Note Information

Author : AbbasXu
Date : 2022-08-16
Title : pytorch基础
Keywords : #pytroch #张量 #AutoGrad #并行计算

张量

简介

几何代数中定义的张量是基于向量和矩阵的推广， 从某种角度来说，张量就是一个数据容器
张量维度	代表含义
0维张量	代表的是标量（数字）
1维张量	代表的是向量
2维张量	代表的是矩阵
3维张量	时间序列数据 股价 文本数据 单张彩色图片(RGB)
除此之外，在我们机器学习的工作中，我们通常需要处理的不止是一张图或者是一段话，而是一组（集合），如下所示：
一张图的表达式为`(width, height, channel) = 3D`，而一组图的表达式为`(batch_size, width, height, channel) = 4D`

张量和Numpy的多维数组比较类似，但是`torch.tensor`提供了GPU计算和自动求梯度等更多功能，这些使 Tensor 这一数据类型更加适合深度学习。

PyTorch中的张量一共支持9种数据类型，每种数据类型都对应**CPU**和**GPU**的两种子类型，如下表所示
数据类型	PyTorch类型	CPU上的张量	GPU上的张量
32位浮点数	torch.float32/torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64位浮点数	torch.float64/torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
16位浮点数	torch.float16/torch.half	N/A	torch.cuda.HalfTensor
8位无符号整数	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8位带符号整数	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16位带符号整数	torch.int16/torch.short	torch.ShortTensor	torch.cuda.ShortTensor

32位带符号整数	torch.int32/torch.int	torch.IntTensor	torch.cuda.IntTensor
64位带符号整数	torch.int64/torch.long	torch.LongTensor	torch.cuda.LongTensor
布尔型	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

创建tensor

[Pytorch 用户手册-tensor]
常用的`tensor`构建方法
函数	功能
Tensor(sizes)	基础构造函数
tensor(data)	类似于np.array
ones(sizes)	全1
zeros(sizes)	全0
eye(sizes)	对角为1，其余为0
arange(s,e,step)	从s到e，步长为step
linspace(s,e,steps)	从s到e，均匀分成step份
rand/randn(sizes)	rand是[0,1)均匀分布；randn是服从N(0，1)的正态分布
normal(mean,std)	正态分布(均值为mean，标准差是std)
randperm(m)	随机排列

张量操作

加法

`x+y` \ `torch.add(x+y)` \ `y.add_(x)` (原值修改)

索引

(类似于numpy)，注意得到的结果与原值共享内存，一动都动，不想修改则使用`copy()`。

维度变化

常见的方法有`torch.view()`和`torch.reshape()`，其中view也是与原值共享内存。
`torch.reshape()`不保证返回拷贝值，因此常用的方法是通过`clone()`创建张量副本。

统计量计算

torch.mean(t)	返回张量均值
torch.var(t)	返回张量方差
torch.std(t)	返回张量标准差
torch.var_mean(t)	返回张量方差和均值
torch.std_mean(t)	返回张量标准差和均值
torch.max(t)	返回张量最大值
torch.argmax(t)	返回张量最大值索引
torch.min(t)	返回张量最小值
torch.argmin(t)	返回张量最小值索引
torch.median(t)	返回张量中位数
torch.sum(t)	返回张量求和结果
torch.logsumexp(t)	返回张量各元素求和结果，适用于数据量较小的情况
torch.prod(t)	返回张量累乘结果
torch.dist(t1, t2)	计算两个张量的闵式距离，可使用不同范式
torch.topk(t)	返回t中最大的k个值对应的指标

矩阵计算


Tensor矩阵运算
函数
描述
torch.t(t)
t转置
torch.eye(n)
创建包含n个分量的单位矩阵
torch.diag(t1)
以t1中各元素，创建对角矩阵
torch.triu(t)
取矩阵t中的上三角矩阵
torch.tril(t)
取矩阵t中的下三角矩阵
CSDN @奔跑的林小川

线性代数运算


矩阵的线性代数运算
函数
描述
torch.trace(A)
矩阵的迹
matrix_rank(A)
矩阵的秩
torch.det(A)
计算矩阵A的行列式
torch.inverse(A)
矩阵求逆
torch.lstsq(A,B)
最小二乘法
CSDN @奔跑的林小川

广播机制

当对两个形状不同的 Tensor 按元素运算时，可能会触发广播(broadcasting)机制：先适当复制元素使这两个 Tensor 形状相同后再按元素运算。

```
x = torch.arange(1, 3).view(1, 2)
print(x)
y = torch.arange(1, 4).view(3, 1)
print(y)
print(x + y)
```

结果：

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

注：广播运算解决张量维度不同的问题，在张量分向量不同时，不同的分量中，有一个需要为1

自动求导

AutoGrad

`torch.Tensor` 是这个包的核心类。如果设置它的属性 `.requires_grad` 为 `True`，那么它将会追踪对于该张量的所有操作。当完成计算后可以通过调用 `.backward()`，来自动计算所有的梯度。这个张量的所有梯度将会自动累加到 `.grad` 属性。其支持对任意计算图的自动梯度计算。(默认False)

- 计算图是由节点和边组成的，其中的一些节点是数据，一些是数据之间的运算
- 计算图实际上就是变量之间的关系
- tensor 和 function 互相连接生成的一个有向无环图

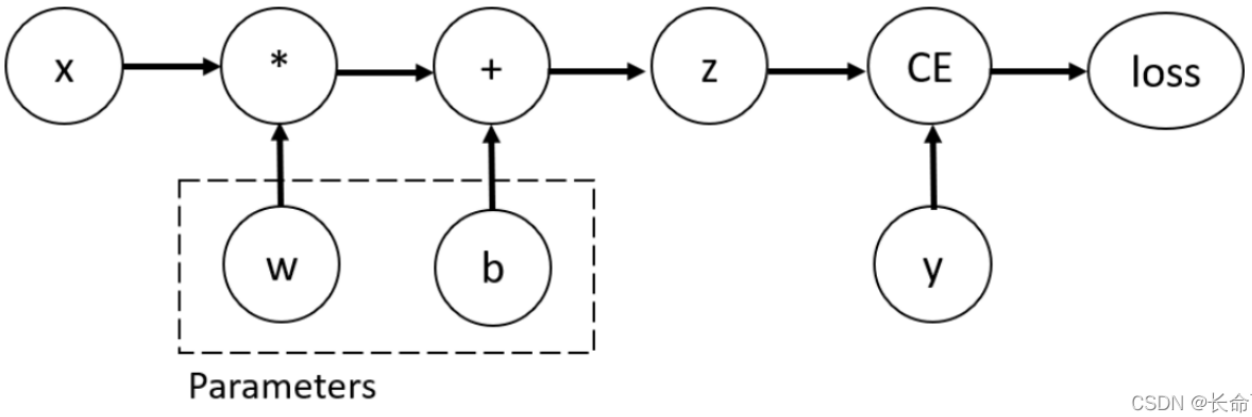
一个简单的例子

```
import torch

x = torch.ones(5)  # input tensor
```

```
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b #x 和 w 矩阵相乘，再加上 bias b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

其计算图如下所示



梯度

tensor tensor求梯度的基础属性

- `requires_grad`表示该tensor是否支持梯度计算。如果`.requires_grad`为 True，那么它会追踪该tensor的所有计算操作。
- `grad`记录该tensor的梯度。在tensor完成计算后，调用`.backward()`计算出所有梯度，同时累加到.grad属性中。
- `grad_fn`会记录作用在该tensor上的计算`Function`。如果该tensor是用户创建而非通过运算得出的，该tensor的`.grad_fn`就是None(如下面的例子所示)
- `is_leaf`:如果一个tensor是用户创建而非用过运算得出，那么该tensor在无环图中就是一个叶子节点，`.is_leaf=True`。
- 引进叶子节点概念的目的：是为了节约内存，在反向传播结束后，非叶子节点的梯度默认会被释放掉，不会记录。

在**Pytorch**中, 反向传播是依靠`.backward()`实现的。
可以通过`.detach()`获得一个新的Tensor, 拥有相同的内容但不需要自动求导。

并行计算

意义

让多个GPU来参与训练，减少训练时间。

CUDA

在编写程序中，当我们使用了`.cuda()`时，其功能是让我们的模型或者数据从CPU迁移到GPU(0)当中，通过GPU开始计算。

- 注：
  - 我们使用GPU时使用的是.cuda()而不是使用.gpu()。这是因为当前GPU的编程接口采用CUDA，但是市面上的GPU并不是都支持CUDA，只有部分NVIDIA的GPU才支持，AMD的GPU编程接口采用的是OpenCL，在现阶段PyTorch并不支持。
  - 数据在GPU和CPU之间进行传递时会比较耗时，我们应当尽量避免数据的切换。
  - GPU运算很快，但是在使用简单的操作时，我们应该尽量使用CPU去完成。
  - 当我们的服务器上有多个GPU，我们应该指明我们使用的GPU是哪一块，如果我们不设置的话，tensor.cuda()方法会默认将tensor保存到第一块GPU上，等价于tensor.cuda(0)，这将会导致爆出out of memory的错误。我们可以通过以下两种方式继续设置。

```
1、#设置在文件最开始部分
import os
os.environ["CUDA_VISIBLE_DEVICE"] = "2" # 设置默认的显卡
2、
CUDA_VISBLE_DEVICE=0,1 python train.py # 使用0，1两块GPU
```

常见并行方法

- 网络结构分布到不同的设备中(Network partitioning)
- 同一层的任务分布到不同数据中(Layer-wise partitioning)
- 不同的数据分布到不同的设备中，执行相同的任务(Data parallelism)