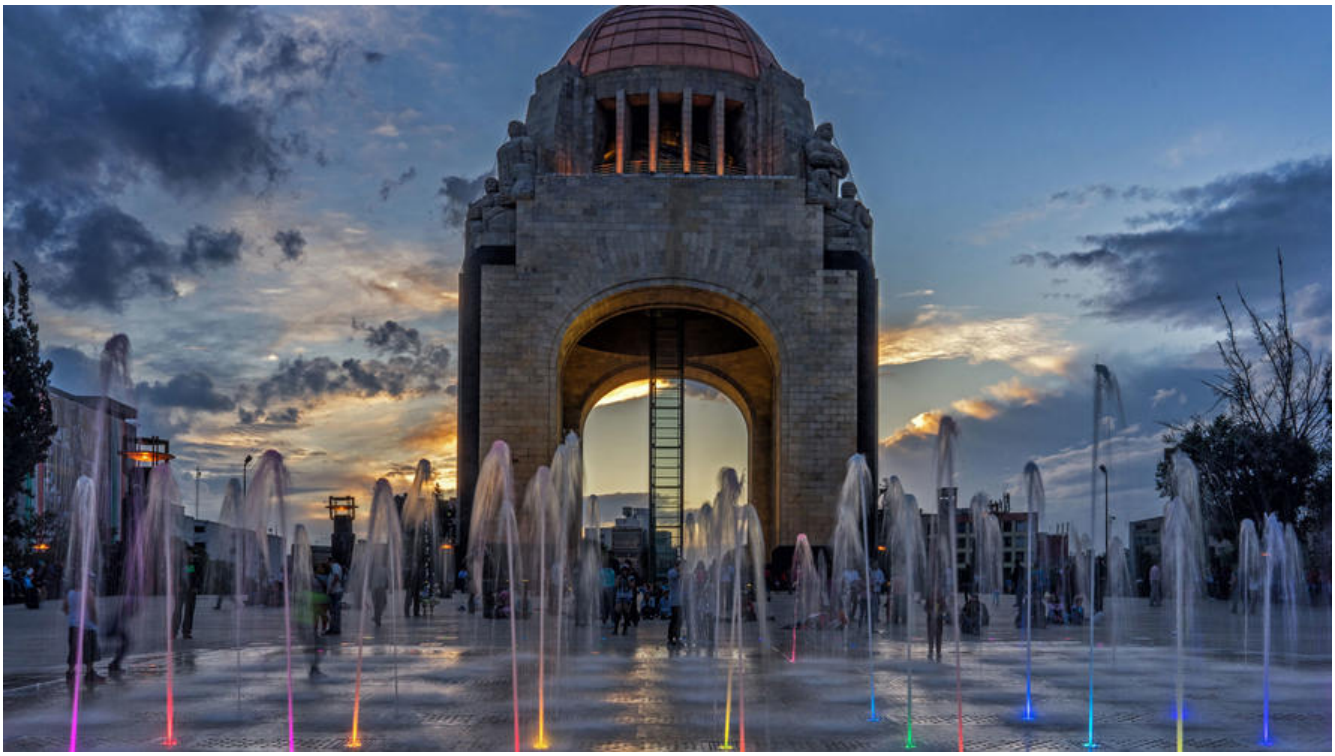
链滴

pytorch 入门笔记 -03- 神经网络

作者：zyk

原文链接：https://ld246.com/article/1639540087993

来源网站：链滴

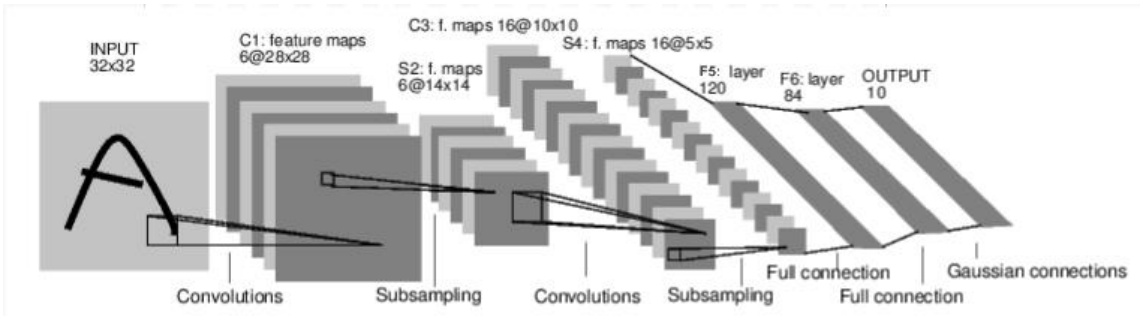


前言

本节主要内容是如何使用 torch.nn 包来构建神经网络。

上一讲已经讲过了 autograd，nn 包依赖 autograd 包来定义模型并求导。
一个 nn.Module 包含各个层和一个 forward(input) 方法，该方法返回 output。

例如：



它是一个简单的前馈神经网络，它接受一个输入，然后一层接着一层地传递，最后输出计算的结果。

神经网络的典型训练过程如下：

1. 定义包含一些可学习的参数(或者叫权重)神经网络模型；
2. 在数据集上迭代；
3. 通过神经网络处理输入；
4. 计算损失(输出结果和正确值的差值大小)；
5. 将梯度反向传播回网络的参数；
6. 更新网络的参数，主要使用如下简单的更新原则： weight = weight - learning_rate * gradient

定义网络

开始定义一个网络：

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self,):
        super(Net, self).__init__()
        # 输入图片通道数为 1，输出通道数为 6，卷积核大小为 (5, 5)
        self.conv1 = nn.Conv2d(1, 6, 5)
        # 输入图片通道数为 6，输出通道数为 16，卷积核大小为 (5, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # 最大池化层，池化层窗口大小为 (2, 2)
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # 改变数据的维度
        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:]
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

```
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)
)
```

在模型中必须要定义 forward 函数，backward 函数（用来计算梯度）会被 autograd 自动创建。

可以在 forward 函数中使用任何针对 Tensor 的操作。

net.parameters() 返回可被学习的参数（权重）列表和值

```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1 的权重

10
torch.Size([6, 1, 5, 5])
```

测试随机输入 32×32。

注：这个网络（LeNet）期望的输入大小是 32×32，如果使用 MNIST 数据集来训练这个网络，请把片大小重新调整到 32×32。

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

tensor([[ 0.0172,  0.1005, -0.1940, -0.0691, -0.0525, -0.0239, -0.0056, -0.0597,
0.0184, -0.0300]], grad_fn=<AddmmBackward0>)
```

将所有参数的梯度缓存清零，然后进行随机梯度的的反向传播：

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

note

torch.nn 只支持小批量输入。整个 torch.nn 包都只支持小批量样本，而不支持单个样本。

例如，nn.Conv2d 接受一个4维的张量，每一维分别是 sSamples * nChannels * Height * Width（本数 * 通道数 * 高 * 宽）。如果你有单个样本，只需使用 input.unsqueeze(0) 来添加其它的维数

在继续之前，我们回顾一下到目前为止用到的类。

回顾:

- torch.Tensor：一个用过自动调用backward() 实现支持自动梯度计算的多维数组，并且保存关于个向量的梯度 w.r.t.
- nn.Module：神经网络模块。封装参数、移动到 GPU 上运行、导出、加载等。
- nn.Parameter：一种变量，当把它赋值给一个Module 时，被自动 地注册为一个参数。
- autograd.Function：实现一个自动求导操作的前向和反向定义，每个变量操作至少创建一个函数点，每一个Tensor 的操作都会创建一个接到创建Tensor 和编码其历史 的函数的Function 节点。

重点如下:

- 定义一个网络
- 处理输入，调用 backword

还剩:

- 计算损失
- 更新网络权重

损失函数

一个损失函数接受一对 (output, target) 作为输入，计算一个值来估计网络的输出和目标值相差多少。

译者注：output 为网络的输出，target 为实际值

nn 包中有很多不同的损失函数。

nn.MSELoss是一个比较简单的损失函数，它计算输出和目标间的**均方误差**，

例如：

```
output = net(input)
target = torch.rand(10)
target = target.view(1, -1)
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

```
tensor(0.4526, grad_fn=<MseLossBackward0>)
```

现在，如果在反向过程中跟随 loss ， 使用它的 .grad_fn 属性，将看到如下所示的计算图。

input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d

-> view -> linear -> relu -> linear -> relu -> linear

-> MSELoss

-> loss

所以，当我们调用 loss.backward() 时,整张计算图都会

根据 loss 进行微分，而且图中所有设置为 requires_grad=True 的张量

将会拥有一个随着梯度累积的 .grad 张量。

为了说明，让我们向后退几步:

```
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
```

```
<MseLossBackward0 object at 0x7f417c5f3d30>
<AddmmBackward0 object at 0x7f417c5f34f0>
<AccumulateGrad object at 0x7f417c5f3d30>
```

反向传播

调用 loss.backward() 获得反向传播的误差。

但是在调用前需要清除已存在的梯度，否则梯度将被累加到已存在的梯度。

现在，我们将调用 loss.backward()，并查看 conv1 层的偏差（bias）项在反向传播前后的梯度。

```
net.zero_grad()
```

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

loss.backward()

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

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bais.grad after backward
tensor([ 0.0040,  0.0098,  0.0213, -0.0162,  0.0075, -0.0018])
```

如何使用损失函数

稍后阅读:

nn 包，包含了各种用来构成深度神经网络构建块的模块和损失函数，完整的文档请查看 here。

剩下的最后一件事:

- 新网络的权重

更新权重

在实践中最简单的权重更新规则是随机梯度下降（SGD）：

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

我们可以使用简单的 Python 代码实现这个规则：

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

但是当使用神经网络是想要使用各种不同的更新规则时，比如 SGD、Nesterov-SGD、Adam、RMS ROP 等，PyTorch 中构建了一个包 torch.optim 实现了所有的这些规则。

使用它们非常简单：

```
import torch.optim as optim

# 创建优化器
optimizer = optim.SGD(net.parameters(), lr=0.01)

# 迭代训练
optimizer.zero_grad() # 梯度清零
output = net(input)
loss = criterion(output, target) # 计算损失
loss.backward() # 反向传播
optimizer.step() # 更新参数
```

注意

观察如何使用 optimizer.zero_grad() 手动将梯度缓冲区设置为零。

这是因为梯度是按 Backprop 部分中的说明累积的。