下面是使用PyTorch实现的简单卷积神经网络（CNN）的例子，这个网络用于在MNIST数据集上进行手写数字识别。这个网络包含两个卷积层，每个卷积层后面都跟着一个ReLU激活函数和一个最大池化层。然后是三个全连接层，前两个全连接层后面都跟着一个ReLU激活函数。网络的输入是1通道的28x28像素的图像，输出是10个类别的概率分布。在训练过程中，使用交叉熵损失函数和随机梯度下降优化器。遍历数据集两次，每次遍历数据集，都会计算网络的输出，计算损失，然后使用优化器更新网络的权重。

以下是对代码的详细说明：

1. 导入所需的库：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

这些库包括PyTorch（用于构建和训练神经网络），torchvision（用于处理图像数据）和transforms（用于进行图像预处理）。

2. 定义网络结构：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)  # 输入通道1，输出通道6，卷积核大小5
        self.pool = nn.MaxPool2d(2, 2)  # 最大池化，核大小2，步长2
        self.conv2 = nn.Conv2d(6, 16, 5)  # 输入通道6，输出通道16，卷积核大小5
        self.fc1 = nn.Linear(16 * 4 * 4, 120)  # 全连接层，输入维度16*4*4，输出维度120
        self.fc2 = nn.Linear(120, 84)  # 全连接层，输入维度120，输出维度84
        self.fc3 = nn.Linear(84, 10)  # 全连接层，输入维度84，输出维度10

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # 第一层卷积 -> ReLU -> 池化
        x = self.pool(F.relu(self.conv2(x)))  # 第二层卷积 -> ReLU -> 池化
        x = x.view(-1, 16 * 4 * 4)  # 展平操作，准备全连接
        x = F.relu(self.fc1(x))  # 第一层全连接 -> ReLU
        x = F.relu(self.fc2(x))  # 第二层全连接 -> ReLU
        x = self.fc3(x)  # 第三层全连接
        return x
```

这个类定义了一个简单的卷积神经网络，包含两个卷积层，两个最大池化层和三个全连接层。

3. 加载数据集：

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,),
(0.5,))])
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True,
num_workers=2)
```

这部分代码首先定义了一个图像预处理流程，然后加载了MNIST数据集，并将其封装成一个数据加载器，用于在训练过程中批量获取数据。

4. 初始化网络和优化器：

```
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

这部分代码初始化了我们定义的网络，定义了损失函数（交叉熵损失），并初始化了优化器（随机梯度下降）。

5. 训练网络：

```
for epoch in range(2):  # 多次遍历数据集
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 2000 == 1999:  # 每2000个批次打印一次
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

这部分代码是训练过程的主循环，每次遍历数据集，计算网络的输出和损失，然后反向传播误差并更新网络的权重。

这个简单的CNN模型是一个基础的深度学习模型，可以作为进一步学习和实验的基础。