教育部产学合作协同育人项目资助
2022年北京交通大学《深度学习》课程

第3讲 深度学习框架简介及PyTorch入门

主讲教师：丛润民
助　　教：杨　宁

目录

3.1 开源框架概述

在这个图像中有不同的分类：**猫**，**骆驼**，**鹿**，**大象**等。

卷积神经网络（CNNs）对于这类图像分类任务十分有效。

从头开始实现一个卷积神经网络模型?

我们可能需要几天（甚至几周）才能得到一个有效的模型，时间成本太高！

深度学习框架让一切变得可能！

3.1 开源框架概述

什么是深度学习框架?

深度学习框架是一种接口、库或工具，利用预先构建和优化好的组件集合定义模型。

一个良好的深度学习框架应具备的关键特性:

- ✓ 性能优化
- ✓ 易于理解和编码
- ✓ 良好的社区支持
- ✓ 并行处理以减少计算
- ✓ 自动计算梯度

目录

3.2 核心组件

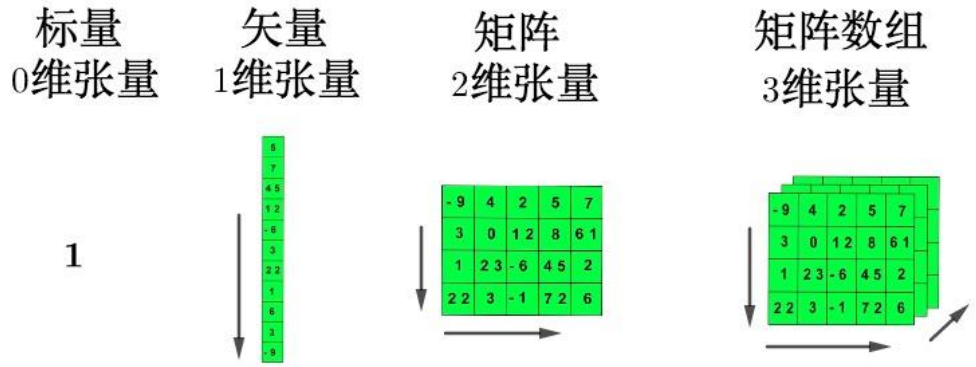
深度学习框架核心组件：

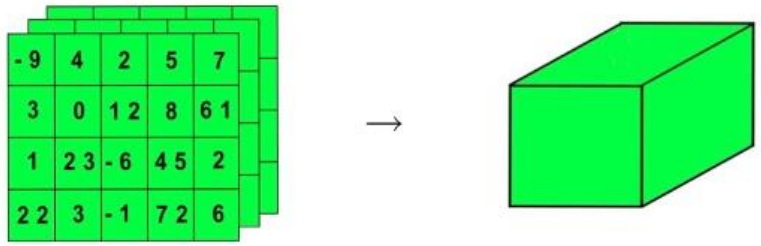
- 张量
- 基于张量的操作
- 计算图
- 自动微分工具
- BLAS、cuBLAS、cuDNN等拓展包

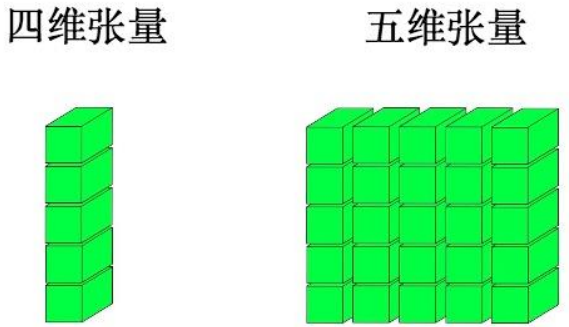
3.2 核心组件-张量

- 张量是深度学习框架中最核心的组件，Tensor实际上就是一个多维数组。

- Tensor对象具有3个属性：
  - rank： number of dimensions
  - shape: number of rows and columns
  - type: data type of tensor's elements





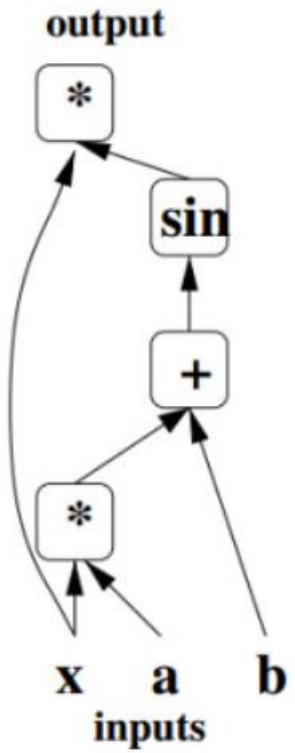


3.2 核心组件-基于张量的操作

■ 张量的相关操作

- 类型转换：字符串转为数字、转为64（32）位浮点类型（整型）等。
- 数值操作：按指定类型与形状生成张量、正态（均匀）分布随机数、设置随机数种子。
- 形状变换：将数据的shape按照指定形状变化、插入维度、将指定维度去掉等。
- 数据操作：切片操作、连接操作等。
- 算术操作：求和、减法、取模、三角函数等。
- 矩阵相关的操作：返回给定对角值的对焦tensor、对输入进行反转、矩阵相乘、求行列式…

3.2 核心组件-计算图



■ 计算图结构

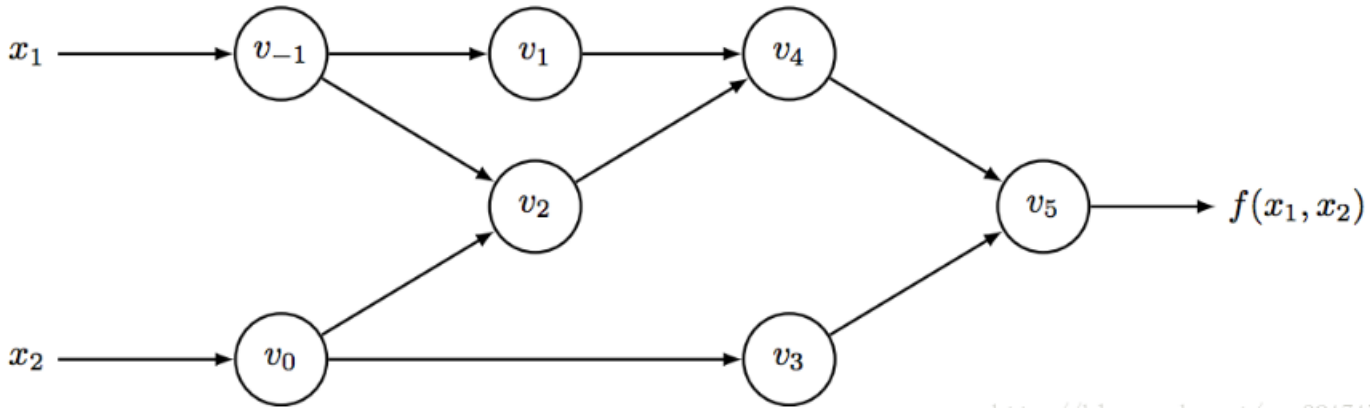
- 用不同的占位符（*，+，**sin**）构成操作结点，以字母**x**、**a**、**b**构成变量结点。
- 用有向线段将这些结点连接起来，组成一个表征运算逻辑关系的清晰明了的“图”型数据结构。

[1] Bengio Y. Learning deep architectures for AI[J]. Foundations and trends® in Machine Learning, 2009, 2(1): 1-127.

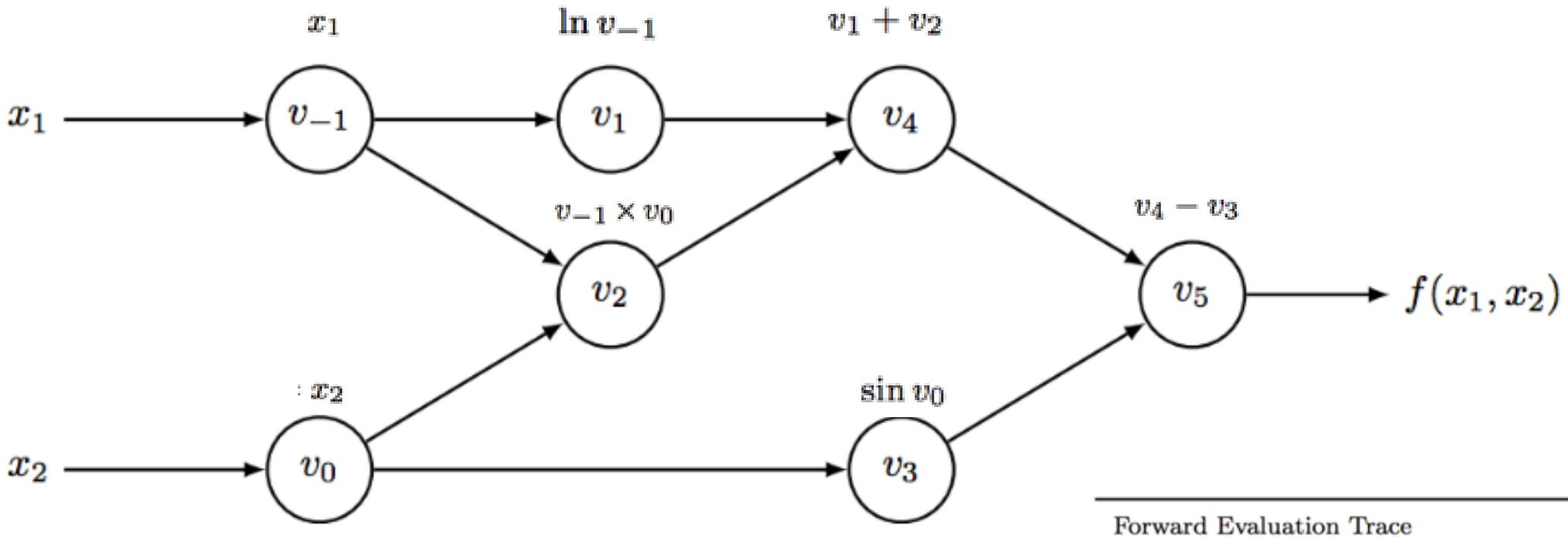
3.2 核心组件-自动微分工具

- 计算图带来的一个好处是让模型训练阶段的梯度计算变得模块化且更为便捷，也就是自动微分法。

- 自动微分Forward Mode：给定函数 $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



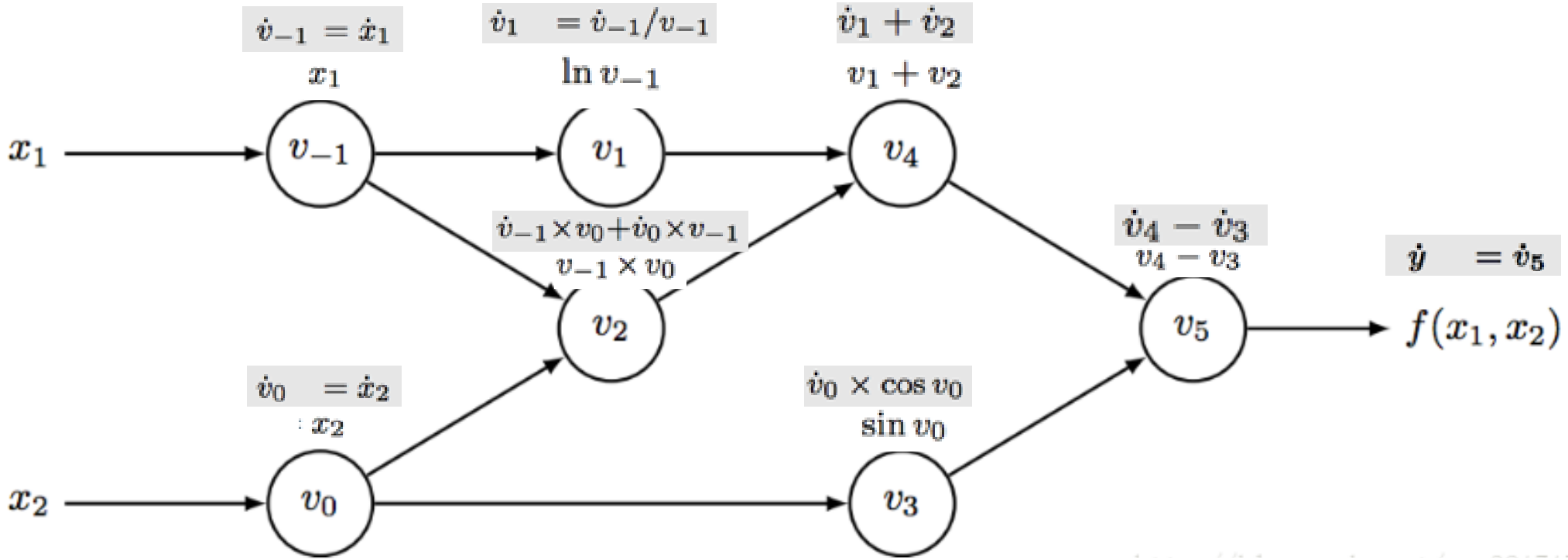
3.2 核心组件-自动微分工具



Forward Evaluation Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

f(x1,x2)=ln(x1)+x1x2−sin(x2)

3.2 核心组件-自动微分工具



f(x1,x2)=ln(x1)+x1x2−sin(x2)

Forward Evaluation Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Derivative Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y}$	$= \dot{v}_5$	$= 5.5$

3.2 核心组件-BLAS、cuBLAS、cuDNN等拓展包

- 通过前面所介绍的组件，已经可以搭建一个全功能的深度学习框架：

  将待处理数据转换为张量，针对张量施加各种需要的操作，通过自动微分对模型展开训练，然后得到输出结果开始测试。

- 存在的缺陷是运行缓慢

- 利用扩展包来进行加速，例如：

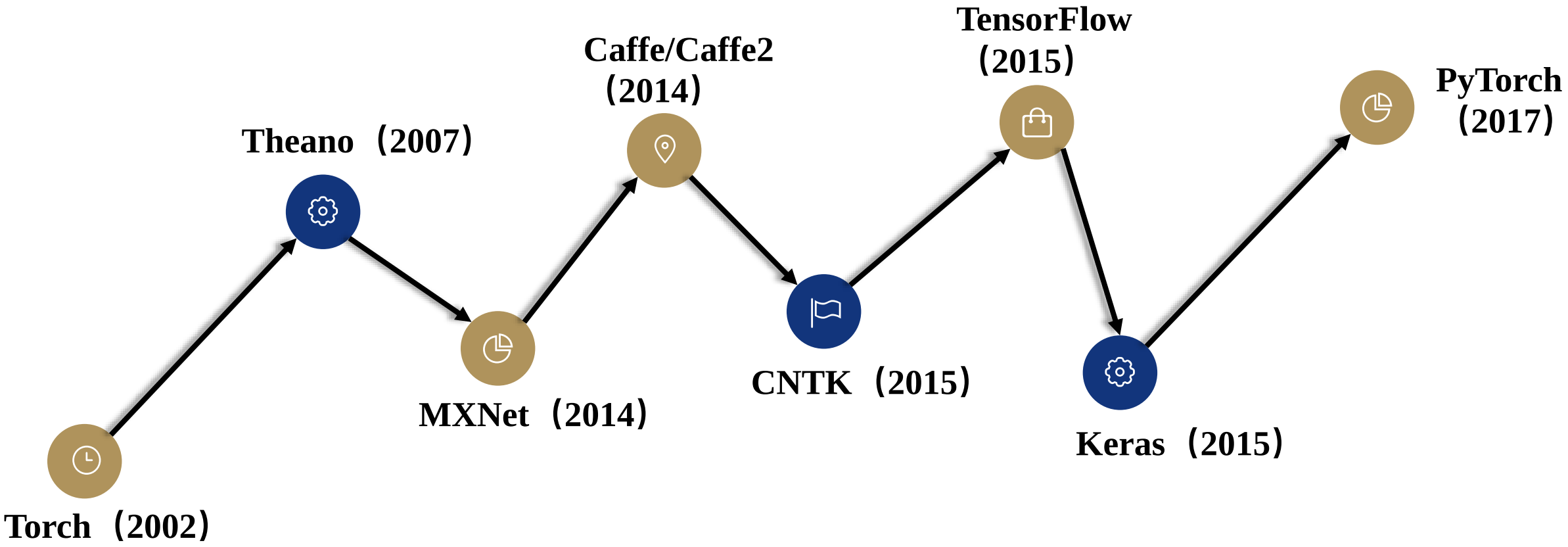
  ✓ Fortran实现的BLAS（基础线性代数子程序）

  ✓ 英特尔的MKL（Math Kernel Library）

  ✓ NVIDIA推出的针对GPU优化的cuBLAS和cuDNN等更据针对性的库

目录

3.3 发展历程


Torch（2002）
Theano（2007）
MXNet（2014）
Caffe/Caffe2
（2014）
CNTK（2015）
TensorFlow
（2015）
Keras（2015）
PyTorch
（2017）

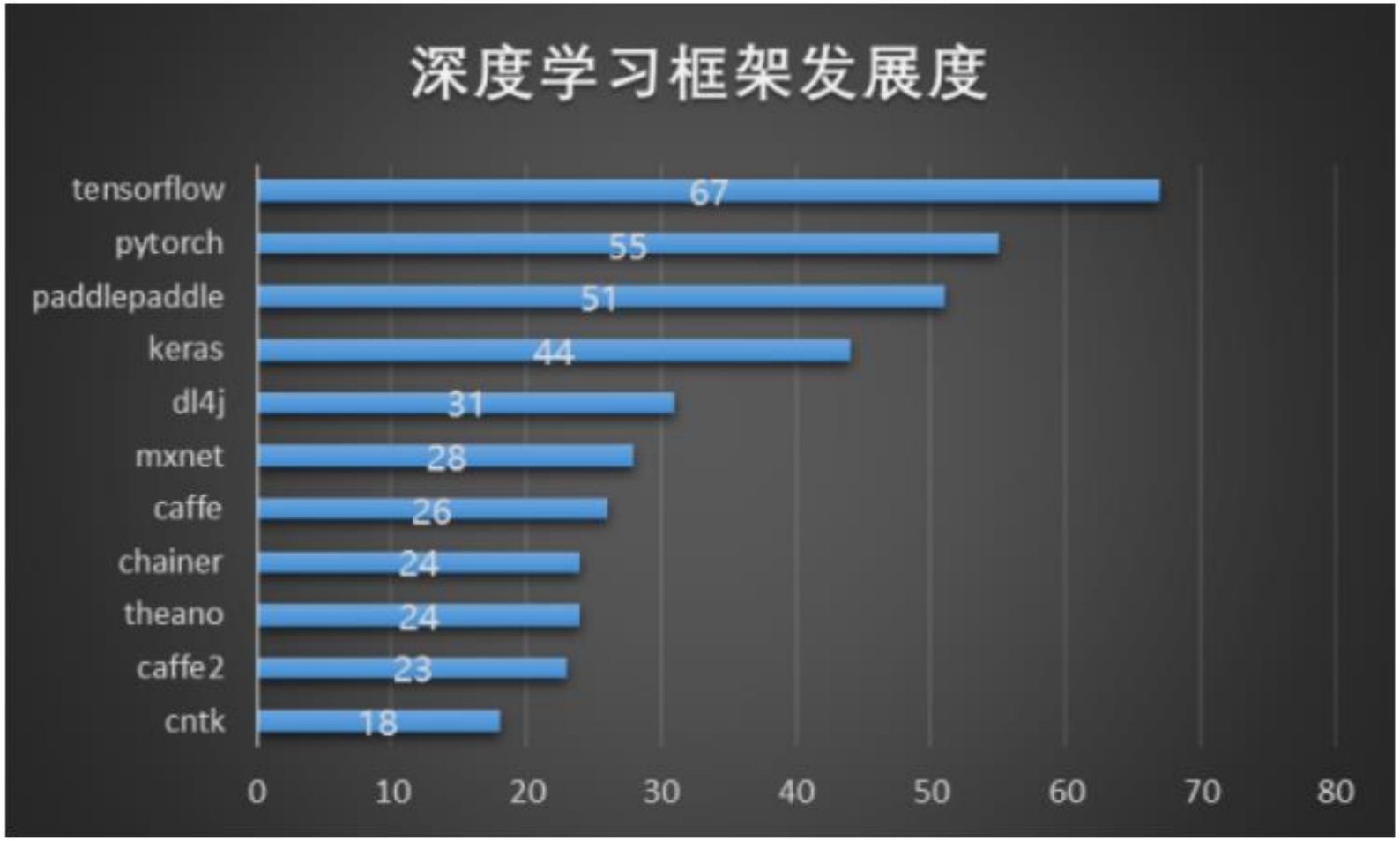
3.3 发展历程

下图是各个主流开源框架在Github上的数据统计（2020年5月）。

框架	机构	支持语言	Stars	Forks	Contributors
Torch	Facebook	Lua	8.5K	2.4K	130
Theano	U.Montreal	Python	9.1K	2.5K	332
MXNet	DMLC	Python/C++/…	18.7K	6.6K	794
Caffe	BVLC	C++/Python	30.3K	18.3K	265
CNTK	Microsoft	C++	16.8K	4.4K	199
TensorFlow	Google	Python/C++/…	145K	81.3K	2496
Keras	fchollet	Python	48.3K	18.3K	817
PyTorch	Facebook	Python/C++/…	38.9K	10K	1406

3.3 发展历程

从业界影响、资源投入、开发生态、文档体系、模型全面性、工业实践和开源热度(GitHub)等七个方面评估各框架的发展状况，结果如下图。



目录

■ 深度学习的主流框架

- ✓ Caffe
- ✓ Theano
- ✓ Torch & PyTorch
- ✓ TensorFlow
- ✓ Keras
- ✓ MXNet

Caffe

- Caffe的全称是Convolutional Architecture for Fast Feature Embedding，它是一个清晰、高效的深度学习框架，于2013年底由加州大学伯克利分校开发，核心语言是C++。它支持命令行、Python和MATLAB接口。Caffe的一个重要特色是可以在不编写代码的情况下训练和部署模型。

https://github.com/BVLC/caffe

- Caffe2是由Facebook组织开发的一个轻量级的深度学习框架，具有模块化和可扩展性等特点。它在原来的Caffe的基础上进行改进，提高了它的表达性，速度和模块化，现在被并入Pytorch项目。Caffe曾经名噪一时，但由于使用不灵活、代码冗长、安装困难、不适用构建循环网络等问题，已经很少被使用。

https://github.com/caffe2

Caffe

优点

- ✓ 核心程序用C++编写，因此更高效，适合工业界开发
- ✓ 网络结构都是以配置文件形式定义，不需要用代码设计网络
- ✓ 拥有大量的训练好的经典模型（AlexNet、VGG、Inception)在其 Model Zoo里

缺点

- ✓ 缺少灵活性和扩展性
- ✓ 依赖众多环境，难以配置
- ✓ 缺乏对递归网络RNN和语言建模的支持，不适用于文本、声音或时间序列数据等其他类型的深度学习应用

Theano

- Theano是深度学习框架的鼻祖，由Yoshua Bengio和蒙特利尔大学的研究小组于2007年创建，是率先广泛使用的深度学习框架。Theano 是一个 Python 库，速度更快，功能强大，可以高效的进行数值表达和计算，可以说是从NumPy矩阵表达向tensor表达的一次跨越，为后来的深度学习框架提供了样板。遗憾的是Theano团队2017年已经停止了该项目的更新，深度学习应用框架的发展进入到了背靠工业界大规模应用的阶段。
- 缺点：

✓ 原始的 Theano 只有比较低级的 API；大型模型可能需要很长的编译时间；不支持多 GPU；有的错误信息的提示没有帮助。

https://github.com/Theano

Torch & PyTorch

- Torch是一个有大量机器学习算法支持的科学计算框架，其诞生已经有十余年，但真正起势得益于Facebook开源了大量Torch的深度学习模块和扩展。Torch的一个特殊之处是采用了Lua编程语言（曾被用来开发视频旅游）。

https://github.com/torch

- PyTorch于2016年10月发布，是一款专注于直接处理数组表达式的低级API。 前身是Torch。Facebook人工智能研究院对PyTorch提供了强力支持。PyTorch支持动态计算图，为更具数学倾向的用户提供了更低层次的方法和更多的灵活性，目前许多新发表的论文都采用PyTorch作为论文实现的工具，成为学术研究的首选解决方案。

https://github.com/pytorch

PyTorch

■ 优点

- ✓ 简洁易用
- ✓ 可以为使用者提供更多关于深度学习实现的细节，如反向传播和其他训练过程
- ✓ 活跃的社区：提供完整的文档和指南
- ✓ 代码很**Pythonic**（简洁、优雅）

■ 缺点

- ✓ 无可视化接口和工具
- ✓ 导出模型不可移植，工业部署不成熟
- ✓ 代码冗余量较大

TensorFlow

- TensorFlow最初由Google Brain团队针对机器学习和深度神经网络进行研究所开发的，目前开源之后可以在几乎各种领域适用。它灵活的架构可以部署在一个或多个CPU、GPU的台式及服务器中，或者使用单一的API应用在移动设备中。TensorFlow可以说是当今十分流行的深度学习框架，Airbnb、DeepMind、Intel、Nvidia、Twitter以及许多其他著名公司都在使用它。TensorFlow提供全面的服务，构建了活跃的社区，完善的文档体系，大大降低了我们的学习成本，另外，TensorFlow有很直观的计算图可视化呈现。模型能够快速的部署在各种硬件机器上，从高性能的计算机到移动设备，再到更小的更轻量的智能终端。
- 但是，TensorFlow相比Pytorch，Caffe等框架，计算速度很慢。而且通过它构建一个深度学习框架需要更复杂的代码，还要忍受重复的多次构建静态图。

https://github.com/tensorflow

TensorFlow

■ 优点

- ✓ 自带tensorboard可视化工具，能够让用户实时监控观察训练过程
- ✓ 拥有大量的开发者，有详细的说明文档、可查询资料多
- ✓ 支持多GPU、分布式训练，跨平台运行能力强
- ✓ 具备不局限于深度学习的多种用途，还有支持强化学习和其他算法的工具

■ 缺点

- ✓ 频繁变动的接口
- ✓ 接口设计过于晦涩难懂

Keras

- Keras于2015年3月首次发布，拥有“为人类而不是机器设计的API”，由Google的Francis Chollet创建与维护，它是一个用于快速构建深度学习原型的高层神经网络库，由纯Python编写而成，以TensorFlow，CNTK，Theano和MXNet为底层引擎，提供简单易用的API接口，能够极大地减少一般应用下用户的工作量。 能够和TensorFlow，CNTK或Theano配合使用。通过Keras的API可以仅使用数行代码就构建一个网络模型，Keras+Theano，Keras+CNTK的模式曾经深得开发者喜爱。目前Keras整套架构已经封装进了TensorFlow，在TF.keras可以完成Keras的所有事情。

https://keras.io/

Keras

■ 优点

- ✓ 更简洁，更简单的API
- ✓ 丰富的教程和可重复使用的代码
- ✓ 更多的部署选项（直接并且通过TensorFlow后端），更简单的模型导出
- ✓ 支持多GPU训练

■ 缺点

- ✓ 过度封装导致丧失灵活性，导致用户在新增操作或是获取底层的数据信息时过于困难
- ✓ 许多BUG都隐藏于封装之中，无法调试细节
- ✓ 初学者容易依赖于 Keras 的易使用性而忽略底层原理

MXNet

- MXNet于2014年由上海交大校友陈天奇与李沐组建团队开发，2017年1月，MXNet项目进入Apache基金会，成为Apache的孵化器项目。MXNet主要用C++编写，强调提高内存使用的效率，甚至能在智能手机上运行诸如图像识别等任务。
- 它拥有类似于 Theano 和 TensorFlow 的数据流图，为多 GPU 配置提供了良好的配置，还有着类似于 Blocks 等更高级别的模型构建块，并且可以在任何硬件上运行（包括手机）。同时MXNet 是一个旨在提高效率和灵活性的深度学习框架，提供了强大的工具来帮助开发人员利用GPU和云计算的全部功能。

https://mxnet.apache.org/

MXNet

■ 优点

- ✓ 灵活的编程模型
- ✓ 从云端到客户端可移植
- ✓ 多语言支持
- ✓ 本地分布式训练
- ✓ 性能优化

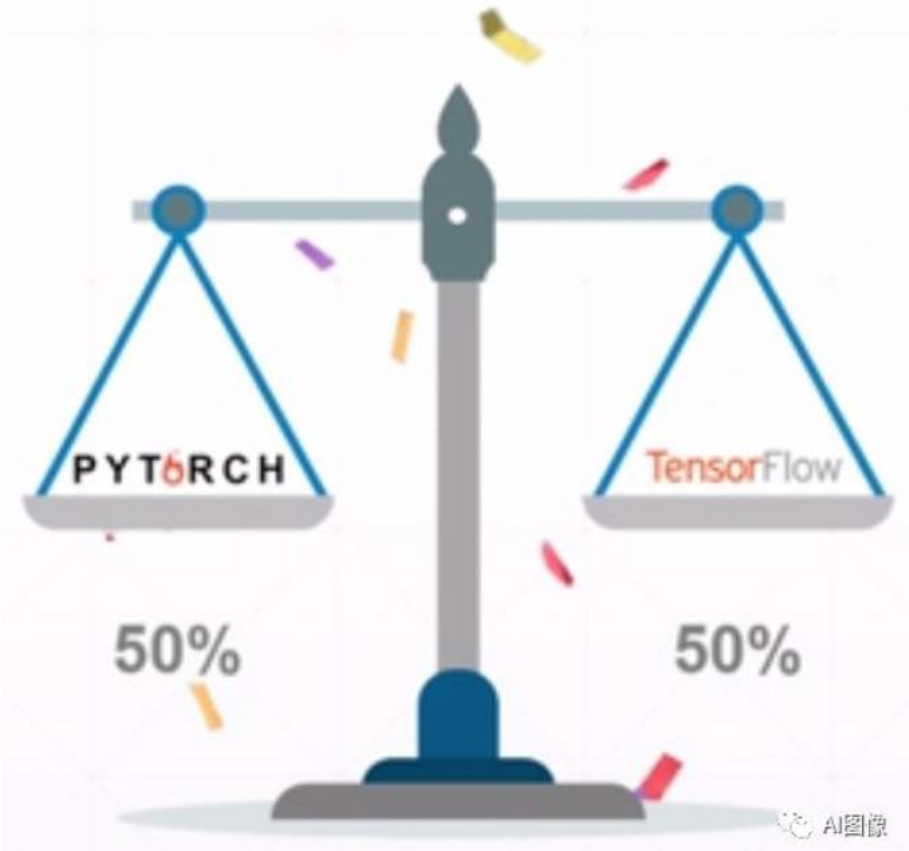
■ 缺点

- ✓ 社区相对PyTorch和TensorFlow来说相对小众

目录

3.5 Tensorflow与PyTorch比较分析

通过分析，目前来看TensorFlow和PyTorch框架是业界使用最为广泛的两个深度学习框架，TensorFlow在工业界拥有完备的解决方案和用户基础，PyTorch 得益于其精简灵活的接口设计，可以快速设计和调试网络模型，在学术界获得好评如潮。



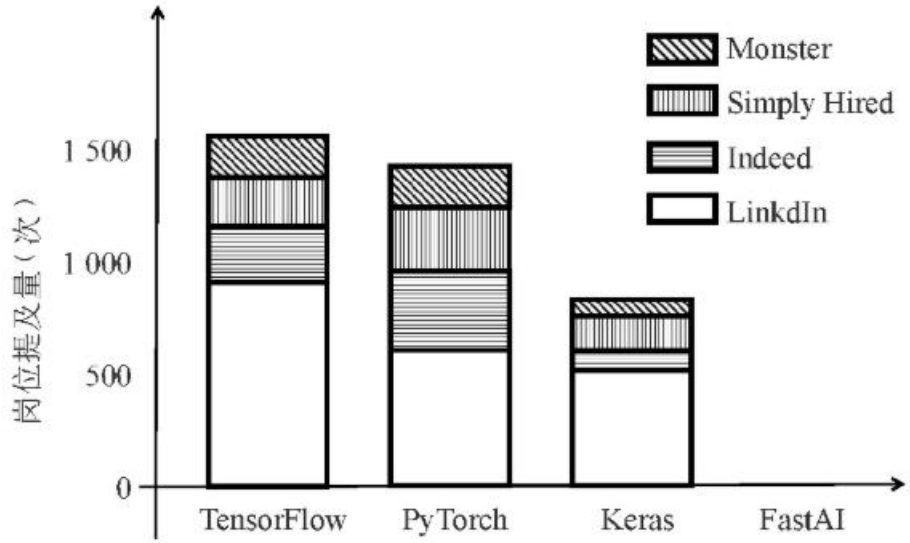
3.5 Tensorflow与PyTorch比较分析

■ **TensorFlow和PyTorch在学术界、工业界的使用现状。**

学术界（左图）：基于五大级会论文的使用率；

工业界（右图）：通过线上招聘启事中的提及率。

Conference	PT 2018	PT 2019	PT Growth	TF 2018	TF 2019	TF Growth
CVPR	82	280	241. 5%	116	125	7. 8%
NAACL	12	66	450. 0%	34	21	−38. 2%
ACL	26	103	296. 2%	34	33	−2. 9%
ICLR	24	70	191. 7%	54	53	−1. 9%
ICML	23	69	200. 0%	40	53	32. 5%



[1] HE H. The State of Machine Learning Frameworks in 2019.(2019-10-10).https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.
[2] HALE J. Which Deep Learning Framework is Growing Fastest.(2019-04-01).https://towardsdatascience.com/which-deeplearning-framework-is-growing-fastest-3f77f14aa318

3.5 Tensorflow与PyTorch比较分析

对比分析

- **运行机制：均是基于张量和有向非循环图（DAG）**
  - TensorFlow：运行代码时,DAG是以静态方式定义的，若需要实现动态DAG，则需要借助TensorFlow Fold库；TensorFlow需要借助特殊的调试工具tfdbg才能进行调试。
  - PyTorch：动态DAG是内置的,可以随时定义、随时更改、随时执行节点，相当灵活；使用者可以使用任何一个喜欢的调试工具,比如PDB、IPDB、PyCharm调试器或者原始的print语句。
- **训练模式：**
  - TensorFlow：使用者必须手动编写代码，并微调要在特定设备上运行的每个操作，以实现分布式训练。
  - PyTorch：需要利用异步执行的本地支持来实现的，其自身在分布式训练比较欠缺。
- **可视化情况：**
  - TensorFlow：内置TensorBoard库强大，可显示模型图，绘制标量变量，实现图像、嵌入可视化、播放音频等功能。
  - PyTorch：可以使用Visdom，但是Visdom提供的功能很简单且有限，可视化效果远远比不上TensorBoard。

[1] 黄玉萍, 梁炜萱, 肖祖环. 基于TensorFlow和PyTorch的深度学习框架对比分析. 现代信息科技, 2020, 4(04): 80-82+87.

3.5 Tensorflow与PyTorch比较分析

对比分析

■ 细化特征比较

序号	参量	TensorFlow	PyTorch
1	安装难易程度	良好	优秀
2	上手难易程度	良好	优秀
3	代码理解程度	良好	优秀
4	API丰富程度	优秀	良好
5	模型丰富程度	优秀	良好
6	社区支持程度	优秀	良好
7	语言支持程度	优秀	良好
8	可视化程度	优秀	良好

■ 使用场景

- TensorFlow：当需要拥有丰富的入门资源、开发大型生产模型、可视化要求较高、大规模分布式模型训练时。
- PyTorch：如果想要快速上手、对于功能性需求不苛刻、追求良好的开发和调试体验、擅长Python化的工具时。

[1] 黄玉萍, 梁炜萱, 肖祖环. 基于TensorFlow和PyTorch的深度学习框架对比分析. 现代信息科技, 2020, 4(04): 80-82+87.

目录

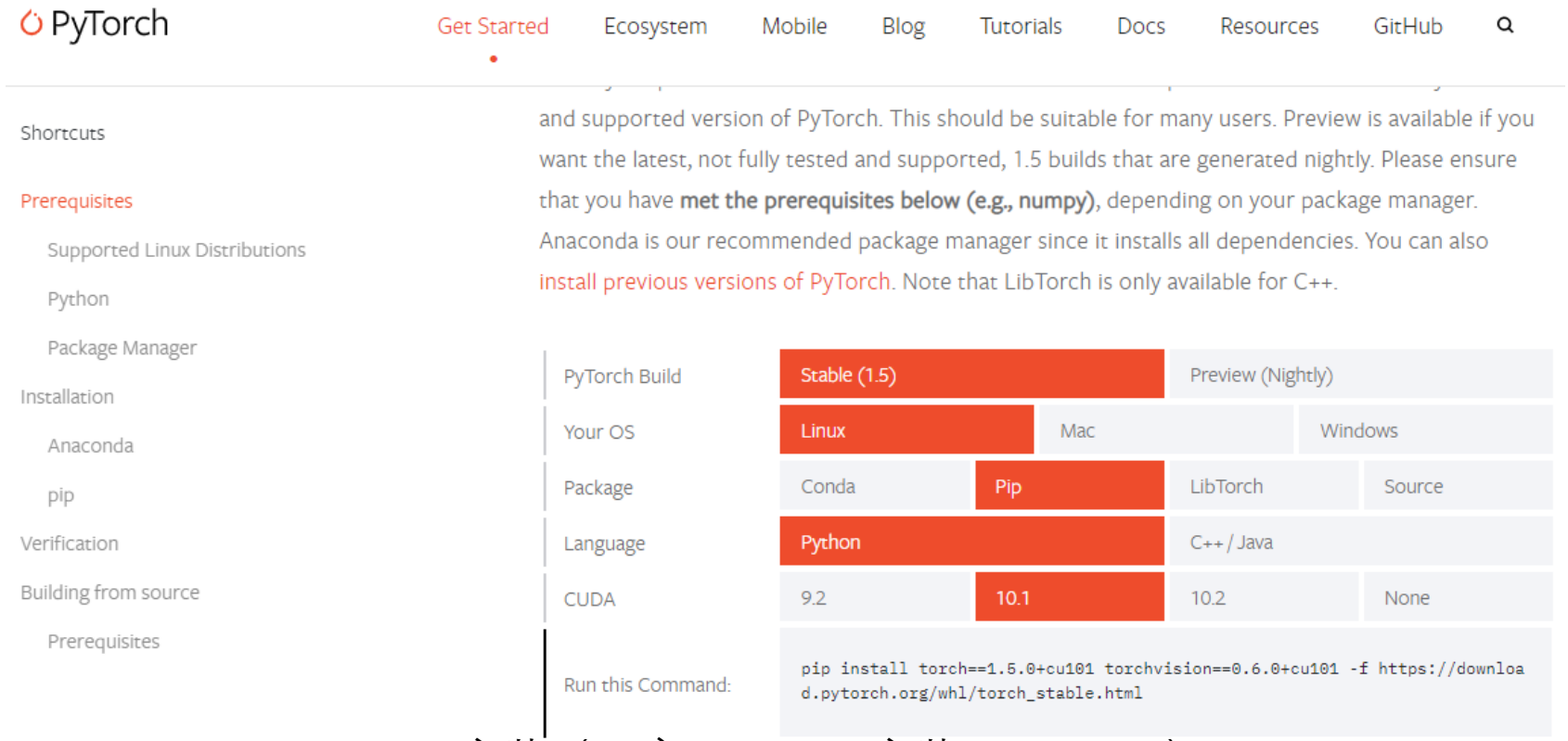
3.6 PyTorch入门-概述

- PyTorch是一个基于Python的库，用来提供一个具有灵活性的深度学习开发平台。PyTorch的工作流程非常接近Python的科学计算库——numpy。
- PyTorch像Python一样简单，可以顺利地与Python数据科学栈集成，取代了具有特定功能的预定义图形，PyTorch为我们提供了一个框架，以便可以在运行时构建计算图，甚至在运行时更改它们。 PyTorch的其他一些优点还包括：多gpu支持，自定义数据加载器和简化的预处理器。

3.6 PyTorch入门-安装

■ 使用pip安装

目前，使用pip安装PyTorch二进制包是最简单、最不容易出错，同时也是最适合新手的安装方式。从PyTorch官网选择操作系统、包管理器pip、Python版本及CUDA版本，会对应不同的安装命令，如图所示： https://pytorch.org/get-started/locally/

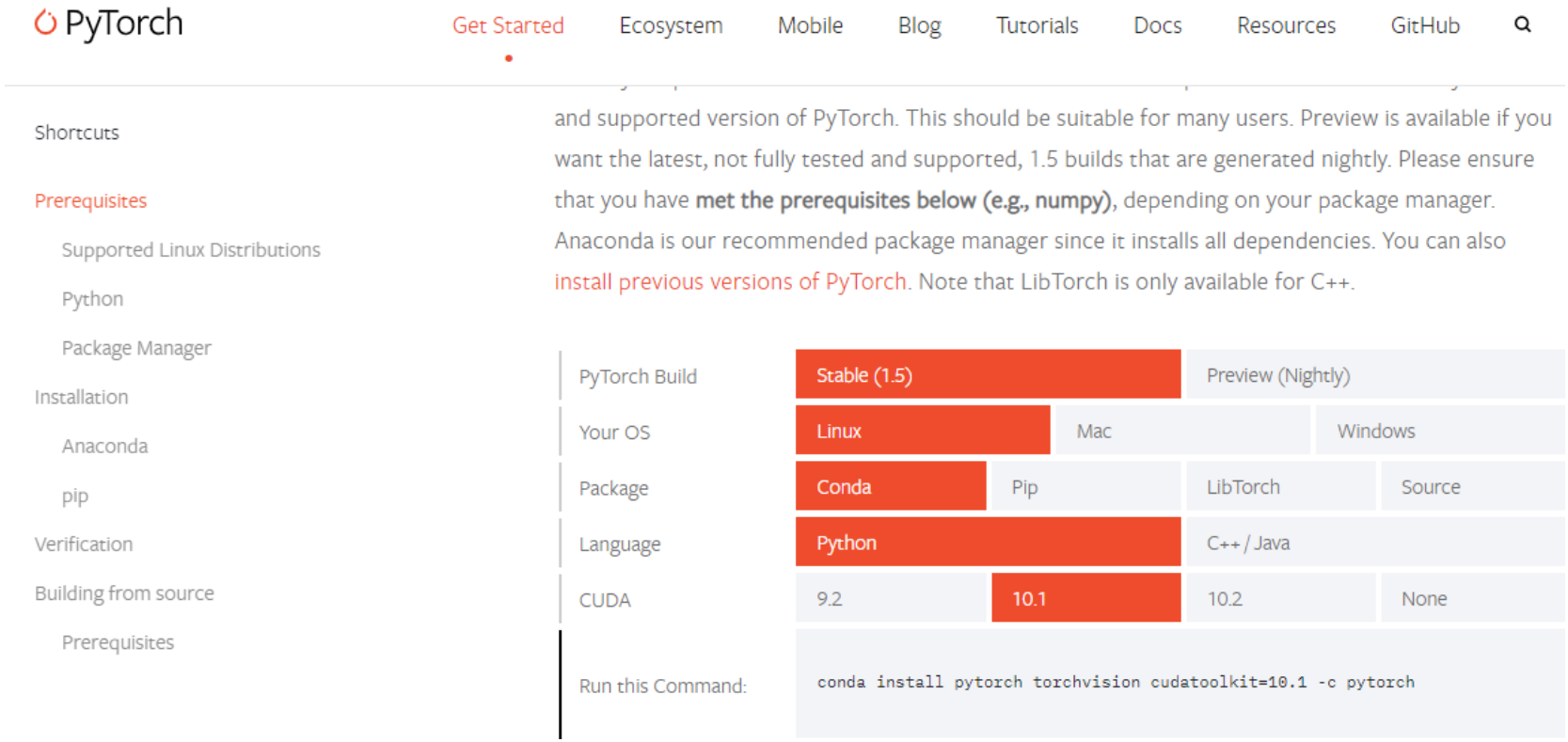


python安装（百度：‘python安装+windows’）

3.6 PyTorch入门-安装

■ 使用conda安装

conda是Anaconda自带的包管理器。如果使用Anaconda作为Python环境，则除了使用pip安装，还可以使用conda进行安装。同样，在PyTorch官网中选择操作系统、包管理器conda、Python版本及CUDA版本，对应不同的安装命令。如图所示：



anaconda安装（百度：‘anaconda安装+windows’）

■ Tensor

Tensor，又名张量。可简单地认为它是一个数组，且支持高效的科学计算。它可以是一个数（标量）、一维数组（向量）、二维数组（矩阵）或更高维的数组（高阶数据）。Tensor和numpy的ndarrays类似，但PyTorch的tensor支持GPU加速。如下定义一个简单的一维矩阵：

```
import torch
torch.FloatTensor([2])
```

与numpy一样，科学计算库非常重要的一点是能够实现高效的数学功能。如下实现一个简单的加法操作：

```
a = torch.FloatTensor([2])
b = torch.FloatTensor([3])
a + b
```

3.6 PyTorch入门-Tensor和autograd

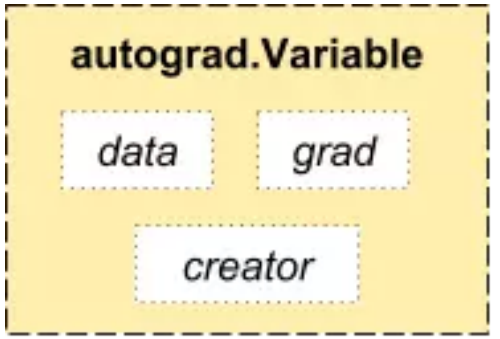
■ autograd

torch.autograd是为了方便用户使用，专门开发的一套自动求导引擎，它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。

1）Variable

PyTorch在autograd模块中实现了计算图的相关功能，autograd中的核心数据结构是Variable。Variable封装了tensor，并记录对tensor的操作记录用来构建计算图。Variable的数据结构如图所示，主要包含如下三个属性：

- data：保存variable所包含的tensor。
- grad：保存data对应的梯度，grad也是variable，而非tensor。
- grad_fn：指向一个Function，记录variable的操作历史。



3.6 PyTorch入门-Tensor和autograd

代码示例一：

```
import numpy as np
import torch
from torch.autograd import Variable

x = Variable(torch.ones(2,2),requires_grad = False)
temp = Variable(torch.zeros(2,2),requires_grad = True)
y = x + temp + 2
y = y.mean()  #求平均数
y.backward()  #反向传递函数，用于求y对前面的变量（x）的梯度
print(x.grad)  # d(y)/d(x)
```

输出：none # 因为requires_grad=False

3.6 PyTorch入门-Tensor和autograd

代码示例二：

```
import numpy as np
import torch
from torch.autograd import Variable

x = Variable(torch.ones(2,2),requires_grad = False)
temp = Variable(torch.zeros(2,2),requires_grad = True)
y = x + temp + 2
y = y.mean()  #求平均数
y.backward()  #反向传递函数，用于求y对前面的变量（x）的梯度
print(temp.grad)  # d(y)/d(temp)
```

输出：tensor([[0.2500, 0.2500],
[0.2500, 0.2500]])

2） 计算图

计算图是一种特殊的有向无环图，用于记录算子与变量之间的关系。一般用矩形表示算子，椭圆形表示变量。

在PyTorch中计算图特点总结如下：

- Autograd根据用户对variable的操作构建计算图。对variable的操作抽象为Function。
- Variable默认是不需要求导的，即requires_grad属性默认为False。
- 多次反向传播时，梯度是累加的。反向传播的中间缓存会被清空。
- PyTorch采用动态图设计，可以很方便地查看中间层的输出，动态地设计计算图结构。

3.6 PyTorch入门-Tensor和autograd

动态图

dynamic graph

动态图与静态图

根据计算图的搭建方式，可以将计算图分为动态图和静态图。

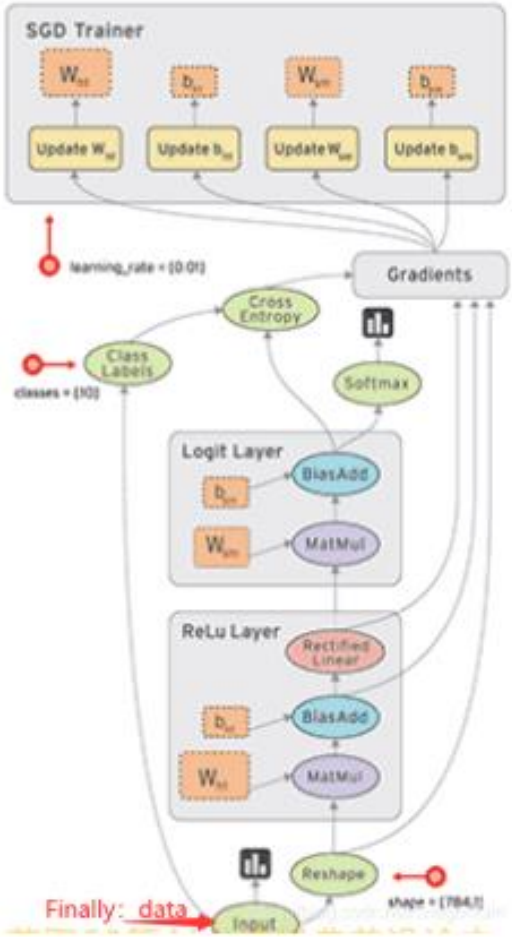
- 动态图采用搭建和运算同时进行：灵活、易调节（debug）
- 静态图为先搭建，后运算：高效但不灵活

pytorch采用动态图机制，tensorflow采用静态图机制

如下左图，tensorflow的搭建机制为先确立好所有的路线流向，再将tensor数据注入跑通整个过程。这也是tensorflow的名字由来

而下右图，pytorch中随着代码的数据定义与构建，同时也确立部分的计算图模块。

3.6 PyTorch入门-Tensor和autograd

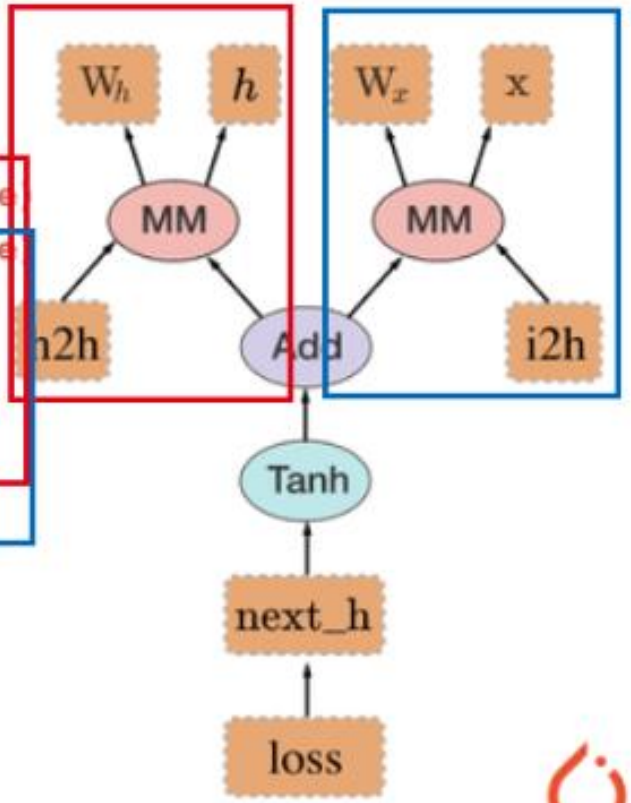


Back-propagation
uses the dynamically created graph

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)

h2h = torch.mm(W_h, prev_h.t())
i2h = torch.mm(W_x, x.t())
next_h = h2h + i2h
next_h = next_h.tanh()

loss = next_h.sum()
loss.backward()  # compute gradients!
```



https://www.cnblogs.com/SakuraYuki/p/13341448.html

■ nn.Module

torch.nn的核心数据结构是Module，它是一个抽象的概念，既可以表示神经网络中的某个层，也可以表示一个包含很多层的神经网络。最常见的做法就是继承nn.Module，编写自己的网络。

- ✓ 能够自动检测到自己的parameter，并将其作为学习参数。
- ✓ 主Module能够递归查找子Module中的parameter。

为了方便用户使用，PyTorch实现了神经网络中绝大多数的layer，这些layer都继承于nn.Module，封装了可学习参数parameter，并实现了forward函数，且专门针对GPU运算进行了CuDNN优化，其速度和性能都十分优异。

可学习的参数：卷积层和全连接层的权重、bias、BatchNorm的γ、β等。
不可学习的参数(超参数)：学习率、batch size、weight decay、模型的深度宽度分辨率等。

3.6 PyTorch入门-nn

具体层可参照官方文档（http://pytorch.org/docs/nn.html），但阅读文档时应主要关注以下几点：

- 构造函数的参数，如nn.Linear(in_features, out_features, bias)，需关注这三个参数的作用。
- 属性、可学习参数和子module。如nn.Linear中有weight和bias两个可学习参数，不包含子module。
- 输入输出的形状，如nn.linear的输入形状是(N, input_features)，输出为(N, output_features)，N是batch_size。

3.6 PyTorch入门-nn

- 要实现一个自定义层大致分以下几个主要的步骤：

1）自定义一个类，继承自Module类，并且一定要实现两个基本的函数

- ✓ 构造函数__init__
- ✓ 前向计算函数forward函数

2）在构造函数_init__中实现层的参数定义。

- ✓ 例如：Linear层的权重和偏置
- ✓ Conv2d层的in_channels, out_channels, kernel_size, stride=1,padding=0, dilation=1, groups=1,bias=True, padding_mode='zeros'这一系列参数

3）在前向传播forward函数里面实现前向运算。

- ✓ 通过torch.nn.functional函数来实现
- ✓ 自定义自己的运算方式。如果该层含有权重，那么权重必须是nn.Parameter类型

4）补充：一般情况下，我们定义的参数是可以求导的，但是自定义操作如不可导，需要实现backward函数。

■ 优化器

PyTorch将深度学习中常用的优化方法全部封装在torch.optim中，其设计十分灵活，能够很方便地扩展成自定义的优化方法。

所有的优化方法都是继承基类optim.Optimizer，并实现了自己的优化步骤。最基本的优化方法是随机梯度下降法（SGD）。需要重点关注的是，优化时如何调整学习率。

➢ 调整学习率

- ✓ 修改optimizer.param_groups中对应的学习率。
- ✓ 新建优化器，由于optimizer十分轻量级，构建开销很小，故可以构建新的optimizer。

■ 随机梯度下降法(SGD)

✓ 每次更新的迭代，只计算一个样本。

代码：

```
#调整学习率，新建一个optimizer
old_lr=0.1
optimizer=optim.SGD([
                {'param':net.features.parameters()},
                {'param':net.classifiers.parameters(),'lr':old_lr*0.5}],lr=1e-5)
```

■ nn.functional

nn中还有一个很常用的模块：nn.funcitonal。nn中的大多数layer在functional中都有一个与之对应的函数。

示例代码：

```
import torch as t
import torch.nn as nn
from torch.autograd import Variable as V

input=V(t.randn(2,3))
model=nn.Linear(3,4)
```

3.6 PyTorch入门-nn

```
output1=model(input)    #得到输出方式1
output2=nn.functional.linear(input,model.weight,model.bias) #得到输出方式2
print(output1==output2)
```

```
>>> output1=model(input)
>>> output2=nn.functional.linear(input,model.weight,model.bias)
>>> print(output1==output2)
tensor([[True, True, True, True],
        [True, True, True, True]])
```

```
b=nn.functional.relu(input)
b2=nn.ReLU()(input)
print(b==b2)
```

```
>>> b=nn.functional.relu(input)
>>> b2=nn.ReLU()(input)
>>> print(b==b2)
tensor([[True, True, True],
        [True, True, True]])
```

3.6 PyTorch入门-nn

■ 构建模型需要注意：

- ✓ 如果模型有可学习的参数时，最好使用nn.Module
- ✓ 激活函数（ReLu、sigmoid、Tanh）、池化（MaxPool）等层没有可学习的参数，可以使用对应的functional函数
- ✓ 卷积、全连接等有可学习参数的网络建议使用nn.Module
- ✓ dropout没有可学习参数，建议使用nn.Dropout

1. 数据处理

在PyTorch中，数据加载可通过自定义的数据集对象实现。数据集对象被抽象为Dataset类，实现自定义的数据集需要继承Dataset，并实现两个Python方法。

- __getitem__：返回一条数据或一个样本。obj[index]等价于obj.__getitem__(index)。
- __len__：返回样本的数量。len(obj)等价于obj.__len__()。

Dataset只负责数据的抽象，一次调用__getitem__只返回一个样本。在训练神经网络时，是对一个batch的数据进行操作，同时还需要对数据进行并行加速等。因此，PyTorch提供了DataLoader帮助实现这些功能。

3.6 PyTorch入门-常用工具

■ **DataLoader函数定义如下：**

DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, num_workers=0, collate_fn=default_collatem, pin_memory=False, drop_last=False)

➢ 参数解释如下：

- ✓ dataset：加载的数据集（Dataset对象）
- ✓ batch_size：batch size（批大小）
- ✓ shuffle：是否将数据打乱
- ✓ sampler：样本抽样
- ✓ num_workers：使用多进程加载的进程数，0代表不使用多进程
- ✓ collate_fn：如何将多个样本数据拼接成一个batch，一般使用默认的拼接方式即可
- ✓ pin_memory：是否将数据保存在pin memory区，pin memory中的数据转到GPU会快一些
- ✓ drop_last：dataset中的数据个数可能不是batch_size的整数倍，drop_last为True会将多出来不足一个batch的数据丢弃

2. 计算机视觉工具包：torchvision

视觉工具包torchvision独立于PyTorch，需要通过pip install torchvision安装。torchvision主要包含以下三个部分：

- ✓ models：提供深度学习中各种经典网络的网络结构及预训练好的模型，包括Net、VGG系列、ResNet系列、Inception系列等。
- ✓ datasets：提供常用的数据集加载，设计上都是继承torch.utils.data.Dataset，主要包括MNIST、CIFAR10/100、ImageNet、COCO等。
- ✓ transforms：提供常用的数据预处理操作，主要包括对Tensor以及PIL Image对象的操作。

3.6 PyTorch入门-常用工具

■ 以MNIST数据集为例：

torchvision.datasets.MNIST（root，train = True，transform = None，target_transform = None，download = False ）

➢ 参数介绍：

- ✓ root（string） - 数据集的根目录在哪里MNIST/processed/training.pt 和 MNIST/processed/test.pt存在
- ✓ train（bool，optional） - 如果为True，则创建数据集training.pt，否则创建数据集test.pt。
- ✓ download（bool，optional） - 如果为true，则从Internet下载数据集并将其放在根目录中。如果已下载数据集，则不会再次下载
- ✓ transform（callable，optional） - 一个函数/转换，它接收PIL图像并返回转换后的版本。例如，transforms.RandomCrop
- ✓ target_transform（callable，optional） - 接收目标并对其进行转换的函数/转换

3、使用GPU加速

在PyTorch中以下数据结构分为CPU和GPU两个版本。

- Tensor
- Variable（包括Parameter）
- nn.Module（包括常用的layer、loss function，以及容器sequential等）

它们都带有一个.cuda方法，调用此方法即可将其转化为对应的GPU对象。如果服务器具有多个GPU，tensor.cuda()方法会将tensor保存到第一块GPU上，等价于tensor.cuda(0)。此时如果想使用第二块GPU，需要手动指定tensor.cuda(1)，而这需要修改大量代码很繁琐。这里有两种替代方式：

■ GPU的使用

- ✓ 调用t.cuda.set_device(1)指定使用第二块GPU，后续的.cuda()都无需更改，切换GPU只需修改这一行代码。
- ✓ 设置环境变量CUDA_VISIBLE_DEVICES
  - export CUDA_VISIBLE_DEVICE=1时（下标是从0开始，1代表第二块GPU）
- ✓ CUDA_VISIBLE_DEVICES还可以指定多个GPU，如：
  - export CUDA_VISIBLE_DEVICES = 0,2,3

4. 保存和加载

在PyTorch中，需要将Tensor、Variable等对象保存到硬盘，以便后续能通过相应的方法加载到内存中。

- ✓ 这些信息最终都是保存成Tensor。
- ✓ 使用t.save和t.load即可完成Tensor保存和加载的功能。
- ✓ 在save/load时可指定使用的pickle模块，在load时还可以将GPU tensor映射到CPU或其他GPU上。
- ✓ 使用方式：
  - t.save(obj, file_name)：保存任意可序列化的对象。
  - obj = t.load(file_name)：方法加载保存的数据。

3.6 PyTorch入门-例子

自定义的简单例子：

- $y=w*sqrt(X^2+bias)$

（1）定义一个自定义层MyLayer

```
class MyLayer(torch.nn.Module):
    def __init__(self, in_features, out_features, bias=True):
        super(MyLayer, self).__init__()  # 和自定义模型一样，第一句话就是调用父类的构造函数
        self.in_features = in_features
        self.out_features = out_features
        self.weight = torch.nn.Parameter(torch.Tensor(in_features, out_features)) # 由于weights是可以训练的，所以使用Parameter来定义。
```

3.6 PyTorch入门-例子

```
        if bias:
            self.bias = torch.nn.Parameter(torch.Tensor(in_features)) # 由于bias是可以训练的，所以使用Parameter来定义
        else:
            self.register_parameter('bias', None)


    def forward(self, input):
        input_=torch.pow(input,2)+self.bias
        input_=torch.sqrt(input_)
        y=torch.matmul(input_,self.weight)
        return y
```

- $y=w*sqrt(X^2+bias)$

（2）自定义模型并且训练

```
import torch
from my_layer import MyLayer # 自定义层

N, D_in, D_out = 10, 5, 3  # 一共10组样本，输入特征为5，输出特征为3
# 先定义一个模型
class MyNet(torch.nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()  # 第一句话，调用父类的构造函数
        self.mylayer1 = MyLayer(D_in,D_out)
    def forward(self, x):
        x = self.mylayer1(x)
        return x
```

3.6 PyTorch入门-例子

```
model = MyNet()
print(model)
```

运行结果为：

```
MyNet(
  (mylayer1): MyLayer()   # 这就是自己定义的一个层
)
```

（3）开始训练

```
# 创建输入、输出数据
x = torch.randn(N, D_in)  #（10，5）
y = torch.randn(N, D_out) #（10，3）

loss_fn = torch.nn.MSELoss(reduction='sum')   #定义损失函数
learning_rate = 1e-4    #定义学习率
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) #构造一个optimizer对象

 for t in range(10):
    # 第一步：数据的前向传播，计算预测值p_pred
    y_pred = model(x)
```

3.6 PyTorch入门-例子

```
# 第二步：计算计算预测值p_pred与真实值的误差
loss = loss_fn(y_pred, y)
print(f"第 {t} 个epoch, 损失是 {loss.item()}")

# 在反向传播之前，将模型的梯度归零
optimizer.zero_grad()

# 第三步：反向传播误差
loss.backward()

# 直接通过梯度一步到位，更新完整个网络的训练参数
optimizer.step()
```

3.6 PyTorch入门-例子

程序运行结果：

第 0 个epoch, 损失是 42.33689498901367

第 1 个epoch, 损失是 42.3133430480957

第 2 个epoch, 损失是 42.289825439453125

第 3 个epoch, 损失是 42.26634979248047

第 4 个epoch, 损失是 42.24290847778320

第 5 个epoch, 损失是 42.21950912475586

第 6 个epoch, 损失是 42.19615173339844

第 7 个epoch, 损失是 42.17283248901367

第 8 个epoch, 损失是 42.14955520629883

第 9 个epoch, 损失是 42.12631607055664

参考资料

1、参考博客： https://blog.csdn.net/qq_27825451/article/details/90705328

2、参考博客： https://blog.csdn.net/weixin_38664232/article/details/94662534

4、参考博客： https://blog.csdn.net/sinat_42239797/article/details/93916790

5、参考博客： https://blog.csdn.net/u014380165/article/details/79119664

3、参考书籍：深度学习框架PyTorch：入门与实践_陈云(著)