# Torch

## Gather

首先，先给出torch.gather函数的函数定义：

**torch.gather(input, dim, index, out=None) → Tensor**

官方给出的解释是这样的：

沿给定轴dim，将输入索引张量index指定位置的值进行聚合。

                    对一个3维张量，输出可以定义为：

                out[i][j][k] = tensor[index[i][j][k]][j][k] # dim=0

                out[i][j][k] = tensor[i][index[i][j][k]][k] # dim=1

                out[i][j][k] = tensor[i][j][index[i][j][k]] # dim=3

**刚开始看上去有点难以理解，但经过研究之后发现原来这个想表述的很简单，先给出几个代码例子让大家自行体会一下。**

>>> import torch

>>> a = torch.Tensor([[1,2],[3,4]])

>>> a

1 2

3 4

[torch.FloatTensor of size 2x2]

>>> b = torch.gather(a,1,torch.LongTensor([[0,0],[1,0]]))

>>> b

1 1

4 3

[torch.FloatTensor of size 2x2]

>>> b = torch.gather(a,1,torch.LongTensor([[1,0],[1,0]]))

>>> b

2 1

4 3

[torch.FloatTensor of size 2x2]

>>> b = torch.gather(a,1,torch.LongTensor([[1,1],[1,0]]))

>>> b

2 2

4 3

[torch.FloatTensor of size 2x2]

很容易就会发现****torch.gather(input, dim, index, out=**None**)中的dim表示的就是第几维度，在这个二维例子中，如果dim=0，****

****那么它表示的就是你接下来的操作是对于第一维度进行的，也就是行；如果dim=1,那么它表示的就是你接下来的操作是对于第****

****二维度进行的，也就是列。index的大小和input的大小是一样的，他表示的是你所选择的维度上的操作，比如这个例子中****

a = torch.Tensor([[1,2],[3,4]]) b = torch.gather(a,1,torch.LongTensor([[0,0],[1,0]])) 其中， dim=1，表示的是在第二维度上操作。

index = torch.LongTensor([[0,0],[1,0]])，[0,0]就是第一行对应元素的下标，也就是对应的是[1,1]； [1,0]就是第二行对应元素的下标，也就是对应的是[4,3]。

！！！特别注意一下，index的类型必须是LongTensor类型的。

## 转置

torch.t(input, out=None) → Tensor

输入一个矩阵（2维张量），并转置0, 1维。  
可以被视为函数transpose(input, 0, 1)的简写函数。

参数:

* input (Tensor) – 输入张量
* out (Tensor, optional) – 结果张量

>>> x = torch.randn(2, 3)

>>> x

0.4834 0.6907 1.3417

-0.1300 0.5295 0.2321

[torch.FloatTensor of size 2x3]

>>> torch.t(x)

0.4834 -0.1300

0.6907 0.5295

1.3417 0.2321

[torch.FloatTensor of size 3x2]

## Tensor基本操作

截取：

x[0:3, 2:4]

Tensor(t) 与list或numpy中的array互相转化：

t=T.Tensor(list)

t=T.Tensor(np.array)

t=T.from\_numpy(np.array)

list=T.tolist(t)

array=t.numpy()

新建空Tensor和全值Tensor

t=T.empty(t.size())

t=T.empty\_like(t)

t=T.full(t.size(),value)

t=T.full\_like(t,value)

采数据

T.index\_select(t, di, indices) //在第di维上将t的indices抽取出来组成新Tensor。

T.masked\_select(t, mask) //按照0-1Tensor mask的格式筛选t，返回一维Tensor

T.nonzero(t) //输出n×2维Tensor，非零元素的index

观察一个Tensor(t)

t.size() //返回size类型

T.numel(t) //返回总元素个数

t.view(d1,d2,d3....)//维度重整

t.unsqueeze(di) //在di个维度处升维、

t.squeeze(di) //若di维是1，压缩，否则不变。若无参数，压缩所有“1”维

T.cat((t,t,...),di) //按第di的维度按照tuple的格式复制t

T.chunk(t,i,di) //在di维上将t分成i份，最后一份的维度不定（若不能整除）

## expand(\*sizes)

返回tensor的一个新视图，单个维度扩大为更大的尺寸。 tensor也可以扩大为更高维，新增加的维度将附在前面。 扩大tensor不需要分配新内存，只是仅仅新建一个tensor的视图，其中通过将stride设为0，一维将会扩展位更高维。任何一个一维的在不分配新内存情况下可扩展为任意的数值。

参数： - sizes(torch.Size or int...)-需要扩展的大小

例：

>>> x = torch.Tensor([[1], [2], [3]])>>> x.size()

torch.Size([3, 1])>>> x.expand(3, 4)

1 1 1 1

2 2 2 2

3 3 3 3

[torch.FloatTensor of size 3x4]

## 常用操作

**生成一个Tensor(t)**

可以与list或numpy中的array互相转化：

t=T.Tensor(list)

t=T.Tensor(np.array)

t=T.from\_numpy(np.array)

list=T.tolist(t)

array=t.numpy()

将numpy矩阵转换为Tensor张量

sub\_ts = torch.from\_numpy(sub\_img) #sub\_img为numpy类型

4.将Tensor张量转化为numpy矩阵

sub\_np1 = sub\_ts.numpy() #sub\_ts为tensor张量

5.将numpy转换为Variable

sub\_va = Variable(torch.from\_numpy(sub\_img))

1. 将Variable张量转化为numpy
2. sub\_np2 = sub\_va.data.numpy()

Tensor和np.array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。 所以有些PyTorch没有但numpy有的操作可以以很小的代价进行“曲线救国“   
指定维度的Tensor：

t=T.Tensor(d1,d2,...)

t=T.ones(d1,d2,...)

t=T.zeros(d1,d2,...)

t=T.eye(d1,d2)

由于Tensor的size是一个专门的类型，所以上面的维度也可以由size类型指定。(eye()的参数仅包括1-2个int)

t=T.Tensor(t.size())

t=T.ones(t.size()) //t=T.ones\_like(t)

t=T.zeros(t.size()) //t=T.zeros\_like(t)

均分区间生成Tensor、

t=T.arange(m,n,step\_length) //[m,n)中m开始以步长step\_length生成

t=T.range(m,n,step\_length) //[m,n-1]中m开始以步长step\_length生成

t=T.linspace(m,n,step\_num) //[m,n]中以m为首项，n为末项，均分区间为step\_num段

随机化生成

t=T.rand(t.size()) //均匀分布

t=T.randn(t.size()) //标准正态分布

t=T.normal(mean,std) //size同t.Tensor()，每个数以对应的均值mean和标准差std[i,j,...]正态采样。

新建空Tensor和全值Tensor

t=T.empty(t.size())

t=T.empty\_like(t)

t=T.full(t.size(),value)

t=T.full\_like(t,value)

指定不同类型数据的Tensor

| **Data type** | **CPU tensor** | **GPU tensor** |
| --- | --- | --- |
| 32-bit floating point | T.FloatTensor | T.cuda.FloatTensor |
| 64-bit floating point | T.DoubleTensor | T.cuda.DoubleTensor |
| 16-bit floating point | T.HalfTensor | T.cuda.HalfTensor |
| 8-bit integer (unsigned) | T.ByteTensor | T.cuda.ByteTensor |
| 8-bit integer (signed) | T.CharTensor | T.cuda.CharTensor |
| 16-bit integer (signed) | T.ShortTensor | T.cuda.ShortTensor |
| 32-bit integer (signed) | T.IntTensor | T.cuda.IntTensor |
| 64-bit integer (signed) | T.LongTensor | T.cuda.LongTensor |

**观察一个Tensor(t)**

t.size() //返回size类型

T.numel(t) //返回总元素个数

t.view(d1,d2,d3....)//维度重整

t.unsqueeze(di) //在di个维度处升维、

t.squeeze(di) //若di维是1，压缩，否则不变。若无参数，压缩所有“1”维

T.cat((t,t,...),di) //按第di的维度按照tuple的格式复制t

T.chunk(t,i,di) //在di维上将t分成i份，最后一份的维度不定（若不能整除）

采数据

T.index\_select(t, di, indices) //在第di维上将t的indices抽取出来组成新Tensor。

T.masked\_select(t, mask) //按照0-1Tensor mask的格式筛选t，返回一维Tensor

T.nonzero(t) //输出n×2维Tensor，非零元素的index

指定不同类型数据的Tensor

t.size() //返回size类型

T.numel(t) //返回总元素个数

t.view(d1,d2,d3....)//维度重整

t.unsqueeze(di) //在di个维度处升维、

t.squeeze(di) //若di维是1，压缩，否则不变。若无参数，压缩所有“1”维

T.cat((t,t,...),di) //按第di的维度按照tuple的格式复制t

T.chunk(t,i,di) //在di维上将t分成i份，最后一份的维度不定（若不能整除）

一些数学功能类似于numpy，这里就不再赘述了，边用边整理吧。

## t.cat

A=torch.ones(2,3)   
D=2\*torch.ones(2,4) #2x4的张量（矩阵）

C=torch.cat((A,D),1)#按维数1（列）拼接

>>> C

tensor([[ 1., 1., 1., 2., 2., 2., 2.],

[ 1., 1., 1., 2., 2., 2., 2.]])

## .numpy()和.item()

如果一个tensor只有一个元素，那么可以使用.item()方法取出这个元素作为普通的python数字。

tensor和numpy之间的转化与0.3.0版本一样，由tensor向numpy转化使用例子如下，注意这种操作的结果是tensor和numpy始终同变化：

a = torch.ones(5)

print(a)

Out: tensor([ 1., 1., 1., 1., 1.])

b = a.numpy()

print(b)

Out: [1. 1. 1. 1. 1.]

1. add\_(1)
2. print(a)

print(b)

Out: tensor([ 2., 2., 2., 2., 2.])

           [2. 2. 2. 2. 2.]

由numpy向tensor转化的例子如下，二者依然是同步变化的（即改变其中一个，另一个自动变化）：

import numpy as np

a = np.ones(5)

b = torch.from\_numpy(a)

np.add(a, 1, out=a)

print(a)

print(b)

Out: [2. 2. 2. 2. 2.]

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

## 广播

****一般语义****

如果遵守以下规则，则两个张量是“可播放的”：

* 每个张量至少有一个维度。
* 迭代尺寸大小时，从尾随尺寸开始，尺寸大小必须相等，其中一个为1，或者其中一个不存在。

例如：

****>>>**** x=torch.empty(5,7,3)****>>>**** y=torch.empty(5,7,3)# same shapes are always broadcastable (i.e. the above rules always hold)

****>>>**** x=torch.empty((0,))****>>>**** y=torch.empty(2,2)# x and y are not broadcastable, because x does not have at least 1 dimension

# can line up trailing dimensions****>>>**** x=torch.empty(5,3,4,1)****>>>**** y=torch.empty( 3,1,1)# x and y are broadcastable.# 1st trailing dimension: both have size 1# 2nd trailing dimension: y has size 1# 3rd trailing dimension: x size == y size# 4th trailing dimension: y dimension doesn't exist

# but:****>>>**** x=torch.empty(5,2,4,1)****>>>**** y=torch.empty( 3,1,1)# x and y are not broadcastable, because in the 3rd trailing dimension 2 != 3

如果两个张量x，y是“broadcastable”，所得到的张量大小的计算方法如下：

* 如果尺寸的数量x和y不相等，则在尺寸较小的张量的前面加1，使它们的长度相等。
* 然后，对于每个维度大小，生成的维度大小是该维度的大小x和y沿该维度的最大值 。

例如：

*# can line up trailing dimensions to make reading easier*>>> x=torch.empty(5,1,4,1)>>> y=torch.empty( 3,1,1)>>> (x+y).size()

torch.Size([5, 3, 4, 1])

*# but not necessary:*>>> x=torch.empty(1)>>> y=torch.empty(3,1,7)>>> (x+y).size()

torch.Size([3, 1, 7])

>>> x=torch.empty(5,2,4,1)>>> y=torch.empty(3,1,1)>>> (x+y).size()RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimension 1

****就地语义****

一个复杂因素是就地操作不允许就地张量由于广播而改变形状。

例如：

****>>>**** x=torch.empty(5,3,4,1)****>>>**** y=torch.empty(3,1,1)****>>>**** (x.add\_(y)).size()torch.Size([5, 3, 4, 1])

# but:****>>>**** x=torch.empty(1,3,1)****>>>**** y=torch.empty(3,1,7)****>>>**** (x.add\_(y)).size()RuntimeError: The expanded size of the tensor (1) must match the existing size (7) at non-singleton dimension 2.

## torch.manual\_seed()

torch.manual\_seed(args.seed) #为CPU设置种子用于生成随机数，以使得结果是确定的   
if args.cuda:   
torch.cuda.manual\_seed(args.seed)#为当前GPU设置随机种子；如果使用多个GPU，应该使用torch.cuda.manual\_seed\_all()为所有的GPU设置种子。

## 模型建立的几种方法

**第一种方法**

# Method 1 -----------------------------------------

class Net1(torch.nn.Module):

def \_\_init\_\_(self):

super(Net1, self).\_\_init\_\_()

self.conv1 = torch.nn.Conv2d(3, 32, 3, 1, 1)

self.dense1 = torch.nn.Linear(32 \* 3 \* 3, 128)

self.dense2 = torch.nn.Linear(128, 10)

def forward(self, x):

x = F.max\_pool2d(F.relu(self.conv(x)), 2)

x = x.view(x.size(0), -1)

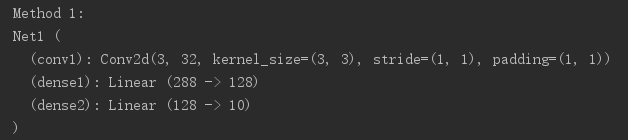
x = F.relu(self.dense1(x))

x = self.dense2(x)

return x

print("Method 1:")

model1 = Net1()print(model1)



这种方法比较常用，早期的教程通常就是使用这种方法。

**第二种方法**

# Method 2 ------------------------------------------class Net2(torch.nn.Module):

def \_\_init\_\_(self):

super(Net2, self).\_\_init\_\_()

self.conv = torch.nn.Sequential(

torch.nn.Conv2d(3, 32, 3, 1, 1),

torch.nn.ReLU(),

torch.nn.MaxPool2d(2))

self.dense = torch.nn.Sequential(

torch.nn.Linear(32 \* 3 \* 3, 128),

torch.nn.ReLU(),

torch.nn.Linear(128, 10)

)

def forward(self, x):

conv\_out = self.conv1(x)

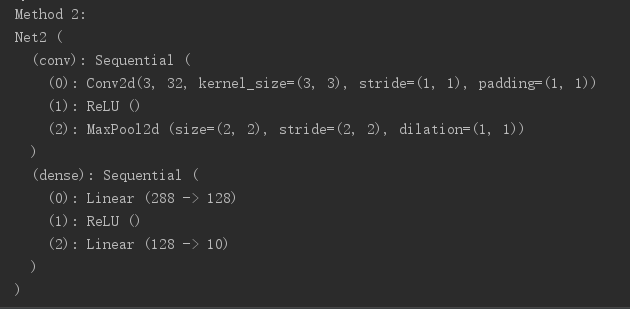
res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 2:")

model2 = Net2()print(model2)



这种方法利用torch.nn.Sequential（）容器进行快速搭建，模型的各层被顺序添加到容器中。缺点是每层的编号是默认的阿拉伯数字，不易区分。

**第三种方法：**

# Method 3 -------------------------------class Net3(torch.nn.Module):

def \_\_init\_\_(self):

super(Net3, self).\_\_init\_\_()

self.conv=torch.nn.Sequential()

self.conv.add\_module("conv1",torch.nn.Conv2d(3, 32, 3, 1, 1))

self.conv.add\_module("relu1",torch.nn.ReLU())

self.conv.add\_module("pool1",torch.nn.MaxPool2d(2))

self.dense = torch.nn.Sequential()

self.dense.add\_module("dense1",torch.nn.Linear(32 \* 3 \* 3, 128))

self.dense.add\_module("relu2",torch.nn.ReLU())

self.dense.add\_module("dense2",torch.nn.Linear(128, 10))

def forward(self, x):

conv\_out = self.conv1(x)

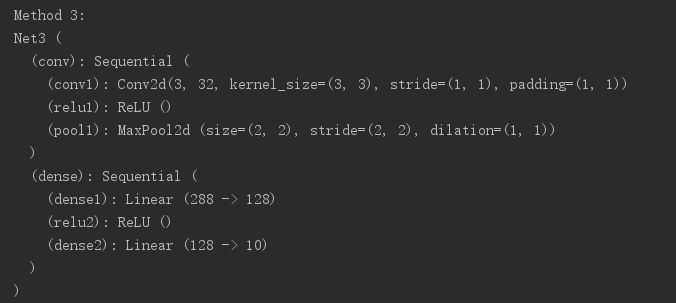
res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 3:")

model3 = Net3()print(model3)



这种方法是对第二种方法的改进：通过add\_module()添加每一层，并且为每一层增加了一个单独的名字。

**第四种方法：**

# Method 4 ------------------------------------------class Net4(torch.nn.Module):

def \_\_init\_\_(self):

super(Net4, self).\_\_init\_\_()

self.conv = torch.nn.Sequential(

OrderedDict(

[

("conv1", torch.nn.Conv2d(3, 32, 3, 1, 1)),

("relu1", torch.nn.ReLU()),

("pool", torch.nn.MaxPool2d(2))

]

))

self.dense = torch.nn.Sequential(

OrderedDict([

("dense1", torch.nn.Linear(32 \* 3 \* 3, 128)),

("relu2", torch.nn.ReLU()),

("dense2", torch.nn.Linear(128, 10))

])

)

def forward(self, x):

conv\_out = self.conv1(x)

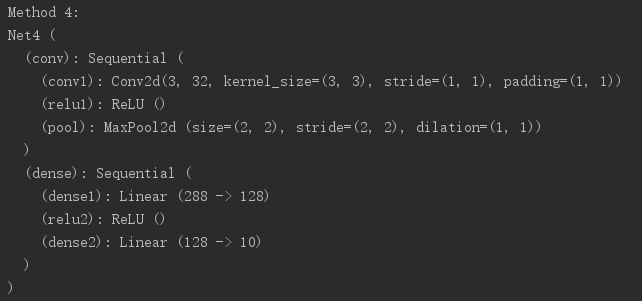
res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 4:")

model4 = Net4()print(model4)



是第三种方法的另外一种写法，通过字典的形式添加每一层，并且设置单独的层名称。

完整代码：

import torchimport torch.nn.functional as Ffrom collections import OrderedDict

# Method 1 -----------------------------------------

class Net1(torch.nn.Module):

def \_\_init\_\_(self):

super(Net1, self).\_\_init\_\_()

self.conv1 = torch.nn.Conv2d(3, 32, 3, 1, 1)

self.dense1 = torch.nn.Linear(32 \* 3 \* 3, 128)

self.dense2 = torch.nn.Linear(128, 10)

def forward(self, x):

x = F.max\_pool2d(F.relu(self.conv(x)), 2)

x = x.view(x.size(0), -1)

x = F.relu(self.dense1(x))

x = self.dense2()

return x

print("Method 1:")

model1 = Net1()print(model1)

# Method 2 ------------------------------------------class Net2(torch.nn.Module):

def \_\_init\_\_(self):

super(Net2, self).\_\_init\_\_()

self.conv = torch.nn.Sequential(

torch.nn.Conv2d(3, 32, 3, 1, 1),

torch.nn.ReLU(),

torch.nn.MaxPool2d(2))

self.dense = torch.nn.Sequential(

torch.nn.Linear(32 \* 3 \* 3, 128),

torch.nn.ReLU(),

torch.nn.Linear(128, 10)

)

def forward(self, x):

conv\_out = self.conv1(x)

res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 2:")

model2 = Net2()print(model2)

# Method 3 -------------------------------class Net3(torch.nn.Module):

def \_\_init\_\_(self):

super(Net3, self).\_\_init\_\_()

self.conv=torch.nn.Sequential()

self.conv.add\_module("conv1",torch.nn.Conv2d(3, 32, 3, 1, 1))

self.conv.add\_module("relu1",torch.nn.ReLU())

self.conv.add\_module("pool1",torch.nn.MaxPool2d(2))

self.dense = torch.nn.Sequential()

self.dense.add\_module("dense1",torch.nn.Linear(32 \* 3 \* 3, 128))

self.dense.add\_module("relu2",torch.nn.ReLU())

self.dense.add\_module("dense2",torch.nn.Linear(128, 10))

def forward(self, x):

conv\_out = self.conv1(x)

res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 3:")

model3 = Net3()print(model3)

# Method 4 ------------------------------------------class Net4(torch.nn.Module):

def \_\_init\_\_(self):

super(Net4, self).\_\_init\_\_()

self.conv = torch.nn.Sequential(

OrderedDict(

[

("conv1", torch.nn.Conv2d(3, 32, 3, 1, 1)),

("relu1", torch.nn.ReLU()),

("pool", torch.nn.MaxPool2d(2))

]

))

self.dense = torch.nn.Sequential(

OrderedDict([

("dense1", torch.nn.Linear(32 \* 3 \* 3, 128)),

("relu2", torch.nn.ReLU()),

("dense2", torch.nn.Linear(128, 10))

])

)

def forward(self, x):

conv\_out = self.conv1(x)

res = conv\_out.view(conv\_out.size(0), -1)

out = self.dense(res)

return out

print("Method 4:")

model4 = Net4()print(model4)

[IMG_270](javascript:void(0);)

## torch.optim.lr\_scheduler

torch.optim.lr\_scheduler 中提供了基于多种epoch数目调整学习率的方法.

torch.optim.lr\_scheduler.ReduceLROnPlateau  :允许基于一些验证测量对学习率进行动态的下降

class torch.optim.lr\_scheduler.LambdaLR(optimizer,lr\_lambda,last\_epoch=-1)

将每一个参数组的学习率设置为初始学习率lr的某个函数倍.当last\_epoch=-1时,设置初始学习率为lr.

参数:

        optimizer(Optimizer对象)--优化器

        lr\_lambda(是一个函数,或者列表(list))--

                                 当是一个函数时,需要给其一个整数参数,使其计算出一个乘数因子,用于调整学习率,通常该输入参数是epoch数目

                                 或者是一组上面的函数组成的列表,

        last\_epoch(int类型):最后一次epoch的索引,默认为-1.

示例:

# 假设有两组个参数组

lambda1 = lambda epoch:epoch // 30

lanbda2 = lambda epoch:0.95\*\*epoch

scheduler = LambdaLR(optimizer,lr\_lambda=[lambdda1,lambda2])

for epoch in range(100):

scheduler.step()

train(...)

validate(...)

class torch.optim.lr\_scheduler.StepLR(optimizer,step\_size,gamma=0.1,last\_epoch=-1)

设置每个参数组的学习率为IMG_256,    IMG_257      ,当last\_epoch=-1时,令lr=lr

参数:

        optimizer(Optimizer对象)--优化器

         step\_size(整数类型): 调整学习率的步长,每过step\_size次,更新一次学习率

        gamma(float 类型):学习率下降的乘数因子

        last\_epoch(int类型):最后一次epoch的索引,默认为-1.

示例:

>>> # Assuming optimizer uses lr = 0.05 for all groups

>>> # lr = 0.05 if epoch < 30

>>> # lr = 0.005 if 30 <= epoch < 60

>>> # lr = 0.0005 if 60 <= epoch < 90

>>> # ...

>>> scheduler = StepLR(optimizer, step\_size=30, gamma=0.1)

>>> for epoch in range(100):

>>> scheduler.step()

>>> train(...)

>>> validate(...)

## 查看模型中间结果

### 创建一个新的模型，手动删去不需要的层

new\_model = AlexNet().cuda()

pretrained\_dict = orig\_model.state\_dict()

model\_dict = new\_model.state\_dict()

# remove the key in pretrained\_dict that do not belong the model\_dict

pretrained\_dicted = {k: v for k, v in pretrained\_dict.items() if k in model\_dict}

# update the model\_dict

model\_dict.update(pretrained\_dicted)

new\_model.load\_state\_dict(model\_dict)

这种方法的优点是可以进行多处删减，实现起来比较清晰。缺点就是实现的过程比较复杂。当模型较大时，效率比较低。而且模型的参数和名称必须与原来的模型对应。

### 直接通过索引删掉相应的层

# remove last fully-connected layer

# the number -2 indicate how many layers should be removed

alexnet.classifier = nn.Sequential(\*list(alexnet.classifier.children())[:-2])

print(alexnet)

这种方法的优点是速度快，方便，缺点是要删除掉某些层，不易实现多处修改。适用于剔除最后几层。

### 用hook监控查看

def get\_features\_hook(self, input, output):

print("hook",output.data.cpu().numpy().shape)

handle=alexnet.classifier[4].register\_forward\_hook(get\_features\_hook)

这种方法的有点是不用改变原有模型的结构，可以实现任意地点的精准查看。缺点是每次调用可能会带来时间上的消耗。

handle.remove()

最后可以通过handle句柄删除hook，这里的handle可以是任意名称。

### hook的原理

再看 nn.Module 的\_\_call\_\_方法（被阉割了，只留下需要关注的部分）：

def \_\_call\_\_(self, \*input, \*\*kwargs):

result = self.forward(\*input, \*\*kwargs)

for hook in self.\_forward\_hooks.values():

#将注册的hook拿出来用

hook\_result = hook(self, input, result)

return result

## nn.Sequential

torch.nn.Sequential是一个Sequential容器，模块将按照构造函数中传递的顺序添加到模块中。另外，也可以传入一个有序模块。 为了更容易理解，官方给出了一些案例：

# Sequential使用实例

model = nn.Sequential(

nn.Conv2d(1,20,5),

nn.ReLU(),

nn.Conv2d(20,64,5),

nn.ReLU()

)

# Sequential with OrderedDict使用实例

model = nn.Sequential(OrderedDict([

('conv1', nn.Conv2d(1,20,5)),

('relu1', nn.ReLU()),

('conv2', nn.Conv2d(20,64,5)),

('relu2', nn.ReLU())

]))

torch.nn.Sequential快速搭建神经网络

为了方便比较，我们先用普通方法搭建一个神经网络。

class Net(torch.nn.Module):

def \_\_init\_\_(self, n\_feature, n\_hidden, n\_output):

super(Net, self).\_\_init\_\_()

self.hidden = torch.nn.Linear(n\_feature, n\_hidden)

self.predict = torch.nn.Linear(n\_hidden, n\_output)

def forward(self, x):

x = F.relu(self.hidden(x))

x = self.predict(x)

return x

net1 = Net(1, 10, 1)

上面class继承了一个torch中的神经网络结构, 然后对其进行了修改；接下来我们来使用torch.nn.Sequential来快速搭建一个神经网络。

net2 = torch.nn.Sequential(

torch.nn.Linear(1, 10),

torch.nn.ReLU(),

torch.nn.Linear(10, 1)

)

我们来打印一下2个神经网络的数据，查看区别:

print(net1)"""

Net (

(hidden): Linear (1 -> 10)

(predict): Linear (10 -> 1)

)

"""

print(net2)"""

Sequential (

(0): Linear (1 -> 10)

(1): ReLU ()

(2): Linear (10 -> 1)

)

"""

我们可以发现，使用torch.nn.Sequential会自动加入激励函数, 但是 net1 中, 激励函数实际上是在 forward() 功能中才被调用的.

torch.nn.Sequential与torch.nn.Module区别与选择

使用torch.nn.Module，我们可以根据自己的需求改变传播过程，如RNN等

如果你需要快速构建或者不需要过多的过程，直接使用torch.nn.Sequential即可。

## Bug

### 动态图

作者：陈诚  
链接：https://www.zhihu.com/question/67209417/answer/344752405  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

**记录loss信息的时候直接使用了输出的Variable。**

应该不止我经历过这个吧...

久久不用又会不小心掉到这个坑里去...

for data, label in trainloader:

......

out = model(data)

loss = criterion(out, label)

loss\_sum += loss # <--- 这里

......

**运行着就发现显存炸了**

观察了一下发现随着每个batch显存消耗在不断增大..

参考了别人的代码发现那句loss一般是这样写 /(ㄒoㄒ)/~~

loss\_sum += loss.data[0]

**这是因为输出的loss的数据类型是Variable。**

而PyTorch的动态图机制就是通过Variable来构建图。主要是使用Variable计算的时候，会记录下新产生的Variable的运算符号，在反向传播求导的时候进行使用。

**如果这里直接将loss加起来，系统会认为这里也是计算图的一部分，也就是说网络会一直延伸变大~那么消耗的显存也就越来越大~~**

总之使用Variable的数据时候要非常小心。不是必要的话尽量使用Tensor来进行计算..

题外话

想更多感受动态图的话，可以通过Variable的grad\_fun来观察到该Variable是通过什么运算得到的（前提是前面的Variable的required\_grad置为True）。

大概是这样

>> z = x + y

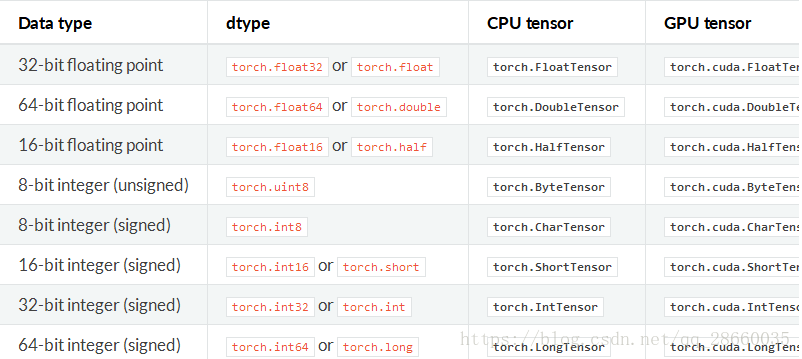
>> z.grad\_fn

out:

<AddBackward1 at 0x107286240>

## 显存消耗计算

先看看我们使用的pytorch数据格式：



平时训练中使用的多是float32 和 int32。

32位的单精度浮点型占用空间为4B，

那么一个batch在网络开始比如说是16×3×224×224，那么所占用的显存也就是16×3×224×224×4B = 9.1875MB

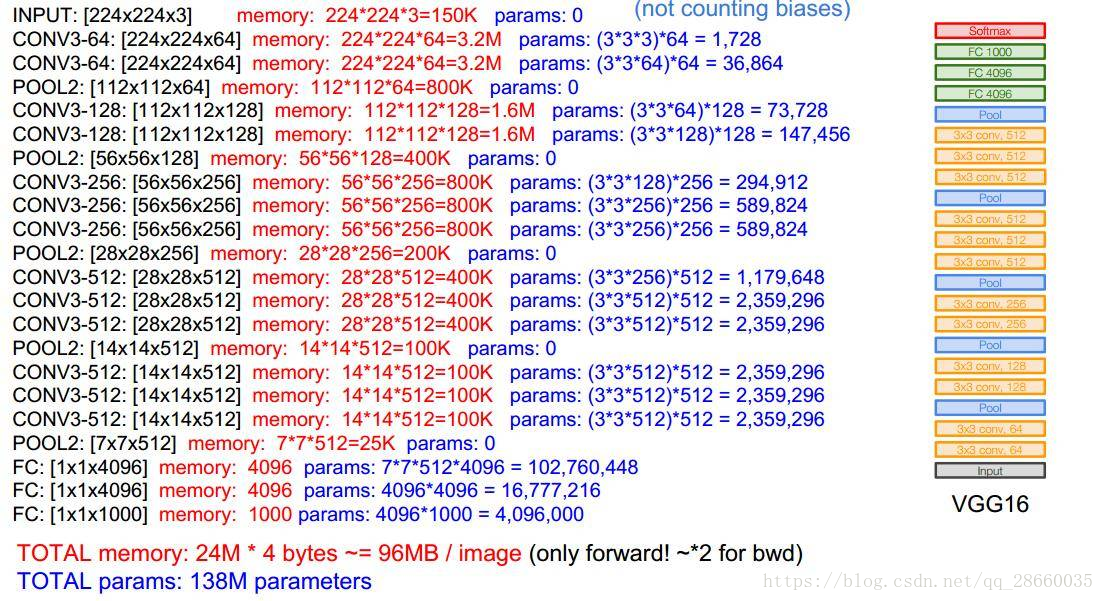
到了网络后期比如说是16×512×14\*14，所占用的显存也就是16×512×14×14×4B = 6.125MB

即使是256的batch\_size，也就是147MB，整个网络如果是19层，为2.728GB，并没有到咱们至少8G的显存。

显存消耗的幕后黑手其实是神经网络中的中间变量以及使用optimizer算法时产生的巨量的中间参数。

显存占用 = 模型参数 + 计算产生的中间变量

以VGG16为例：



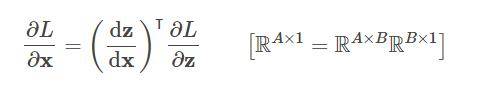
原文博主注意到上图中在计算的时候默认的数据格式是8-bit而不是32-bit，所以最后的结果要乘上一个4，即552mb。

其实只要一计算，就可以知道当batch\_size是256时，中间变量所产生的参数量是有多庞大。。。

反向传播时，中间变量+原来保存的中间变量，存储量会翻倍。

而且有些适用于移动端的网络mobilenet等，计算量是变少了，但对显存占用变大了，原因就是中间参数存储增加了。

我们在backward的时候需要保存下来的中间值。输出是L，然后输入x，我们在backward的时候要求L对x的梯度，这个时候就需要在计算链L和x中间的z：



dz/dx这个中间值当然要保留下来以用于计算，所以粗略估计，backward的时候中间变量的占用了是forward的两倍！

优化器和动量

要注意，优化器也会占用我们的显存！

为什么,看这个式子:

IMG_257

上式是典型的SGD随机下降法的总体公式，权重W在进行更新的时候，会产生保存中间变量IMG_258，也就是在优化的时候，模型中的params参数所占用的显存量会翻倍。

当然这只是SGD优化器，其他复杂的优化器如果在计算时需要的中间变量多的时候，就会占用更多的内存。

## 网络的参数量

params = list(model.named\_parameters())

(name, param) = params[28]

## 点相乘、矩阵相乘

一，对应点相乘，x.mul(y) ，即点乘操作，点乘不求和操作，又可以叫作Hadamard product；点乘再求和，即为卷积

data = [[1,2], [3,4], [5, 6]]

tensor = torch.FloatTensor(data)

tensor

Out[27]:

tensor([[ 1., 2.],

[ 3., 4.],

[ 5., 6.]])

tensor.mul(tensor)

Out[28]:

tensor([[ 1., 4.],

[ 9., 16.],

[ 25., 36.]])

二，矩阵相乘，x.mm(y) ， 矩阵大小需满足： (i, n)x(n, j)

tensor

Out[31]:

tensor([[ 1., 2.],

[ 3., 4.],

[ 5., 6.]])

tensor.mm(tensor.t()) # t()是转置

Out[30]:

tensor([[ 5., 11., 17.],

[ 11., 25., 39.],

[ 17., 39., 61.]])

## Backward

torch.nn.Parameter和from torch.autograd import Variable定义的变量默认有梯度。直接torch.zero的变量没有梯度

torch.autograd.Variable是Autograd的核心类，它封装了Tensor，并整合了反向传播的相关实现

Varibale包含三个属性：

* data：存储了Tensor，是本体的数据
* grad：保存了data的梯度，本事是个Variable而非Tensor，与data形状一致
* grad\_fn：指向Function对象，用于反向传播的梯度计算之用

### 带参数的backward

jacobian矩阵

IMG_256

对其求导：

IMG_257

k.backward(parameters)接受的参数parameters必须要和k的大小一模一样，然后作为k的系数传回去，backward里传入的参数是每次求导的一个**系数。**

# backward on non-scalar output

m = v(t.FloatTensor([[2, 3]]), requires\_grad=True)

n = v(t.zeros(1, 2))

n[0, 0] = m[0, 0] \*\* 2

n[0, 1] = m[0, 1] \*\* 3

n.backward(t.FloatTensor([[1, 1]]))

print('\*'\*10)

print('=====non scalar output======')

print('input')

print(m.data)

print('input gradients are')

print(m.grad.data)

k.backward(parameters)接受的参数parameters必须要和k的大小一模一样，然后作为k的系数传回去，backward里传入的参数是每次求导的一个**系数。**

### 求jacobian行列式

# jacobian

j = t.zeros(2 ,2)

k = v(t.zeros(1, 2))

m.grad.data.zero\_()

k[0, 0] = m[0, 0] \*\* 2 + 3 \* m[0 ,1]

k[0, 1] = m[0, 1] \*\* 2 + 2 \* m[0, 0]# [1, 0] dk0/dm0, dk1/dm0

k.backward(t.FloatTensor([[1, 0]]), retain\_variables=True) # 需要两次反向求导

j[:, 0] = m.grad.data

m.grad.data.zero\_()# [0, 1] dk0/dm1, dk1/dm1

k.backward(t.FloatTensor([[0, 1]]))

j[:, 1] = m.grad.dataprint('jacobian matrix is')print(j)

## **data**

|  |  |
| --- | --- |
| 1  2  3  4  5 | import torch as t  from torch.autograd import Variable    x = Variable(t.ones(2, 2), requires\_grad = True)  x  # 实际查询的是.data，是个Tensor |

实际上查询x和查询x.data返回结果一致，

Variable containing:

 1 1

 1 1

 [torch.FloatTensor of size 2x2]

[回到顶部](https://www.cnblogs.com/hellcat/p/8439055.html" \l "_labelTop)

## **梯度求解**

构建一个简单的方程:y = x[0,0] + x[0,1] + x[1,0] + x[1,1]，Variable的运算结果也是Variable，但是，中间结果反向传播中不会被求导()

这和TensorFlow不太一致，TensorFlow中中间运算果数据结构均是Tensor，

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | y = x.sum()    y  """  　　Variable containing:  　　 4  　　[torch.FloatTensor of size 1]  """ |

可以查看目标函数的.grad\_fn方法，它用来求梯度，

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | y.grad\_fn  """  　　　　<SumBackward0 at 0x18bcbfcdd30>  """    y.backward()  # 反向传播  x.grad  # Variable的梯度保存在Variable.grad中  """  　　Variable containing:  　　 1  1  　　 1  1  　　[torch.FloatTensor of size 2x2]  """ |

grad属性保存在Variable中，新的梯度下来会进行累加，可以看到再次求导后结果变成了2，

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | y.backward()  x.grad  # 可以看到变量梯度是累加的  """      Variable containing:       2  2       2  2      [torch.FloatTensor of size 2x2]  """ |

所以要归零，

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | x.grad.data.zero\_()  # 归零梯度，注意，在torch中所有的inplace操作都是要带下划线的，虽然就没有.data.zero()方法    """   0  0   0  0  [torch.FloatTensor of size 2x2]  """ |

## backward带多个参数

jacobian矩阵

IMG_256

对其求导：

IMG_257

k.backward(parameters)接受的参数parameters必须要和k的大小一模一样，然后作为k的系数传回去，backward里传入的参数是每次求导的一个系数。

m = v(t.FloatTensor([[2, 3]]), requires\_grad=True)

# jacobian

j = t.zeros(2 ,2)

k = v(t.zeros(1, 2))

m.grad.data.zero\_()

k[0, 0] = m[0, 0] \*\* 2 + 3 \* m[0 ,1]

k[0, 1] = m[0, 1] \*\* 2 + 2 \* m[0, 0]# [1, 0] dk0/dm0, dk1/dm0

k.backward(t.FloatTensor([[1, 0]]), retain\_variables=True) # 需要两次反向求导

j[:, 0] = m.grad.data

m.grad.data.zero\_()# [0, 1] dk0/dm1, dk1/dm1

k.backward(t.FloatTensor([[0, 1]]))

j[:, 1] = m.grad.dataprint('jacobian matrix is')print(j)

### 疑问：

## [指定的GPU](https://www.cnblogs.com/darkknightzh/p/6836568.html)

直接终端中设定：

CUDA\_VISIBLE\_DEVICES=1 python my\_script.py

python代码中设定：

import os

os.environ["CUDA\_VISIBLE\_DEVICES"] = "2"

使用函数 set\_device

import torch

torch.cuda.set\_device(id)

## PyTorch 常用方法总结

涉及的方法有下面几种

torch.cat() torch.Tensor.expand()

torch.squeeze() torch.Tensor.repeat()

torch.Tensor.narrow() torch.Tensor.view()

torch.Tensor.resize\_() torch.Tensor.permute()

1

2

3

4

5

6

7

拼接张量

torch.cat(seq, dim=0, out=None) → Tensor

在指定的维度dim上对序列seq进行连接操作。

参数：

seq (sequence of Tensors) - Python序列或相同类型的张量序列

dim (int, optional) - 沿着此维度连接张量

out (Tensor, optional) - 输出参数

x = torch.randn(2, 3)

x

-0.5866 -0.3784 -0.1705

-1.0125 0.7406 -1.2073

[torch.FloatTensor of size 2x3]

torch.cat((x, x, x), 0)

-0.5866 -0.3784 -0.1705

-1.0125 0.7406 -1.2073

-0.5866 -0.3784 -0.1705

-1.0125 0.7406 -1.2073

-0.5866 -0.3784 -0.1705

-1.0125 0.7406 -1.2073

[torch.FloatTensor of size 6x3]

torch.cat((x, x, x), 1)

-0.5866 -0.3784 -0.1705 -0.5866 -0.3784 -0.1705 -0.5866 -0.3784 -0.1705

-1.0125 0.7406 -1.2073 -1.0125 0.7406 -1.2073 -1.0125 0.7406 -1.2073

[torch.FloatTensor of size 2x9]

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

拼接张量2

torch.stack((Tensor), dim)

例子：

a = torch.IntTensor([[1,2,3],[11,22,33]])

b= torch.IntTensor([[4,5,6],[44,55,66]])

c=torch.stack([a,b],0)

d=torch.stack([a,b],1)

e=torch.stack([a,b],2)

print(c)

print(d)

print(e)

>>> print(c)

tensor([[[ 1, 2, 3],

[11, 22, 33]],

[[ 4, 5, 6],

[44, 55, 66]]], dtype=torch.int32)

>>> print(d)

tensor([[[ 1, 2, 3],

[ 4, 5, 6]],

[[11, 22, 33],

[44, 55, 66]]], dtype=torch.int32)

>>> print(e)

tensor([[[ 1, 4],

[ 2, 5],

[ 3, 6]],

[[11, 44],

[22, 55],

[33, 66]]], dtype=torch.int32)

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

27

28

c, dim = 0时， c = [ a, b]

d, dim =1 时， d = [ [a[0] , b[0] ] , [a[1], b[1] ] ]

e, dim = 2 时， e=[[[a[0][0],b[0][0]],[a[0][1],b[0][1]],[a[0][2],b[0][2]]],[[a[1][0],b[1][0]],[a[1][1],b[0][1]],[a[1][2],b[1][2]]]]e = [ [ [ a[0][0], b[0][0] ] , [ a[0][1], b[0][1] ] , [ a[0][2],b[0][2] ] ] , [ [ a[1][0], b[1][0] ] , [ a[1][1], b[0][1] ] , [ a[1][2],b[1][2] ] ] ]e=[[[a[0][0],b[0][0]],[a[0][1],b[0][1]],[a[0][2],b[0][2]]],[[a[1][0],b[1][0]],[a[1][1],b[0][1]],[a[1][2],b[1][2]]]]

扩大张量

torch.Tensor.expand(\*sizes) → Tensor

返回张量的一个新视图，可以将张量的单个维度扩大为更大的尺寸。

张量也可以扩大为更高维，新增加的维度将附在前面。 扩大张量不需要分配新内存，仅仅是新建一个张量的视图。任意一个一维张量在不分配新内存情况下都可以扩展为任意的维度。

传入-1则意味着维度扩大不涉及这个维度。

参数：

sizes (torch.Size or int…) – 想要扩展的目标维度

例子：

x = torch.Tensor([[1], [2], [3]])

x.size()

torch.Size([3, 1])

x.expand(3, 4)

1 1 1 1

2 2 2 2

3 3 3 3

[torch.FloatTensor of size 3x4]

1

2

3

4

5

6

7

8

压缩张量

torch.squeeze(input, dim=None, out=None) → Tensor  
除去输入张量input中数值为1的维度，并返回新的张量。如果输入张量的形状为（A×1×B×C×1×D）（ A \times 1 \times B \times C \times 1 \times D ）（A×1×B×C×1×D），那么输出张量的形状为（A×B×C×D）（ A \times B \times C \times D ）（A×B×C×D）。

当通过dim参数指定维度时，维度压缩操作只会在指定的维度上进行。如果输入向量的形状为（A×1×B）（ A \times 1 \times B ）（A×1×B），squeeze(input, 0)会保持张量的维度不变，只有在执行squeeze(input, 1)时，输入张量的形状会被压缩至（A×B）（ A \times B ）（A×B）。

如果一个张量只有1个维度，那么它不会受到上述方法的影响。

输出的张量与原张量共享内存，如果改变其中的一个，另一个也会改变。

参数:

input (Tensor) – 输入张量

dim (int, optional) – 如果给定，则只会在给定维度压缩

out (Tensor, optional) – 输出张量

例子：

x = torch.zeros(2, 1, 2, 1, 2)

x.size()

torch.Size([2, 1, 2, 1, 2])

y = torch.squeeze(x)

y.size()

torch.Size([2, 2, 2])

y = torch.squeeze(x, 0)

y.size()

torch.Size([2, 1, 2, 1, 2])

y = torch.squeeze(x, 1)

y.size()

torch.Size([2, 2, 1, 2])

1

2

3

4

5

6

7

8

9

10

11

12

重复张量

torch.Tensor.repeat(\*sizes)

沿着指定的维度重复张量。不同于expand()方法，本函数复制的是张量中的数据。

参数：

size (torch.size ot int…) - 沿着每一维重复的次数  
例子：

x = torch.Tensor([1, 2, 3])

x.repeat(4, 2)

1 2 3 1 2 3

1 2 3 1 2 3

1 2 3 1 2 3

1 2 3 1 2 3

[torch.FloatTensor of size 4x6]

1

2

3

4

5

6

7

torch.Tensor.unfold(dim, size, step) → Tensor

返回一个新的张量，其中元素复制于有原张量在dim维度上的数据，复制重复size次，复制时的步进值为step。

参数：

dim (int) - 目标维度

size (int) - 复制重复的次数（展开维度）

step (int) - 步长  
例子：

x = torch.arange(1, 8)

x

1

2

3

4

5

6

7

[torch.FloatTensor of size 7]

x.unfold(0, 2, 1)

1 2

2 3

3 4

4 5

5 6

6 7

[torch.FloatTensor of size 6x2]

x.unfold(0, 2, 2)

1 2

3 4

5 6

[torch.FloatTensor of size 3x2]

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

缩小张量

torch.Tensor.narrow(dimension, start, length) → Tensor

返回一个经过缩小后的张量。操作的维度由dimension指定。缩小范围是从start开始到start+length。执行本方法的张量与返回的张量共享相同的底层内存。

参数：

dimension (int) – 要进行缩小的维度

start (int) – 开始维度索引

length (int) – 缩小持续的长度  
例子：

x = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

x.narrow(0, 0, 2)

1 2 3

4 5 6

[torch.FloatTensor of size 2x3]

x.narrow(1, 1, 2)

2 3

5 6

8 9

[torch.FloatTensor of size 3x2]

1

2

3

4

5

6

7

8

9

10

张量变形

torch.Tensor.view(\*args) → Tensor

返回一个有相同数据但是不同形状的新的向量。

返回的装两必须与原张量有相同的数据和相同的元素个数，但是可以有不同的尺寸。

参数：

args (torch.Size or int…) - 理想的指定尺寸

x = torch.randn(4, 4)

x.size()

torch.Size([4, 4])

y = x.view(16)

y.size()

torch.Size([16])

1

2

3

4

5

6

重设张量尺寸

torch.Tensor.resize\_(\*sizes)

将张量的尺寸调整为指定的大小。如果元素个数比当前的内存大小大，就将底层存储大小调整为与新元素数目一致的大小。  
如果元素个数比当前内存小，则底层存储不会被改变。原来张量中被保存下来的元素将保持不变，但新内存将不会被初始化。  
参数：

sizes (torch.Size or int…) - 需要调整的大小

例子：

x = torch.Tensor([[1, 2], [3, 4], [5, 6]])

x.resize\_(2, 2)

x

1 2

3 4

[torch.FloatTensor of size 2x2]

1

2

3

4

5

6

置换张量维度

torch.Tensor.permute(\*dims)

将执行本方法的张量的维度换位。

参数：

dim (int) - 指定换位顺序

例子：

x = torch.randn(2, 3, 5)

x.size()

torch.Size([2, 3, 5])

x.permute(2, 0, 1).size()

torch.Size([5, 2, 3])

1

2

3

4

5

查看张量单个元素的字节数

torch.Tensor.element\_size() → int

查看某类型张量单个元素的字节数。

例子：

torch.FloatTensor().element\_size()

4