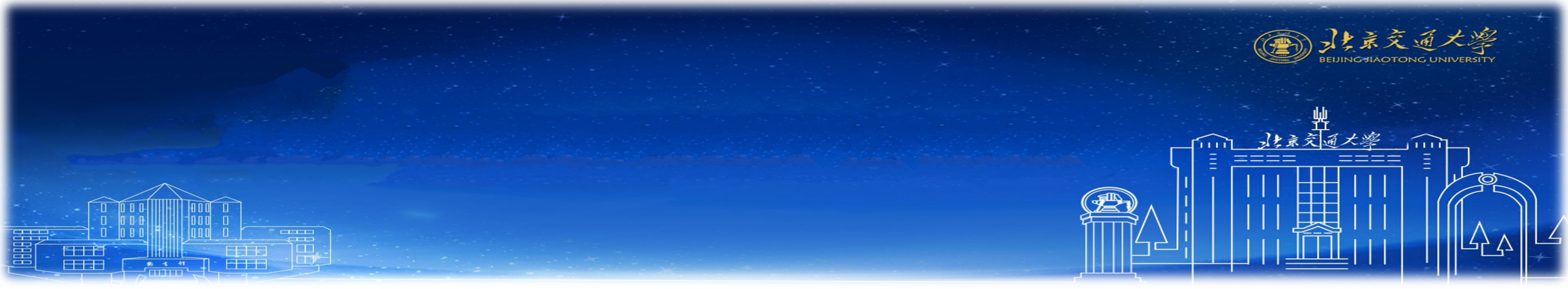
# 实验1 PyTorch基本操作实验

北京交通大学 《深度学习》课程组

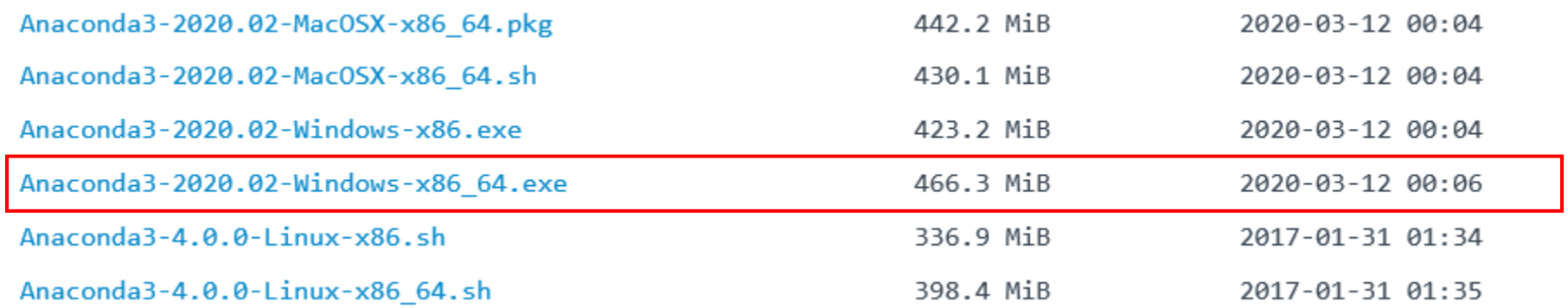


# 目录

# 1.1 Anoconda安装

**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。也可以下载轻量级的**miniconda**(只包含了python和conda)

可以直接从 Anaconda官网[1]下载，但因为Anaconda的服务器在国外，所以下载速度可能会比较慢，也可以从Anaconda仓库网站[2]下载，也可以使用清华的镜像[3]来下载。选择合适你的版本下载，这里选择**Anaconda2020.02-Windowsx86_64.exe**版本.

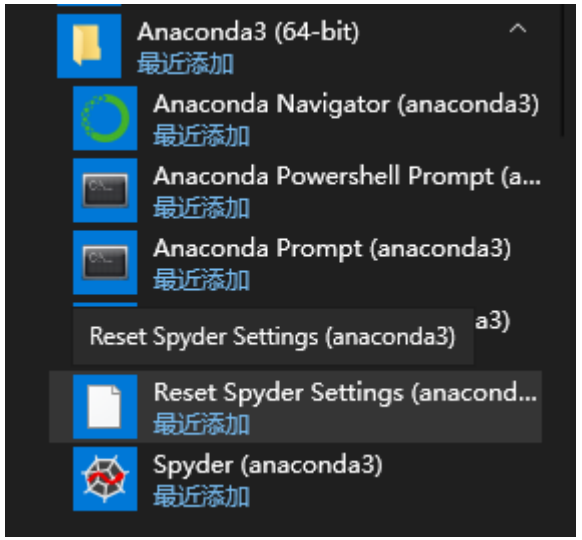


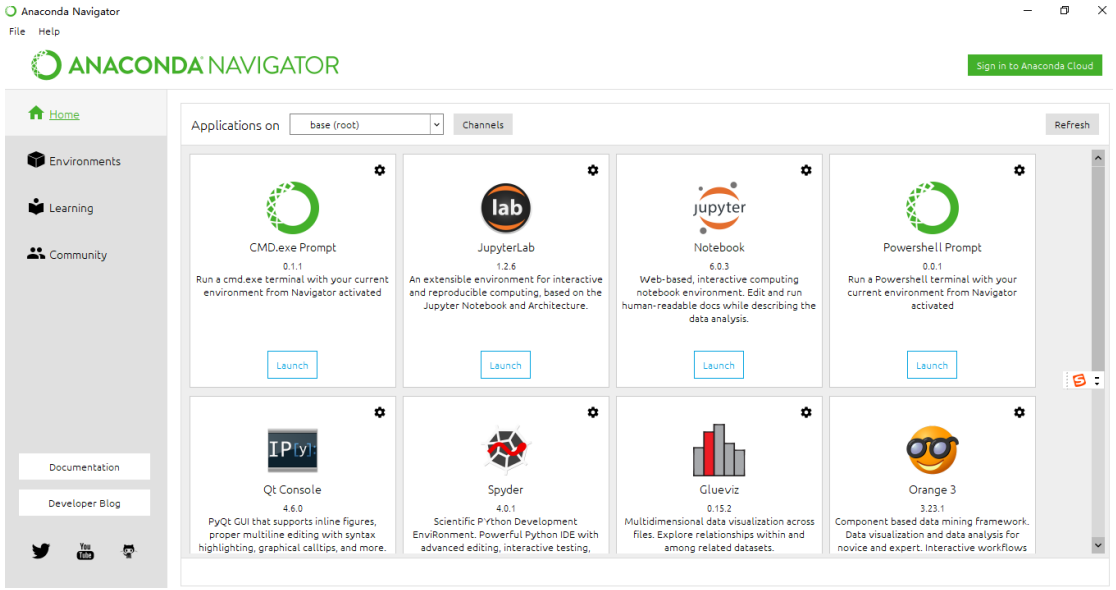
| | | |
|---|---|---|
| Anaconda3-2020.02-MacOSX-x86_64.pkg | 442.2 MiB | 2020-03-12 00:04 |
| Anaconda3-2020.02-MacOSX-x86_64.sh | 430.1 MiB | 2020-03-12 00:04 |
| Anaconda3-2020.02-Windows-x86.exe | 423.2 MiB | 2020-03-12 00:04 |
| Anaconda3-2020.02-Windows-x86_64.exe | 466.3 MiB | 2020-03-12 00:06 |
| Anaconda3-4.0.0-Linux-x86.sh | 336.9 MiB | 2017-01-31 01:34 |
| Anaconda3-4.0.0-Linux-x86_64.sh | 398.4 MiB | 2017-01-31 01:35 |

[1] https://www.anaconda.com/
[2] https://repo.anaconda.com/archive/
[2] https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/

# 1.1 Anoconda安装

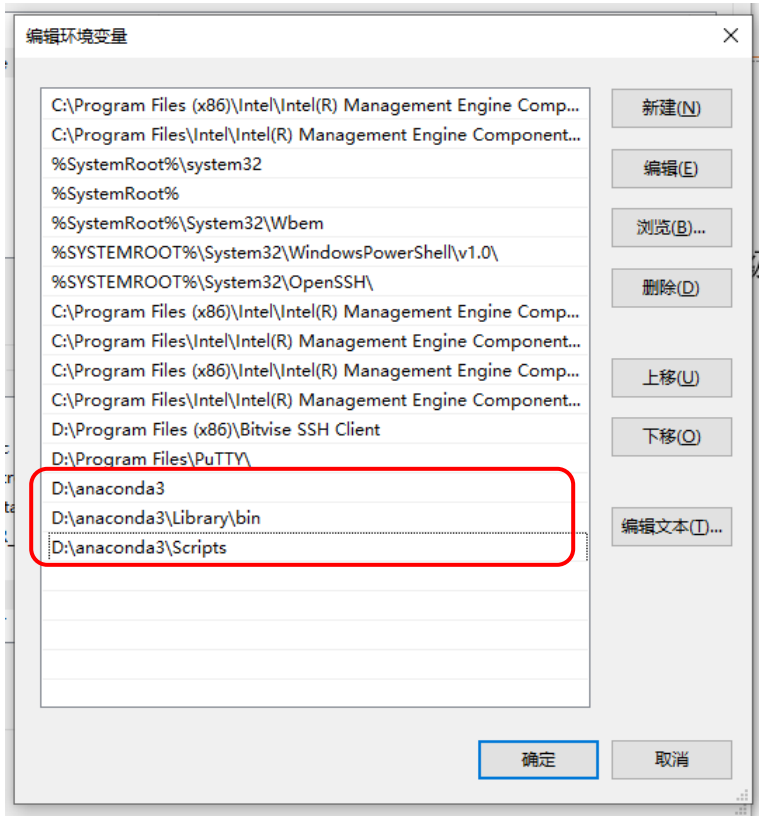
下载之后，点击安装，按照提示依次点击下一步，直到安装结束。安装完成后，在开始菜单会出现安装后的软件，如左图所示。打开程序**Anaconda Navigator**，启动后可以看到**Anaconda**的环境界面，如右图所示。

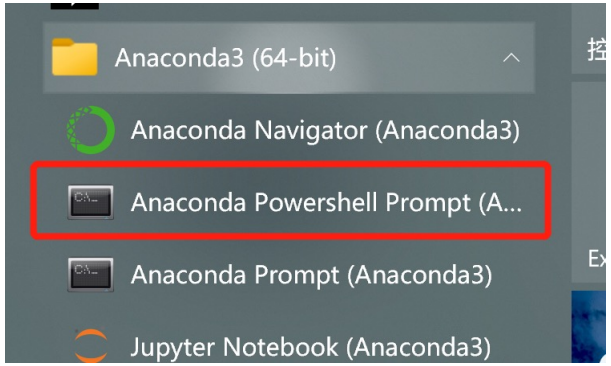


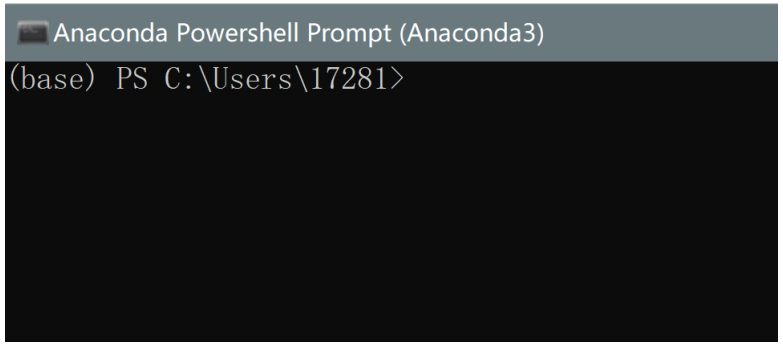



# 1.1 Anoconda安装

安装完成后，进行**Anaconda**的环境变量配置，电脑->属性->高级系统设置->环境变量->系统变量找到Path，点击编辑，加入三个文件夹的存储路径，如左图所示，然后选择确定。点击 **anaconda 控制台**打开，打开命令行，输入**conda info**，显示如右图，说明环境变量配置成功。
这里的(base)表示默认进入的是conda的base环境。



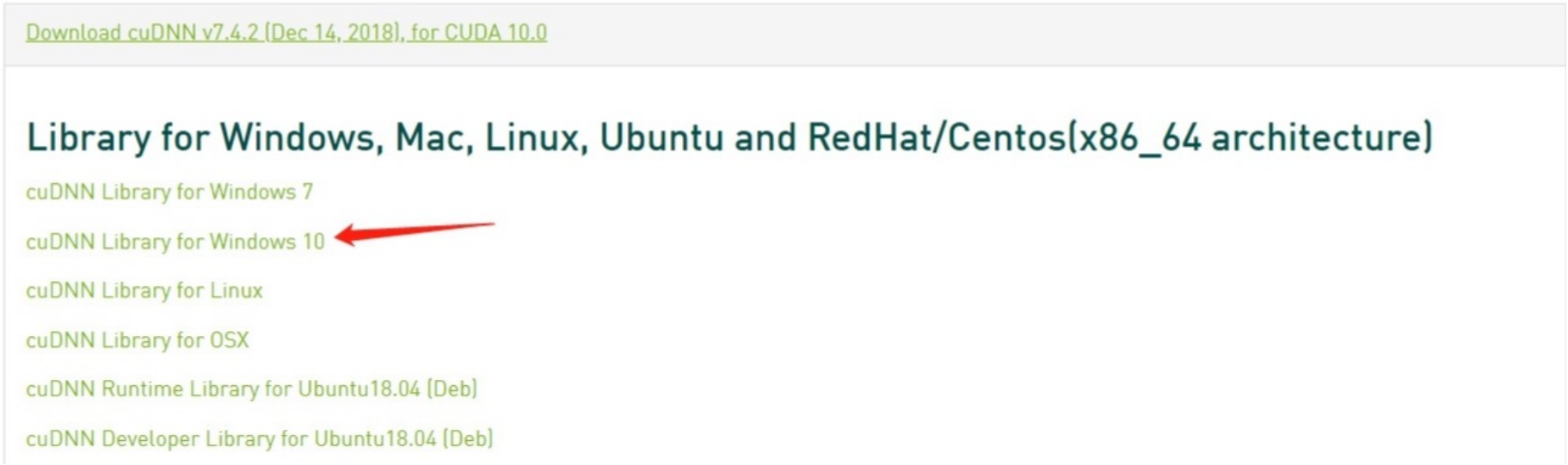
# 1.2 配置GPU

1.安装显卡驱动 ： https://www.nvidia.com/Download/index.aspx?lang=cn


2.安装cuda ： https://developer.nvidia.com/cuda-toolkit-archive

# 1.2 配置GPU

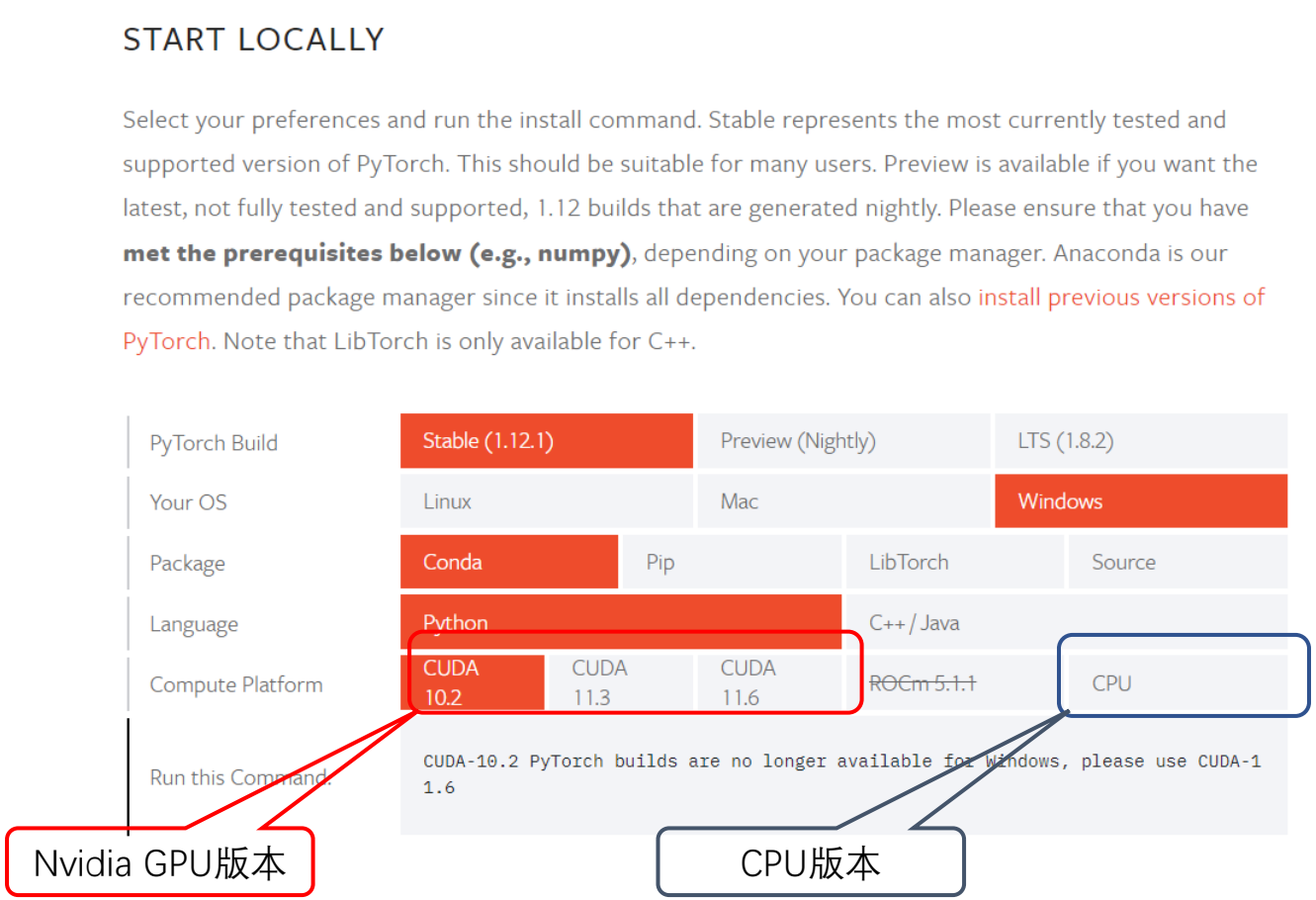
1.安装加速包cuDNN ： https://developer.nvidia.com/cudnn-download-survey

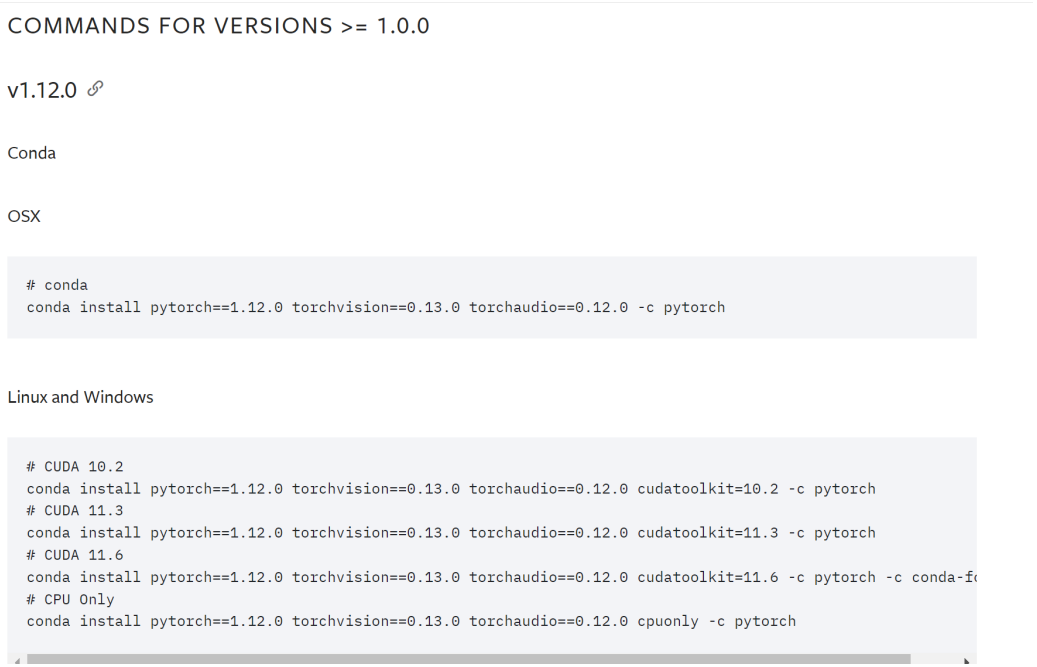



2.下载之后将其解压，将其中的文件夹复制到CUDA中的安装目录中。

# 1.3 Pytorch安装

进入Pytorch官网，官网提供**pip**和**conda**两种安装方式。根据电脑配置进行选择，会生成相应的安装命令。



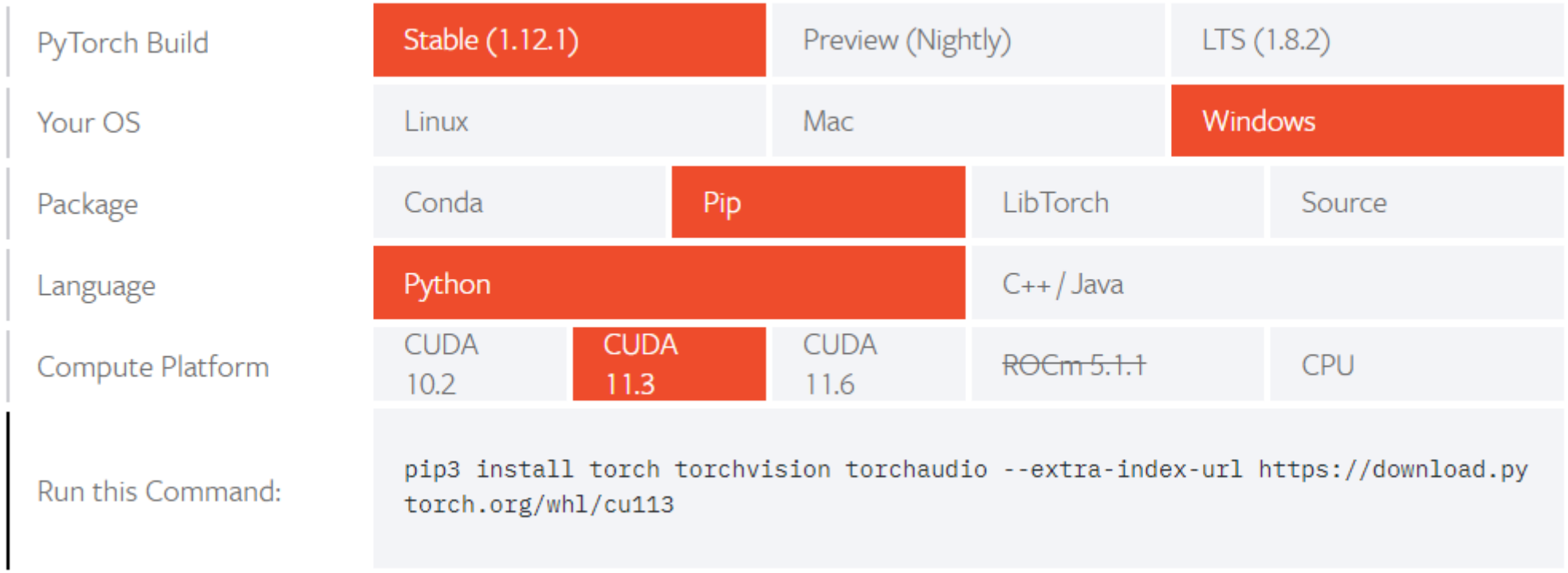




注：安装GPU版本的pytorch时，要查找相应的CUDA版本

# 1.3 Pytorch安装

**pip命令安装：windows+pip+python3.7+None**

复制命令：pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu113 然后在终端中运行



# 1.3 Pytorch安装

**Conda命令安装：windows+Conda+python3.7+None**

复制命令：conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch 然后在终端中运行

| | | | | |
|---|---|---|---|---|
| PyTorch Build | **Stable (1.12.1)** | Preview (Nightly) | LTS (1.8.2) | |
| Your OS | Linux | Mac | **Windows** | |
| Package | **Conda** | Pip | LibTorch | Source |
| Language | **Python** | C++ / Java | | |
| Compute Platform | CUDA 10.2 | **CUDA 11.3** | CUDA 11.6 | ~~ROCm 5.1.1~~ | CPU |
| Run this Command: | `conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch` | | | |

## Pytorch安装验证

安装完pytorch后，输入一下命令进行验证是否正确安装。
首先，输入python，进入python命令行，然后再输入：

```
(tl) C:\WINDOWS\system32>python
Python 3.6.15 (default, Dec  3 2021, 18:25:24) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import torch
>>>
>>> x = torch.rand(2, 3)
>>> x
tensor([[0.1331, 0.5841, 0.4420],
        [0.9782, 0.2066, 0.9182]])
```

如果能运行没有问题，则安装成功

# 1.3 Pytorch安装

## GPU验证

```
(tl) C:\WINDOWS\system32>python
Python 3.6.15 (default, Dec  3 2021, 18:25:24) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.cuda.is_available()
True
```

如果是True，则代表安装的GPU版本没有错误，GPU可用

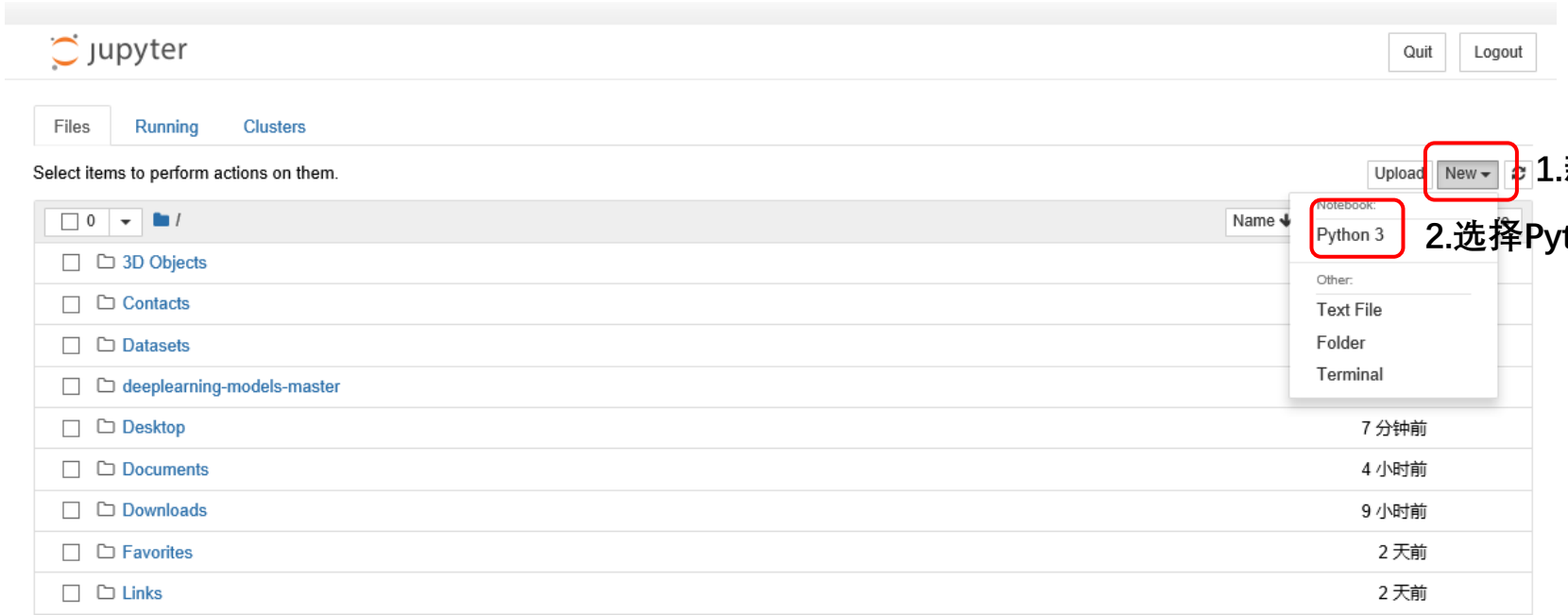
# 1.4 Jupyter Notebook

**Anoconda** 提供了一个交互式笔记本**Jupyter Notebook** ，可以支持运行40多种编程语言。本课程的后续代码在Jupyter Notebook平台上运行以及讲解。

Jupyter Notebook会将cell中的运行结果保存起来。

在cmd命令框中运行：**jupyter notebook**，这时在浏览器打开 http://localhost:8888 （通常会自动打开）

打开**Jupyter Notebook**后，新建python文件：new->python3



# 1.4 Jupyter Notebook

新建完文件后，就可以在空的命令框中编写代码。

验证pytorch可以在jupyter上运行。

```
In [1]: import torch

In [6]: import torchvision
        print(torch.__version__)

        1.10.2

In [5]: a = torch.rand(2, 3)

In [7]: a

Out[7]: tensor([[0.1001, 0.3671, 0.7426],
                [0.7534, 0.2803, 0.3867]])
```

# 目录

# 2 基本数据处理与计算操作

在深度学习中，我们通常会频繁地对数据进行操作。而在PyTorch中，**torch.Tensor**是存储和变换数据的主要工具。**Tensor**和**NumPy**的多维数组非常类似。

然而，**Tensor**提供了GPU计算和自动求梯度等更多功能，这些使**Tensor**更加适合深度学习。

"tensor"这个单词一般可译作“张量”，张量可以看作是一个多维数组。**标量**可以看作是0维张量，**向量**可以看作1维张量，**矩阵**可以看作是2维张量。

## 创建Tensor

- 导入PyTorch

| | |
|---|---|
| In [1]: | import torch |

- 创建一个2x3的未初始化的**Tensor**

| | |
|---|---|
| In [2]: | x = torch.empty(2, 3)<br>print(x) |
| Out [2]: | tensor([[1.1710e+32, 4.5782e-41, 1.1710e+32],<br>[4.5782e-41, 2.1459e+20, 9.2489e-04]]) |

# 2.1 创建Tensor

- 创建一个2x3的随机初始化的**Tensor**

```
In [3]:  x = torch.rand(2, 3)
         print(x)
Out [3]: tensor([[0.8891, 0.7304, 0.1292],
                 [0.8943, 0.6942, 0.1651]])
```

- 创建一个2x3的long型全0的**Tensor**

```
In [4]:  x = torch.zeros(2, 3, dtype=torch.long)
         print(x)
Out [4]: tensor([[0, 0, 0],
                 [0, 0, 0]])
```

- 直接根据数据创建

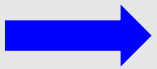
```
In [5]:  x = torch.tensor([[5.5, 3],[2.2,5]])
         print(x)
Out [5]: tensor([[5.5000, 3.0000],
                 [2.2000, 5.0000]])
```

# 2.1 创建Tensor

- 我们还可以通过现有的**Tensor**来创建
  此类方法会默认重用输入**Tensor**的一些属性，例如数据类型，除非自定义数据类型。

```
In [7]: # 返回的tensor默认具有相同的torch.dtype和torch.device
        x = x.new_ones(2, 3)
        print(x)

        # 指定新的数据类型
        x = torch.randn_like(x, dtype=torch.float)
        print(x)
Out [7]: tensor([[1., 1., 1.],
                 [1., 1., 1.]], dtype=torch.float64)
         tensor([[ 0.5344,  0.5095,  0.3691],
                 [ 0.0160,  1.4369,  1.3419]])
```



与x相同的数据类型

- 通过**shape**或者**size()**来获取**Tensor**的形状

```
In [6]: # 返回的torch.Size其实就是一个tuple, 支持所有tuple的操作。
        print(x.size())
        print(x.shape)
Out [6]: torch.Size([2, 3])
         torch.Size([2, 3])
```

# 2.1 创建Tensor

- 其他创建**Tensor**的函数（可查阅官方API[1]）

| 函数 | 功能 |
| --- | --- |
| Tensor(*sizes) | 基础构造函数 |
| tensor(data,) | 类似np.array的构造函数 |
| ones(*sizes) | 全1Tensor |
| zeros(*sizes) | 全0Tensor |
| eye(*sizes) | 对角线为1，其他为0 |
| arange(s,e,step) | 从s到e，步长为step |
| linspace(s,e,steps) | 从s到e，均匀切分成steps份 |
| rand/randn(*sizes) | 均匀/标准分布 |
| normal(mean,std)/uniform(from,to) | 正态分布/均匀分布 |
| randperm(m) | 随机排列 |

[1] https://pytorch.org/docs/stable/index.html

# 2.2 Tensor的相关操作

- 算术操作

在PyTorch中，同一种操作可能有很多种形式，下面用加法作为例子

In [1]:

```
# 加法形式一
x = torch.rand(2, 3)
y = torch.rand(2, 3)
print(x + y)

# 加法形式二
print(torch.add(x, y))

# 加法形式三，inplace（原地操作）
y.add_(x)
print(y)

# PyTorch操作的inplace版本都有后缀“_”, 例如x.copy_(y), x.t_()
```

Out [1]:

```
tensor([[0.6693, 0.6933, 0.6357],
        [1.2661, 1.2059, 1.4899]])
……
```

# 2.2 Tensor的相关操作

- 索引

我们还可以使用类似NumPy的索引操作来访问**Tensor**的一部分
需要注意的是：**索引出来的结果与原数据共享内存，也即修改一个，另一个会跟着修改。**

```
In [2]: y = x[0, :]
        y += 1
        print(y)
        print(x[0, :]) # 源tensor也被改了
Out [2]: tensor([1.0910, 1.5265, 1.3833])
         tensor([1.0910, 1.5265, 1.3833])
```

其他更高级的选择函数（可查阅官方API）

| 函数 | 功能 |
| --- | --- |
| index_select(input, dim, index) | 在指定维度dim上选取，比如选取某些行、某些列 |
| masked_select(input, mask) | 例子如上，a[a>0]，使用ByteTensor进行选取 |
| nonzero(input) | 非0元素的下标 |
| gather(input, dim, index) | 根据index，在dim维度上选取数据，输出的size与index一样 |

# 2.2 Tensor的相关操作

- 改变形状

用view()来改变Tensor的形状

| | |
|---|---|
| In [3]: | y = x.view(6)<br>z = x.view(-1, 2) # -1所指的维度可以根据其他维度的值推出来<br>print(x.size(), y.size(), z.size()) |
| Out [3]: | torch.Size([2, 3]) torch.Size([6]) torch.Size([3, 2]) |

注意**view()**返回的新**Tensor**与源**Tensor**虽然可能有不同的**size**，但是是共享**data**的，也即更改其中的一个，另外一个也会跟着改变。(顾名思义，view仅仅是改变了对这个张量的观察角度，内部数据并未改变)

| | |
|---|---|
| In [4]: | x += 1<br>print(x)<br>print(y) # y也加了1 |
| Out [4]: | tensor([[2.0910, 2.5265, 2.3833],<br>[1.4564, 1.3117, 1.5181]])<br>tensor([2.0910, 2.5265, 2.3833, 1.4564, 1.3117, 1.5181]) |

# 2.2 Tensor的相关操作

如果我们想返回一个真正新的副本（即不共享data内存）该怎么办呢？

Pytorch还提供了一个**reshape()**方法可以改变形状，但是此函数并不能保证返回的是其拷贝，所以不推荐使用。我们推荐先用**clone()**创造一个副本然后再使用**view()**。

| | |
|---|---|
| In [5]: | x_cp = x.clone().view(6)<br>x -= 1<br>print(x)<br>print(x_cp) |
| Out [5]: | tensor([[1.0910, 1.5265, 1.3833],<br>[0.4564, 0.3117, 0.5181]])<br>tensor([2.0910, 2.5265, 2.3833, 1.4564, 1.3117, 1.5181]) |

# 2.2 Tensor的相关操作

另外，PyTorch还支持一些线性函数，具体用法可参考官方文档。

| 函数 | 功能 |
| --- | --- |
| trace | 对角线元素之和(矩阵的迹) |
| diag | 对角线元素 |
| triu/tril | 矩阵的上三角/下三角，可指定偏移量 |
| mm/bmm | 矩阵乘法，batch的矩阵乘法 |
| addmm/addbmm/addmv/addr/baddbmm.. | 矩阵运算 |
| t | 转置 |
| dot/cross | 内积/外积 |
| inverse | 求逆矩阵 |
| svd | 奇异值分解 |

# 2.3 广播机制

当我们对两个形状不同的**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

```
In [1]: x = torch.arange(1, 3).view(1, 2)
        print(x)
        y = torch.arange(1, 4).view(3, 1)
        print(y)
        print(x + y)
Out [1]: tensor([[1, 2]])
        tensor([[1],
                [2],
                [3]])
        tensor([[2, 3],
                [3, 4],
                [4, 5]])
```

计算过程触发了广播机制

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。

# 2.4 Tensor和NumPy相互转换

我们可以使用**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！**

还有一个常用的将NumPy中的array转换成**Tensor**的方法就是**torch.tensor()**，需要注意的是，此方法总是会进行数据拷贝（就会消耗更多的时间和空间），所以返回的**Tensor**和原来的数据不再共享内存。

## Tensor转NumPy数组

| | |
|---|---|
| In [1]: | a = torch.ones(3)<br>b = a.numpy()<br>print(a, b)<br><br>a += 1<br>print(a, b)<br>b += 1<br>print(a, b) |
| Out [1]: | tensor([1., 1., 1.]) [1. 1. 1.]<br>tensor([2., 2., 2.]) [2. 2. 2.]<br>tensor([3., 3., 3.]) [3. 3. 3.] |

# 2.4 Tensor和NumPy相互转换

## NumPy数组转Tensor

| | |
|---|---|
| In [1]: | import numpy as np<br>a = np.ones(3)<br>b = torch.from_numpy(a)<br>print(a, b)<br><br>a += 1<br>print(a, b)<br>b += 1<br>print(a, b) |
| Out [1]: | [1. 1. 1.] tensor([1., 1., 1.], dtype=torch.float64)<br>[2. 2. 2.] tensor([2., 2., 2.], dtype=torch.float64)<br>[3. 3. 3.] tensor([3., 3., 3.], dtype=torch.float64) |

使用torch.tensor()将NumPy数组转换成Tensor（不再共享内存）

| | |
|---|---|
| In [2]: | c = torch.tensor(a)<br>a += 1<br>print(a, c) |
| Out [2]: | [4. 4. 4.] tensor([3., 3., 3.], dtype=torch.float64) |

# 2.5 Tensor on GPU

用方法**to()**可以将Tensor在CPU和GPU（需要硬件支持）之间相互移动。

In [3]:

```
# 以下代码只有在PyTorch GPU版本上才会执行
if torch.cuda.is_available():
    device = torch.device("cuda")          # GPU
    y = torch.ones_like(x, device=device)  # 直接创建一个在GPU上的Tensor
    x = x.to(device)                       # 等价于 .to("cuda")
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # to()还可以同时更改数据类型
```

Out [3]:

```
tensor([[2, 3]], device='cuda:0')
tensor([[2., 3.]], dtype=torch.float64)
```

第一条数据位于GPU上
第二条数据位于CPU上

# 2.6 自动求梯度

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。

**Tensor**是**autograd**包的核心类，如果将其属性**.requires_grad**设置为**True**，它将开始追踪(track)在其上的所有操作（这样就可以利用链式法则进行梯度传播了）。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。

如果不想要被继续追踪，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，这样梯度就传不过去了。此外，还可以用**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，因为在评估模型时，我们并不需要计算可训练参数（**requires_grad=True**）的梯度。

**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。

## Tensor

创建一个Tensor并设置requires_grad=True:

```
In [1]:  x = torch.ones(2, 2, requires_grad=True)
         print(x)
         print(x.grad_fn)
Out [1]: tensor([[1., 1.],
                 [1., 1.]], requires_grad=True)
         None
```

x是直接创建的，所以它没有grad_fn

做运算操作：

```
In [2]:  y = x + 2
         print(y)
         print(y.grad_fn)
Out [2]: tensor([[3., 3.], [3., 3.]], grad_fn=<AddBackward0>)
         <AddBackward0 object at 0x7fab97795828>
```

y是通过一个加法操作创建的，所以它有一个为<AddBackward>的grad_fn

# 2.6 自动求梯度

更复杂的运算操作：

| | |
|---|---|
| In [3]: | z = y * y * 3<br>out = z.mean()<br>print(z)<br>print(out) |
| Out [3]: | tensor([[27., 27.], [27., 27.]], grad_fn=<MulBackward0>)<br>tensor(27., grad_fn = <MeanBackward0>) |

通过**.requires_grad_()**来用in-place的方式改变**requires_grad**属性：

| | |
|---|---|
| In [4]: | a = torch.randn(2, 2)  # 缺失情况下默认 requires_grad = False<br>a = ((a * 3) / (a - 1))<br>print(a.requires_grad)  **# False**<br>a.requires_grad_(True)<br>print(a.requires_grad)  **# True**<br>b = (a * a).sum()<br>print(b.grad_fn) |
| Out [4]: | False<br>True<br><SumBackward0 object at 0x7fab977a33c8> |

# 2.6 自动求梯度

## 梯度

| | |
|---|---|
| In [1]: | x = torch.ones(2, 2, requires_grad=True)<br>y = x + 2<br>z = y * y * 3<br>out = z.mean()<br>out.backward()  # 等价于 out.backward(torch.tensor(1.))<br><br>print(x.grad) |
| Out [1]: | tensor([[4.5000, 4.5000],<br>[4.5000, 4.5000]]) |

令**out**为$o$，因为

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

所以

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

# 2.6 自动求梯度

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都需要把梯度**清零**。

In [2]:
```
# 再来反向传播一次，注意grad是累加的
out2 = x.sum()
out2.backward()
print(x.grad)

out3 = x.sum()
x.grad.data.zero_()
out3.backward()
print(x.grad)
```

Out [2]:
```
tensor([[5.5000, 5.5000],
        [5.5000, 5.5000]])

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

→ 梯度未清零，累加梯度

→ 梯度清零后，x的梯度为1

# 2.6 自动求梯度

注意在**y.backward()**时，如果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。这样的原因简单来说就是为了避免向量（甚至更高维张量）对张量求导，因此转换成标量对张量进行求导。

■ 举例

假设形状为$m \times n$ 的矩阵 $\mathbf{X}$ 经过运算得到了$p \times q$的矩阵 $\mathbf{Y}$，$\mathbf{Y}$ 又经过运算得到了$s \times t$的矩阵 $\mathbf{Z}$。那么按照梯度计算的规则，$\frac{dZ}{dY}$ 应该是一个 $s \times t \times p \times q$ 四维张量，$\frac{dY}{dX}$ 是一个 $p \times q \times m \times n$ 的四维张量，而如何对4维张量进行计算是一个比较复杂的问题。

为了避免这个问题，我们**不允许张量对张量求导，只允许标量对张量求导，求导结果是和自变量同形的张量。**所以必要时我们要把张量通过将所有张量的元素加权求和的方式转换为标量，举个例子，假设 $y$ 由自变量 $x$ 计算而来，$w$ 是和 $y$ 同形的张量，则 $\mathrm{y.backward}(w)$ 的含义是：先计算 $l = \mathrm{torch.sum}(y * w)$，因此 $l$ 是一个标量，然后我们再求 $l$ 对自变量 $x$ 的导数。

# 2.6 自动求梯度

- **y.backward()**的具体例子

| | |
|---|---|
| In [1]: | x = torch.tensor([1.0, 2.0, 3.0, 4.0], requires_grad=True)<br>y = 2 * x<br>z = y.view(2, 2)<br>print(z) |
| Out [1]: | tensor([[2., 4.],<br>[6., 8.]], grad_fn=<ViewBackward>) |

- 此时 $z$ 不是一个标量，所以在调用**backward()**时需要传入一个和 $z$ 同形的权重向量进行加权求和来得到一个标量。

| | |
|---|---|
| In [2]: | v = torch.tensor([[1.0, 0.1], [0.01, 0.001]], dtype=torch.float)<br>z.backward(v)<br>print(x.grad) |
| Out [2]: | tensor([2.0000, 0.2000, 0.0200, 0.0020]) |

# 目录

## 基本概念

- **线性回归**（Linear Regression）是机器学习和统计学中最基础和广泛应用的模型，是一种对自变量和因变量之间关系进行建模的回归分析。

- 从机器学习的角度来看，自变量就是样本的特征向量 $\mathrm{x} \in R^d$（每一维对应一个自变量），因变量是标签 $y$，这里 $\mathrm{y} \in R$ 是连续值。假设空间是一组参数化的线性函数

$$f(\mathrm{x}; w, b) = xw^T + b,$$

- 其中，权重向量 $\boldsymbol{w}$ 和偏置 $\boldsymbol{b}$ 是线性回归需要学习的参数， 函数 $f(\mathrm{x}; w, b) \in R$ 称为**线性模型**。

## 实现线性回归

- 使用Tensor 和 autograd来实现一个线性回归，具体的步骤有：
  - 生成和读取数据集
  - 构建模型
  - 初始化模型参数
  - 定义损失函数和优化算法
  - 训练模型

- 导入本次实验所需的包或模块，其中matplotlib包可用于作图，用来显示生成的数据的二维图。

```
1  import torch
2  from IPython import display
3  from matplotlib import pyplot as plt
4  import numpy as np
5  import random
```

# 3.1 手动实现线性回归

## 生成数据

- 本实验先构造一个简单训练数据集，通过这个数据集可以直观的比较模型训练出来的参数和真实的模型参数的区别。设训练数据集样本数为1000，输入个数（**特征数**）为2，给定随机生成的批量样本特征 $\mathrm{X} \in \mathbb{R}^{1000\times 2}$。这里使用线性回归模型的**真实权重** $\mathrm{w} = [2, -3.4]$和**偏差** $b = 4.2$, 以及一个随机噪声项$\varepsilon$ 来生成标签。

$$y = xw^T + b + \varepsilon$$

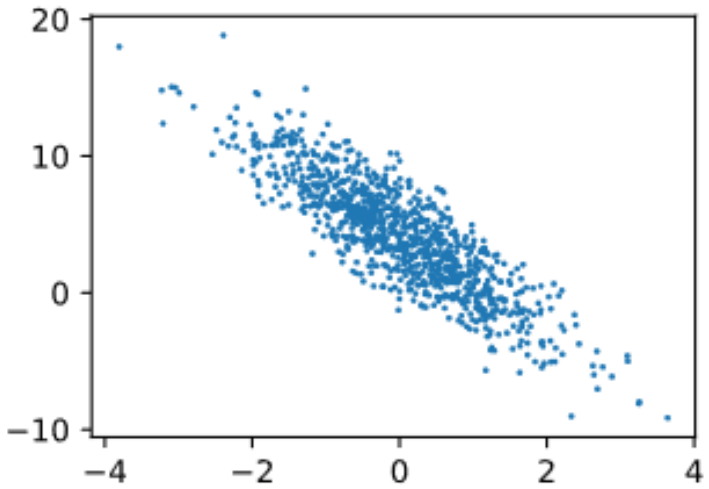
- 其中，噪声项$\varepsilon$服从均值0、标准差为0.01的正态分布。噪声代表了数据中无意义的干扰。

```
1  num_inputs = 2
2  num_examples = 1000
3  true_w = [2, -3.4]
4  true_b = 4.2
5  features = torch.tensor(np.random.normal(0, 1, (num_examples, num_inputs)), dtype=torch.float)
6  labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
7  labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.float)
```

# 3.1 手动实现线性回归

- 使用 **use_svg_paly()** 和 **set_figsize()** 两个函数来可视化所生成的数据。

```
1  def use_svg_display():
2      # 用矢量图显示
3      display.set_matplotlib_formats('svg')
4  def set_figsize(figsize=(3.5, 2.5)):
5      use_svg_display()
6      # 设置图的尺寸
7      plt.rcParams['figure.figsize'] = figsize
8  set_figsize()
9  plt.scatter(features[:, 1].numpy(), labels.numpy(), 1);
```



## 读取数据

- 在模型训练的时候，需要遍历数据集并不断读取小批量的数据样本。这里本实验定义一个函数 **data_iter()** 它每次返回 ***batch_size*** （批量大小）个随机样本的特征和标签。

```
1  def data_iter(batch_size, features, labels):
2      num_examples = len(features)
3      indices = list(range(num_examples))
4      random.shuffle(indices)  # 样本的读取顺序是随机的
5      for i in range(0, num_examples, batch_size):
6          j = torch.LongTensor(indices[i: min(i + batch_size, num_examples)]) # 最后一次可能不足一个batch
7          yield  features.index_select(0, j), labels.index_select(0, j)
8
```

# 3.1 手动实现线性回归

## 构建模型

- 在构建模型之前，需要将权重和偏置初始化。本实验将权重初始化成**均值**为0、**标准差**为0.01的正态随机数，偏置初始化为0。

```
w = torch.tensor(np.random.normal(0, 0.01, (num_inputs, 1)), dtype=torch.float32)
b = torch.zeros(1, dtype=torch.float32)
```

- 在后面的模型训练中，需要对这些参数求梯度来迭代参数的值，因此要设置requires_grad = True

```
w.requires_grad_(requires_grad=True)
b.requires_grad_(requires_grad=True)
```

- 下面使用 **mm()** 函数做矩阵乘法，来实现线性回归的模型。

```
def linreg(X, w, b):
    return torch.mm(X, w) + b
```

# 3.1 手动实现线性回归

## 损失函数和优化算法

- 本实验使用平方损失来定义线性回归的损失函数。在实现中，我们需要把**真实值** $y$ 变形成**预测值** $y\_hat$ 形状。以下的函数返回的结果和$y_hat$的形状相同。

```
1  def squared_loss(y_hat, y):
2      return (y_hat - y.view(y_hat.size())) ** 2 / 2
```

- 以下的 sgd 函数实现了小批量随机梯度下降算法。它通过不断迭代模型参数来优化损失函数。这里自动求梯度模块计算得到的梯度是一个批量样本的梯度和。我们将它除以批量大小来得到平均值。

```
1  def sgd(params, lr, batch_size):
2      for param in params:
3          param.data -= lr * param.grad / batch_size # 注意这里更改param时用的param.data
```

## 模型训练

- 在训练过程中，模型将会多次迭代更新参数。在每次迭代中，根据当前读取的小批量数据样本（特征 x 和标签 y），通过调用反向函数**backward**计算小批量随机梯度，并调用优化算法 **sgd**迭代模型参数。

```
1   lr = 0.03
2   num_epochs = 3
3   batch_size = 10
4   net = linreg
5   loss = squared_loss
6
7   for epoch in range(num_epochs):       # 训练模型一共需要num_epochs个迭代周期
8       # 在每一个迭代周期中，会使用训练数据集中所有样本一次
9       for X, y in data_iter(batch_size, features, labels):       # x和y分别是小批量样本的特征和标签
10          l = loss(net(X, w, b), y).sum()     # l是有关小批量X和y的损失
11          l.backward()     # 小批量的损失对模型参数求梯度
12          sgd([w, b], lr, batch_size)     # 使用小批量随机梯度下降迭代模型参数
13          w.grad.data.zero_()    # 梯度清零
14          b.grad.data.zero_()
15      train_l = loss(net(features, w, b), labels)
16      print('epoch %d, loss %f' % (epoch + 1, train_l.mean().item()))
```

# 3.1 手动实现线性回归

- 训练完成后，打印出学到的参数和用来生成训练集的真实参数，通过比较发现它们之间非常接近。

```
1  print(true_w, '\n', w)
2  print(true_b, '\n', b)
```

- 输出的结果：

```
1  [2, -3.4]
2  tensor([[ 2.0000], [-3.3996]], requires_grad=True)
3  4.2
5  tensor([4.1995], requires_grad=True)
```

## Torch.nn 模块简介

- **torch.nn** 模块是Pytorch为神经网络设计的模块化接口， 该模块定义了大量的神经网络层。**nn** 利用 **autograd** 来定义模型，其核心数据结构是 **Module**。
- 下表给出了部分 **nn** 中所包含模块（其它模块可查阅官方API）：

| 模块 | 作用 |
| --- | --- |
| torch.nn.Module() | Module是所有神经网络模块的基类 |
| torch.nn.Linear() | Liner用于对输入数据进行线性变换 |
| torch.nn.Sequential() | Sequential是一个顺序容器, 其中模块的添加顺序与在构造函数中传递模块时的顺序相同 |
| torch.nn.MSELoss | MSELoss用于衡量输入x和目标y中每个元素之间的均方误差的标准。 |

# 3.2 利用torch.nn实现线性回归

## 读取数据

- PyTorch提供了 **data** 库来读取数据。由于data常用作变量名，这里将导入的 **data** 模块用 **Data** 代替。对前面的读取数据部分可以使用 **data** 库来处理。在每一次迭代中，使用 **Data** 随机读取包含10个数据样本的小批量。

```
lr = 0.03
import torch.utils.data as Data
batch_size = 10
# 将训练数据的特征和标签组合
dataset = Data.TensorDataset(features, labels)

# 把 dataset 放入 DataLoader
data_iter = Data.DataLoader(
    dataset=dataset,      # torch TensorDataset format
    batch_size=batch_size,      # mini batch size
    shuffle=True,               # 是否打乱数据 (训练集一般需要进行打乱)
    num_workers=2,              # 多线程来读数据，注意在Windows下需要设置为0
)
```

# 3.2 利用torch.nn实现线性回归

## 构建模型

- 构建模型的过程中，最常见的方法就是继承 **nn.Module**,然后构建自己的网络。一个 **nn.Module** 实例需要包含一些层以及返回输出的前向传播方法。下面利用 **nn.Module** 构建一个线性回归模型。

```
1   class LinearNet(nn.Module):
2       def __init__(self, n_feature):
3           super(LinearNet, self).__init__()
4           self.linear = nn.Linear(n_feature, 1)
5
6   # forward 定义前向传播
7       def forward(self, x):
8           y = self.linear(x)
9           return y
10
11  net =LinearNet(num_inputs)
```

# 3.2 利用torch.nn实现线性回归

- 除了继承 **nn.Moudle** 来构建线性回归模型，还可以利用 **nn.Sequential** 结合 **nn.Linear** 来搭建模型

```
#写法一
net = nn.Sequential(
    nn.Linear(num_inputs, 1)
    # 此处可以添加其它层
)

# 写法二
# net = nn.Sequential()
# net.add_module('linear', nn.Linear(num_inputs, 1))
# net.add_module……

# 写法三
# from collections import OrderedDict
# net = nn.Sequential(OrderedDict([
#                 ('linear', nn.Linear(num_inputs, 1))
#             # ……]))
```

# 3.2 利用torch.nn实现线性回归

## 模型参数初始化

- 在使用定义的模型net之前，需要对模型中的一些参数进行初始化。Pytorch在**init**模块中提供了许多初始化参数的方法。我们可以调用**init.normal**模块通过正态分布对线性回归中的权重和偏差进行初始化

```
1  from torch.nn import init
2
3  init.normal_(net[0].weight, mean=0, std=0.01)
4  init.constant_(net[0].bias, val=0)  #也可以直接修改bias的data：net[0].bias.data.fill_(0)
```

上述代码，将参数中的每个元素随机初始化为了均值为0，标准差为0.01的正态分布，同时将偏差初始化为零

# 3.2 利用torch.nn实现线性回归

## 损失函数和优化

- Pytorch在 $torch.nn$ 中提供了各种损失函数，这些损失函数实现为 **nn.Module** 的子类，可以将这些损失函数作为一种特殊的层。

```
1  loss = nn.MSELoss()
```

- Pytorch在 **torch.optim** 模块中提供了诸如 $\boldsymbol{SGD}$ ， $\boldsymbol{Adam}$ 和 $\boldsymbol{RMSProop}$ 等优化算法。本例将使用小批量随机梯度下降算法进行优化。

```
1  import torch.optim as optim
2  optimizer = optim.SGD(net.parameters(), lr=0.03)  #梯度下降的学习率指定为0.03
3
4  # 可以为不同的子网络设置不同学习率
5  # optimizer =optim.SGD([
6  #     # 如果不指定学习率，则用默认的最外层学习率
7  #     {'params': net.subnet1.parameters()}, # lr=0.03
8  #     {'params': net.subnet2.parameters(), 'lr': 0.01}
9  # ], lr=0.03)
```

# 3.2 利用torch.nn实现线性回归

## 模型训练

- 训练模型时，可以调用 **optim** 中的 **step()** 函数来迭代模型参数。

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        output = net(X)
        l = loss(output, y.view(-1, 1))
        optimizer.zero_grad() # 梯度清零，等价于net.zero_grad()
        l.backward()
        optimizer.step()
    print('epoch %d, loss: %f' % (epoch, l.item()))
```

- 部分输出如下所示

```
epoch 1, loss: 0.000457
epoch 2, loss: 0.000081
epoch 3, loss: 0.000198
```

# 3.3 常用损失函数

## 平均绝对误差 (MAE)

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n} H(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}^{(i)} - y^{(i)}|$$

- torch.nn中的实现

```
1  loss = nn.L1Loss()
```

## 均方误差 (MSE)

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n} H(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{n}\sum_{i=1}^{n} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

- torch.nn中的实现

```
1  loss = nn.MSELoss()
```

# 3.3 常用损失函数

## 二元交叉熵损失函数

- 假设训练数据集的样本数为n，根据二元交叉熵损失函数的定义

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n} H(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}\log\hat{y}^{(i)} + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})$$

- torch.nn中的实现

```
1  loss = nn.BCELoss ()
```

## 包含sigmoid层的二元交叉熵损失函数

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n} H(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}\log\sigma(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \sigma(\hat{y}^{(i)}))$$

- torch.nn中的实现

```
1  loss = nn.BCEWithLogitsLoss() # 包含了sigmoid运算
```

# 3.3 常用损失函数

## 交叉熵损失函数

- 假设训练数据集的样本数为n，根据交叉熵损失函数的定义：

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n} H(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log \hat{y}^{(i)}$$

- 我们给出交叉熵损失函数的实现：

```
# 交叉熵损失函数
def cross_entropy(y_hat, y):
    """
    Parameters
    ----------
    y_hat: 预测值，shape is (batch_size, class_num)
    y: 真值，shape is (batch_size, )

    Returns
    -------
    """
    return - torch.log(y_hat.gather(1, y.view(-1, 1)))

```

# 3.3 常用损失函数

## 交叉熵损失函数

- 假设训练数据集的样本数为n，根据交叉熵损失函数的定义

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n} H(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}\log\hat{y}^{(i)}$$

- torch.nn中的实现

```
1    loss = nn.CrossEntropyLoss()
```

- 值得注意的是，分开定义 **softmax** 运算和交叉熵损失函数可能会造成数值不稳定。因此，PyTorch提供了一个包括 **softmax** 运算和交叉熵损失计算的函数。它的数值稳定性更好。所以在输出层，我们不需要再进行 **softmax** 操作：

```
1    def net(X):
2        return softmax(torch.mm(X.view((-1, num_inputs)), W) + b)
```

```
1    def net(X):
2        return torch.mm(X.view((-1, num_inputs)), W) + b
```

## 模型预测及评价（分类问题）

- 对于分类问题，给定任一样本特征，模型可以预测每个输出类别的概率。通常，我们把预测概率最大的类别作为输出类别。如果它与真实类别（标签）一致，说明这次预测是正确的。我们使用准确率（accuracy）来评价模型的表现。它等于正确预测数量与总预测数量之比。

- 下面我们给出准确率计算函数的实现

```
1  def accuracy(y_hat, y):
2      return (y_hat.argmax(dim=1) == y).float().mean().item()
```

- 评价模型 **net** 在数据集 **data_iter** 上的准确率

```
1  def evaluate_accuracy(data_iter, net):
2      acc_sum, n = 0.0, 0
3      for X, y in data_iter:
4          acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()
5          n += y.shape[0]
6      return acc_sum / n
```

# 目录

## 人工构造的数据集

- 导入所需要的报或模块

```
1  import torch
2  import matplotlib.pyplot as plt
```

- 随机生成实验所需要的二分类数据集 $x1$ 和 $x2$ ，分别对应的标签 $y1$ 和 $y2$ .

```
1  n_data = torch.ones(50, 2)  # 数据的基本形态
2  x1 = torch.normal(2 * n_data, 1)  # shape=(50, 2)
3  y1 = torch.zeros(50)  # 类型0 shape=(50, 1)
4  x2 = torch.normal(-2 * n_data, 1)  # shape=(50, 2)
5  y2 = torch.ones(50)  # 类型1 shape=(50, 1)
6
7  # 注意 x, y 数据的数据形式一定要像下面一样 (torch.cat 是合并数据)
8  x = torch.cat((x1, x2), 0).type(torch.FloatTensor)
9  y = torch.cat((y1, y2), 0).type(torch.FloatTensor)
```

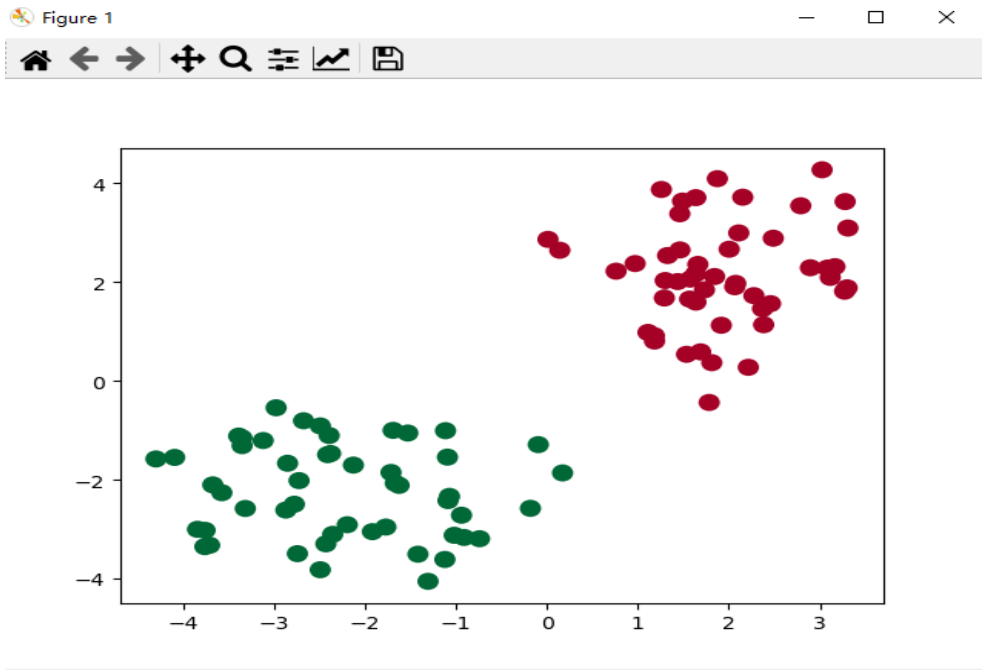
# 4.1 数据集介绍

## 人工构造的数据集

- 使用matplotlib将所构造的数据可视化

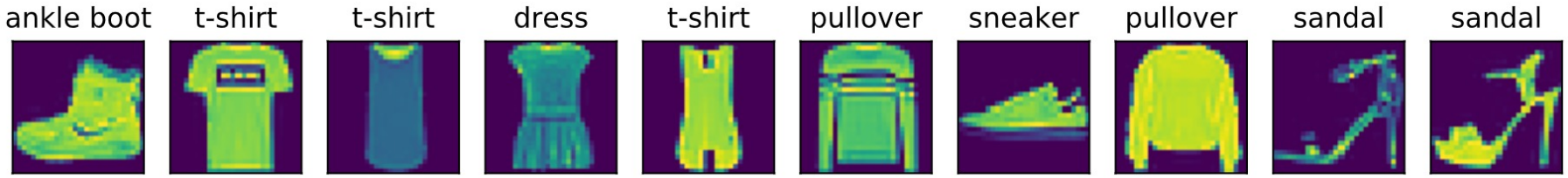
```
1  plt.scatter(x.data.numpy()[:, 0], x.data.numpy()[:, 1], c=y.data.numpy(), s=100, lw=0,
2  cmap='RdYlGn')
3  plt.show()
```

- 打印结果



# 4.1 数据集介绍

## Fashion-MNIST数据集（一个多类图像分类数据集）




- 训练集：60,000
- 测试集：10,000
- 每个样本的数据格式为：28*28*1（高*宽*通道）
- 类别（10类）：dress（连衣裙）、coat（外套）、sandal（凉鞋）、shirt（衬衫）、sneaker（运动鞋）、bag（包）和ankle boot（短靴）
- 导入需要的包或模块

```
1  import torch
2  import torchvision
3  import torchvision.transforms as transforms
```

## 加载Fashion-MNIST数据集（采用已经划分好的训练集和测试集）

- 指定参数 **transform = transforms.ToTensor()**，将所有数据转换为 **Tensor**，**transforms.ToTensor()** 可以将尺寸为 $(H \times W \times C)$ 且数据位于[0, 255]的PIL图片或者数据类型为 **np.uint8** 的NumPy数组转换为尺寸为$(C \times H \times W)$，数据类型为 **torch.float32** 且数据位于[0.0, 1.0]的 **Tensor**。

```
mnist_train = torchvision.datasets.FashionMNIST(root='~/Datasets/FashionMNIST', train=True,
download=True, transform=transforms.ToTensor())
mnist_test = torchvision.datasets.FashionMNIST(root='~/Datasets/FashionMNIST', train=False,
download=True, transform=transforms.ToTensor())
```

- 通过 **DataLoader** 读取小批量数据样本

```
batch_size = 256
train_iter = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True,
num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=False,
num_workers=num_workers)
```

# 4.2 实验内容

## 一、Pytorch基本操作考察

1. 使用 **Tensor** 初始化一个 $1\times3$ 的矩阵 $M$ 和一个 $2\times1$ 的矩阵 $N$，对两矩阵进行减法操作（要求实现三种不同的形式），给出结果并分析三种方式的不同（如果出现报错，分析报错的原因），同时需要指出在计算过程中发生了什么
2. ① 利用 **Tensor** 创建两个大小分别 $3\times2$ 和 $4\times2$ 的随机数矩阵 $P$ 和 $Q$ ，要求服从均值为0，标准差0.01为的正态分布；② 对第二步得到的矩阵 $Q$ 进行形状变换得到 $Q$ 的转置 $Q^T$ ；③ 对上述得到的矩阵 $P$ 和矩阵 $Q^T$ 求矩阵相乘
3. 给定公式 $y_3 = y_1 + y_2 = x^2 + x^3$ ，且 $x = 1$。利用学习所得到的Tensor的相关知识，求$y_3$对$x$的梯度，即$\frac{dy_3}{dx}$。

   要求在计算过程中，在计算 $x^3$ 时中断梯度的追踪，观察结果并进行原因分析

   提示, 可使用 with torch.no_grad()， 举例:

   ```
   with torch.no_grad():
       y = x * 5
   ```

## 二、动手实现 logistic 回归

1. 要求动手从0实现 **logistic 回归**（只借助Tensor和Numpy相关的库）在人工构造的数据集上进行训练和测试，并从loss以及训练集上的准确率等多个角度对结果进行分析
   （可借助nn.BCELoss或nn.BCEWithLogitsLoss作为损失函数，从零实现二元交叉熵为选作）
2. 利用 torch.nn 实现 **logistic 回归**在人工构造的数据集上进行训练和测试，并对结果进行分析，并从loss以及训练集上的准确率等多个角度对结果进行分析

## 三、动手实现 softmax 回归

1. 要求动手从0实现 **softmax 回归**（只借助Tensor和Numpy相关的库）在Fashion-MNIST数据集上进行训练和测试，并从loss、训练集以及测试集上的准确率等多个角度对结果进行分析
   （要求从零实现交叉熵损失函数）
2. 利用torch.nn实现 **softmax 回归**在Fashion-MNIST数据集上进行训练和测试，并从loss，训练集以及测试集上的准确率等多个角度对结果进行分析

# 4.2 实验内容

## 提交作业要求

两种提交实验报告的方式（选择一种提交）：

- word版报告：
  - 根据实验报告中的相应提示内容，完成相应实验报告的部分，展示实验结果时需要图文并茂，并进行相应的分析。
  - 在word文件中需要粘贴关键代码，尽量将代码进行格式化，不要直接截图。
    - 代码格式化网站 http://www.planetb.ca/syntax-highlight-word 或 http://codeinword.com/
  - 同时要求提交实验的完整代码，不同的题目放在不同的.ipynb文件中，需要有相应的注释。
- jupyter版报告：
  - 根据实验报告中的相应提示内容，完成相应实验报告的部分，需要保留相应的代码运行结果、图等内容。
  - 在给定的jupyter模板中提供了编写代码的部分，代码直接编写在相应的块中即可，需要有相应的注释。
  - 实验代码全部写在jupyter中，无需另外提交其他.ipynb文件