        return c1, c2, c3, c4, c5
```

4 小结

通过该任务的学习，基本了解了CNN相关的基础知识和原理，以及几个比较经典的CNN网络。并学会了使用PyTorch框架构建CNN模型和训练，然后PyTorch里面还有很多预训练模型，我们可以通过迁移学习来直接使用这些预训练模型。

虽然理论知识的学习很重要，但是我们也要重视实践。怎么样选择最合适的模型，怎么样进行训练，怎么样解决过拟合，以及怎么样进一步优化模型，提高准确率，这都是我们在实际应用中将会遇到的问题。只有真正经历了这些问题，并且逐一解决，我们才能得到成长。