代码基本结构

卷积神经网络

定义了卷积层两层，池化层两层，最后一层为FC层，输出结果

第一个维度是深度，后两个是宽度和高度

卷积层1

1 28 28到25 26 26

nn.Conv2d()方法中，指定in_channels=1，out_channels=25，kernel_size=3

BatchNorm2d()方法进行了数据的归一化处理

ReLU()将数据过一遍ReLU函数

```
        self.ConvolutionLayer1 = nn.Sequential(
            nn.Conv2d(1, 25, kernel_size=3),
            nn.BatchNorm2d(25),
            nn.ReLU(inplace=True)
        )
```

池化层1

25 26 26到25 13 13

kernel_size指定最大池化操作时的窗口大小，stride指定最大池化操作时窗口移动的步长

```
        self.MaxPoolingLayer2 = nn.Sequential(
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

卷积层2

25 13 13到50 3 3

```
        self.ConvolutionLayer3 = nn.Sequential(
            nn.Conv2d(25, 50, kernel_size=3),
            nn.BatchNorm2d(50),
            nn.ReLU(inplace=True)
        )
```

池化层2

50 3 3到50 5 5

```
        self.MaxPoolingLayer4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

FC层

50 5 5到1024到128到12

给出12种字每个字的可能性

```
        self.FCLayer = nn.Sequential(
            nn.Linear(50 * 5 * 5, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 12)
        )
```

MySet

继承Dataset类，加载自己的数据，为在DataLoader中初始化做准备

```
class MySet(data.Dataset):

    def __init__(self,root):
        # 所有图片的绝对路径
        imgs=os.listdir(root)
        self.imgs = [imgs[0]] * len(imgs)
        for i in range(1,len(imgs)+1):
            wjm = str(i) + '.bmp'
            self.imgs[i-1] = os.path.join(root,wjm)
        print(self.imgs)
        self.transforms=transform

    def __getitem__(self, index):
        img_path = self.imgs[index]
        pil_img = Image.open(img_path)
        if self.transforms:
            data = self.transforms(pil_img)
        else:
            pil_img = np.asarray(pil_img)
            data = torch.from_numpy(pil_img)
        return data

    def __len__(self):
        return len(self.imgs)
```

主方法

首先进行数据预处理。transforms.ToTensor()将图片转换成PyTorch中处理的对象Tensor,并且进行标准化（数据在0~1之间）

transforms.Normalize()做归一化。它进行了减均值，再除以标准差。两个参数分别是均值和标准差

```
data_tf = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize([0.5], [0.5])])
```

加载训练数据和测试数据进行训练和测试

```
train_dataset = ImageFolder('train\\', transform=data_tf)
test_dataset = MySet('./ttest')
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset,shuffle=False)
# 选择模型
model = cnn.CNN()
# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(),lr=0.001, betas=(0.9, 0.999), eps=1e-
08, weight_decay=0)

# 训练模型
for i in range(1,num_epoches):
    for data in train_loader:
        # img是3*28*28列表 label是长度64列表
        img = data[0]
        img = img.narrow(1,0,1)
        label = data[1]
        img = Variable(img)
        label = Variable(label)
        out = model(img)
        loss = criterion(out, label)
        print_loss = loss.data.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print('epoch: {}, 损失: {:.4}'.format(i, loss.data.item()))

# 模型评估
model.eval()
for data in test_loader:
    img = data
    img = Variable(img)
    if torch.cuda.is_available():
        img = img.cuda()
    out = model(img)
    for i in out:
        print(zh(i.argmax().item()))
```

设计实验改进网络并论证

正则化技术，drophard，batch normalization

面试-跑代码，提问代码，神经网络

Batch

引入Batch Size

未引入Batch Size

```
epoch: 1, loss: 3.213
epoch: 2, loss: 0.0001621
epoch: 3, loss: 9.537e-07
epoch: 4, loss: 3.588e-05
epoch: 5, loss: 0.0
epoch: 6, loss: 0.0
epoch: 7, loss: 0.0003221
```

Batch Size = 64

```
epoch: 1, loss: 1.387
epoch: 2, loss: 0.2262
epoch: 3, loss: 0.0488
epoch: 4, loss: 0.09297
epoch: 5, loss: 0.08952
epoch: 6, loss: 0.02063
epoch: 7, loss: 0.1221
epoch: 8, loss: 0.042
epoch: 9, loss: 0.01556
epoch: 10, loss: 0.07355
epoch: 11, loss: 0.02383
```

在实验中观察到，设置Batch Size = 64时，训练速度更快，耗时更短，这是因为减少了调整次数，计算非常耗时，同时神经网络常是非凸的，虽然loss下降非常快，但网络最终可能收敛到初始点附近的局部最优点。

适当Batch Size使得梯度下降方向更加准确，并且通过更多的epoch最后也能达到指定精度。

为什么说Batch size的增大能使网络的梯度更准确? 梯度的方差表示：
$Var(g)=Var(1m\Sigma mi=1g(xi,yi))=1m2Var(g(x1,y1)+g(x2,y2)+\cdots+g(xm,ym))$ 由于样本是随机选取的，满足独立同分布，所以所有样本具有相同的方差$Var(g(xi,yi))$ 所以上式可以简化成
$Var(g)=1mVar(g(xi,yi))$ 可以看出当Batch size为m时，样本的方差减少m倍，梯度就更准确了。

假如想要保持原来数据的梯度方差，可以增大学习率$lr\ 1mVar(lr*g(xi,yi))$,只要lr取$\sqrt{m}$，上式就变成
$Var(g(xi,yi))$ 这也说明batch size设置较大时，一般学习率要增大。但是lr的增大不是一开始就设置的很大，而是在训练过程中慢慢变大。

Batch Size = 128

```
epoch: 1, loss: 1.973
epoch: 2, loss: 1.183
epoch: 3, loss: 0.5793
epoch: 4, loss: 0.1457
epoch: 5, loss: 0.3091
epoch: 6, loss: 0.09631
epoch: 7, loss: 0.2889
epoch: 8, loss: 0.09457
epoch: 9, loss: 0.09937
epoch: 10, loss: 0.02096
epoch: 11, loss: 0.03208
```

在每个epoch，加入这行代码

```
        learning_rate += 0.001
```

可以看到也达到了类似的效果

使用不同的优化器

要使用torch.optim，你必须构造一个optimizer对象，这个对象能保存当前的参数状态并且基于计算梯度进行更新。

要构造一个优化器，你必须给他一个包含参数（必须都是variable对象）进行优化，然后可以指定optimizer的参数选项，比如学习率，权重衰减。

SGD

它的全称是stochastic gradient descent

当训练数据N很大时，计算总的cost function来求梯度代价很大，所以一个常用的方法是计算训练集中的小批量（minibatches），这就是SGD。

minibatch的大小是一个超参数，通常使用2的指数，是因为在实际中许多向量化操作实现的时候，如果输入数据量是2的倍数，那么运算更快。

SGD的缺点：在浅维度上进展缓慢；到local minima 或者 saddle point会导致gradient为0，无法移动。而事实上，saddle point 问题在高维问题中会更加常见。

```
optimizer = optim.SGD(model.parameters(),lr = 0.01, momentum = 0.9)
```

```
epoch: 1, loss: 1.124
epoch: 2, loss: 0.529
epoch: 3, loss: 0.3221
epoch: 4, loss: 0.0533
epoch: 5, loss: 0.1094
epoch: 6, loss: 0.02553
epoch: 7, loss: 0.1049
epoch: 8, loss: 0.01592
epoch: 9, loss: 0.167
Test Loss: 0.146000, Acc: 0.959366
```

Adam

Adam算法是两位科学家提出的随机优化方法

`Adam(Adaptive Moment Estimation)`本质上是带有动量项的`RMSprop`，它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。它的优点主要在于经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。

它的优点有

1、结合了Adagrad善于处理稀疏梯度和RMSprop善于处理非平稳目标的优点; 2、对内存需求较小; 3、为不同的参数计算不同的自适应学习率; 4、也适用于大多非凸优化-适用于大数据集和高维空间。

```
optimizer = optim.Adam(model.parameters(),lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)
```

可以看出，性能显著好于SGD优化器（训练速度更快，损失更小，正确率更高），最后决定选用此优化器

```
epoch: 1, loss: 0.04275
epoch: 2, loss: 0.03706
epoch: 3, loss: 0.0004642
epoch: 4, loss: 0.001192
epoch: 5, loss: 0.1436
epoch: 6, loss: 0.001139
epoch: 7, loss: 1.363e-05
epoch: 8, loss: 0.001549
epoch: 9, loss: 2.323e-05
Test Loss: 0.097276, Acc: 0.982094
```

使用ReLU函数

选择隐层神经元的输出激活函数时，可以选择sigmoid，可以选择ReLu

本实验选择ReLU函数

`sigmoid 的梯度消失问题，ReLU 的导数就不存在这样的问题`

对比sigmoid类函数主要变化是： 1）单侧抑制 2）相对宽阔的兴奋边界 3）稀疏激活性。

`ReLU和sigmoid函数gradient特性不同。sigmoid的gradient在饱和区域非常平缓，接近于0，很容易造成vanishing gradient的问题，减缓收敛速度。vanishing gradient在网络层数多的时候尤其明显，是加深网络结构的主要障碍之一。相反，Relu的gradient大多数情况下是常数，有助于解决深层网络的收敛问题。Relu的另一个优势是在生物上的合理性，它是单边的，相比sigmoid，更符合生物神经元的特征。 而提出sigmoid，主要是因为它们全程可导。还有表达区间问题，sigmoid区间是0到1，或着-1到1，在表达上，尤其是输出层的表达上有优势。`

为什么引入Relu呢？ 第一，采用sigmoid等函数，算激活函数时（指数运算），计算量大，反向传播求误差梯度时，求导涉及除法，计算量相对大，而采用Relu激活函数，整个过程的计算量节省很多。 第二，对于深层网络，sigmoid函数反向传播时，很容易就会出现梯度消失的情况（在sigmoid接近饱和区时，变换太缓慢，导数趋于0，这种情况会造成信息丢失），从而无法完成深层网络的训练。 第三，Relu会使一部分神经元的输出为0，这样就造成了网络的稀疏性，并且减少了参数的相互依存关系，缓解了过拟合问题的发生

```
epoch: 1, loss: 1.298
epoch: 2, loss: 0.3448
epoch: 3, loss: 0.27
epoch: 4, loss: 0.3229
epoch: 5, loss: 0.06962
epoch: 6, loss: 0.02071
epoch: 7, loss: 0.0324
epoch: 8, loss: 0.2552
epoch: 9, loss: 0.1716
epoch: 10, loss: 0.07788
epoch: 11, loss: 0.02734
```

使用dropout层

一种防止过拟合的方法,dropout,即丢弃某些神经元的输出.由于每次训练的过程里,丢弃掉哪些神经元的输出都是随机的,从而可以使得模型不过分依赖于某些神经元的输出,从而达到防止过拟合的目的

并不是简单地丢弃掉某些神经元的输出,对留下的输出,我们要改变他们的值,以保证丢弃前后的输出的期望不变

```
        self.FCLayer = nn.Sequential(
            nn.Linear(50 * 5 * 5, 1024),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2),
            nn.Linear(1024, 128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(128, 12)
        )
```

可以视作避免过拟合

```
epoch: 1, loss: 0.4028
epoch: 2, loss: 0.1407
epoch: 3, loss: 0.3145
epoch: 4, loss: 0.005723
epoch: 5, loss: 0.2504
epoch: 6, loss: 0.03534
epoch: 7, loss: 0.004302
epoch: 8, loss: 0.01858
epoch: 9, loss: 0.001253
Test Loss: 0.105601, Acc: 0.978650
```

数据强化

通过cv框架增强训练数据

```
def rotate_whole_img(img):
    """
       rotate noise
       rotate angle is 0 - 20
    """
    angle = np.random.randint(0, 10)
    h, w = img.shape[:2]
    center = (w / 2, h / 2)
    M = cv2.getRotationMatrix2D(center, angle, 1)
    im = cv2.warpAffine(img, M, (w, h), borderValue=(255, 255, 255))
    return im

root = './ttest'
root1 = './tttest'
imgs = os.listdir(root)
print(imgs)
# 读取图像
for i in imgs:
    img = cv2.imread(root + "/" + i)
    im_rotate = rotate_whole_img(img)
    cv2.imwrite(root1 + "/" + i,im_rotate)
```

如倾斜训练集中的图片，加强训练样本的多样性。

正确率上升0.2%（平均）

对网络设计的理解

简介

Convolutional Neural Network(卷积神经网络，CNN)

识别图片：提取每一个pixel形成向量，最后在每一个类别维度上输出结果。每一层的神经元侦测某种pattern，侦测到即被激活，hidden layer中越后的层次侦测的pattern越复杂

如100x100的图片，它需要，100x100x3维的向量来描述，假设第一层有1000个神经元，那就需要30000x1000个参数

CNN：简化神经网络的架构，过滤掉某些不需要的权重，不使用fully connected network

根据

过滤参数

pattern小于图片，有以下特征

1. 对neuron，**不需要**看整张image来识别pattern。它只需要连接到小区域
2. pattern出现在image的不同位置，**不需要**训练不同的detector来侦测不同方位的同一种pattern，这两个detector可以共享参数
3. 做**下采样**，不影响识别，可以这样减小image

流程

图片-convolution（特征1，2处理）-max pooling（特征3处理）-(……)-flatten-fully connected feedforward network

Convolution

Convolution Layer中，有一组Filter

每一个filter，等同于neuron，本质是matrix

matrix中每一个值，就是network的parameter，类似于weight和bias，必须需要学习出来。matrix的大小，决定侦测范围的大小。

先将matrix放在左上角，做内积inner product（对应位置元素相乘求总和），然后进行挪动（事先决定距离stride）

如6x6的矩阵，用3x3的filter，最后转化为内积的4x4的matrix

filter是主对角线上全是1，其它都是-1，那么就是侦测有没有主对角线都是1的块，如果内积为3，说明pattern符合

所有filter处理的结果集合为feature map

Colorful Image

彩色是RGB组成的，是一个立方体

filter也应该选用立方体

Versus

convolution是fully connected去掉一些权重的结果

相当于矩阵拉直成向量以后，只有部分输入有权重的结果(9/36)，达到看部分输入图的效果，用比较少的参数。并且用同一个filter的neuron的weight是一样的（连接的输入不同）

Max Pooling

用不同的filter得到不同的matrix result

在每一个matrix result中，分组，4个取1个value（最大值、平均值……）

这样可以让image缩小，成为原来的1/4

最后6x6通过convolution和max pooling，得到2x2，每个pixel的深度取决于filter的个数

Flatten

将堆叠的feature map拉直，变成向量，放入fully connected神经网络

PyTorch使用

PyTorch简介

- 支持GPU，灵活，支持动态神经网络
- 底层代码易于理解
- 命令式体验
- 自定义扩展

能用矩阵就用矩阵，不要用循环（速度非常慢）

Tensors（张量）

Tensors类似numpy的ndarrays

```
from __future__ import print_function
import torch
```

构造一个5x3矩阵，不初始化

```
x = torch.empty(5,3)
```

构造一个随机初始化的矩阵

```
x = torch.rand(5,3)
```

构造一个全0且数据类型是long的矩阵

```
x = torch.zeros(5,3,dtype=torch.long)
```

数据定义张量

```
x = torch.tensor([5.5,3])
```

基于tensor创建tensor

```
x = x.new_ones(5,3,dtype=torch.double)
#5*3的张量，元素都是1，device和dtype属性继承自另一个张量x
x = torch.randn_like(x,dtype=torch.float)
#继承尺寸，改变dtype
```

获取维度

```
print(x.size())
```

加法

```
x = torch.rand(5,3)
y = torch.rand(5,3)
result = torch.empty(5,3)
x + y
torch.add(x,y,out=result)
y.add_(x)    #in-place加法，x加给y
```

改变大小

想改变一个 tensor 的大小或者形状，你可以使用 torch.view

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8) # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
>torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

如果你有一个元素 tensor ，使用 .item() 来获得这个 value 。

```
x = torch.randn(1)
print(x)
print(x.item())
tensor([ 0.9422])
0.9422121644020081
```

PyTorch自动微分

创建一个张量，设置 requires_grad=True 来跟踪与它相关的计算

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
```

输出：

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

针对张量做一个操作

```
y = x + 2
print(y)
#输出
tensor([[3., 3.],
[3., 3.]], grad_fn=<AddBackward0>)
```

y 作为操作的结果被创建，所以它有 grad_fn(指向Function对象，用于反向传播的梯度计算之用)

```
print(y.grad_fn)
#输出
<AddBackward0 object at 0x7fe1db427470>
```

针对 y 做更多的操作：

```
z = y * y * 3
out = z.mean()  #求平均值
print(z, out)
#输出
tensor([[27., 27.],
[27., 27.]], grad_fn=<MulBackward0>)
tensor(27., grad_fn=<MeanBackward0>)
```

.requires_grad_(...) 会改变张量的 requires_grad 标记。输入的标记默认为 False ， 如果没有提供 相应的参数。

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)
#输出
False
True
<SumBackward0 object at 0x7fe1db427dd8>
```

梯度

现在后向传播，因为输出包含了一个标量，out.backward() 等同于 out.backward(torch.tensor(1.))。

```
out.backward()
#打印梯度 d(out)/dx
print(x.grad)
#输出
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

PyTorch神经网络

神经网络

用torch.nn可以建立神经网络

现在对于自动梯度(autograd)有一些了解，神经网络是基于自动梯度 (autograd)来定义一些模型。一个 nn.Module 包括层和一个方法 forward(input) 它会返回输出(output)。

一个典型的神经网络训练过程包括以下几点：

1.定义一个包含可训练参数的神经网络

2.迭代整个输入

3.通过神经网络处理输入

4.计算损失(loss)

5.反向传播梯度到神经网络的参数

6.更新网络的参数，典型的用一个简单的更新方法：weight = weight - learning_rate *gradient

定义神经网络

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
net = Net()
print(net)
```

输出

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

你刚定义了一个前馈函数，然后反向传播函数被自动通过 autograd 定义了。你可以使用任何张量操作在前馈函数上。

一个模型可训练的参数可以通过调用 net.parameters() 返回：

```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight
```

让我们尝试随机生成一个 32x32 的输入。注意：期望的输入维度是 32x32 。为了使用这个网络在 MNIST 数据及上，你需要把数据集中的图片维度修改为 32x32。

```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

输出

```
tensor([[-0.0233,  0.0159, -0.0249,  0.1413,  0.0663,  0.0297, -0.0940, -0.0135,
          0.1003, -0.0559]], grad_fn=<AddmmBackward>)
```

把所有参数梯度缓存器置零，用随机的梯度来反向传播

```
net.zero_grad()
out.backward(torch.randn(1, 10))
```

只支持小批量，不支持单个样本，如果只有一个样本，请使用`input.unsqueeze(0)`来添加假维度

回顾：

- `torch.Tensor` -支持自动微分操作（如backward（））的多维数组。也包含了张量的梯度。
- `nn.Module` - 神经网络模块。方便的封装参数的方式，有助手将其移动到GPU，导出，加载等。
- `nn.Parameter` - 参数-一种张量，当作为属性分配给模块时，它会自动注册为参数。
- `autograd.Function` -实现自动微分操作的前向和后向定义。每个Tensor操作都至少创建一个函数节点，该节点连接到创建张量的函数并对其历史进行编码。

在此，我们完成了：

1.定义一个神经网络

2.处理输入以及调用反向传播

还剩下：

1.计算损失值

2.更新网络中的权重

损失函数

一个损失函数需要一对输入：模型输出和目标，然后计算一个值来评估输出距离目标有多远。

有一些不同的损失函数在 nn 包中。一个简单的损失函数就是 nn.MSELoss ，这计算了均方误差。

例如：

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()
loss = criterion(output, target)
print(loss)
```

输出：

```
tensor(1.3389, grad_fn=<MseLossBackward>)
```

现在，如果你跟随损失到反向传播路径，可以使用它的 .grad_fn 属性，你将会看到一个这样的计算图：

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

所以，当我们调用 loss.backward()，整个图都会微分，而且所有的在图中的requires_grad=True 的张量将会让他们的 grad 张量累计梯度。

为了演示，我们将跟随以下步骤来反向传播。

```
print(loss.grad_fn)  # MSELoss
print(loss.grad_fn.next_functions[0][0])  # Linear
print(loss.grad_fn.next_functions[0][0].next_functions[0][0])  # ReLU
```

反向传播

为了实现反向传播损失，我们所有需要做的事情仅仅是使用 loss.backward()。你需要清空现存的梯度，要不然梯度将会和现存的梯度累计到一起。

现在我们调用 loss.backward()，然后看一下 con1 的偏置项在反向传播之前和之后的变化。

```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

输出：

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([-0.0054,  0.0011,  0.0012,  0.0148, -0.0186,  0.0087])
```

更新神经网络参数

最简单的更新规则就是随机梯度下降。

```
weight = weight - learning_rate * gradient
```

我们可以使用 python 来实现这个规则：

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

尽管如此，如果你是用神经网络，你想使用不同的更新规则，类似于 SGD, Nesterov-SGD, Adam, RMSProp, 等。为了让这可行，我们建立了一个小包：torch.optim 实现了所有的方法。使用它非常的简单。

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```