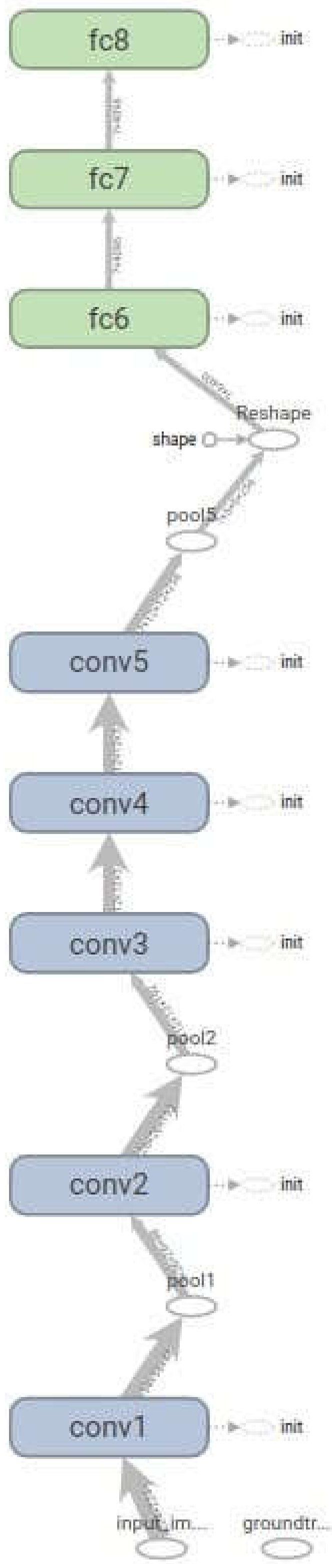
AlexNet

在2012年, 由 Alex Krizhevsky (https://www.cs.toronto.edu/~kriz/), Ilya Sutskever (http://www.cs.toronto.edu/~ilya/), Geoffrey Hinton (http://www.cs.toronto.edu/~hinton/)提出了一种使用卷积神经网络的方法, 以 0.85 (http://image-net.org/challenges/LSVRC/2012/results.html#abstract) 的 `top-5` 正确率一举获得当年分类比赛的冠军, 超越使用传统方法的第二名10个百分点, 震惊了当时的学术界, 从此开启了人工智能领域的新篇章.

下面复现一次 `AlexNet`，首先来看它的网络结构



可以看出 `AlexNet` 就是几个卷积池化堆叠后连接几个全连接层, 下面就让我们来尝试仿照这个结构来解决[cifar10 (https://www.cs.toronto.edu/~kriz/cifar.html)](https://www.cs.toronto.edu/~kriz/cifar.html)分类问题.

In [1]:

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

依照上面的结构，我们可以定义 AlexNet

In [2]:

```
class AlexNet(nn.Module):
    def __init__(self):
        super().__init__()

        # 第一层是 5x5 的卷积，输入的 channels 是 3，输出的 channels 是 64，步长是 1，没有 padd
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, 5),
            nn.ReLU(True))

        # 第二层是 3x3 的池化，步长是 2，没有 padding
        self.max_pool1 = nn.MaxPool2d(3, 2)

        # 第三层是 5x5 的卷积，输入的 channels 是 64，输出的 channels 是 64，步长是 1，没有 pad
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 64, 5, 1),
            nn.ReLU(True))

        # 第四层是 3x3 的池化，步长是 2，没有 padding
        self.max_pool2 = nn.MaxPool2d(3, 2)

        # 第五层是全连接层，输入是 1204，输出是 384
        self.fc1 = nn.Sequential(
            nn.Linear(1024, 384),
            nn.ReLU(True))

        # 第六层是全连接层，输入是 384，输出是 192
        self.fc2 = nn.Sequential(
            nn.Linear(384, 192),
            nn.ReLU(True))

        # 第七层是全连接层，输入是 192，输出是 10
        self.fc3 = nn.Linear(192, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.max_pool1(x)
        x = self.conv2(x)
        x = self.max_pool2(x)

        # 将矩阵拉平
        x = x.view(x.shape[0], -1)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

In [3]:

```
alexnet = AlexNet()
```

打印一下网络的结构

In [4]:

```
alexnet
```

Out[4]:

```
AlexNet(
  (conv1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace)
  )
  (max_pool1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
  (conv2): Sequential(
    (0): Conv2d(64, 64, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace)
  )
  (max_pool2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
  (fc1): Sequential(
    (0): Linear(in_features=1024, out_features=384, bias=True)
    (1): ReLU(inplace)
  )
  (fc2): Sequential(
    (0): Linear(in_features=384, out_features=192, bias=True)
    (1): ReLU(inplace)
  )
  (fc3): Linear(in_features=192, out_features=10, bias=True)
)
```

我们验证一下网络结构是否正确，输入一张 32 x 32 的图片，看看输出

In [5]:

```
# 定义输入为 (1, 3, 32, 32)
input_demo = Variable(torch.zeros(1, 3, 32, 32))
output_demo = alexnet(input_demo)
print(output_demo.shape)
```

```
torch.Size([1, 10])
```

In [7]:

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

In [8]:

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

```
F:\Notebook\pytorch_learning\utils.py:52: UserWarning: volatile was removed and now
has no effect. Use `with torch.no_grad():` instead.
  im = Variable(im.cuda(), volatile=True)
F:\Notebook\pytorch_learning\utils.py:53: UserWarning: volatile was removed and now
has no effect. Use `with torch.no_grad():` instead.
  label = Variable(label.cuda(), volatile=True)


Epoch 0. Train Loss: 1.702650, Train Acc: 0.378357, Valid Loss: 1.759744, Valid Acc:
0.398240, Time 00:00:14
Epoch 1. Train Loss: 1.249858, Train Acc: 0.554727, Valid Loss: 1.367568, Valid Acc:
0.522053, Time 00:00:13
Epoch 2. Train Loss: 1.020488, Train Acc: 0.641724, Valid Loss: 1.037739, Valid Acc:
0.637757, Time 00:00:13
Epoch 3. Train Loss: 0.870957, Train Acc: 0.692755, Valid Loss: 1.024721, Valid Acc:
0.648536, Time 00:00:13
Epoch 4. Train Loss: 0.755434, Train Acc: 0.735074, Valid Loss: 1.336667, Valid Acc:
0.580993, Time 00:00:13
Epoch 5. Train Loss: 0.659368, Train Acc: 0.767923, Valid Loss: 0.794070, Valid Acc:
0.729035, Time 00:00:13
Epoch 6. Train Loss: 0.577186, Train Acc: 0.796475, Valid Loss: 0.836620, Valid Acc:
0.724881, Time 00:00:13
Epoch 7. Train Loss: 0.504485, Train Acc: 0.820952, Valid Loss: 0.914811, Valid Acc:
0.709059, Time 00:00:13
Epoch 8. Train Loss: 0.439278, Train Acc: 0.844609, Valid Loss: 1.062373, Valid Acc:
0.685522, Time 00:00:13
Epoch 9. Train Loss: 0.374409, Train Acc: 0.867008, Valid Loss: 2.212047, Valid Acc:
0.545787, Time 00:00:13
Epoch 10. Train Loss: 0.327986, Train Acc: 0.884071, Valid Loss: 1.199214, Valid Ac
c: 0.689775, Time 00:00:13
Epoch 11. Train Loss: 0.281260, Train Acc: 0.900675, Valid Loss: 1.485560, Valid Ac
c: 0.660206, Time 00:00:13
Epoch 12. Train Loss: 0.239985, Train Acc: 0.914402, Valid Loss: 1.121117, Valid Ac
c: 0.723892, Time 00:00:13
Epoch 13. Train Loss: 0.209129, Train Acc: 0.926071, Valid Loss: 1.173841, Valid Ac
c: 0.727255, Time 00:00:13
Epoch 14. Train Loss: 0.180579, Train Acc: 0.937460, Valid Loss: 1.526777, Valid Ac
c: 0.684533, Time 00:00:13
Epoch 15. Train Loss: 0.162098, Train Acc: 0.944054, Valid Loss: 1.705790, Valid Ac
c: 0.682456, Time 00:00:13
Epoch 16. Train Loss: 0.156392, Train Acc: 0.946132, Valid Loss: 2.873939, Valid Ac
c: 0.598892, Time 00:00:13
Epoch 17. Train Loss: 0.184734, Train Acc: 0.940437, Valid Loss: 2.599990, Valid Ac
c: 0.574565, Time 00:00:13
Epoch 18. Train Loss: 0.114861, Train Acc: 0.960558, Valid Loss: 1.701673, Valid Ac
c: 0.704411, Time 00:00:13
Epoch 19. Train Loss: 0.115083, Train Acc: 0.960658, Valid Loss: 1.675298, Valid Ac
c: 0.717366, Time 00:00:13
```

可以看到，训练 20 次，AlxeNet 能够在 cifar 10 上取得 70% 左右的测试集准确率

In [ ]:

```

```