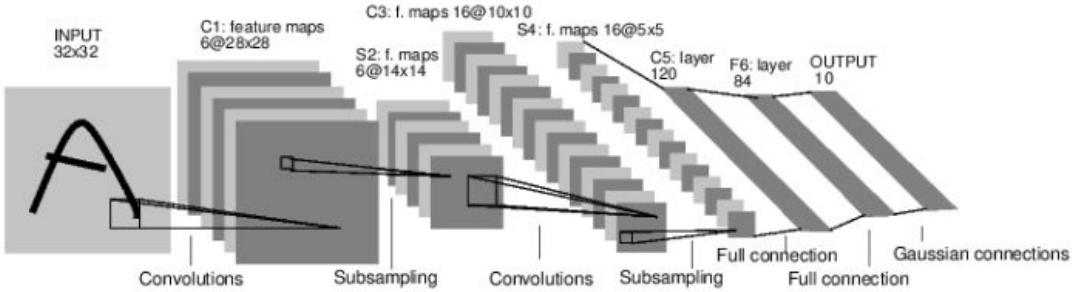
PyTorch 神经网络

在PyTorch中`torch.nn`专门用于实现神经网络。其中`nn.Module`包含了网络层的搭建，以及`forward(input)`并返回网络的输出`output`.

> 经典的LeNet网络，用于对字符进行分类
>
> 
>

对于神经网络，标准的训练流程：

- 定义一个多层神经网络
- 对数据集的预处理并准备作为网络的输出
- 将数据输入到网络
- 反向传播梯度计算
- 更新网络的参数，例如`weight = weight - learning_rate * gradient`

补充：PyTorch常用工具包

- `torch`：类似于NumPy的张量库， 强GPU支持
- `torch.autograd`：基于tape的自动区别库，支持torch之中的所有可区分张量运行
- `torch.nn`：为最大化灵活性未涉及、与 autograd 深度整合的神经网络库
- `torch.optim`：与 torch.nn 一起使用的优化包，包含 SGD、RMSProp、LBFGS、Adam 等标准优化方式
- `torch.multiprocessing`：python 多进程并发，进程之间 torch Tensors 的内存共享；
- `torch.utils`：数据载入器。具有训练器和其他便利功能
- `torch.legacy(.nn/.optim)`：处于向后兼容性考虑，从 Torch 移植来的 legacy 代码；

1. 定义网络

首先定义一个神经网络（两层卷积，三层全连接）：

```
1 import torch
2 import torch.nn as nn
```

```
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()  # 用于调用父类（超类）的一个方法
        # 输出图像时单通道，conv1 kernel_size=5*5, 输出通道 6
        self.conv1 = nn.Conv2d(1, 6, 5)  # 输入通道， 输出通道，核尺寸
        # conv2 kernal_size = 5*5 输出通道 16
        self.conv2 = nn.Conv2d(6, 16, 5)  # 输入通道， 输出通道，核尺寸
        # 全连接层
        self.fc1 = nn.Linear(16*5*5, 120)  # 输入尺寸， 输出尺寸
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # max_pooling 采用一个（2， 2）的滑动窗口
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # F.max_pool2d(输入数据， 滑动窗口)为二维最大池化层
        # F.relu()即为激活函数， kernel大小是方形的话，如（2， 2）用2即可
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        # 修改tensor尺寸,转为全连接层的输入数据
        # 三个全连接层
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        # 除了batch维度以外的所有维度
        size = x.size()[1:]
        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)
)  # 网络结构
```

注：此处必须实现forward函数，而backward函数在采用`autograd`时就自动定义好了，在`forward`方法可以采用任何的张量操作。

`net.parameters()`可以返回网络的训练参数，例如：

```
params = list(net.parameters())
print('参数数量:', len(params))
# conv1.weight
print('第一个参数大小：',  params[0].size())
```

输出：

```
参数数量：10
第一个参数大小：  torch.Size([6, 1, 5, 5])
```

为了测试这个网络，随机生成一个32*32的输入：

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

输出：

```
tensor([[[[ 0.4259,  0.2679,  0.0262,  ..., -1.0569,  0.4705,
 0.4008],
          [ 1.8223,  0.8138, -0.1017,  ..., -1.0569,  0.7585,
 0.0635],
          [ 0.4283,  1.1474,  0.0711,  ...,  0.6135,  0.0446,
 0.3045],
          ...,
          [ 1.1644,  0.6545,  1.5432,  ..., -0.0858,  0.5562,
-0.2700],
          [-1.0411,  0.8606, -1.7346,  ..., -0.7582, -1.6704,
 2.0953],
          [-0.8799,  1.0659, -0.0133,  ..., -1.1116, -0.8369,
 1.5369]]]])
tensor([[ 0.0973, -0.0196,  0.0732, -0.0156, -0.1001,  0.0292,
 0.0037,  0.1253,
          0.0886, -0.0057]], grad_fn=<AddmmBackward>)
```

然后反向传播，反向传播需要先清空梯度缓存，再反向传播随机梯度。

```
# 清空所有参数的梯度缓存，然后计算随机梯度进行反向传播
net.zero_grad()
out.backward(torch.randn(1, 10))
```

注：

> `torch.nn`只支持**小批量（mini-batches）**数据，不能输入单个样本。
>
> 例如：
>
> `nn.Conv2d`接受的输入是一个4维张量--`nSamples * nChannels * Height * Width`所以若输入的是单个样本，则需要采用`input.unsqueeze(0)`来扩充一个假的batch维度，从3维变为4维。

2.损失函数

损失函数的输入是`(output, target)`，即网络输出和真实标签对的数据，然后返回一个数值表示网络输出和真实输出的差距。

以均方误差`nn.MSELoss`为例：

```
output = net(input)
    # 定义伪标签
    target = torch.randn(10)
    # 调整大小，和output相同形状
    target = target.view(1, -1)
    criterion = nn.MSELoss()  # 损失函数
    loss = criterion(output, target)
    print(loss)
```

输出：

```
tensor(1.3050, grad_fn=<MseLossBackward>)
```

此过程中，整个网络的数据输入到输出经历的计算图：

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

即：数据从输入层到输出层，计算`loss`

调用`loss.backward()`，整个图都是可以微分的。即：包括`loss`，图中所有张量只要其属性`requires_grad=True`，那么其梯度`.grad`都会随着梯度一直累计。

代码表示：

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

输出：

```
<MseLossBackward object at 0x0000027BD4E696A0>
<AddmmBackward object at 0x0000027BD6D9EB38>
<AccumulateGrad object at 0x0000027BD4E696A0>
```

3.反向传播

反向传播需要先清空当前梯度缓存（使用`.zero_grad()`方法）然后调用`loss.backward()`方法。

例：

`conv1`层的偏置参数`bias`反向传播：

```
net.zero_grad()  # 清空所有参数的梯度缓存
print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()  # loss反向传播

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([ 0.0069,  0.0021,  0.0090, -0.0060, -0.0008, -0.0073])
```

注：关于`torch.nn`库：[torch.nn官方文档]

4.更新权重

采用随机梯度下降（stochastic Gradient Descent SGD）方法更新权重：

`weight = weight - learning_rate*gradient`

代码实现：

```
learning_rate = 0.01
for f in net.parameters():  # 将网络中每一个参数进行遍历
    f.data.sub_(f.grad.data * learning_rate)  # sub_()方法对变量数据本身进行更改替换
```

其他优化方法采用`torch.optim`库，例如：

```python
import torch.optim as optim

optimizer = optim.SGD(net.parameters(), lr=0.01)  # 创建优化器
# 训练过程中的参数更新：
optimizer.zero_grad()  # 清空梯度缓存
output = net(input)  # 前向传播
loss = criterion(output, target)  # 计算损失
loss.backward()  # 反向传播
optimizer.step()  # 更新权重
```