用PyTorch训练神经网络

首先，类似于第2章，本章会先用PyTorch的求梯度搭建和训练神经网络。而不过多使用PyTorch的顶层功能。原因在于，只有多看多写相对底层的功能，才能在以后对网络的修改时游刃有余。

随后，我们逐步用PyTorch自动功能代替自己写的代码，最终呈现出简洁的形式。

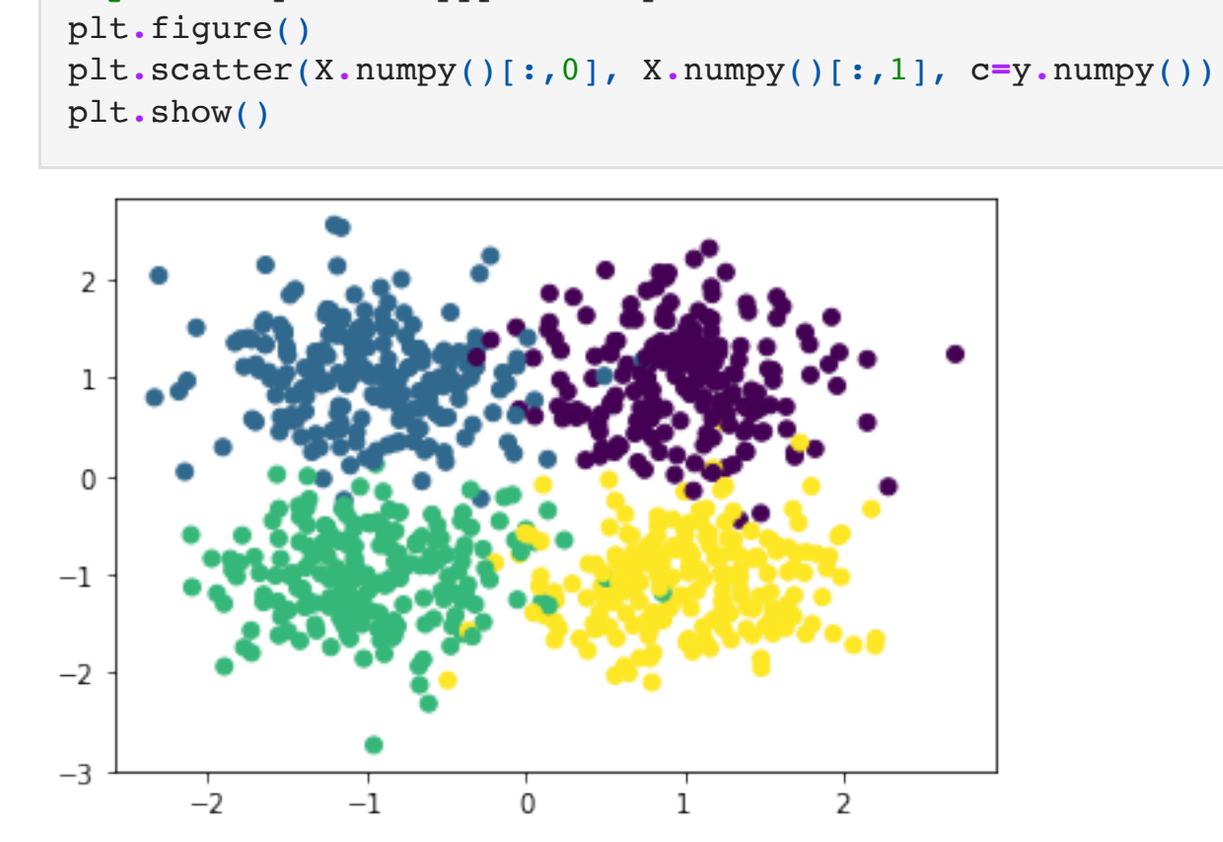
```python
import torch
```

准备数据

在训练神经网络之前，我们先准备好和上一章一样的数据（由于torch和numpy生成的随机数不同，点的具体位置可能发生改变）：

```python
N = 200
var = 0.5

torch.manual_seed(0)
x1 = torch.randn(N, 1) * var + 1
y1 = torch.randn(N, 1) * var + 1
l1 = torch.randn(N, 1) * 0 + 0
x3 = torch.randn(N, 1) * var - 1
y3 = torch.randn(N, 1) * var - 1
l3 = torch.randn(N, 1) * 0 + 1
x4 = torch.randn(N, 1) * var - 1
y4 = torch.randn(N, 1) * var - 1
l4 = torch.randn(N, 1) * 0 + 2
x2 = torch.randn(N, 1) * var + 1
y2 = torch.randn(N, 1) * var - 1
l2 = torch.randn(N, 1) * 0 + 3
X1 = torch.cat((x1, x2, x3, x4), dim=0)
X2 = torch.cat((y1, y2, y3, y4), dim=0)
X = torch.cat((X1, X2), dim=1)
y = torch.cat((l1, l2, l3, l4), dim=0)

idx = torch.randperm(X.shape[0])
X = X[idx,:]
y = y[idx,:].flatten()

# one hot 编码
def OneHot(y, K):
    E = y.flatten().shape[0]
    Y = torch.zeros((E, K))
    for i in range(E):
        Y[i, int(y.flatten()[i])] = 1
    return Y
Y = OneHot(y, 4)
```

```python
import matplotlib.pyplot as plt
plt.figure()
plt.scatter(X.numpy()[:,0], X.numpy()[:,1], c=y.numpy())
plt.show()
```



只用PyTorch求梯度

搭建网络

从第2章我们可以看出，我们只需要定义前向传播的函数，PyTorch就可以自动实现梯度的计算，那么让我们开始：

我们先定义一些基本的参数（和上一章一样）：

```python
# 学习率
alpha = 0.5

# 生成随机的模型参数
torch.manual_seed(0)
W1 = torch.nn.Parameter(torch.randn(2, 3) * 0.01, requires_grad=True)
b1 = torch.nn.Parameter(torch.randn(1, 3) * 0.01, requires_grad=True)
W2 = torch.nn.Parameter(torch.randn(3, 3) * 0.01, requires_grad=True)
b2 = torch.nn.Parameter(torch.randn(1, 3) * 0.01, requires_grad=True)
W3 = torch.nn.Parameter(torch.randn(3, 4) * 0.01, requires_grad=True)
b3 = torch.nn.Parameter(torch.randn(1, 4) * 0.01, requires_grad=True)
```

只定义前向传播

```python
def forward(X, W1, W2, W3, b1, b2, b3):
    A0 = X

    Z1 = torch.matmul(A0, W1) + b1
    A1 = torch.sigmoid(Z1)

    Z2 = torch.matmul(A1, W2) + b2
    A2 = torch.sigmoid(Z2)

    Z3 = torch.matmul(A2, W3) + b3
    A3 = torch.sigmoid(Z3)

    return A3
```

定义损失函数

```python
def loss(pred, target):
    E = pred.shape[0]
    return ((-target*torch.log(pred)-(1-target)*torch.log(1-pred)).sum()).sum()/E
```

至此，神经网络已经搭建完毕。

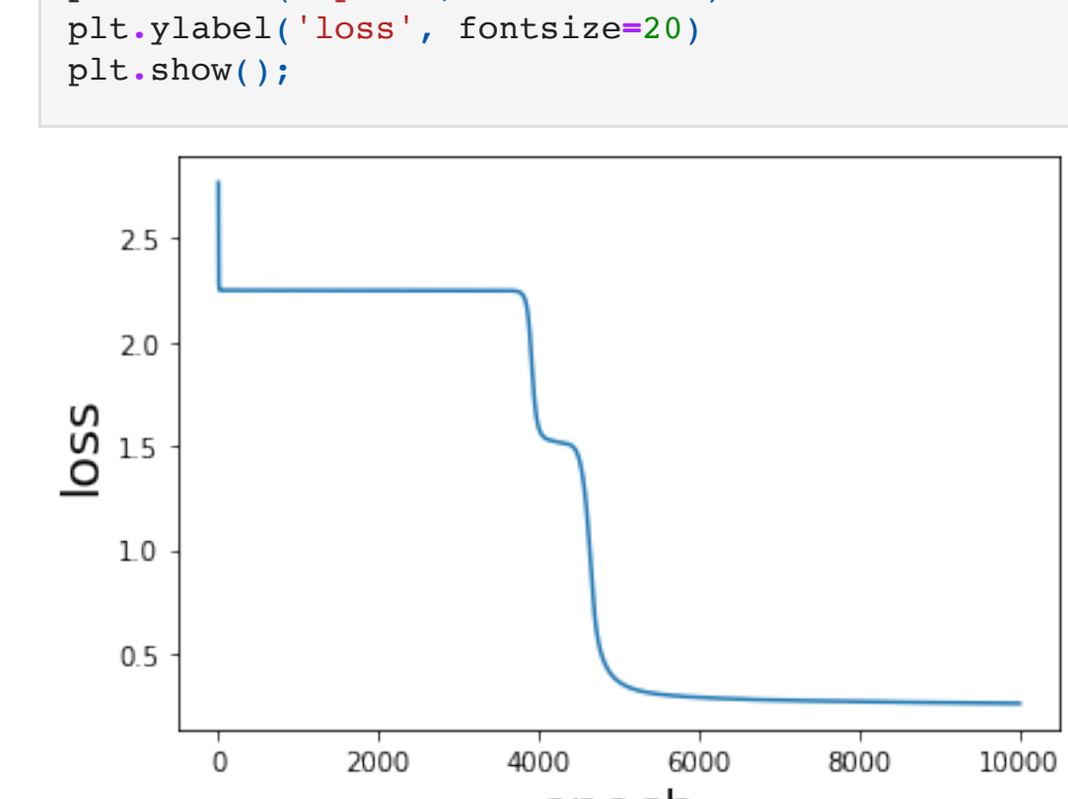
下面我们就可以开始训练了。

训练神经网络

现在就可以开始训练了！

```python
Loss = []
for i in range(10000):
    pred = forward(X, W1, W2, W3, b1, b2, b3)
    L = loss(pred, Y)

    # 反向传播
    L.backward()
    # 记录下loss
    Loss.append(L.item())

    # 参数更新
    W1.data = W1.data - alpha * W1.grad
    W2.data = W2.data - alpha * W2.grad
    W3.data = W3.data - alpha * W3.grad
    b1.data = b1.data - alpha * b1.grad
    b2.data = b2.data - alpha * b2.grad
    b3.data = b3.data - alpha * b3.grad

    # 清除之前的梯度
    W1.grad.zero_()
    W2.grad.zero_()
    W3.grad.zero_()
    b1.grad.zero_()
    b2.grad.zero_()
    b3.grad.zero_()
```

```python
plt.figure()
plt.plot(Loss)
plt.xlabel('epoch', fontsize=20)
plt.ylabel('loss', fontsize=20)
plt.show();
```



看一下准确率：

```python
yhat = forward(X, W1, W2, W3, b1, b2, b3)
(torch.argmax(yhat, dim=1) == y).sum() / y.numel()
```

```
tensor(0.9550)
```

没有达到100%，这是由于在这个实验里，数据点的有很大的噪音。有很多点已经跑到另一类的中间了，机器学习很难把它们挑出来。实际上，在这个实验中，他们也不应该被挑出来，因为他们的错误位置可以理解为测量误差造成的。也就是说，按照我们的设计，所有第一象限的点本来就应该分类为第0类，第二象限第1类，第三象限第2类，第四象限第3类。如果一个点明明在另一个象限却正确分类，反而说明机器学习的结果是"错"的，这个现象其实被称为**过度拟合**。以后会讲到。

逐步引入PyTorch自带功能

接下来，我们会逐渐用PyTorch自带功能替换刚才自己写的代码，逐渐简化代码的书写。

optimizer

`torch.optim` 提供了大量的基于梯度的优化算法[1]，例如Adagrad, Adam, RMSprop, SGD。以后会介绍某些重要的算法。

要使用pytorch的optimizer，我们需要定义3个东西：

- 优化算法，也就是常提到的Adam, SGD等。这里我们以SGD为例
- 需要被优化等参数，也就是本例子中的 `W` 和 `b`
- 学习率（这里我们采取固定的学习率）

[1] https://pytorch.org/docs/stable/optim.html

下面就是针对本例子的定义

`optimizer = torch.optim.SGD([W1,W2,W3,b1,b2,b3], lr=0.5)`

optimizer带来的好处就是可以轻松的清除梯度以及更新参数。让我们来看一下引入optimizer之后的全部代码：

前面的代码：要注意的是，为了提升训练的效果，我们会在输出层加一个归一化的函数，让所有的输出都在[0,1]直接，甚至让每一行的和等于1。这样就相当于网络对每一类的输出的和等于1，这就符合了概率公理

- 非负性
- 归一化
- 可加性

因此，每一个输出都可以**被当作概率来看待**。要注意的是，通过神经网络的计算我们可以看出，这里的输出和所谓的概率没有任何关系，至少从意义上来说是这样的。但是由于其数学特点，我们可以把它们假象称概率，并且利用概率的方法来解读/分析网络以及结果。由此可以引申出一个新的神经网络的领域：贝叶斯神经网络和概率学习。

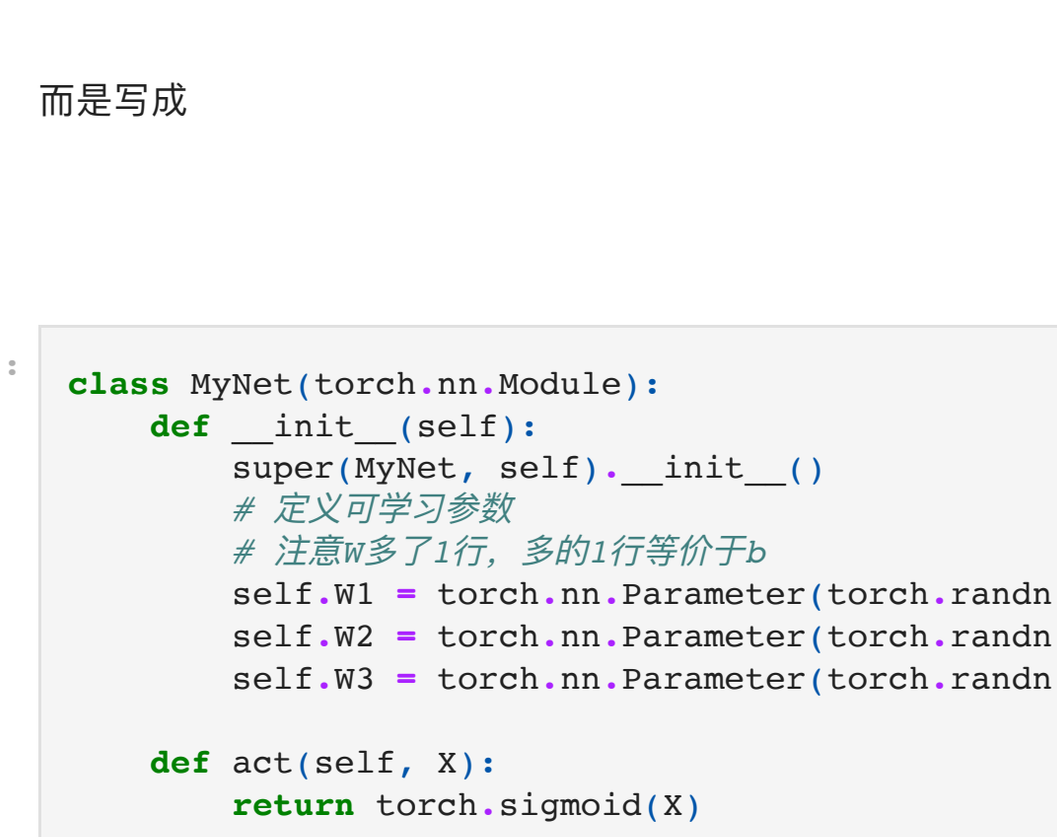
```python
torch.manual_seed(0)
W1 = torch.nn.Parameter(torch.randn(2, 3) * 0.01, requires_grad=True)
b1 = torch.nn.Parameter(torch.randn(1, 3) * 0.01, requires_grad=True)
W2 = torch.nn.Parameter(torch.randn(3, 3) * 0.01, requires_grad=True)
b2 = torch.nn.Parameter(torch.randn(1, 3) * 0.01, requires_grad=True)
W3 = torch.nn.Parameter(torch.randn(3, 4) * 0.01, requires_grad=True)
b3 = torch.nn.Parameter(torch.randn(1, 4) * 0.01, requires_grad=True)

def forward(X, W1, W2, W3, b1, b2, b3):
    A0 = X
    Z1 = torch.matmul(A0, W1) + b1
    A1 = torch.sigmoid(Z1)
    Z2 = torch.matmul(A1, W2) + b2
    A2 = torch.sigmoid(Z2)
    Z3 = torch.matmul(A2, W3) + b3
    A3 = torch.sigmoid(Z3)
    return A3
```

引入optimizer之后的部分

```python
# 定义optimizer
optimizer = torch.optim.SGD([W1,W2,W3,b1,b2,b3], lr=alpha)

Loss = []
for i in range(10000):
    pred = forward(X, W1, W2, W3, b1, b2, b3)
    L = loss(pred, Y)

    # 清除梯度
    optimizer.zero_grad()
    # 反向传播
    L.backward()
    # 记录下loss
    Loss.append(L.item())
    # 参数更新
    optimizer.step()
```

可以看出，参数更新和清除梯度的代码简化了许多。下面我们会看到结果和之前是一样的：

```python
plt.figure()
plt.plot(Loss)
plt.xlabel('epoch', fontsize=20)
plt.ylabel('loss', fontsize=20)
plt.show();
```



引入class

我们再定义神经网络的部分，我们首先定义了 `W` 和 `b` ，然后在每次forward里面都要把这些参数放进函数里面用于计算。（当然也可以不放进去，而是直接在函数里面调用 `W` 和 `b` 当成全局变量，但是看从代码层面是不严谨的，而且代码变多之后会命名混乱等问题）。

如果我们能把 `W` 和 `b` 直接存在forward函数里的话，就方便多了。我们可以通过类（class）来实现。而且我们不需要自己定义class，因为pytorch已经提供了一个类。

为了进一步简化，我们不再使用

$$X\cdot W+b$$

而是写成

$$\begin{bmatrix}X & \mathbf{1}\end{bmatrix}\cdot\begin{bmatrix}W\\ b\end{bmatrix}$$

```python
class MyNet(torch.nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        # 定义可学习参数
        # 注意W多了1行，多的1行等价于b
        self.W1 = torch.nn.Parameter(torch.randn(2+1, 3) * 0.01, requires_grad=True)
        self.W2 = torch.nn.Parameter(torch.randn(3+1, 3) * 0.01, requires_grad=True)
        self.W3 = torch.nn.Parameter(torch.randn(3+1, 4) * 0.01, requires_grad=True)

    def act(self, X):
        return torch.sigmoid(X)

    def forward(self, X):
        # 扩展出一列1
        A0 = torch.hstack([X, torch.ones(X.shape[0],1)])

        Z1 = torch.matmul(A0, self.W1)
        A1 = self.act(Z1)
        A1 = torch.hstack([A1, torch.ones(A1.shape[0],1)])

        Z2 = torch.matmul(A1, self.W2)
        A2 = self.act(Z2)
        A2 = torch.hstack([A2, torch.ones(A2.shape[0],1)])

        Z3 = torch.matmul(A2, self.W3)
        A3 = self.act(Z3)
        return A3
```

要注意的是，这个类里可以自己随意定义函数，随意添加任何计算所需的内容。但是必须用 `forward` 来定义前向传播的函数，PyTorch会自动识别这个函数为前向传播的内容。举例：

```python
# 定义一个实例
net = MyNet()
# 使用前向传播
prediction = net(X)
```

这里我们可以发现，我们使用 `net` 的时候，并没有用 `net.forward(X)`，而是直接用的 `net(X)`，也就是说PyTorch自动调用了 `net` 中的 `forward` 方法。所以不能定义成别的名字。但是其他的函数就无所谓。

除此之外， `torch.nn.Module` 还提供一个方法来提取所有的可学习参数： `net.parameters()`，利用这个函数，我们可以轻而易举的提取整个类里面的可学习参数，然后放进optimizer里。

让我们来看一下引入 `torch.nn.Module` 类之后的代码：

```python
torch.manual_seed(0)

# 生成网络
net = MyNet()

# 定义optimizer
optimizer = torch.optim.SGD(net.parameters(), lr=alpha)

# 训练网络
Loss = []
for i in range(10000):
    pred = net(X)
    L = loss(pred, Y)

    optimizer.zero_grad()
    L.backward()
    Loss.append(L.item())
    optimizer.step()
```

就这么简单，让我们来看一下训练过程和结果

```python
plt.figure()
plt.plot(Loss)
plt.xlabel('epoch', fontsize=20)
plt.ylabel('loss', fontsize=20)
plt.show();
```

可以看出，结果和之前相似。之所以不完全相同，是因为生成的初始参数不同了。因为我们把 `W` 和 `b` 合起来了，所以生成的随机值并不对应相同。

可复用性

可复用性是好代码的重要评判标准。在上面的代码中，我们发现，其实每一层的运算是类似的，但是我们重复的写了，这是愚蠢的。为了解决这个问题，我们可以以层为单元来搭建这个网络，我们仍然有 `torch.nn.Module`：

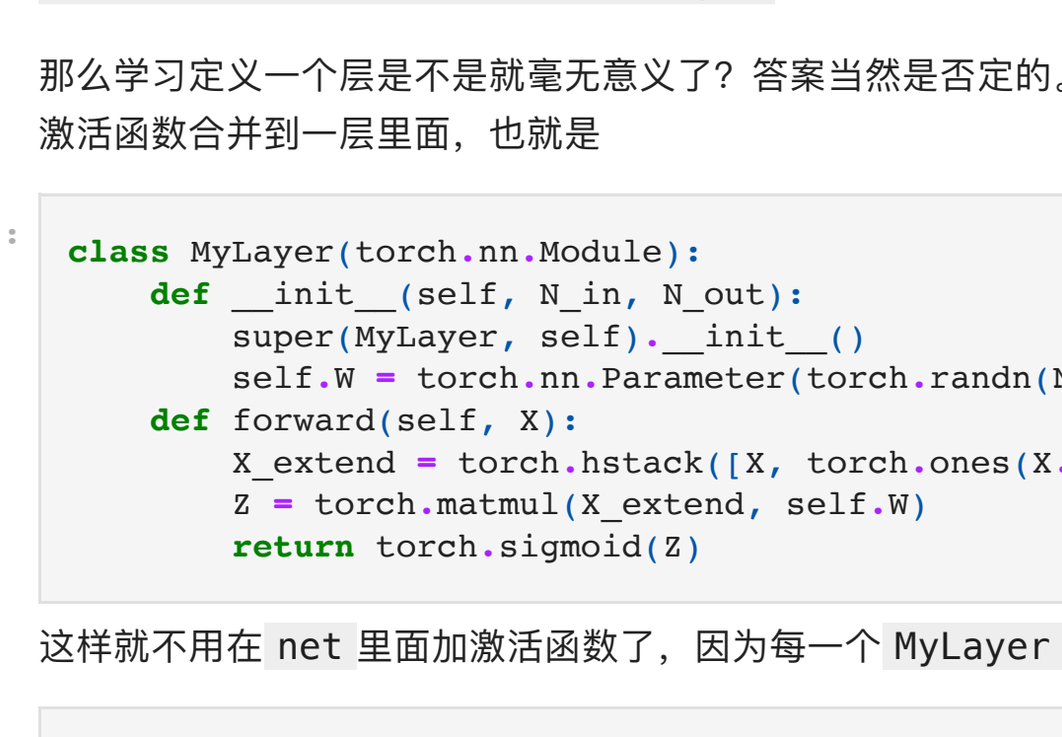
```python
class weightedsum(torch.nn.Module):
    def __init__(self, N_in, N_out):
        super(weightedsum, self).__init__()
        self.W = torch.nn.Parameter(torch.randn(N_in+1, N_out)*0.01, requires_grad=True)
    def forward(self, X):
        X_extend = torch.hstack([X, torch.ones(X.shape[0],1)])
        Z = torch.matmul(X_extend, self.W)
        return Z
```

注意这里的初始化时需要给入输入和输出的数量。这样，我们只需要生成许多个层，然后拼起来即可。

```python
class MyNet(torch.nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.MAC1 = weightedsum(2,3)
        self.MAC2 = weightedsum(3,3)
        self.MAC3 = weightedsum(3,4)

    def act(self, X):
        return torch.sigmoid(X)

    def forward(self, X):
        Z1 = self.MAC1(X)
        A1 = self.act(Z1)
        Z2 = self.MAC2(A1)
        A2 = self.act(Z2)
        Z3 = self.MAC3(A2)
        A3 = self.act(Z3)
        return A3
```

可以看出，代码的可读性强多了。那么我们来测试一下：

```python
torch.manual_seed(0)
net = MyNet()

optimizer = torch.optim.SGD(net.parameters(), lr=alpha)

Loss = []
for i in range(10000):
    pred = net(X)
    L = loss(pred, Y)

    optimizer.zero_grad()
    L.backward()
    Loss.append(L.item())
    optimizer.step()

plt.figure()
plt.plot(Loss)
plt.xlabel('epoch', fontsize=20)
plt.ylabel('loss', fontsize=20)
plt.show();
```



讨论： 这里我们自己搭建的层只有加权求和的功能，其实PyTorch已经提供了这个功能，也就是说，我们不需要自己定义 `weightedsum`，然后再使用它：

`self.MAC1 = weightedsum(2,3)`

而可以直接使用

`self.MAC1 = torch.nn.Linear(2,3)`

那么学习定义一个层是不是就毫无意义了？答案当然是否定的。当你能够自己写它的时候，你就可以自己任意的更改。比如我们可以把加权求和和激活函数合并到一层里面，也就是

```python
class MyLayer(torch.nn.Module):
    def __init__(self, N_in, N_out):
        super(MyLayer, self).__init__()
        self.W = torch.nn.Parameter(torch.randn(N_in+1, N_out)*0.01, requires_grad=True)
    def forward(self, X):
        X_extend = torch.hstack([X, torch.ones(X.shape[0],1)])
        Z = torch.matmul(X_extend, self.W)
        return torch.sigmoid(Z)
```

这样就不用在 `net` 里面加激活函数了，因为每一个 `MyLayer` 已经包含了激活函数，所以 `net` 可以这样写：

```python
class MyNet(torch.nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.layer1 = MyLayer(2,3)
        self.layer2 = MyLayer(3,3)
        self.layer3 = MyLayer(3,4)

    def forward(self, X):
        A1 = self.layer1(X)
        A2 = self.layer2(A1)
        A3 = self.layer3(A2)
        return torch.nn.functional.softmax(A3, dim=1)
```

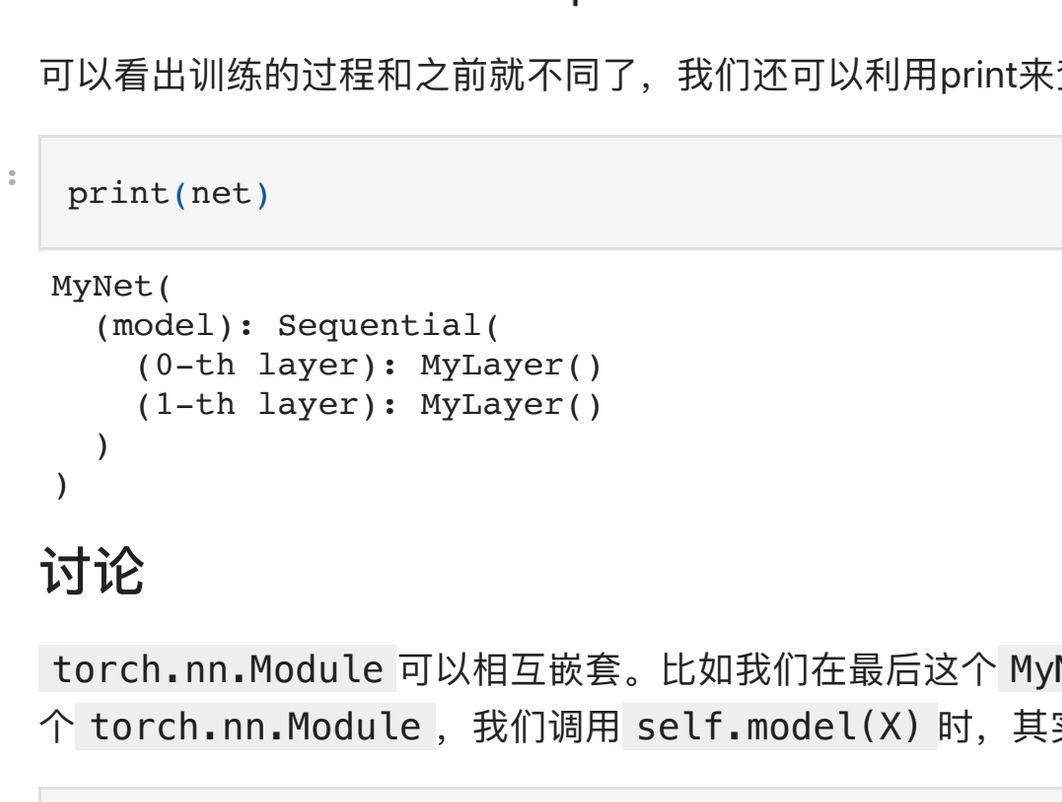
代码的灵活性

可以看出，比起第5章，代码已经有了天壤之别。但是我们的代码仍然需要一些灵活性，比如现在的网络是一个2-3-3-4的网络，但是我想搭建一个2-3-4-3-4-5-3-6-3-5-4-5-3-4-5-4-5-4-4-5-4的网络，我们不可能手动定义 `self.layer1` ……一直到 `self.layer100` 。这时候我们就需要自动的把这些层生成，然后找一个像列表一样的东西，把它们放在一起，这就是 `torch.nn.Sequential` 。下面让我们来看一下

```python
class MyNet(torch.nn.Module):
    def __init__(self, topology):
        super(MyNet, self).__init__()
        self.model = torch.nn.Sequential()
        for i in range(len(topology)-1):
            self.model.add_module(f'{i}-th layer', MyLayer(topology[i], topology[i+1]))

    def forward(self, X):
        return self.model(X)
```

可以看出，我们定义了一个 `Sequential` ，在循环里不断用 `.add_module` 往里面添加 `nn.Module` 。注意到我们之前定义的 `MyLayer` 就是继承于 `nn.Module` ，所以可以直接加进去。每一层的输入和输出从变量 `topology` 里依次读取。让我们来试一下：

```python
# 随便定义一个网络的结构
topology = [2,10,4]

torch.manual_seed(0)
net = MyNet(topology)

optimizer = torch.optim.SGD(net.parameters(), lr=alpha)

Loss = []
for i in range(10000):
    pred = net(X)
    L = loss(pred, Y)

    optimizer.zero_grad()
    L.backward()
    Loss.append(L.item())
    optimizer.step()

plt.figure()
plt.plot(Loss)
plt.xlabel('epoch', fontsize=20)
plt.ylabel('loss', fontsize=20)
plt.show();
```



可以看出训练的过程和之前并不同了，我们还可以利用print来查看一个网络

```python
print(net)
```

```
MyNet(
  (model): Sequential(
    (0-th layer): MyLayer()
    (1-th layer): MyLayer()
  )
)
```

讨论

`torch.nn.Module` 可以相互嵌套。比如我们在最后这个 `MyNet` 的forward里调用的是 `self.model(X)`，其实这里的 `self.model` 本来就是一个 `torch.nn.Module`，我们调用 `self.model(X)` 时，其实也就是调用了它内部的forward函数，看下面这个代码：

```python
net.model.forward(X)
```

```
tensor([[4.7462e-04, 9.9875e-01, 5.6821e-04, 1.9276e-08],
        [7.3738e-03, 9.9262e-01, 1.8904e-04, 2.3220e-07],
        [5.4361e-04, 9.9875e-01, 5.5858e-04, 2.5502e-08],
        ...,
        [3.7638e-03, 9.9574e-01, 2.6280e-04, 1.4781e-07],
        [3.6919e-04, 9.9897e-01, 6.8081e-04, 1.8694e-08],
        [3.9896e-04, 9.9890e-01, 6.2628e-04, 1.7238e-08]],
       grad_fn=<SigmoidBackward0>)
```

其实，之前的 `torch.nn.Linear`，`MyLayer`，`weightedsum`，`MyNet` 全都是 `torch.nn.Module` 的子类，他们都可以相互组合和嵌套。

如果我们仔细观察最后一个 `MyNet` ，我们可以发现，我们其实可以不需要 `MyNet` 来包裹它里面的 `self.model` ，因为 `MyNet` 除了 `self.model` 以外，没有任何其他东西，所以说这种情况下可以用 `Sequential` 定义一个模型：

```python
net = torch.nn.Sequential()
for i in range(len(topology)-1):
    net.add_module(f'{i}-th layer', MyLayer(topology[i], topology[i+1]))
net
```

```
Sequential(
  (0-th layer): MyLayer()
  (1-th layer): MyLayer()
)
```

或者我们不用 `MyLayer` ，而是每次添加 `torch.nn.Linear` 以及一个激活函数

```python
net = torch.nn.Sequential()
for i in range(len(topology)-1):
    net.add_module(f'{i}-th MAC', torch.nn.Linear(topology[i], topology[i+1]))
    net.add_module(f'{i}-th ACT', torch.nn.Sigmoid())
net
```

```
Sequential(
  (0-th MAC): Linear(in_features=2, out_features=10, bias=True)
  (0-th ACT): Sigmoid()
  (1-th MAC): Linear(in_features=10, out_features=4, bias=True)
  (1-th ACT): Sigmoid()
)
```

总结

这一节，我们从最基础的PyTorch功能逐步推进到一个完整的PyTorch神经网络，最后有介绍了 `Sequential` 用于增加网络搭建的灵活性。

在这里可以看出，搭建神经网络的方法多种多样