最近在学习PyTorch，用的是《深度学习框架PyTorch：入门与实战》这本书，然后做些笔记来加深印象。

书籍开源地址：https://github.com/chenyuntc/pytorch-book

PyTorch第一步

1 Tensor

Tensor是PyTorch中重要的数据结构，可认为是一个高维数组。它可以是一个数（标量）、一维数组（向量）、二维数组（矩阵）以及更高维的数组。Tensor和Numpy的ndarrays类似，但Tensor可以使用GPU进行加速。Tensor的使用和Numpy及Matlab的接口十分相似，下面通过几个例子来看看Tensor的基本使用。

1.1 构建矩阵和加法运算

注意：函数名后面带下划线_ 的函数会修改Tensor本身。

```
from __future__ import print_function
import torch as t
t.__version__

# 构建 5x3 矩阵，只是分配了空间，未初始化
x = t.Tensor(5, 3)

# 使用[0,1]均匀分布随机初始化二维数组
x = t.rand(5, 3)

# 查看x的形状
print(x.size())

# 查看列的个数, 两种写法等价
x.size()[1]
x.size(1)

# 加法
x = t.rand(5, 3)
y = t.rand(5, 3)
# 加法的第一种写法
x + y
# 加法的第二种写法
t.add(x, y)
# 加法的第三种写法：指定加法结果的输出目标为result
result = t.Tensor(5, 3) # 预先分配空间
t.add(x, y, out=result) # 输入到result

# 加法的另外一种表示形式：将一个数加到另一个数上面
# 注意，函数名后面带下划线_ 的函数会修改Tensor本身。
y.add(x) # 普通加法，不改变y的内容
y.add_(x) # inplace 加法，y变了
```

1.2 Tensor和Numpy中的数组

Tensor还支持很多操作，包括数学运算、线性代数、选择、切片等等，其接口设计与Numpy极为相似。

Tensor和Numpy的数组之间的互操作非常容易且快速。对于Tensor不支持的操作，可以先转为Numpy数组处理，之后再转回Tensor。

Tensor和numpy对象共享内存，所以他们之间的转换很快，而且几乎不会消耗什么资源。但这也意味着，如果其中一个变了，另外一个也会随之改变。

```
# Tensor的选取操作与Numpy类似
# 选取第二列
x[:, 1]

# 新建一个全1的Tensor
a = t.ones(5)
# Tensor -> Numpy
b = a.numpy()

import numpy as np
# 新建一个全1的numpy数组
a = np.ones(5)
# Numpy->Tensor
b = t.from_numpy(a)
```

1.3 scalar

如果你想获取某一个元素的值，可以使用`scalar.item`。 直接`tensor[idx]`得到的还是一个tensor: 一个0维的tensor，一般称为scalar.

```
a = t.ones(5)
scalar = a[0]
scalar
scalar.size() #0-dim
scalar.item() # 使用scalar.item()能从中取出python对象的数值

# 注意tensor和scalar的区别
tensor = t.tensor([2])
tensor
tensor.size()
tensor.item() # 只有一个元素的tensor也可以调用`tensor.item()`
```

1.4 内存共享

需要注意的是，`t.tensor()`或者`tensor.clone()`总是会进行数据拷贝，新tensor和原来的数据不再共享内存。所以如果你想共享内存的话，建议使用`torch.from_numpy()`或者`tensor.detach()`来新建一个tensor, 二者共享内存。

```
# 拷贝不共享内存的数据
old_tensor = tensor
new_tensor = old_tensor.clone()
new_tensor[0] = 1111
old_tensor, new_tensor

# 共享内存
new_tensor = old_tensor.detach()
new_tensor[0] = 1111
old_tensor, new_tensor
```

1.5 GPU加速

Tensor可通过`.cuda` 方法转为GPU的Tensor，从而享受GPU带来的加速运算。

```
# 在不支持CUDA的机器下，下一步还是在CPU上运行
device = t.device("cuda:0" if t.cuda.is_available() else "cpu")
x = x.to(device)
y = y.to(x.device)
z = x + y
```

2 autograd: 自动微分

深度学习的算法本质上是通过反向传播求导数，而PyTorch的**`autograd`**模块则实现了此功能。在Tensor上的所有操作，autograd都能为它们自动提供微分，避免了手动计算导数的复杂过程。

要想使得Tensor使用autograd功能，只需要设置`tensor.requries_grad=True`.

```
# 为tensor设置 requires_grad 标识，代表着需要求导数
# pytorch 会自动调用autograd 记录操作
x = t.ones(2, 2, requires_grad=True)

y = x.sum()
y.grad_fn
y.backward() # 反向传播,计算梯度
# y = x.sum() = (x[0][0] + x[0][1] + x[1][0] + x[1][1])
# 每个值的梯度都为1
x.grad
```

注意：`grad`在反向传播过程中是累加的(accumulated)，这意味着每一次运行反向传播，梯度都会累加之前的梯度，所以反向传播之前需把梯度清零。

```
# 梯度清零
x.grad.data.zero_()
```

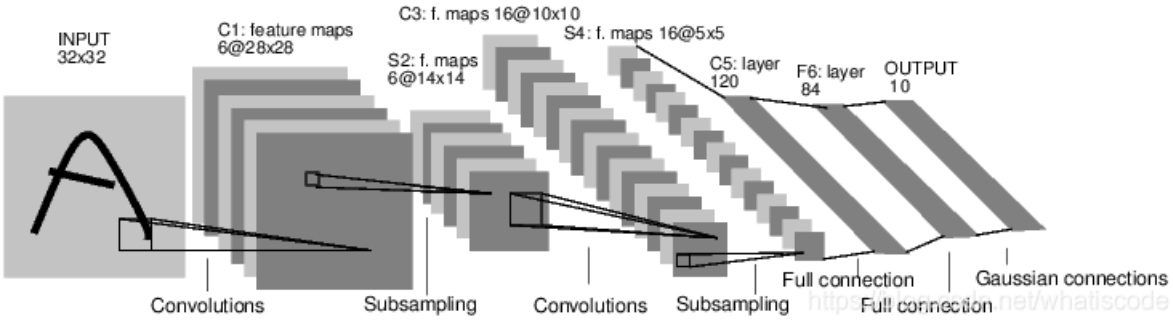
3 神经网络

Autograd实现了反向传播功能，但是直接用来写深度学习的代码在很多情况下还是稍显复杂，torch.nn是专门为神经网络设计的模块化接口。nn构建于 Autograd之上，可用来定义和运行神经网络。

nn.Module是nn中最重要的类，可把它看成是一个网络的封装，包含网络各层定义以及forward方法，调用forward(input)方法，可返回前向传播的结果。

下面就以最早的卷积神经网络：LeNet为例，来看看如何用`nn.Module`实现。LeNet的网络结构如下图所示。这是一个基础的前

向传播(feed-forward)网络: 接收输入，经过层层传递运算，得到输出。



3.1 定义一个神经网络

定义网络时，需要继承`nn.Module`，并实现它的forward方法，把网络中具有可学习参数的层放在构造函数`__init__`中。如果某一层(如ReLU)不具有可学习的参数，则既可以放在构造函数中，也可以不放，但建议不放在其中，而在forward中使用`nn.functional`代替。

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

class Net(nn.Module):
    def __init__(self):
        # nn.Module子类的函数必须在构造函数中执行父类的构造函数
        # 下式等价于nn.Module.__init__(self)
        super(Net, self).__init__()

        # 第一个卷积层 '1'表示输入图片为单通道, '6'表示输出通道数（即卷积核的个数）, '5'表示卷积核大小为5*5
        self.conv1 = nn.Conv2d(1, 6, 5)
        # 第二个卷积层
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 仿射层/全连接层，y = Wx + b
        # 第一个参数表示前面一层神经元的个数，第二个参数表示后面一层神经元的个数
        self.fc1   = nn.Linear(16*5*5, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        # 卷积 -> 激活 -> 池化
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # 在torch里面，view函数相当于numpy的reshape, '-1'表示自适应
        x = x.view(x.size()[0], -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

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

只要在nn.Module的子类中定义了forward函数，backward函数就会自动被实现(利用`autograd`)。在`forward` 函数中可使用任何tensor支持的函数，还可以使用if、for循环、print、log等Python语法，写法和标准的Python写法一致。

3.2 神经网络的网络参数

网络的可学习参数通过`net.parameters()`返回，`net.named_parameters`可同时返回可学习的参数及名称。

```
params = list(net.parameters())
print(len(params))

# 打印所有参数的名字和大小，都是用Tensor表示
for name,parameters in net.named_parameters():
    print(name,':',parameters.size())
```

3.3 神经网络的输入和输出

forward函数的输入和输出都是Tensor。

```
# 随机输入并输出
input = t.randn(1, 1, 32, 32)
out = net(input)
```

```
out.size()

net.zero_grad() # 所有参数的梯度清零
out.backward(t.ones(1,10)) # 反向传播
```

需要注意的是，torch.nn只支持mini-batches，不支持一次只输入一个样本，即一次必须是一个batch。但如果只想输入一个样本，则用 `input.unsqueeze(0)`将batch_size设为 1 。例如 `nn.Conv2d` 输入必须是4维的，形如 $nSamples \times nChannels \times Height \times Width$ nSamples \times nChannels \times Height \times Width nSamples × nChannels × Height × Width。可将nSample设为1，即1$ \times nChannels \times Height \times Width$。

3.4 损失函数

nn实现了神经网络中大多数的损失函数，例如nn.MSELoss用来计算均方误差，nn.CrossEntropyLoss用来计算交叉熵损失。

```
output = net(input)
target = t.arange(0,10).view(1,10).float()
criterion = nn.MSELoss()
loss = criterion(output, target)
loss # loss是个scalar
```

如果对loss进行反向传播溯源(使用`gradfn`属性)，可看到它的计算图如下：

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

当调用`loss.backward()`时，该图会动态生成并自动微分，也即会自动计算图中参数(Parameter)的导数。

```
# 运行.backward，观察调用之前和调用之后的grad
net.zero_grad() # 把net中所有可学习参数的梯度清零
print('反向传播之前 conv1.bias的梯度')
print(net.conv1.bias.grad)
loss.backward()
print('反向传播之后 conv1.bias的梯度')
print(net.conv1.bias.grad)
```

3.5 优化器

在反向传播计算完所有参数的梯度后，还需要使用优化方法来更新网络的权重和参数，例如随机梯度下降法(SGD)的更新策略为：weight = weight - learning_rate * gradient

手动实现为：

```
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)# inplace 减法
```

`torch.optim`中实现了深度学习中绝大多数的优化方法，例如RMSProp、Adam、SGD等，更便于使用，因此大多数时候并不需要手动写上述代码。

```
import torch.optim as optim
#新建一个优化器，指定要调整的参数和学习率
optimizer = optim.SGD(net.parameters(), lr = 0.01)

# 在训练过程中
# 先梯度清零(与net.zero_grad()效果一样)
optimizer.zero_grad()

# 计算损失
output = net(input)
loss = criterion(output, target)

#反向传播
loss.backward()

#更新参数
optimizer.step()
```

3.6 数据加载与预处理

在深度学习中数据加载及预处理是非常复杂繁琐的，但PyTorch提供了一些可极大简化和加快数据处理流程的工具。同时，对于常用的数据集，PyTorch也提供了封装好的接口供用户快速调用，这些数据集主要保存在torchvison中。

`torchvision`实现了常用的图像数据加载功能，例如Imagenet、CIFAR10、MNIST等，以及常用的数据转换操作，这极大地方便了数据加载，并且代码具有可重用性。