



中南大學  
CENTRAL SOUTH UNIVERSITY



# 深度学习

授课人：李仪

liyi1002@csu.edu.cn

<http://faculty.csu.edu.cn/liyi>

中南大学 自动化学院



中南大學  
CENTRAL SOUTH UNIVERSITY



# 第1章 绪论

授课人：李仪

liyi1002@csu.edu.cn

<http://faculty.csu.edu.cn/liyi>

中南大学 自动化学院



# 主要内容

01

人工智能与神经网络

02

深度学习

03

建立PyTorch的开发环境

04

张量基础

05

初识PyTorch框架

06

本章小结



# 01





## 人工智能学科的诞生：

1956年夏季，在美国的达特茅斯(Dartmouth)大学举办了一次长达2个月的研讨会，与会者认真热烈地讨论用机器模拟人类智能的问题。会上，首次使用了**人工智能**（Artificial Intelligence, AI）这一术语，标志着人工智能学科的诞生



## 1.1 人工智能与神经网络

### 人工智能的三次热潮：

**第一次热潮：**从人工智能诞生开始，一直到20世纪70年代初，特点：以命题逻辑、谓词逻辑等知识表达、启发式搜索算法为代表，同时这个时期还出现了感知机

**第二次热潮：**从20世纪80年代初开始，一直到1987年前后，特点：主要研究专家系统、知识工程、医疗诊断等，同时还出现了Hopfield神经网络、BP算法等，这为后面神经网络的发展奠定了基础

**第三次热潮：**从2012年开始，一直到现在，特点：2012年AlexNet在图像识别上取得重大突破，直接掀起了新一轮的人工智能热潮。2016年3月，AlphaGo战胜了国际顶尖围棋职业选手李世石，人工智能再次引起人们的空前关注



## 人工智能与神经网络:

最近两次人工智能热潮都是由神经网络掀起的，再加上卷积神经网络在图像识别、语音处理等领域的成功应用，很多人直接把神经网络等同于人工智能。实际上，神经网络只是人工智能研究的一个子问题。

## 人工智能的三大学派:

- 符号主义 (Symbolicism)
- 联结主义 (Connectionism)，其中神经网络就属于联结主义学派，该学派也称为仿生学派 (Bionicsism) 或生理学派 (Physiologism)
- 行为主义 (Actionism)



## 人工神经网络 (Artificial Neural Network, ANN) :

- ANN的兴起是以1943年提出的M-P模型的出现为标志。M-P模型是一种数学模型，奠定了神经网络模型的基础。
- 1986年，Meclelland和Rumelhart等人发展了BP算法，提出一种基于梯度信息的参数修正算法，为神经网络的训练提供了一种非常成功的参数学习方法。目前，正在盛行的深度学习中各种网络模型也均采用BP算法来训练。





# 02





## 1.2.1 什么是深度学习

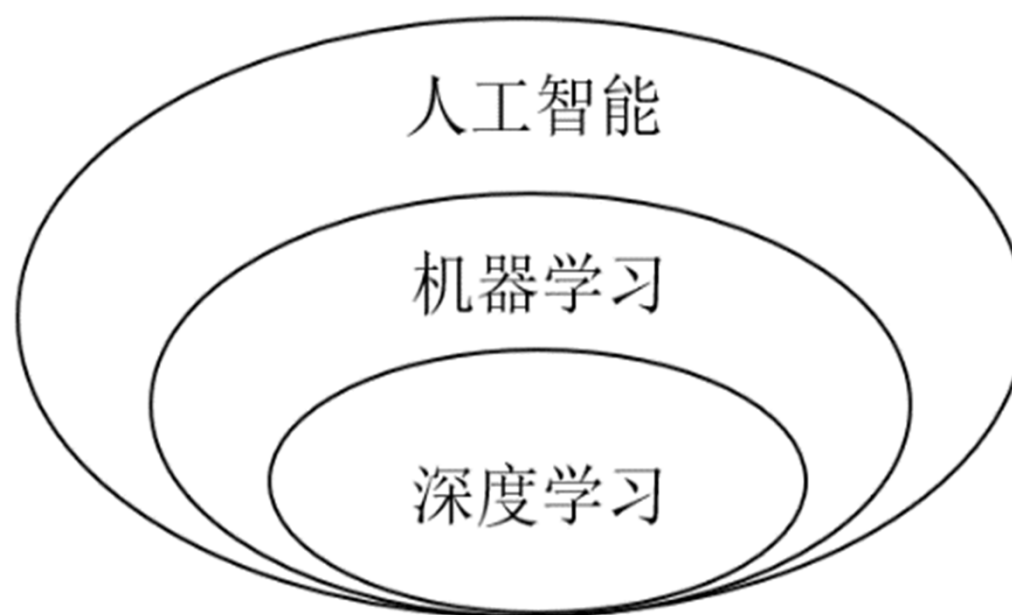
**深度学习：**使用深层神经网络来处理多维数据的一种神经网络学习方法。

**“深度”的内涵：**

- ① **大算力。**加深就意味着参数的大量增加，需要大量的计算资源来支撑海量参数的学习。这在以前，限于技术条件和财力，人们难以获得足够的算力支撑；但现在，随着GPU技术的发展，大算力的支持已经成为现实；
- ② **大数据。**参数的大量增多，不但需要强大的算力支持，而且还需要大量数据支持，否则我们也难以训练这些海量的参数；
- ③ **梯度消失和梯度爆炸。**随着网络层数的增多，从高层逐层反向传递的梯度信息可能会越来越弱，以至于传到底层时梯度几乎为零，从而造成底层参数无法得到更新，导致网络无法收敛；也有可能是在反向传递时，大于1的梯度值不断相乘，导致梯度值越来越大，使得网络处于震荡状态而无法收敛。



## 人工智能、深度学习、机器学习的关系：





## 1.2.2 深度学习的发展过程

在卷积神经网络方面（面向图像处理）：

**LeNet:** 最早的深度学习模型，LannYeCun等人于1998年提出的卷积神经网络

深度学习的概念：由Hinton以及他的学生Salakhutdinov于2006年提出，论述了梯度消失的解决方法

**AlexNet:** LeNet的加宽版，Hinton的学生Krizhevsky Alex于2012年提出，在当年的ImageNet视觉挑战赛（ImageNet Large Scale Visual Recognition Challenge, ILSVRC）上以巨大的优势获得冠军，标志着深度学习时代的来临。

**GoogLeNet和VGG:** 均于2014年提出，GoogLeNet是当年的ILSVRC冠军，通过设计和开发Inception模块，使得模型的参数大幅度减少；VGG则继续加深网络，通过扩展网络的深度来获取性能的提升。



## 1.2.2 深度学习的发展过程

**ResNet:** 2015年被提出，当年获得ILSVRC冠军，网络层数很多，是一种真正的深度神经网络，较好解决了因深度增加而出现性能退化的问题。

**AlphaGo:** 2016年3月，谷歌公司的AlphaGo以4:1战胜了国际顶尖围棋职业选手李世石，掀起了人工智能研究的又一轮风暴

**EfficientNet:** 2019年，Google公司开发了一种高效的深度神经网络——EfficientNet，该网络仍然是至今为止最好的图像识别网络之一。



## 1.2.2 深度学习的发展过程

在循环神经网络方面（面向自然语言处理）：

**LSTM:** 长短时记忆网络（Long Short Term Memory Network, LSTM），由德国科学家Schmidhuber于1997年提出的一种循环神经网络，直到现在LSTM一直都是NLP领域中的重要处理模型（Schmidhuber本人也因此被尊称为“LSTM之父”）

**Transformer:** 谷歌团提出的一种计算架构，该架构完全抛弃了传统神经网络（如CNN和RNN等）的做法，采用纯注意力机制，通过多头自注意力的堆叠来构建模型，Transformer首先在自然语言处理（NLP）领域中取得了非常惊人的成绩



## 1.2.2 深度学习的发展过程

**预训练模型时代：**基于Transformer框架构建的大型预训练模型BERT、GPT以及BERTology系列模型等相继出现，使得深度神经网络走进了一个新时代——预训练模型时代。预训练模型解决了自然语言处理标注数据不足的问题，使得运用经济、便宜的大规模文本语料来训练大模型成为可能，泛化能力等相关性能在包括文本理解和文本生成在内的NLP任务上均获得大幅度的提升。

**ViT：**Transformer还被引入到了机器视觉的目标检测任务中，这就是著名的Visual Transformer (ViT)





### 1.2.3 深度学习的基础网络

- (1) **全连接神经网络 (Fully Connected Neural Network, FCNN)**。这是一种最为常用的神经网络，通常用于数值拟合或分类，所以有时候也称为**分类网络**。
- (2) **卷积神经网络 (Convolutional Neural Network, CNN)**。这是最为著名的深度神经网络，在早期几乎成为深度学习的代名词。它主要用于提取图像的特征，其后面往往跟着一个全连接网络，用于对提取的特征进行分类。
- (3) **循环神经网络 (Recurrent Neural Network, RNN)**。这类网络主要用于处理序列数据，尤其是早期的文本处理几乎都是利用这类神经网络来完成的。
- (4) **基于注意力机制的神经网络 (Attention Mechanism-based Neural Network)**。这类网络主要指基于Transformer发展而形成的网络模型，它抛弃了传统的CNN和RNN，采用纯注意力机制的一种网络。





### 1.2.3 深度学习的基础网络

本书中，一般用“深度神经网络”来统称这些网络，或者统称由这些网络堆叠而形成的其他复杂网络，具体含义应根据上下文来判别。



# 03



建立PyTorch的开发环境\*



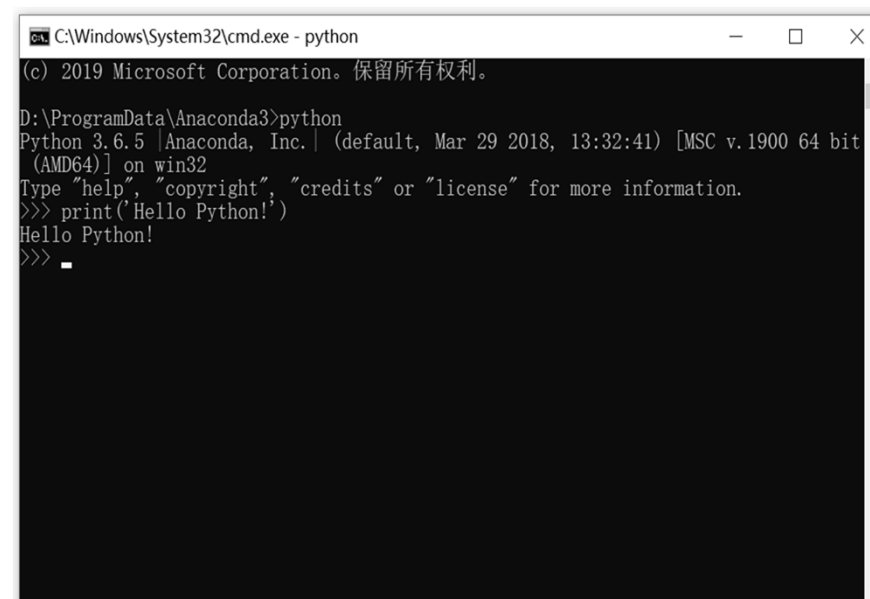
# (本节内容自学)

## 1.3.1 Anaconda与Python的安装

Python只是一个基本的编程语言，它没有包含在代码开发过程中一般还要使用很多其他的包和模块。通常按下列步骤安装Python:

(1) 安装**Anaconda**。Anaconda是一个科学计算环境，它不但包含了Python，而且还包含了一些常用的包和库，如numpy, scrip, matplotlib等。因此，成功安装了Anaconda以后，就相当于安装Python以及相关的库和包等。Anaconda的官网是<https://www.anaconda.com>。（笔者从官网上下载了安装文件Anaconda3-Windows-x86\_64.exe）

(2) 配置**Anaconda3**。笔者安装时Anaconda3的根目录为D:\ProgramData\Anaconda3，安装产生的绝大部分文件都在此目录下，其中包括文件python.exe等。文件python.exe是用于启动Python，方法是：用cmd命令进入命令提示符并进入到D:\ProgramData\Anaconda3目录下，然后输入“python”并回车即可进入Python运行环境：



```
C:\Windows\System32\cmd.exe - python
(c) 2019 Microsoft Corporation. 保留所有权利。

D:\ProgramData\Anaconda3>python
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit
 (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello Python!')
Hello Python!
>>> _
```



## 1.3.2 PyCharm/VSCode和PyTorch的安装——开发环境的安装

### PyCharm的安装:

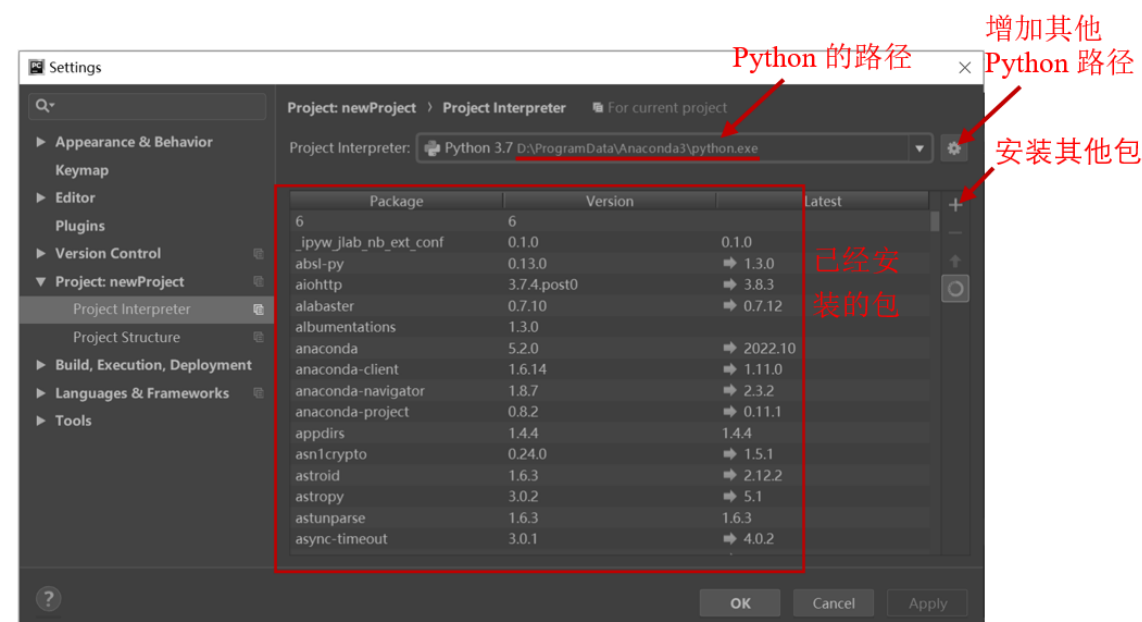
PyCharm是一款专门针对 Python的编辑器，其功能强大、配置简单。该工具的官方下载地址是<https://www.jetbrains.com/pycharm>。笔者下载了PyCharm社区版，安装文件为**pycharm-community-2018.1.exe**。双击该文件，并按提示进行安装即可。



## 1.3.2 PyCharm和PyTorch的安装

### PyCharm的配置:

在PyCharm中设置Python路径的方法是：打开PyCharm，选择菜单“File | Setting...”，打开设置界面，并在左边展开相应的项目名（这里为newProject），然后选择Project Interpreter，如图1-3所示。这时会看到，针对该项目的Python为D:\ProgramData\Anaconda3\python.exe；窗口正中央列出的都是目前可用的包。





## 1.3.2 PyCharm和PyTorch的安装

### 安装Torch（Anaconda并不包含Torch）：

用cmd进入命令提示符界面后，然后用下列pip命令安装：

**pip install torch**

如果要卸载，则用下列命令：

**pip uninstall torch**

**注： PyTorch是Python版的Torch，即PyTorch = Python+Torch**



### 1.3.3 PyTorch的Hello World程序

The screenshot shows the PyCharm IDE interface. The main editor window displays a Python file named `test.py` with the following code:

```
1 import torch                #引入torch库
2 print("Hello World")        #打印字符串"Hello World"
3 print(torch.__version__)     #打印torch的版本号
4 print(torch.cuda.is_available()) #检测电脑上是否有可用的GPU
5
```

The left sidebar shows the project structure with `newProject` and `test.py`. The bottom panel shows the Run output for `test.py`:

```
Run: test x
D:\ProgramData\Anaconda3\python.exe D:/newProject/test.py
Hello World
1.6.0
True
Process finished with exit code 0
```

The status bar at the bottom indicates that power save mode is on and code insight and background tasks are disabled.





# 04







## 1.4.1 张量的定义及其物理含义

**张量** (Tensor)：可以简单理解为一种“数据立方体”，是数据建模和表示的一种手段；也可以理解为**多维数组**的推广：

- 零阶张量是标量（一般数值），
- 一阶张量是一维数组（向量），
- 二阶张量是二维数组（矩阵），
- 三阶张量是三维数组（数据立方体），...

“n阶张量”也称为“n维张量”，表示有n个维的张量。



## 1.4.1 张量的定义及其物理含义

定义张量的方法:

(1) 已知内容, 用`torch.tensor()`函数来定义张量:

```
import torch
x0 = torch.tensor(2)      #0阶张量, 形状为torch.Size([]), 亦写为()
x1 = torch.tensor([2])    #1阶张量, 形状为torch.Size([1]), 亦写为(1)
x2 = torch.tensor([2,3])  #1阶张量, 形状为torch.Size([2]), 亦写为(2)
x3 = torch.tensor([[2,3,4], #2阶张量, 形状为torch.Size([2, 3]), 亦写为(2, 3)
                    [5,6,7]])
x4 = torch.tensor([[[2,3,4], #2阶张量, 形状为torch.Size([3, 3]), 亦写为(3, 3)
                    [5, 6, 7],
                    [8, 9, 10]])

print('x0的阶数为: {}, 形状为: {}'.format(x0.ndim, x0.size()))
print('x1的阶数为: {}, 形状为: {}'.format(x1.ndim, x1.size()))
print('x2的阶数为: {}, 形状为: {}'.format(x2.ndim, x2.size()))
print('x3的阶数为: {}, 形状为: {}'.format(x3.ndim, x3.size()))
print('x4的阶数为: {}, 形状为: {}'.format(x4.ndim, x4.size()))
```

x0的阶数为: 0, 形状为: torch.Size([])
x1的阶数为: 1, 形状为: torch.Size([1])
x2的阶数为: 1, 形状为: torch.Size([2])
x3的阶数为: 2, 形状为: torch.Size([2, 3])
x4的阶数为: 2, 形状为: torch.Size([3, 3])



### 1.4.1 张量的定义及其物理含义

一般地，一个 $n$ 阶（维）张量的形状可表示为下列格式：

$$(D_1, D_2, \dots, D_n)$$

其中， $D_1$ 为该张量第1维的大小， $D_2$ 为第2维的大小， $\dots$ ， $D_n$ 为第 $n$ 维的大小。



### 1.4.1 张量的定义及其物理含义

(2) 已知形状，用`torch.randn()`、`torch.rand()`、`torch.randint()`等函数来定义张量：  
例如，下列语句都是按照形状`torch.Size([32, 3, 224, 224])`分别生成了相应的张量：

```
x5 = torch.randn(32,3,224,224)
```

```
x6 = torch.rand(32,3,224,224)
```

```
x7 = torch.randint(0,6,[32,3,224,224])
```



## 1.4.1 张量的定义及其物理含义

张量中元素的数据类型:

可以用张量的属性dtype输出其中元素的数据类型。例如，下面语句输出张量x1和x5的数据类型:

```
print(x1.dtype, x5.dtype)
```

torch.tensor()默认生成torch.int64类型张量，而torch.Tensor()默认生成torch.float32类型张量。如:

```
x = torch.tensor([2,3])    #torch.int64  
x = torch.Tensor([2,3])    #torch.float32
```

torch.tensor()还可以自动识别数据类型:

```
x = torch.tensor([2,3.]) #torch.float32 (自动识别)
```



## 1.4.1 张量的定义及其物理含义

也可以在定义时显式说明数据类型，例如：

```
x = torch.ByteTensor([2,3])    #torch.uint8
x = torch.CharTensor([2,3])    #torch.int8
x = torch.ShortTensor([2,3])   #torch.int16
x = torch.IntTensor([2,3])     #torch.int32
x = torch.LongTensor([2,3])    #torch.int64
x = torch.FloatTensor([2,3])   #torch.float32
x = torch.DoubleTensor([2,3])  #torch.float64
```

也可以定义后对数据类型进行转换，如：

```
x = torch.tensor([2,3]).byte()  #torch.uint8
x = torch.tensor([2,3]).char()  #torch.int8
x = torch.tensor([2,3]).short() #torch.int16
x = torch.tensor([2,3]).int()   #torch.int32
x = torch.tensor([2,3]).long()  #torch.int64
x = torch.tensor([2,3]).float() #torch.float32
x = torch.tensor([2,3]).double() #torch.float64
```

或者模仿下面语句说明数据类型：

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



## 1.4.1 张量的定义及其物理含义

### 张量的物理含义：

张量的作用是用于对数据进行建模和表示。它可以表示图像，也可以表示编码后的文本，或者表示模型计算的中间结果和输出结果。例如：

- 形状为`torch.Size([300, 400])`的张量可表示一张灰色图像，其中300和400分别表示图像的高和宽（单位为像素）；形状为`torch.Size([3, 300, 400])`的张量可表示一张RGB彩色图像，其中300和400同上，3表示图像的通道数；形状为`torch.Size([32, 3, 300, 400])`的张量可表示一个批量的图像，其中3, 300和400同上，32表示一个批量（batch）中有32张这样的图像，即32表示批量的大小。



## 1.4.1 张量的定义及其物理含义

### 张量的物理含义：

- 文本数据在输入模型之前，需要先对其进行索引编码（整数编码）。编码后可能表示成形状为`torch.Size([128, 20, 512])`的张量，其中128可能表示批量的大小，即一次输入模型的文本条数为128，20表示每条文本序列的固定长度，512则可能是表示文本序列中每个元素（如单词）的向量的长度，即每个元素被表示为长度为512的向量。





## 1.4.2 张量的切片操作

如果把张量看成是一个（超）数据立方体，那么“切片操作”就是从该立方体中切除若干个小块出来，也可能是替换或更新其中的若干个小块。

对张量的切片是按维进行的，基本格式如下：

$$\text{tensor}[\dots, \underbrace{\text{start: end: step}}_{\text{第 } i \text{ 维}}, \dots]$$

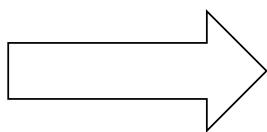
**作用：**取出所有满足第*i*维上索引为start, start+1\*step, start+2\*step, ..., end-1的元素，并按“原来顺序”组成新的张量。



## 1.4.2 张量的切片操作

例如，我们执行下列代码：

```
x=torch.randint(0,10,[4,10])  
print(x)  
print(x[:,3:8:2])
```



```
tensor([[2, 9, 2, 0, 0, 2, 6, 7, 9, 4],  
        [1, 1, 6, 1, 2, 9, 4, 1, 3, 0],  
        [0, 6, 5, 7, 9, 2, 1, 6, 0, 6],  
        [6, 5, 3, 9, 4, 6, 1, 8, 0, 5]])
```

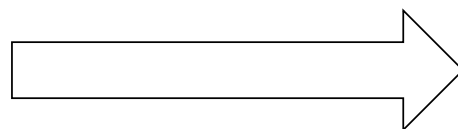
```
tensor([[0, 2, 7],  
        [1, 9, 1],  
        [7, 2, 6],  
        [9, 6, 8]])
```



## 1.4.2 张量的切片操作

如果步长step为1，则step可以省略；如果step省略了，则对应的最后一个冒号“:”也可以省略。  
例如，下面三个语句是等价的：

```
print(x[:,3:8:1])  
print(x[:,3:8:])  
print(x[:,3:8])
```



**x** =

```
tensor([[2, 9, 2, 0, 0, 2, 6, 7, 9, 4],  
        [1, 1, 6, 1, 2, 9, 4, 1, 3, 0],  
        [0, 6, 5, 7, 9, 2, 1, 6, 0, 6],  
        [6, 5, 3, 9, 4, 6, 1, 8, 0, 5]])
```

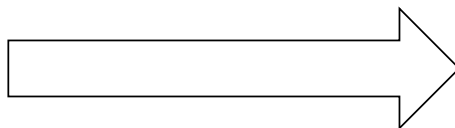
```
tensor([[0, 0, 2, 6, 7],  
        [1, 2, 9, 4, 1],  
        [7, 9, 2, 1, 6],  
        [9, 4, 6, 1, 8]])
```



## 1.4.2 张量的切片操作

start和end可以设置为负数，表示倒数（从右往左数）的意思，但step不能为负数。例如，执行下面语句：

```
print(x[:, -4: -1:])
```



**x =**

```
tensor([[2, 9, 2, 0, 0, 2, 6, 7, 9, 4],  
        [1, 1, 6, 1, 2, 9, 4, 1, 3, 0],  
        [0, 6, 5, 7, 9, 2, 1, 6, 0, 6],  
        [6, 5, 3, 9, 4, 6, 1, 8, 0, 5]])
```

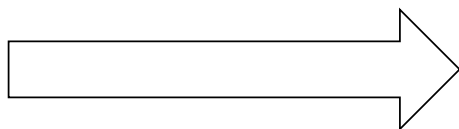
```
tensor([[6, 7, 9],  
        [4, 1, 3],  
        [1, 6, 0],  
        [1, 8, 0]])
```



## 1.4.2 张量的切片操作

如果同时对两个维进行切片操作，则取它们的“交集”。例如，执行下面语句：

```
print(x[1:20:2, 3:8:])
```



```
tensor([[1, 2, 9, 4, 1],  
        [9, 4, 6, 1, 8]])
```

注：如果end超过了维的最大长度，则以最大长度为准。

$x =$

```
tensor([[2, 9, 2, 0, 0, 2, 6, 7, 9, 4],  
        [1, 1, 6, 1, 2, 9, 4, 1, 3, 0],  
        [0, 6, 5, 7, 9, 2, 1, 6, 0, 6],  
        [6, 5, 3, 9, 4, 6, 1, 8, 0, 5]])
```

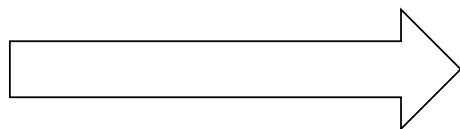


## 1.4.2 张量的切片操作

还可以利用列表来对张量进行切片。

例如，在第2维上使用由索引构成的列表[3,1,0,0]，而且列表中可以有重复的索引，然后将该列表放在相应的维上，得到`x[:, [3,1,0,0]]`，接着输出其内容：

`print(x[:, [3,1,0,0]])`



$x =$

```
tensor([[2, 9, 2, 0, 0, 2, 6, 7, 9, 4],  
        [1, 1, 6, 1, 2, 9, 4, 1, 3, 0],  
        [0, 6, 5, 7, 9, 2, 1, 6, 0, 6],  
        [6, 5, 3, 9, 4, 6, 1, 8, 0, 5]])
```

```
tensor([[0, 9, 2, 2],  
        [1, 1, 1, 1],  
        [7, 6, 0, 0],  
        [9, 5, 6, 6]])
```

注：如果只对第1维进行切片操作，其他维“原封不动”，则其他维可以省略。  
例如，`x[1::2, :]`和`x[1::2]`是一样的。



## 1.4.2 张量的切片操作

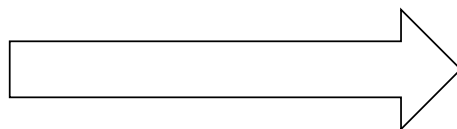
按切片表达式对原张量进行部分赋值。例如，执行下面语句：

```
x=torch.randint(0,10,[4,5])
```

```
print(x)
```

```
x[:,2::2] = torch.zeros(4,2)
```

```
print(x)
```



```
tensor([[2, 9, 2, 0, 0],  
        [2, 6, 7, 9, 4],  
        [1, 1, 6, 1, 2],  
        [9, 4, 1, 3, 0]])
```

```
tensor([[2, 9, 0, 0, 0],  
        [2, 6, 0, 9, 0],  
        [1, 1, 0, 1, 0],  
        [9, 4, 0, 3, 0]])
```

结果张量x的第3列和第5列的值均被改为0。

**注：** `torch.zeros(4,2)`用于产生形状为 $4 \times 2$ 、元素全为0的张量（简称**全0张量**）；`torch.ones(4,2)`用于产生形状为 $4 \times 2$ 、元素全为1的张量（简称**全1张量**）



## 1.4.2 张量的切片操作

已知一个张量 $x$ ，现在要生成跟 $x$ 的形状一样的全0张量和全1张量，则可分别用下面两个语句来实现：

```
y1 = torch.zeros_like(x)  
y2 = torch.ones_like(x)
```

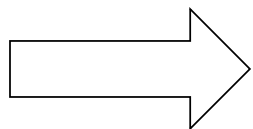




## 1.4.2 张量的切片操作

张量还支持条件类型的切片操作。  
例如，执行下列代码：

```
x=torch.randint(-6,8,[3,4])  
print(x)  
print(x[x<=0])
```



```
tensor([[ -1, -4,  5,  0],  
        [ 1,  2,  7, -5],  
        [-4,  2,  1,  7]])  
  
tensor([-1, -4,  0, -5, -4])
```

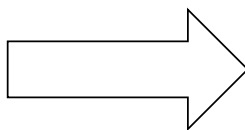
说明： `x[x<=0]` 返回的是将 `x` 中所有满足条件 (`<=0`) 的元素重新组成一个一维张量。



## 1.4.2 张量的切片操作

注意：如果对 $x[x \leq 0]$ 赋值，则是对 $x$ 中所有满足条件（ $\leq 0$ ）的元素修改为相应的数值。例如，执行下列代码：

```
x=torch.tensor([[ -1, -4,  5,  0],  
                [  1,  2,  7, -5],  
                [-4,  2,  1,  7]])  
  
print(x)  
x[x<=0] = 0 #赋值  
print(x)
```



```
tensor([[ -1, -4,  5,  0],  
        [  1,  2,  7, -5],  
        [-4,  2,  1,  7]])  
  
tensor([[0, 0, 5, 0],  
        [1, 2, 7, 0],  
        [0, 2, 1, 7]])
```

说明：张量 $x$ 中那些值小于或等于0的元素都被修改为0了。



## 1.4.3 面向张量的数学函数

### 1. sum()函数

执行下列代码：

```
x=torch.randint(0,6,[2,3])
```

```
print(x)
```

```
print(x.sum())
```

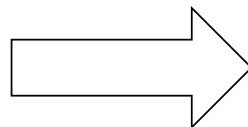
```
print(x.sum(dim=0))
```

```
print(x.sum(dim=1))
```

#求x中所有元素之和

#沿着第1维进行相加

#沿着第2维进行相加



```
tensor([[4, 1, 0],  
        [0, 4, 2]])
```

```
tensor(11)
```

```
tensor([4, 5, 2])
```

```
tensor([5, 6])
```

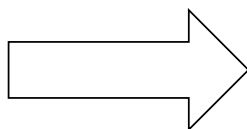


## 1.4.3 面向张量的数学函数

### 2. min()和max()函数

先观察min()函数的效果。执行下列代码：

```
x=torch.randint(-6,6, [2,3])  
print(x)  
print(x.min())  
print(x.min(dim=0) )  
print(x.min(dim=1))
```



```
tensor([[ 4, -5,  0],  
        [ 0, -2, -4]])  
  
tensor(-5)  
  
torch.return_types.min(  
  values=tensor([ 0, -5, -4]), #在第 1 维上的最小值  
  indices=tensor([1, 0, 1])) #在第 1 维上最小值的索引  
  
torch.return_types.min(  
  values=tensor([-5, -4]), #在第 2 维上的最小值  
  indices=tensor([1, 2])) #在第 2 维上最小值的索引
```

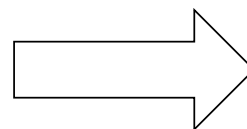


## 1.4.3 面向张量的数学函数

### 3. mean()和sqrt()函数

mean()函数是对张量中所有元素求平均值，也可以沿着指定的维来计算平均值。例如，执行下列代码：

```
x=torch.randint(-6,6,[2,3])  
print(x)  
print(x.float().mean())  
print(x.float().mean(dim=0))    #沿着第1维计算平均值  
print(x.float().mean(dim=1))    #沿着第2维计算平均值
```



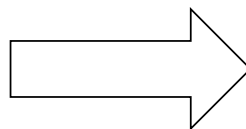
```
tensor([[ 4, -5,  0],  
        [ 0, -2, -4]])  
  
tensor(-1.1667)  
  
tensor([ 2.0000, -3.5000, -2.0000])  
tensor([-0.3333, -2.0000])
```



### 1.4.3 面向张量的数学函数

sqrt()函数是对张量中的元素分别进行开方运算。  
例如，执行下列代码：

```
x=torch.randint(0,6,[2,3])  
print(x)  
print(x.float().sqrt())
```



```
tensor([[4, 1, 0],  
        [0, 4, 2]])
```

```
tensor([[2.0000, 1.0000, 0.0000],  
        [0.0000, 2.0000, 1.4142]])
```



## 1.4.3 面向张量的数学函数

### 4. argmax()和argmin()函数

argmax()函数用于返回最大值的索引。例如，执行下列语句：

```
x=torch.randint(-6,6,[2,3])  
print(x)  
print(x.argmax(dim=0))    #输出第1维上最大值的索引  
print(x.argmax(dim=1))    #输出第2维上最大值的索引
```



```
tensor([[ 3, -1,  0],  
        [ 1,  1,  5]])
```

```
tensor([0, 1, 1])    第 1 维上最大值的索引构成的张量  
tensor([0, 2])      第 2 维上最大值的索引构成的张量
```

说明：argmin()函数和argmax()函数的使用方法一样。argmax()函数多用于处理分类结果，它可以获取最可能类别的索引，是网络模型实现数据分类最常用的函数之一。



## 1.4.3 面向张量的数学函数

### 5. to()方法

该方法用将张量转移到指定的设备上。例如，下列语句分别将张量x转移GPU上和CPU上：

```
x.to('cuda')
```

```
x.to('cpu')
```

### 6. item()函数

对于只有一个元素的张量，该函数可用于提取该元素值，把它转化为一般数值。假设有下列张量x：

```
x = torch.tensor([[[[2]]]])
```

那么，x.item()为普通的整数2。该函数常常用于将元素从张量中“脱离”出来。





## 1.4.4 张量的变形

### 1. reshape()方法

该方法用于对一个张量的形状进行改变，从而得到另一个张量。例如，下列第二条语句执行后，从原来张量x得到形状为(10,4,5)的新张量y：

```
x=torch.randint(0,6,[10,20])
```

```
y = x.reshape(10,4,5)          #等价于y = x.view(10,4,5)
```

注：上述第二条语句执行后，x还是保持原来的形状(10, 20)不变。

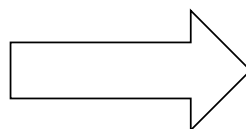


## 1.4.4 张量的变形

### 2. unsqueeze()和squeeze()方法——升维和降维

unsqueeze()方法用于为张量增加一个长度为1的维（即升维），而squeeze()方法则用于去掉长度为1的维度（即降维）。观察下列代码：

```
x=torch.randint(0,6,[10,20])
y1 = x.unsqueeze(0)    #增加第1维，维的长度为1
y2 = x.unsqueeze(1)    #增加第2维，维的长度为1
print(x.shape)
print(y1.shape)
print(y2.shape)
print('-----')
x=torch.randint(0,6,[1,1,1,10,20])
y3 = x.squeeze(2) #去掉第3维
y4 = x.squeeze(3) #无效，因为第4维的长度不是1
y5 = x.squeeze()  #去掉x中所有长度为1的维
print(x.shape)
print(y3.shape)
print(y4.shape)
print(y5.shape)
```



```
torch.Size([10, 20])
torch.Size([1, 10, 20])
torch.Size([10, 1, 20])
-----
torch.Size([1, 1, 1, 10, 20])
torch.Size([1, 1, 10, 20])
torch.Size([1, 1, 1, 10, 20])
torch.Size([10, 20])
```

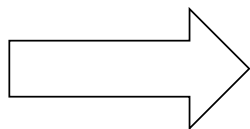


## 1.4.4 张量的变形

### 3. transpose()、t()和permute()函数

这三个函数主要用于调换维的位置，或者称张量的转置。但t()只适用于阶为2的张量。例如，执行下列语句：

```
x=torch.randint(0,6,[2,4])  
y = x.t()    #交换第1维和第2维 （只适用于2阶张量）  
print(x.shape,y.shape)  
x=torch.randint(0,6,[2,4,6,8])  
y = x.transpose(0,2) #交换第1维和第3维  
print(x.shape,y.shape)
```



```
torch.Size([2, 4]) torch.Size([4, 2])  
torch.Size([2, 4, 6, 8]) torch.Size([6, 4, 2, 8])
```



## 1.4.5 张量的常用数学运算

### 1. 基本数学运算

基本数学运算是指通常意义下张量相加、相减、相乘和相除。如果两个张量的形状完全一样，那么它们可以进行通常意义下按元素相加、相减、相乘和相除，得到的结果还是跟原来张量的形状一样。

```
x=torch.randint(0,6,[2,3])
```

```
y=torch.randint(1,8,[2,3])
```

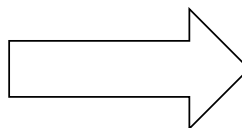
```
print(x)
```

```
print(y)
```

```
print('x/y结果如下: ')
```

```
print(x.float()/y) #浮点数才能进行除运算
```

注：对于 $x*y$ ，假设 $x$ 为张量， $y$ 为标量（一般的数值或0阶张量），那么 $x*y$ 是表示将 $x$ 中的每个元素乘以 $y$ 后得到的新张量。对于 $x+y$ 、 $x-y$ 和 $x/y$ ，亦有类似的结论。



```
tensor([[3, 5, 0],  
        [1, 1, 5]])  
tensor([[1, 2, 7],  
        [4, 2, 5]])  
x/y 结果如下：  
tensor([[3.0000, 2.5000, 0.0000],  
        [0.2500, 0.5000, 1.0000]])
```

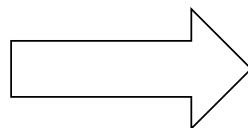


## 1.4.5 张量的常用数学运算

### 2. 点积运算dot()

dot()函数可用于实现两个同等长度的1维张量的点积运算（元素相乘，再求和）。例如，执行下列代码：

```
x=torch.randint(0,6,[4])  
y=torch.randint(-5,6,[4])  
z = torch.dot(x,y)  
print(x)  
print(y)  
print(z)
```



```
tensor([5, 3, 2, 0])  
tensor([ 4, -1,  0,  2])  
tensor(17)
```

注：参与运算的x和y必须是等长的1维张量，实际上它们就是向量了。

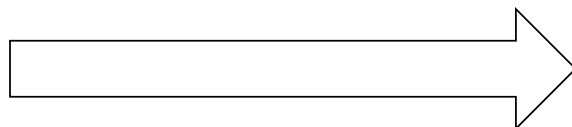


## 1.4.5 张量的常用数学运算

### 3. 矩阵相乘mm()

矩阵是指由2维张量表示的矩阵，矩阵相乘是指传统数学意义上的矩阵相乘。假设x和y是这样的张量，矩阵相乘的前提是x的第2维和y的第1维的长度要相等，且x和y都必须是2维（阶）张量。例如，执行下列代码：

```
x=torch.randint(0,5,[2,3]) #矩阵x，其第2维的长度为3（2×3矩阵）  
y=torch.randint(-2,3,[3,4]) #矩阵y，其第1维的长度亦为3（3×4矩阵）  
z = torch.mm(x,y)          #产生2×4矩阵  
print(x)  
print(y)  
print(z)
```



```
tensor([[2, 1, 2],  
        [1, 1, 0]])  
tensor([[ 0, -2,  0,  2],  
        [ 2, -2,  2,  1],  
        [-1,  2,  1, -1]])  
  
tensor([[ 0, -2,  4,  3],  
        [ 2, -4,  2,  3]])
```



## 1.4.5 张量的常用数学运算

### 4. 带批量大小的矩阵相乘bmm()

对于多张单通道图像，一般用下面格式表示：**(batch\_size, height, width)**，其中，batch\_size表示一个批量（batch）中图像的数量（批量大小），height和width分别表示图像的高和宽。利用bmm()函数便可实现多张图像对的相乘运算（带批量大小的矩阵相乘）。例如，执行下列代码：

```
x=torch.randint(0,5,[32, 300, 400]) #批量大小为32
```

```
y=torch.randint(-2,3,[32, 400, 500]) #批量大小为32
```

```
#x和y的第1维的长度（批量大小）为必须相等
```

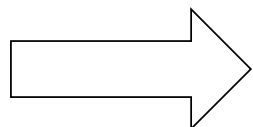
```
#x的第3维和y的第2维的长度要相等
```

```
z = torch.bmm(x,y)
```

```
print(x.shape)
```

```
print(y.shape)
```

```
print(z.shape)
```



```
torch.Size([32, 300, 400])
```

```
torch.Size([32, 400, 500])
```

```
torch.Size([32, 300, 500]) #x 和 y 的运算结果
```



## 1.4.5 张量的常用数学运算

### 5. 含多个维度的矩阵相乘`matmul()`

一个张量可以包含多个维度。一般可以理解为，最后两个维度用于刻画矩阵，而前面的维度则用于对矩阵进行“分组”。当需要对两组已经“分组”的矩阵进行相乘的时候，可以用函数`torch.matmul()`来实现。

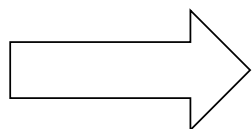
例如，下列代码是执行对两个4维张量进行相乘：

```
x=torch.randint(0,5,[5,7,2,3]) #35个2×3矩阵（先分为5组，再分为7组）
```

```
y=torch.randint(-2,3,[5,7,3,4]) #35个3×4矩阵（先分为5组，再分为7组）
```

```
z = torch.matmul(x,y)
```

```
print(x.shape,'*',y.shape,'-->',z.shape)
```



`torch.Size([5, 7, 2, 3]) * torch.Size([5, 7, 3, 4]) --> torch.Size([5, 7, 2, 4])`





## 1.4.5 张量的常用数学运算

函数`matmul()`的功能比较强，它可以实现函数`mm()`和函数`bmm()`的功能。例如，对于下面的张量`x`和`y`：

```
x=torch.randint(0,5,[2,3]) #矩阵x，其第2维的长度为3（2×3矩阵）  
y=torch.randint(-2,3,[3,4]) #矩阵y，其第1维的长度亦为3（3×4矩阵）
```

下面两条语句是等价的：

```
z = torch.mm(x,y)  
z = torch.matmul(x,y)
```

这表明，函数`matmul()`可以实现函数`mm()`的功能。



## 1.4.5 张量的常用数学运算

类似地，对于下面的张量x和y：

```
x=torch.randint(0,5,[32, 300, 400])  
y=torch.randint(-2,3,[32, 400, 500])
```

下面两条语句是等价的：

```
z = torch.bmm(x,y)  
z = torch.matmul(x,y)
```

这也表明了函数matmul()可以实现函数bmm()的功能。



## 1.4.5 张量的常用数学运算

### 6. 常用的数学函数

这里提及的数学函数包括四舍五入、求指数、取对数、求幂等函数。这些函数都是分别对张量中的元素求函数值，然后形成新的张量。

例如，执行下列代码：

```
x=torch.randint(1,5,[3,4])  
x = x/2.  
y = torch.log2(x)    #求log2(x)  
print(x)  
print(y)
```



```
tensor([[2.0000, 2.0000, 0.5000, 1.5000],  
        [0.5000, 1.5000, 1.5000, 0.5000],  
        [1.0000, 1.5000, 1.5000, 1.5000]])  
  
tensor([[ 1.0000,  1.0000, -1.0000,  0.5850],  
        [-1.0000,  0.5850,  0.5850, -1.0000],  
        [ 0.0000,  0.5850,  0.5850,  0.5850]])
```

注：torch.log2(x)是对x中的元素分别计算以2为底的对数，从而构成新的张量，而且新张量和原来张量x的形状是一样的。



## 1.4.5 张量的常用数学运算

对其他函数的使用方法，举例说明如下：

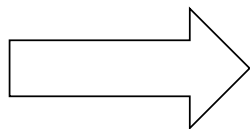
<code>y = torch.round(x)</code>	#四舍五入
<code>y = torch.exp(x)</code>	#求 $e^x$
<code>y = torch.log(x)</code>	#求 $\log_e(x)$
<code>y = torch.log10(x)</code>	#求 $\log_{10}(x)$
<code>y = torch.pow(x,2)</code>	#幂运算, <code>torch.pow(x,2)</code> 等价于 $x^{**2}$



## 1.4.6 张量的广播机制

顾名思义，广播就是一点到多点的发送。这里的“广播”似乎也有类似的原理：它是指一个数据复制为多个数据，进而支持形状不同的两个张量的运算。先观察下面代码：

```
x=torch.randint(1,5, [3,1])  
y=torch.randint(-3,5,[1,4])  
print(x)  
print(y)  
print(x+y)
```



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



## 1.4.6 张量的广播机制

前面代码中，张量x和y的形状分别为(3, 1)和(1, 4)，它们形状是不同的，但是它们却能够相加并得到相应的结果。那么，它们是如何执行这种运算呢？实际上，它们用到了广播机制：

(1) 首先，PyTorch会将x唯一的1列复制（广播）为4列，使得其列数跟y的列数一样，结果x变为：

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

(2) 然后，再将y唯一的1行复制（广播）为3行，以跟x的行数一致。

```
tensor([[ -1,  1, -1,  3],  
        [ -1,  1, -1,  3],  
        [ -1,  1, -1,  3]])
```

注意：这并不是任意两个形状不同的张量都可以用广播机制实现相加功能。实际上，认真分析广播机制可以发现，复制操作是针对长度为1的维进行的；如果复制后仍然无法使得两个张量的形状一样，那么就无法实现这两个张量的相加。

这样，x和y的形状就完全一样了，最后按元素进行相加即可得到上述结果。



## 1.4.7 梯度的自动计算

完成梯度计算是深度学习框架的基本功能，因为深度学习模型需要梯度来更新参数，从而达到参数学习之目的。PyTorch框架也不例外，它也可以非常容易地实现梯度的计算。了解梯度计算的基本原理有利于掌握深度学习的基础理论和方法。先观察下列函数：

$$z = 2 \cdot x^2 - 6 \cdot y^2$$

$$f = z^2$$

根据导数的求导公式知道， $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z} \frac{\partial z}{\partial x} = 2 \cdot z \cdot 4 \cdot x = 8 \cdot z \cdot x$ ，同理 $\frac{\partial f}{\partial y} = -24 \cdot z \cdot y$ 。如果令 $x = 3.0$ ,  $y = 2.0$ ，则 $z = -6$ 。于是， $f$ 关于 $x$ 在 $x = 3.0$ 上的梯度为-144.0， $f$ 关于 $y$ 在 $y = 2.0$ 上的梯度为288.0， $f$ 关于 $z$ 在 $z = -6.0$ 上的梯度为-12.0。

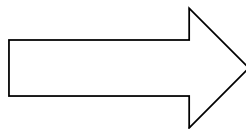


## 1.4.7 梯度的自动计算

在PyTorch框架中，我们可以用下列代码来计算上述导数：

```
x=torch.tensor([3.],requires_grad=True)
y=torch.tensor([2.],requires_grad=True)
z = 2*x**2-6*y**2
f = z**2
f.backward() #自动求导
print('f的值为: ',f.item())
print('f关于x的梯度为: ',x.grad.item())
print('f关于y的梯度为: ',y.grad.item())
```

注：张量有一个属性——`requires_grad`，其默认值为`False`，表示不能计算它的梯度（以提高计算效率和节省内存等）。因此，在定义张量 $x$ 和 $y$ 时，需要显式说明它们的`requires_grad`属性值为`True`。



f 的值为: 36.0  
f 关于 x 的梯度为: -144.0  
f 关于 y 的梯度为: 288.0

当执行`f.backward()`的时候，PyTorch会调用反向传播算法自动计算 $f$ 关于 $x$ 和 $y$ 在 $x = 3.0, y = 2.0$ 上的导数（梯度）





## 1.4.7 梯度的自动计算

### 保留中间结果梯度的方法——hook（钩子）

在自动计算梯度的过程中，中间结果的梯度不会自动被保留下来。但我们可以为某一个中间变量注册一个hook（钩子），从而利用该hook来获取（勾住）中间变量的梯度信息。例如，为获得中间变量 $z$ 的梯度，可以用下列代码实现：

```
def get_z_grad(g): #定义一个hook
    global z_grad #定义全局变量，用于存放梯度
    z_grad = g
    return None
x = torch.tensor([3.], requires_grad=True)
y = torch.tensor([2.], requires_grad=True)
z = 2 * x ** 2 - 6 * y ** 2
f = z ** 2
z.register_hook(get_z_grad) #注册该hook，但必须在f.backward()之前注册hook
f.backward() #自动求导
print('f关于z的梯度为：', z_grad.item())
```

执行这些代码后，得到 $f$ 关于 $z$ 的梯度为-12.0。

执行上述代码后，得到 $f$ 关于 $z$ 的梯度为-12.0。



## 1.4.8 张量与其他对象的相互转换

### 1. 张量与numpy数组之间的转换

(1) 将numpy数组转换为张量：直接将numpy数组作为内容来定义张量，或者利用 `torch.from_numpy()` 函数来实现。例如，下列代码先产生一个numpy数组a，然后用两种方法将之转化为张量：

```
a = [[4,1,0], [0,4,2]]  
a = np.array(a)           #先生成一个numpy数组a  
b = torch.tensor(a)       #转化为张量b  
c = torch.from_numpy(a)   #转化为张量c
```

可以验证，b和c是一样的张量。



## 1.4.8 张量与其他对象的相互转换

(2) 将给定的张量转化为数组：可以利用`np.array()`, `numpy()`等函数来实现。例如，下列代码可以将张量`x`转化为数组`a`和`b`：

```
x = torch.randint(0,6,[2,3])  
a = np.array(x)  
b = x.numpy()
```

可以验证，`a`和`b`是完全相等的。



## 1.4.8 张量与其他对象的相互转换

**张量和numpy数组的区别：**

在PyTorch中张量可以在GPU上运行，从而提高效率；而numpy数组只能在CPU上运行，效率会低得多。此外，numpy数组中元素的类型可以是数值型，也可以是字符串型；而张量中元素的类型只能是数值类型。



## 1.4.8 张量与其他对象的相互转换

### 2. 张量与PIL图像之间的转换

张量主要是在模型当中“流动”，因而在数据预处理、模型调试等过程中可能需要在张量与PIL(Python Image Library, 是Python的图像处理库与Opencv方式并列) 图像之间进行转换。PIL方式一般读取的图像为(H,W,C)需要转换成张量形式 (C,H,W) , 。

下面代码先从磁盘上读取图像文件campus.jpg, 得到PIL格式文件, 然后将之转化为张量, 再接着调用to\_pil\_image()函数将张量又转化为PIL格式文件:

```
path = r'./data/Interpretability/images'
name = 'campus.jpg'
img_path = path + '\\' + name
origin_img = Image.open(img_path).convert('RGB') #打开图片并转换为RGB模型
#PIL---->Tensor (转为张量)
img1 = np.array(origin_img) #先转为numpy数组
img1 = torch.ByteTensor(img1) #再转为张量
#Tensor---->PIL (又转回PIL图像)
pil_img2 = to_pil_image(np.array(img1), mode='RGB')
plt.imshow(pil_img2) #显示图像
```

plt.show()



## 1.4.8 张量与其他对象的相互转换

使用transforms模块进行图像格式转换也是经常使用的方法。例如，下列代码利用transforms模块将PIL图像转换为张量，然后又转换为PIL图像：

```
import torchvision.transforms as transforms
#PIL---->Tensor （转为张量）
tfs = transforms.Compose([transforms.ToTensor()]) #在.Compose()中可添加多个操作，对图
片进行改变
img2 = tfs(origin_img)

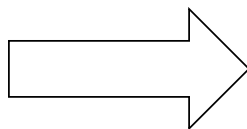
#Tensor---->PIL （又转回PIL图像）
pil_img2 = transforms.ToPILImage()(img2)
```



## 1.4.9 张量的拼接

张量拼接是在特征融合等应用中经常使用到的操作。拼接是按某一维进行的。当按照某一维进行拼接时，除了该维的长度可以不相等以外，其他维的长度必须相等，否则不能拼接。例如，下列代码对x1和x2按照第2维进行拼接：

```
x1 = torch.randint(0,6,[3,4])  
x2 = torch.randint(0,6,[3,2])  
x = torch.cat([x1,x2],dim=1)  
print(x1)  
print(x2)  
print(x)
```



```
tensor([[5, 0, 2, 5],  
       [3, 0, 2, 3],  
       [5, 5, 2, 5]])  
tensor([[3, 5],  
       [1, 4],  
       [0, 2]])  
  
tensor([[5, 0, 2, 5, 3, 5],  
       [3, 0, 2, 3, 1, 4],  
       [5, 5, 2, 5, 0, 2]])
```

可见，张量x1和x2分别有4列和2列，拼接后产生的张量包含了6列，正是这4列和2列“并排放在一起”的结果。



## 1.4.9 张量的拼接

也可以同时对更多个高维的张量进行拼接。例如，下列代码先定义4个4维张量，然后按第2维进行拼接，结果得到形状为(3, 20, 5, 6, 7)的张量x：

```
x1 = torch.randint(0,6,[3,2,5,6,7])  
x2 = torch.randint(0,6,[3,5,5,6,7])  
x3 = torch.randint(0,6,[3,6,5,6,7])  
x4 = torch.randint(0,6,[3,7,5,6,7])  
x = torch.cat([x1,x2,x3,x4],dim=1)
```

结果，x的形状为torch.Size([3, 20, 5, 6, 7])。





# 05





基于PyTorch框架的深度网络模型一般是通过继承Module类来实现的，主要分为三个步骤来完成：

- (1) 定义深度网络模型类，它继承Module类；
- (2) 在模型类中定义网络层；
- (3) 在模型类的forward()方法中，编写网络的业务逻辑，即利用已定义的网络层，构建逻辑上的神经网络。



## 1.5.1 一个简单的网络模型

```
(1) import torch
(2) import torch.nn as nn
(3) class MyModel(nn.Module): #定义深度神经网络模型类
(4)     def __init__(self):
(5)         super().__init__()
(6)         self.features = nn.Sequential(
(7)             nn.Conv2d(3, 20, 5), #第 1 个卷积层
(8)             nn.ReLU(),           #激活函数 relu
(9)             nn.Conv2d(20, 10, 3), #第 2 个卷积层
(10)        )
(11)        #自适应平均池化层（但该层可以放到 forward()方法中）
(12)        self.avgpool = nn.AdaptiveAvgPool2d((32, 32))
(13)        self.fc = nn.Linear(10*32*32, 2) #全连接层
(14)    def forward(self,x):
(15)        out = self.features(x)           #输入两个卷积层
(16)        out = torch.max_pool2d(out, 2, 2) #输入池化层
(17)        out = self.avgpool(out)           #输入自适应平均池化层
(18)        out = out.reshape(x.shape[0], -1) #扁平化
(19)        out = self.fc(out)                #输入全连接层
(20)        return out
(21) mymodel = MyModel()
(22) x = torch.randn(32,3,224,224)         #x 为模型的输入张量
(23) pre_y = mymodel(x)                     #pre_y 为输出的张量
(24) print('输入数据 x 的形状为: ', x.shape)
(25) print('输入数据 pre_y 的形状为: ', pre_y.shape)
```



## 1.5.1 一个简单的网络模型

对于这种网络模型类（先不管类和程序的作用是什么），说明以下几点：

- （1）网络模型类是通过继承nn.Module类来实现。
- （2）nn.Conv2d卷积层接收4维张量的输入，且其形状需满足下列格式：

**(batch\_size, channels, height, width)**

其中，batch\_size表示批量的大小，channels表示张量的通道数，height和width分别为通道的高度和宽度。



## 1.5.1 一个简单的网络模型

由于上述代码中，第一个卷积层的channels设置为3，因此该模型只能接收通道数为3的图像作为输入。例如，对上述模型而言，下面四个张量中前两个张量是正确的输入，后面两个张量是错误的输入：

<code>x = torch.randn(32, 3, 224, 224)</code>	#正确
<code>x = torch.randn(1, 3, 300, 200)</code>	#正确
<code>x = torch.randn(32, 4, 224, 224)</code>	#错误，通道数必需为3
<code>x = torch.randn(1, 32, 3, 224, 224)</code>	#错误，维的数量（阶）必须为4



## 1.5.1 一个简单的网络模型

(3) 在`__init__(self)`函数中，行(6)-(13)上的代码为初始化部分，分别建立了两个卷积层、一个自适应平均池化层和一个全连接层。

(4) 在模型类的`forward()`方法中，将输入`x`送入两个卷积层，然后进入最大池化层和自适应平均池化层，接着进行扁平化，最后送入全连接层，其输出即为整个模型的输出。这些代码的作用实际上是相当于将定义的卷积层和全连接层等“连接”起来，在逻辑上形成一个神经网络，从而实现网络的计算功能。

(5) 行(23)所示的语句实际上在调用`forward()`方法，或者说，在调用模型实例时该方法被默认调用，即`mymodel(x)`等价于`mymodel.forward(x)`。



## 1.5.1 一个简单的网络模型

(6) 自适应平均池化层`self.avgpool`的主要作用之一是通过池化将其输入转变为统一的形状，而不管其输入的形状是什么。这样，由于`self.avgpool`输出的形状是固定的，所以其后面的全连接层也可以固定，从而使得模型可以接收任意形状的通道输入（即通道的高和宽可以取任意值），而不需要修改模型的结构。

(7) 两个卷积层放在一个序列容器`nn.Sequential`中，这样做的目的主要是为了在`forward()`方法中写业务逻辑时变得简单一些。

(8) 一般来说，`__init__(self)`函数部分主要用于定义网络层，尤其是那些有参数的网络层和可能需要放入GPU运行的网络层，这样在实例化模型类的时候可以一次性放入GPU中，而在`forward()`方法中定义网络层，则需要逐一重新放入GPU。



## 1.5.2 访问网络模型的各个网络层

一个网络模型是由一系列网络层或模块组成，我们可以将这些网络层或模块逐一“拆”出来。这可以利用nn.Module的children()方法来实现。例如，对于上面定义的网络模型mymodel，下列代码可以逐一获得各个网络层：

```
for k,layer in  
enumerate(mymodel.children()): #调用  
children()方法获取各个网络层  
    print('第%d层（块）如下：'%(k+1))  
    print(layer)
```

第 1 层（块）如下：

```
Sequential(  
  (0): Conv2d(3, 20, kernel_size=(3, 3), stride=(1, 1))  
  (1): ReLU()  
  (2): Conv2d(20, 10, kernel_size=(3, 3), stride=(1, 1))  
)
```

第 2 层（块）如下：

```
AdaptiveAvgPool2d(output_size=(32, 32))
```

第 3 层（块）如下：

```
Linear(in_features=10240, out_features=2, bias=True)
```





## 1.5.2 访问网络模型的各个网络层

如果想同时获得各层的名称，则可用`named_children()`方法来完成：

```
for k,(name,layer) in enumerate(mymodel.named_children()):  
    print('第%d层（块）的名称为： %s'%(k+1, name))  
    print(layer)
```

```
第 1 层（块）的名称为： features  
Sequential(  
  (0): Conv2d(3, 20, kernel_size=(3, 3), stride=(1, 1))  
  (1): ReLU()  
  (2): Conv2d(20, 10, kernel_size=(3, 3), stride=(1, 1))  
)  
第 2 层（块）的名称为： avgpool  
AdaptiveAvgPool2d(output_size=(32, 32))  
第 3 层（块）的名称为： fc  
Linear(in_features=10240, out_features=2, bias=True)
```



## 1.5.2 访问网络模型的各个网络层

获取所有的网络层——调用modules()方法：

```
for k,layer_block in  
enumerate(mymodel.modules()): #调用  
modules()方法  
    print( '----- %d -----'%(k+1) )  
    print(layer_block)
```

可以看到，modules()方法确实可以获得所有的网络层。但是，它不但输出容器内的所有网络层，而且整个容器也一并输出。因此，输出结果有部分重复了。当然，如果需要的话，可以增加一些条件来选择即可。

```
----- 1 -----  
MyModel(  
  (features): Sequential(  
    (0): Conv2d(3, 20, kernel_size=(3, 3), stride=(1, 1))  
    (1): ReLU()  
    (2): Conv2d(20, 10, kernel_size=(3, 3), stride=(1, 1))  
  )  
  (avgpool): AdaptiveAvgPool2d(output_size=(32, 32))  
  (fc): Linear(in_features=10240, out_features=2, bias=True)  
)  
----- 2 -----  
Sequential(  
  (0): Conv2d(3, 20, kernel_size=(3, 3), stride=(1, 1))  
  (1): ReLU()  
  (2): Conv2d(20, 10, kernel_size=(3, 3), stride=(1, 1))  
)  
----- 3 -----  
Conv2d(3, 20, kernel_size=(3, 3), stride=(1, 1))  
----- 4 -----  
ReLU()  
----- 5 -----  
Conv2d(20, 10, kernel_size=(3, 3), stride=(1, 1))  
----- 6 -----  
AdaptiveAvgPool2d(output_size=(32, 32))  
----- 7 -----  
Linear(in_features=10240, out_features=2, bias=True)
```



## 1.5.2 访问网络模型的各个网络层

如果只需要某一个特定的网络层（而不全部），可以先用`print(mymodel)`查看网络的结构，然后利用网络层的名称或索引来访问特定的网络层。例如，第2个卷积层在容器`Sequential`中的索引为2，容器的名称为`features`，因而可以用下列代码访问该卷积层：

```
layer = mymodel.features[2]
```

如果需要，也可以将数据输入该网络层进行处理。例如：

```
x = torch.randn(32,20,100,100) #构造模拟数据，但要符合网络层输入形状  
out = layer(x)                #将x输入该网络层  
print(out.shape)              #输出的形状为torch.Size([32, 10, 98, 98])
```



## 1.5.3 访问模型参数及模型保存和加载方法

### (1) 访问模型包含的所有参数

可以调用网络模型的parameters()方法来获取模型包含的所有参数。例如，利用下列代码可输出模型mymodel中的所有参数以及统计模型的参数总量：

```
param_num = 0
for param in mymodel.parameters():
    param_num += torch.numel(param) #统计模型参数
    print(param.shape)              #输出模型各层的参数形状
print('该网络参数的总量为：', param_num)
```

```
torch.Size([20, 3, 5, 5])
torch.Size([20])
torch.Size([10, 20, 3, 3])
torch.Size([10])
torch.Size([2, 10240])
torch.Size([2])
该网络参数的总量为： 23812
```



## 1.5.3 访问模型参数及模型保存和加载方法

### (2) 参数的冻结方法

参数有一个重要的属性——`requires_grad`，默认情况下该属性的值为`True`，表示参数是可更新的（可学习的）。如果被设置为`False`，则表示参数是不可学习的，亦即参数被冻结了。例如，下列语句可对模型`mymodel`的所有参数进行冻结：

```
for param in mymodel.parameters():  
    param.requires_grad = False #冻结参数
```

如果想同时获得参数本身和参数的名称，可用下列代码来实现：

```
for param in mymodel.named_parameters():  
    print('参数名称为：', param[0], '参数的形状为：', param[1].shape )
```



## 1.5.3 访问模型参数及模型保存和加载方法

### (3) 模型和参数的保存方法：

#### (a) 仅保存模型参数的方式

模型参数是放在模型参数字典当中，因此只保存该字典即可。例如，下列代码仅保存模型mymodel的参数：

```
torch.save(mymodel.state_dict(), 'mymodel.pth') #文件扩展名推荐为pth或ph或其他
```

由于只保存模型的参数，因此在恢复模型时，要先创建一个结构完全一样的模型：

```
my_new_model = MyModel()
```

然后读取模型参数：

```
mymodel_paramters = torch.load('mymodel.pth')
```

最后用读到的参数更新模型my\_new\_model：

```
my_new_model.load_state_dict(mymodel_paramters)
```

这时，模型my\_new\_model跟保存时的模型mymodel完全一样（包括结构和参数）。



## 1.5.3 访问模型参数及模型保存和加载方法

### (b) 保存整个模型

保存整个模型也是利用`torch.save()`函数来完成，但其形式更为简单，只需将模型“直接保存”下来即可。例如，下列语句是将模型`mymodel`作为一个整体保存下来（包括网络结构及其参数）：

```
torch.save(mymodel, 'mymodel.pth')
```

加载时调用下列语句：

```
my_new_model = torch.load('mymodel.pth')
```

这时得到的模型`my_new_model`跟原来模型`mymodel`是完全一样的。但注意，加载时原来用于实例化模型`mymodel`的类`MyModel`要在`.py`文件中存在。

显然，由于在保存整个模型时，除了保存参数以外还要保存模型的结构，因此所占用的磁盘空间就大一些，保存时间也多一些。



# 06

本章小结







## 本章主要内容：

- 人工智能和神经网络的发展过程
- 深度学习的概念
- PyTorch开发环境的建立
- 张量的概念及其使用方法
- PyTorch程序的开发步骤
- 网络层和网络参数的访问方法