一、Tensor

Tensor的基本类型

```
32位浮点型：torch.FloatTensor。 （默认）
64位整型：torch.LongTensor。
32位整型：torch.IntTensor。
16位整型：torch.ShortTensor。
64位浮点型：torch.DoubleTensor。 #除以上数字类型外，还有 byte和chart型
#类型转换
long=tensor.long() half=tensor.half() int_t=tensor.int()
flo = tensor.float() short = tensor.short() ch = tensor.char()
bt = tensor.byte()
```

创建Tensor

```
创建随机矩阵：x = torch.rand(5, 3)
创建空矩阵：x = torch.empty(5, 3)
创建0矩阵：x = torch.zeros(5, 3, dtype=torch.long)
创建1矩阵：a = torch.ones(5)
创建单位矩阵，对角线为1,其他为0：eye=torch.eye(2,2)
矩阵转tensor(标量)：x = torch.tensor([5.5, 3])
根据现有张量创建1矩阵：a = torch.ones_like(b) #torch.zeros_like(b)
根据现有张量(修改了dtype)：x = torch.randn_like(x, dtype=torch.float)
```

Tensor查询

```
查询维度： print(x.shape) print(x.size())
取出标量Tensor的值：x.item() #张量中只有一个元素的tensor也可以调用
查询数据类型：x.type()
```

Tensor操作

```
#与numpy基本一致
#沿行取最大值和对应列索引
max_value, max_idx = torch.max(x, dim=1)

#沿行求和
sum_x = torch.sum(x, dim=1)

#加法：
x = y = torch.rand(5, 3)
print(x + y)

```

```
#替换
y.add_(x) #不同于y=y+x，前者的y是同一个地址，后者的y不同于之前的地址

#索引（等同于Numpy）：
print(x[:, 1])

#重置维度：
y = x.view(16) z = x.view(-1, 8) #等同于Numpy的reshape
```

Numpy转换

```
#torch转numpy：
a = torch.ones(5)
b = a.numpy()

#numpy转torch：
a = np.ones(5)
b = torch.from_numpy(a) #分配相同地址
#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。
```

CUDA

```
#CPU移植到GPU
cpu_a=torch.rand(4, 3)
gpu_a=cpu_a.cuda()

#GPU移植到CPU
cpu_b=gpu_a.cpu()

#多GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
 gpu_b=cpu_b.to(device)
```

二、自动求导

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2 z = y * y * 3 out = z.mean()
```

```
out.backward() #求梯度
#out.backward(torch.ones_like(out)) #out是向量就必须赋予其权重，标量默认None
print(x.grad) #打印x的梯度
```

得到 $o = \frac{1}{4}\sum_i z_i,\ z_i = 3(x_i + 2)^2$ and $z_i \mid_{x_i = 1} = 27.$

因此, $\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i + 2)$, hence $\frac{\partial o}{\partial x_i} \mid_{x_i = 1} = \frac{9}{2} = 4.5.$

三、神经网络

首先定义一个网络:

```
import torch
import torch.nn as nn
import torch.nn.functional as F #包含了一些不可学习的常用函数(如ReLU，pool，DropOut)
class Net(nn.Module): #继承nn.Module，并实现它的forward方法
 def __init__(self):
 super(Net, self).__init__() #必须在构造函数中执行父类的构造函数
 self.conv1 = nn.Conv2d(1, 6, 3) #1表示输入为单通道，6表示输出通道数，3表示卷积核为3*3
 self.fc1 = nn.Linear(1350, 10) #线性层，输入1350个特征(forward函数决定)，输出10个特征
 def forward(self, x):
 x.size() # [1, 1, 32, 32]
 x = self.conv1(x) #根据卷积的尺寸计算公式，计算结果是30
 x = F.relu(x) #[1, 6, 30, 30]
 x = F.max_pool2d(x, (2, 2)) #我们使用池化层，计算结果是15
 x = F.relu(x) #[1, 6, 15, 15]
 x = x.view(x.size()[0], -1) #[1, 1350]
 x = self.fc1(x)
 return x
for parameters in net.parameters():
 print(parameters)
for name,parameters in net.named_parameters():
 print(name,':',parameters.size())
```

- 定义一个网络
- 前向传播
- 计算损失

- 反向传播
- 更新网络权重

```
#定义网络
net=Net()

#前向传播
input = torch.randn(1, 1, 32, 32) #网络的输入
out = net(input) #网络的输出

#计算损失
target = torch.arange(0,10).view(1,10).float() # 随机值作为样例
 criterion = nn.MSELoss()
 loss = criterion(output, target)

 #反向传播
 optimizer = optim.SGD(net.parameters(), lr=0.01) #创建优化器SGD
 optimizer.zero_grad() # 梯度缓存清零
 loss.backward() #反向传播

 # 更新权重
 optimizer.step()
```

移植到GPU

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net.to(device)
inputs, labels = inputs.to(device), labels.to(device)
```