PyTorch深度学习基础

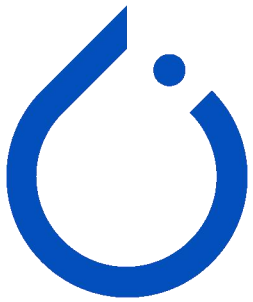
泰迪智能科技（武汉）有限公司

目 录

contents

第一部分

引言



- 深度学习发展历程
- PyTorch介绍
- 其他学习框架

深度学习发展历程

深度学习的发展历程可以追溯到上世纪50年代的感知机模型，但直到近年来才迎来了爆发性的发展。深度学习的发展主要分为以下几个阶段：

1.感知机时代（1950s - 1960s）：由Frank Rosenblatt提出的感知机是神经网络的早期形式，但受限于其单层结构和线性可分问题的限制。

2.冷静时期（1960s - 1990s）：深度学习的研究进入低谷，主要是因为多层神经网络的训练问题和计算能力的限制。

3.神经网络复兴（2000s - 2010s）：随着更强大的计算机和更好的算法出现，深度学习开始复苏。突破性的工作包括卷积神经网络（CNN）和递归神经网络（RNN）的应用。

4.深度学习的崛起（2010s以后）：深度学习在图像识别、自然语言处理、语音识别等领域取得了重大突破，其中深度卷积神经网络和递归神经网络的广泛应用引领了这一发展浪潮。

5.硬件和框架支持（2010s以后）：图形处理单元（GPU）和专用深度学习芯片的发展，以及开源框架如PyTorch和TensorFlow和的兴起，进一步推动了深度学习的普及和应用。

深度学习的快速发展为计算机视觉、自然语言处理、自动驾驶、医学诊断等领域带来了巨大的进步，成为人工智能的核心技术之一，吸引了广泛的研究和产业投入。

PyTorch介绍

PyTorch是一个Python程序库，有助于构建深度学习项目，由Facebook的人工智能研究小组开发和维护。它强调灵活性，并允许用深度学习领域惯用的Python来表示深度学习模型。它的易用性使得它在研究社区中有了早期的使用者，并且在第1次发布之后的几年里，它已经成为应用程序中使用最广泛的深度学习工具之一。

PyTorch是一个用于深度学习和人工智能研究的开源机器学习框架。并且从Pytorch2.0开始，PyTorch引入compile功能，compile底层是一系列新技术，包含了TorchDynamo、AOTAutograd、PrimTorch和TorchInductor。这几个新功能都是由Python编写，并支持Dynamic Shapes。Dynamic Shapes供用户发送不同大小的张量，但是却又不需要重新编译。

截止到2023年10月18日，PyTorch已经是更新到了2.1版本。

此外，PyTorch官方也发布了对PyTorch的详细使用教程和API文档，方便大家学习和使用。

PyTorch官方文档：https://pytorch.org/docs/stable/index.html#

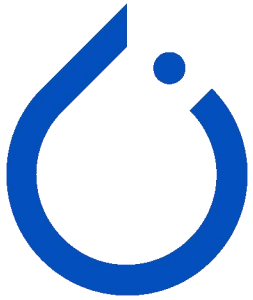
PyTorch中文文档：https://pytorch.apachecn.org/

其他深度学习框架

深度学习发展到现在，除了PyTorch之外，还有一些其他深度学习框架，这些深度学习框架虽然使用的用户量远远不如PyTorch，但是在某些领域下也是有自己的应用场景，例如：Tensorflow（由于其对Keras的完全封装，使得在使用上变得更简单，对于初学者而言更为友好。）等等。如下表所示：

维护机构	框架名称
谷歌	Tensorflow
百度	PaddlePaddle
伯克利人工智能研究小组和伯克利视觉和学习中心	Caffe
亚马逊	MXNet
微软	CNTK

第二部分　PyTorch环境安装



- 计算资源介绍
- PyTorch虚拟环境创建
- PyTorch环境安装

计算资源介绍

深度学习模型包含大量的参数需要训练，通常也需要大量的计算能力，常见的计算资源包括GPU，TPU， CPU等等，但是其计算能力有较大差别。

1. GPU（图形处理单元）：GPU是深度学习中最常用的计算资源之一。它们具有大量的并行处理单元，适用于高性能的张量运算，可显著加速训练深度学习模型。NVIDIA是一家主要的GPU制造商，其CUDA平台是深度学习框架（如PyTorch 和TensorFlow）的主要支持者。
2. TPU（张量处理单元）：TPU是Google开发的专用深度学习处理器，用于在Google Cloud上进行深度学习工作负载的加速。它们在某些任务上（如：图像生成，大规模分布式训练等等）可以比GPU更高效。
3. CPU（中央处理单元）：CPU相对于GPU、TPU和其他专用硬件来说在深度学习中的性能较低，因此在处理大规模深度学习任务时通常不是首选的选择。主要用于小规模测试任务。

PyTorch虚拟环境创建

PyTorch深度学习框架，也是依赖在一些基础库，如Numpy，networkx等等，上进行进一步开发的。这些基础库有可能又被其他库所依赖，如果将这些环境全部安装到基础环境可能会导致基础库版本冲突。那么这个时候就需要使用到虚拟环境，创建一个单独的虚拟环境来使用PyTorch深度学习框架。

使用虚拟环境主要有以下几个优点：

1. **隔离环境**：虚拟环境允许我们在同一计算机上创建多个独立的Python环境，每个环境可以拥有不同版本的Python和不同的软件包。这有助于隔离项目之间的依赖关系，防止冲突和混乱。
2. **版本控制**：虚拟环境使我们能够明确定义和控制项目所使用的Python和库的版本。这对于确保项目在不同的开发环境中保持一致性非常重要。
3. **清理和维护**：通过使用虚拟环境，我们可以更轻松地清理或卸载特定项目的依赖项，而不会影响到其他项目。这对于管理和维护多个项目非常有帮助。
4. **灵活性**：我们可以为每个项目创建自定义的虚拟环境，以满足特定项目的需求。这使得在项目之间切换和测试不同配置变得更加容易。
5. **便于分享**：如果我们需要与他人分享项目或协作开发，虚拟环境可确保其他人可以轻松地重现项目的环境，而无需担心依赖问题。

PyTorch虚拟环境创建

在使用虚拟环境时，我们通常会使用工具如virtualenv、conda或pyenv，这些工具使得创建、激活和管理虚拟环境变得非常方便。

在此我们演示使用conda去创建和管理虚拟环境。

首先，确保自己电脑中的conda可以正常使用，在命令提示符（CMD）中输入以下命令查看conda版本。

```
C:\Users\ming>conda --version
conda 4.14.0
```

返回conda的版本信息则说明conda环境正常可用。

接下来输入以下命令创建虚拟环境。

conda create -n torch2 python=3.9

这行命令会帮我们创建一个名称为torch2的虚拟环境，虚拟环境的Python版本为python3.8。

注意：在创建虚拟环境时，如果电脑中开启了代理，请先关闭电脑中的代理，再执行命令。

Pytorch虚拟环境创建

查看虚拟环境是否创建成功

conda env list

```
C:\Users\ming>conda env list
# conda environments:
#
base                  D:\ProgramSoftware\Python\Miniconda3
cv2                   D:\ProgramSoftware\Python\Miniconda3\envs\cv2
pyspark               D:\ProgramSoftware\Python\Miniconda3\envs\pyspark
pytorch               D:\ProgramSoftware\Python\Miniconda3\envs\pytorch
qt                    D:\ProgramSoftware\Python\Miniconda3\envs\qt
spider                D:\ProgramSoftware\Python\Miniconda3\envs\spider
tf2                   D:\ProgramSoftware\Python\Miniconda3\envs\tf2
tmp                   D:\ProgramSoftware\Python\Miniconda3\envs\tmp
torch2                D:\ProgramSoftware\Python\Miniconda3\envs\torch2
```

该命令会打印出当前所有的conda虚拟环境，输入以下命令激活即可使用。

conda activate torch2

```
C:\Users\ming>conda activate torch2

(torch2) C:\Users\ming>
```

激活成功后在终端路径前方就会显示当前终端所使用的Python虚拟环境名称。

Pytorch环境安装

PyTorch

接下来在torch2虚拟环境中安装PyTorch2.1版本。

在安装前需要首先确定自己需要安装的PyTorch版本，PyTorch版本一般分为GPU版本和CPU版本。

在安装GPU版本的时候还需要去下载相关的CUDA驱动才可以正常使用，如果电脑中没有NVIDIA显卡那么只需安装CPU版本的PyTorch即可。

如何确定自己电脑中是否有NVIDIA显卡，可以参照以下教程：

http://minglog.hzbmmc.com/2023/03/09/Pytorch%20GPU%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0%E7%8E%AF%E5%A2%83%E5%AE%89%E8%A3%85/

如果电脑中存在NVIDIA显卡，那么接下来就需要去查看CUDA版本，CUDA驱动是向下兼容的，对应的驱动程序依赖的CUDA版本只能比本机中安装的CUDA驱动低，才可以正常运行。

在CMD中输入 nvidia-smi 查看电脑中的CUDA驱动版本

```
(torch2) C:\Users\ming>nvidia-smi
Wed Oct 18 16:52:51 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 537.58                 Driver Version: 537.58       CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                     TCC/WDDM  | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce MX450         WDDM  | 00000000:01:00.0 Off |                  N/A |
| N/A   55C    P8              N/A / ERR! |      0MiB /  2048MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+

(torch2) C:\Users\ming>
```

我这里的CUDA驱动版本是12.2，那么我在安装PyTorch时就只能选择比12.2低的版本。

PyTorch环境安装

进入PyTorch官网：https://pytorch.org/

在官网中可以根据我们电脑的硬件状况和操作系统版本，自动帮我们生成下载PyTorch的命令。由于我的CUDA版本是12.2所以我可以选择CUDA12.1对应的版本下载。

如果CUDA版本过低，可选择升级CUDA驱动，或者以点击此链接，下载低版本PyTorch。

此时生成的下载命令为：

pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121

将生成的命令粘贴到虚拟环境终端即可开始下载PyTorch。

```
(torch2) C:\Users\ming>pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121
Looking in indexes: https://download.pytorch.org/whl/cu121
Collecting torch
  Downloading https://download.pytorch.org/whl/cu121/torch-2.1.0%2Bcu121-cp38-cp38-win_amd64.whl (2474.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.5/2.5 GB 1.8 MB/s eta 0:00:00
Collecting torchvision
  Downloading https://download.pytorch.org/whl/cu121/torchvision-0.16.0%2Bcu121-cp38-cp38-win_amd64.whl (5.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.8/5.8 MB 28.3 MB/s eta 0:00:00
Collecting torchaudio
  Downloading https://download.pytorch.org/whl/cu121/torchaudio-2.1.0%2Bcu121-cp38-cp38-win_amd64.whl (4.0 MB)
  Downloading https://download.pytorch.org/whl/idna-3.4-py3-none-any.whl (61 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 61.5/61.5 kB ? eta 0:00:00
Collecting urllib3<1.27,>=1.21.1 (from requests->torchvision)
  Downloading https://download.pytorch.org/whl/urllib3-1.26.13-py2.py3-none-any.whl (140 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 140.6/140.6 kB ? eta 0:00:00
Collecting certifi>=2017.4.17 (from requests->torchvision)
  Downloading https://download.pytorch.org/whl/certifi-2022.12.7-py3-none-any.whl (155 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 155.3/155.3 kB 9.1 MB/s eta 0:00:00
Collecting mpmath>=0.19 (from sympy->torch)
  Downloading https://download.pytorch.org/whl/mpmath-1.3.0-py3-none-any.whl (536 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 536.2/536.2 kB 35.1 MB/s eta 0:00:00
Installing collected packages: mpmath, urllib3, typing-extensions, sympy, pillow, numpy, networkx, MarkupSafe, idna, fsspec,
 filelock, charset-normalizer, certifi, requests, jinja2, torch, torchvision, torchaudio
Successfully installed MarkupSafe-2.1.2 certifi-2022.12.7 charset-normalizer-2.1.1 filelock-3.9.0 fsspec-2023.4.0 idna-3.4 j
inja2-3.1.2 mpmath-1.3.0 networkx-3.0 numpy-1.24.1 pillow-9.3.0 requests-2.28.1 sympy-1.12 torch-2.1.0+cu121 torchaudio-2.1.
0+cu121 torchvision-0.16.0+cu121 typing-extensions-4.4.0 urllib3-1.26.13
```

最后在终端中生成以上内容则说明下载完成。

检查是否安装成功

在虚拟环境终端中输入Python，进入虚拟环境的Python环境。

输入以下命令：

import torch

torch.cuda.is_available()

```
(torch2) C:\Users\ming>python
Python 3.8.18 (default, Sep 11 2023, 13:39:12) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>>
>>> torch.cuda.is_available()
True
>>>
```

返回的结果为True，说明环境安装成功。并且GPU环境可用。

如果返回结果为False，说明GPU环境不可用，请检查CUDA版本后安装合适的PyTorch版本。

安装的CPU版本的PyTorch，则只需import torch导入不报错即可。

PyTorch环境安装

为了后续可以将虚拟环境放在Juputer Notebook中使用，我们还需要将虚拟环境添加到Jupyter环境内核中。

首先在虚拟环境中下载jupyter

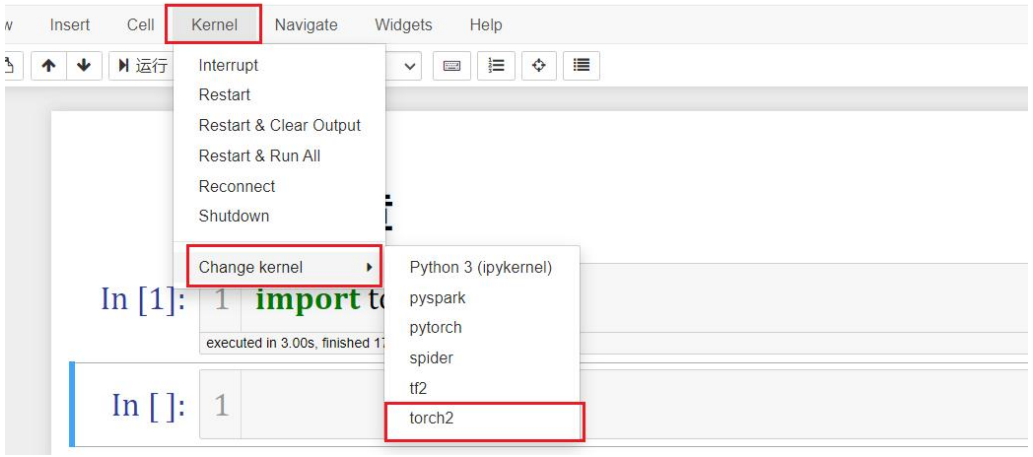
pip install jupyter –i https://pypi.tuna.tsinghua.edu.cn/simple

然后执行以下命令将torch2虚拟环境添加到Jupyter内核中。

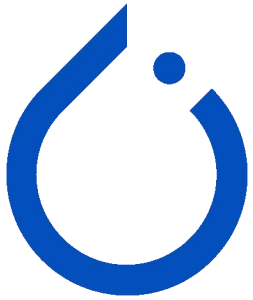
python -m ipykernel install --user --name=torch2

```
(torch2) C:\Users\ming>python -m ipykernel install --user --name=torch2
Installed kernelspec torch2 in C:\Users\ming\AppData\Roaming\jupyter\kernels\torch2
```

此时进入到Jupyter Notebook环境，即可切换到torch2内核。



第三部分　PyTorch张量操作



- 什么是Tensor？如何创建张量？
- Tensor与ndarray的相互转换
- 随机抽样
- Tensor的序列化与反序列化
- 张量的元素操作
- 张量的计算操作
- PyTorch自动求导机制

什么是Tensor？

“Tensor”（张量）是一个数学和计算机科学术语，用于表示多维数组的数据结构。在深度学习和机器学习中，张量是一种常见的数据结构，用于存储和表示多维的数据，如标量（0维张量，即单个数值）、向量（1维张量，如一维数组）、矩阵（2维张量，如二维数组）以及更高维度的数据。张量可以包含整数、浮点数或其他数据类型，具体取决于应用的需要，点击此处查看PyTorch所有数据类型。

在深度学习中，张量是神经网络的核心数据结构。神经网络的层、权重、激活函数和输入都表示为张量。张量的多维性使其适用于处理各种复杂的数据，如图像、文本、音频等，并且能够在GPU和TPU等计算资源上执行并行式计算，大大提高资源利用率。

创建简单张量

创建简单张量

- torch.tensor(list)
- ✓ e.g.: tensor1 = torch.tensor([[1, 2, 3], [2, 3, 6]])

创建未初始化张量（张量的值会取决于内存中的随机初始化值或者取决于之前内存中的内容）

- torch.empty(size)
- ✓ e.g.: tensor2 = torch.empty(size=(2, 3))
- torch.empty_like(tensor1) # 已知与某个张量维度一致
- ✓ e.g.: torch.empty_like(tensor1)
- torch.Tensor(dim1, dim2) # 直接传入生成张量的维度到torch.Tensor()时与torch.empty()效果等价
- ✓ e.g.: tensor3 = torch.Tensor(2, 3)

查看张量的基本属性与方法

张量的尺寸

- torch.Tensor.shape
- torch.Tensor.size()

张量的元素类型

- torch.Tensor.dtype

张量的维度

- torch.Tensor.ndim

张量元素个数

- torch.Tensor.numel()
- torch.numel(torch.Tensor)

对张量进行转置

- torch.Tensor.T
- torch.Tensor.t()

获取张量中非零元素所在位置

- torch.Tensor.nonzero() # 返回一个二维张量，第一列表示行数，第二列表示列数

张量所在设备

- torch.Tensor.device

查看张量梯度

- torch.Tensor.grad # 该属性只有在反向传播计算梯度后才会有具体的值

张量与数组的相互转化

将数组转化为张量

- torch.Tensor(np.ndarray)
- ✓ torch.Tensor(array1)
- torch.from_numpy(np.ndarray)
- ✓ torch.from_numpy(array1)
- torch.asarray(np.ndarray)
- ✓ torch.asarray(array1)

第三个方法asarray()的功能更多，不仅可以将数组转化为张量，还可以将其他数据类型转化为张量，例如：标量，列表，元组等等。
而from_numpy()函数只能将数组转化为张量，如果是其他类型的数据则会报错。

将张量转化为数组

- np.array(torch.Tensor)
- ✓ np.array(tensor1)
- torch.Tensor.numpy()
- ✓ tensor1.numpy()

创建特殊张量

等差张量

- torch.range(start=0, end, step=1) # 该方法即将淘汰与torch.arange()等价
- torch.arange(start=0, end, step=1)
- torch.linspace(start, end, steps)

等比张量

- torch.logspace(start, end, steps, base=10.0)

全0张量

- torch.zeros(size) # 已知张量维度
- torch.zeros_like(input) # 已知与某个张量尺寸一致

全1张量

- torch.ones(size) # 已知张量维度
- torch.ones_like(input) # 已知与某个张量尺寸一致

创建相同元素的张量

- torch.full(size, fill_value) # 已知张量维度
  - ✓ e.g.: torch.full((3, 3), 2.3) # <==> torch.ones(size=(3, 3)) * 2.3
- torch.full_like(input, fill_value) # 已知与某个张量尺寸一致
  - ✓ e.g.: torch.full_like(input=tensor1, fill_value=2.3)

创建对角线元素为1的二维张量（单位矩阵）

- torch.eye(n)

随机抽样

设定随机种子数

设定随机种子数后，在Pytorch中使用“随机系列”方法生成的随机张量值就会固定下来，不会每次运行时更新，这对于复现某些模型的结果非常有用

- torch.manual_seed(1000)
- torch.initial_seed() # 查看生成的随机种子数

创建从0~(n-1)的随机整数序列

该方法得到的结果可以理解为将torch.arange()方法得到的结果进行打乱输出

- torch.randperm(n)

均匀分布

- torch.rand(size) # 已知尺寸，返回[0, 1)上的均匀分布随机数
- torch.rand_like(input) #已知和某个张量维度相同，返回[0, 1)上的均匀分布随机数
- torch.Tensor.uniform_(from=0, to=1) #使用均匀分布随机数对张量中的值进行填充，该方法会直接修改torch.Tensor中的值，该方法可以指定均匀分布范围[from, to)

伯努利分布

- torch.bernoulli(tensor2) #以上方的tensor2的结果作为概率进行伯努利采样，结果为0和1构成的张量，1表示该位置元素被取出

随机抽样

多项式分布

- torch.multinomial(input, num_samples, replacement=True)
- input：多项式分布权重
- num_samples：采样数量
- replacement：是否有放回采样，默认是有放回

✓ e.g.:

```
1.  weight = torch.Tensor([5, 3, 1, 9])  # 定义
2.  torch.multinomial(weight, 40, replacement=True)  # 3这个位置的权重最大，那么3肯定取出的次数最多
```

泊松分布 $P(X=k)=\frac{\lambda^k}{k!}e^{-\lambda}, k=0,1,\cdots$

- torch.poisson(input)
- input：包含泊松分布参数的张量

✓ e.g.:

```
1.  rates = torch.rand(4, 4) * 5  # rate parameter between 0 and 5
2.  torch.poisson(rates)
```

随机抽样

随机整数

当high省略时，生成随机整数的范围为[0, low)

当high不省略时，生成随机整数的范围为[low, high)

- torch.randint(low=0, high, size) # 已知尺寸
- torch.randint_like(input, low=0, high) # 已知和某个张量维度相同

- low：最小值，默认为0
- high：最大值

✓ e.g.:

```
1.  torch.randint(20, 50, size=(5, 5))
2.  torch.randint_like(tensor2, 20, 50)
```

随机抽样——正态分布

标准正态分布

- torch.randn(size) # 已知尺寸
- torch.randn_like(input) # 已知和某个张量维度相同

一般正态分布

torch.normal(mean, std)

1. 多个均值，多个标准差

此时生成的每个元素都有对应一个正态分布

✓ e.g.: torch.normal(mean=torch.arange(1, 11, 1.0), std=torch.arange(1, 0, -0.1))

2. 一个均值，多个标准差

此时生成的元素均值相同，标准差不同

✓ e.g.: torch.normal(mean=torch.Tensor([1.0,]), std=torch.arange(1, 0, -0.1))

3. 多个均值，一个标准差

此时生成的元素均值不同，标准差相同

✓ e.g.: torch.normal(mean=torch.arange(1, 11, 1.0), std=torch.Tensor([1.0,]))

4. 一个均值一个标准差

此时就需要去指定生成的张量尺寸

✓ e.g.: torch.normal(mean=0, std=1, size=(3, 3))

张量的序列化与反序列化

序列化：将张量保存到硬盘

➢ torch.save(obj: object, f: Union[str, os.PathLike, BinaryIO, IO[bytes]])

- obj：要保存的对象
- f：保存的文件句柄/路径

✓ e.g.: torch.save(tensor2, 'tensor2.pt')

反序列化：将硬盘上的张量文件读取到内存

torch.load(f, map_location=None)

- f：要读取的文件句柄/路径
- map_location：张量读取进来保存的设备位置（例如：cpu, cuda, cuda:index等等）

✓ e.g.: tensor_read = torch.load('tensor2.pt', map_location='cuda:0’)

此时需要注意一个一个问题，默认的情况下张量存储到硬盘中会记录张量保存前的设备位置，在导入的时候也会自动导入到相应的设备。这个时候会存在一个问题，如果我在服务器中保存了cuda:2设备中的某个张量到硬盘，然后再将硬盘读取到我本机的电脑中，我本机电脑有没有cuda:2设备，这个时候就会报错。就需要通过指定map_location参数将张量读取到某个设备内存中。

张量的元素操作

张量的索引和切片

张量支持类似于Numpy中的元素索引和切片，例如：取出部分元素，按照条件对元素进行筛选等等。

✓ e.g.:

```
# 获取tensor4中第2行第3列的值
tensor4[1, 2]
# 切片出tensor4中行下标值为1, 2；列下标值2, 3, 4的部分元素值
tensor4[1:, 2:]
# 筛选出tensor4中大于5的元素
tensor4[tensor4 > 5]
```

张量尺寸修改

- torch.Tensor.reshape(shape)
- ✓ e.g.: tensor4.reshape(shape=(4, 3))
- torch.Tensor.view(*shape)
- ✓ e.g.: tensor4.view(4, 3)

张量的元素操作

张量的拼接（不生成新维度）

以下三种写法等价：

- torch.cat(tensors, dim=0)
- torch.concat(tensors, dim=0)
- torch.concatenate(tensors, dim=0)

- tensors：拼接的张量列表
- dim：张量拼接的维度

✓ torch.cat([tensor4, tensor5], dim=1) # 横向拼接

✓ torch.cat([tensor4, tensor5], dim=0) # 纵向拼接

张量的拼接（生成新维度）

- torch.stack(tensors, dim=0)

- tensors：拼接的张量列表
- dim：张量拼接的维度

✓ torch.stack([tensor4, tensor5], dim=0) # 堆叠到第一个维度

✓ torch.stack([tensor4, tensor5], dim=1) # 堆叠到第二个维度

张量的元素操作

张量的切分（指定切分份数，如果不能整切，那么最后一个不足的维度会单独作为一个张量）

- chunk(input, chunks, dim=0)
  - input：待切分张量
  - chunks：切分份数
  - dim：切分维度
- ✓ e.g.: torch.chunk(tensor4, 2, dim=0)

张量的切分（指定切分大小，如果不能整切，那么最后一个不足的维度会单独作为一个张量）

- torch.split(tensor: torch.Tensor, split_size_or_sections: Union[int, List[int]], dim: int = 0)
  - tensor：待切分张量
  - split_size_or_sections：切分后每个张量大小或不同大小构成的列表
  - dim：切分维度
- ✓ e.g.: torch.split(tensor4, 2, dim=0)

张量的元素操作

汇聚操作（选讲）：相当于是对input Tensor进行重采样

```
out[i][j][k] = input[index[i][j][k]][j][k]  # if dim == 0
out[i][j][k] = input[i][index[i][j][k]][k]  # if dim == 1
out[i][j][k] = input[i][j][index[i][j][k]]  # if dim == 2
```

tensor6 = torch.gather(input=tensor4, dim=0, index=torch.LongTensor([[0, 1, 0, 1], [1, 0, 1, 0], [0, 0, 0, 1]]))

原始：

2 (0, 0)	0 (0, 1)	2 (0, 2)	8 (0, 3)
6 (1, 0)	7 (1, 1)	2 (1, 2)	0 (1, 3)
3 (2, 0)	2 (2, 1)	0 (2, 2)	6 (2, 3)

dim=0

torch.LongTensor(
[[0, 1, 0, 1],
[1, 0, 1, 0],
[0, 0, 0, 1]]))

2 (0, 0)	7 (1, 1)	2 (0, 2)	0 (1, 3)
6 (1, 0)	0 (0, 1)	2 (1, 2)	8 (0, 3)
2 (0, 0)	0 (0, 1)	2 (0, 2)	0 (1, 3)

张量的元素操作

散射操作（选讲）：散射操作可以看成是汇聚操作的逆运算。

```
self[index[i][j][k]][j][k] = src[i][j][k]  # if dim == 0
self[i][index[i][j][k]][k] = src[i][j][k]  # if dim == 1
self[i][j][index[i][j][k]] = src[i][j][k]  # if dim == 2
```

torch.zeros(tensor6.shape, dtype=torch.int64).scatter_(dim=0, index=torch.LongTensor([[0, 1, 0, 1], [1, 0, 1, 0], [0, 0, 0, 1]]), src=tensor6)

原始：

0 (0, 0)	0 (0, 1)	0 (0, 2)	0 (0, 3)
0 (1, 0)	0 (1, 1)	0 (1, 2)	0 (1, 3)
0 (2, 0)	0 (2, 1)	0 (2, 2)	0 (2, 3)

dim=0

torch.LongTensor(
[[0, 1, 0, 1],
[1, 0, 1, 0],
[0, 0, 0, 1]]))

src:

2 (0, 0)	7 (1, 1)	2 (0, 2)	0 (1, 3)
6 (1, 0)	0 (0, 1)	2 (1, 2)	8 (0, 3)
2 (0, 0)	0 (0, 1)	2 (0, 2)	0 (1, 3)

按照下标依次进行赋值：

2 (0, 0)	0 (0, 1)	2 (0, 2)	8 (0, 3)
6 (1, 0)	7 (1, 1)	2 (1, 2)	0 (1, 3)
0 (2, 0)	0 (2, 1)	0 (2, 2)	0 (2, 3)

张量的元素操作

获取指定的行和列

- torch.index_select(input, dim, index)
- ✓ e.g.: torch.index_select(tensor4, 0, torch.LongTensor([0, 2])) # 获取指定的行 <==> tensor4[[0, 2], :]
- ✓ e.g.: torch.index_select(tensor4, 1, torch.LongTensor([1, 3])) # 获取指定的列 <==> tensor4[:, [1, 3]]

使用掩膜获取指定元素

- torch.masked_select(input, mask)
- ✓ e.g.:

```
# 生成掩膜
mask = torch.BoolTensor((tensor4>4).bool())
# 使用掩膜提取tensor4中数据
torch.masked_select(tensor4, mask)
```

张量的元素操作

维度修改

1. 维度交换：torch.Tensor.transpose(dim)　　e.g.: tensor4 = tensor4.transpose(0, 1)
2. 维度重排：torch.Tensor.permute(*dims)　　e.g.: tensor4 = tensor4.permute(1, 0)
3. 增加维度：torch.Tensor.unsqueeze(dim)　　e.g.: tensor41 = tensor4.unsqueeze(1).unsqueeze(-1)
4. 降低维度：torch.Tensor.squeeze(dim)　　e.g.: tensor41.squeeze(dim=1) # 当不指定具体维度时会将所有维度为1的维度删除
5. 删除维度：torch.Tensor.unbind(dim)　　e.g.: tensor4.unbind(0) # 例如将dim=0的维度删除，而tensor4原始有3行，所以最后删除该维度时会生成3个张量，每个张量就是一行。

删除维度与降低维度的不同点在于，降低维度只能够将维度为1的维度删除，而删除维度会强制删除某个维度，如果维度不为1，那么会沿着该维度进行切分。

张量的计算操作

张量的简单计算

尺寸相同时可以直接对元素相运算；尺寸不同时支持广播运算规则。

```
print("tensor4:", tensor4.shape)
print("tensor7:", tensor7.shape)
print("tensor4 - tensor7:\n", tensor4 - tensor7)
```

executed in 16ms, finished 10:30:50 2023-10-23

```
tensor4: torch.Size([3, 4])
tensor7: torch.Size([4])
tensor4 - tensor7:
 tensor([[ 1.9346,  0.1155,  1.3009,  7.8812],
        [ 5.9346,  7.1155,  1.3009, -0.1188],
        [ 2.9346,  2.1155, -0.6991,  5.8812]])
```

查看广播计算张量的变化结果

tensor4b, tensor7b = torch.broadcast_tensors(tensor4, tensor7)

print("tensor4b:", tensor4b.shape)

print("tensor7b:", tensor7b.shape)

```
# 查看广播计算张量的变化结果
tensor4b, tensor7b = torch.broadcast_tensors(tensor4, tensor7)
print("tensor4b:", tensor4b.shape)
print("tensor7b:", tensor7b.shape)
```

executed in 12ms, finished 10:30:50 2023-10-23

```
tensor4b: torch.Size([3, 4])
tensor7b: torch.Size([3, 4])
```

张量之间的计算规则与数组基本一致，只有一条要求，必须要求相互计算的张量位于同一个设备上。

张量的计算操作

矩阵乘法：符号@

```
1 tensor81 = torch.randn(size=(5, 2))
2 tensor82 = torch.randn(size=(2, 10))
3 (tensor81 @ tensor82).shape
```

executed in 24ms, finished 10:30:53 2023-10-23

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

限定张量元素值范围，夹紧到区间 [min,max]

- torch.clamp(input, min=None, max=None)

```
1 a = torch.randn(2, 4)
2 print(a)
3 torch.clamp(a, -0.5, 0.5)
```

executed in 23ms, finished 10:30:54 2023-10-23

```
tensor([[ 1.6036,  0.6247,  0.4567, -0.4334],
        [ 0.4349,  0.7140, -1.7777, -1.5939]])

tensor([[ 0.5000,  0.5000,  0.4567, -0.4334],
        [ 0.4349,  0.5000, -0.5000, -0.5000]])
```

向下取整

- torch.floor(input)

```
1 a = torch.randn(4)
2 print(a)
3 torch.floor(a)
```

executed in 9ms, finished 10:30:55 2023-10-23

```
tensor([ 0.3045, -0.6240, -2.2791, -0.5166])

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

向上取整

- torch.ceil(input)

```
1 torch.ceil(a)
```

executed in 12ms, finished 10:30:56 2023-10-23

```
tensor([ 1., -0., -2., -0.])
```

张量的计算操作

计算除法并获取余数

- torch.fmod(input, other)

```
1 torch.fmod(torch.Tensor([-3, -2, -1, 1, 2, 3]), 2)
```

executed in 13ms, finished 10:30:56 2023-10-23

```
tensor([-1., -0., -1.,  1.,  0.,  1.])
```

获取浮点数小数部分

- torch.frac(input)

```
1 torch.frac(torch.Tensor([1., 2.5, -3.2]))
```

executed in 10ms, finished 10:30:57 2023-10-23

```
tensor([ 0.0000,  0.5000, -0.2000])
```

线性插值

$$out_i = start_i + weight \times (end_i\text{-}start_i)$$

```
]: 1 start = torch.arange(1., 5.)
   2 end = torch.ones((4,)) * 10
   3 print(start)
   4 print(end)
```

executed in 12ms, finished 10:30:58 2023-10-23

```
tensor([1., 2., 3., 4.])
tensor([10., 10., 10., 10.])
```

```
]: 1 torch.lerp(start, end, 0.5)
```

executed in 10ms, finished 10:30:58 2023-10-23

```
5]: tensor([5.5000, 6.0000, 6.5000, 7.0000])
```

自动求导机制

在Pytorch中每个张量都有一个requires_grad属性去控制该张量在进行计算的时候是否需要求出计算梯度。当我们需要求某个张量在计算过程中的梯度，只需将requires_grad设置为True即可。

在进行相关计算后，调用结果张量的backward()方法即可将计算过程中涉及到的所有张量关于结果变量的梯度计算出来。

要查看某个张量对于结果张量的梯度，只需调用张量的grad属性即可。

```
1.  x = torch.tensor([2.0], requires_grad=True)
2.  a = torch.tensor([4.0], requires_grad = True)
3.  # 执行计算
4.  y = x * a
5.  # 反向传播，计算梯度
6.  y.backward()
7.  # 查看执行计算后的梯度
8.  print(x.grad)
9.  print(a.grad)
```

```
tensor([4.])
tensor([2.])
```

```
1 # 再次计算，会发生梯度累加，变量的梯度不会自动归零
2 y = x * a
```

executed in 6ms, finished 12:59:50 2023-10-23

```
1 # 反向传播，计算梯度
2 y.backward()
3 # 查看执行计算后的梯度
4 print(x.grad)
5 print(a.grad)
```

executed in 8ms, finished 12:59:51 2023-10-23

```
tensor([8.])
tensor([4.])
```

需要注意的是，再次执行计算后梯度会进行累加，所以后续在模型训练时，每次迭代都需要对参数梯度进行置零操作。

GPU相关

1. torch.cuda.current_device() # 查看当前GPU编号
2. torch.cuda.is_available() # 查看当前GPU是否可用
3. torch.cuda.device_count() # 查看可用GPU数量
4. torch.cuda.get_device_name(0) # 查看当前GPU型号
5. torch.cuda.set_device(0) # 设置当前使用的GPU，只有在有多张显卡时才使用
6. # 设置GPU可见性，当有多个GPU时，只想要程序访问其中某几个GPU这个时候可以设置环境变量

```
import os
os.environ['CUDA_VISIBLE_DEVICES'] = "0,1"  # 设备之间使用, 进行隔开
```

CPU相关

在 PyTorch 中，OpenMP（Open Multi-Processing）线程数控制着底层的并行处理，主要用于多核 CPU 上的并行计算。它对于CPU计算密集型任务非常重要，尤其是在训练深度学习模型时。

以下是 OpenMP 线程数的主要作用：

1. **并行化计算**：在深度学习中，许多操作可以进行并行计算，例如矩阵乘法、卷积、梯度计算等。OpenMP 线程数用于将这些操作并行化，以充分利用多核 CPU 的计算资源，提高训练速度。
2. **提高CPU利用率**：通过增加线程数，可以更好地利用 CPU 的多核心，从而提高 CPU 利用率。这对于高性能计算任务非常重要，因为它可以加速任务的执行，减少计算时间。
3. **降低计算时间**：通过合理设置 OpenMP 线程数，你可以在训练深度学习模型时显著减少计算时间。在大型数据集和复杂模型的情况下，有效的并行计算可以大大加速训练过程。
4. **控制并行度**：OpenMP 线程数允许你控制并行计算的级别。你可以根据系统的硬件配置和任务的性质来选择合适的线程数，以平衡性能和资源消耗。

默认情况下，OpenMP 线程数通常设置为系统上可用的 CPU 核心数

```
torch.get_num_threads()  # 查看当前OpenMP线程数
torch.set_num_threads(8)  # 设置当前OpenMP线程数
```

谢谢