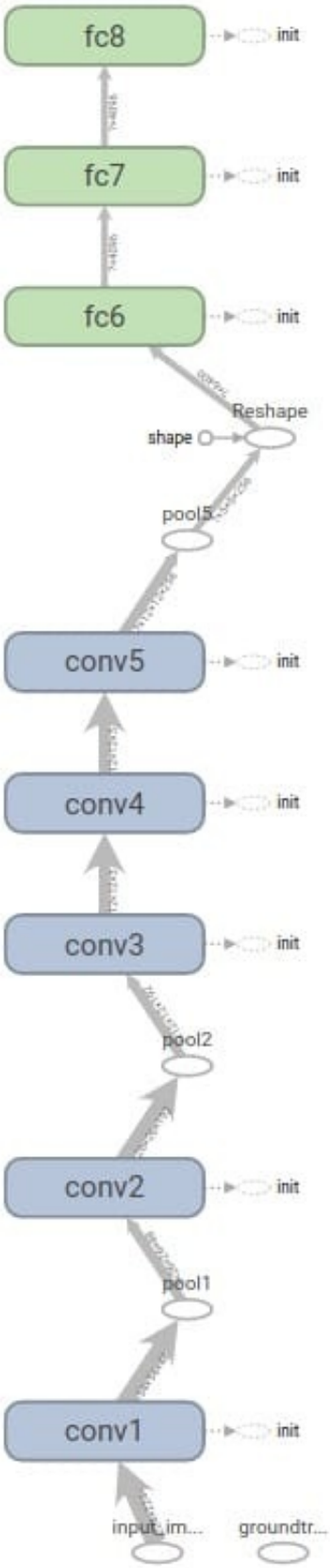
AlexNet

在2012年, 由 Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton提出了一种使用卷积神经网络的方法, 以 0.85 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.

这次的课程我们就来复现一次`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](#)分类问题.

```
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

```
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 padding
        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 padding
        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
```

```
alexnet = AlexNet()
```

打印一下网络的结构

```
alexnet
```

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

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

```
# 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])
```

```
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()
```

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

```
Epoch 0. Train Loss: 1.705926, Train Acc: 0.374520, Valid Loss: 1.530230, Valid Acc:
0.450257, Time 00:00:12
Epoch 1. Train Loss: 1.246754, Train Acc: 0.555766, Valid Loss: 1.431630, Valid Acc:
0.513350, Time 00:00:11
Epoch 2. Train Loss: 1.010857, Train Acc: 0.644222, Valid Loss: 1.190140, Valid Acc:
0.584751, Time 00:00:10
Epoch 3. Train Loss: 0.855797, Train Acc: 0.698789, Valid Loss: 0.994367, Valid Acc:
0.657041, Time 00:00:12
Epoch 4. Train Loss: 0.747337, Train Acc: 0.739250, Valid Loss: 1.434243, Valid Acc:
0.570906, Time 00:00:11
Epoch 5. Train Loss: 0.656047, Train Acc: 0.771639, Valid Loss: 0.981409, Valid Acc:
0.684434, Time 00:00:13
Epoch 6. Train Loss: 0.567300, Train Acc: 0.801251, Valid Loss: 0.834201, Valid Acc:
0.722409, Time 00:00:13
```

```
Epoch 7. Train Loss: 0.497418, Train Acc: 0.824309, Valid Loss: 0.978063, Valid Acc:
0.702927, Time 00:00:11
Epoch 8. Train Loss: 0.429598, Train Acc: 0.850164, Valid Loss: 0.945215, Valid Acc:
0.706487, Time 00:00:11
Epoch 9. Train Loss: 0.369809, Train Acc: 0.868666, Valid Loss: 1.063736, Valid Acc:
0.699565, Time 00:00:11
Epoch 10. Train Loss: 0.319623, Train Acc: 0.886589, Valid Loss: 0.987968, Valid Acc:
0.714201, Time 00:00:11
Epoch 11. Train Loss: 0.274986, Train Acc: 0.901934, Valid Loss: 1.396439, Valid Acc:
0.662975, Time 00:00:11
Epoch 12. Train Loss: 0.238880, Train Acc: 0.915181, Valid Loss: 1.130270, Valid Acc:
0.717860, Time 00:00:10
Epoch 13. Train Loss: 0.204735, Train Acc: 0.928688, Valid Loss: 1.373174, Valid Acc:
0.699367, Time 00:00:10
Epoch 14. Train Loss: 0.176767, Train Acc: 0.937900, Valid Loss: 2.049194, Valid Acc:
0.650811, Time 00:00:10
Epoch 15. Train Loss: 0.166942, Train Acc: 0.943374, Valid Loss: 1.377223, Valid Acc:
0.719838, Time 00:00:10
Epoch 16. Train Loss: 0.146755, Train Acc: 0.948449, Valid Loss: 1.456631, Valid Acc:
0.722211, Time 00:00:11
Epoch 17. Train Loss: 0.124807, Train Acc: 0.957241, Valid Loss: 1.543074, Valid Acc:
0.730024, Time 00:00:10
Epoch 18. Train Loss: 0.126861, Train Acc: 0.956702, Valid Loss: 1.417461, Valid Acc:
0.732595, Time 00:00:12
Epoch 19. Train Loss: 0.103216, Train Acc: 0.964954, Valid Loss: 2.383004, Valid Acc:
0.653283, Time 00:00:12
```

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