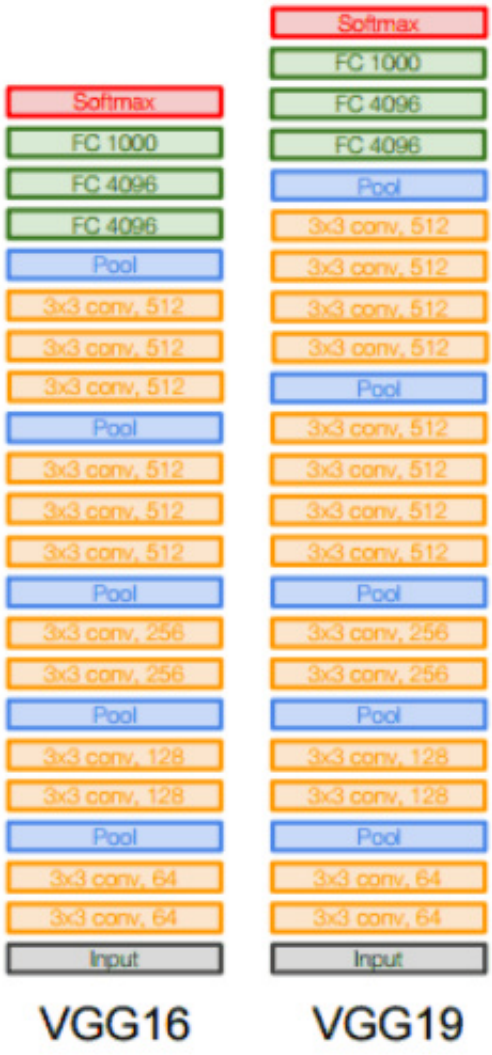
VGGNet

vggNet 是第一个真正意义上的深层网络结构，其是 ImageNet2014年的冠军，得益于 python 的函数和循环，我们能够非常方便地构建重复结构的深层网络。

vgg 的网络结构非常简单，就是不断地堆叠卷积层和池化层，下面是一个简单的图示



vgg 几乎全部使用 3 x 3 的卷积核以及 2 x 2 的池化层，使用小的卷积核进行多层的堆叠和一个大的卷积核的感受野是相同的，同时小的卷积核还能减少参数，同时可以有更深的结构。

vgg 的一个关键就是使用很多层 3 x 3 的卷积然后再使用一个最大池化层，这个模块被使用了很多次，下面我们照着这个结构来写一写

```
import numpy as np
import torch
from torch import nn
from torch.autograd import Variable
from torchvision.datasets import CIFAR10
```

我们可以定义一个 vgg 的 block，传入三个参数，第一个是模型层数，第二个是输入的通道数，第三个是输出的通道数，第一层卷积接受的输入通道就是图片输入的通道数，然后输出最后的输出通道数，后面的卷积接受的通道数就是最后的输出通道数

```
def vgg_block(num_convs, in_channels, out_channels):
    net = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
nn.ReLU(True)] # 定义第一层

    for i in range(num_convs-1): # 定义后面的很多层
        net.append(nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1))
        net.append(nn.ReLU(True))

    net.append(nn.MaxPool2d(2, 2)) # 定义池化层
    return nn.Sequential(*net)
```

我们可以将模型打印出来看看结构

```
block_demo = vgg_block(3, 64, 128)
print(block_demo)
```

```
Sequential(
  (0): Conv2d (64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace)
  (2): Conv2d (128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace)
  (4): Conv2d (128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): ReLU(inplace)
  (6): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
)
```

```
# 首先定义输入为 (1, 64, 300, 300)
input_demo = Variable(torch.zeros(1, 64, 300, 300))
output_demo = block_demo(input_demo)
print(output_demo.shape)
```

```
torch.Size([1, 128, 150, 150])
```

可以看到输出就变为了 (1, 128, 150, 150)，可以看到经过了这一个 vgg block，输入大小被减半，通道数变成了 128

下面我们定义一个函数对这个 vgg block 进行堆叠

```
def vgg_stack(num_convs, channels):
    net = []
    for n, c in zip(num_convs, channels):
        in_c = c[0]
        out_c = c[1]
        net.append(vgg_block(n, in_c, out_c))
    return nn.Sequential(*net)
```

作为实例，我们定义一个稍微简单一点的 vgg 结构，其中有 8 个卷积层

```
vgg_net = vgg_stack((1, 1, 2, 2, 2), ((3, 64), (64, 128), (128, 256), (256, 512),
(512, 512)))
print(vgg_net)
```

```
Sequential(
  (0): Sequential(
    (0): Conv2d (3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (1): Sequential(
    (0): Conv2d (64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (2): Sequential(
    (0): Conv2d (128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (3): Sequential(
    (0): Conv2d (256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (4): Sequential(
    (0): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
)
```

我们可以看到网络结构中有个 5 个 最大池化，说明图片的大小会减少 5 倍，我们可以验证一下，输入一张 256 x 256 的图片看看结果是什么

```
test_x = Variable(torch.zeros(1, 3, 256, 256))
test_y = vgg_net(test_x)
print(test_y.shape)
```

```
torch.Size([1, 512, 8, 8])
```

可以看到图片减小了 2^5 倍，最后再加上几层全连接，就能够得到我们想要的分类输出

```
class vgg(nn.Module):
    def __init__(self):
        super(vgg, self).__init__()
        self.feature = vgg_net
        self.fc = nn.Sequential(
            nn.Linear(512, 100),
            nn.ReLU(True),
            nn.Linear(100, 10)
        )
    def forward(self, x):
        x = self.feature(x)
        x = x.view(x.shape[0], -1)
        x = self.fc(x)
        return x
```

然后我们可以训练我们的模型看看在 cifar10 上的效果

```
from utils import train
```

```
def data_tf(x):
    x = np.array(x, dtype='float32') / 255
    x = (x - 0.5) / 0.5 # 标准化，这个技巧之后会讲到
    x = x.transpose((2, 0, 1)) # 将 channel 放到第一维，只是 pytorch 要求的输入方式
    x = torch.from_numpy(x)
    return x

train_set = CIFAR10('./data', train=True, transform=data_tf)
train_data = torch.utils.data.DataLoader(train_set, batch_size=64, shuffle=True)
test_set = CIFAR10('./data', train=False, transform=data_tf)
test_data = torch.utils.data.DataLoader(test_set, batch_size=128, shuffle=False)

net = vgg()
optimizer = torch.optim.SGD(net.parameters(), lr=1e-1)
criterion = nn.CrossEntropyLoss()
```

```
train(net, train_data, test_data, 20, optimizer, criterion)
```

```
Epoch 0. Train Loss: 2.303118, Train Acc: 0.098186, Valid Loss: 2.302944, Valid Acc:
0.099585, Time 00:00:32
Epoch 1. Train Loss: 2.303085, Train Acc: 0.096907, Valid Loss: 2.302762, Valid Acc:
0.100969, Time 00:00:33
Epoch 2. Train Loss: 2.302916, Train Acc: 0.097287, Valid Loss: 2.302740, Valid Acc:
0.099585, Time 00:00:33
Epoch 3. Train Loss: 2.302395, Train Acc: 0.102042, Valid Loss: 2.297652, Valid Acc:
0.108782, Time 00:00:32
Epoch 4. Train Loss: 2.079523, Train Acc: 0.202026, Valid Loss: 1.868179, Valid Acc:
0.255736, Time 00:00:31
Epoch 5. Train Loss: 1.781262, Train Acc: 0.307625, Valid Loss: 1.735122, Valid Acc:
0.323279, Time 00:00:31
Epoch 6. Train Loss: 1.565095, Train Acc: 0.400975, Valid Loss: 1.463914, Valid Acc:
0.449565, Time 00:00:31
Epoch 7. Train Loss: 1.360450, Train Acc: 0.495225, Valid Loss: 1.374488, Valid Acc:
0.490803, Time 00:00:31
Epoch 8. Train Loss: 1.144470, Train Acc: 0.585758, Valid Loss: 1.384803, Valid Acc:
0.524624, Time 00:00:31
Epoch 9. Train Loss: 0.954556, Train Acc: 0.659287, Valid Loss: 1.113850, Valid Acc:
0.609968, Time 00:00:32
Epoch 10. Train Loss: 0.801952, Train Acc: 0.718131, Valid Loss: 1.080254, Valid Acc:
0.639933, Time 00:00:31
Epoch 11. Train Loss: 0.665018, Train Acc: 0.765945, Valid Loss: 0.916277, Valid Acc:
0.698972, Time 00:00:31
Epoch 12. Train Loss: 0.547411, Train Acc: 0.811241, Valid Loss: 1.030948, Valid Acc:
0.678896, Time 00:00:32
```

```
Epoch 13. Train Loss: 0.442779, Train Acc: 0.846228, Valid Loss: 0.869791, Valid Acc:
0.732496, Time 00:00:32
Epoch 14. Train Loss: 0.357279, Train Acc: 0.875440, Valid Loss: 1.233777, Valid Acc:
0.671677, Time 00:00:31
Epoch 15. Train Loss: 0.285171, Train Acc: 0.900096, Valid Loss: 0.852879, Valid Acc:
0.765131, Time 00:00:32
Epoch 16. Train Loss: 0.222431, Train Acc: 0.923374, Valid Loss: 1.848096, Valid Acc:
0.614023, Time 00:00:31
Epoch 17. Train Loss: 0.174834, Train Acc: 0.939478, Valid Loss: 1.137286, Valid Acc:
0.728639, Time 00:00:31
Epoch 18. Train Loss: 0.144375, Train Acc: 0.950587, Valid Loss: 0.907310, Valid Acc:
0.776800, Time 00:00:31
Epoch 19. Train Loss: 0.115332, Train Acc: 0.960878, Valid Loss: 1.009886, Valid Acc:
0.761175, Time 00:00:31
```

可以看到，跑完 20 次，vgg 能在 cifar 10 上取得 76% 左右的测试准确率