# **Pytorch教程一-张量的属性**

PyTorch从0.4版本开始提出了张量属性（Tensor Attributes）的概念。张量属性的主要内容包含了[torch.dtype](https://pytorch.org/docs/1.9.0/tensor_attributes.html" \l "torch.torch.dtype" \o "torch.torch.dtype),[torch.device](https://pytorch.org/docs/1.9.0/tensor_attributes.html" \l "torch.torch.device" \o "torch.torch.device)和[torch.layout](https://pytorch.org/docs/1.9.0/tensor_attributes.html" \l "torch.torch.layout" \o "torch.torch.layout)这三个部分。这里提及的不是获取张量的data属性、和梯度等属性。

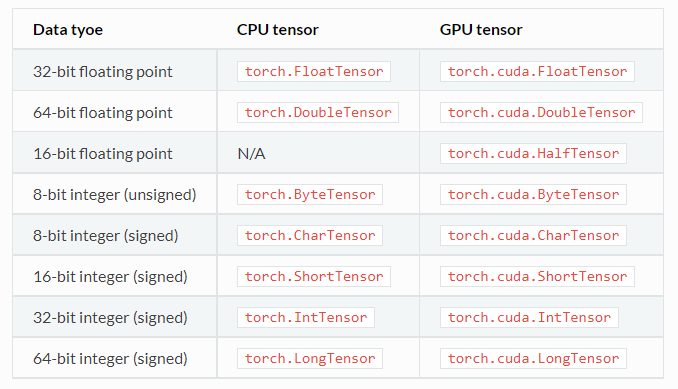
**官方文档提到：**

Each torch.Tensor has a [torch.dtype](https://pytorch.org/docs/1.9.0/tensor_attributes.html" \l "torch.torch.dtype" \o "torch.torch.dtype),[torch.device](https://pytorch.org/docs/1.9.0/tensor_attributes.html" \l "torch.torch.device" \o "torch.torch.device), and [torch.layout](https://pytorch.org/docs/1.9.0/tensor_attributes.html" \l "torch.torch.layout" \o "torch.torch.layout).

接下来我们对这三个属性进行系统的学习。

1. **torch.dtype**

跟其他编程语言一样数据都会定义一些数据类型，比如Python中的int型，char型和float型等等。同样在PyTorch中张量也存在着不同的数据类型，由于tensor可以运行在CPU和GPU中。因此tensor也就存在了两种形态：（其中unsigned表示无符号的意思）



另外在笔者使用的PyTorch版本（1.9.0）中，张量的类型有以下12种的数据类型如下所示（表格数据来源于官方文档），在实际的项目工程中我们常用的就是**float32、int8,uint8,int32**这些类型。



很显然根据实际张量的大小使用不同的数据类型的好处也是很明显的：不仅可以保证一定的数据精度，还能在一定程度上减少内存的使用，尤其是我们在进行深度学习时进行模型数据加载和训练的时候。

注意：在torch中张量的默认数据类型是torch.float32类型，具体的可以使用以下代码查看：

import torch

print(torch.get\_default\_dtype())

# 输出

torch.float32

接下来我们来看一些案例，这些案例主要与张量的类型有关的操作：

**（1）检查张量是否是浮点数类型：**

import torch

tensor\_1 = torch.tensor([1,2,3])

print(torch.is\_floating\_point(tensor\_1))

print(tensor\_1.dtype)

# 输出

False

torch.int64

可以看出，is\_floating\_point()可以用来判断一个张量的是否是浮点类型的，也可以看出非布尔型的整型的张量的类型是torch.int64类型。

1. **复数数据类型**

tensor\_2 = torch.tensor([1,2.0,3],dtype=torch.complex64)

print(tensor\_2)

print(torch.is\_complex(tensor\_2))

# 输出

tensor([1.+0.j, 2.+0.j, 3.+0.j])

True

1. **不同类型的张量进行运算结果：**

这里先给出在Torch中操作数类型的高低顺序：

**复数 > 浮点数 > 整数 > bool数**

我们先创建几个不同类型的张量（修改于官方文档）：

uint\_tensor = torch.ones((2,3), dtype=torch.uint8)  
int\_tensor = torch.ones((2,3), dtype=torch.int)  
float\_tensor = torch.ones((2,3), dtype=torch.float)  
long\_tensor = torch.ones((2,3), dtype=torch.long)  
double\_tensor = torch.ones((2,3), dtype=torch.double)  
complex\_float\_tensor = torch.ones((2,3), dtype=torch.complex64)  
complex\_double\_tensor = torch.ones((2,3), dtype=torch.complex128)  
bool\_tensor = torch.ones((2,3), dtype=torch.bool)  
  
long\_zerodim = torch.tensor((2,3), dtype=torch.long)  
int\_zerodim = torch.tensor((2,3), dtype=torch.int)

**那么有以下的操作:**

*# 1.int64加上常数结果为默认的int64类型*print(torch.add(5, 5))  
print(torch.add(5, 5).dtype)  
  
*# 2.torch.int 与常数相加结果为torch.int32类型*print(int\_tensor + 10)  
print((int\_tensor + 10).dtype)  
  
*# 3.torch.int与torch.long数据相加为torch.int64类型*print(int\_tensor + long\_tensor)  
print((int\_tensor + long\_tensor).dtype)  
  
*# 4. troch.float与torch.double相加结果为torch.float64类型*print(float\_tensor + double\_tensor)  
print((float\_tensor + double\_tensor).dtype)  
  
*# 5.torch.complex64与torch.complex128相加结果为torch.complex128*print(complex\_float\_tensor + complex\_double\_tensor)  
print((complex\_float\_tensor + complex\_double\_tensor).dtype)

# 输出如下：

tensor(10)

torch.int64

tensor([[11, 11, 11],

[11, 11, 11]], dtype=torch.int32)

torch.int32

tensor([[2, 2, 2],

[2, 2, 2]])

torch.int64

tensor([[2., 2., 2.],

[2., 2., 2.]], dtype=torch.float64)

torch.float64

tensor([[2.+0.j, 2.+0.j, 2.+0.j],

[2.+0.j, 2.+0.j, 2.+0.j]], dtype=torch.complex128)

torch.complex128

**至于此我们可以得到不同类型的张量进行运算的规律：**

☑ 如果标量操作数的类型比张量操作数的类型更高，PyTorch使用足够大小以容纳该类别的所有标量操作数的类型。

☑ 如果没有更高类别的零维度（常数tensor）操作数，PyTorch使用足够大小和类别的类型以容纳所有维度的操作数。

上述只是简单的举例，更多的不同操作类型数据之间的运算实验大家自己研究。

1. **torch.device**

之前说过，PyTorch的张量可以运行在CPU和GPU上，我们也有这样的需求：想知道程序中的常量是存在于CPU或GPU上。此时PyTorch正好提供了一个张量的属性方法供我们使用。

使用torch.device我们可以知道torch.Tensor分配到的设备。分配的设备有cpu和cuda两种，cuda即gpu。

在工程之前我们先要查看一下环境中的cuda的相关情况：代码如下：

*# 先检查一下cuda是否可用*print(torch.cuda.is\_available())  
  
*# 检查可用的cuda的数量*print(torch.cuda.device\_count())  
  
*# 返回当前cuda使用的设备的索引号*print(torch.cuda.current\_device())

# 在本人的电脑上输出结果如下(本人电脑是有一块gpu)：

True

1

0

我们可以利用字符或字符+序号的方式来分配设备

1. **使用字符串**

print(torch.device('cuda:0'))  
print(torch.device('cpu'))  
print(torch.device('cuda'))

# 输出

cuda:0

cpu

Cuda

1. **使用字符和序号**

print(torch.device('cuda', 0))  
print(torch.device('cpu', 0))

# 输出

cuda:0

cpu:0

那么在cuda上设置一个tensor,就可以这样：

*# 在CPU上设置一个tensor*cuda1 = torch.device('cuda:0')  
cuda\_a\_tensor = torch.randn((2,3), device=cuda1)  
print(cuda\_a\_tensor)  
print(cuda\_a\_tensor.dtype)  
print(cuda\_a\_tensor.device)

# 输出

tensor([[ 0.3755, 0.6157, 0.9005],

[-0.9571, -1.0113, -0.1565]], device='cuda:0')

torch.float32

cuda:0

这样我们就将一个tensor储存在cuda上了。

另外以下三种方式的效果是一样的：

torch.randn((2,3), device=torch.device('cuda:0'))  
torch.randn((2,3), device='cuda:0')  
torch.randn((2,3), device=0)

1. **tensor类型转换**

Tensor在cpu 和 cuda 设备的转换我们使用 'to' 来实现：

device\_cpu = torch.device("cpu") *# 声明cpu设备*device\_cuda = torch.device('cuda') *#设备gpu设备*data = torch.Tensor([1,2,3,4,5])  
print(data.device)  
data\_1 = data.to(device\_cpu)  
print(data\_1.device)  
data\_2 = data.to(device\_cuda)  
print(data\_2.device)

# 输出

cpu

cpu

cuda:0

当然我们也可以只用以下方法：

data = torch.Tensor([1,2,3,4,5])

print(data.dtype)  
print(data.to('cuda'))

# 输出

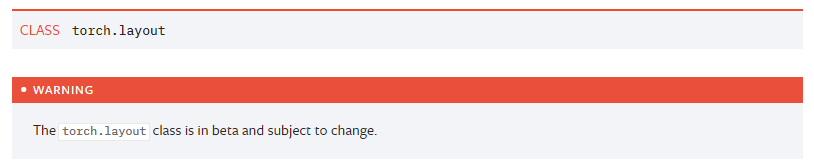
torch.float32

tensor([1., 2., 3., 4., 5.], device='cuda:0')

在工程中我们常用：device = [torch](https://so.csdn.net/so/search?q=torch&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/shaopeng568/article/details/_blank).device("cuda:0" if torch.cuda.is\_available() else "cpu")来设置tensor的存储设备。

1. **torch.layout**

官方文档中提及到：



也就是说这个内存布局相关的类还是会发生改变的，torch.[layout](https://so.csdn.net/so/search?q=layout&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Li7819559/article/details/_blank)代表对应torch.tensor变量的内存布局。这个也比较好理解,可以理解为张量在物理设备中的储存结构，好比数据结构中的顺序存储和链式存储一样是一种数据存储。

目前的笔者使用的PyTorch中，torch.layout 可选torch.stried或torch.sparse\_coo。前者用于密集张量后者用于稀疏张量(可类比稀疏矩阵-0很多的矩阵)。

官方文档中的例子如下：

x = torch.tensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
print(x.stride())

# 输出

（5,1）

这是什么意思了？意思是：第 k 个步幅表示在张量的第 k 维中从一个元素到下一个元素所需的内存跳跃。

具体到输出（5,1）就是：第一维中，一个元素到下一个元素需要5个内存，而在第二个维度中，一个元素到下一个元素需要1个内存。官方文档说使用这个layout概念使得高效地执行许多张量操作成为可能。还有一个**torch.memory\_format**，感兴趣的小伙伴可以去了解一下。

1. **总结**

以上就是本次的文章的全部内容，下次文章我们来看一下Torch的张量属性相关的知识，下一节内容我们开看一下张量操作相关的内容。