训练预热 Warmup

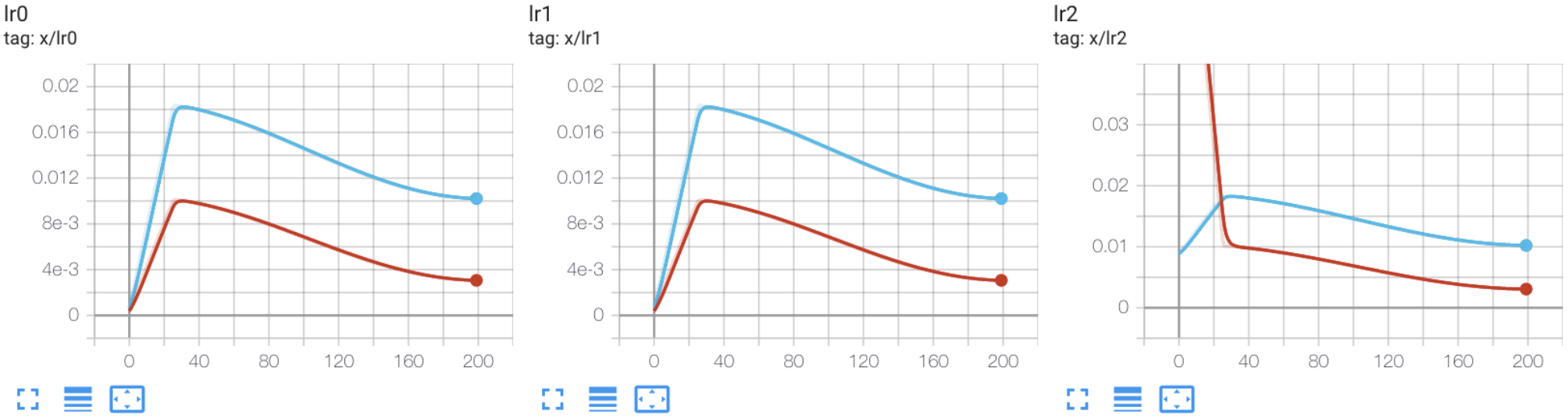
由于刚开始训练时，模型的权重(weights)是随机初始化的，此时若选择一个较大的学习率，可能带来模型的不稳定（振荡），选择Warmup预热学习率的方 式，可以使得开始训练的几个epoch或者一些step内学习率较小，在预热的小学习率下，模型可以慢慢趋于稳定，等模型相对稳定后在选择预先设置的学习率进行训练，使得模型收敛速度变得更快，模型效果更佳。

训练预热 Warmup

腾讯课堂

Warmup slowly ramps training parameters from their initial (more stable) values to their default training values. It's common to ramp learning rate for example from 0 to some initial value over the first few epochs to avoid early training instabilities, nan's, etc.

Warmup effects are visible in Tensorboard learning rate plots, which are automatically tracked. Example below shows about 30 epochs of warmup on a custom dataset, one plot per parameter group. Last plot shows different warmup strategies (i.e. different hyps).



余弦退火调整学习率 CosineAnnealingLR

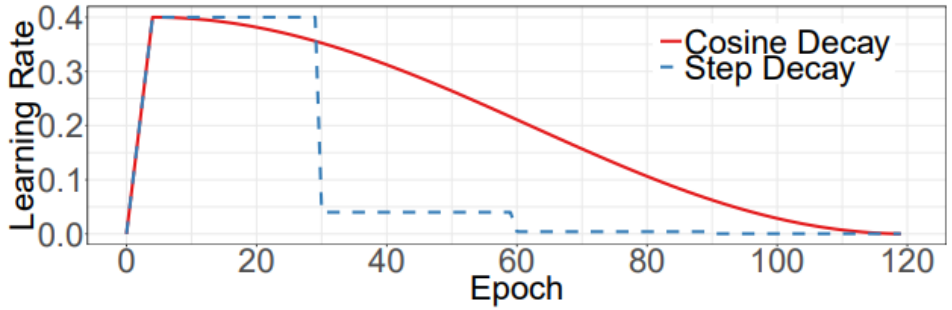
https://arxiv.org/pdf/1608.03983.pdf

Within the i-th run, we decay the learning rate with a cosine annealing for each batch as follows:

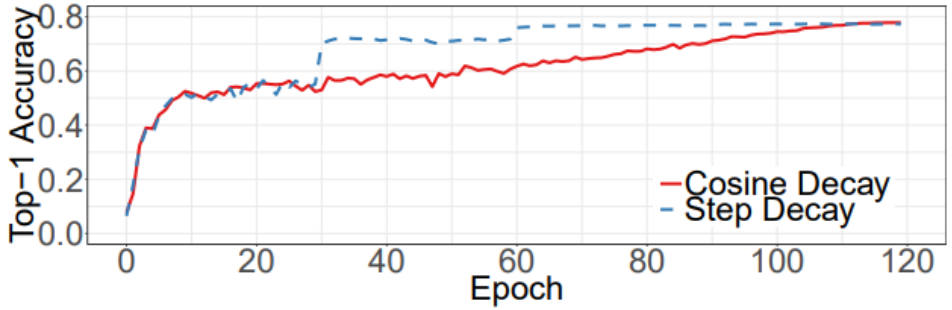
$$\eta_t = \eta_{min}^i + \frac{1}{2}(\eta_{max}^i - \eta_{min}^i)(1 + \cos(\frac{T_{cur}}{T_i}\pi)), \tag{5}$$

where η_{min}^i and η_{max}^i are ranges for the learning rate, and T_{cur} accounts for how many epochs have been performed since the last restart. Since T_{cur} is updated at each batch iteration t, it can take discredited values such as 0.1, 0.2, etc. Thus, $\eta_t = \eta_{max}^i$ when $t = 0$ and $T_{cur} = 0$. Once $T_{cur} = T_i$, the cos function will output -1 and thus $\eta_t = \eta_{min}^i$.

余弦退火调整学习率 CosineAnnealingLR



(a) Learning Rate Schedule



(b) Validation Accuracy

Figure 3: Visualization of learning rate schedules with warm-up. Top: cosine and step schedules for batch size 1024. Bottom: Top-1 validation accuracy curve with regard to the two schedules.

自动计算锚框

Autoanchor

Anchor给出了目标宽高的初始值，需要回归的是目标真实宽高与初始宽高的偏移量，而不使用anchor的做法需要回归宽高的绝对量。

Autoanchor only runs when the best possible recall (BPR，最大可能召回率) is under threshold

AutoAnchor runs before training to ensure your anchors are a good fit for your data. If they are not, then new anchors are computed and evolved and attached to your model automatically. No action is needed on your part. You can disable autoanchor with python train.py --noautoanchor.

AutoAnchor will attach anchors automatically to your model.pt file (i.e. last.pt or best.pt).

AutoAnchor displays the computed anchors on screen and advises you to update your yaml if you'd like to use them in the future.

超参数进化

hyperparameter evolution

Hyperparameter evolution is a method of Hyperparameter Optimization using a Genetic Algorithm (GA) for optimization.

Hyperparameters in ML control various aspects of training, and finding optimal values for them can be a challenge.

Traditional methods like grid searches can quickly become intractable due to

1) the high dimensional search space
2) unknown correlations among the dimensions
3) expensive nature of evaluating the fitness at each point, making GA a suitable candidate for hyperparameter searches.

自动混合精度训练
Automatic mixed precision (AMP) training

AMP is enabled by default for all model training on GPU.

All YOLOv5 checkpoints are saved in FP16. All GPU inference is performed in FP16.

混合精度预示着有不止一种精度的Tensor，在PyTorch的AMP模块里有2种：torch.FloatTensor和torch.HalfTensor。

- torch.HalfTensor的优势就是存储小、计算快、更好的利用CUDA设备的Tensor Core。因此训练的时候可以减少显存的占用（可以增加batchsize了），同时训练速度更快；
- torch.HalfTensor的劣势就是：数值范围小（更容易Overflow / Underflow）、舍入误差（Rounding Error，导致一些微小的梯度信息达不到16bit精度的最低分辨率，从而丢失）。

自动预示着Tensor的dtype类型会自动变化，也就是框架按需自动调整tensor的dtype（其实不是完全自动，有些地方还是需要手工干预）

Automatic mixed precision (AMP) training

腾讯课堂

如何在PyTorch中使用自动混合精度？

使用autocast + GradScaler。

1. autocast

使用torch.cuda.amp模块中的autocast 类。当进入autocast的上下文后，可支持AMP的CUDA ops 会把tensor的dtype转换为半精度浮点型，从而在不损失训练精度的情况下加快运算。刚进入autocast的上下文时，tensor可以是任何类型，不需要在model或者input上手工调用.half()，框架会自动做，这也是自动混合精度中“自动”一词的由来。另外一点就是，autocast上下文应该只包含网络的前向过程（包括loss的计算），而不要包含反向传播，因为BP的op会使用和前向op相同的类型。

2. GradScaler

使用torch.cuda.amp.GradScaler，需要在训练最开始之前实例化一个GradScaler对象。通过放大loss的值来防止梯度的underflow（这只是BP的时候传递梯度信息使用，真正更新权重的时候还是要把放大的梯度再unscale回去）。

断点续训

You use --resume by itself with no arguments, or by pointing to a last.pt to resume from:

```
python train.py --resume  # resume from most recent last.pt
```

```
python train.py --resume runs/exp0/weights/last.pt  # resume from specific weights
```

腾讯课堂

Multi-GPU Training

There are three ways to learn multi-GPU with PyTorch.

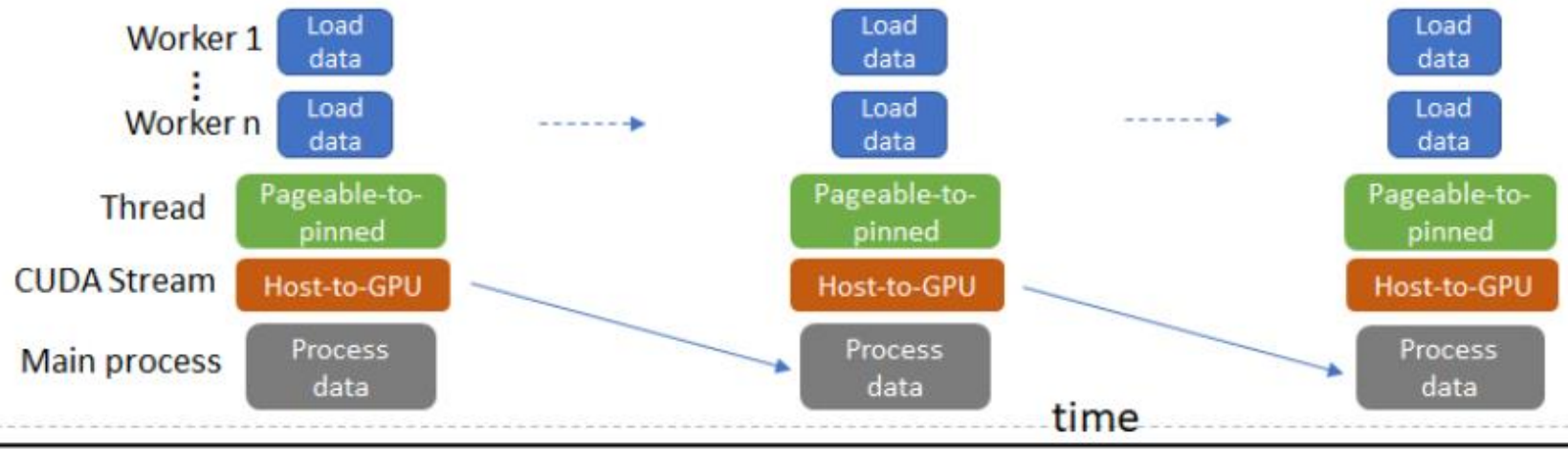
- **DataParallel**
- **Custom DataParallel**
- **Distributed DataParallel**

DataParallel is the most basic method provided by PyTorch, but there is GPU memory imbalance issues.

Custom DataParallel solves the problem of GPU memory to some extent, but the problem is that it does not utilize the GPU properly.

Distributed DataParallel is a feature of PyTorch that was originally created for distributed learning, but it can also be used for multi-GPU learning, without memory imbalance issues and inability to utilize the GPU.

Parallelizing data loading



Pipeline data loading and host-to-device transfer of next batch with processing current batch

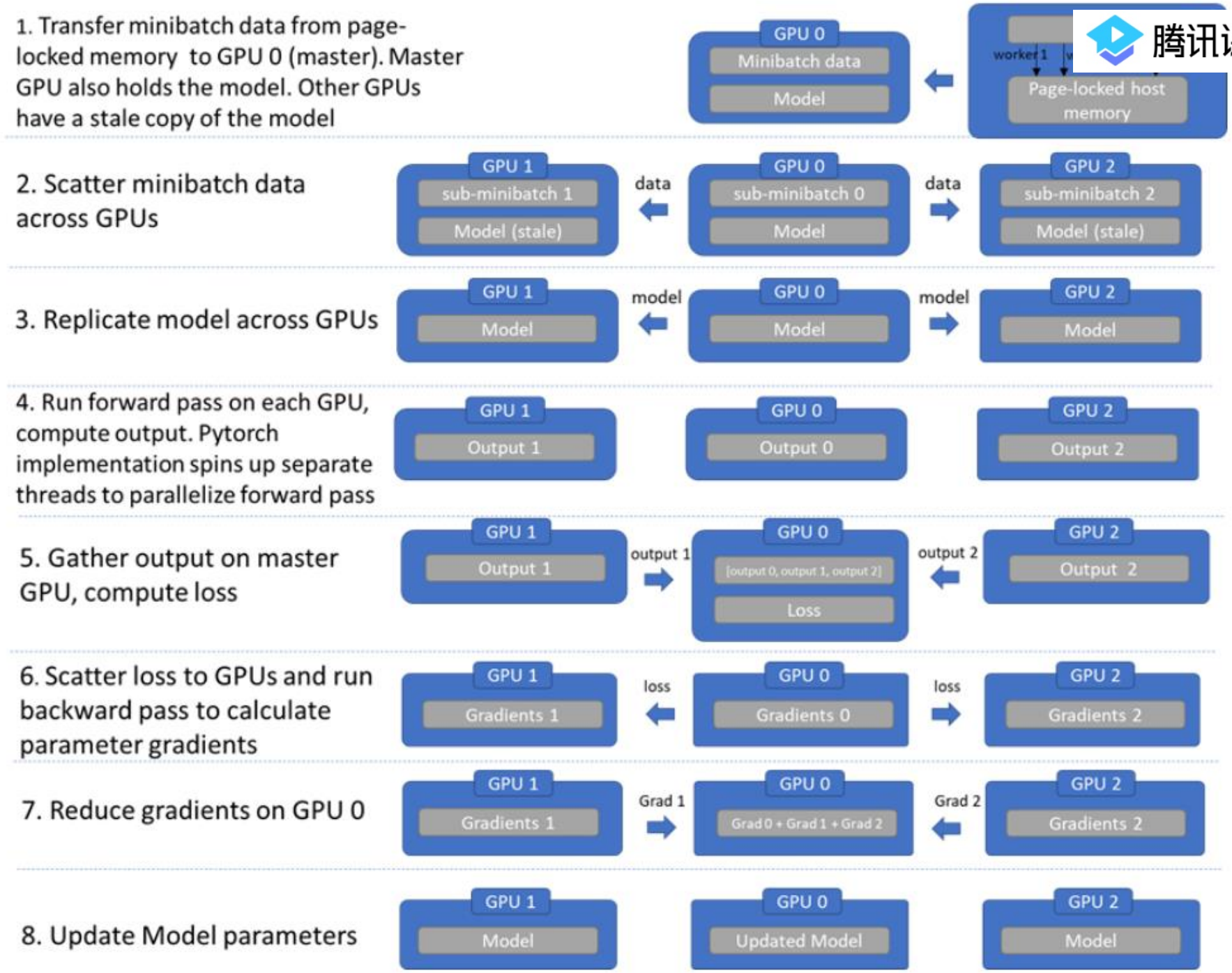
Data Parallel

One GPU (0) acts as the master GPU and coordinates data transfer.

Implemented in PyTorch data_parallel module

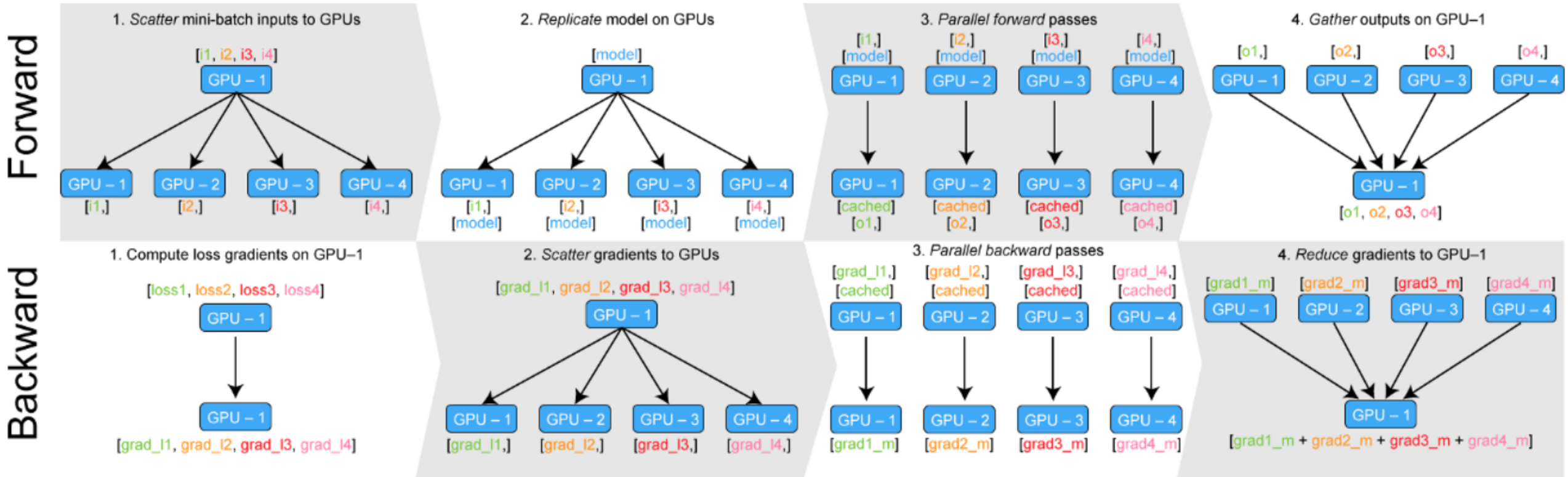
用于单机多卡，

不适用多机多卡

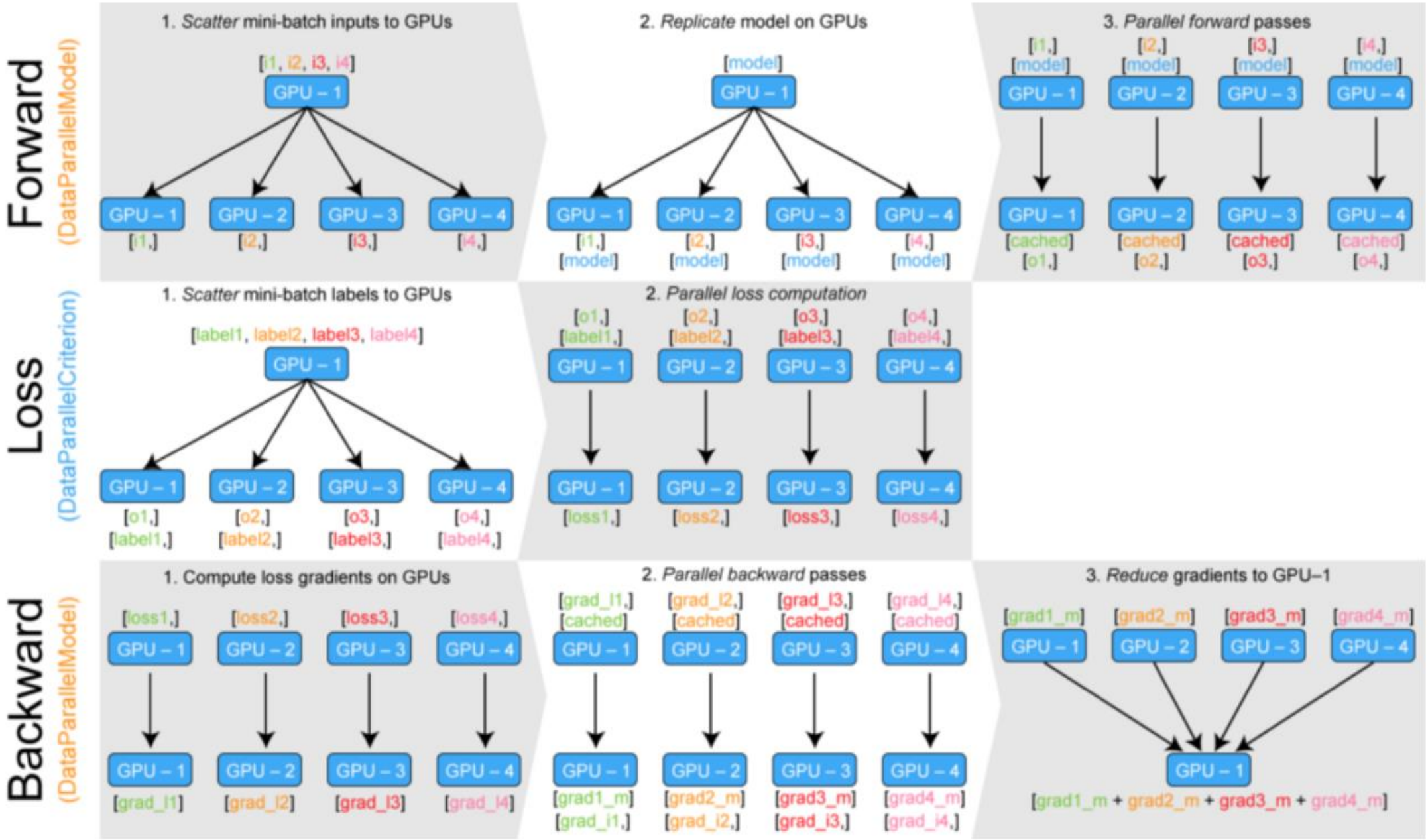


腾讯课堂

腾讯课堂


Forward
1. Scatter mini-batch inputs to GPUs
[i1, i2, i3, i4]
GPU – 1
GPU – 1
GPU – 2
GPU – 3
GPU – 4
[i1,]
[i2,]
[i3,]
[i4,]
2. Replicate model on GPUs
[model]
GPU – 1
GPU – 1
GPU – 2
GPU – 3
GPU – 4
[i1,]
[model]
[i2,]
[model]
[i3,]
[model]
[i4,]
[model]
3. Parallel forward passes
[i1,]
[model]
[i2,]
[model]
[i3,]
[model]
[i4,]
[model]
GPU – 1
GPU – 2
GPU – 3
GPU – 4
GPU – 1
GPU – 2
GPU – 3
GPU – 4
[cached]
[o1,]
[cached]
[o2,]
[cached]
[o3,]
[cached]
[o4,]
4. Gather outputs on GPU–1
[o1,]
[o2,]
[o3,]
[o4,]
GPU – 1
GPU – 2
GPU – 3
GPU – 4
GPU – 1
[o1, o2, o3, o4]
Backward
1. Compute loss gradients on GPU–1
[loss1, loss2, loss3, loss4]
GPU – 1
GPU – 1
[grad_l1, grad_l2, grad_l3, grad_l4]
2. Scatter gradients to GPUs
[grad_l1, grad_l2, grad_l3, grad_l4]
GPU – 1
GPU – 1
GPU – 2
GPU – 3
GPU – 4
[grad_l1,]
[grad_l2,]
[grad_l3,]
[grad_l4,]
3. Parallel backward passes
[grad_l1,]
[cached]
[grad_l2,]
[cached]
[grad_l3,]
[cached]
[grad_l4,]
[cached]
GPU – 1
GPU – 2
GPU – 3
GPU – 4
GPU – 1
GPU – 2
GPU – 3
GPU – 4
[grad1_m]
[grad2_m]
[grad3_m]
[grad4_m]
4. Reduce gradients to GPU–1
[grad1_m]
[grad2_m]
[grad3_m]
[grad4_m]
GPU – 1
GPU – 2
GPU – 3
GPU – 4
GPU – 1
[grad1_m + grad2_m + grad3_m + grad4_m]

Custom DataParallel



Distributed Data Parallel

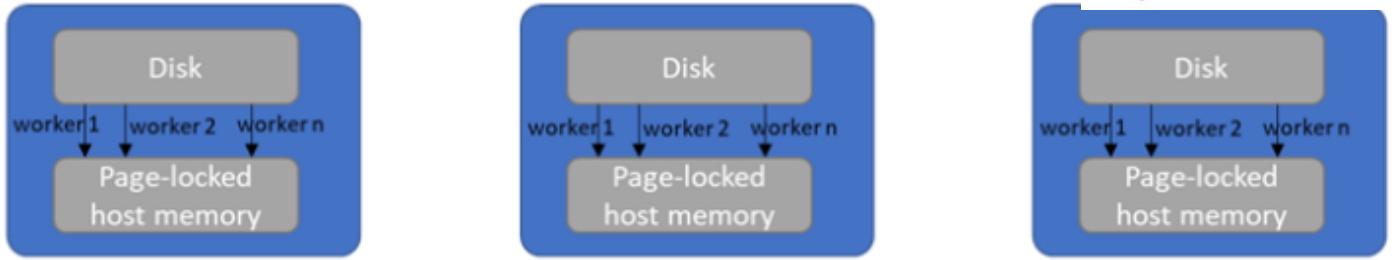
No master GPUs

Implemented in PyTorch DistributedDataParallel module

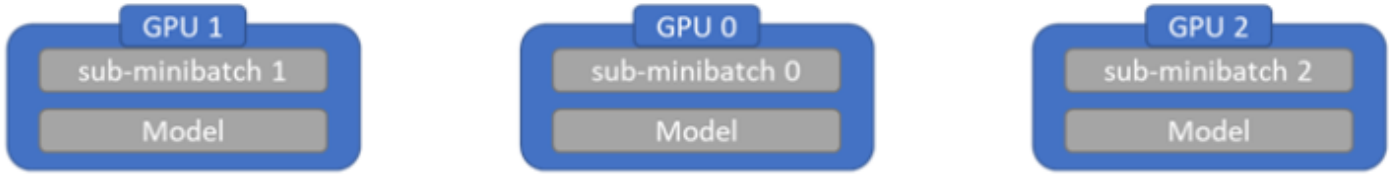
可用于单机多卡，

和多机多卡

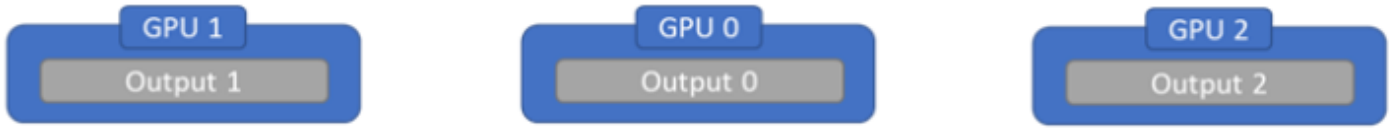
1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data



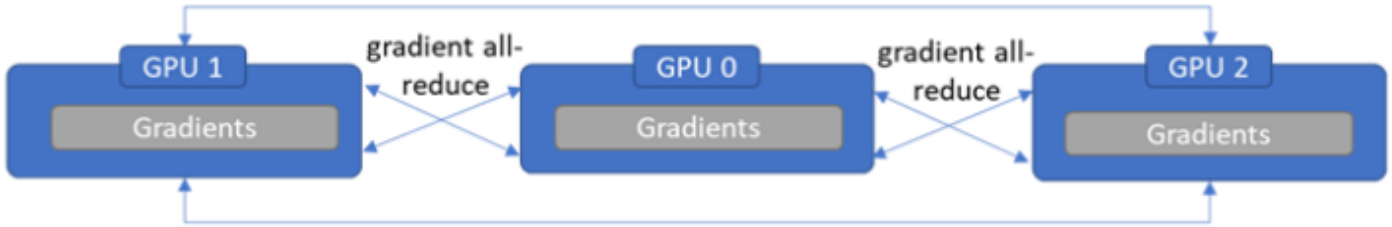
2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



3. Run forward pass on each GPU, compute output



4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation



5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required

YOLOv5 Multi-GPU Training

https://github.com/ultralytics/yolov5/issues/475

腾讯课堂

腾讯课堂