pytorch基础知识

- pytorch基础知识
  - 张量
    - 创建tensor
    - 常见张量操作（类比numpy）
    - 广播机制
  - 自动求导
    - Autograd 简介
    - 并行计算简介
      - 常见的并行的方法

> PyTorch是由Facebook人工智能研究小组开发的一种基于Lua编写的**Torch**库的Python实现的深度学习库，目前被广泛应用于学术界和工业界

`torch.Tensor`:**存储和变换数据的主要工具**

- Tensor和Numpy的**多维数组**非常类似，除此，Tensor还提供了**GPU计算和自动求梯度**等功能。

张量

创建tensor

1.`随机初始化矩阵`

- **torch.rand()**
-

2.`全0矩阵的构建`

- torch.zeros(n,m,dtype=torch.datatype(eg:**long**))
- 可以通过torch.zero_()和torch.zeros_like()将现有矩阵转为全0矩阵
  3.`张量的构建`
- torch.tensor()

```
x = x.new_ones(4, 3, dtype=torch.double)
# 创建一个新的全1矩阵tensor，返回的tensor默认具有相同的torch.dtype和torch.device
# 也可以像之前的写法 x = torch.ones(4, 3, dtype=torch.double)
print(x)
x = torch.randn_like(x, dtype=torch.float)
# 重置数据类型
print(x)
```

```
# 结果会有一样的size
# 获取它的维度信息
print(x.size())
print(x.shape)
```

常见张量操作（类比numpy）

1. 维度变换
   torch.view() / torch.reshape()
   其中，**torch.view()**会改变原始张量，**推荐的方法**是我们先用 **clone()** 创造一个张量副本然后再使用 **torch.view()** 进行函数维度变换 。（torch.shape()并不能保证返回的是拷贝值）

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。

自动求导

> PyTorch 中，所有神经网络的核心是 autograd 包。autograd包为张量上的所有操作提供了自动求导机制。它是一个在运行时定义（define-by-run）的框架，这意味着反向传播是根据代码如何运行来决定的，并且每次迭代可以是不同的。

Autograd 简介

- `设置属性`：**.requires_grad :True** →将会追踪对于该张量的所有操作
- `计算梯度`：**.backward()** →这个张量的所有梯度将会自动累加到 **.grad** 属性
  PS： 如果 y 不是标量，需要传入一个与 y 同形的Tensor。
- `阻止一个张量被跟踪历史`：**.detach()**

> 为了防止跟踪历史记录(和使用内存），可以将代码块包装在 with torch.no_grad(): 中

注意：grad在反向传播过程中是累加的(accumulated)，这意味着每一次运行反向传播，梯度都会累加之前的梯度，所以一般在反向传播之前需把梯度清零。

- 再来反向传播一一次，注意grad是累加的

```
out2 = x.sum()
out2.backward()
print(x.grad)

out3 = x.sum()
x.grad.data.zero_()
out3.backward()
print(x.grad)
```

> 如果我们想要修改 tensor 的数值，但是又不希望被 autograd 记录(即不会影响反向传播)， 那么我们可以对 tensor.data 进行操作。

```
x = torch.ones(1,requires_grad=True)

print(x.data) # 还是一个tensor
print(x.data.requires_grad) # 但是已经是独立于计算图之外

y = 2 * x
x.data *= 100 # 只改变了值，不会记录在计算图，所以不会影响梯度传播

y.backward()
print(x) # 更改data的值也会影响tensor的值
print(x.grad)
```

并行计算简介

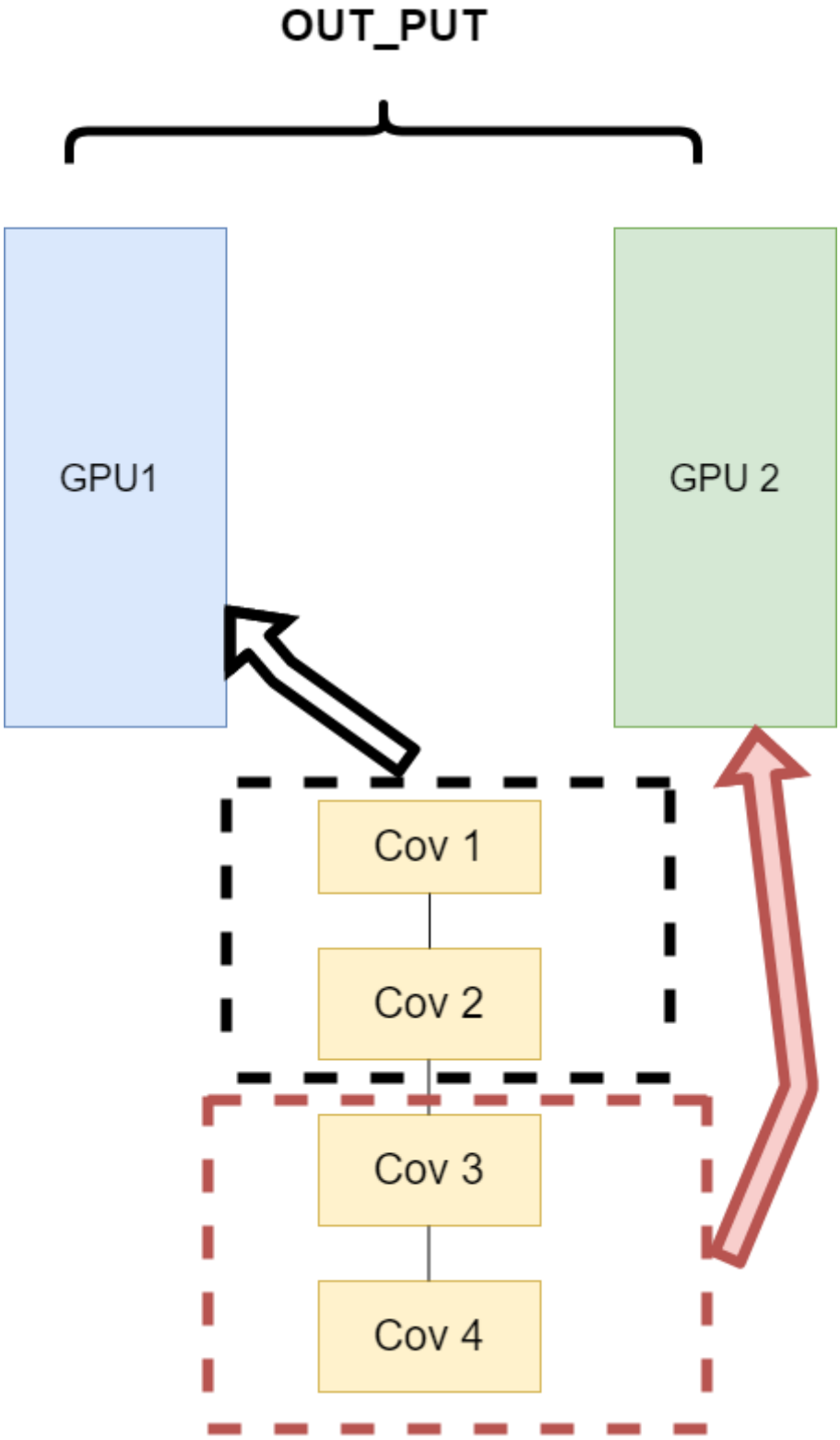
> 我们使用GPU时使用的是.cuda()而不是使用.gpu()。这是因为当前GPU的编程接口采用CUDA，但是市面上的GPU并不是都支持CUDA，只有部分NVIDIA的GPU才支持，AMD的GPU编程接口采用的是OpenCL，在现阶段PyTorch并不支持。

ps: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。

```
#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
CUDA_VISBLE_DEVICE=0,1 python train.py # 使用
0，1两块GPU
```

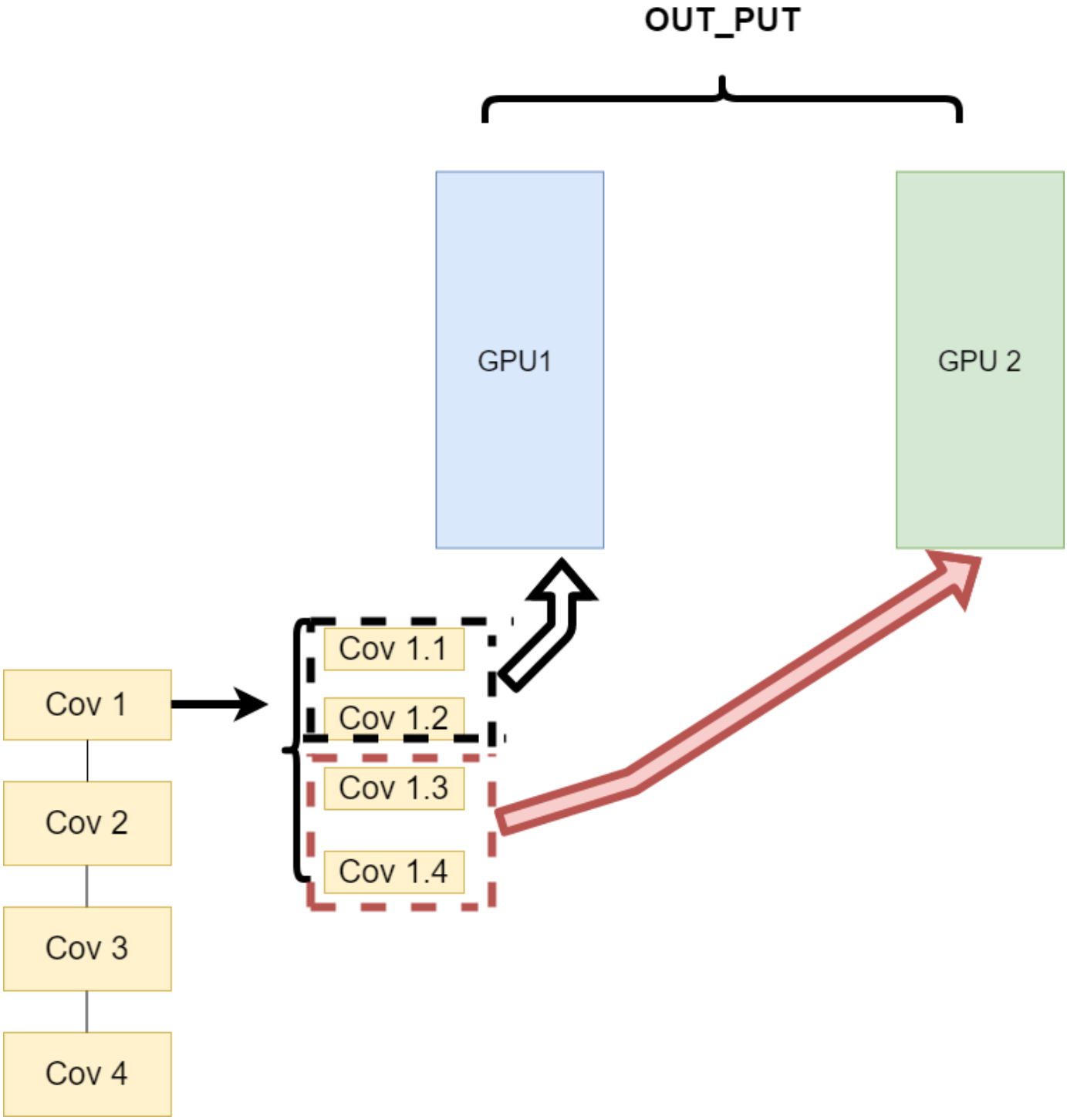
常见的并行的方法

1.**网络结构分布到不同的设备中(Network partitioning)**



> 这里遇到的问题就是，不同模型组件在不同的GPU上时，GPU之间的传输就很重要，对于GPU之间的通信是一个考验。但是GPU的通信在这种密集任务中很难办到，所以这个方式慢慢淡出了视野

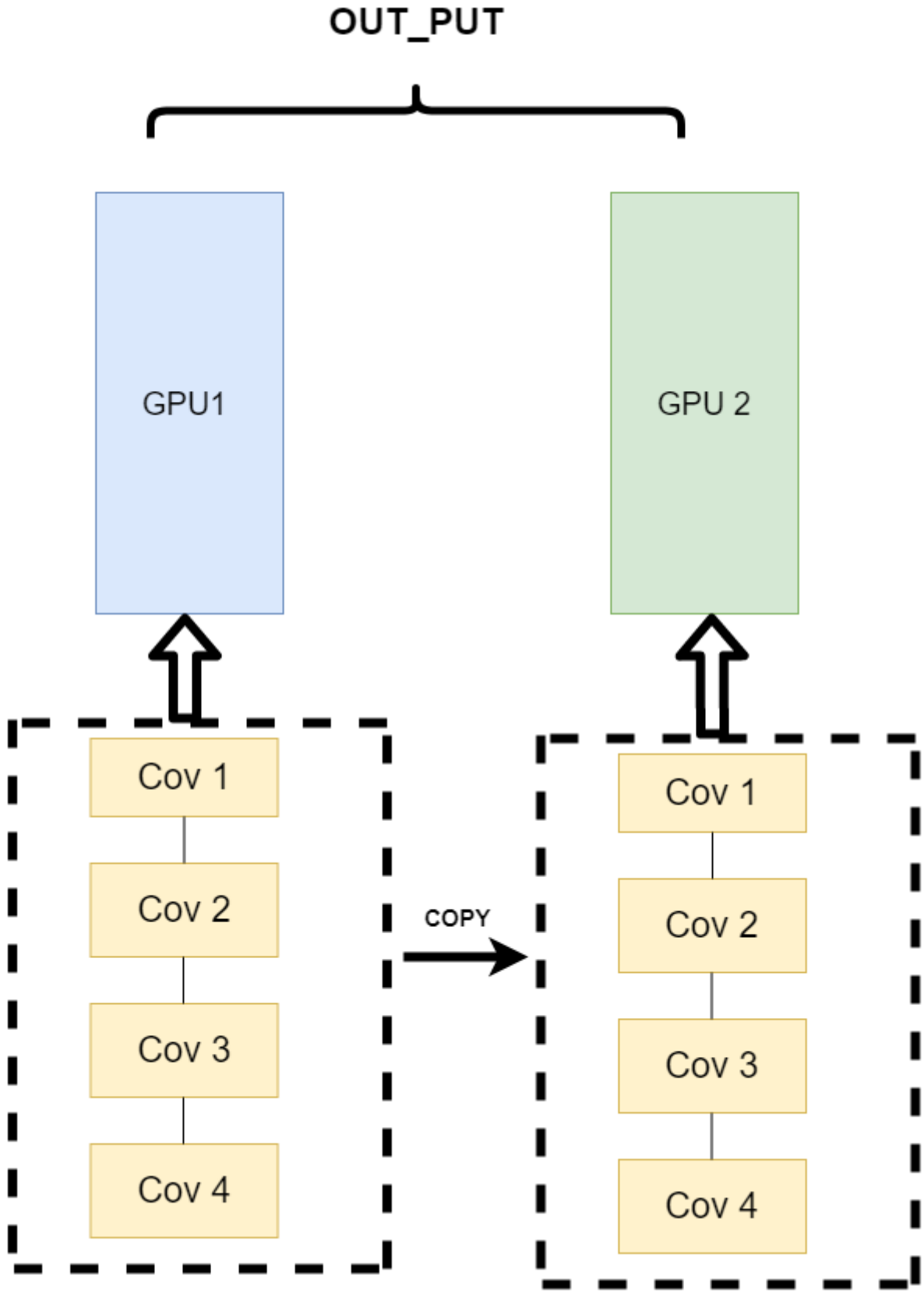
2.**同一层的任务分布到不同数据中(Layer-wise partitioning)**



第二种方式就是，同一层的模型做一个拆分，让不同的GPU去训练同一层模型的部分任务。

这样可以保证在不同组件之间传输的问题，但是在我们需要大量的训练，同步任务加重的情况下，会出现和第一种方式一样的问题。

3. **不同的数据分布到不同的设备中，执行相同的任务(Data parallelism)**



第三种方式有点不一样，它的逻辑是，我不再拆分模型，我训练的时候模型都是一整个模型。但是我将输入的数据拆分。所谓的拆分数据就是，同一个模型在不同GPU中训练一部分数据，然后再分别计算一部分数据之后，只需要将输出的数据做一个汇总，然后再反传。