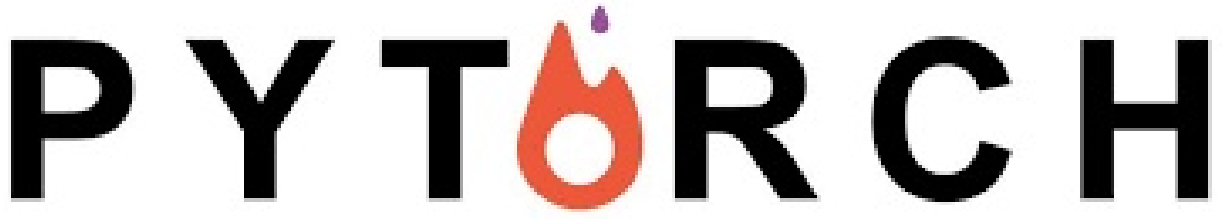


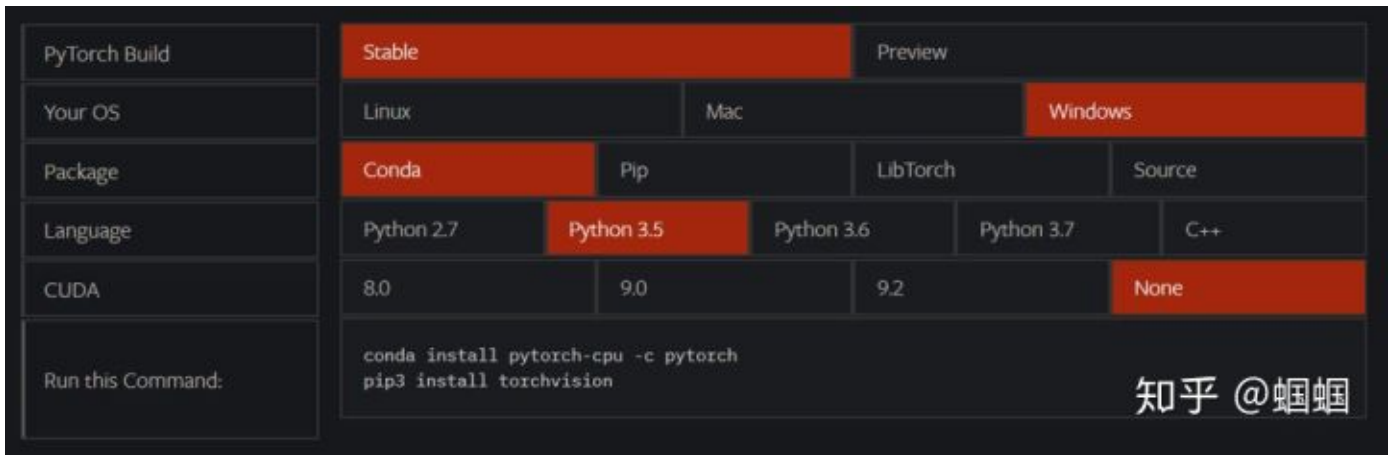
PyTorch简明笔记[1]-Tensor的初始化和基本操作

蝈蝈
深度学习小学生，NLP爱好者。个人分享公众号：SimpleAI
关注他

> 听麻麻说，偷偷收藏而不感谢是不礼貌的，至少应该点个赞~我觉得麻麻说的对！

> 不断地被人安利PyTorch，终于忍不住诱惑决定入坑了。
> 当初学习TensorFlow的时候，没有系统性地学习。之前TF的英文官网一直看不了，而中文版的文档又很烂，导致学起来一直不那么爽，每次搭建模型的时候，都要大量的回来查阅文档，对很多基本的用法搞不清楚。
> 当我翻看PyTorch官网的时候，一下子就爱上了它那清晰的文档和友好的入门指南。所以决定好好地系统性地把PyTorch学一学。所以，记一份适合自己的更加清晰简明的笔记，把基础打牢固，就很有必要了。
> 这份笔记的目的，主要是**方便随时查阅**，不必去看详细的冗长的原始文档。也方便跟我一样的小白可以**迅速入门，进行实践。**
> **本篇是PyTorch简明笔记第[1]篇.**

安装PyTorch应该不用我多说，他们的官网很人性化地给出了各种环境应该怎么安装，网址：pytorch.org/get-started...



安装完以后，在python里面试试 `import torch` ，没有报错就安装好了。

一、定义/初始化张量Define tensors

tensor，即“张量”。实际上跟numpy数组、向量、矩阵的格式基本一样。但是是专门针对GPU来设计的，可以运行在GPU上来加快计算效率。

PyTorch中定义tensor，就跟numpy定义矩阵、向量差不多，例如定义一个5×3的tensor，每一项都是0的张量：

```
x = torch.zeros(5,3)
```

如果想查看某个tensor的**形状**的话，使用：
`z.size()` ，或者 `z.shape` ，但是前者更常用。

> 下面列举一些**常用**的定义tensor的方法：

常数初始化：

- `torch.empty(size)` 返回形状为size的空tensor
- `torch.zeros(size)` 全部是0的tensor
- `torch.zeros_like(input)` 返回跟input的tensor一个size的全零tensor
- `torch.ones(size)` 全部是1的tensor
- `torch.ones_like(input)` 返回跟input的tensor一个size的全一tensor
- `torch.arange(start=0, end, step=1)` 返回一个从start到end的序列，可以只输入一个end参数，就跟python的range()一样了。实际上PyTorch也有range()，但是这个要被废掉了，替换成arange了
- `torch.full(size, fill_value)` 这个有时候比较方便，把fill_value这个数字变成size形状的张量

随机抽样（随机初始化）：

- `torch.rand(size)` [0,1)内的均匀分布随机数
- `torch.rand_like(input)` 返回跟input的tensor一样size的0-1随机数
- `torch.randn(size)` 返回标准正太分布N(0,1)的随机数
- `torch.normal(mean, std, out=None)` 正态分布。这里注意，mean和std都是tensor，返回的形状由mean和std的形状决定，一般要求两者形状一样。如果，mean缺失，则默认为均值0，如果std缺失，则默认标准差为1.

更多的随机抽样方法，参见链接：
pytorch.org/docs/stable...

二、基本操作、运算 Basic operations

1.tensor的切片、合并、变形、抽取操作

(Indexing, Slicing, Joining, Mutating)

> 这里我就简单总结一些重要的tensor基本操作：

- `torch.cat(seq, dim=0, out=None)` ,把一堆tensor丢进去，按照dim指定的维度拼接、堆叠在

```
Out[71]: tensor([[1, 2, 3]])

 #按第0维度堆叠，对于矩阵，相当于“竖着”堆
In [72]: print(torch.cat((x,x,x),0))
tensor([[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]])

 #按第1维度堆叠，对于矩阵，相当于“横着”拼
In [73]: print(torch.cat((x,x,x),1))
tensor([[1, 2, 3, 1, 2, 3, 1, 2, 3]])
```

- `torch.chunk(tensor, chunks, dim=0)` 把tensor切成块，数量由chunks指定。
  例如：

```
In [74]: a = torch.arange(10)
In [75]: a
Out[75]: tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [76]: torch.chunk(a,4)
Out[76]: (tensor([0, 1, 2]), tensor([3, 4, 5]), tensor([6, 7, 8]), tensor([9]))
```

- 切块还有 `torch.split(tensor, split_size_or_sections, dim=0)` 具体区别大家自行查阅文档
- 按index选择： `torch.index_select(input, dim, index, out=None)`
- 按mask选择： `torch.masked_select(input, mask, out=None)`
- 经常会使用的“压扁”函数： `torch.squeeze(input)` ,压缩成1维。注意，压缩后的tensor和原来的tensor共享地址
- 改变形状： `torch.reshape(input, shape)` 以及 `tensor.view(shape)` .前者是把tensor作为函数的输入，后者是任何tensor的函数。实际上，二者的返回值，都只是让我们从另一种视角看某个tensor，所以不会改变本来的形状，除非你把结果又赋值给原来的tensor。下面给一个例子对比二者的用法：

```
In [82]: a
Out[82]: tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# 单纯的调用view函数：
In [83]: a.view(2,5)
Out[83]:
tensor([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]])

# a的形状并不会变化
In [84]: print(a)
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# 试试reshape函数：
In [86]: torch.reshape(a,[5,2])
Out[86]:
tensor([[0, 1],
        [2, 3],
        [4, 5],
        [6, 7],
        [8, 9]])

# a的形状依然不会变化：
In [87]: a
Out[87]: tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

2.基本数学操作

- 加法直接加： `x+y`
  或者用 `torch.add(x,y)` .
  实际上， `.add()` 可以接受三个参数： `torch.add(input, value, out=None)`
  out怎么用呢？一般，如果直接 `torch.add(x,y)` ，那么x，y本身都不会变化的。但是如果设置 `out=x` ，那么x就变变成加和后的值。

特别的，若想进行**in-place**操作，就比方说y加上x，y的值就改变了，就可以用 `y.add_(x)` 这样y就直接被改变了。Torch里面所有带“_”的操作，都是in-place的。例如 `x.copy_(y)`

- 乘法： `torch.mul(input, other, out=None)` 用input乘以other
- 除法： `torch.div(input, other, out=None)` 用input除以other
- 指数： `torch.pow(input, exponent, out=None)`
- 开根号： `torch.sqrt(input, out=None)`
- 四舍五入到整数： `torch.round(input, out=None)`
- **argmax函数**： `torch.argmax(input, dim=None, keepdim=False)` 返回指定维度最大值的序号，**dim给定的定义是：the demention to reduce**.也就是把dim这个维度的，变成这个维度的最大值的index。例如：
- sigmoid函数： `torch.sigmoid(input, out=None)`
- tanh函数： `torch.tanh(input, out=None)`
- `torch.abs(input, out=None)` 取绝对值
- `torch.ceil(input, out=None)` 向上取整，等于向下取整+1
- `torch.clamp(input, min, max, out=None)` **刀削**函数，把输入数据规范在min-max区间，超过范围的用min、max代替

太多了，基本上，numpy里面有的数学函数这里都有，能想到的的基本都有。所以更详细的内容，还是去查看文档吧：
pytorch.org/docs/stable...

三、Torch Tensor与Numpy的互相转换

- **Tensor→Numpy**
  直接用 `.numpy()` 即可。但是注意，转换后，**numpy的变量和原来的tensor会共用底层内存地址**，所以如果原来的tensor改变了，numpy变量也会随之改变。参见下面的例子：

```
In [11]: a = torch.ones(2,4)
In [12]: a
Out[12]:
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.]])

In [13]: b = a.numpy()
In [14]: b
Out[14]:
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)

In [15]: a.add_(1)
Out[15]:
tensor([[2., 2., 2., 2.],
        [2., 2., 2., 2.]])

In [16]: b
```

- **Numpy→Tensor**
  用 `torch.from_numpy()` 来转换。参见下面例子：

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

输出：

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

同样，两者会共用内存地址。

> 好啦，本篇就这么些了，快去练习一下吧！
> 参考链接：
> PyTorch文档：
> pytorch.org/docs/torch

> **如果喜欢我的教程，欢迎关注我的专栏**：
> 【 DeepLearning.ai学习笔记】
> 和我一起一步步学习深度学习。
> 欢迎来微信公众号 ***SimpleAI*** ヾ(๑╹◡╹)ﾉ"踩踩哟~
>
> **专栏文章目录：**
> 【DL笔记1】Logistic回归：最基础的神经网络
> 【DL笔记2】神经网络编程原则&Logistic Regression的算法解析
> 【DL笔记3】一步步用python实现Logistic回归
> 【DL笔记4】神经网络详解，正向传播和反向传播
> 【DL笔记5】TensorFlow搭建神经网络：手写数字识别
> 【DL笔记6】从此明白了卷积神经网络（CNN）
> 【DL笔记7】他山之玉——窥探CNN经典模型
> 【DL碎片1】神经网络参数初始化的学问
> 【DL碎片2】神经网络中的优化算法
> 【DL碎片3】神经网络中的激活函数及其对比
> 【DL碎片4】深度学习中的的超参数调节
> 【DL碎片5】深度学习中的正则化Regularization
> . . . . . .

编辑于 2018-12-04

「真诚赞赏，手留余香」