第三章 PyTorch基礎：Tensor

- 出處: https://github.com/chenyuntc/pytorch-book

3.1 Tensor

Tensor，又名張量，讀者可能對這個名詞似曾相識，因它不僅在PyTorch中出現過，它也是Theano、TensorFlow、 Torch和MxNet中重要的資料結構。關於張量的本質不乏深度的剖析，但從工程角度來講，可簡單地認為它就是一個陣列，且支援高效的科學計算。它可以是一個數（標量）、一維陣列（向量）、二維陣列（矩陣）和更高維的陣列（高階資料）。Tensor和Numpy的ndarrays類似，但PyTorch的tensor支援GPU加速。

本節將系統講解tensor的使用，力求面面俱到，但不會涉及每個函數。對於更多函數及其用法，可通過在IPython/Notebook中使用函數名加?查看幫助文檔，或查閱PyTorch官方文檔.
http://docs.pytorch.org

```
In [1]:
# Let's begin
from __future__ import print_function
import torch  as t
t.__version__
Out[1]:
'0.4.0a0+200fb22'
```

3.1.1 基礎操作

- 從介面的角度來講，對tensor的操作可分為兩類：
  1. Torch.function，如torch.save等。
  2. 另一類是tensor.function，如tensor.view等。
  3. 為方便使用，對tensor的大部分操作同時支援這兩類介面。
- 從存儲的角度來講，對tensor的操作又可分為兩類：
  1. 不會修改自身的資料，如a.add(b)，加法的結果會返回一個新的tensor。
  2. 會修改自身資料，如a.add_(b)，加法的結果仍存儲在a中，a被修改了。
  3. 函數名以_結尾的都是inplace方式, 即會修改調用者自己的資料，在實際應用中需加以區分。

創建Tensor

在PyTorch中新建tensor的方法有很多，具體如表3-1所示。

表3-1: 常見新建tensor的方法

函數	功能
Tensor(*sizes)	基礎構造函數
tensor(data,)	類似np.array的構造函數
ones(*sizes)	全1Tensor
zeros(*sizes)	全0Tensor
eye(*sizes)	對角線為1，其他為0
arange(s,e,step	從s到e，步長為step
linspace(s,e,steps)	從s到e，均勻切分成steps份
rand/randn(*sizes)	均勻/標準分佈
normal(mean,std)/uniform(from,to)	正態分佈/均勻分佈
randperm(m)	隨機排列

表3-1所示的這些創建方法都可以在創建的時候指定資料類型dtype和存放device(cpu/gpu).

其中使用Tensor函數新建tensor是最複雜多變的方式，它既可以接收一個list，並根據list的資料新建tensor，也能根據指定的形狀新建tensor，還能傳入其他的tensor，舉幾個例子。

```
In [2]:
# 指定 tensor 的形狀
a = t.Tensor(2, 3)
a # 數值取決於記憶體空間的狀態，print 時候可能 overflow
Out[2]:
tensor(1.00000e-37 *
       [[-8.9677,  0.0003, -8.9884],
        [ 0.0003,  0.0000,  0.0000]])
In [3]:
# 用 list 的資料創建 tensor
b = t.Tensor([[1,2,3],[4,5,6]])
b
Out[3]:
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])
In [4]:
b.tolist() # 把 tensor 轉為 list
Out[4]:
[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
```

Tensor.size()返回torch.Size物件，它是tuple的子類，但其使用方式與tuple略有區別

```
In [5]:
b_size = b.size()
b_size
Out[5]:
torch.Size([2, 3])
In [6]:
b.numel() # b中元素總個數，2*3，等價於b.nelement()
Out[6]:
6
In [7]:
# 創建一個和b形狀一樣的tensor
c = t.Tensor(b_size)
# 創建一個元素為2和3的tensor
d = t.Tensor((2, 3))
c, d
Out[7]:
(tensor(1.00000e-36 *
        [[-1.0148,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000]]), tensor([ 2.,  3.]))
```

除了tensor.size()，還可以利用 tensor.shape 直接查看tensor的形狀，tensor.shape等價於tensor.size()

```
In [8]:
c.shape
Out[8]:
torch.Size([2, 3])
```

需要注意的是，t.Tensor(*sizes)創建tensor時，系統不會馬上分配空間，只是會計算剩餘的記憶體是否足夠使用，使用到tensor時才會分配，而其它操作都是在創建完tensor之後馬上進行空間分配。其它常用的創建tensor的方法舉例如下。

```
In [9]:
t.ones(2, 3)
Out[9]:
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
In [10]:
t.zeros(2, 3)
Out[10]:
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

```
In [11]:
t.arange(1, 6, 2)
Out[11]:
tensor([ 1.,  3.,  5.])
In [12]:
t.linspace(1, 10, 3)
Out[12]:
tensor([  1.0000,   5.5000,  10.0000])
In [13]:
t.randn(2, 3, device=t.device('cpu'))
Out[13]:
tensor([[-0.7008, -0.0361, -0.3875],
        [ 1.8125, -0.0834, -2.0546]])
In [14]:
t.randperm(5) # 長度為 5 的隨機排列
Out[14]:
tensor([ 0,  2,  3,  1,  4])
In [15]:
t.eye(2, 3, dtype=t.int) # 對角線為 1, 不要求行列數一致
Out[15]:
tensor([[ 1,  0,  0],
        [ 0,  1,  0]], dtype=torch.int32)
```

Torch.tensor是在0.4版本新增加的一個新版本的創建tensor方法，使用的方法，和參數幾乎和np.array完全一致

```
In [16]:
scalar = t.tensor(3.14159)
print('scalar: %s, shape of sclar: %s' %(scalar, scalar.shape))
scalar: tensor(3.1416), shape of sclar: torch.Size([])

In [17]:
vector = t.tensor([1, 2])
print('vector: %s, shape of vector: %s' %(vector, vector.shape))
vector: tensor([ 1,  2]), shape of vector: torch.Size([2])

In [18]:
tensor = t.Tensor(1,2) # 注意和 t.tensor([1, 2])的區別
tensor.shape
Out[18]:
torch.Size([1, 2])
```

```
In [19]:
matrix = t.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
matrix,matrix.shape
Out[19]:
(tensor([[ 0.1000,  1.2000],
         [ 2.2000,  3.1000],
         [ 4.9000,  5.2000]]), torch.Size([3, 2]))

In [20]:
t.tensor([[0.11111, 0.222222, 0.3333333]],
                    dtype=t.float64,
                    device=t.device('cpu'))
Out[20]:
tensor([[ 0.1111,  0.2222,  0.3333]], dtype=torch.float64)

In [21]:
empty_tensor = t.tensor([])
empty_tensor.shape
Out[21]:
torch.Size([0])
```

常用Tensor操作

通過tensor.view方法可以調整tensor的形狀，但必須保證調整前後元素總數一致。view不會修改自身的資料，返回的新tensor與源tensor共用記憶體，也即更改其中的一個，另外一個也會跟著改變。在實際應用中可能經常需要添加或減少某一維度，這時候squeeze和unsqueeze兩個函數就派上用場了

```
In [22]:
a = t.arange(0, 6)
a.view(2, 3)
Out[22]:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
In [23]:
b = a.view(-1, 3) # 當某一維為-1 的時候，會自動計算它的大小
b.shape
Out[23]:
torch.Size([2, 3])
```

```
In [24]:
b.unsqueeze(1) # 注意形狀，在第 1 維（下標從 0 開始）上增加“1”
#等價於 b[:,None]
b[:, None].shape
Out[24]:
torch.Size([2, 1, 3])
In [25]:
b.unsqueeze(-2) # -2 表示倒數第二個維度
Out[25]:
tensor([[[ 0.,  1.,  2.],

         [ 3.,  4.,  5.]]])
In [26]:
c = b.view(1, 1, 1, 2, 3)
c.squeeze(0) # 壓縮第 0 維的“1”
Out[26]:
tensor([[[[ 0.,  1.,  2.],
          [ 3.,  4.,  5.]]]])
In [27]:
c.squeeze() # 把所有維度為“1”的壓縮
Out[27]:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
In [28]:
a[1] = 100
b # a 修改，b 作為 view 之後的，也會跟著修改
Out[28]:
tensor([[  0., 100.,   2.],
        [  3.,   4.,   5.]])
```

Resize是另一種可用來調整size的方法，但與view不同，它可以修改tensor的大小。如果新大小超過了原大小，會自動分配新的記憶體空間，而如果新大小小於原大小，則之前的資料依舊會被保存，看一個例子。

```
In [29]:
b.resize_(1, 3)
b
Out[29]:
tensor([[   0.,  100.,    2.]])
```

```
In [30]:
b.resize_(3, 3) # 舊的資料依舊保存著，多出的大小會分配新空間
b
Out[30]:
tensor([[   0.0000,  100.0000,    2.0000],
        [   3.0000,    4.0000,    5.0000],
        [  -0.0000,    0.0000,    0.0000]])
```

索引操作

Tensor 支持與 numpy.ndarray 類似的索引操作，語法上也類似，下面通過一些例子，講解常用的索引操作。如無特殊說明，索引出來的結果與原 tensor 共用記憶體，也即修改一個，另一個會跟著修改。

```
In [31]:
a = t.randn(3, 4)
a
Out[31]:
tensor([[-0.1855, -0.4570,  1.6097, -0.9747],
        [-0.5941, -0.8057, -0.6049,  1.5967],
        [-0.4694, -1.0633, -0.2432, -0.4794]])
In [32]:
a[0] # 第0行(下標從0開始)
Out[32]:
tensor([-0.1855, -0.4570,  1.6097, -0.9747])
In [33]:
a[:, 0] # 第0列
Out[33]:
tensor([-0.1855, -0.5941, -0.4694])
In [34]:
a[0][2] # 第0行第2個元素，等價於a[0, 2]
Out[34]:
tensor(1.6097)
In [35]:
a[0, -1] # 第0行最後一個元素
Out[35]:
tensor(-0.9747)

In [36]:
a[:2] # 前兩行
Out[36]:
tensor([[-0.1855, -0.4570,  1.6097, -0.9747],
        [-0.5941, -0.8057, -0.6049,  1.5967]])
```

```
In [37]:
a[:2, 0:2] # 前兩行，第0,1列
Out[37]:
tensor([[-0.1855, -0.4570],
        [-0.5941, -0.8057]])
In [38]:
print(a[0:1, :2]) # 第0行，前兩列
print(a[0, :2]) # 注意兩者的區別：形狀不同
tensor([[-0.1855, -0.4570]])
tensor([-0.1855, -0.4570])
In [39]:
# None類似於np.newaxis, 為a新增了一個軸
# 等價於a.view(1, a.shape[0], a.shape[1])
a[None].shape
Out[39]:
torch.Size([1, 3, 4])
In [40]:
a[None].shape # 等價於a[None,:,:]
Out[40]:
torch.Size([1, 3, 4])
In [41]:
a[:,None,:].shape
Out[41]:
torch.Size([3, 1, 4])
In [42]:
a[:,None,:,None,None].shape
Out[42]:
torch.Size([3, 1, 4, 1, 1])
In [43]:
a > 1 # 返回一個ByteTensor
Out[43]:
tensor([[ 0,  0,  1,  0],
        [ 0,  0,  0,  1],
        [ 0,  0,  0,  0]], dtype=torch.uint8)

In [44]:
a[a>1] # 等價於a.masked_select(a>1)
# 選擇結果與原tensor不共用記憶體空間
Out[44]:
tensor([ 1.6097,  1.5967])
```

```
In [45]:
a[t.LongTensor([0,1])] # 第0行和第1行
Out[45]:
tensor([[-0.1855, -0.4570,  1.6097, -0.9747],
        [-0.5941, -0.8057, -0.6049,  1.5967]])
```

其它常用的選擇函數

表3-2 常用的選擇函數

函數	功能
index_select(input, dim, index)	在指定維度dim上選取，比如選取某些行、某些列
masked_select(input, mask)	例子如上，a[a>0]，使用ByteTensor進行選取
non_zero(input)	非0元素的下標
gather(input, dim, index)	根據index，在dim維度上選取資料，輸出的size與index一樣

Gather是一個比較複雜的操作，對一個2維tensor，輸出的每個元素如下：

out[i][j] = input[index[i][j]][j] # dim=0

out[i][j] = input[i][index[i][j]] # dim=1

三維tensor的gather操作同理，下面舉幾個例子。

```
In [46]:
a = t.arange(0, 16).view(4, 4)
a
Out[46]:
tensor([[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.],
        [ 12.,  13.,  14.,  15.]])
In [47]:
# 選取對角線的元素
index = t.LongTensor([[0,1,2,3]])
a.gather(0, index)
Out[47]:
tensor([[  0.,   5.,  10.,  15.]])
```

```
In [48]:
# 選取反對角線上的元素
index = t.LongTensor([[3,2,1,0]]).t()
a.gather(1, index)
Out[48]:
tensor([[  3.],
        [  6.],
        [  9.],
        [ 12.]])
In [49]:
# 選取反對角線上的元素，注意與上面的不同
index = t.LongTensor([[3,2,1,0]])
a.gather(0, index)
Out[49]:
tensor([[ 12.,   9.,   6.,   3.]])
In [50]:
# 選取兩個對角線上的元素
index = t.LongTensor([[0,1,2,3],[3,2,1,0]]).t()
b = a.gather(1, index)
b
Out[50]:
tensor([[  0.,   3.],
        [  5.,   6.],
        [ 10.,   9.],
        [ 15.,  12.]])
```

與gather相對應的逆操作是scatter_，gather把資料從input中按index取出，而scatter_是把取出的數據再放回去。注意scatter_函數是inplace操作。

out = input.gather(dim, index)

-->近似逆操作

out = Tensor()

out.scatter_(dim, index)

```
In [51]:
# 把兩個對角線元素放回去到指定位置
c = t.zeros(4,4)
c.scatter_(1, index, b)
Out[51]:
tensor([[ 0.,  0.,  0.,  3.],
        [ 0.,  5.,  6.,  0.],
        [ 0.,  9., 10.,  0.],
        [12.,  0.,  0., 15.]])
```

對tensor的任何索引操作仍是一個tensor，想要獲取標準的python物件數值，需要調用tensor.item(), 這個方法只對包含一個元素的tensor適用

```
In [52]:
a[0,0] #依舊是 tensor)
Out[52]:
tensor(0.)
In [53]:
a[0,0].item() # python float
Out[53]:
0.0
In [54]:
d = a[0:1, 0:1, None]
print(d.shape)
d.item() # 只包含一個元素的 tensor 即可調用 tensor.item,與形狀無關
torch.Size([1, 1, 1])
Out[54]:
0.0
In [55]:
# a[0].item()  ->
# raise ValueError: only one element tensors can be converted to Python scalars
```

高級索引

- PyTorch在0.2版本中完善了索引操作，目前已經支援絕大多數numpy的高級索引1。高級索引可以看成是普通索引操作的擴展，但是高級索引操作的結果一般不和原始的Tensor貢獻內出。
- https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#advanced-indexing↩

```
In [56]:
x = t.arange(0,27).view(3,3,3)
x
Out[56]:
tensor([[[  0.,   1.,   2.],
         [  3.,   4.,   5.],
         [  6.,   7.,   8.]],

        [[  9.,  10.,  11.],
         [ 12.,  13.,  14.],
         [ 15.,  16.,  17.]],

        [[ 18.,  19.,  20.],
         [ 21.,  22.,  23.],
         [ 24.,  25.,  26.]]])
```

```
In [57]:
x[[1, 2], [1, 2], [2, 0]] # x[1,1,2]和 x[2,2,0]
Out[57]:
tensor([ 14.,  24.])
In [58]:
x[[2, 1, 0], [0], [1]] # x[2,0,1],x[1,0,1],x[0,0,1]
Out[58]:
tensor([ 19.,  10.,   1.])
In [59]:
x[[0, 2], ...] # x[0] 和 x[2]
Out[59]:
tensor([[[  0.,   1.,   2.],
         [  3.,   4.,   5.],
         [  6.,   7.,   8.]],

        [[ 18.,  19.,  20.],
         [ 21.,  22.,  23.],
         [ 24.,  25.,  26.]]])
```

Tensor類型

- Tensor有不同的資料類型，如表3-3所示，每種類型分別對應有CPU和GPU版本(HalfTensor除外)。
- 默認的tensor是FloatTensor，可通過t.set_default_tensor_type 來修改默認tensor類型(如果默認類型為GPU tensor，則所有操作都將在GPU上進行)。Tensor的類型對分析記憶體佔用很有說明。
- 例如對於一個size為(1000, 1000, 1000)的FloatTensor，它有1000*1000*1000=10^9個元素，每個元素占32bit/8 = 4Byte記憶體，所以共占大約4GB記憶體/顯存。HalfTensor是專門為GPU版本設計的，同樣的元素個數，顯存佔用只有FloatTensor的一半，所以可以極大緩解GPU顯存不足的問題，但由於HalfTensor所能表示的數值大小和精度有限1，所以可能出現溢出等問題。

表3-3: tensor資料類型

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	torch.float32 or torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64-bit floating point	torch.float64 or torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
16-bit floating point	torch.float16 or torch.half	torch.HalfTensor	torch.cuda.HalfTensor
8-bit integer (unsigned)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8-bit integer (signed)	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16-bit integer (signed)	torch.int16 or torch.short	torch.ShortTensor	torch.cuda.ShortTensor
32-bit integer (signed)	torch.int32 or torch.int	torch.IntTensor	torch.cuda.IntTensor
64-bit integer (signed)	torch.int64 or torch.long	torch.LongTensor	torch.cuda.LongTensor

- 各資料類型之間可以互相轉換，type(new_type)是通用的做法，同時還有float、long、half等快捷方法。CPU tensor與GPU tensor之間的互相轉換通過tensor.cuda和tensor.cpu方法實現，此外還可以使用tensor.to(device)。Tensor還有一個new方法，用法與t.Tensor一樣，會調用該tensor對應類型的構造函數，生成與當前tensor類型一致的tensor。 torch.*_like(tensora)可以生成和tensora擁有同樣屬性(類型，形狀，cpu/gpu)的新tensor tensor.new_*(new_shape) 新建一個不同形狀的tensor。
- https://stackoverflow.com/questions/872544/what-range-of-numbers-can-be-represented-in-a-16-32-and-64-bit-ieee-754-syste

```
In [60]:
# 設置預設 tensor，注意參數是字串
t.set_default_tensor_type('torch.DoubleTensor')
In [61]:
a = t.Tensor(2,3)
a.dtype # 現在 a 是 DoubleTensor,dtype 是 float64
Out[61]:
torch.float64
In [62]:
# 恢復之前的默認設置
t.set_default_tensor_type('torch.FloatTensor')
```

```
In [63]:
# 把 a 轉成 FloatTensor，等價於 b=a.type(t.FloatTensor)
b = a.float()
b.dtype
Out[63]:
torch.float32
In [64]:
c = a.type_as(b)
c
Out[64]:
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
In [65]:
a.new(2,3) # 等價於 torch.DoubleTensor(2,3)，建議使用 a.new_tensor
Out[65]:
tensor([[ 4.6390e-310,  1.4147e+161,  1.4917e+20],
        [ 2.0093e+174,  1.4327e+228,  1.3404e-317]], dtype=torch.float64)
In [66]:
t.zeros_like(a) #等價於 t.zeros(a.shape,dtype=a.dtype,device=a.device)
Out[66]:
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]], dtype=torch.float64)
In [67]:
t.zeros_like(a, dtype=t.int16) #可以修改某些屬性
Out[67]:
tensor([[ 0,  0,  0],
        [ 0,  0,  0]], dtype=torch.int16)
```

```
In [68]:
t.rand_like(a)
Out[68]:
tensor([[ 0.8882,  0.7037,  0.5756],
        [ 0.7113,  0.2868,  0.7597]], dtype=torch.float64)
In [69]:
a.new_ones(4,5, dtype=t.int)
Out[69]:
tensor([[ 1,  1,  1,  1,  1],
        [ 1,  1,  1,  1,  1],
        [ 1,  1,  1,  1,  1],
        [ 1,  1,  1,  1,  1]], dtype=torch.int32)
In [70]:
a.new_tensor([3,4]) #
Out[70]:
tensor([ 3.,  4.], dtype=torch.float64)
```

逐元素操作

這部分操作會對tensor的每一個元素(point-wise，又名element-wise)進行操作，此類操作的輸入與輸出形狀一致。

表3-4: 常見的逐元素操作

函數	功能
abs/sqrt/div/exp/fmod/log/pow..	絕對值/平方根/除法/指數/求餘/求冪..
cos/sin/asin/atan2/cosh..	相關三角函數
ceil/round/floor/trunc	上取整/四捨五入/下取整/只保留整數部分
clamp(input, min, max)	超過min和max部分截斷
sigmod/tanh..	啟動函數

- 對於很多操作，例如div、mul、pow、fmod等，PyTorch都實現了運算子重載，所以可以直接使用運算子。如a ** 2 等價於torch.pow(a,2), a * 2等價於torch.mul(a,2)。
- 其中clamp(x, min, max)的輸出滿足以下公式：clamp常用在某些需要比較大小的地方，如取一個tensor的每個元素與另一個數的較大值。

```
In [157]:
a = t.arange(0, 6).view(2, 3)
t.cos(a)
Out[157]:
tensor([[1.0000000000, 0.5403022766, -0.4161468446],
        [-0.9899924994, -0.6536436081, 0.2836622000]])
In [158]:
a % 3 # 等價於 t.fmod(a, 3)
Out[158]:
tensor([[ 0.,  1.,  2.],
        [ 0.,  1.,  2.]])
In [159]:
a ** 2 # 等價於 t.pow(a, 2)
Out[159]:
tensor([[  0.,   1.,   4.],
        [  9.,  16.,  25.]])
In [160]:
# 取 a 中的每一個元素與 3 相比較大的一個 (小於 3 的截斷成 3)
print(a)
t.clamp(a, min=3)
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
Out[160]:
tensor([[ 3.,  3.,  3.],
        [ 3.,  4.,  5.]])
In [162]:
b = a.sin_() # 效果同 a = a.sin();b=a ,但是更高效節省顯存
a
Out[162]:
tensor([[0.0000000000, 0.8414709568, 0.9092974067],
        [0.1411200017, -0.7568024993, -0.9589242935]])
```

歸併操作

此類操作會使輸出形狀小於輸入形狀，並可以沿著某一維度進行指定操作。如加法sum，既可以計算整個tensor的和，也可以計算tensor中每一行或每一列的和。

表3-5: 常用歸併操作

函數	功能
mean/sum/median/mode	均值/和/中位數/眾數
norm/dist	範數/距離
std/var	標準差/方差
cumsum/cumprod	累加/累乘

- 以上大多數函數都有一個參數**dim**，用來指定這些操作是在哪個維度上執行的。關於dim(對應於Numpy中的axis)的解釋眾說紛紜，這裡提供一個簡單的記憶方式：

  假設輸入的形狀是(m, n, k)

  如果指定dim=0，輸出的形狀就是(1, n, k)或者(n, k)

  如果指定dim=1，輸出的形狀就是(m, 1, k)或者(m, k)

  如果指定dim=2，輸出的形狀就是(m, n, 1)或者(m, n)

- Size中是否有"1"，取決於參數keepdim，keepdim=True會保留維度1。注意，以上只是經驗總結，並非所有函數都符合這種形狀變化方式，如cumsum。

```
In [75]:
b = t.ones(2, 3)
b.sum(dim = 0, keepdim=True)
Out[75]:
tensor([[ 2.,  2.,  2.]])
In [76]:
# keepdim=False，不保留維度"1"，注意形狀
b.sum(dim=0, keepdim=False)
Out[76]:
tensor([ 2.,  2.,  2.])
In [77]:
b.sum(dim=1)
Out[77]:
tensor([ 3.,  3.])
In [78]:
a = t.arange(0, 6).view(2, 3)
print(a)
a.cumsum(dim=1) # 沿著行累加
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
Out[78]:
tensor([[  0.,   1.,   3.],
        [  3.,   7.,  12.]])
```

比較

比較函數中有一些是逐元素比較，操作類似於逐元素操作，還有一些則類似於歸併操作。

表3-6: 常用比較函數

函數	功能
gt/lt/ge/le/eq/ne	大於/小於/大於等於/小於等於/ 等於/不等
topk	最大的k個數
sort	排序
max/min	比較兩個tensor最大最小值

表中第一行的比較操作已經實現了運算子重載，因此可以使用a>=b、a>b、a!=b、a==b，其返回結果是一個ByteTensor，可用來選取元素。max/min這兩個操作比較特殊，

以max來說，它有以下三種使用情況：

1. t.max(tensor)：返回tensor中最大的一個數
2. t.max(tensor,dim)：指定維上最大的數，返回tensor和下標
3. t.max(tensor1, tensor2): 比較兩個tensor相比較大的元素

比較一個tensor和一個數，可以使用clamp函數。下面舉例說明。

```
In [79]:
a = t.linspace(0, 15, 6).view(2, 3)
a
Out[79]:
tensor([[  0.,   3.,   6.],
        [  9.,  12.,  15.]])
In [80]:
b = t.linspace(15, 0, 6).view(2, 3)
b
Out[80]:
tensor([[ 15.,  12.,   9.],
        [  6.,   3.,   0.]])
In [81]:
a>b
Out[81]:
tensor([[ 0,  0,  0],
        [ 1,  1,  1]], dtype=torch.uint8)
In [82]:
a[a>b] # a 中大於 b 的元素
Out[82]:
tensor([  9.,  12.,  15.])
In [83]:
t.max(a)
Out[83]:
tensor(15.)
```

```
In [84]:
t.max(b, dim=1)
# 第一個返回值的 15 和 6 分別表示第 0 行和第 1 行最大的元素
# 第二個返回值的 0 和 0 表示上述最大的數是該行第 0 個元素
Out[84]:
(tensor([ 15.,   6.]), tensor([ 0,  0]))
In [85]:
t.max(a,b)
Out[85]:
tensor([[ 15.,  12.,   9.],
        [  9.,  12.,  15.]])
In [86]:
# 比較 a 和 10 較大的元素
t.clamp(a, min=10)
Out[86]:
tensor([[ 10.,  10.,  10.],
        [ 10.,  12.,  15.]])
```

線性代數

PyTorch的線性函數主要封裝了Blas和Lapack，其用法和介面都與之類似。

表3-7: 常用的線性代數函數

函 數	功 能
trace	對角線元素之和(矩陣的跡)
diag	對角線元素
triu/tril	矩陣的上三角/下三角，可指定偏移量
mm/bmm	矩陣乘法，batch的矩陣乘法
addmm/addbmm/addmv/addr/badbmm..	矩陣運算
t	轉置
dot/cross	內積/外積
inverse	求逆矩陣
svd	奇異值分解

矩陣的轉置會導致存儲空間不連續，需調用它的.contiguous方法將其轉為連續。

http://pytorch.org/docs/torch.html#blas-and-lapack-operations

```
In [87]:
b = a.t()
b.is_contiguous()
Out[87]:
False
In [88]:
b.contiguous()
Out[88]:
tensor([[  0.,   9.],
        [  3.,  12.],
        [  6.,  15.]])
```

3.1.2 Tensor和Numpy

Tensor和Numpy陣列之間具有很高的相似性，彼此之間的交互操作也非常簡單高效。需要注意的是，Numpy和Tensor共用記憶體。由於Numpy歷史悠久，支持豐富的操作，所以當遇到Tensor不支援的操作時，可先轉成Numpy陣列，處理後再轉回tensor，其轉換開銷很小。

```
In [89]:
import numpy as np
a = np.ones([2, 3],dtype=np.float32)
a
Out[89]:
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
In [90]:
b = t.from_numpy(a)
b
Out[90]:
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
In [91]:
b = t.Tensor(a) # 也可以直接將 numpy 物件傳入 Tensor
b
Out[91]:
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
In [92]:
a[0, 1]=100
b
Out[92]:
tensor([[   1.,  100.,    1.],
        [   1.,    1.,    1.]])
```

```
In [93]:
c = b.numpy() # a, b, c 三個物件共用記憶體
c
Out[93]:
array([[  1., 100.,   1.],
       [  1.,   1.,   1.]], dtype=float32)
```

注意： 當 numpy 的資料類型和 Tensor 的類型不一樣的時候，資料會被複製，不會共用記憶體。

```
In [152]:
a = np.ones([2, 3])
# 注意和上面的a 的區別（dtype 不是 float32）
a.dtype
Out[152]:
dtype('float64')
In [153]:
b = t.Tensor(a) # 此處進行拷貝，不共用記憶體
b.dtype
Out[153]:
torch.float32
In [154]:
c = t.from_numpy(a) # 注意 c 的類型（DoubleTensor）
c
Out[154]:
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]], dtype=torch.float64)
In [155]:
a[0, 1] = 100
b # b 與 a 不共用記憶體，所以即使 a 改變了，b 也不變
Out[155]:
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
In [156]:
c # c 與 a 共用記憶體
Out[156]:
tensor([[  1., 100.,   1.],
        [  1.,   1.,   1.]], dtype=torch.float64)
```

注意： 不論輸入的類型是什麼，t.tensor 都會進行資料拷貝，不會共用記憶體

```
In [99]:
tensor = t.tensor(a)
In [100]:
tensor[0,0]=0
a
Out[100]:
array([[  1., 100.,   1.],
       [1.,1.,1.]])
```

廣播法則

廣播法則(broadcast)是科學運算中經常使用的一個技巧，它在快速執行向量化的同時不會佔用額外的記憶體/顯存。 Numpy的廣播法則定義如下：

1. 讓所有輸入陣列都向其中shape最長的陣列看齊，不足的部分通過在前面加1補齊
2. 兩個陣列要麼在某一個維度的長度一致，要麼其中一個為1，否則不能計算
3. 當輸入陣列的某個維度的長度為1時，計算時沿此維度複製擴充成一樣的形狀

- PyTorch當前已經支持了自動廣播法則，但是筆者還是建議讀者通過以下兩個函數的組合手動實現廣播法則，這樣更直觀，更不易出錯：

1. Unsqueeze或者view，或者tensor[None],：為資料某一維的形狀補1，實現法則1
2. Expand或者expand_as，重複陣列，實現法則3；該操作不會複製陣列，所以不會佔用額外的空間。

- 注意，repeat實現與expand相類似的功能，但是repeat會把相同資料複製多份，因此會佔用額外的空間。

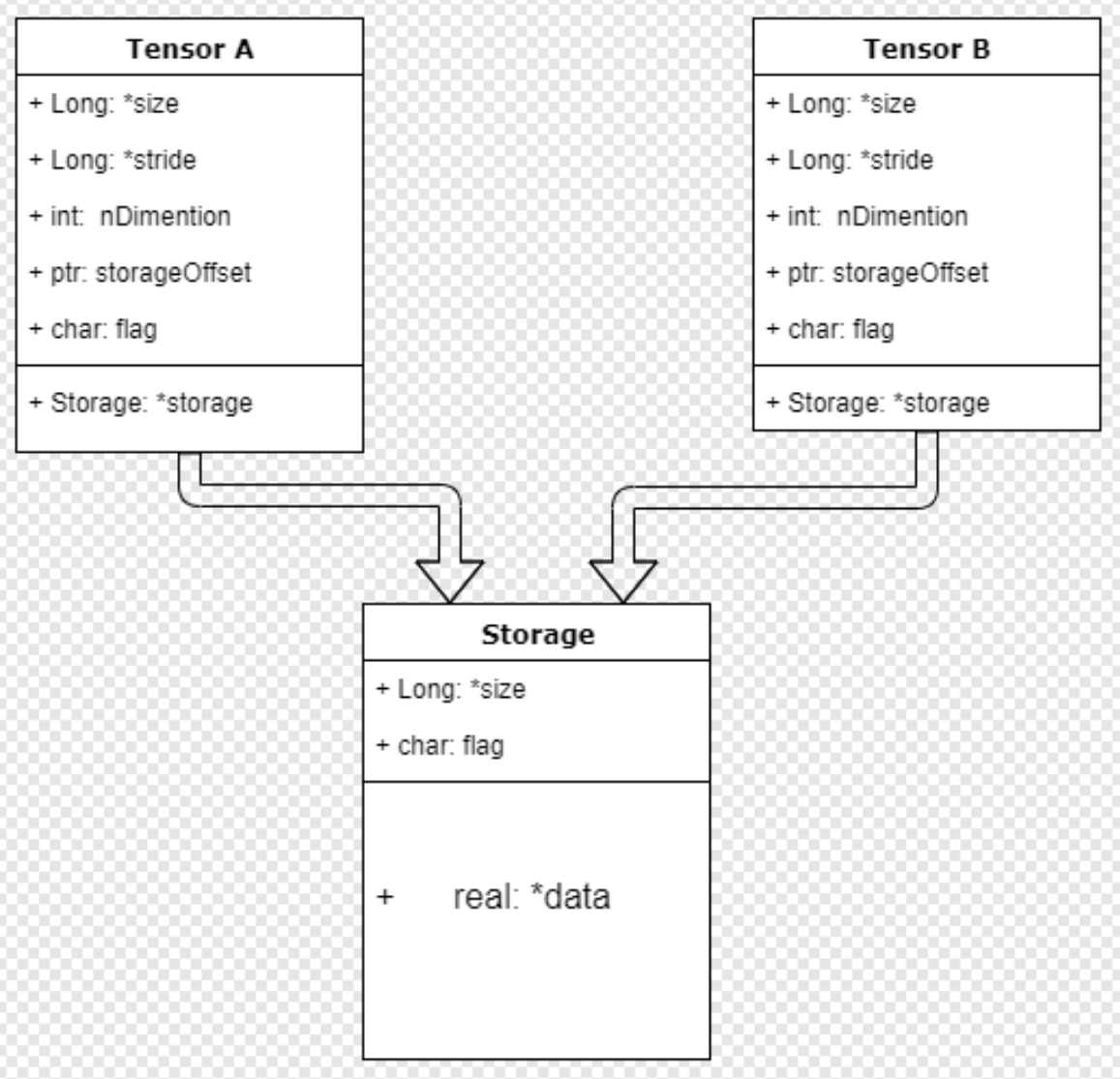
```
In [101]:
a = t.ones(3, 2)
b = t.zeros(2, 3,1)
In [102]:
# 自動廣播法則
# 第一步：a是2維,b是3維，所以先在較小的a前面補1 ，
#                即：a.unsqueeze(0)，a的形狀變成（1，3，2），b的形狀是（2，3，1），
# 第二步:   a和b在第一維和第三維形狀不一樣，其中一個為1 ，
#                可以利用廣播法則擴展，兩個形狀都變成了（2，3，2）
a+b
Out[102]:
tensor([[[ 1.,  1.],
         [ 1.,  1.],
         [ 1.,  1.]],

        [[ 1.,  1.],
         [ 1.,  1.],
         [ 1.,  1.]]])
```

```
In [103]:
# 手動廣播法則
# 或者 a.view(1,3,2).expand(2,3,2)+b.expand(2,3,2)
a[None].expand(2, 3, 2) + b.expand(2,3,2)
Out[103]:
tensor([[[ 1.,  1.],
         [ 1.,  1.],
         [ 1.,  1.]],
        [[ 1.,  1.],
         [ 1.,  1.],
         [ 1.,  1.]]])
In [104]:
# expand不會佔用額外空間，只會在需要的時候才擴充，可極大節省記憶體
e = a.unsqueeze(0).expand(10000000000000, 3,2)
```

3.1.3 內部結構

Tensor的資料結構如下圖所示。



3.1.3 內部結構

- Tensor分為頭資訊區(Tensor)和存儲區(Storage)，資訊區主要保存著tensor的形狀（size）、步長（stride）、資料類型（type）等資訊，而真正的資料則保存成連續陣列。
- 由於資料動輒成千上萬，因此資訊區元素佔用記憶體較少，主要記憶體佔用則取決於tensor中元素的數目，也即存儲區的大小。

一般來說一個tensor有著與之相對應的storage, storage是在data之上封裝的介面，便於使用，而不同tensor的頭資訊一般不同，但卻可能使用相同的資料。下面看兩個例子。

```
In [105]:
a = t.arange(0, 6)
a.storage()
Out[105]:
 0.0
 1.0
 2.0
 3.0
 4.0
 5.0
[torch.FloatStorage of size 6]

In [106]:
b = a.view(2, 3)
b.storage()
Out[106]:
 0.0
 1.0
 2.0
 3.0
 4.0
 5.0
[torch.FloatStorage of size 6]
In [107]:
# 一個物件的 id 值可以看作它在記憶體中的位址
# storage 的記憶體位址一樣，即是同一個 storage
id(b.storage()) == id(a.storage())
Out[107]:
True
```

```
In [108]:
# a 改變，b也隨之改變，因為他們共用 storage
a[1] = 100
b
Out[108]:
tensor([[   0.,  100.,    2.],
        [   3.,    4.,    5.]])
In [109]:
c = a[2:]
c.storage()
Out[109]:
 0.0
 100.0
 2.0
 3.0
 4.0
 5.0
[torch.FloatStorage of size 6]
In [110]:
c.data_ptr(), a.data_ptr() # data_ptr 返回 tensor 首元素的記憶體位址
# 可以看出相差 8，這是因為 2*4=8--相差兩個元素，每個元素占 4 個位元組(float)
Out[110]:
(93894489135160, 93894489135152)
```

```
In [111]:
c[0] = -100 # c[0]的記憶體位址對應 a[2]的記憶體位址
a
Out[111]:
tensor([   0.,  100., -100.,    3.,    4.,    5.])
In [112]:
d = t.Tensor(c.storage())
d[0] = 6666
b
Out[112]:
tensor([[ 6666.,   100.,  -100.],
        [    3.,     4.,     5.]])
In [113]:
# 下麵 4 個 tensor 共用 storage
id(a.storage()) == id(b.storage()) == id(c.storage()) == id(d.storage())
Out[113]:
True
In [114]:
a.storage_offset(), c.storage_offset(), d.storage_offset()
Out[114]:
(0, 2, 0)

In [115]:
e = b[::2, ::2] # 隔 2 行/列取一個元素
id(e.storage()) == id(a.storage())
Out[115]:
True
In [116]:
b.stride(), e.stride()
Out[116]:
((3, 1), (6, 2))
In [117]:
e.is_contiguous()
Out[117]:
False
```

- 可見絕大多數操作並不修改tensor的資料，而只是修改了tensor的頭資訊。這種做法更節省記憶體，同時提升了處理速度。在使用中需要注意。 此外有些操作會導致tensor不連續，這時需調用tensor.contiguous方法將它們變成連續的資料，該方法會使資料複製一份，不再與原來的資料共用storage。
- 另外可以思考一下，之前說過的高級索引一般不共用stroage，而普通索引共用storage，這是為什麼？（提示：普通索引可以通過只修改tensor的offset，stride和size，而不修改storage來實現）

3.1.4 其它有關Tensor的話題

- GPU/CPU
- Tensor可以很隨意的在GPU/CPU上傳輸。使用tensor.cuda(device_id)或者tensor.cpu()。另外一個更通用的方法是tensor.to(device)。

```
In [118]:
a = t.randn(3, 4)
a.device
Out[118]:
device(type='cpu')
In [119]:
if t.cuda.is_available():
    a = t.randn(3,4, device=t.device('cuda:1'))
    # 等價於
    # a.t.randn(3,4).cuda(1)
    # 但是前者更快
    a.device
In [120]:
device = t.device('cpu')
a.to(device)
Out[120]:
tensor([[-0.4362, -0.4324, -0.3985,  0.2730],
        [-0.1356, -0.3448,  1.5713, -0.8333],
        [-0.0089,  1.4516, -0.2062, -0.4085]])
```

注意

儘量使用tensor.to(device), 將device設為一個可配置的參數，這樣可以很輕鬆的使程式同時相容GPU和CPU

資料在GPU之中傳輸的速度要遠快於記憶體(CPU)到顯存(GPU), 所以儘量避免頻繁的在記憶體和顯存中傳輸資料。

持久化

- Tensor的保存和載入十分的簡單，使用t.save和t.load即可完成相應的功能。在save/load時可指定使用的pickle模組，在load時還可將GPU tensor映射到CPU或其它GPU上

```
In [121]:
if t.cuda.is_available():
    a = a.cuda(1) # 把a轉為GPU1上的tensor,
    t.save(a,'a.pth')


    # 載入為b, 存儲於GPU1上(因為保存時tensor就在GPU1上)
    b = t.load('a.pth')
    # 載入為c, 存儲於CPU
    c = t.load('a.pth', map_location=lambda storage, loc: storage)
    # 載入為d, 存儲於GPU0上
    d = t.load('a.pth', map_location={'cuda:1':'cuda:0'})
```

向量化

向量化計算是一種特殊的平行計算方式，相對於一般程式在同一時間只執行一個操作的方式，它可在同一時間執行多個操作，通常是對不同的資料執行同樣的一個或一批指令，或者說把指令應用於一個陣列/向量上。

向量化可極大提高科學運算的效率，Python本身是一門高階語言，使用很方便，但這也意味著很多操作很低效，尤其是for迴圈。在科學計算程式中應當極力避免使用Python原生的for迴圈。

```
In [122]:
def for_loop_add(x, y):
    result = []
    for i,j in zip(x, y):
        result.append(i + j)
    return t.Tensor(result)
In [123]:
x = t.zeros(100)
y = t.ones(100)
%timeit -n 10 for_loop_add(x, y)
%timeit -n 10 x + y
777 µs ± 17 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
The slowest run took 10.00 times longer than the fastest. This could mean that an intermediate
result is being cached.
6.2 µs ± 8.48 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

可見二者有超過幾十倍的速度差距，因此在實際使用中應儘量調用內建函數(buildin-function)，這些函數底層由C/C++實現，能通過執行底層優化實現高效計算。因此在平時寫代碼時，就應養成向量化的思維習慣，千萬避免對較大的tensor進行逐元素遍歷。

此外還有以下幾點需要注意:

1. 大多數t.function都有一個參數out，這時候產生的結果將保存在out指定tensor之中。
2. t.set_num_threads 可以設置PyTorch進行CPU多執行緒平行計算時候所佔用的執行緒數，這個可以用來限制PyTorch所佔用的CPU數目。
3. t.set_printoptions 可以用來設置列印tensor時的數值精度和格式。下面舉例說明。

```
In [124]:
a = t.arange(0, 20000000)
print(a[-1], a[-2]) # 32bit 的 IntTensor 精度有限導致溢出
b = t.LongTensor()
t.arange(0, 20000000, out=b) # 64bit 的 LongTensor 不會溢出
b[-1],b[-2]
tensor(1.6777e+07) tensor(1.6777e+07)
Out[124]:
(tensor(2.0000e+07), tensor(2.0000e+07))
In [125]:
a = t.randn(2,3)
a
Out[125]:
tensor([[ 0.4317,  2.1251,  1.0684],
        [ 1.0297,  1.0137, -0.8691]])
In [126]:
t.set_printoptions(precision=10)
a
Out[126]:
tensor([[0.4316843748, 2.1250586510, 1.0684313774],
        [1.0297036171, 1.0137003660, -0.8690901995]])
```

3.1.5 小試牛刀：線性回歸

線性回歸是機器學習入門知識，應用十分廣泛。線性回歸利用數理統計中回歸分析，來確定兩種或兩種以上變數間相互依賴的定量關係的，其表達形式為$y = wx + b + e$，e為誤差服從均值為0的正態分佈，首先讓我們來確認線性迴歸的損失函數

$$loss = \sum_i^N \frac{1}{2}(y_i - (wx_i + b))^2$$

然後利用隨機梯度下降法更新參數w和b來最小化損失函數，最終學得w和b的數值。

```
In [142]:
import torch as t
%matplotlib inline
from matplotlib import pyplot as plt
from IPython import display

device = t.device('cpu') #如果你想用 gpu，改成 t.device('cuda:0')
In [143]:
# 設置亂數種子，保證在不同電腦上運行時下面的輸出一致
t.manual_seed(1000)

def get_fake_data(batch_size=8):
    ''' 產生亂數據：y=x*2+3，加上了一些雜訊'''
    x = t.rand(batch_size, 1, device=device) * 5
    y = x * 2 + 3 +  t.randn(batch_size, 1, device=device)
    return x, y

In [144]:
# 來看看產生的 x-y 分佈
x, y = get_fake_data(batch_size=16)
plt.scatter(x.squeeze().cpu().numpy(), y.squeeze().cpu().numpy())
Out[144]:
<matplotlib.collections.PathCollection at 0x7fb5f2e5f780>
```