



智能计算系统

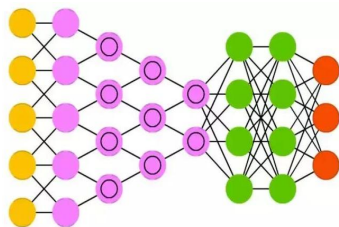
第四章 编程框架使用

中国科学院计算技术研究所

李威 副研究员

liwei2017@ict.ac.cn

Driving Example



编程框架



Bang

第四章将学习到实现深度学习算法所使用的编程框架的简单用法

提纲

- ▶ 编程框架概述
- ▶ PyTorch概述
- ▶ PyTorch编程模型及基本用法
- ▶ 基于PyTorch的模型推理实现
- ▶ 基于PyTorch的模型训练实现
- ▶ 驱动范例

为什么需要编程框架?



深度学习算法得到广泛关注，越来越多的公司、程序员需要使用深度学习算法

为什么需要编程框架?

计算误差对权重 w_{ij} 的偏导数是两次使用链式法则得到的:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

在右边的最后一项中, 只有加权和 net_j 取决于 w_{ij} , 因此

$$\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{k=1}^n w_{kj} o_k \right) = o_i.$$

神经元 j 的输出对其输入的导数就是激活函数的偏导数 (这里假定使用逻辑函数):

$$\frac{\partial o_j}{\partial \text{net}_j} = \frac{\partial}{\partial \text{net}_j} \varphi(\text{net}_j) = \varphi(\text{net}_j)(1 - \varphi(\text{net}_j))$$

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import axes3d
4 from matplotlib import style
5
6 def SGD(samples, y, step_size=2, max_iter_count=1000):
7     m, var = samples.shape
8     theta = np.zeros(2)
9     y = y.flatten()
10    #进入循环内
11    loss = 1
12    iter_count = 0
13    iter_list=[]
14    loss_list=[]
15    theta1=[]
16    theta2=[]
```

有必要将算法中的常用操作封装成组件提供给程序员, 以提高深度学习算法开发效率

但如果 j 是网络中任一内层, 求 E 关于 o_j 的导数就不太简单了。

考虑 E 为接受来自神经元 j 的输入的所有神经元 $L = u, v, \dots, w$ 的输入的函数,

$$\frac{\partial E(o_j)}{\partial o_j} = \frac{\partial E(\text{net}_u, \text{net}_v, \dots, \text{net}_w)}{\partial o_j}$$

并关于 o_j 取全微分, 可以得到该导数的一个递归表达式:

$$\frac{\partial E}{\partial o_j} = \sum_{l \in L} \left(\frac{\partial E}{\partial \text{net}_l} \frac{\partial \text{net}_l}{\partial o_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial \text{net}_l} \frac{\partial \text{net}_l}{\partial o_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial \text{net}_l} w_{jl} \right)$$

```
23 rand1 = np.random.randint(0,m,1)
24 h = np.dot(theta,samples[rand1].T)
25 #关键点, 只需要一个样本点来更新权值
26 for i in range(len(theta)):
27     theta[i] =theta[i] - step_size*(1/m)*(h - y[rand1])*samples[rand1,i]
28 #计算总体的损失精度, 等于各个样本损失精度之和
29 for i in range(m):
30     h = np.dot(theta.T, samples[i])
31     #每组样本点损失的精度
32     every_loss = (1/(var*m))*np.power((h - y[i]), 2)
33     loss = loss + every_loss
34
35 print("iter_count: ", iter_count, "the loss:", loss)
36
```

算法理论复杂

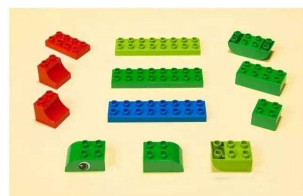
代码实现工作量大

为什么需要编程框架?



- ▶ 深度学习算法具有多层结构，每层的运算由一些基本操作构成
- ▶ 这些基本操作中存在大量共性运算，如卷积、池化、激活等。将这些共性运算操作封装起来，可以提高编程实现效率
- ▶ 面向这些封装起来的操作，硬件程序员可以基于硬件特征，有针对性的进行充分优化，使其能充分发挥硬件的效率

定义

- ▶ 随着深度学习研究的深入，深度学习算法变得愈加复杂，研究人员需要花更多的时间和精力在算法的实现上
- ▶ **深度学习编程框架**：将深度学习算法中的基本操作封装成一系列组件，这一系列深度学习组件，即构成一套深度学习框架
- ▶ 编程框架能够帮助算法开发人员更简单的实现已有算法，或设计新的算法。也有助于硬件程序员更有针对性的对关键操作进行优化，使其能充分发挥硬件效率



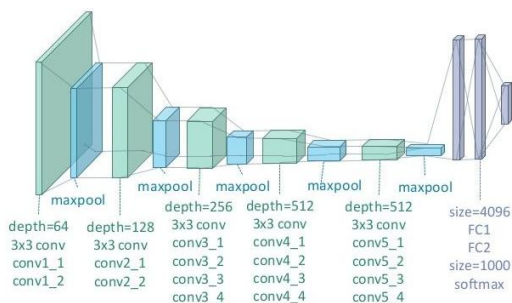
国内外主流编程框架

序号	框架名称	发布者	首次发布时间	logo
1	PyTorch	Facebook (Meta)	2017	
2	TensorFlow	Google	2015	
3	Keras	Google	2015	
4	Caffe	BVLC	2013	
5	PaddlePaddle	百度	2018	
6	Mindspore	华为	2019	

提纲

- ▶ 编程框架概述
- ▶ PyTorch概述
- ▶ PyTorch编程模型及基本用法
- ▶ 基于PyTorch的模型推理实现
- ▶ 基于PyTorch的模型训练实现
- ▶ 驱动范例

Driving example-VGG19



神经网络



在计算设备运行

```
1 def run_style_transfer(cnn, normalization_mean, normalization_std,
2                       content_img, style_img, input_img, num_steps=300,
3                       style_weight=1000000, content_weight=1):
4     model, style_losses = get_style_model_and_losses(cnn,
5                                                     normalization_mean, normalization_std, style_img, content_img)
6
7     input_img.requires_grad_(True)
8     model.requires_grad_(False)
9
10    optimizer = get_input_optimizer(input_img)
11    run = [0]
12    while run[0] <= num_steps:
13        def closure():
14            with torch.no_grad():
15                input_img.clamp_(0, 1)
16                optimizer.zero_grad()
17            model(input_img)
18            style_score = 0
19            content_score = 0
20            for s1 in style_losses:
21                style_score += s1.loss
22            for c1 in content_losses:
23                content_score += c1.loss
24            style_score *= style_weight
25            content_score *= content_weight
26            loss = style_score + content_score
27            loss.backward()
28            run[0] += 1
29            return style_score + content_score
30
31        optimizer.step(closure)
32
33    with torch.no_grad():
34        input_img.clamp_(0, 1)
35
36    return input_img
```

PyTorch实现

```
__dlp_entry__ void Proposal(...) {
...
__nram__ half scores[..];
__nramset__ half(scores, ..);
...
__bang_maxpool(..);
...
}
```

编程语言实现算子，
集成到编程框架中

PyTorch起源-Torch

- ▶ Torch是瑞士亚普研究所 (IDIAP) 在2002年发布的一款机器学习框架，采用LuaJIT脚本编程语言为接口，内核采用C/C++来实现
- ▶ 核心是易于使用的神经网络和优化库，同时在实现复杂的神经网络拓扑结构方面具有最大的灵活性
- ▶ 在GPU上具有较高的性能
- ▶ 作者：Ronan Collobert, Samy Bengio, Johnny Mariéthoz



PyTorch简介

- ▶ PyTorch=Py+Torch
- ▶ 2017年Facebook AI Research开源
- ▶ PyTorch是一个基于Torch的Python开源机器学习库，用于自然语言处理等应用程序，具有强大的GPU加速的张量计算（如Numpy）能力
- ▶ 产生背景：
 - ▶ 编程语言提供了面向深度学习的高效编程库（NumPy、Eigen、Torch等）
 - ▶ Python开源生态蓬勃发展

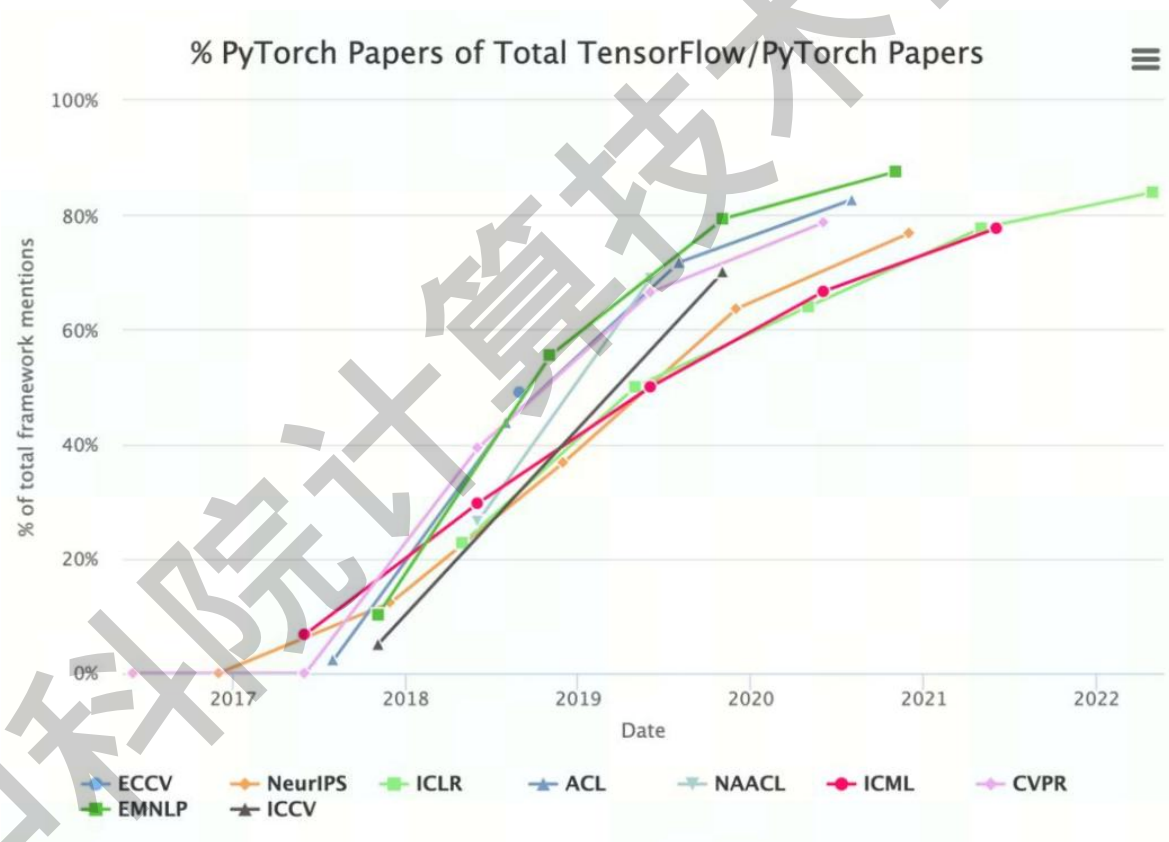
设计原则

- ▶ 基于Python
- ▶ 把研究人员放在首位
- ▶ 易用
- ▶ 高性能
- ▶ 内部实现简单，节省学习时间

Framework	Throughput (higher is better)					
	AlexNet	VGG-19	ResNet-50	MobileNet	GNMTv2	NCF
Chainer	778 ± 15	N/A	219 ± 1	N/A	N/A	N/A
CNTK	845 ± 8	84 ± 3	210 ± 1	N/A	N/A	N/A
MXNet	1554 ± 22	113 ± 1	218 ± 2	444 ± 2	N/A	N/A
PaddlePaddle	933 ± 123	112 ± 2	192 ± 4	557 ± 24	N/A	N/A
TensorFlow	1422 ± 27	66 ± 2	200 ± 1	216 ± 15	9631 ± 1.3%	4.8e6 ± 2.9%
PvTorch	1547 ± 316	119 ± 1	212 ± 2	463 ± 17	15512 ± 4.8%	5.4e6 ± 3.4%

Table 1: Training speed for 6 models using 32bit floats. Throughput is measured in images per second for the AlexNet, VGG-19, ResNet-50, and MobileNet models, in tokens per second for the GNMTv2 model, and in samples per second for the NCF model. The fastest speed for each model is shown in bold.

- ▶ EMNLP、ACL、ICLR三家顶会的PyTorch的占比已经超过80%，这一占比数字在其他会议中也都保持在70%之上



发展历程

时间	版本号	更新内容
2017.1	0.1	第一个发布版本
2017.8	0.2	支持高阶导数、分布式计算、张量广播等功能
2018.4	0.4	支持Windows操作系统，将张量和变量合并为张量
2018.11	1.0	对不同后端的分布式计算有了完善的支持，加强了对C++前端的支持；发布PyTorch Hub，为用户提供一系列预训练模型
2019.5	1.1	支持TensorBoard对张量的可视化
2019.10	1.3	增加了对移动端的处理，增加了对模型量化功能的支持
2022.11	1.13	BetterTransformer 的稳定版
2023.3	2.0	目前的最新版，增加了编译模式

学习资料

- ▶ 一些有趣的应用
 - ▶ <https://github.com/ritchieng/the-incredible-pytorch>
- ▶ 斯坦福课程关于Pytorch的介绍
 - ▶ <https://cs230.stanford.edu/blog/pytorch/>
- ▶ 官方github
 - ▶ <https://github.com/pytorch>

提纲

- ▶ 编程框架概述
- ▶ PyTorch概述
- ▶ PyTorch编程模型及基本用法
- ▶ 基于PyTorch的模型推理实现
- ▶ 基于PyTorch的模型训练实现
- ▶ 驱动范例

1、NumPy基础

- ▶ Python语言提供的高性能编程库，提供大量库函数和操作
- ▶ 提供针对高维数组的批量化处理
- ▶ 能够高效处理机器学习、计算机视觉、基于数组的数学任务
- ▶ NumPy中最重要的数组类为ndarray，也叫array

创建ndarray的方法

```
import numpy as np

my_data1 = np.array([1,2])

my_data2 = np.array([[1.0,2.0],[3.0,4.0],[5.0,6.0]], np.float32)

my_data3 = np.arange(3)           #[0,1,2]

my_data4 = np.zeros((3,2))       #创建3行2列，元素值全为0的数组

my_data5 = np.ones((3,2))        #创建3行2列，元素值全为1的数组

my_data6 = np.random.random((3,2)) #创建3行2列，元素值为0到1之间随机值的数组

my_data7 = np.full((3,2), 7)     #创建3行2列，元素值全为7的数组

my_data8 = np.eye(3,3)

my_data9 = np.empty((3,2))       #创建3行2列，元素值为随机值的数组
```

▶ np.array (object, dtype)

▶ object: 数组

▶ dtype: 可选, 数组元素的数据类型

```
my_data1 = np.array([0,1,2])
```

```
my_data2 = np.array([[0,1], [2,3],[4,5]])
```

```
my_data3 = np.array([[[0,0,0,0],  
                      [1,1,1,1],  
                      [2,2,2,2]],  
                    [[3,3,3,3],  
                     [4,4,4,4],  
                     [5,5,5,5]])
```

0	1	2
---	---	---

my_data1

0	1
2	3
4	5

my_data2

3	3	3	3
4	4	4	4
5	5	5	5

0	0	0	0
1	1	1	1
2	2	2	2

my_data3

▶ np.arange (start,stop,step,dtype)

- ▶ 返回一个具有n个元素的行向量，向量中的元素值为 [start,(start+step),(start+2*step),..., (start+(n-1)*step)]
- ▶ start: 可选，表示起始数值，默认为0
- ▶ stop: 终止数值 (不含)
- ▶ step: 可选，表示步长，默认为1，如果有step则必须给出 start

```
my_data1 = np.arange(3,dtype = np.int32)    #输出[0,1,2]
```

```
my_data2 = np.arange(3,9,3)                #输出[3,6]
```

```
my_data3 = np.arange(3,10,3)               #输出[3,6,9]
```

▶ `np.eye(N,M,k)`

- ▶ 返回一个N*M的二维数组，对角线元素为1，其余元素均为0
- ▶ M: 可选，表示数组的列数，没有就默认为N
- ▶ k: 可选，为0表示主对角线，为负表示对角线左移，为正表示对角线右移

```
my_data1 = np.eye(3)
```

```
my_data2 = np.eye(3,2)
```

```
my_data3 = np.eye(3,5,k=1)
```

```
[[1,0,0]  
 [0,1,0]  
 [0,0,1]]
```

my_data1

```
[[1,0]  
 [0,1]  
 [0,0]]
```

my_data2

```
[[0,1,0,0,0]  
 [0,0,1,0,0]  
 [0,0,0,1,0]]
```

my_data3

▶ `np.eye(num_class)[label_array]`

- ▶ 在深度学习算法处理中，对标签数组`label_array`进行one-hot编码，返回编码后得到的`ndarray`
- ▶ `num_class`：标签类别数量

```
my_data4 = np.eye(3)[[0,1,0,2]]
```

```
[[1,0,0]
```

```
[0,1,0]
```

```
[1,0,0]
```

```
[0,0,1]]
```

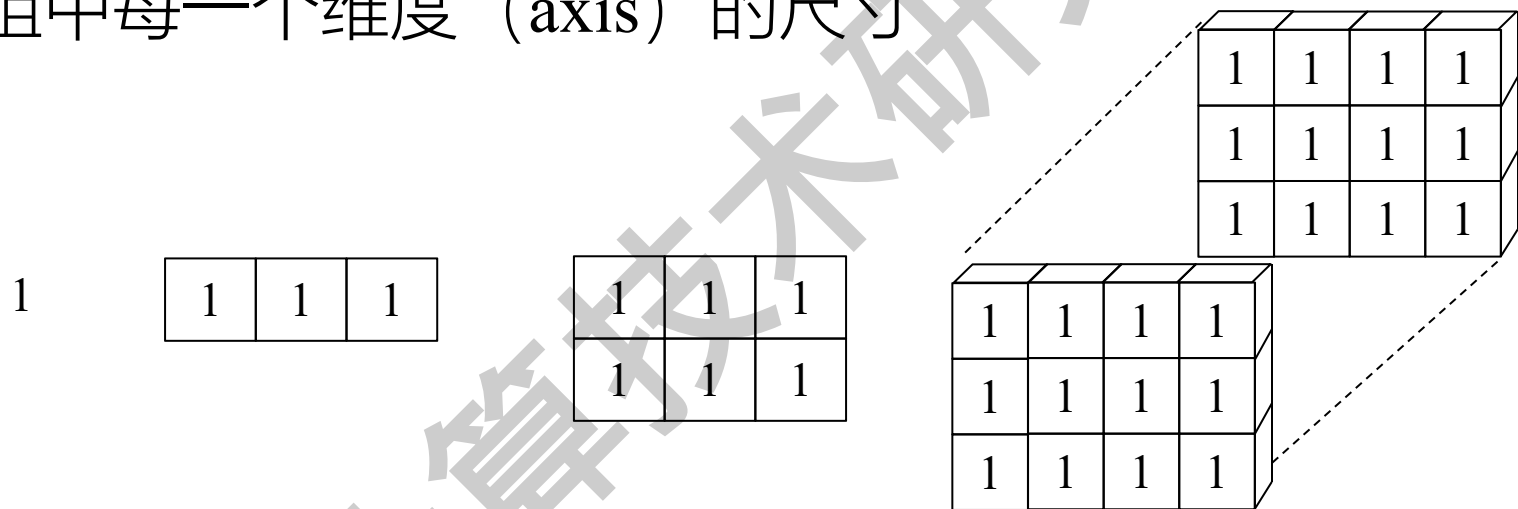
```
my_data4
```

ndarray的属性

序号	属性	说明
1	<code>ndarray.ndim</code>	数组维度 (axis)
2	<code>ndarray.shape</code>	数组在每个维度上的尺寸
3	<code>ndarray.size</code>	数组中的元素个数, 等于shape中各元素的乘积
4	<code>ndarray.dtype</code>	数组元素的数据类型(默认为float64)
5	<code>ndarray.itemsize</code>	数组中每个元素的字节数

ndarray的形状属性 (shape)

- 表示数组中每一个维度 (axis) 的尺寸



ndim	0	1	2	3
对应数据形式	标量	向量	矩阵	三维数组
shape	()	(3)	(2,3)	(2,3,4)

形状操作

- ▶ `np.reshape(a,newshape)`
 - ▶ 将原始数组按照`newshape`重塑并返回
 - ▶ 数组元素数量不变，数组元素数据不变

```
my_data1 = np.arange(4).reshape((2,2))
```

```
my_data2 = np.reshape(my_data1,(4,1))
```

```
[[0,1]  
 [2,3]]
```

my_data1

```
[[0]  
 [1]  
 [2]  
 [3]]
```

my_data2

▶ `np.resize(a,new_shape)`

▶ 按照`new_shape`对数组`a`重塑，如果`new_shape`中元素数量大于原数组，则从原数组第一个元素开始复制数据补充到新数组中。

▶ `a.resize(new_shape)`是对多出的数组元素位置补0

```
my_data1 = np.arange(4)
```

```
my_data2 = np.resize(my_data1,(3,2))
```

```
my_data1.resize(3,2)
```

```
[[0,1]
```

```
[2,3]
```

```
[0,1]]
```

```
[[0,1]
```

```
[2,3]
```

```
[0,0]]
```

索引

- 索引序号从0开始，若为负则表示从数组末尾开始向前计数索引

```
my_data1 = np.arange(8)
my_data2 = my_data1[np.array([2,-2])]
my_data3 = my_data1.reshape((4,2))
my_data4 = my_data3[2,1]
```

[0,1,2,3,4,5,6,7]

my_data1

[2,6]

my_data2

[[0,1]

[2,3]

[4,5]

[6,7]]

my_data3

5

my_data4

数学函数

序号	函数	描述
1	sin cos tan	正弦函数 余弦函数 正切函数
2	arcsin arccos arctan	反正弦函数 反余弦函数 反正切函数
3	round floor ceil	四舍五入 向下取整 向上取整
4	sum diff	指定轴上元素和 指定轴上相邻元素的差
5	exp log	指数函数 对数函数

序号	函数	描述
6	add subtract multiply divide	加法函数 减法函数 乘法函数 除法函数
7	real imag	取复数的实部 取复数的虚部
8	sqrt square	平方根函数 平方函数
9	maximum minimum	取最大值函数 取最小值函数
10	clip	将数组元素值截取到一定范围内

NumPy特点总结

▶ 优点

- ▶ 易用：直接对数组进行操作，并提供多种常用内嵌API供用户直接调用
- ▶ 高性能：大部分NumPy代码是基于C语言实现的，相比Python代码有更高的实现性能

▶ 局限性

- ▶ 原生的NumPy只能在CPU上实现
- ▶ 为了在GPU上高效运行Python计算库，出现了CuPy、Numba、PyCUDA、PyTorch等

2、张量 (tensor)

- ▶ **张量**是计算图上的数据载体，用张量统一表示所有的数据，张量在计算图的节点之间传递
- ▶ 张量对应了神经网络中在各个节点之间传递、流动的数据
- ▶ 张量可以看做是 n 维的数组，数组的维数即为张量的**阶数**

阶数	对应数据形式
0	标量
1	向量
2	矩阵
n	n 维数组

- ▶ 与NumPy中的ndarray不同，PyTorch中的张量可以运行在GPU或深度学习处理器上，因此具有较高的性能

tensor的创建

▶ 直接创建

```
my_data1 = torch.tensor([0.0,1.0,2.0,3.0],requires_grad = True) #使用requires_grad参数来表示  
#该张量是否需要计算梯度  
my_data2 = torch.ones(3,2) #创建3行2列，元素值全为1的张量  
my_data3 = torch.zeros(3,2) #创建3行2列，元素值全为0的张量  
my_data4 = torch.rand(3,2) #创建3行2列的张量，元素值为[0,1)区间随机值
```

▶ 从其他张量创建

```
my_data5 = torch.rand_like(my_data1) #创建与my_data1有相同shape的张量，元素值为[0,1)区间随机值  
my_data6 = torch.zeros_like(my_data1) #创建与my_data1的shape相同，元素值全为0的张量  
my_data7 = torch.ones_like(my_data1) #创建与my_data1的shape相同，元素值全为1的张量
```

Tensor与NumPy数组的互相转换

- ▶ 从NumPy生成tensor

```
my_data1 = np.array([0,1,2,3])  
my_data2 = torch.from_numpy(my_data1)
```

- ▶ 从tensor到NumPy

```
my_data3 = torch.zeros(2,2)  
my_data4 = my_data3.numpy()
```

- ▶ 使用from_numpy()和numpy()函数产生的tensor和NumPy数组共享内存，对其中一个的更改也会使另一个随着改变

用于创建张量的常用操作 (torch.)

操作	含义
from_numpy zeros/ones eye	将NumPy数组转换为张量 创建元素值全为0/1的张量 创建对角元素为1, 其余元素为0的张量 (用法同NumPy)
cat/concat split stack take	连接多个张量 切分张量 沿新维度连接多个张量 从输入张量中取出指定元素组成新张量
normal rand randn	返回由正态分布随机数组成的张量 返回[0,1)区间均匀分布随机数组成的张量 返回由标准正态分布随机数组成的张量

张量的常用属性

属性名	含义
<code>dtype</code>	<code>tensor</code> 存储的数据类型
<code>shape</code>	<code>tensor</code> 各阶的长度（同NumPy中 <code>ndarray</code> 的 <code>shape</code> 属性）
<code>device</code>	存储 <code>tensor</code> 的设备对象
<code>grad</code>	默认为None，当调用 <code>backward()</code> 进行反向传播后为该 <code>tensor</code> 的梯度值

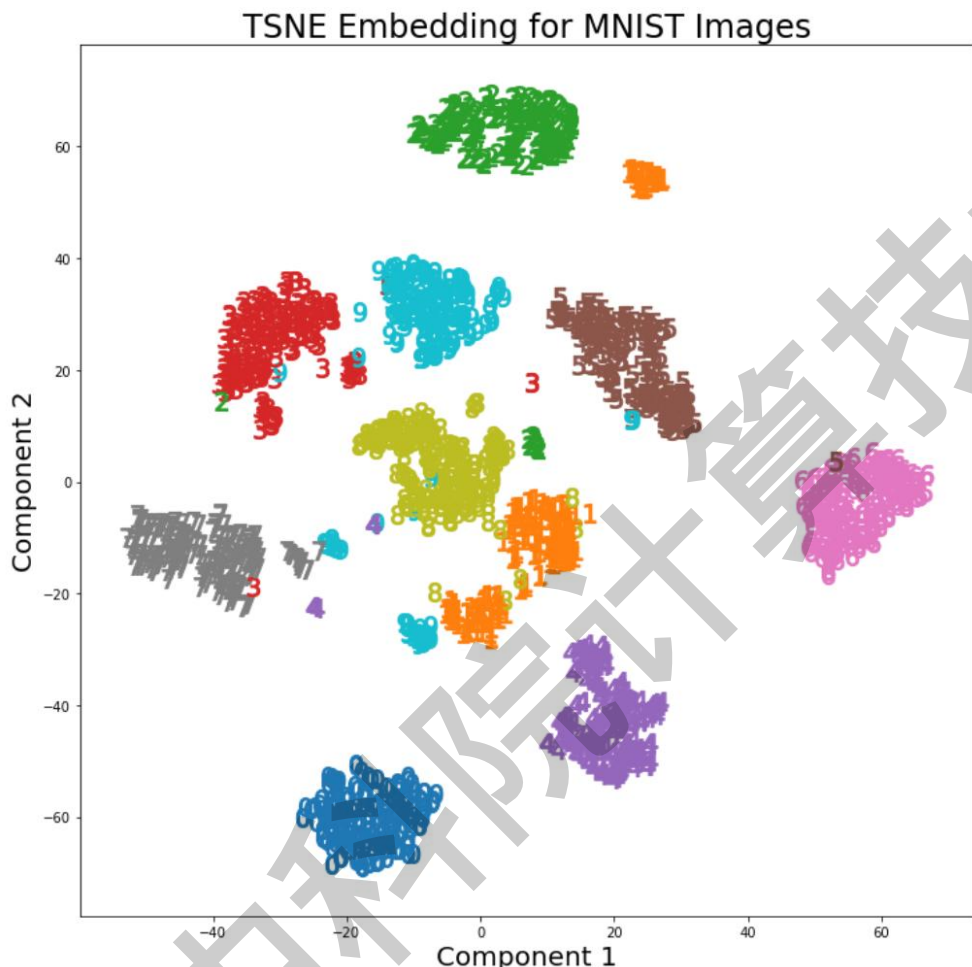
张量的数据类型 (dtype)

PyTorch数据类型	说明
torch.float16/torch.half torch.float32/torch.float torch.float64/torch.double	16位浮点 32位浮点 64位浮点
torch.uint8	8位无符号整型
torch.int8 torch.int16/torch.short torch.int32/torch.int torch.int64/torch.long	8位整型 16位整型 32位整型 64位整型
torch.bool	布尔型
torch.bfloat16	Brain Float16

▶ 示例：`my_data1=torch.zeros([2,2], dtype=torch.int8)`

深度学习为什么不需要全部float32

深度学习算法特性

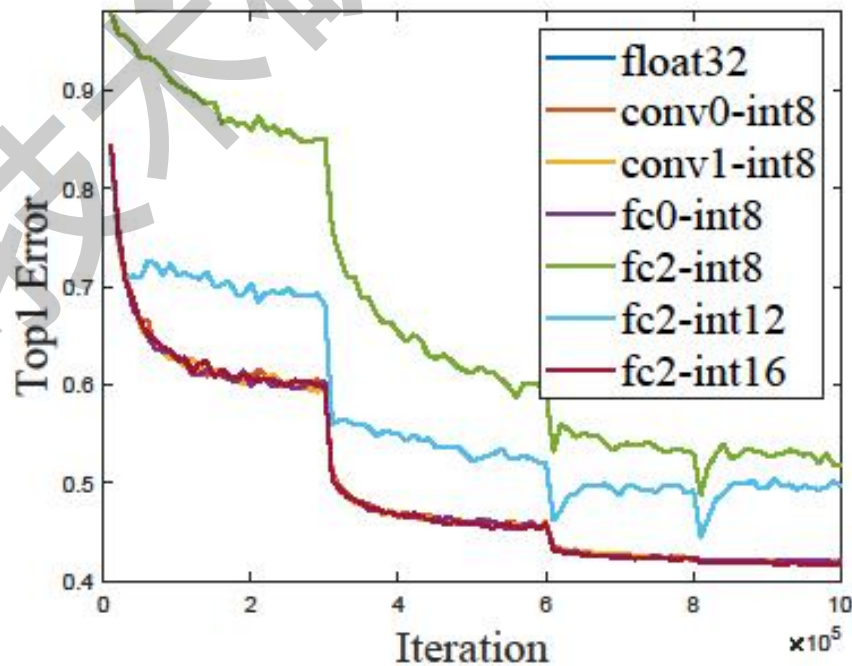
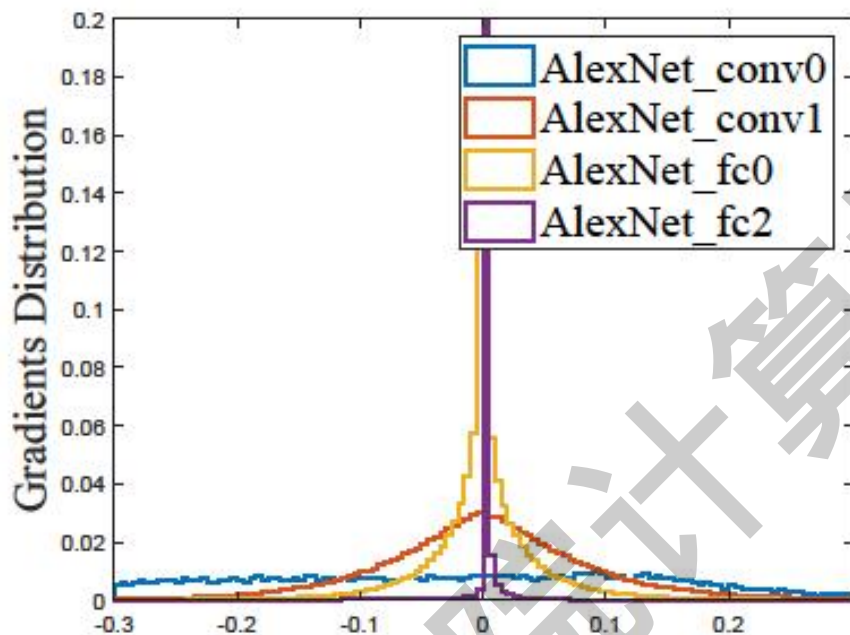


t-SNE可视化深度学习数据

- 这是一个典型的手写体识别任务的网络特征（神经元）二维可视化图
- 不同类别（颜色）的数据间距
- “大间距”意味着容忍非精确计算
- 低位宽从原理上看对于深度学习任务是可行的

数据位宽与算法精度

- 不同数据对位宽的需求是不同的



每层数据都有其保持网络收敛的最低位宽需求，
每层数据的位宽需求与数据分布之间存在关系

训练时不需要高位宽

- ▶ CNN网络，分类、检测、分割任务下的低位宽训练
 - ▶ 神经元和权值自适应8bit，梯度8-16bit——精度无损

Classification Network	float32 Acc	Adaptive Acc	Activation int8	Weight int8	Activation int8	Gradient int16
AlexNet	58.0	58.22	100%	100%	22.5%	77.5%
VGG16	71.0	70.6	100%	100%	31.3%	68.7%
Inception_BN	73.0	72.8	100%	100%	4.5%	95.5%
ResNet50	76.4	76.2	100%	100%	0.8%	99.2%
ResNet152	78.8	78.2	100%	100%	1.7%	98.3%
MobileNet v2	72.0	70.5	100%	100%	0.7%	99.2%
SSD Detection Network	float32 mAP	Adaptive mAP	Activation int8	Weight int8	Activation int8	Gradient int16
COCO_VGG	43.1	42.4	100%	100%	31.4%	68.6%
VOC_VGG	77.3	77.2	100%	100%	34.3%	65.7%
VOC_ResNet101	73.4	73.1	100%	100%	7.4%	83.6%
IMGDET_ResNet101	44.1	44.4	100%	100%	28.6%	71.4%
Segmentation Network	float32 meanIoU	Adaptive meanIoU	Activation int8	Weight int8	Activation int8	Gradient int16
deeplab-v1	70.1	69.9	100%	100%	1.0%	99.0%

张量的device属性

- ▶ torch.device指定tensor所在的设备名、设备序号
- ▶ 用'cuda:n'表示第n个GPU设备，用'dlp:n'表示第n个深度学习处理器

```
#在GPU上创建一个张量
my_data1 = torch.tensor([0,1,2,3],device=torch.device('cuda:1'))

my_data2 = torch.tensor([0,1,2,3],device=torch.device('cuda',1))

my_data3 = torch.tensor([0,1,2,3],device='cuda:1')

#设置默认的device类型
my_device = 'cuda' if torch.cuda.is_available() else 'cpu'
my_data4 = torch.tensor([0,1,2,3],device=my_device)
```

张量属性的转换

- ▶ `tensor.to()`: 进行张量的数据类型或设备类型转换

```
#张量设备类型转换
my_data1 = torch.tensor([0,1,2,3])
my_data1 = my_data1.to('cuda')

#张量数据类型转换
my_data1 = my_data1.to(torch.double)
```

- ▶ 将GPU上的张量转换成NumPy数组，需要先将张量转换到CPU上，再转换成NumPy

```
my_data2 = torch.tensor([0,1,2,3],device = 'cuda:1')
my_data3 = my_data2.cpu().numpy()
```

张量属性的转换

- ▶ `tensor.reshape(*shape)`: 进行张量的形状属性转换

```
my_data1 = torch.tensor([0,1,2,3])  
my_data2 = my_data1.reshape(2,2)
```

- ▶ 单个维度上的shape值可以为-1, 表示该维度上的shape值需要根据其他维度的shape来推算

```
my_data3 = torch.arange(6)  
my_data4 = my_data3.reshape(-1,2)           #形状转换为 (3,2)
```

张量属性的转换

张量从CPU转换到DLP上

```
import torch
import torch_dlp

a = torch.randn(8192,8192, dtype = torch.float)
b = torch.randn(8192,8192, dtype = torch.float)

#在CPU上计算
c = torch.matmul(a,b)

#在DLP上计算
device = torch.device('dlp')
a_dlp = a.to(device)
b_dlp = b.to(device)
c_dlp = torch.matmul(a_dlp, b_dlp)
```

转换到DLP的两种方式:

- ▶ a.dlp()
- ▶ a.to(torch.device('dlp'))

张量的复制

- ▶ `tensor.clone()`: 对张量进行复制, 返回一个完全相同的 `tensor`, 新张量会保存在新的内存中, 且仍然留在计算图中

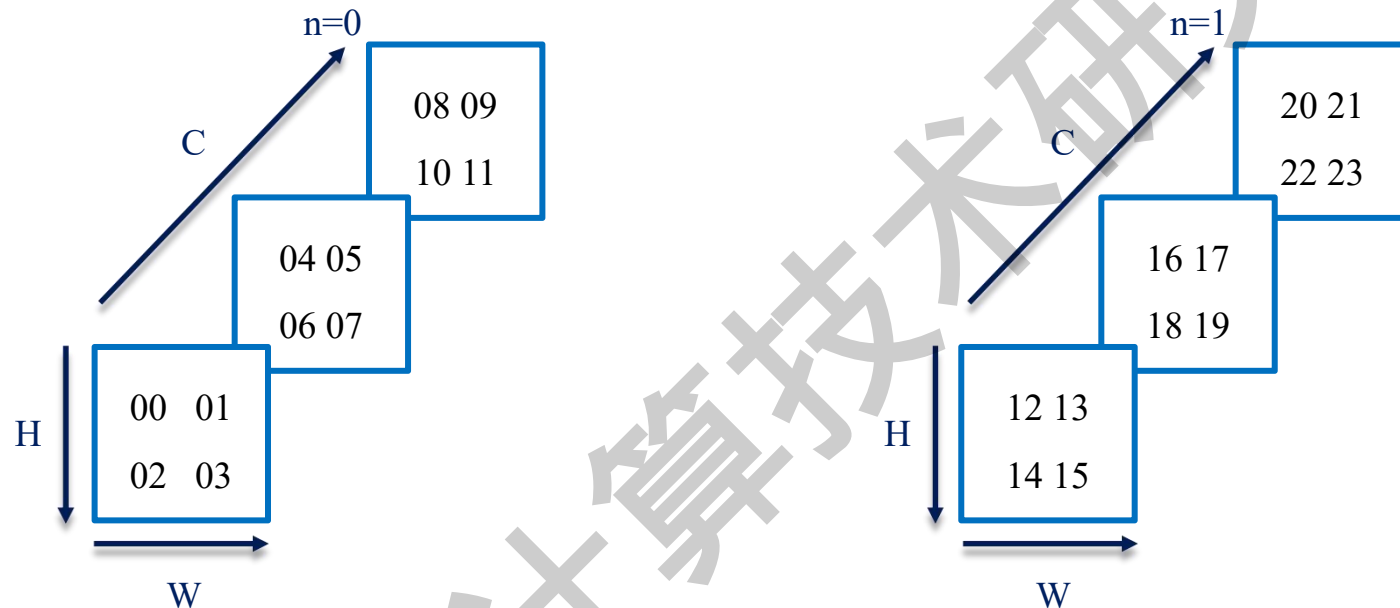
```
my_data1 = torch.tensor([0,1,2,3])  
my_data2 = my_data1.clone()
```

张量的数据格式 (data layout)

- ▶ 张量数据可以有多种数据格式，代表了多维数组以何种线性存储方式在存储空间中存储
- ▶ PyTorch、GPU中采用NCHW，TensorFlow、CPU中采用NHWC
 - ▶ N: 一批次的数据个数 (batch size)
 - ▶ C: 通道数 (channel)
 - ▶ H: 高度 (height)
 - ▶ W: 宽度 (width)
- ▶ 一张RGB彩色图像包含3个原色 (红色、绿色、蓝色) 通道，对应的张量表示中， $N=1$ ， $C=3$
- ▶ 用于训练或推理任务的多张RGB图像， $N=\text{batch size}$ ， $C=3$

张量的数据格式 (data layout)

- 以 $N=2$, $C=3$, $H=2$, $W=2$ 的数据为例



- 数据在计算设备中按照1维来存储
 - NCHW: 按照 $W \rightarrow H \rightarrow C \rightarrow N$ 的顺序存储

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- NHWC: 按照 $C \rightarrow W \rightarrow H \rightarrow N$ 的顺序存储

00	04	08	01	05	09	02	06	10	03	07	11	12	16	20	13	17	21	14	18	22	15	19	23
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

张量的索引

- PyTorch的索引方法基本与NumPy相同

```
my_data1 = torch.tensor([0,1,2,3])
```

```
my_data1[1] = 7
```

```
my_data1[-2] = 8      #负号表示从后往前查找
```

```
my_data1=[0,7,8,3]
```

```
my_data2 = torch.arange(0,6).reshape([2,3])
```

```
print(my_data2[1,1])      #(4)
```

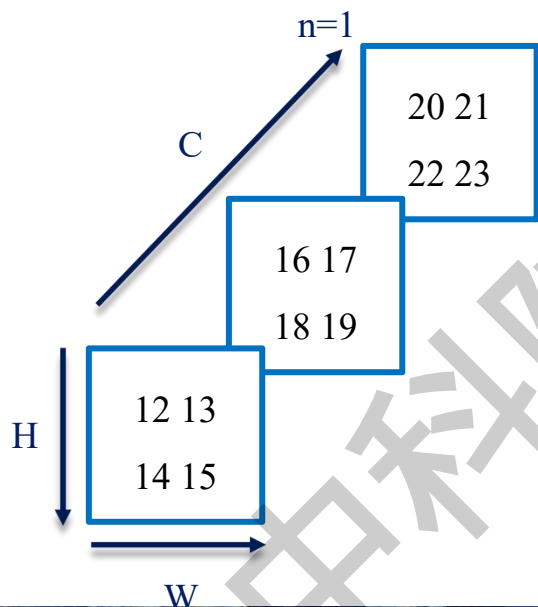
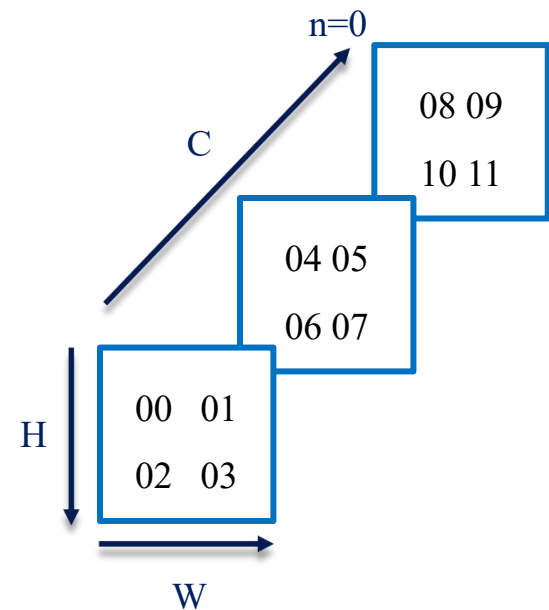
```
print(my_data2[0])       #([0,1,2])
```

```
print(my_data2[:,0])     #([0,3])
```

张量的切片

- ▶ 用[start:end:step]形式来表示切片，其含义为：从start开始，以step为步长开始读取张量，到end终止（不包含end）
- ▶ start、end、step均可以缺省，start缺省为0，end缺省为该维度最后一个元素，step缺省为1
- ▶ start可以为负数，step不能为负数

张量的切片索引



```
import torch
```

```
# my_data1包含2张图片，每张图片为3通道，每通道为2*2
```

```
my_data1 = torch.arange(0,24).view([2,3,2,2])
```

```
#读取其中一个像素点
```

```
print(my_data1[1,1,1,1]) # (19)
```

```
#读取两张图片中通道数为0、1，高度方向序号为1的像素
```

```
print(my_data1[:,0:2,1,:])
```

```
#读取第一张图片，通道为1、2，高度方向序号为0、1，宽
```

```
#度方向序号为0的像素
```

```
print(my_data1[0,-2:,0:,0])
```

张量的维度压缩、扩展

- ▶ `torch.squeeze(input)`: 将张量中所有值为1的维度移除

```
my_data1 = torch.zeros(3,1,2,1,5)
```

```
my_data2 = torch.squeeze(my_data1) #形状变为 (3,2,5)
```

```
my_data3 = torch.squeeze(my_data1,0) #形状变为 (3,1,2,1,5)
```

```
my_data4 = torch.squeeze(my_data1,1) #形状变为 (3,2,1,5)
```

- ▶ `torch.unsqueeze(input,dim)`: 在第dim维插入值为1的维度

```
my_data1 = torch.tensor([1,2,3,4])
```

```
my_data2 = torch.unsqueeze(my_data1,0) #得到新张量 ([[1,2,3,4]])
```

```
my_data3 = torch.unsqueeze(my_data1,1) #得到新张量([ [1,  
# [2],  
# [3],  
# [4]])
```

张量的自动求导支持

- ▶ PyTorch支持自动求导，用户定义好操作的前向计算和反向梯度计算规则，PyTorch能够在训练时自动调用计算图算子，完成整个网络的自动求导
- ▶ 使用requires_grad参数来设置张量是否需要自动求导，默认为false
- ▶ 对于一个计算操作来说，如果所有输入中有一个输入需要求导，则输出就要求求导；如果所有输入都不需要求导，则输出也不需要求导

```
my_data1 = torch.tensor([0.0,1.0,2.0,3.0],requires_grad = True) #使用requires_grad参数来表示  
#该张量是否需要自动求导，默  
#认为False
```

3、操作(operation)

- ▶ PyTorch基于张量开展各种类型的计算操作。每个操作接收若干个张量作为输入，操作完成后更新原张量或生成新张量作为输出
- ▶ 计算操作是使用PyTorch实现模型训练和推理的基础

```
import torch  
my_data1 = torch.tensor([-1.3,2.8,3.5,-4.2,-5.6,6.99])  
my_data2 = torch.mul(my_data1,100)
```

定义计算操作的方法

- ▶ 可以采用 `torch.operation`、`tensor.operation`、`tensor.operation_` 等形式来实现张量的计算操作

```
my_data1 = torch.tensor([-1.3,2.8,3.5,-4.2,-5.6,6.99])
```

```
my_data2 = torch.add(my_data1, 100)
```

```
my_data3 = my_data1.add(100)
```

```
my_data4 = my_data1.add_(100)
```

原位 (in-place) 操作

- ▶ 指在存储原张量的内存上直接计算更新张量值，而不是先复制张量再计算更新。其标志是在原操作语句后添加“_”
- ▶ 与Python语言中的+=、*=类似

计算操作	对应的原位操作	计算操作	对应的原位操作
tensor.add	tensor.add_	tensor.clamp	tensor.clamp_
tensor.abs	tensor.abs_	tensor.clip	tensor.clip_
tensor.addcmul	tensor.addcmul_	tensor.eq	tensor.eq_
tensor.arcsin	tensor.arcsin_	tensor.exp	tensor.exp_
tensor.bitwise_not	tensor.bitwise_not_	tensor.logical_or	tensor.logical_or_
tensor.bitwise_xor	tensor.bitwise_xor_	tensor.norm	tensor.norm_
tensor.ceil	tensor.ceil_	tensor.sigmoid	tensor.sigmoid_

原位（in-place）操作

- ▶ 原位操作能够节省内存占用，在进行深度学习算法推理时，使用原位操作能够有效减少模型占用的内存
- ▶ 原位操作会覆盖原张量，如果在模型训练时使用原位操作来更新张量梯度，则每次迭代计算所得梯度值将被覆盖，从而破坏模型的训练过程
- ▶ 对于多个张量同时引用一个张量的情况，对该张量进行原位操作会影响其他张量的操作

操作的广播 (broadcasting) 机制

```
my_data1 = torch.tensor([-1.3,2.8,3.5,-4.2,-5.6,6.99])  
my_data2 = torch.add(my_data1, 100)
```

Shape不同的张量仍然可以计算

- ▶ 对于参与计算操作的多个张量，如果张量维度不匹配，可以使用PyTorch的广播机制对不匹配的张量维度进行扩展，最终将这些张量均扩展为维度相同
- ▶ 能够进行广播机制的条件：
 - ▶ 每个张量都有至少1个维度
 - ▶ 从张量末尾的维度开始对齐扩展，在对齐后的同一维度中，仅下列情况之一才允许进行广播操作：1) 维度尺寸相同；2) 维度尺寸不同但其中一个维度尺寸为1；3) 其中一个张量没有该维度

```
my_data1 = torch.ones(2,3)
my_data2 = torch.ones(2,2)
my_data3 = torch.add(my_data1, my_data2)
```

不可广播

```
my_data1 = torch.ones(2,3,3)
my_data2 = torch.tensor([[[1],[2],[3]],
                          [[4],[5],[6]])]
my_data3 = torch.add(my_data1, my_data2)
```

可以广播

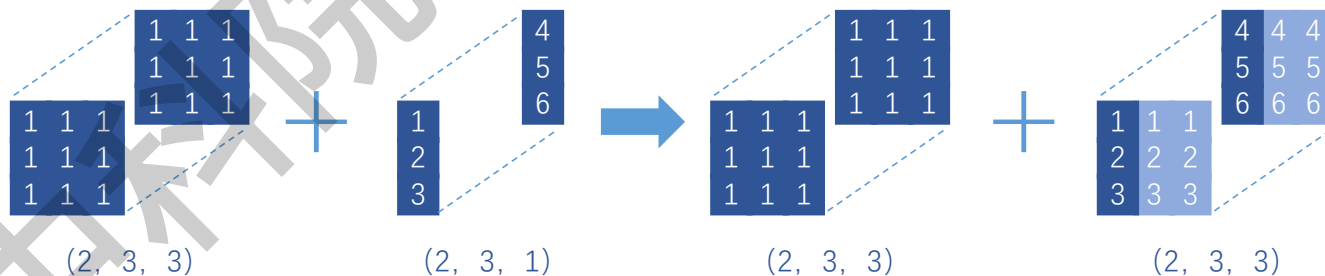
```
my_data1 = torch.ones(2,3,3)
my_data2 = torch.tensor([[[1],[2],[3]]])
my_data3 = torch.add(my_data1, my_data2)
```

可以广播

```
my_data1 = torch.ones(2,3,3)
my_data2 = torch.tensor([[[1],[2],[3]],
                          [[4],[5],[6]])
my_data3 = torch.add(my_data1, my_data2)
```

可以广播

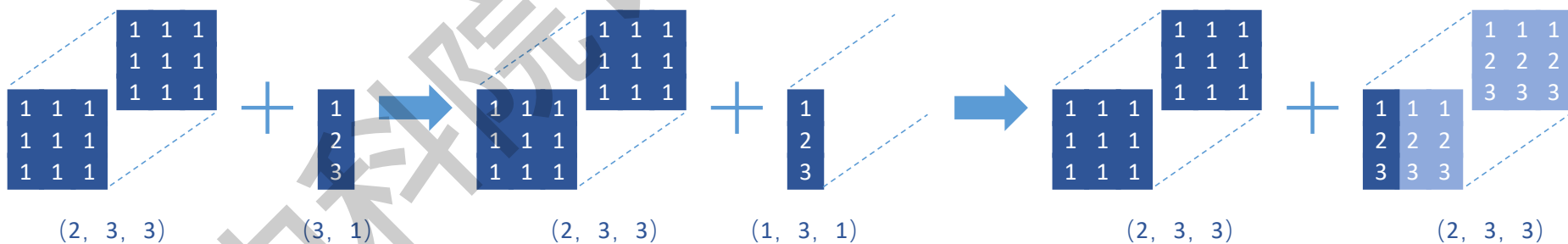
- 对于维度数量相同的张量，比较每个维度对应的维度尺寸，若维度尺寸不同但其中一个维度尺寸为1，则将其维度尺寸扩展为另一张量的维度尺寸



```
my_data1 = torch.ones(2,3,3)
my_data2 = torch.tensor([[1],[2],[3]])
my_data3 = torch.add(my_data1, my_data2)
```

可以广播

- 对于维度数量不同的张量，首先从张量末尾的维度开始对齐扩展，对缺少的维度尺寸补1，再沿每个维度方向进行尺寸对比及扩展



原位操作的广播

- ▶ 如果是执行原位操作的张量需要维度扩展或改变，则编译报错
- ▶ 如果作为原位操作参数的张量需要维度扩展或改变，则仍可通过广播机制完成张量操作

```
my_data1 = torch.ones(2,3,1)
my_data2 = torch.ones(2,3,3)
my_data3 = my_data1.add_(my_data2)
```

编译报错

```
my_data1 = torch.ones(2,3,3)
my_data2 = torch.ones(2,1,1)
my_data3 = my_data1.add_(my_data2)
```

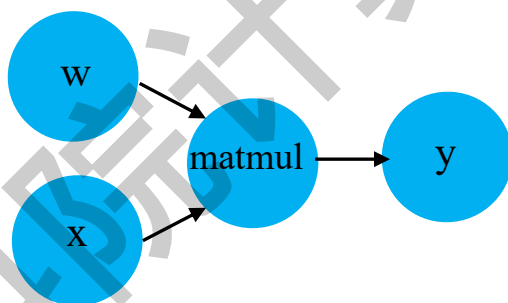
可以广播

常用计算操作 (torch./tensor.)

函数名称	功能
new_tensor	返回一个新张量
new_zeros/new_ones	返回尺寸与原张量相同, 元素值全为0/1的新张量
grad	训练时得到的张量梯度
add/subtract/multiply/divide	加/减/乘/除计算
bitwise_and/bitwise_or/bitwise_not	按位与/或/非操作
sin/cos/tanh	正弦/余弦/正切计算
where(condition,x,y)	按条件输出不同张量
to	张量数据类型、设备类型转换
sort	按指定维度将张量元素升序/降序排列
round/ceil/floor	四舍五入/向上取整/向下取整操作
transpose	转置计算

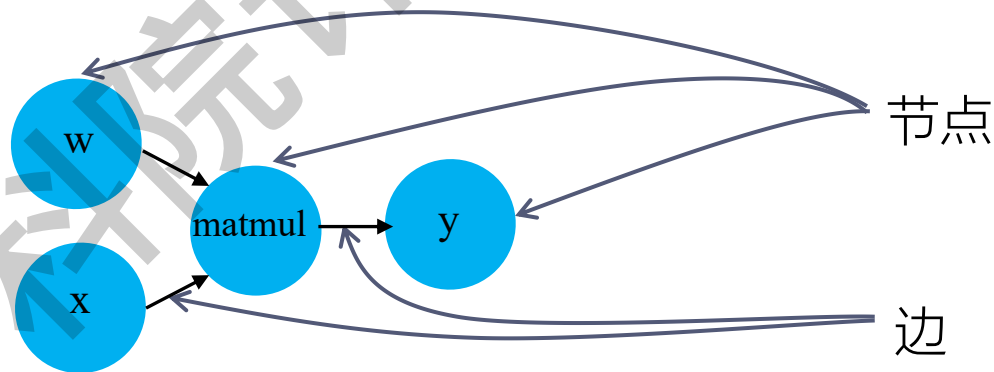
4、计算图

- ▶ 编程框架中使用有向图来描述计算过程。有向图中包含一组节点和边
- ▶ 支持通过多种高级语言来构建计算图 (C++/Python)
- ▶ 计算图对应了神经网络的结构
- ▶ 示例: $y=w*x$



节点和边

- ▶ **节点**一般用来表示各类操作，包括数学运算、变量读写、数据填充等，也可以表示输入数据、模型参数、输出数据
- ▶ **边**表示“节点”之间的输入输出关系。分为两类：
 - ▶ 一类是**传递具体数据**的边。传递的数据即为张量（tensor）。
 - ▶ 一类是**表示节点之间控制依赖关系**的边。这类边不传递数据，只表示节点执行的顺序：必须前序节点计算完成，后序节点才开始计算。



静态图 vs. 动态图

▶ 静态图

- ▶ 先定义整张图，再运行
- ▶ 可以对图进行全局优化，获得更快的运算速度
- ▶ 调试不方便

▶ 动态图

- ▶ 即时运行，网络模型可在运行时修改
- ▶ 代码编写灵活，可立即获得执行结果，调试方便
- ▶ 优化不方便

静态图 vs. 动态图

TensorFlow1.x: 静态图

```
x = tf.placeholder(tf.float32,shape=(1,2))
y = tf.placeholder(tf.float32)
w = tf.variable(tf.random_normal((2,1)))

y_pred = tf.matmul(x,w)
loss= y_pred-y
grad_w = tf.gradients(loss,w)
update_w = w.assign(w-alpha*grad_w)
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(300):
        sess.run([loss,update_w],
                 feed_dict={x:valuex,y:valuey})
```

- 1、构建静态图
- 2、每一次iteration中重复执行同样的图

PyTorch: 动态图

```
x = torch.randn(1,2)
y = torch.randn(1,1)
w = torch.randn(2,1,requires_grad=True)
```

```
for i in range(300):
    y_pred=x.mm(w)
    loss = y_pred-y
    loss.backward()
```

```
#loss.backward() #再次执行将报错
```

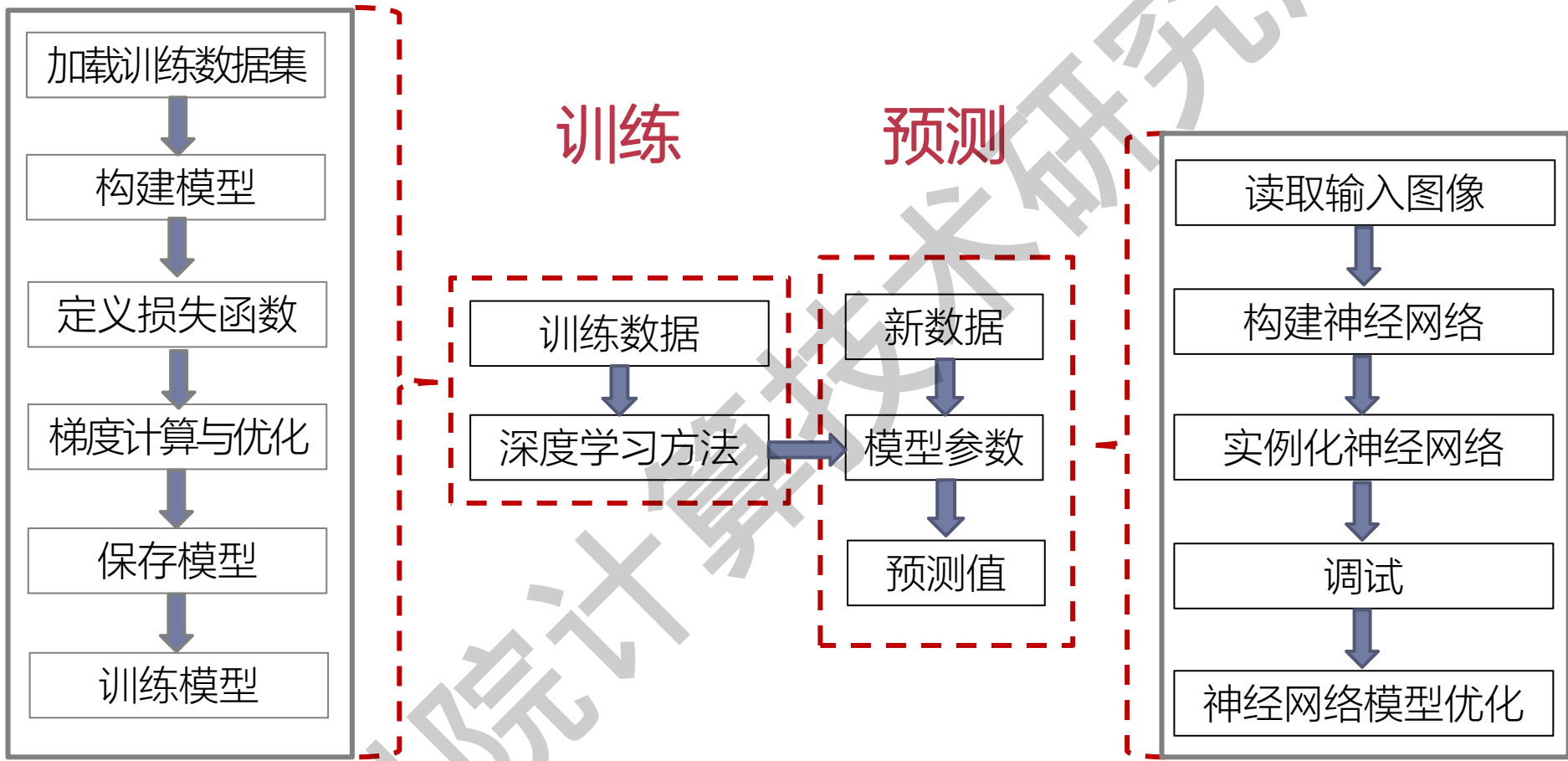
- 1、每一次iteration中构建并执行新图
- 2、在反向传播结束之后，整个计算图就在内存中被释放了

现有编程框架中采用的图模式

	静态图	动态图
PyTorch	✓	✓
TensorFlow	✓	✓
Caffe2	✓	
PaddlePaddle	✓	✓
MindSpore	✓	✓

提纲

- ▶ 编程框架概述
- ▶ PyTorch概述
- ▶ PyTorch编程模型及基本用法
- ▶ 基于PyTorch的模型推理实现
- ▶ 基于PyTorch的模型训练实现
- ▶ 驱动范例



1、读取输入图像

- ▶ 可以利用**PIL**、**OpenCV**、**torchvision.io**等来读取输入图像
- ▶ **PIL** (Python Imaging Library) : Python自带的图像处理库
 - ▶ 支持图像存储, 显示和处理, 能够处理几乎所有的图片存储模式, 包括28个与图片处理相关的模块或类
 - ▶ 读入的数据格式为**PIL.JpegImagePlugin.JpegImageFile**
 - ▶ 读入的图像格式为 (W, H, C)

PIL库中的Image类

操作	说明
<code>open(filename)</code>	加载图像文件
<code>mode</code>	图像的色彩模式，包括L（灰度图像）、RGB(彩色图像)等
<code>size</code>	二元组，表示图像的宽度和高度，单位是像素
<code>save(filename, format)</code>	保存图像
<code>convert(mode)</code>	将图像转换为新的模式
<code>resize(size)</code>	将图像大小调整为size

```
from PIL import Image
```

```
image = Image.open(image_name)
```

PIL与张量的格式转换

- ▶ 使用torchvision包中自带的transforms模块完成PIL与张量数据的格式转换
- ▶ torchvision.transforms模块包含了图像转换函数，这些函数可以作用于PIL对象和Tensor对象
- ▶ transforms.Compose(): 将多个transforms操作组合在一起

```
import torchvision.transforms as transforms
```

```
transforms.Compose([  
    transforms.CenterCrop(5), #裁剪图像  
    transforms.ToTensor()  
])
```

常用transforms操作

操作	说明
Compose	将多个transforms组合在一起
ToPILImage	将数据格式为(C, H, W)的张量、或(H, W, C)的NumPy数组转换成PIL图像
ToTensor	将数值范围在[0,255]区间的PIL格式的图像、或数据格式为(H, W, C)的NumPy数组转换成(C, H, W)、torch.float类型的张量，数值范围为[0.0, 1.0]
Normalize	对输入数据归一化
Resize	对输入图像调整大小

输入图像以PIL形式读入，返回张量

```
import torch
from PIL import Image
import torchvision.transforms as transforms

loader = transforms.Compose([transforms.ToTensor()])

def image_loader(image_name):
    image = Image.open(image_name).convert('RGB')
    image = loader(image).unsqueeze(0)
    return image.to(device, torch.float)
```

输入张量转换成PIL图像

```
import torch
from PIL import Image
import torchvision.transforms as transforms

unloader = transforms.ToPILImage()

def image_unloader(tensor):
    image = tensor.cpu().clone()
    image = image.squeeze(0)
    image = unloader(image)
    return image
```

使用OpenCV方法读入图像

- ▶ OpenCV (Open Source Computer Vision Library) 是一个开源计算机视觉和机器学习软件库
- ▶ 具有C++、Python、Java、MATLAB接口, 支持Windows, Linux, Android以及Mac操作系统

使用OpenCV方法读入图像

- ▶ `imread(filename)`: 加载图像文件
- ▶ 读入的数据为numpy.ndarray格式, 读入的数据类型为uint8, 取值范围为0-255
- ▶ OpenCV中表示彩色图像使用的是BGR格式, 而不是RGB格式, 因此, 当OpenCV配合其他工具包使用时, 需要进行格式转换

```
import cv2
image = cv2.imread(image_name)
resize_image = cv2.resize(image,(224,224))
resize_image = cv2.cvtColor(resize_image,cv2.COLOR_BGR2RGB)
```

使用torchvision.io包读入图像

- ▶ torchvision.io包提供了对图像、视频文件的读写操作
- ▶ torchvision.io.read_image(path,mode): 读入JPEG或PNG格式的图像, 并保存为3维RGB或灰度张量, 返回输出张量outputtensor[channel,height,width], 数据格式uint8, 值范围[0,255]
- ▶ torchvision.io.read_video(filename): 从文件中读入视频, 返回视频及音频帧

```
from torchvision.io import read_image
import torchvision.transforms as transforms
```

```
#读入图像
```

```
img = read_image('example.jpg')
```

```
#将读入的张量转换为PIL图像
```

```
my_img = transforms.ToPILImage()(img)
```

加载图像

```
loader = transforms.Compose([  
    transforms.Resize(imsz),  
    transforms.ToTensor()]) #转换为torch tensor
```

```
def image_loader(image_name):    #图像加载函数  
    image = Image.open(image_name)  
    image = loader(image).unsqueeze(0)  
    return image.to(device,torch.float)
```

```
my_img = image_loader('style.jpg')
```

显示读入图像

- ▶ `matplotlib.pyplot`是`matplotlib`的一个基于状态的接口。提供了一种类似MATLAB的隐式绘图方式，主要用于交互式绘图和简单的编程绘图生成

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([1,2])
y = np.cos(x)
plt.plot(x,y)
```

常用绘图操作

操作	说明
<code>ion()</code>	打开交互模式
<code>ioff()</code>	关闭交互模式
<code>imshow(image)</code>	在屏幕上显示图像
<code>title(imagetitle)</code>	为图像设置标题
<code>pause(interval)</code>	启动持续interval秒的GUI事件循环。如果有活动的图形，它将在执行该命令前更新和显示，并且在该命令执行期间运行GUI事件循环（如果有）
<code>plot()</code>	绘制线图
<code>figure()</code>	创建一个图形对象的函数，用于后续的绘图操作
<code>show()</code>	显示所有打开的图形

2、构建神经网络

- ▶ 可以自定义神经网络模型，也可以直接调用PyTorch框架中提供的模型
- ▶ 自定义模型
 - ▶ PyTorch提供`torch.nn`、`torch.nn.Module`、`torch.nn.functional`等模块，用于自定义神经网络模型
- ▶ 直接调用预训练模型
 - ▶ PyTorch提供`torchvision.models`，包含了用于处理不同任务的各种模型，如图像分类、语义分割、目标检测、关键点检测等

torch.nn.Module

- ▶ PyTorch使用模块（module）来表示神经网络
- ▶ torch.nn.Module是用于封装PyTorch模型及组件的基类，自定义模型需继承该基类
- ▶ 包含了__init__以及forward方法等
- ▶ __init__方法定义了Module的内部状态，自定义模型时需调用该方法进行模型的初始化
- ▶ forward定义了每次调用需执行的计算操作，自定义的子类会将其覆盖

自定义网络模型的方法

- ▶ 需要继承`torch.nn.Module`类
- ▶ 首先通过`__init__`方法初始化整个模型，定义模型结构及待学习的参数，再使用`forward`方法定义模型的前向计算过程
- ▶ PyTorch支持模型的自动梯度计算，因此在`forward()`中无需定义反向计算过程

```
import torch.nn as nn
import torch.nn.functional as F

class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__() #获取当前类的父类(即nn.Module),并调用父类的构造函数
        self.conv = nn.Conv2d(3, 64, kernel_size = 3, stride = 1)

    def forward(self, x):
        return F.relu(self.conv(x))
```

parameter与buffer

- ▶ 模块类 (module) 中包含两种不同状态的参数：
 - ▶ parameters: 可学习的参数, 即反向传播时可以被优化器更新的参数
 - ▶ buffers: 不可学习的参数, 即反向传播时不可以被优化器更新的参数
- ▶ Parameter被保存在state_dict之中
- ▶ Buffer分成两种: persistent和non-persistent。前者保存在state_dict中, 而后者不包含在state_dict中
- ▶ 保存模型时保存的是包含在state_dict中的参数 (parameter、persistent buffer)
- ▶ 对注册过的parameter和buffer, 在执行module.to(device)操作时, 可以自动进行设备移动

torch.nn.Module的常用属性或方法 (1/2)

属性或方法	说明
parameters()	返回包含了模块中所有parameter的迭代器 (iterator)
buffers()	返回包含了模块中所有buffer的迭代器
state_dict()	返回引用了模块中所有状态的字典，包含了parameter和persistent buffer参数
register_parameter(name,param)	在模块中注册一个parameter。parameter保存在state_dict中
register_buffer(name ,tensor,persistent=True)	在模块中注册一个buffer。persistent为True的buffer保存在state_dict中
add_module(name,module)	将名为name的子模块module添加到当前模块中，子模块通常为torch.nn.Conv2d、torch.nn.ReLU等

torch.nn.Module的常用属性或方法(2/2)

属性或	说明
requires_grad_(requires_grad=True)	原位设置parameter的requires_grad属性。当需要对模型进行精调，或仅对模型的一部分进行训练时，可以使用该方法将不需要变化的部分模块冻结
to(device)/to(dtype)/to(tensor)	转换为指定的设备/数据类型/张量
type(dst_type)	将所有parameter和buffer原位转换为dst_type
zero_grad()	将所有parameter的梯度设置为0
train()	设置模块为训练模式
eval()	设置模块为评估模式

创建parameter的两种方法(1/2)

- ▶ 在定义module时将成员变量（如self.weight）通过nn.Parameter()创建，得到的参数会自动注册为parameters

```
import torch.nn as nn
```

```
class myModule(nn.Module):
```

```
    def __init__(self):
```

```
        super(myModule, self).__init__()
```

```
        self.weight = nn.Parameter(torch.randn(2, 2)) #torch.nn.Parameter继承自  
                                                       #torch.tensor，定义了模块中  
                                                       #的可学习参数
```

```
    def forward(self, x):
```

```
        return x.mm(self.weight)
```

创建parameter的两种方法(2/2)

- ▶ 直接通过`nn.Parameter()`创建普通`Parameter`对象，得到的`parameter`对象再通过`register_parameter()`注册

```
class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__()
        weight = nn.Parameter(torch.randn(2, 2))
        self.register_parameter('weight', weight)

    def forward(self, x):
        return x.mm(self.weight)
```

创建buffer的方法

- ▶ 创建张量，得到的张量对象再通过register_buffer()注册

```
class myModule(nn.Module):  
    def __init__(self):  
        super(myModule, self).__init__()  
        my_buffer = torch.ones(2, 2)  
        self.register_buffer('my_buffer', my_buffer)  
  
    def forward(self, x):  
        return x.mm(self.my_buffer)
```

torch.nn中的计算功能

- 在torch.nn中定义了一些计算功能类，如卷积层、池化层、线性层、归一化层以及非线性激活等操作

计算类型	计算操作
卷积层	nn.Conv1d, nn.Conv2d, nn.Conv3d
池化层	nn.MaxPool1d, nn.MaxPool2d, nn.MaxPool3d, nn.AvgPool1d, nn.AvgPool2d, nn.AvgPool3d
非线性激活	nn.ELU, nn.ReLU, nn.SELU, nn.Sigmoid, nn.Tanh
归一化层	nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d, nn.InstanceNorm1d, nn.InstanceNorm2d, nn.InstanceNorm3d,
循环神经网络层	nn.RNN, nn.LSTM, nn.GRU
线性层	nn.Linear
损失函数	nn.L1Loss, nn.MSELoss, nn.CrossEntropyLoss

torch.nn.functional

- ▶ 包含多种计算函数，可以通过接口直接调用
- ▶ 可以在自定义模型的forward方法中直接调用
- ▶ 常用的函数包括

功能类型	函数
卷积	conv1d、conv2d、conv3d、conv_transpose1d、conv_transpose2d、conv_transpose3d
池化	avg_pool1d、avg_pool2d、avg_pool3d、max_pool1d、max_pool2d、max_pool3d
激活	threshold、relu、elu、softmax、tanh、batch_norm
损失函数	cross_entropy、mse_loss

torch.nn.functional和torch.nn计算操作的比较

- ▶ torch.nn.xx是一个类，通常在模块的__init__方法中通过对其进行实例化来定义模块的成员变量，并在forward方法中进行模型中可学习参数的更新
- ▶ torch.nn.functional.xx是一个函数，通常用于forward方法中，函数中传递的参数需要在__init__方法中创建并初始化
- ▶ 以Conv2d为例说明二者的区别

```
class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)
```

```
torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)
```

用法区别-torch.nn

```
import torch.nn as nn

class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__()
        #实例化nn.Conv2d
        self.conv = nn.Conv2d(3,64,kernel_size=3)

    def forward(self, x):
        return self.conv(x)
```

用法区别-torch.nn.functional

```
import torch.nn as nn
import torch.nn.functional as F

class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__()
        self.weight = nn.Parameter(torch.randn(3,1))
        self.bias = nn.Parameter(torch.randn(1))

    def forward(self, x):
        #F.conv2d函数传递的参数需要在__init__方法中创建并初始化
        return F.conv2d(x, self.weight, self.bias)
```

torch.nn.Sequential

- ▶ 一种序列容器，继承自Module类，将一系列计算操作封装成一个序列
- ▶ 在__init__()方法中按照顺序定义所包含的所有操作，封装到nn.Sequential中
- ▶ 在forward()方法中接收输入并传递给序列中的第一个操作，按顺序将前一个操作的输出传递给下一个操作的输入，最终返回最后一个操作的输出

```
import torch.nn as nn

class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__()
        self.features = nn.Sequential(nn.Conv2d(3,64,kernel_size=3), nn.ReLU())

    def forward(self, x):
        return self.features(x)
```

利用torch.nn.Sequential构建自定义模块

- ▶ 可以利用torch.nn.Sequential，通过数组切片索引的方式，构建自定义计算模块

```
class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__()
        self.layers = nn.Sequential(nn.Conv2d(3,64,kernel_size=3),
                                    nn.ReLU(),
                                    nn.Conv2d(64,192,kernel_size=3),
                                    nn.ReLU())

    def forward(self,x):
        return self.layers(x)

model = myModule()

conv1 = nn.Sequential(*model.layers[:1]) #conv1=conv
conv2 = nn.Sequential(*model.layers[:3]) #conv2=conv+relu+conv
```

直接调用预训练模型

- ▶ 基于某一种网络结构，首先在一个初始任务场景、初始数据集上训练好一个模型，然后再应用到目标任务上，针对目标任务的特征、数据集，对训练好的模型进行精调（fine-tune），最终满足目标任务的需求
- ▶ 在应用到目标任务时，不需要从零开始训练模型，只需要调用已有的预训练模型参数，进行简单的微调，能够节省大量的计算资源和计算时间

torchvision.models

- ▶ PyTorch的torchvision包中提供了torchvision.models子包，其中包含了大量的预训练模型，可用于图像分类、语音分割、目标检测、视频分类、关键点检测等任务
- ▶ 对每一种网络模型，提供多种精度的权重支持，使用时可根据实际需求来加载合适的权重

```
import torchvision.models as models
```

```
#仅加载网络结构，不需要加载预训练模型参数  
my_model1 = models.vgg19()
```

```
#加载网络结构，同时加载一种权重  
my_model2 = models.vgg19(weights = VGG19_Weights.IMAGENET1K_V1)
```

```
#加载网络结构，同时加载当前最优权重  
my_model2 = models.vgg19(weights = VGG19_Weights.DEFAULT)
```

预训练模型的应用 (1/2)

- ▶ 每种预训练模型对其输入均有各自的规格要求（尺寸、像素值等），因此，在应用该预训练模型之前，需要根据这些要求对输入进行预处理
- ▶ Torchvision针对每种权重对输入的要求，提供了预处理方法，可以通过使用.transforms()来实现

```
from torchvision.models import vgg19, VGG19_Weights

my_weights = VGG19_Weights.DEFAULT

#初始化预处理方法
preprocess = my_weights.transforms()

#对输入图像应用预处理方法
my_input_img = preprocess(input_img)
```

预训练模型的应用 (2/2)

- ▶ 在`torchvision.models.vgg`的源代码中，`vgg`网络中包含了3个顺序执行的子模块：`vgg.features`，`vgg.avgpool`，`vgg.classifier`
- ▶ `vgg.features`为`vgg`网络模型中一系列顺序执行的特征层组合，其从`vgg`网络的第一层卷积层开始，到最后一层卷积层（`conv+relu`或`conv+batchnorm+relu`）结束
- ▶ `vgg.classifier`为`vgg`网络模型后段用于进行分类的特征层顺序组合，包括`linear+relu+dropout+linear+relu+dropout+linear`

```
import torchvision.models as models  
  
my_model = models.vgg19().features.to('cuda').eval()
```

https://pytorch.org/vision/stable/_modules/torchvision/models/vgg.html#vgg19

model.train与model.eval (1/2)

- ▶ 在训练开始之前添加 `model.train()`，在测试时添加 `model.eval()`
- ▶ 二者差别主要体现在对Batch Normalization和Dropout操作的处理上，`model.eval()`的作用是不启用BN和 Dropout
- ▶ Dropout：测试的模型应该是最终得到的模型，而这个模型应该是一个完整的模型

model.train与model.eval (2/2)

- ▶ BN层主要涉及到四个需要更新的参数：
 - ▶ running_mean
 - ▶ running_var
 - ▶ weight(gamma)
 - ▶ bias(beta)
- ▶ model.train()保证训练时BN层用每一批数据的均值和方差
- ▶ model.eval()控制BN层中的running_mean,running_std不更新,采用训练结束后的running_mean,running_std来规范化图像

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad // \text{ scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

torchvision.models提供的预训练模型

模型类型	提供的预训练模型
图像分类	AlexNet, ConvNeXt, DenseNet, EfficientNet, EfficientNetV2, GoogLeNet, Inception V3, MaxVit, MNASNet, MobileNet V2, MobileNet V3, RegNet, ResNet, ResNeXt, ShuffleNet V2, SqueezeNet, SwinTransformer, VGG, Vision Transformer, Wide ResNet
量化的图像分类模型 (INT8)	GoogLeNet, Inception V3, MobileNet V2, MobileNet V3, ResNet, ResNeXt, ShuffleNet V2
语义分割	DeepLabV3, FCN, LRASPP
目标检测	Faster R-CNN, FCOS, RetinaNet, SSD, SSDlite
实例分割	Mask R-CNN
关键点检测	Keypoint R-CNN
视频分类	Video MViT, Video ResNet, Video S3D
光流	RAFT

3、实例化神经网络模型

- ▶ 采用如下步骤进行神经网络模型的实例化
 - ▶ 完成神经网络模块（module）的定义，包括__init__()方法和forward()方法的定义
 - ▶ 对模型中的参数（weight、bias等）进行初始化
 - ▶ 实例化模型结构，将结构相关参数传递给module
 - ▶ 定义模型的输入数据
 - ▶ 将模型输入传入实例化后的模型，获取模型输出

模型参数的初始化方法

- ▶ 使用torch.nn.init模块
- ▶ 使用torch.nn.Module.apply函数
- ▶ 使用self.modules

中科院计算技术研究所

使用torch.nn.init模块完成初始化

- ▶ module中的parameter和buffer参数默认为CPU上执行的32位浮点数，在定义module时可以将其设置为任意的数据类型及设备类型
- ▶ 在module的构造函数__init__()中，可以使用torch.nn.init模块来对parameter和buffer参数进行初始化

```
import torch.nn as nn
```

```
#在函数nn.Linear(in_features,out_features)中，权重、偏置的初值默认为服从 (-  
#sqrt (1/in_features), sqrt (1/in_features) ) 的均匀分布
```

```
my_layer = nn.Linear(3, 64)
```

```
#将权重重新初始化为全0
```

```
nn.init.zeros_(my_layer.weight)
```

常用的参数初始化函数

函数	说明
<code>nn.init.constant_()</code>	初始化为常数参数
<code>nn.init.zeros_()</code>	初始化为全0参数
<code>nn.init.ones_()</code>	初始化为全1参数
<code>nn.init.eye_()</code>	初始化为单位矩阵
<code>nn.init.orthogonal_()</code>	初始化为正交矩阵
<code>nn.init.uniform_()</code>	初始化为均匀分布的参数
<code>nn.init.xavier_uniform_()</code>	初始化为xavier均匀分布的参数
<code>nn.init.kaiming_uniform_()</code>	初始化为kaiming均匀分布的参数
<code>nn.init.normal_()</code>	初始化为正态分布的参数
<code>nn.init.xavier_normal_()</code>	初始化为xavier正态分布的参数
<code>nn.init.kaiming_normal_()</code>	初始化为kaiming正态分布的参数

Xavier初始化 (1/3)

- ▶ Xavier Glorot, Yoshua Bengio, Understanding the difficulty of training deep feedforward neural networks, Journal of Machine Learning Research, 2010
- ▶ Glorot条件: 为保证神经网络模型的稳定性和有效性, 避免梯度消失或梯度爆炸, 对模型的初始化需满足两个条件:
 - ▶ $\forall(i, i'), Var[z^i] = Var[z^{i'}]$, 即前向传播时每一层激活值的方差保持一致
 - ▶ $\forall(i, i'), Var\left[\frac{\partial Cost}{\partial s^i}\right] = Var\left[\frac{\partial Cost}{\partial s^{i'}}\right]$, 即反向传播时每一层对状态的梯度方差保持一致

其中, i, i' 表示层号, z^i 表示第 i 层的激活值, $s^i = z^i W^i + b^i$ 为激活函数的输入

Xavier初始化 (2/3)

- ▶ 经推导，第*i*层权重 W_i 的方差需满足：

$$\text{Var}[W_i] = \frac{2}{n_i + n_{i+1}}$$

- ▶ 为此，满足均匀分布的权重 W 的初值需满足：

$$W \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right)$$

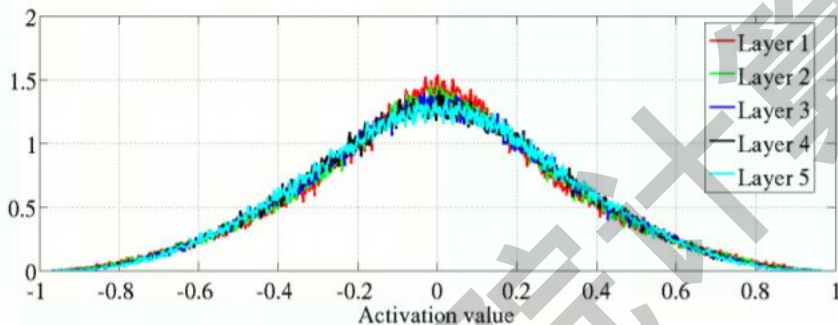
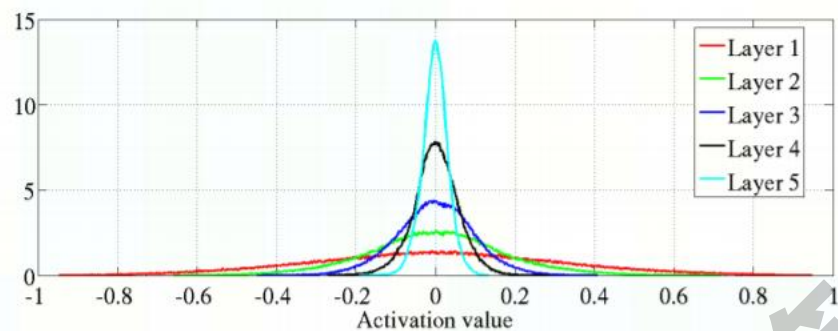
- ▶ 满足正态分布的权重 W 的初值需满足

$$W \sim N\left(0, \frac{\sqrt{2}}{\sqrt{n_i + n_{i+1}}}\right)$$

i 表示当前层号， n_i 表示第*i*层中包含的神经元个数

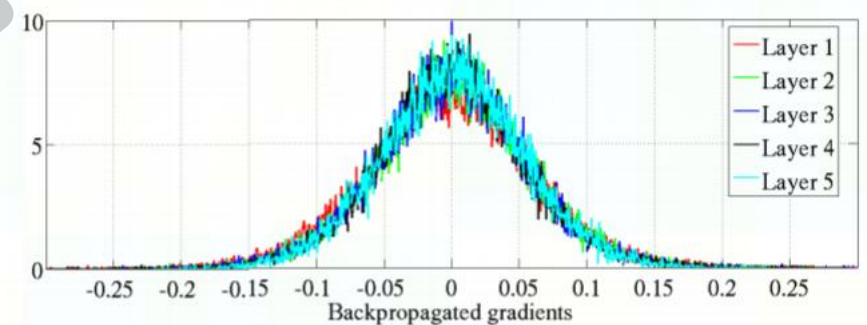
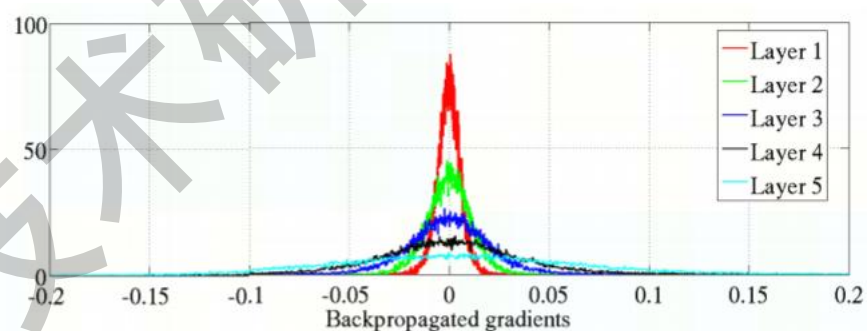
Xavier初始化 (3/3)

- 实验结果：采用 $\tanh(x)$ 激活函数



激活值标准直方图

采用Xavier方法的网络，各层的激活值较为一致，满足Glorot条件



反向传播梯度标准直方图

采用Xavier方法的网络，各层的梯度较为一致，满足Glorot条件

Kaiming初始化 (1/2)

- ▶ Xavier方法适用于关于0对称、在原点处具有单位导数（如 $f'(0)=1$ ）的激活函数，如tanh。对于ReLU、PReLU这种非对称的激活函数的效果并不好
- ▶ 对于ReLU函数，当其输入小于0时输出为0，影响了输出的分布模式
- ▶ Kaiming初始化在满足Glorot条件的基础上，考虑使用ReLU激活函数情况下的权重参数初始化问题

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *IEEE Computer Society* (2015).

Kaiming初始化 (2/2)

- ▶ 为此，在使用了ReLU激活函数的神经网络中，满足均匀分布的权重 W 的初值需满足：

$$W \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_i}}, \frac{\sqrt{6}}{\sqrt{n_i}}\right)$$

- ▶ 满足正态分布的权重 W 的初值需满足

$$W \sim N\left(0, \frac{\sqrt{2}}{\sqrt{n_i}}\right)$$

i 表示当前层号， n_i 表示第 i 层中包含的神经元个数

- ▶ 适用于具有ReLU激活函数的神经网络，在计算机视觉任务上应用较多

使用torch.nn.Module.apply函数完成初始化(1/2)

- ▶ torch.nn.Module.apply(fn): 对模块中的每个子模块（包括模块自身）递归的应用fn方法完成初始化
- ▶ 为神经网络中的每一个/每一种子模块分别定义初始化方法fn，再应用apply(fn)函数实现对每个子模块的初始化

```
class myModule(nn.Module):
    def __init__(self,input_dim,output_dim):
        super(myModule, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(input_dim,output_dim,kernel_size=3),
            nn.BatchNorm2d(output_dim),
            nn.ReLU())

    def forward(self,x):
        return self.layers(x)
```

使用torch.nn.Module.apply函数完成初始化(2/2)

```
#根据子模块计算类型的不同定义不同的初始化方法
```

```
def weights_init(m):  
    if type(m) == nn.Conv2d:  
        nn.init.kaiming_normal_(m.weight)  
    elif type(m) == nn.BatchNorm2d:  
        nn.init.ones_(m.weight)
```

```
#传递模型结构相关参数
```

```
model = myModule(3, 64)
```

```
#将weights_init中定义的初始化方法应用到每一个子模块中
```

```
model.apply(weights_init)
```

使用self.modules()完成初始化

- ▶ self.modules()继承了所定义的module类拥有的方法，按顺序返回此前定义的所有层
- ▶ 遵循深度优先遍历，如遇到Sequential则会继续深入，直到到达最底层子模块
- ▶ 在__init__()方法中，使用self.modules()循环完成所有子模块的初始化

使用self.modules()完成初始化示例

```
class myModule(nn.Module):
    def __init__(self,input_dim,output_dim):
        super(myModule, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(input_dim,output_dim,kernel_size=3),
            nn.BatchNorm2d(output_dim),
            nn.ReLU())
        #完成所有子模块的初始化
        for m in self.modules():
            if type(m) == nn.Conv2d:
                nn.init.kaiming_normal_(m.weight)
            elif type(m) == nn.BatchNorm2d:
                nn.init.ones_(m.weight)

    def forward(self,x):
        return self.layers(x)
```

实例化神经网络模型

- ▶ 完成了模型结构的定义以及模型中参数的初始化后，即可对神经网络模型进行实例化，并将模型的输入数据传递给模型，获取输出结果

```
#实例化，传递模型结构相关参数  
model = myModule(3, 64)
```

```
#定义输入数据  
img = torch.randn(1,3,32,32)
```

```
#获取模型输出结果  
ret = model(img)
```

4、调试

- ▶ 使用python调试工具
 - ▶ 使用交互式调试库python的pdb，进行设置断点，单步执行，查看代码、变量、参数等调试功能
- ▶ 打印模型结构、参数等信息
- ▶ 使用TensorBoard实现数据可视化

打印模型信息

- ▶ 可以使用print语句打印模型结构、参数等信息

```
print(model)
```

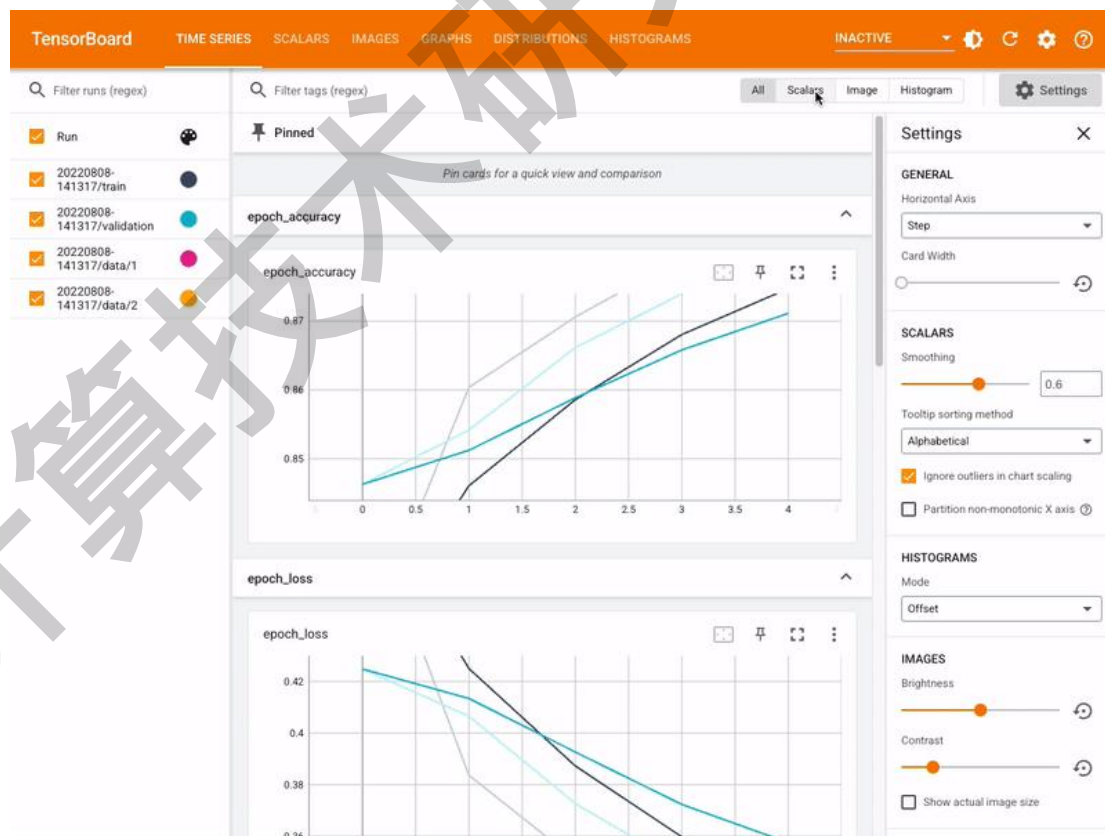
- ▶ 使用第三方torchsummary库，逐层打印模型的形状、参数量等信息

```
from torchsummary import summary  
from torchvision.models import vgg19
```

```
my_model = vgg19()  
summary(my_model, (3,224,224))
```

使用TensorBoard实现数据可视化 (1/3)

- ▶ TensorBoard提供了对神经网络模型结构、参数等的可视化功能，可用于查看、分析神经网络模型的结构、权重、损失值、精度等



使用TensorBoard实现数据可视化 (2/3)

- ▶ PyTorch提供`torch.utils.tensorboard`，用于引入`tensorboard`工具，实现神经网络模型及张量的可视化
- ▶ 主要通过`SummaryWriter`类来实现数据的可视化，其定了多种添加数据显示的方法

方法	说明
<code>add_scalar/scalars()</code>	添加一个/多个标量数据
<code>add_histogram()</code>	添加直方图
<code>add_image/images()</code>	添加一个/多个图像数据
<code>add_graph()</code>	添加模型计算图
<code>add_video()</code>	添加视频数据
<code>add_audio()</code>	添加音频数据
<code>add_text()</code>	添加文本数据

使用TensorBoard实现数据可视化 (3/3)

```
import torch.nn as nn
from torch.utils.tensorboard import SummaryWriter

class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__()
        self.conv = nn.Conv2d(3, 64, kernel_size = 3, stride = 1)
    def forward(self, x):
        return self.conv(x)

model = myModule()
img = torch.randn(1,3,32,32)

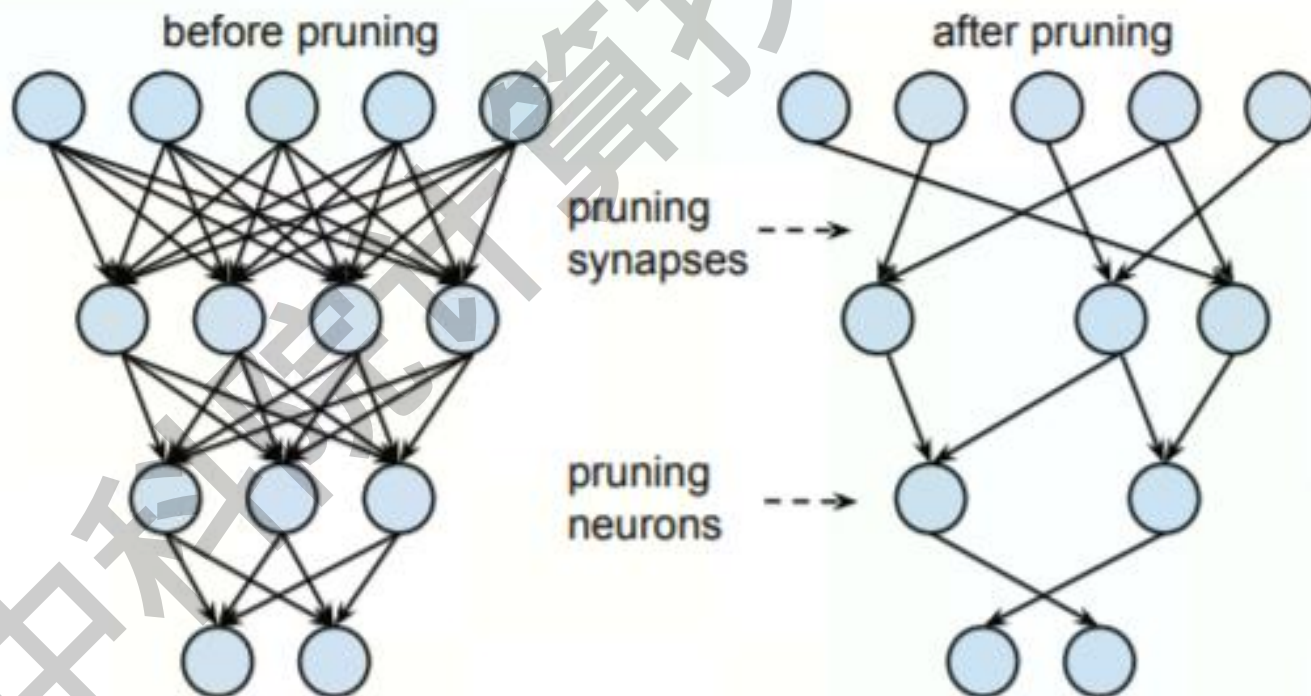
writer = SummaryWriter('./tensorboard') #构造SummaryWriter实例, 指定可视化数据写入路径
writer.add_graph(model, img) #在TensorBoard中可视化模型计算图
writer.close()
```

5、神经网络模型优化

- ▶ 使用`torch.nn.utils.prune`对神经网络模型进行剪枝操作
- ▶ 使用`torch.quantization`进行神经网络模型的量化

神经网络模型剪枝 (1/2)

- 神经网络所包含的大量参数中，有一部分是冗余且对输出结果无贡献的，将这些参数裁减掉，可以提升网络的计算速度，且不影响最终的精度



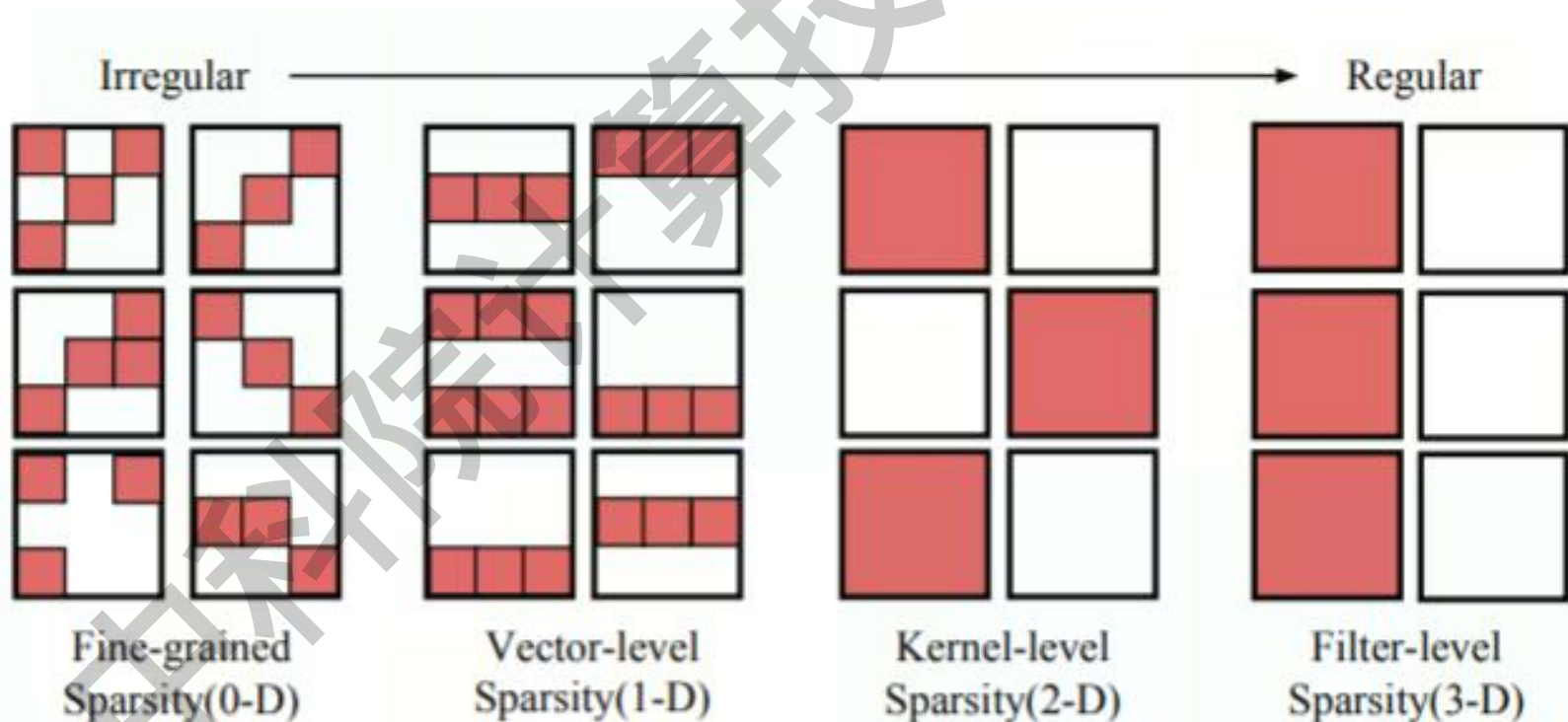
神经网络模型剪枝 (2/2)

Network	Top-1 Error	Top-5 Error	Parameters	Compression Rate
LeNet-300-100 Ref	1.64%	-	267K	
LeNet-300-100 Pruned	1.59%	-	22K	12×
LeNet-5 Ref	0.80%	-	431K	
LeNet-5 Pruned	0.77%	-	36K	12×
AlexNet Ref	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	6.7M	9×
VGG-16 Ref	31.50%	11.32%	138M	
VGG-16 Pruned	31.34%	10.88%	10.3M	13×

数据来源: Han, Song, etc. Learning both weights and connections for efficient neural networks.

常用的剪枝方法分类

- ▶ 非结构化剪枝：按一定规则对单个参数进行裁剪
- ▶ 结构化剪枝：按一定结构规则对一组参数进行裁剪，如裁减掉整个卷积核，或裁减掉卷积核的某一行或某一列



使用torch.nn.utils.prune对神经网络模型剪枝

函数	说明
<code>prune.RandomUnstructured(amount)</code>	随机对张量中amount个单元进行剪枝
<code>prune.RandomStructured(amount)</code>	随机对张量中的amount个通道进行剪枝
<code>prune.L1Unstructured(amount)</code>	对张量中amount个具有最小L1正则化值的单元进行剪枝
<code>prune.random_unstructured(module, name, amount)</code>	随机对module模块中名为name的参数进行剪枝, 剪枝掉amount个单元
<code>prune.l1_unstructured(module, name, amount)</code>	对module模块中名为name的参数进行剪枝, 剪枝掉具有最小L1正则化值的amount个单元
<code>prune.random_structured(module, name, amount, dim)</code>	沿着指定的dim维度, 随机对module模块中名为name的参数进行剪枝, 剪枝掉amount个通道
<code>prune.global_unstructured()</code>	按照指定的剪枝方法进行全局剪枝

使用prune.global_unstructured()剪枝示例

```
import torchvision.models as models
import torch.nn.utils.prune as prune

#实例化vgg19网络
my_model = models.vgg19()

#定义需要剪枝的参数
prune_list = []
for m in my_model.modules():
    if isinstance(m, torch.nn.Conv2d):
        prune_list.append((m,'weight'))

prune_list = tuple(prune_list)
#全局剪枝
prune.global_unstructured(
    prune_list,
    pruning_method = prune.L1Unstructured,
    amount = 0.2)
```

动态量化

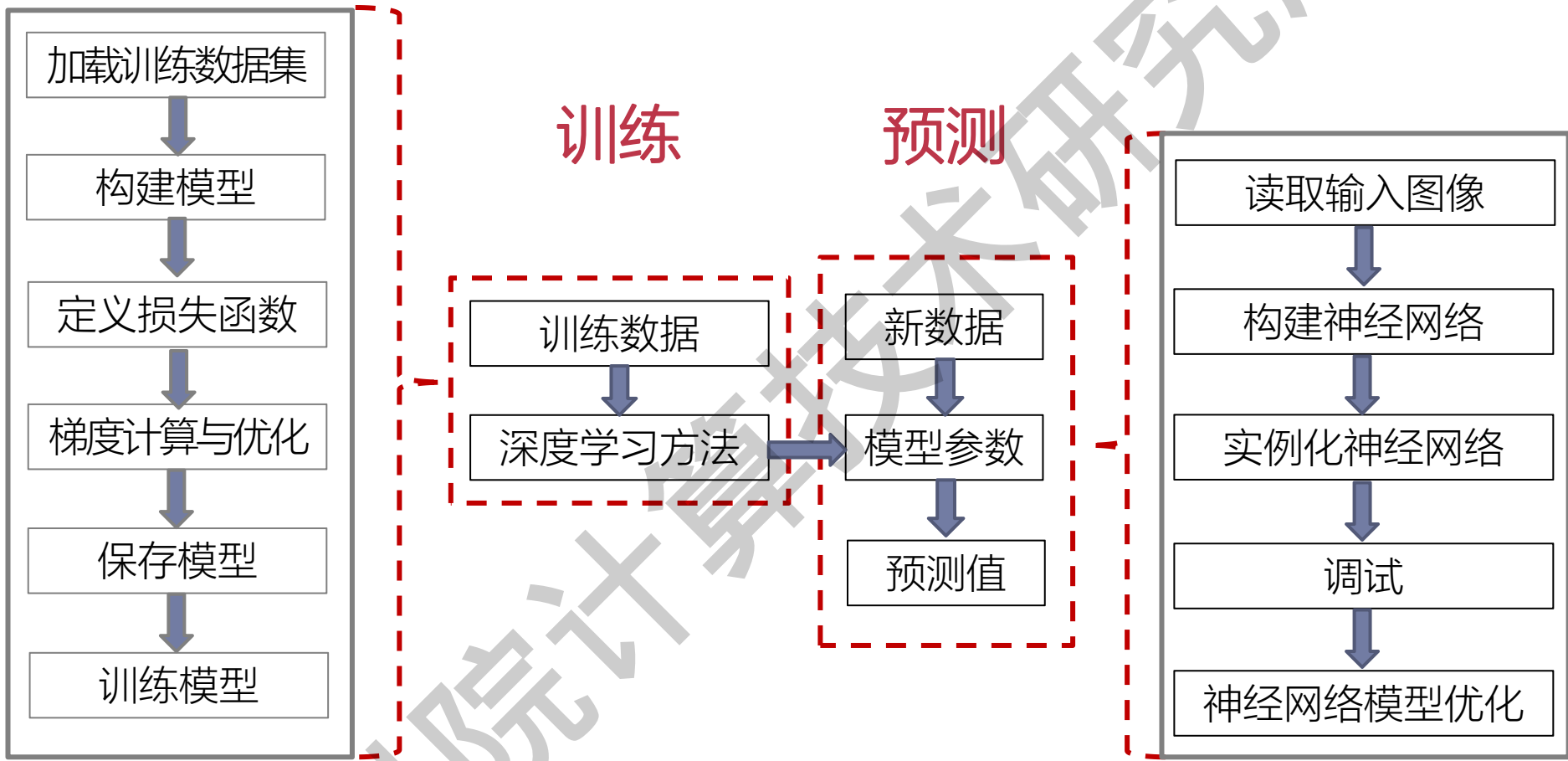
- ▶ 神经网络模型的量化，指将模型中的权重和/或激活数据从多位宽的浮点格式，转换为低位宽的整数型格式
- ▶ 量化过程中使用到的缩放系数与参与量化张量的数值范围有关
- ▶ 权重参数的数值范围是固定的，其缩放系数也是固定的
- ▶ 动态量化：每一层激活数据的数值范围随计算过程变化，需要动态确定缩放系数，根据缩放系数动态完成数据格式转换
- ▶ PyTorch提供`torch.quantization.quantize_dynamic()`用于模型的动态量化

torch.quantization.quantize_dynamic()使用示例

```
class myModule(nn.Module):  
    def __init__(self):  
        super(myModule, self).__init__()  
        self.conv = nn.Conv2d(3, 64, kernel_size = 3, stride = 1)  
  
    def forward(self, x):  
        return self.conv(x)  
  
model_before_quantize = myModule()  
model_quantized = torch.quantization.quantize_dynamic(  
    model = model_before_quantize,  
    qconfig_spec={nn.Conv2d},  
    dtype = torch.qint8
```

提纲

- ▶ 编程框架概述
- ▶ PyTorch概述
- ▶ PyTorch编程模型及基本用法
- ▶ 基于PyTorch的模型推理实现
- ▶ 基于PyTorch的模型训练实现
- ▶ 驱动范例



1、加载训练数据集

- ▶ PyTorch提供了两个数据加载原语：`torch.utils.data.Dataset`和`torch.util.data.DataLoader`，可实现构建数据集、加载数据集等功能



- ▶ 在`torchvision.datasets`中还提供了一些常用数据集的dataset实现，可以直接加载使用

torch.util.data.Dataset

- ▶ 数据集的抽象类，自定义数据集需继承该类
- ▶ 通过复写其中的 `__getitem__()` 方法，定义数据读取及数据预处理方法
- ▶ 通过复写其中的 `__len__()` 方法，定义统计数据集规模的方法

torch.util.data.Dataset使用示例

```
from torch.utils.data import Dataset
import torch

class myDataset(Dataset):
    def __init__(self, inputdata, label):    #读取输入数据
        self.data = inputdata
        self.label = label
    def __getitem__(self,index):    #定义数据获取方式
        return self.data[index], self.label[index]
    def __len__(self):    #定义数据集规模获取方式
        return self.data.size(0)

inputdata = torch.randn(3,2)
label = torch.ones(3)
my_dataset = myDataset(inputdata,label)    #构建自定义数据集
```

torchvision.datasets

- ▶ torchvision.datasets中提供了大量的内建数据集，以及面向数据集的常用操作

任务类别	数据集名称
图像分类	Caltech101/256, CIFAR10/100, EMNIST, FakeData, FashionMNIST, ImageNet, LSUN, MNIST
图像检测或分割	CocoDetection, CelebA, Kitti, SBDataset, VOCSegmentation, VOCSegmentation
光流	HI1K, KittiFlow
图像字幕	CocoCaptions
视频分类	HMDB51, Kinetics, UCF101

- ▶ 所有的内建数据集均为torch.utils.data.Dataset的基类，均已定义好__getitem__和__len__方法

torch.util.data.DataLoader

- ▶ 使用 `Dataset` 构建完成的自定义数据集，以及 `torchvision.datasets` 中内建的数据集，均可以作为参数传递给 `torch.util.data.DataLoader` 类，实现数据集的加载
- ▶ `torch.util.data.DataLoader(dataset, batch_size=1, shuffle=False, num_workers=0)`

```
import torch.utils.data as Data
```

```
my_dataset = torchvision.datasets.ImageNet(path)
```

```
loader = Data.DataLoader(  
    dataset = my_dataset,  
    batch_size = 500,  
    shuffle = True)
```

2、模型训练

- ▶ 使用PyTorch进行模型训练时，首先需要定义损失函数的计算方法，然后构建优化器实现对模型的梯度计算及更新
- ▶ 反向传播过程中，可以利用内建的性能分析工具、梯度检查函数等，验证训练过程的正确性和有效性

损失函数定义

- ▶ 损失函数可以自己定义，也可以直接使用PyTorch提供的内建损失函数
- ▶ 自定义损失函数可以通过定义模块实例来实现

```
class my_loss(nn.Module):  
    def __init__(self):  
        super().__init__()  
    def forward(self,y_n,y):  
        return torch.mean(torch.abs(y_n-y))  
  
loss = my_loss() #实例化损失函数  
loss_compute = loss(input,target) #计算损失函数值
```

内建损失函数

损失函数	说明
<code>nn.L1Loss()</code>	计算预测值与真实值的平均绝对误差 (MAE)
<code>nn.MSELoss()</code>	计算预测值与真实值的均方误差 (MSE)
<code>nn.CrossEntropyLoss()</code>	计算预测值与真实值的交叉熵损失函数
<code>nn.NLLLoss</code>	计算预测值与真实值的负对数似然损失
<code>nn.BCELoss</code>	计算预测值与真实值的二分类交叉熵损失函数

▶ 使用示例

```
loss = nn.MSELoss()           #实例化损失函数
y_n = torch.randn(5, requires_grad=True)
y = torch.randn(5)
my_loss = loss(y_n, y)       #计算损失函数值
```

神经网络模型训练

- ▶ 损失函数定义完成后，需要使用优化器对模型进行训练，包括计算梯度，优化梯度并更新参数等步骤
- ▶ 计算梯度时需要构建计算图并进行反向传播，根据计算图中的张量属性，控制其节点是否需要计算梯度
- ▶ PyTorch提供`torch.optim`包来实现多种梯度优化算法，`torch.optim`包中集成了当前最常用的一些优化方法

构建计算图来计算梯度

- ▶ 计算图的主要作用在于能够在前向计算过程中保存所有中间节点的计算结果，便于反向传播时构建反向传播路径，并利用链式法则完成自动求导。
- ▶ 计算图在一次反向传播后会被立即销毁，释放存储空间，下次调用时需要再次创建
- ▶ 因此，只有在训练时计算图是必需的，而如果只是单纯的推理，可以选择不创建计算图，以节省存储占用和资源消耗

使用torch.no_grad禁用计算图

- ▶ 可以使用torch.no_grad上下文管理器，在其作用域内定义的所有计算，仍然可以前向传播得到计算输出，但不会反向传播计算梯度，也不会创建计算图

```
my_data1 = torch.randn(3,requires_grad=True)
with torch.no_grad():
    my_data2 = my_data1 * 10
```

使用torch.no_grad禁用计算图

- ▶ 可以使用torch.no_grad上下文管理器，在其作用域内定义的所有计算，仍然可以前向传播得到计算输出，但不会反向传播计算梯度，也不会创建计算图

```
my_data1 = torch.randn(3,requires_grad=True)
with torch.no_grad():
    my_data2 = my_data1 * 10
```

```
my_data1 = torch.randn(3,requires_grad=True)

@torch.no_grad()
def nograd(x):
    return x * 10

my_data2 = nograd(my_data1)
```

计算图的反向传播

- ▶ 对于requires_grad属性为True的张量，其前向计算过程中的后继张量，除了使用no_grad管理的，其它张量均默认requires_grad=True，即需要计算梯度，需要创建计算图
- ▶ 可以使用tensor.backward()函数计算当前张量相对于计算图中所有requires_grad属性也为True张量的梯度
- ▶ 每次调用backward()后，计算图会被释放掉

```
my_data1 = torch.randn(1,requires_grad=True)
my_data2 = my_data1 * 10    #my_data2的requires_grad属性也为True

my_data2.backward()        #计算my_data2相对于my_data1的梯度
```

计算图的叶节点 (1/2)

- ▶ 叶节点分为两种情况：1) `requires_grad=False`的张量；2) 由用户直接创建而不是通过某些计算得到的、且 `requires_grad=True`的张量
- ▶ 可以通过`tensor.is_leaf`查看一个张量是否是叶张量
- ▶ 在反向传播时，仅有`requires_grad=True`的节点才会计算梯度，而其中仅有叶节点的`.grad`属性（即其梯度值）会被保存在内存中。非叶张量如果想保留`.grad`属性，需要设置其`retain_grad`参数

计算图的叶节点 (2/2)

#叶张量

```
my_data1 = torch.randn(3,requires_grad=True)
```

```
my_data2 = torch.randn(3,requires_grad=False)
```

```
my_data3 = torch.randn(3).cuda() #默认requires_grad=False, 不是通过额外操作获得
```

#非叶张量

```
my_data4 = torch.randn(3,requires_grad=True) + 1 #通过额外的+1操作获得
```

```
my_data5 = torch.randn(3,requires_grad=True).cuda() #通过额外的.cuda操作获得
```

通过detach()方法修改计算图

- ▶ `tensor.detach()`: 返回一个新张量, 该张量从当前计算图中剥离, 成为一个新的叶张量, 新张量的 `requires_grad=False`, 即不需要计算梯度
- ▶ 返回的张量与原张量共享相同内存, 对原张量或新张量的原位修改 (如尺寸、`stride`等的原位修改) 均会报错

```
my_data1 = torch.randn(3,requires_grad=True)
my_data2 = my_data1 + 1 #my_data2加入计算图, 其requires_grad=True
with torch.no_grad():
    my_data3 = my_data2 * 2 #my_data3不加入计算图, 其requires_grad=False

my_data4 = my_data2.detach() #my_data4从原计算图中剥离, 其requires_grad=False
```

自定义函数的梯度计算

- ▶ 在使用`loss.backward()`计算梯度时，如果`loss`前向计算使用的均为PyTorch中的内建操作函数，则PyTorch能自动根据各操作对应的梯度计算方法完成自动求导
- ▶ `torch.autograd`包提供了用于自动求导的类和函数
- ▶ 如果在模型中使用了某些不可微函数，或需要依赖非PyTorch库（如NumPy），则需要自定义操作的前向计算和反向计算方法，以便于后续PyTorch利用链式法则完成自动求导
- ▶ 为了保证自定义函数的正确性，可以使用`torch.autograd.gradcheck()`、`torch.autograd.detect_anomaly()`等函数来验证梯度计算功能的准确性

自定义计算操作示例

```
class my_sin(Function):      # torch.autograd.Function为自定义自动求导操作的基类
    @staticmethod
    def forward(ctx,input):  #ctx为上下文目标，可用于存储反向计算信息
        output = torch.sin(input)
        ctx.save_for_backward(output) #保存需要在反向计算时用到的张量
        return output

    @staticmethod
    def backward(ctx,grad_output): #grad_output为上游节点传递的梯度值
        output, = ctx.saved_tensors
        return torch.cos(output) * grad_output

def sin(input):
    return my_sin.apply(input)
```

自定义操作的验证、调试

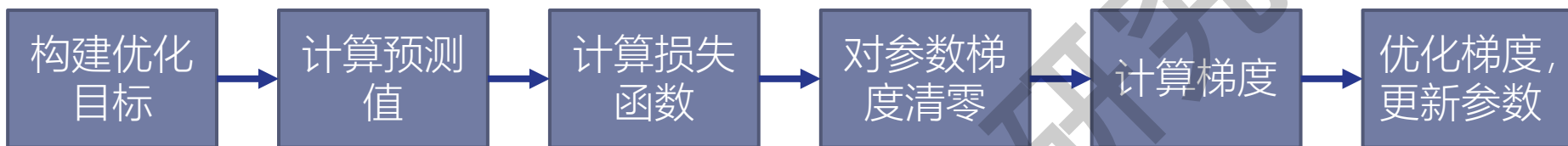
```
import torch
from torch import autograd

x = torch.randn(10, requires_grad=True)
#对使用数值求导法求得的梯度和采用自定义方法求得的梯度进行数值比较，以检查
#自定义backward()方法是否正确
test = autograd.gradcheck(my_sin.apply, x, eps=1e-3)
```

▶ 数值求导法：
$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

```
#启动自动求导异常检测，可以在反向传播时打印异常发生路径
with autograd.detect_anomaly():
    x = torch.randn(10, requires_grad=True)
    y_n = sin(x)
    y_n.backward()
```

使用torch.optim包优化梯度



- ▶ 完成梯度计算后，可以使用torch.optim包来优化梯度、更新模型参数
- ▶ 支持的常用梯度优化算法：Adadelta、Adagrad、Adam、RMSprop、SGD、LBFGS
- ▶ `torch.optim.Optimizer(params, defaults)`：所有优化器的基类。其中，`params`为需要优化的模型参数列表，`defaults`为包含了如`learning rate`等优化选项的字典

Optimizer的常用操作

操作	说明
<code>add_param_group()</code>	添加一组参数到待优化参数列表中
<code>load_state_dict()</code>	加载优化器状态字典
<code>state_dict()</code>	返回优化器的状态字典，包含了优化器的状态信息、使用的超参数等
<code>step()</code>	执行一次梯度优化计算及参数更新
<code>zero_grad()</code>	将所有待优化参数的梯度清零，避免多次优化过程中梯度值累加

优化器使用示例

```
my_model = vgg19()
loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(my_model.parameters(),lr=0.01)

for input,y in dataset:
    optimizer.zero_grad()           #将所有参数梯度清零
    y_n = my_model(input)
    my_loss = loss(y_n, y)
    my_loss.backward()              #计算梯度
    optimizer.step()                #执行一次梯度优化、参数更新
```

模块的state_dict与优化器的state_dict

- ▶ `torch.nn.Module.state_dict`中包含了模型中的parameter以及persistent buffer 两类参数，记录了神经网络中包含可学习参数的层（如卷积层、线性层，而非maxpool、ReLU等不含可学习参数的层）对应的weight、bias参数等
- ▶ `torch.nn.Module.state_dict`的键即为layer.weight/bias
- ▶ `torch.optim.state_dict`记录了优化器的状态、待优化参数、优化器中使用的超参数等

模块的state_dict示例

```
class myModule(nn.Module):
    def __init__(self):
        super(myModule, self).__init__()
        self.features = nn.Sequential(nn.Conv2d(3,64,kernel_size=3), nn.ReLU())

    def forward(self, x):
        return self.features(x)

model = myModule()
print(model.state_dict().keys())
```

▶ 输出结果：

```
odict_keys(['features.0.weight', 'features.0.bias'])
```

优化器的state_dict示例

```
optimizer = optim.SGD(model.parameters(),lr=0.01)
for name in optimizer.state_dict():
    print(name, '\t', optimizer.state_dict()[name])
```

▶ 输出结果：

```
state      {}
param_group [{"lr":0.01, 'momentum':0, 'dampening':0, 'weight_decay':0,
'nesterov':False, 'params':[36728930461,63729150318]}]
```

3、模型的保存与恢复

- ▶ 使用`torch.save()`保存模型
 - ▶ **官方推荐：**可以仅保存模型的`state_dict`，模型保存格式为`.pt`或`.pth`。使用`model.load_state_dict()`恢复模型
 - ▶ 也可以自定义保存的内容，如将模型的`state_dict`、优化器的`state_dict`、`epoch`、`loss`值等一起保存为检查点，检查点文件的常见保存格式为`.tar`。使用`torch.load()`恢复模型

保存模型的state_dict示例

```
path = os.path.join(model_dir, 'model.pt')  
torch.save(model.state_dict(), path) #保存模型的state_dict到指定路径
```

- ▶ 模型的state_dict()会随着模型的训练过程的进行而更新，因此，如果想保存某个指定的state_dict()，需要使用：

```
model_save = deepcopy(model.state_dict()) #对state_dict及其子对象深度拷贝  
# model_save = model.state_dict() #不可使用这种方法，因为该方法仅能得到对  
#state_dict()的引用，其值会随训练过程变化  
torch.save(model_save, path)
```

推理时装载模型的state_dict示例

- ▶ 在进行神经网络模型推理时，首先完成模型的初始化，再使用`model.load_state_dict()`装载模型参数

```
model = vgg19()
model.to(device)
path = os.path.join(model_dir, 'model.pt')
model.load_state_dict(torch.load(path)) #使用保存的state_dict()来装载模型参数
model.eval() #在执行推理前必须调用，将dropout和批归一化层设置为评估模式

y_n = model(x) #获得推理输出
```

保存模型的检查点示例

- ▶ 检查点文件可用于模型的再次训练，因此，除了模型的 `state_dict()`，在检查点文件中还可以将优化器的 `state_dict`、`epoch`、`loss` 值等一起保存下来。

```
path = os.path.join(model_dir, 'model.rar')
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss
}, path)
```

- ▶ 再次训练时可以使用 `torch.load` 将检查点文件中所有参数一起恢复

恢复模型的检查点文件

- ▶ 可以使用`torch.load`恢复检查点文件中的参数
- ▶ 可用于推理或继续训练

```
path = os.path.join(model_dir, 'model.rar')
model = vgg19()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```
checkpoint = torch.load(path)
epoch = checkpoint['epoch']
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
loss = checkpoint['loss']
```

提纲

- ▶ 深度学习编程框架的概念
- ▶ PyTorch概述
- ▶ PyTorch编程模型及基本用法
- ▶ 基于PyTorch的模型推理实现
- ▶ 基于PyTorch的模型训练实现
- ▶ 驱动范例

Recall: 非实时风格迁移算法



- ▶ 给定一张风格图像 a 和一张内容图像 p ;
- ▶ 风格图像经过 CNN 生成的 feature maps 组成风格特征集 A ; 内容图像 p 通过 CNN 生成的 feature maps 组成内容特征集 P ;
- ▶ 输入一张随机噪声图像 x , 随机噪声图像 x 通过 CNN 生成的 feature maps 构成内容特征和风格特征集合 F 和 G , 目标损失函数由 A, P, F, G 计算得到;
- ▶ 优化函数是希望调整图像 x , 使其最后看起来既保持内容图像 p 的内容, 又有风格图像 a 的风格。

Recall: 非实时风格迁移算法

- ▶ 论文中使用的CNN网络为在imageNet上训练好的VGG19, 去除了最后的全连接层和softmax。
- ▶ 内容损失函数:
 - ▶ 只取conv4单层特征来计算内容损失;
 - ▶ 计算内容图片特征和噪声图片特征之间的欧式距离;

$$L_{content}(\mathbf{p}, \mathbf{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2$$

- l : 用于计算内容特征的层数.
- F_{ij}^l : 生成图片在第 l 层第 i 个特征图上位置 j 处的特征值
- P_{ij}^l : 内容图片在第 l 层第 i 个特征图上位置 j 处的特征值
- p : 内容图片
- x : 生成图片

Recall: 非实时风格迁移算法

▶ 风格损失函数:

- ▶ 取 conv1~conv5共5层的特征来计算风格损失;
- ▶ 使用相关矩阵来表示图像的风格;

$$L_{style}(\mathbf{a}, \mathbf{x}) = \sum_l w_l E_l$$

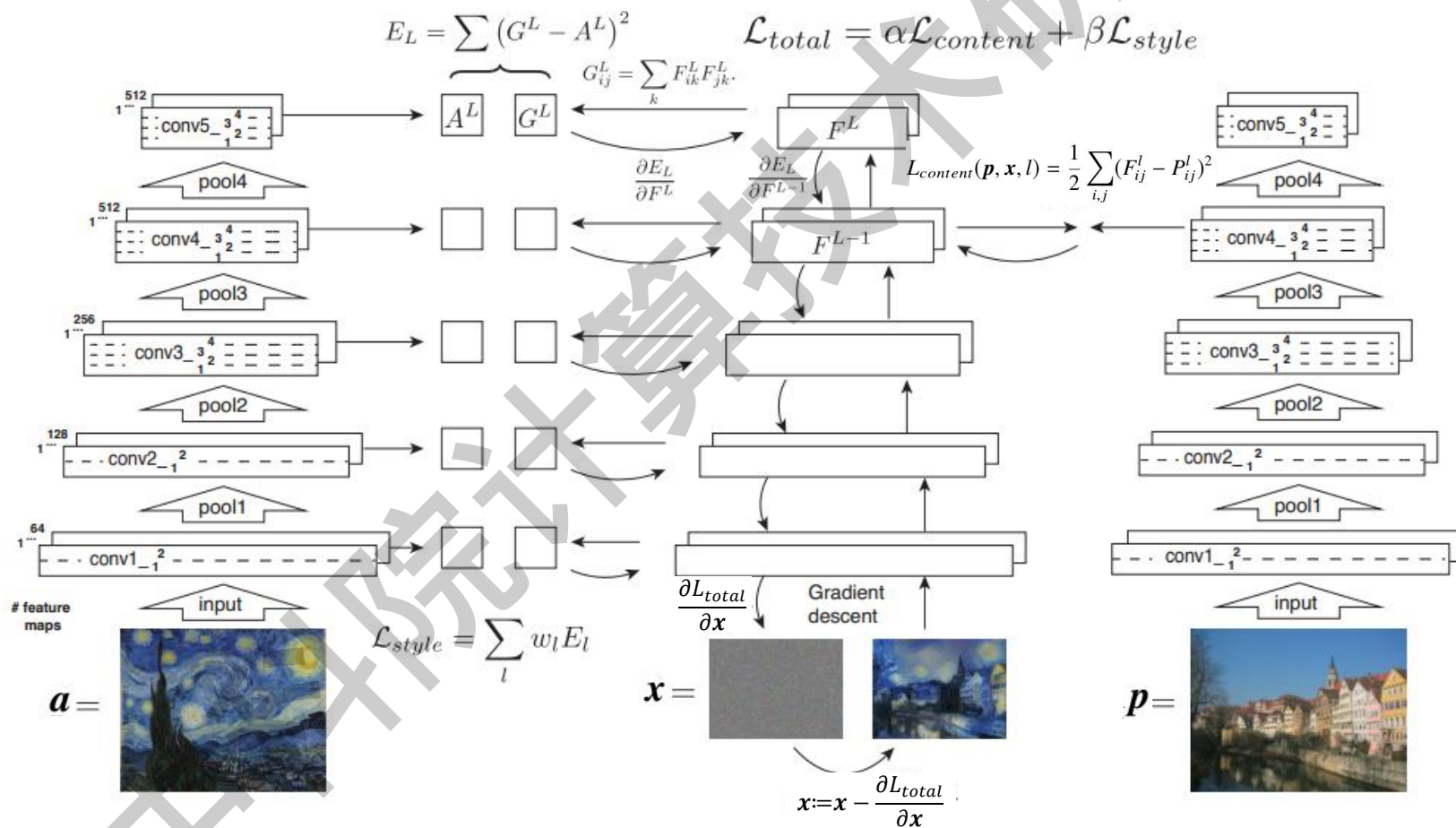
$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \quad (\text{gram矩阵})$$

- L : 用于计算风格特征的层数
- w_l : 第 l 层的 E_l 用于计算风格损失的权重, 文中都取0.2
- \mathbf{a} : 初始风格图片
- \mathbf{x} : 生成图片
- A_{ij}^l : 风格图片在 l 层第 i 个特征图和第 j 个特征图的内积
- G_{ij}^l : 生成图片在 l 层第 i 个特征图和第 j 个特征图的内积
- M_l : 第 l 层的输出特征图的大小
- N_l : 第 l 层的输出特征图的数目
- F_{ik}^l : 第 l 层第 i 个特征图上位置 k 处的特征值

Recall: 非实时风格迁移算法

$$L_{total}(\mathbf{p}, \mathbf{a}, \mathbf{x}) = \alpha L_{content}(\mathbf{p}, \mathbf{x}) + \beta L_{style}(\mathbf{a}, \mathbf{x})$$



加载依赖包

```
from __future__ import print_function

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from PIL import Image
import matplotlib.pyplot as plt

import torchvision.transforms as transforms
import torchvision.models as models

import copy
```

代码来源: https://pytorch.org/tutorials/advanced/neural_style_tutorial.html

加载内容图像和风格图像

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

#指定输出图像尺寸, 如果使用GPU则输出图像尺寸为512, 否则为128
imgsize = 512 if torch.cuda.is_available() else 128

loader = transforms.Compose([
    transforms.Resize(imgsize),
    transforms.ToTensor()]) #转换为torch tensor

def image_loader(image_name):    #图像加载函数
    image = Image.open(image_name)
    image = loader(image).unsqueeze(0)
    return image.to(device, torch.float)

style_img = image_loader('style.jpg')
content_img = image_loader('content.jpg')
assert style_img.size() == content_img.size(), 'style and content images are of the same size'
```

显示内容图像和风格图像

```
unloader = transforms.ToPILImage() #转换为PIL图像
```

```
plt.ion()
```

```
def imshow(tensor, title=None):
```

```
    image = tensor.cpu().clone() #复制输入图像
```

```
    image = image.squeeze(0) #去掉第0维
```

```
    image = unloader(image)
```

```
    plt.imshow(image)
```

```
    if title is not None:
```

```
        plt.title(title)
```

```
    plt.pause(0.001)
```

```
plt.figure()
```

```
imshow(style_img, title = 'Style Image')
```

```
plt.figure()
```

```
imshow(content_img, title = 'Content Image')
```

创建输入图像

```
input_img = torch.randn(content_image.data.size(),device = device)
```

```
plt.figure()
```

```
imshow(input_img, title = 'Input Image')
```

内容损失函数定义

```
class ContentLoss(nn.Module):  
  
    def __init__(self, target):  
        super(ContentLoss, self).__init__()  
        self.target = target.detach()  
  
    def forward(self, input):  
        self.loss = F.mse_loss(input, self.target)  
        return input
```

风格损失函数定义

```
def gram_matrix(input):  
    # a为batch_size(=1),b为通道数,  
    # (c,d)为特征图高度、宽度  
    a, b, c, d = input.size()  
    features = input.reshape(a * b, c * d) #将输入特征图形状转换为(a * b, c * d)  
    G = torch.mm(features, features.t()) #计算特征图内积  
    return G.div(a * b * c * d)
```

```
class StyleLoss(nn.Module):  
    def __init__(self, target_feature):  
        super(StyleLoss, self).__init__()  
        self.target = gram_matrix(target_feature).detach()  
    def forward(self, input):  
        G = gram_matrix(input)  
        self.loss = F.mse_loss(G, self.target)  
        return input
```

图像预处理

```
class Normalization(nn.Module):  
    def __init__(self, mean, std):  
        super(Normalization, self).__init__()  
        self.mean = torch.tensor(mean).reshape(-1, 1, 1)  
        self.std = torch.tensor(std).reshape(-1, 1, 1)  
  
    def forward(self, img):  
        return (img - self.mean) / self.std
```

计算损失函数 (1/3)

```
cnn = models.vgg19(pretrained=True).features.to(device).eval()
content_layers_default = ['conv_4']
style_layers_default = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']

def get_style_model_and_losses(cnn, normalization_mean, normalization_std, style_img,
                               content_img, content_layers=content_layers_default,
                               style_layers=style_layers_default):
    normalization = Normalization(normalization_mean, normalization_std).to(device)
    content_losses = []
    style_losses = []
    model = nn.Sequential(normalization)
```

计算损失函数 (2/3)

```
i = 0 # increment every time we see a conv
for layer in cnn.children():
    if isinstance(layer, nn.Conv2d):
        i += 1
        name = 'conv_{}'.format(i)
    elif isinstance(layer, nn.ReLU):
        name = 'relu_{}'.format(i)
        layer = nn.ReLU(inplace=False)
    elif isinstance(layer, nn.MaxPool2d):
        name = 'pool_{}'.format(i)
    elif isinstance(layer, nn.BatchNorm2d):
        name = 'bn_{}'.format(i)
    else:
        raise RuntimeError('Unrecognized layer: {}'.format(layer.__class__.__name__))
    model.add_module(name, layer)
```

计算损失函数 (3/3)

```
if name in content_layers:
    target = model(content_img).detach()
    content_loss = ContentLoss(target)
    model.add_module('content_loss_{}'.format(i), content_loss)
    content_losses.append(content_loss)
if name in style_layers:
    target_feature = model(style_img).detach()
    style_loss = StyleLoss(target_feature)
    model.add_module('style_loss_{}'.format(i), style_loss)
    style_losses.append(style_loss)

for i in range(len(model) - 1, -1, -1):
    if isinstance(model[i], ContentLoss) or isinstance(model[i], StyleLoss):
        break
model = model[:i + 1]
return model, style_losses, content_losses
```

风格迁移算法 (1/3)

```
def get_input_optimizer(input_img):  
    optimizer = optim.LBFGS([input_img])  
    return optimizer  
  
def run_style_transfer(cnn, normalization_mean, normalization_std,  
                      content_img, style_img, input_img, num_steps=300,  
                      style_weight=1000000, content_weight=1):  
    print('Building the style transfer model..')  
    model, style_losses, content_losses = get_style_model_and_losses(cnn,  
                                                                    normalization_mean,  
                                                                    normalization_std,  
                                                                    style_img,  
                                                                    content_img)  
  
    input_img.requires_grad_(True)  
    model.requires_grad_(False)  
  
    optimizer = get_input_optimizer(input_img)
```

风格迁移算法 (2/3)

```
run = [0]
while run[0] <= num_steps:
    def closure():
        with torch.no_grad():
            input_img.clamp_(0, 1)
        optimizer.zero_grad()
        model(input_img)
        style_score = 0
        content_score = 0
        for sl in style_losses:
            style_score += sl.loss
        for cl in content_losses:
            content_score += cl.loss
        style_score *= style_weight
        content_score *= content_weight
        loss = style_score + content_score
```

风格迁移算法 (3/3)

```
loss.backward()
run[0] += 1
if run[0] % 50 == 0:
    print('run {}'.format(run))
    print('Style Loss : {:.4f} Content Loss: {:.4f}'.format(
        style_score.item(), content_score.item()))
return style_score + content_score

optimizer.step(closure)

with torch.no_grad():
    input_img.clamp_(0, 1)

return input_img
```

风格迁移算法运行

```
cnn_normalization_mean = torch.tensor([0.485, 0.456, 0.406]).to(device)
cnn_normalization_std = torch.tensor([0.229, 0.224, 0.225]).to(device)

output = run_style_transfer(cnn, cnn_normalization_mean, cnn_normalization_std,
                             content_img, style_img, input_img)

plt.figure()
imshow(output, title='Output Image')

plt.ioff()
plt.show()
```

小结

- ▶ 深度学习编程框架的概念

深度学习编程框架的概念及分类

- ▶ PyTorch概述

PyTorch的历史、发展历程

- ▶ PyTorch编程模型及基本用法

Numpy基础，PyTorch中常用的张量、操作、计算图等概念

- ▶ 基于PyTorch的模型推理及训练实现

PyTorch模型训练、推理等

- ▶ 驱动范例

基于PyTorch的非实时风格迁移实现



谢谢大家!

中科院计算技术研究所