

PyTorch入门

1. **Tensors**张量
2. **Autograd**自动求导
3. 神经网络
4. 训练一个分类器

1. Tensors张量

3

1. Tensors张量

2. Autograd自动求导

3. 神经网络

4. 训练一个分类器

1. Tensors 张量的概念

4

Tensor 实际上就是一个多维数组 (multidimensional array)

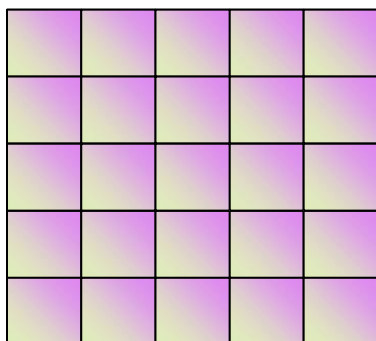
标量 (0阶张量)



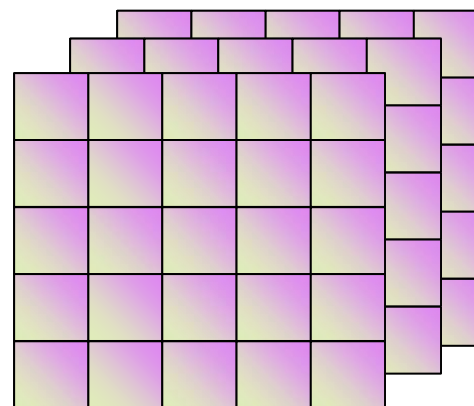
向量 (1阶张量)



矩阵 (2阶张量)



张量 (大于等于3阶张量)



1.Tensors张量的概念

5

- 创建张量的几种方法
 - 用现有数据创建张量，使用`torch.tensor()`
 - 如`torch.tensor([[1., -1.], [1., -1.]])`
 - 要创建具有特定大小的张量，请使用`torch.*`
 - 如`torch.randn()` #满足标准正态分布的一组随机数据
 - 创建与另一个张量具有相同大小的张量，请使用`torch.*_like`
 - 如`torch.rand_like()`
 - 创建与其他张量具有相似类型但大小不同的张量，请使用`tensor.new_*`创建操作。

1.Tensors张量的概念

6

- 查看张量的属性
 - 查看Tensor类型
 - `tensor1 = torch.randn(2,3)` #形状为(2,3)一组从标准正态分布中随机抽取的数据
 - `tensor1.dtype` # `torch.float32`
 - 查看Tensor维度和形状
 - `tensor1.shape` #查看形状或尺寸
 - `tensor1.ndim` #查看维度
 - 查看Tensor是否存储在GPU上
 - `tensor1.is_cuda`
 - 查看Tensor的梯度
 - `tensor1.grad`

1.Tensors张量的概念

7

- **Tensor在CPU和GPU之间转换，以及numpy之间的转换**
 - CPU tensor转GPU tensor
 - `cpu_tensor.cuda()`
 - GPU tensor 转CPU tensor
 - `gpu_tensor.cpu()`
 - numpy转为CPU tensor
 - `torch.from_numpy(numpy_test)`
 - CPU tensor转为numpy数据
 - `cpu_tensor.numpy()`
 - 注意：
 - GPU tensor不能直接转为numpy数组，必须先转到CPU tensor。
 - 如果tensor是标量的话，可以直接使用 `item()` 函数（只能是标量）将值取出来：`loss_output.item()`

1.Tensors张量的概念

● Tensor与NumPy的异同

对比项	NumPy	Tensor
相同点	可以定义多维数组，进行切片、改变维度、数学运算等	可以定义多维数组，进行切片、改变维度、数学运算等
不同点	<div>1、产生的数组类型为 numpy.ndarray； 2、会将ndarray放入 CPU中进行运算； 3、导入方式为import numpy as np，后续 通过np.array([1,2])建立数组； 4、numpy中没有x.type()的用法，只能使用 type(x)。</div>	<div>1、产生的数组类型为torch.Tensor； 2、会将tensor放入GPU中进行加速运算（如果有GPU） 3、导入方式为import torch，后续通过 torch.tensor([1,2])或torch.Tensor([1,2])建立数组； 4、Tensor中查看数组类型既可以使用type(x)，也可以使用 x.type()。但是更加推荐采用x.type()（这种方式能看到 更具体信息） 5、tensor 含义 device（是否使用GPU）， requires_grad(是否需要求导)等设置参数。</div>

1.Tensors张量的概念

9

- Tensor与NumPy的函数对比

操作类别	Numpy	PyTorch
数据类型	np.ndarray	torch.Tensor
	np.float32	torch.float32; torch.float
	np.float64	torch.float64; torch.double
	np.int64	torch.int64; torch.long
从已有数据构建	np.array([3.2, 4.3], dtype=np.float16)	torch.tensor([3.2, 4.3], dtype=torch.float16)
	x.copy()	x.clone()
	np.concatenate	torch.cat
线性代数	np.dot	torch.mm
属性	x.ndim	x.dim()
	x.size	x.nelement()
形状操作	x.reshape	x.reshape(相当于 tensor.contiguous().view()); x.view
	x.flatten	x.view(-1); nn Flatten()
类型转换	np.floor(x)	torch.floor(x); x.floor()
比较	np.less	x.lt
	np.less_equal/np.greater	x.le/x.gt
	np.greater_equal/np.equal/np.not_equal	x.ge/x.eq/x.ne
随机种子	np.random.seed	torch.manual_seed

1.Tensors张量的概念

● Python、PyTorch 1.x与TensorFlow2.x的比较

类别	Python	PyTorch 1+	TensorFlow 2+
类型	nn.nd	Tensor	Tensor
自动求导	无	支持,示例 x=torch.tensor([2.0,3.6],requires_grad=True)	支持，①对变量求导示例 v=tf.Variable([3.2, 4.3], dtype=tf.float16), #TensorFlow一般使用梯度磁带tf.GradientTape来记录正向运算过程，然后反播磁带自动得到梯度值。 ②对常量也可求导，需要增加watch。 ③对tf.Variable可以通过参数trainable控制是否可学习，缺省是True。
是否支持GPU	不支持	支持	支持
常量示例	5.6	torch.tensor([5.6])	a=tf.constant([3.2, 4.3], dtype=tf.float16)
变量示例	x=10.5	torch.tensor([5.6])	v=tf.Variable([3.2, 4.3], dtype=tf.float16)

1.Tensor张量乘法

11

1. 二维矩阵乘法`torch.mm()`

`torch.mm(mat1, mat2, out=None)`

其中 $mat1 \in \mathbb{R}^{n \times m}$, $mat2 \in \mathbb{R}^{m \times d}$, 输出的 $out \in \mathbb{R}^{n \times d}$

该函数一般只用来计算两个二维矩阵的矩阵乘法，并且不支持broadcast操作。

1.Tensor张量乘法

12

2. 三维带batch的矩阵乘法 torch.bmm()

由于神经网络训练一般采用mini-batch，经常输入的是三维带batch的矩阵，所以提供torch.bmm(bmat1, bmat2, out=None)

其中 $bmat1 \in \mathbb{R}^{b \times n \times m}$, $bmat2 \in \mathbb{R}^{b \times m \times d}$, 输入出的 $out \in \mathbb{R}^{b \times n \times d}$

该函数的两个输入必须是三维矩阵并且第一维相同（表示Batch维度），不支持broadcast操作

1.Tensor张量乘法

13

3. 多维矩阵乘法torch.matmul()

`torch.matmul(input, other, out=None)`

支持broadcast操作，使用起来比较复杂。针对多维数据matmul()乘法，可以认为该乘法使用使用两个参数的后两个维度来计算，其他的维度都可以认为是batch维度。

假设两个输入的维度分别是`input(1000 × 500 × 99 × 11)`，`other(500 × 11 × 99)`那么我们可以认为`torch.matmul(input, other, out=None)`乘法首先是进行后两位矩阵乘法得到 $(99 \times 11) \times (11 \times 99) \Rightarrow (99 \times 99)$ ，然后分析两个参数的batch size分别是 (1000×500) 和500，可以广播成为 (1000×500) ，因此最终输出的维度是 $(1000 \times 500 \times 99 \times 99)$ 。

1.Tensors张量乘法

14

4. 矩阵逐元素(Element-wise)乘法 torch.mul()

`torch.mul(mat1, other, out=None)`

其中 `other` 乘数可以是标量，也可以是任意维度的矩阵，只要满足最终相乘是可以broadcast的即可。

1.Tensors张量乘法

15

5. 两个运算符@ 和 *

@

: 矩阵乘法, 自动执行适合的矩阵乘法函数

*: element-wise乘法

2. Autograd自动求导

16

1. Tensors张量

2. Autograd自动求导

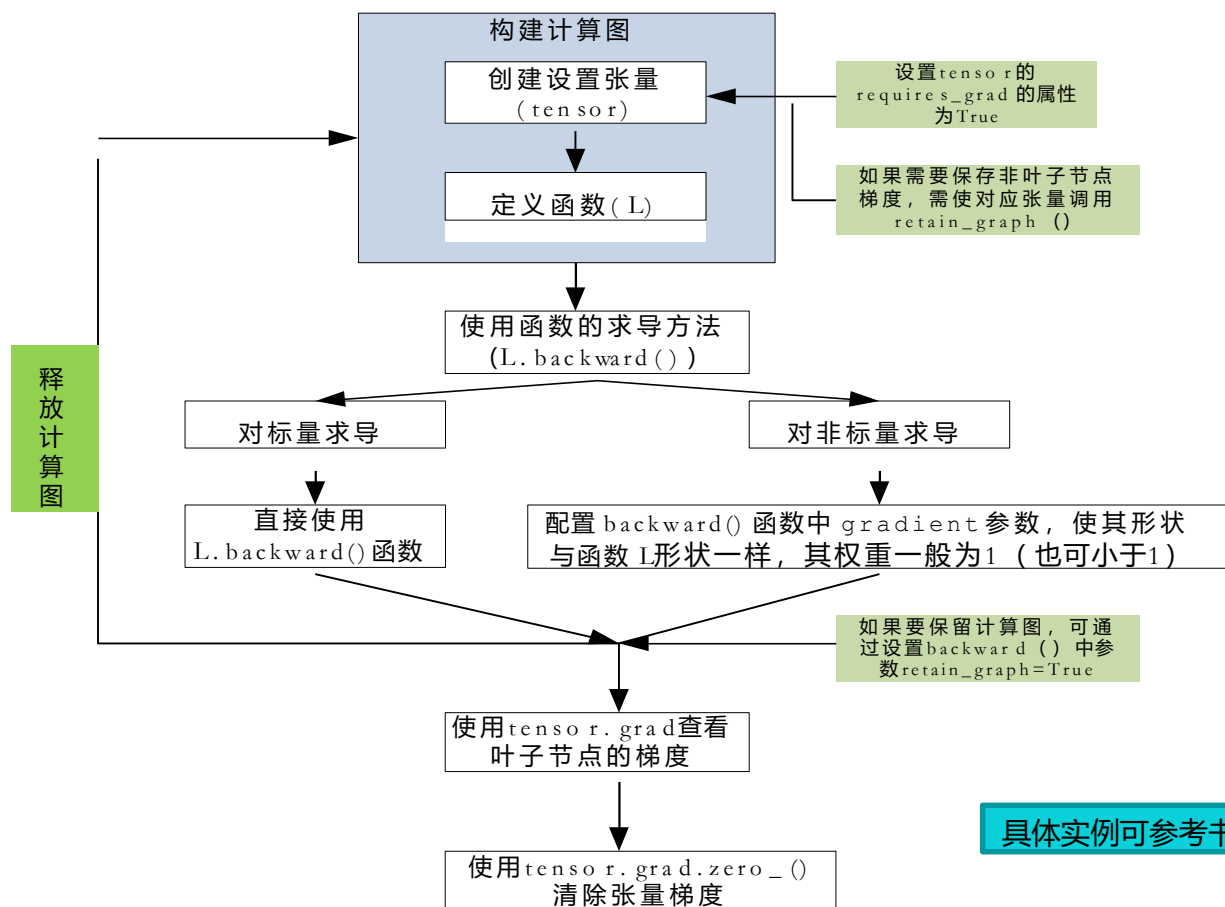
3. 神经网络

4. 训练一个分类器

2. Autograd自动求导

17

• PyTorch 1.x的自动微分机制



具体实例可参考书中2.7小节内容

2. Autograd自动求导

18

- PyTorch 1.x的Tensor不参与求导的几种方式

张量操作	新建/共享内存	留在计算图中	使用场景
<code>tensor.clone()</code>	新建	是 (即 <code>tensor</code> 与 <code>tensor.clone()</code> 的 <code>requires_grad</code> 一致)	常用在神经网络中某个单元需要重复使用，但不参与求导的场景下
<code>tensor.detach()</code>	共享	否	常用在神经网络中仅要利用张量数值，而不需要追踪导数的场景下
<code>tensor.clone().detach()</code>	新建	否	只做简单的数据复制，既不数据共享，也不对梯度共享，从此两个张量无关联。

2. Autograd自动求导

19

PyTorch之自动梯度

在训练一个神经网络时，**梯度**的计算是一个关键的步骤，它为神经网络的**优化**提供了关键数据。

但是在面临复杂神经网络的时候导数的计算就成为一个难题，要求人们解出复杂、高维的方程是不现实的。

这就是自动求导出现的原因，当前最流行的深度学习框架如PyTorch、Tensorflow等都提供了**自动微分**的支持，让人们只需要很少的工作就能神奇般地自动计算出复杂函数的梯度。

2. Autograd自动求导

20

`requires_grad`属性

`requires_grad`属性默认为False，也就是Tensor变量默认是不需要求导的。

如果一个节点的`requires_grad`是True，那么所有依赖它的节点`requires_grad`也会是True。

换言之，如果一个节点依赖的所有节点都不需要求导，那么它的`requires_grad`也会是False。在反向传播的过程中，该节点所在的子图会被排除在外。

2. Autograd自动求导

21

Function类

我们已经知道PyTorch使用动态计算图(DAG)记录计算的全过程,DAG的节点是Function对象,边表示数据依赖,从输出指向输入。因此Function类在PyTorch自动求导中位居核心地位,但是用户通常不会直接去使用。

每当对Tensor施加一个运算的时候,就会产生一个Function对象,它产生运算的结果,记录运算的发生,并且记录运算的输入。Tensor使用.grad_fn属性记录这个计算图的入口。反向传播过程中,autograd引擎会按照逆序,通过Function的backward依次计算梯度。

2. Autograd自动求导

22

backward函数

backward函数是反向传播的入口点，在需要被求导的节点上调用

backward函数会计算梯度值到相应的节点上。

backward函数是反向求导数，使用链式法则求导。

backward需要一个重要的参数grad_tensor，对非标量节点求导,需要指定grad_tensors，grad_tensors的shape必须和y的相同。

但如果节点只含有一个标量值，这个参数就可以省略（例如最普遍的loss.backward()与loss.backward(torch.tensor(1))等价）

2. Autograd自动求导

23

grad属性

backward函数本身没有返回值，它计算出来的梯度存放在叶子节点的grad属性中。

PyTorch文档中提到，如果grad属性不为空，新计算出来的梯度值会直接加到旧值上面。为什么不直接覆盖旧的结果呢？

这是因为有些Tensor可能有多个输出，那么就需要调用多个backward。叠加的处理方式使得backward不需要考虑之前有没有被计算过导数，只需要加上就行了，这使得设计变得更简单。

因此我们用户在反向传播之前，常常需要用zero_grad函数对导数手动清零，确保计算出来的是正确的结果。

2. Autograd自动求导

24

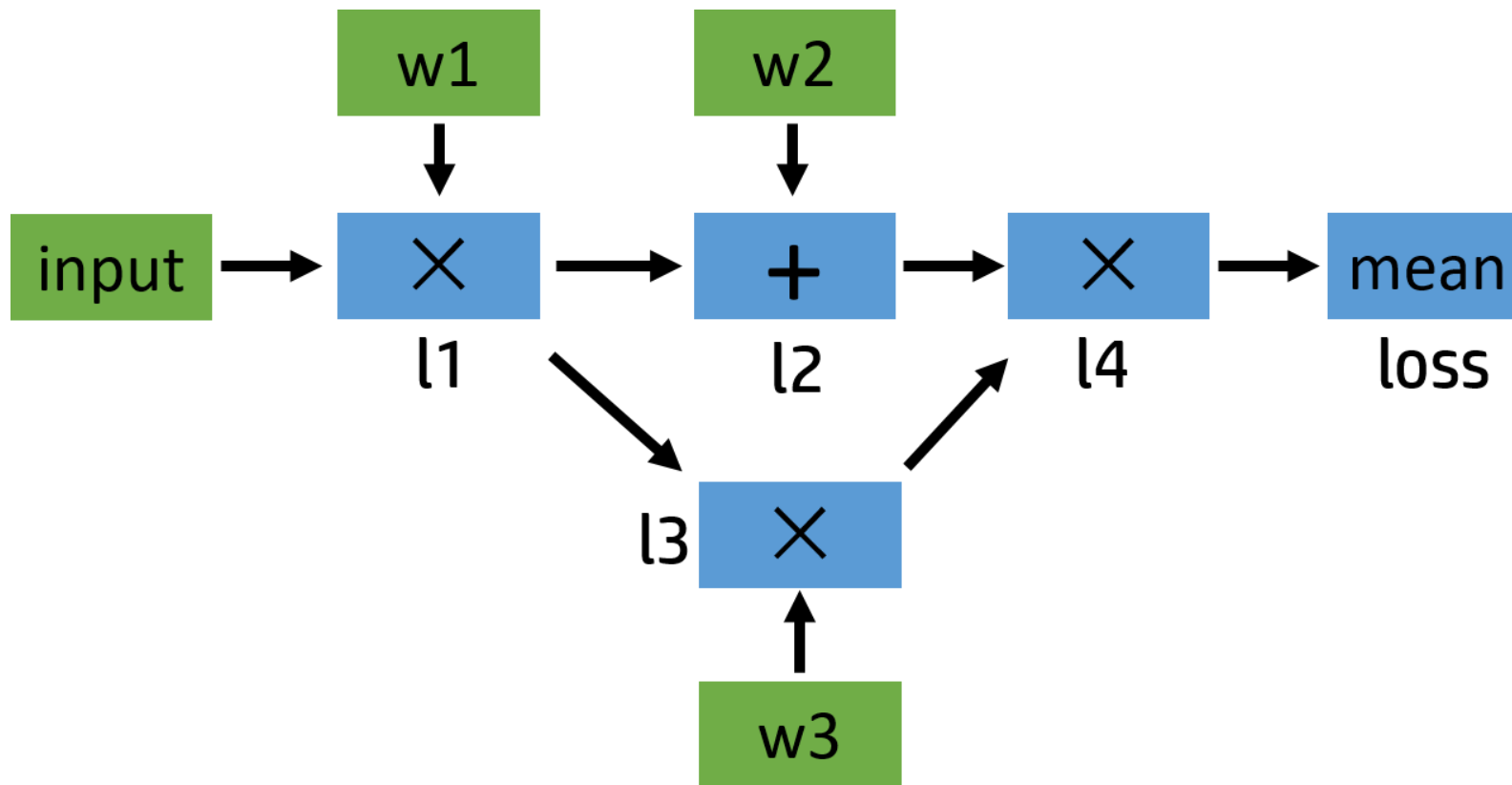
$l1 = \text{input} \times w1$

$l2 = l1 + w2$

$l3 = l1 \times w3$

$l4 = l2 \times l3$

$\text{loss} = \text{mean}(l4)$



2. Autograd自动求导

25

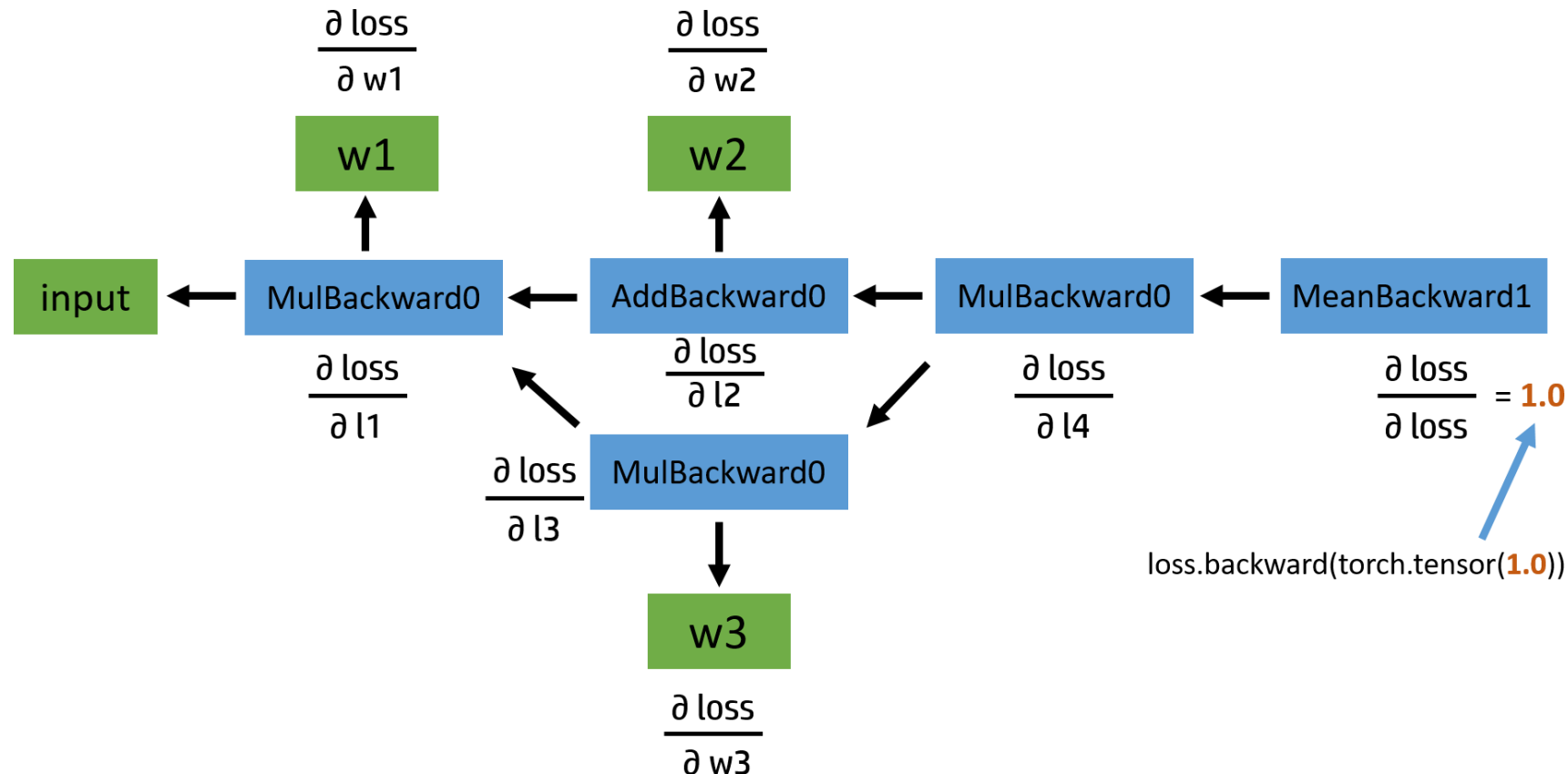
$l1 = \text{input} \times w1$

$l2 = l1 + w2$

$l3 = l1 \times w3$

$l4 = l2 \times l3$

$\text{loss} = \text{mean}(l4)$



只有input 一个变量是requires_grad=False 的

<https://zhuanlan.zhihu.com/p/69294347>

3. 神经网络

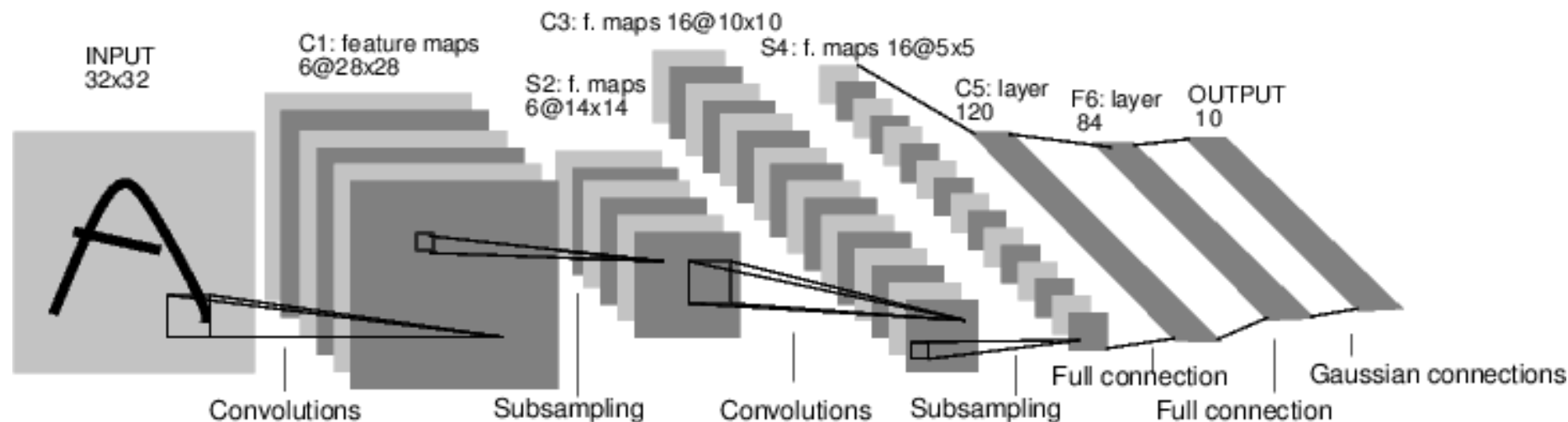
26

1. **Tensors**张量
2. **Autograd**自动求导
3. 神经网络
4. 训练一个分类器

3. 神经网络

27

可以使用torch.nn包来构建神经网络. 你已经知道autograd包,nn包依赖autograd包来定义模型并求导.一个nn.Module包含各个层和一个forward(input)方法,该方法返回output。

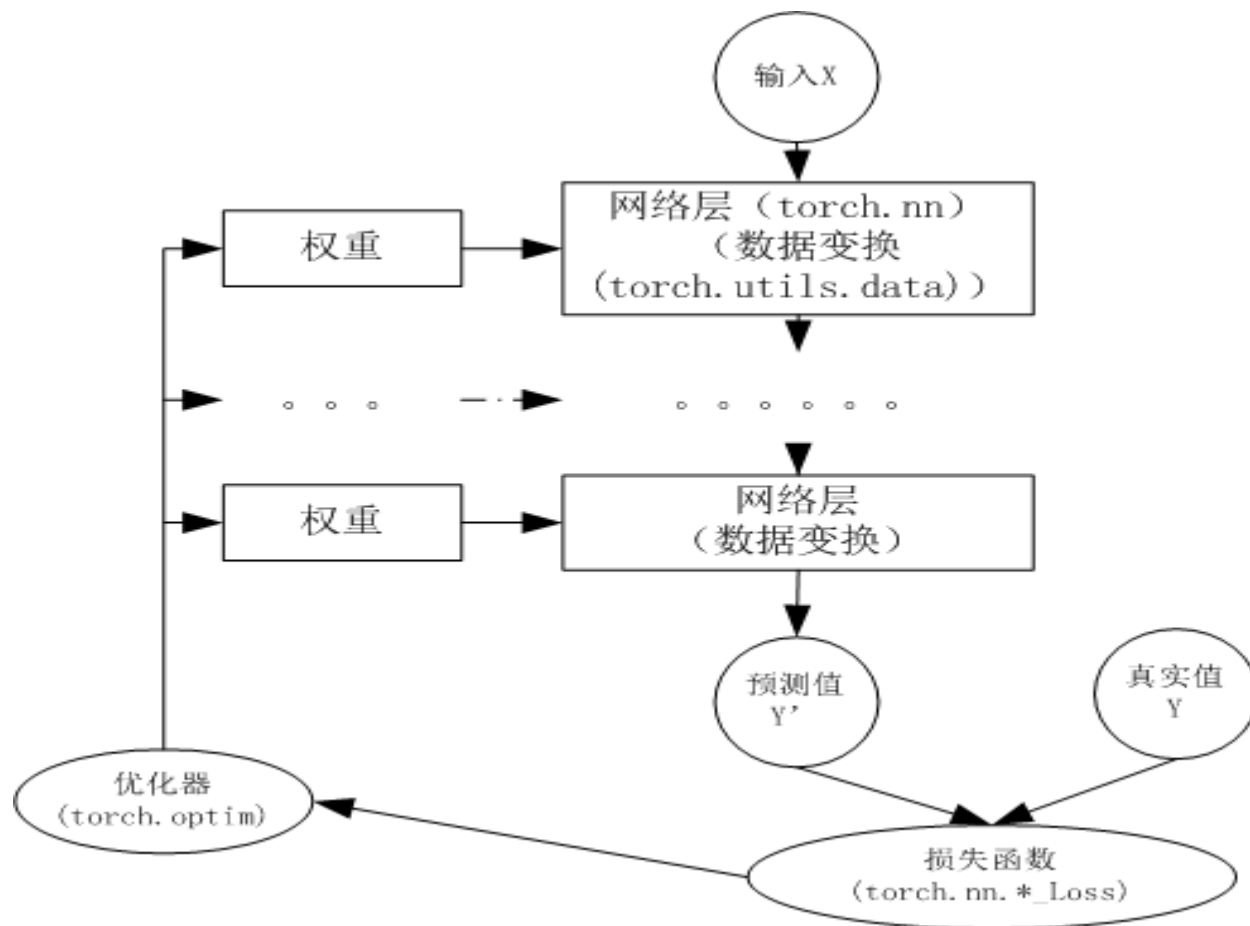


典型的神经网络

3. 神经网络

28

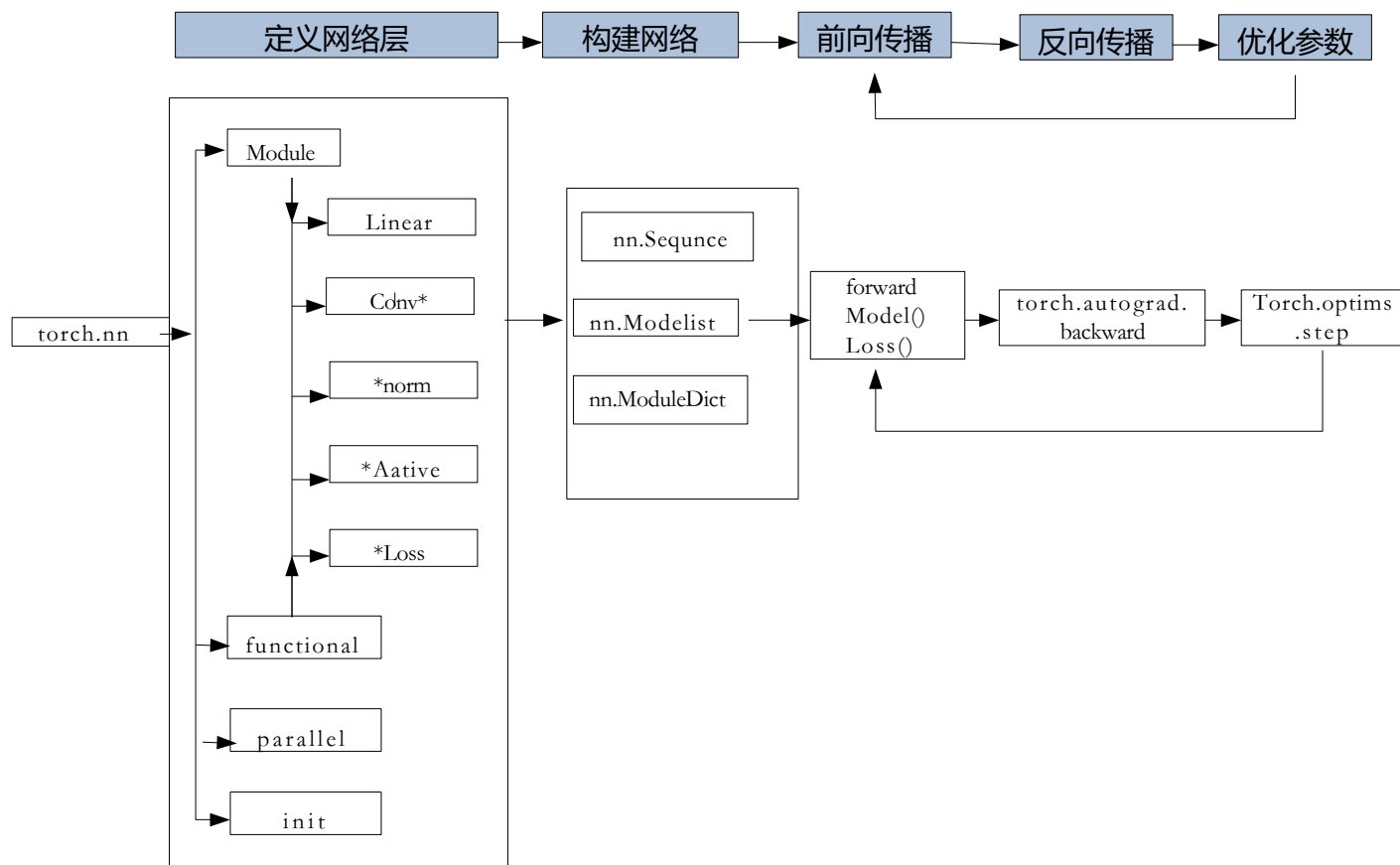
• 神经网络关键组件及相互关系



3. 神经网络

29

PyTorch构建网络工具



3. 神经网络

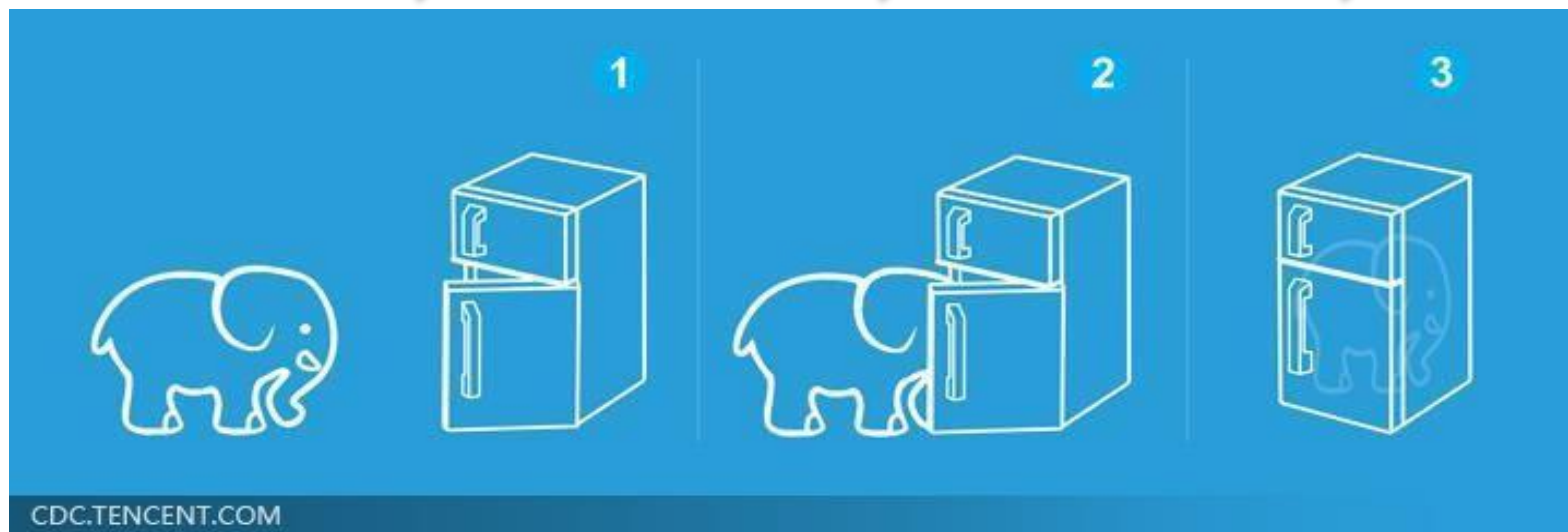
30

神经网络的典型训练过程如下:

- 定义神经网络模型,它有一些可学习的参数(或者权重);
- 在数据集上迭代;
- 通过神经网络处理输入;
- 计算损失(输出结果和正确值的差距大小)
- 将梯度反向传播回网络的参数;
- 更新网络的参数,主要使用如下简单的更新原则:
$$\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$$

深度学习的三个步骤

31



深度学习很简单.....

3. 神经网络

32

`torch.Tensor`-支持自动编程操作（如`backward()`）的多维数组。同时保持梯度的张量。

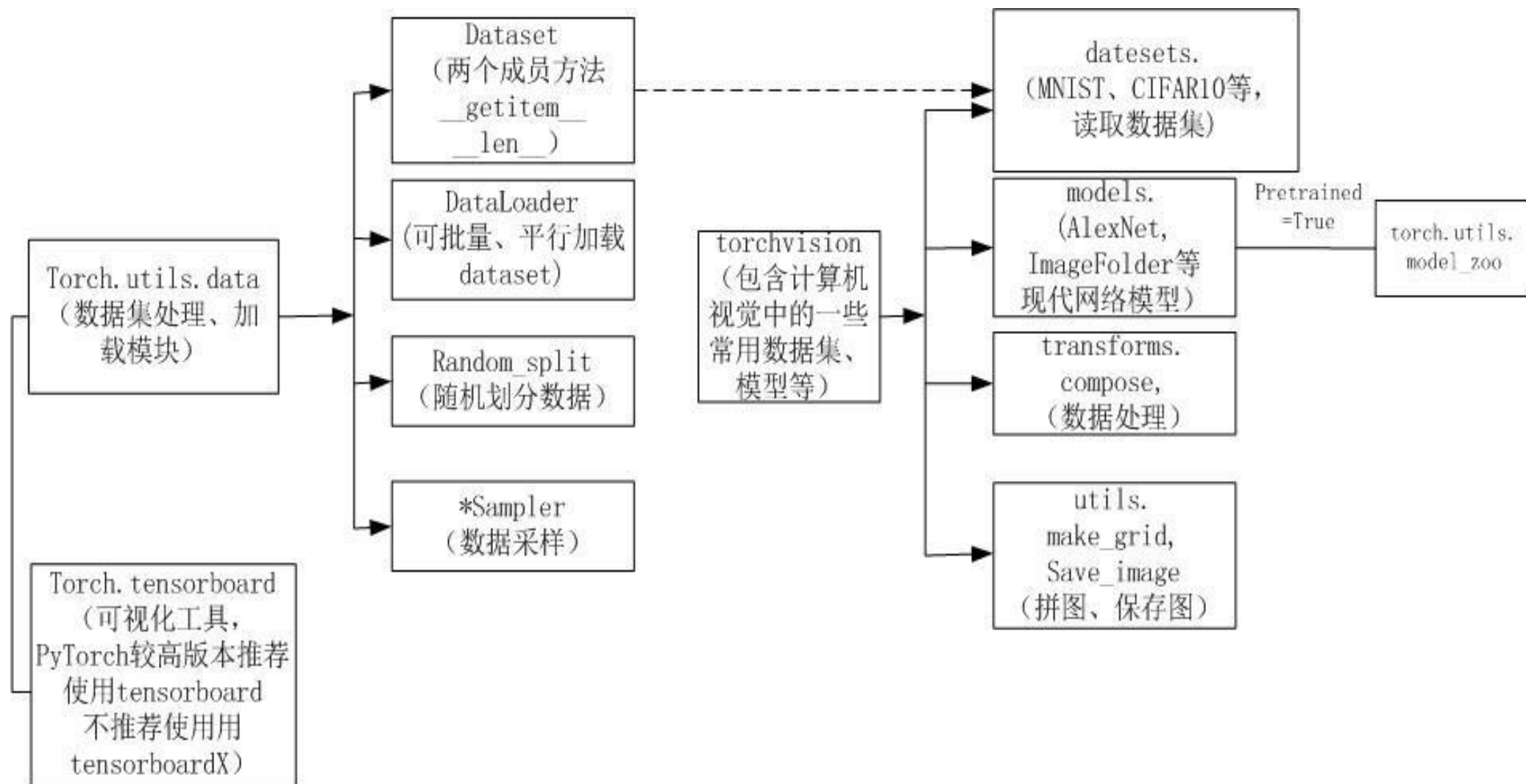
`nn.Module`-神经网络模块.封装参数,移动到GPU上运行,导出,加载等

`nn.Parameter`-一种张量,当把它赋值给一个Module时,被自动的注册为参数。

`autograd.Function`-实现一个自动求导操作的前向和反向定义, 每个张量操作都会创建至少一个Function节点, 该节点连接到创建张量并对其历史进行编码的函数。

数据处理工具箱

33



4. 训练一个分类器

34

1. **Tensors**张量
2. **Autograd**自动求导
3. 神经网络
4. 训练一个分类器

4. 训练一个分类器

35

训练一个分类器流程

加载训练集和测试集

定义一个卷积神经网络

定义损失函数

在训练集上训练网络

在测试集上测试网络

4. 训练一个分类器

36

torch.nn.Linear

PyTorch的`nn.Linear ()` 是用于设置网络中的全连接层的，需要注意的是全连接层的输入与输出都是二维张量，一般形状为`[batch_size, size]`，不同于卷积层要求输入输出是四维张量。

`in_features`指的是输入的二维张量的大小，即输入的`[batch_size, size]`中的`size`。

`out_features`指的是输出的二维张量的大小，即输出的二维张量的形状为`[batch_size, output_size]`，当然，它也代表了该全连接层的神经元个数。

从输入输出的张量的`shape`角度来理解，相当于一个输入为`[batch_size, in_features]`的张量变换成了`[batch_size, out_features]`的输出张量。

4. 训练一个分类器

37

torch.nn

计算图和autograd是十分强大的工具，可以定义复杂的操作并自动求导；然而对于大规模的网络，autograd太过于底层。在构建神经网络时，我们经常考虑将计算安排成层，其中一些具有可学习的参数，它们将在学习过程中进行优化。

TensorFlow里，有类似Keras，TensorFlow-Slim和TFLearn这种封装了底层计算图的高度抽象的接口，这使得构建网络十分方便。

在PyTorch中，包nn 完成了同样的功能。nn包中定义一组大致等价于层的模块。一个模块接受输入的tensor，计算输出的tensor，而且还保存了一些内部状态比如需要学习的tensor的参数等。nn包中也定义了一组损失函数（loss functions），用来训练神经网络。

4. 训练一个分类器

38

torch.optim

使用optim包定义优化器 (Optimizer) 。Optimizer将会为我们更新模型的权重。

这里我们使用Adam优化方法; optim包还包含了许多别的优化算法。

Adam构造函数的第一个参数告诉优化器应该更新哪些张量。

learning_rate = 1e-4

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

1. IAN GOODFELLOW等, 《深度学习》, 人民邮电出版社, 2017
2. Andrew Ng, <http://www.deeplearning.ai>
3. Christopher M. Bishop, Pattern Recognition and Machine Learning, Springer-Verlag, 2006
4. 李宏毅, 《一天搞懂深度学习》
5. 吴茂贵等, 《Python深度学习基于PyTorch》, 机械工业出版社, 2020