

—武大本科生课程



第14讲 PyTorch深度学习框架及应用

(Lecture 14 PyTorch Deep learning framework and application)

武汉大学计算机学院

内容目录

1. 相关软件的安装
2. 什么是PYTORCH以及PYTORCH基础
3. 自动求梯度
4. PyTorch 神经网络

1.相关软件的安装

安装Anaconda 3

1.1 Anaconda下载及环境配置

Anaconda是一个用于科学计算的Python发行版，支持Linux、Mac和Window系统，提供了包管理 与环境管理的功能，可以很方便地解决Python并存、切换，以及各种第三方包安装的问题。

关于下载：可以直接从 Anaconda官网下载，但因为Anaconda的服务器在国外，所以下载速度可能会很慢，这里推荐使用清华的镜像来下载。选择合适你的版本下载。

Anaconda Installers

Windows

Python 3.8

64-Bit Graphical Installer (477 MB)

32-Bit Graphical Installer (409 MB)

MacOS

Python 3.8

64-Bit Graphical Installer (440 MB)

64-Bit Command Line Installer (433 MB)

Linux

Python 3.8

64-Bit (x86) Installer (544 MB)

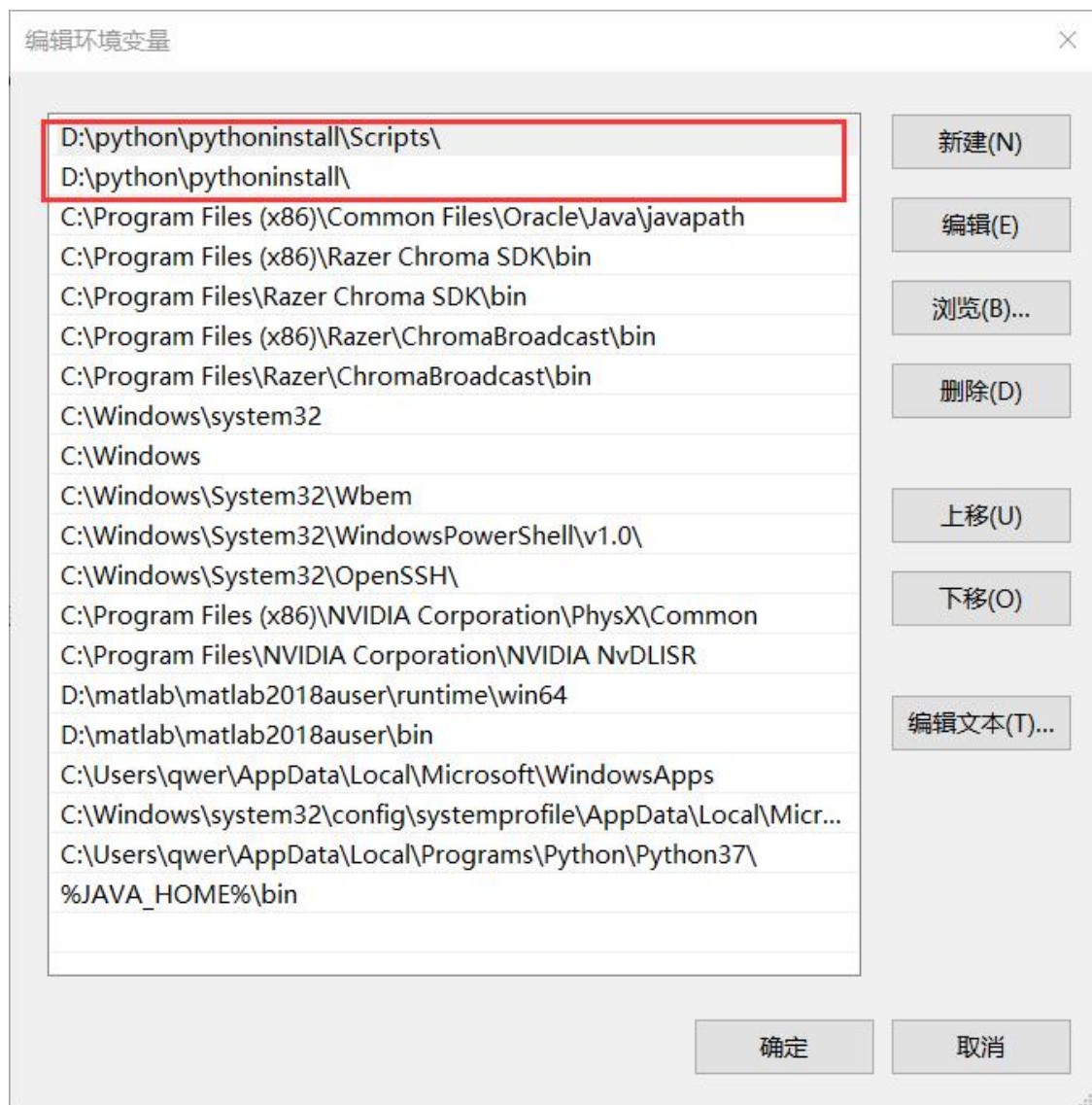
64-Bit (Power8 and Power9) Installer (285 MB)

64-Bit (AWS Graviton2 / ARM64) Installer (413 M)

64-bit (Linux on IBM Z & LinuxONE) Installer (292 M)

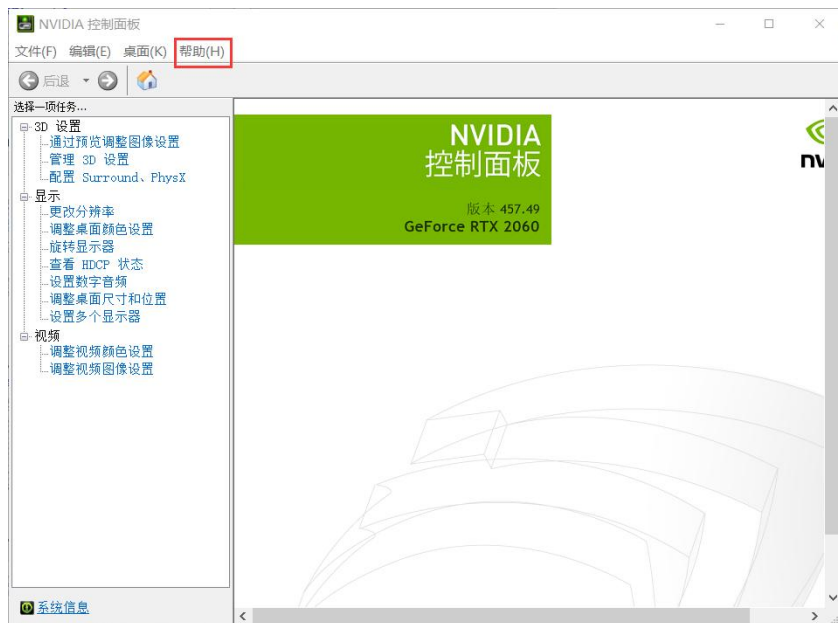
关于环境配置：首先找到安装的路径，以及文件中Scripts的路径。然后打开控制面板->高级系统设置->环境变量->系统变量，找到Path，点击编辑，然后选择新建加上这两个文件夹的路径。右图是我的添加完成后的结果。

完成上述过程后，基本的环境配置就完成了，可以打开Anaconda配置自己需要的虚拟环境。

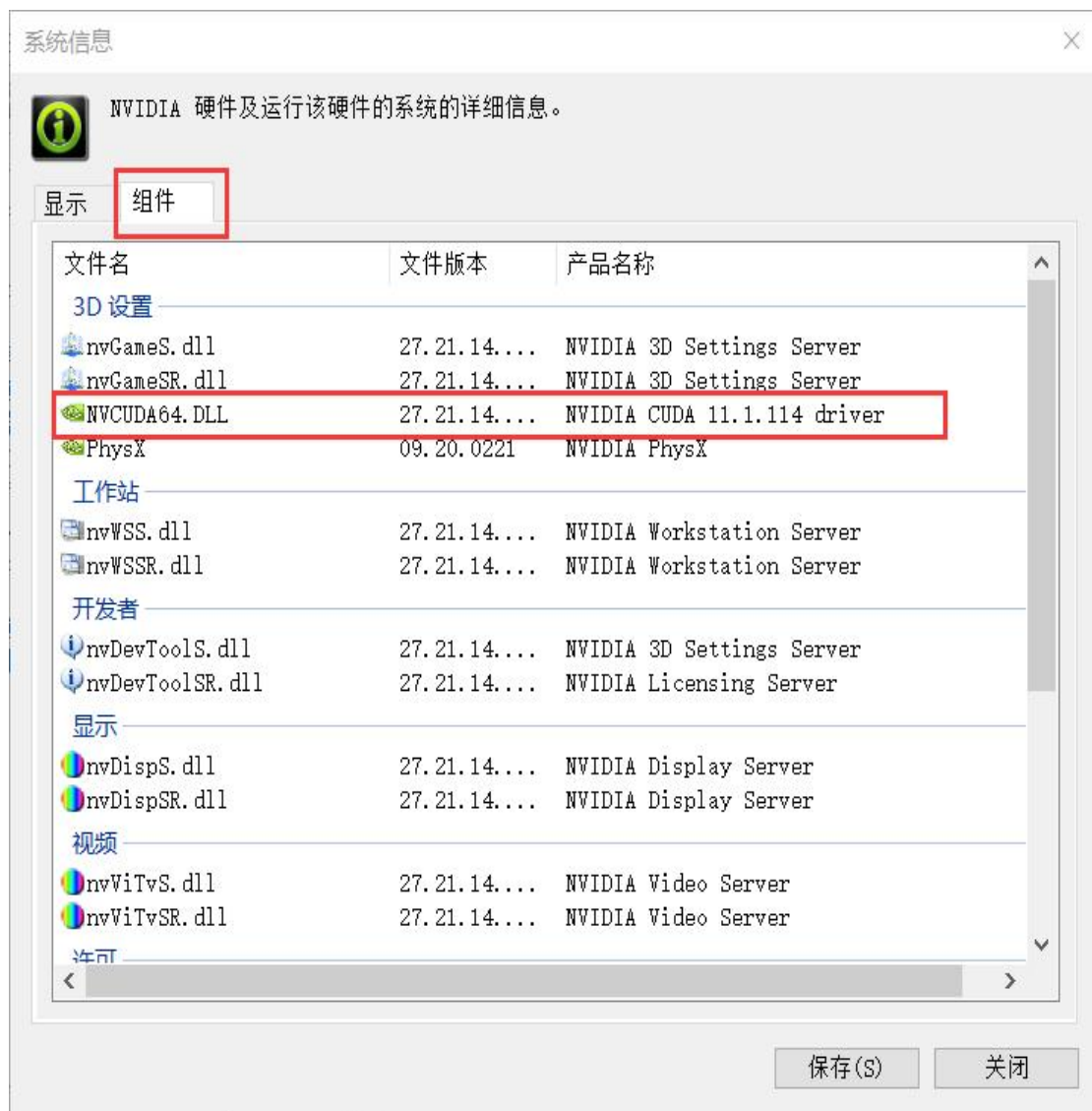


1.2 安装PyTorch&torchvision

首先查看自己的gpu版本（如果没有gpu忽略这步）。在桌面空白处右键单击，依次点击NVIDIA控制面板→帮助→系统信息→组件，查看CUDA版本。



由右图我们可以知道这台电脑
最高支持11.1版本的CUDA。



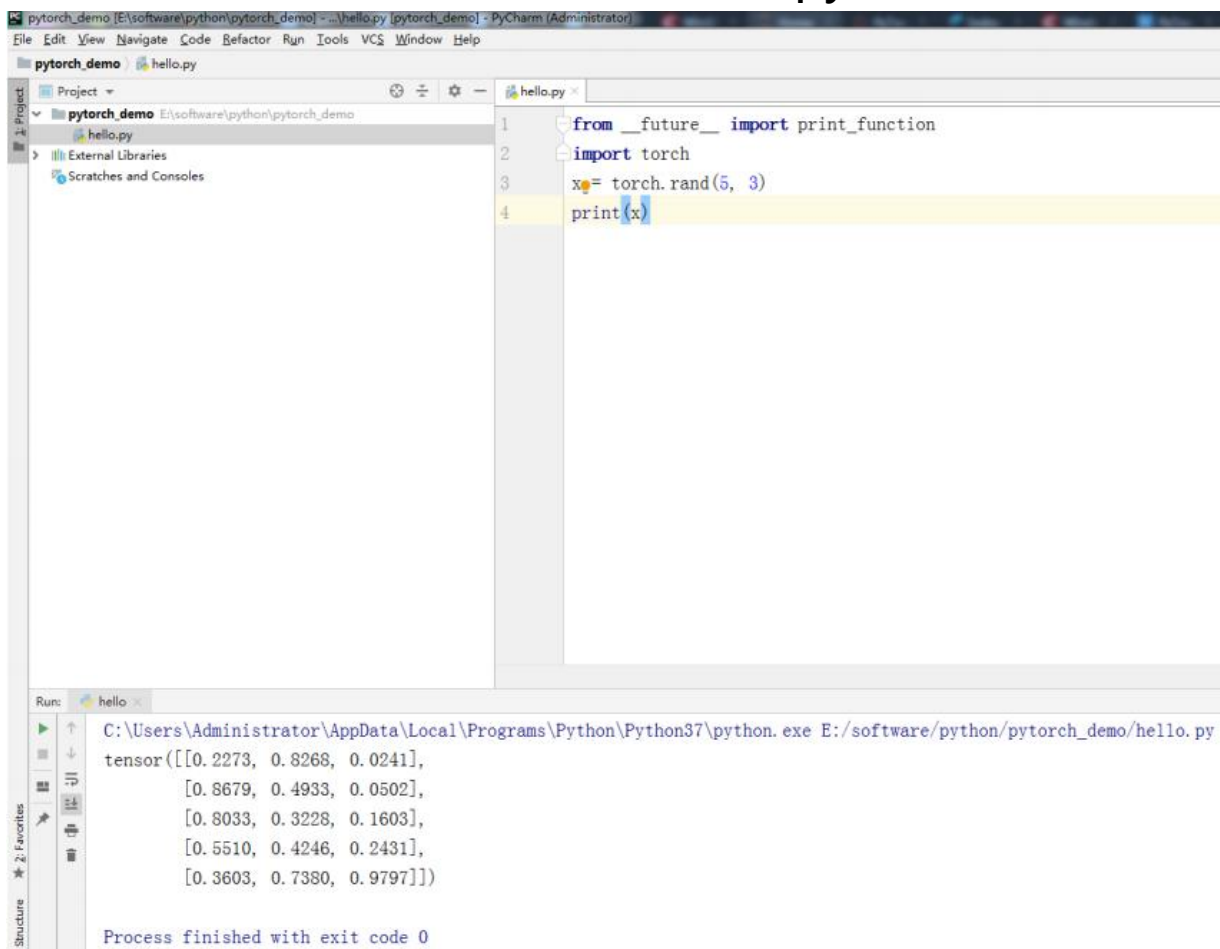
现在正式进入安装阶段，这里提供使用pip和 conda两种环境下安装的步骤截图

(1) 使用pip: windows+pip+python3.7+CUDA (None)
没有GPU的同学在CUDA选择None，有GPU的同学在查看自己电脑支持的CUDA版本后选择对应的版本。
然后复制红框中的命令在cmd中执行。

| | | | | | |
|-------------------|--|------------|-------------------|------------|-----|
| PyTorch Build | Stable (1.0) | | Preview (Nightly) | | |
| Your OS | Linux | Mac | Windows | | |
| Package | Conda | Pip | LibTorch | Source | |
| Language | Python 2.7 | Python 3.5 | Python 3.6 | Python 3.7 | C++ |
| CUDA | 8.0 | 9.0 | 10.0 | None | |
| Run this Command: | <pre>pip3 install https://download.pytorch.org/whl/cpu/torch-1.0.1-cp37-cp37m-win_amd64.whl pip3 install torchvision</pre> | | | | |

Previous versions of PyTorch

安装成功后检验是否安装成功，打开pycharm运行一个小demo：



The screenshot shows the PyCharm IDE interface. The top toolbar includes buttons for File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, and Help. The main editor window displays a file named `hello.py` with the following code:

```
1 from __future__ import print_function
2 import torch
3 x = torch.rand(5, 3)
4 print(x)
```

The bottom panel shows the Run output for the `hello` configuration. The command executed is `C:\Users\Administrator\AppData\Local\Programs\Python\Python37\python.exe E:/software/python/pytorch_demo/hello.py`. The output is a 5x3 tensor of random values:

```
tensor([[0.2273, 0.8268, 0.0241],
        [0.8679, 0.4933, 0.0502],
        [0.8033, 0.3228, 0.1603],
        [0.5510, 0.4246, 0.2431],
        [0.3603, 0.7380, 0.9797]])
```

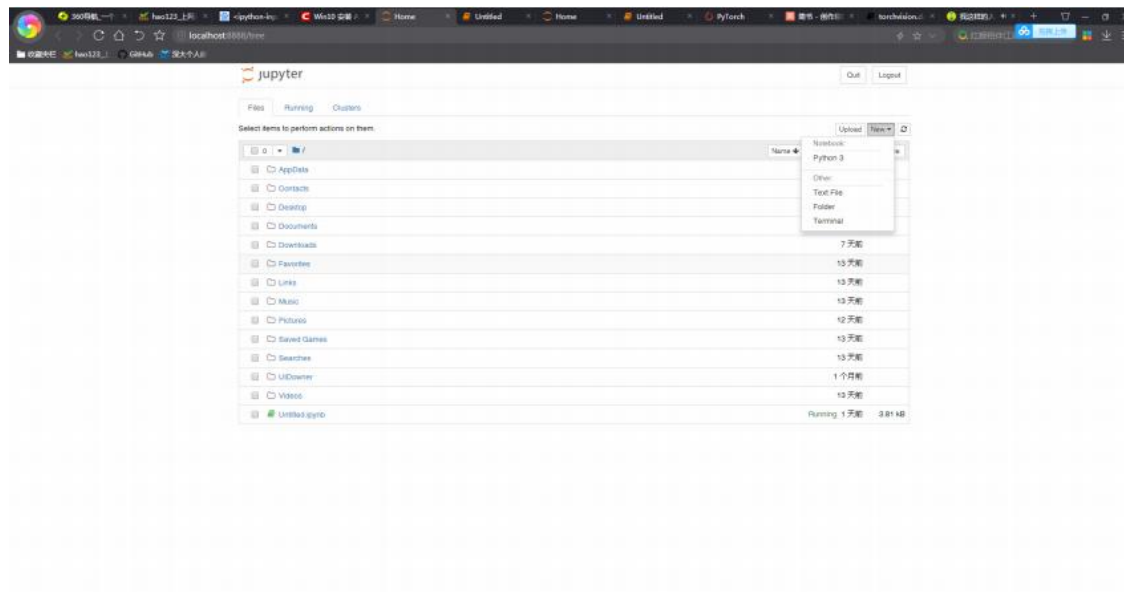
The process finished with exit code 0.

成功运行则表示安装成功。

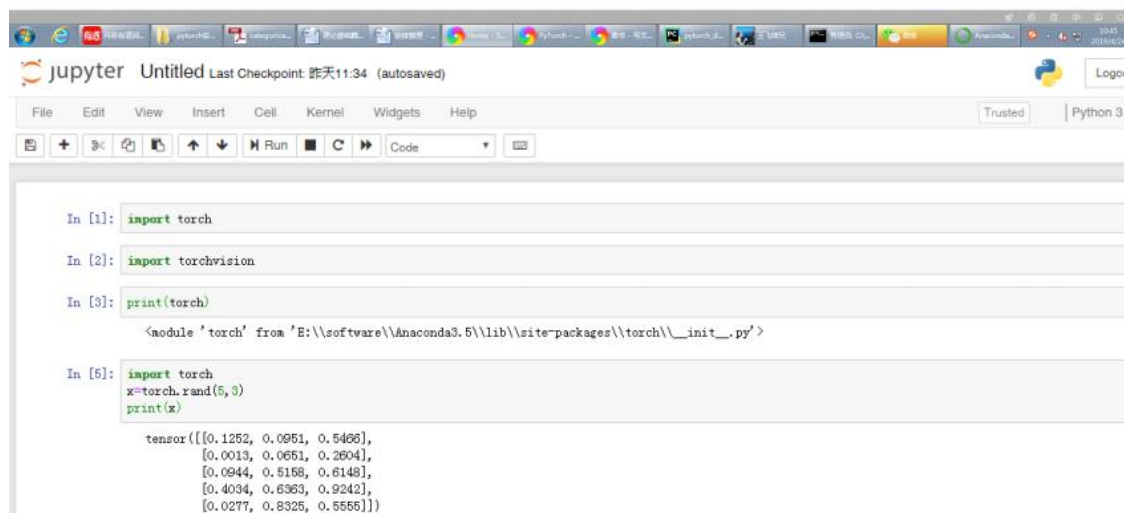
(2) 使用conda: windows+conda+python3.7+CUDA (None)
同样复制红框中命令在cmd中运行。

| | | | | | |
|-------------------|---|------------|-------------------|------------|--------|
| PyTorch Build | Stable (1.0) | | Preview (Nightly) | | |
| Your OS | Linux | | Mac | Windows | |
| Package | Conda | Pip | | LibTorch | Source |
| Language | Python 2.7 | Python 3.5 | Python 3.6 | Python 3.7 | C++ |
| CUDA | 8.0 | 9.0 | 10.0 | None | |
| Run this Command: | <code>conda install pytorch-cpu torchvision-cpu -c pytorch</code> | | | | |

安装完毕后，验证是否安装成功，打开Anaconda的Jupyter新建python文件，运行demo：



出现这个结果，那么恭喜你，至此PyTorch & Anaconda3已经安装成功.

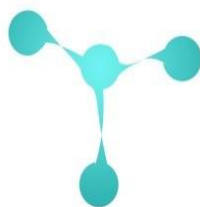


2.什么是PYTORCH以及PYTORCH基础

2.1 Pytorch简介

随着深度学习的发展，深度学习框架如雨后春笋般诞生于高校和公司中。尤其是近两年，Google、Facebook、Microsoft等巨头都围绕深度学习重点投资了一系列新兴项目，他们也一直在支持一些开源的深度学习框架。目前研究人员正在使用的深度学习框架不尽相同，有 TensorFlow、Caffe、Theano、Keras等，常见的深度学习框架如图所示。这些深度学习框架被应用于计算机视觉、语音识别、自然语言处理与生物信息学等领域，并获取了极好的效果。

Microsoft
CNTK



Caffe

Caffe2

PYTORCH

Chainer

K Keras

TensorFlow

theano

dy/net

mxnet

GLUON

有如此多的深度学习框架，为何我们选择Pytorch？

- **简洁**：PyTorch的设计追求最少的封装，尽量避免重复造轮子。不像TensorFlow中充斥着session、graph、operation、name_scope、variable、tensor、layer等全新的概念，PyTorch的设计遵循tensor→variable(autograd)→nn.Module 三个由低到高的抽象层次，分别代表高维数组（张量）、自动求导（变量）和神经网络（层/模块），而且这三个抽象之间联系紧密，可以同时进行修改和操作。简洁的设计带来的另外一个好处就是代码易于理解。PyTorch的源码只有TensorFlow的十分之一左右，更少的抽象、更直观的设计使得PyTorch的源码十分易于阅读。在很多人眼里，PyTorch的源码甚至比许多框架的文档更容易理解。
- **速度**：PyTorch的灵活性不以速度为代价，在许多评测中，PyTorch的速度表现胜过TensorFlow和Keras等框架。框架的运行速度和程序员的编码水平有极大关系，但同样的算法，使用PyTorch实现的那个更有可能快过用其他框架实现的。
- **易用**：PyTorch是所有的框架中面向对象设计的最优雅的一个。PyTorch的面向对象的接口设计来源于Torch，而Torch的接口设计以灵活易用而著称，Keras作者最初就是受Torch的启发才开发了Keras。PyTorch继承了Torch的衣钵，尤其是API的设计和模块的接口都与Torch高度一致。PyTorch的设计最符合人们的思维，它让用户尽可能地专注于实现自己的想法，即所思即所得，不需要考虑太多关于框架本身的束缚。
- **活跃的社区**：PyTorch提供了完整的文档，循序渐进的指南，作者亲自维护的论坛供用户交流和请教问题。Facebook 人工智能研究院对PyTorch提供了强力支持，作为当今排名前三的深度学习研究机构，FAIR的支持足以确保PyTorch获得持续的开发更新，不至于像许多由个人开发的框架那样昙花一现。

2.2 PyTorch基础

0维张量：标量

1维张量：向量

2维张量：矩阵

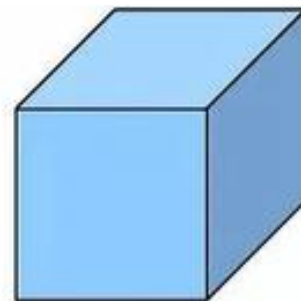
张量示意图



1d-tensor



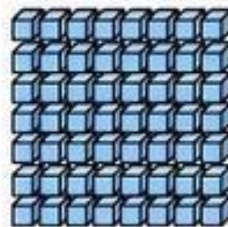
2d-tensor



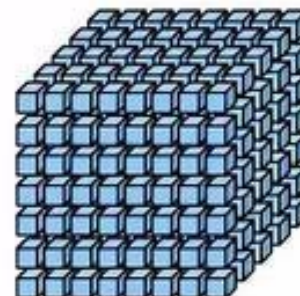
3d-tensor



4d-tensor



5d-tensor



6d-tensor

创建张量的几种方式

- 用现有数据创建张量，使用`torch.tensor()`。
如`torch.tensor([[1.,-1.], [1.-1.]])`
- 要创建具有特定大小的张量，请使用`torch.*`。
如`torch.randn()` #满足标准正态分布的一组随机数据
- 创建与另一个张量具有相同大小的张量，请使用`torch.*_like`。
如`torch.rand_like()`
- 创建与其他张量具有相似类型但大小不同的张量，请使用`tensor.new_*`创建操作。

•查看张量的属性

- 查看Tensor类型:

`tensor1 = torch.randn(2,3)` #形状为(2,3)一组从标准正太分布中随机抽取的数据

`tensor1.dtype` # `torch.float32`

- 查看Tensor维度和形状:

`tensor1.shape` #查看形状或尺寸

`tensor.ndim` #查看维度

- 查看Tensor是否存储在GPU上:

`tensor1.is_cuda`

- 查看Tensor的梯度:

`tensor1.grad`

接下来我们来看具体操作：

创建一个未初始化的5*3张量（切记不是全零）

```
import torch
# 这个是用来生成一个为未初始化的5*3的张量，切记不是全零
x = torch.empty(5, 3)
print(x)
"""
tensor([[2.7712e+35, 4.5886e-41, 7.2927e-04],
        [3.0780e-41, 3.8725e+35, 4.5886e-41],
        [4.4446e-17, 4.5886e-41, 3.9665e+35],
        [4.5886e-41, 3.9648e+35, 4.5886e-41],
        [3.8722e+35, 4.5886e-41, 4.4446e-17]])
"""
```

创建一个随机初始化的5*3张量

```
x = torch.rand(5, 3)
print(x)
"""
tensor([[0.9600, 0.0110, 0.9917],
        [0.9549, 0.1732, 0.7781],
        [0.8098, 0.5300, 0.5747],
        [0.5976, 0.1412, 0.9444],
        [0.6023, 0.7750, 0.5772]])
"""
```


创建一个全零的5*3张量，可以指定每个元素类型

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
"""tensor([[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]])"""
```

还可以根据数据直接创建张量

```
x = torch.tensor([5.5, 3])
print(x)
"""
tensor([5.5000, 3.0000])
"""
```

张量的操作

通过上一节，我们已经设计了x张量，这里提供了三种加法的方式

- 加法形式一

```
1 y = torch.rand(5, 3)
2 print(x + y)
```

- 加法形式二

```
1 print(torch.add(x, y))
```

还可指定输出：

```
1 result = torch.empty(5, 3)
2 torch.add(x, y, out=result)
3 print(result)
```

- 加法形式三、inplace

```
1 # adds x to y
2 y.add_(x)
3 print(y)
```

输出结果为：

```
1 tensor([[ 1.3967,  1.0892,  0.4369],
2         [ 1.6995,  2.0453,  0.6539],
3         [-0.1553,  3.7016, -0.3599],
4         [ 0.7536,  0.0870,  1.2274],
5         [ 2.5046, -0.1913,  0.4760]])
```

改变形状

用 `view()` 来改变 `Tensor` 的形状：

```
1 y = x.view(15)
2 z = x.view(-1, 5) # -1所指的维度可以根据其他维度的值推出来
3 print(x.size(), y.size(), z.size())
```

输出：

```
1 torch.Size([5, 3]) torch.Size([15]) torch.Size([3, 5])
```

另外一个常用的函数就是 `item()`，它可以将一个标量 Tensor 转换成一个 Python number

```
1 x = torch.randn(1)
2 print(x)
3 print(x.item())
```

输出：

```
1 tensor([2.3466])
2 2.3466382026672363
```

前面我们看到如何对两个形状相同的 Tensor 做按元素运算。当对两个形状不同的 Tensor 按元素运算时，可能会触发广播（broadcasting）机制：先适当复制元素使这两个 Tensor 形状相同后再按元素运算。例如：

```
1 x = torch.arange(1, 3).view(1, 2)
2 print(x)
3 y = torch.arange(1, 4).view(3, 1)
4 print(y)
5 print(x + y)
```

输出：

```
1 tensor([[1, 2]])
2 tensor([[1],
3         [2],
4         [3]])
5 tensor([[2, 3],
6         [3, 4],
7         [4, 5]])
```

由于 x 和 y 分别是1行2列和3行1列的矩阵，如果要计算 $x + y$ ，那么 x 中第一行的2个元素被广播（复制）到了第二行和第三行，而 y 中第一列的3个元素被广播（复制）到了第二列。如此，就可以对2个3行2列的矩阵按元素相加。

TENSOR 和NUMPY相互转换

我们很容易用 `numpy()` 和 `from_numpy()` 将 Tensor 和 NumPy 中的数组相互转换。但是需要注意的一点是：这两个函数所产生的的 Tensor 和 NumPy 中的数组共享相同的内存（所以他们之间的转换很快），改变其中一个时另一个也会改变。

Tensor 转 NumPy

使用 `numpy()` 将 Tensor 转换成 NumPy 数组:

```
1 a = torch.ones(5)
2 b = a.numpy()
3 print(a, b)
4
5 a += 1
6 print(a, b)
7 b += 1
8 print(a, b)
```

输出:

```
1 tensor([1., 1., 1., 1., 1.]) [1. 1. 1. 1. 1.]
2 tensor([2., 2., 2., 2., 2.]) [2. 2. 2. 2. 2.]
3 tensor([3., 3., 3., 3., 3.]) [3. 3. 3. 3. 3.]
```

NumPy数组转Tensor

使用 `from_numpy()` 将NumPy数组转换成Tensor:

```
1 import numpy as np
2 a = np.ones(5)
3 b = torch.from_numpy(a)
4 print(a, b)
5
6 a += 1
7 print(a, b)
8 b += 1
9 print(a, b)
```

输出:

```
1 [1. 1. 1. 1. 1.] tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
2 [2. 2. 2. 2. 2.] tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
3 [3. 3. 3. 3. 3.] tensor([3., 3., 3., 3., 3.], dtype=torch.float64)
```

所有在CPU上的Tensor (除了CharTensor) 都支持与NumPy数组相互转换。

3. 自动求梯度 (autograd)

在深度学习中，我们经常需要对函数求梯度（gradient）。PyTorch提供的autograd包能够根据输入和前向传播过程自动构建计算图，并执行反向传播。本节将介绍如何使用autograd包来进行自动求梯度的有关操作。

概念： 将Tensor的属性.requires_grad设置为True，它将开始追踪在其上的所有操作，完成计算后，可以调用.backward()来完成所有梯度计算。此Tensor的梯度将累积到.grad属性中。

如果y.backward()时，y是标量，则不需要为backward()传入任何参数；否则需要传入一个与y同形的Tensor

如果不想被继续追踪，则可以调用.detach()将其从追踪记录中分离出来，这样就可以防止将来的计算被追踪，这样梯度就传不下去。此外，还可以调用with torch.no_grad()将不想被追踪的操作代码块包裹起来，这种方法在评估模型的时候很常用。

Function是另外一个很重要的类。Tensor和Function互相结合就可以构建一个记录有整个计算过程的有向无环图（DAG）。每个Tensor都有一个.grad_fn属性，该属性即创建该Tensor的Function，就是说该Tensor是不是通过某些运算得到的，若是，则grad_fn返回一个与这些运算相关的对象，否则是None。

如果你想计算导数，你可以调用 `Tensor.backward()`。如果 `Tensor` 是标量（即它包含一个元素数据），则不需要指定任何参数 `backward()`，但是如果它有更多元素，则需要指定一个 `gradient` 参数来指定张量的形状。

创建一个张量，设置 `requires_grad=True` 来跟踪与它相关的计算

```
import torch

x = torch.ones(2, 2, requires_grad=True)
print(x)
```

输出：

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

针对张量做一个操作

```
y = x + 2  
print(y)
```

输出：

```
tensor([[3., 3.],  
        [3., 3.]], grad_fn=<AddBackward0>)
```

y 作为操作的结果被创建，所以它有 grad_fn

```
print(y.grad_fn)
```

输出：

```
<AddBackward0 object at 0x0000020375A1E948>
```

针对 y 做更多的操作：

```
z = y * y * 3
out = z.mean()
print(z, out)
```

输出：

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>) tensor(27., grad_fn=<MeanBackward0>)
```

.requires_grad_(...) 会改变张量的 requires_grad 标记。输入的标记默认为 False，如果没有提供相应的参数。

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)
```

输出：

```
False
True
<SumBackward0 object at 0x0000022013EEEA08>
```

梯度：

我们现在后向传播，因为输出包含了一个标量，`out.backward()` 等同于 `out.backward(torch.tensor(1.))`。

打印梯度 $d(\text{out})/dx$

```
out.backward()
print(x.grad)
```

输出：

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

原理：

我们令 `out` 为 o ，因为

$$o = \frac{1}{4} \sum_{i=1}^4 z_i = \frac{1}{4} \sum_{i=1}^4 3(x_i + 2)^2 \quad (1)$$

所以

$$\left. \frac{\partial o}{\partial x_i} \right|_{x_i=1} = \frac{9}{2} = 4.5 \quad (2)$$

所以上面的输出是正确的。

量都为向量的函数 $\vec{y} = f(\vec{x})$ ，那么 \vec{y} 关于 \vec{x} 的梯度就是一个雅可比矩阵（Jacobian matrix）：

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \quad (3)$$

而 `torch.autograd` 这个包就是用来计算一些雅可比矩阵的乘积的。例如，如果 v 是一个标量函数的 $l = g(\vec{y})$ 的梯度：

$$v = \left(\frac{\partial l}{\partial y_1} \quad \cdots \quad \frac{\partial l}{\partial y_m} \right) \quad (4)$$

那么根据链式法则我们有 l 关于 \vec{x} 的雅可比矩阵就为：

$$vJ = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix} \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} & \cdots & \frac{\partial l}{\partial x_n} \end{pmatrix} \quad (5)$$

注意：grad在反向传播过程中是累加的(accumulated)，这意味着每一次运行反向传播，梯度都会累加之前的梯度，所以一般在反向传播之前需把梯度清零。

现在让我们看一个雅可比向量积的例子：

```
x = torch.randn(3, requires_grad=True)
y = x * 2
while y.data.norm() < 1000:
    y = y * 2
print(y)
```

输出：

```
tensor([-946.8287, 1088.6514, -407.4377], grad_fn=<MulBackward0>)
```

现在在这种情况下， y 不再是一个标量。`torch.autograd` 不能够直接计算整个雅可比，但是如果只想要雅可比向量积，只需要简单的传递向量给 `backward` 作为参数。

```
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
y.backward(v)
print(x.grad)
```

输出：

```
tensor([1.0240e+02, 1.0240e+03, 1.0240e-01])
```

你还可以通过将代码包裹在 `with torch.no_grad()`，来停止对从跟踪历史中的 `.requires_grad=True` 的张量自动求导。

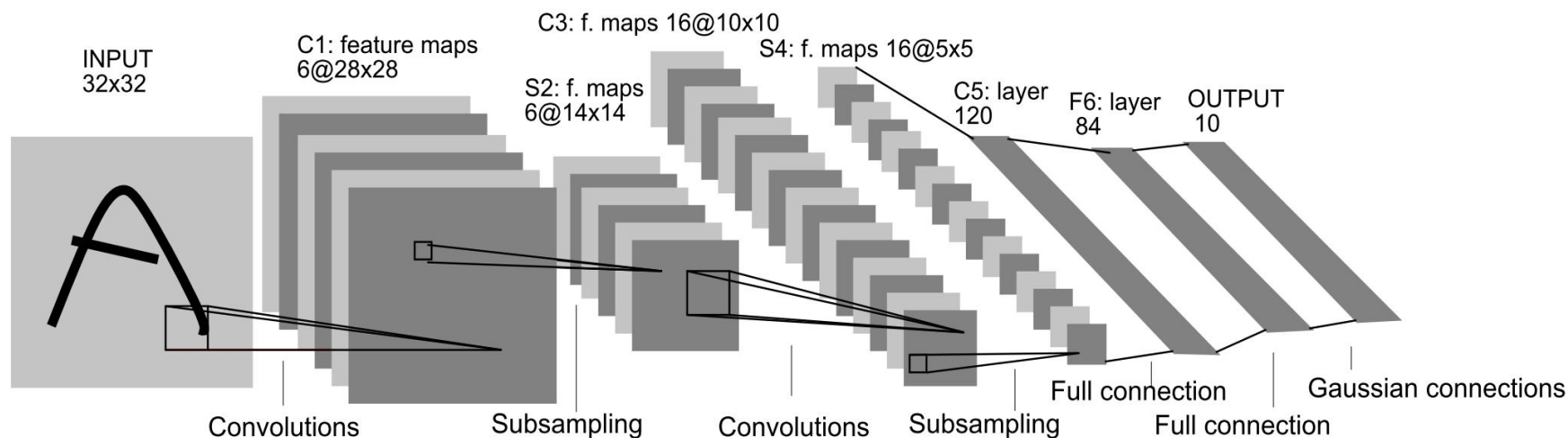
```
print(x.requires_grad)
print((x ** 2).requires_grad)
with torch.no_grad():
    print((x ** 2).requires_grad)
```

输出：

```
True
True
False
```


4. PyTorch 神经网络

- 神经网络可以通过 `torch.nn` 包来构建。
- 现在对于自动梯度(`autograd`)有一些了解，神经网络是基于自动梯度 (`autograd`)来定义一些模型。
- 一个 `nn.Module` 包括层和一个方法 `forward(input)` 它会返回输出(`output`)。
- 例如，看一下数字图片识别的网络：



这是一个简单的前馈神经网络，它接收输入，让输入一个接着一个的通过一些层，最后给出输出。

一个典型的神经网络训练过程包括以下几点：

1. 定义一个包含可训练参数的神经网络
2. 迭代整个输入
3. 通过神经网络处理输入
4. 计算损失(loss)
5. 反向传播梯度到神经网络的参数
6. 更新网络的参数，典型的用一个简单的更新方法： $\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$

定义神经网络

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5
6 class Net(nn.Module):
7     def __init__(self):
8         super(Net, self).__init__()
9         # 1 input image channel, 6 output channels, 5x5 square convolution
10        # kernel
11        self.conv1 = nn.Conv2d(1, 6, 5)
12        self.conv2 = nn.Conv2d(6, 16, 5)
13        # an affine operation:  $y = Wx + b$ 
14        self.fc1 = nn.Linear(16 * 5 * 5, 120)
15        self.fc2 = nn.Linear(120, 84)
16        self.fc3 = nn.Linear(84, 10)
17
18    def forward(self, x):
19        # Max pooling over a (2, 2) window
20        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
21        # If the size is a square you can only specify a single number
22        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
23        x = x.view(-1, self.num_flat_features(x))
24        x = F.relu(self.fc1(x))
25        x = F.relu(self.fc2(x))
26        x = self.fc3(x)
27        return x
```

```

29     def num_flat_features(self, x):
30         size = x.size()[1:] # all dimensions except the batch dimension
31         num_features = 1
32         for s in size:
33             num_features *= s
34         return num_features
35
36
37 net = Net()
38 print(net)
39

```

输出:

```

Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)

```

我们刚定义了一个前馈函数，然后反向传播函数被自动通过 autograd 定义了。你可以使用任何张量操作在前馈函数上。
一个模型可训练的参数可以通过调用 `net.parameters()` 返回：

```
params = list(net.parameters())  
print(len(params))  
print(params[0].size()) # conv1's .weight
```

输出：

```
10  
torch.Size([6, 1, 5, 5])
```

让我们尝试随机生成一个 32x32 的输入。注意：期望的输入维度是 32x32。为了使用这个网络在MNIST 数据集上，你需要把数据集中的图片维度修改为 32x32。

```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

输出：

```
tensor([[[-0.0439, -0.1526, -0.0933, -0.1044, -0.0714, -0.0639, -0.0029, -0.1176,
          -0.0248, -0.1056]], grad_fn=<AddmmBackward>])
```

把所有参数梯度缓存器置零，用随机的梯度来反向传播

```
net.zero_grad()
out.backward(torch.randn(1, 10))
```

在此，我们完成了：

- 1.定义一个神经网络
- 2.处理输入以及调用反向传播

还剩下：

- 1.计算损失值
- 2.更新网络中的权重

损失函数

一个损失函数需要一对输入：模型输出和目标，然后计算一个值来评估输出距离目标有多远。

有一些不同的损失函数在 `nn` 包中。一个简单的损失函数就是 `nn.MSELoss`，这计算了均方误差。

例如：

```
output = net(input)
target = torch.randn(10) # a dummy target, for example
target = target.view(1, -1) # make it the same shape as output
criterion = nn.MSELoss()
loss = criterion(output, target)
print(loss)
```

输出：

```
tensor(2.2861, grad_fn=<MseLossBackward>)
```


现在，如果你跟随损失到反向传播路径，可以使用它的 `.grad_fn` 属性，你会看到一个这样的计算图：

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d  
      -> view -> linear -> relu -> linear -> relu -> linear  
      -> MSELoss  
      -> loss
```

所以，当我们调用 `loss.backward()`，整个图都会微分，而且所有的在图中的 `requires_grad=True` 的张量将会让他们的 `grad` 张量累计梯度。

为了演示，我们将跟随以下步骤来反向传播。

```
print(loss.grad_fn) # MSELoss
print(loss.grad_fn.next_functions[0][0]) # Linear
print(loss.grad_fn.next_functions[0][0].next_functions[0][0]) # ReLU
```

输出：

```
<MseLossBackward object at 0x000001C6DF5CEE08>
<AddmmBackward object at 0x000001C68042A888>
<AccumulateGrad object at 0x000001C6DF5CEE08>
```

反向传播

为了实现反向传播损失，我们所有需要做的事情仅仅是使用 `loss.backward()`。你需要清空现存的梯度，要不然帝都将会和现存的梯度累计到一起。

现在我们调用 `loss.backward()`，然后看一下 `conv1` 的偏置项在反向传播之前和

```
net.zero_grad() # zeroes the gradient buffers of all parameters
print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)
loss.backward()
print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

输出：

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([ 0.0016, -0.0029,  0.0034, -0.0049, -0.0101, -0.0012])
```

现在我们看到了，如何使用损失函数。唯一剩下的事情就是更新神经网络的参数。

更新神经网络参数：最简单的更新规则就是随机梯度下降

$$\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$$

我们可以使用 python 来实现这个规则：

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

尽管如此，如果是使用神经网络，想使用不同的更新规则，类似于 SGD, Nesterov-SGD, Adam, RMSProp, 等。为了让这可行，我们建立了一个小包：torch.optim 实现了所有的方法。使用它非常的简单。

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad() # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step() # Does the update
```

End of this lecture.

Thanks!