05--Performance Tuning Guide

原文地址：

- Performance Tuning Guide
- Author: Szymon Migacz

性能调优指南是一套优化和最佳实践，可以加速PyTorch中深度学习模型的训练和推理。所介绍的技术通常只需改变几行代码就可以实现，并且可以应用于所有领域的各种深度学习模型。

General optimizations

Enable async data loading and augmentation

torch.utils.data.DataLoader 支持在单独的工作子进程中进行异步数据加载和数据扩充。`DataLoader`的默认设置是 `num_workers=0`，这意味着数据加载是同步的，并且在主进程中完成。因此，主训练过程必须等待数据可用才能继续执行。

设置`num_workers > 0`可以实现异步数据加载和训练与数据加载之间的重叠。`num_workers`应该根据工作负载、CPU、GPU和训练数据的位置进行调整。

`DataLoader`接受`pin_memory`参数，其默认值为`False`。当使用GPU时，最好设置`pin_memory=True`，这将指示`DataLoader`使用钉住的内存，并使内存更快地从主机异步复制到GPU上。

Disable gradient calculation for validation or inference

PyTorch从所有涉及需要梯度的张量的操作中保存中间缓冲区。通常情况下，验证或推理不需要梯度。`torch.no_grad()`上下文管理器可用于在指定的代码块中禁用梯度计算，这将加速执行并减少所需的内存量。

Disable bias for convolutions directly followed by a bacth norn

torch.nn.Conv2d() 有偏见参数，默认为True（Conv1d 和 Conv3d 也是如此）。

如果一个 `nn.Conv2d` 层后面直接有一个 `nn.BatchNorm2d` 层，那么就不需要卷积中的偏置，而是使用 `nn.Conv2d` (..., `bias=False` , ...) 。不需要偏置，因为在第一步，`BatchNorm` 减去了平均值，这有效地抵消了偏置的影响。

这也适用于 1d 和 3d 卷积，只要 `BatchNorm`（或其他归一化层）在与卷积偏差相同的维度上归一化。

torchvision 提供的模型已经实现了这种优化。

Use `parameter.grad = None` instead of `model.zero_grad()` or `optimizer.zero_grad()`

Instead of calling :

```
model.zero_grad()
# or
optimizer.zero_grad()
```

to zero out gradients, use the following method instead :

```
for param in model.parameters():
    param.grad = None
```

第二个代码片断没有将每个单独参数的内存清零，同时，随后的后向传递使用赋值而不是加法来存储梯度，这减少了内存操作的数量。

将梯度设置为 `None` 与将其设置为零的数值行为略有不同，有关更多详细信息，请参阅 文档。

或者，从 PyTorch 1.7 开始，调用 `model` 或 `optimizer.zero_grad(set_to_none=True)`。

Fuse pointwise operations

Pointwise operations -- 逐点运算（逐元素加法、乘法、数学函数 - `sin()`、`cos()`、`sigmoid()` 等）可以融合到单个内核中，以摊销内存访问时间和内核启动时间。

PyTorch JIT 可以自动融合内核，尽管编译器中可能还没有实现其他融合机会，并且并非所有设备类型都受到同等支持。

逐点操作是受内存限制的，对于每个操作 PyTorch 都会启动一个单独的内核。每个内核从内存中加载数据，执行计算（这一步通常很便宜）并将结果存储回内存中。

融合运算器对多个融合点运算只启动一个内核，并且只向内存加载/存储一次数据。这使得 JIT 对于激活函数、优化器、自定义RNN单元等非常有用。

在最简单的情况下，融合可以通过对函数定义应用 torch.jit.script 装饰器来实现，例如。

```
@torch.jit.script
def fused_gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

Refer to TorchScript documentation for more advanced use cases.

Enable channels_last memory format for computer vision models

PyTorch 1.5 引入了对卷积网络的 channels_last 内存格式的支持。这种格式旨在与 AMP 结合使用，以进一步加速使用 Tensor Cores 的卷积神经网络。

对channel_last的支持是实验性的，但它有望用于标准的计算机视觉模型（如ResNet-50，SSD）。要将模型转换为channel_last格式，请遵循Channels Last内存格式教程。该教程包括一个关于转换现有模型的部分。

Checkpoint intermediate buffers

缓冲区检查点是一种减轻模型训练的内存容量负担的技术。它不是存储所有层的输入来计算反向传播中的上游梯度，而是存储几个层的输入，而其他层在反向传播期间重新计算。减少的内存需求可以增加批量大小，从而提高利用率。

检查点的目标应谨慎选择。最好不要存储大的层的输出，这些输出的再计算成本小。目标层的例子是激活函数（如 `ReLU`、`Sigmoid`、`Tanh`）、上/下采样和具有小的积累深度的矩阵-向量运算。

PyTorch支持本地的 torch.utils.checkpoint API，以自动执行检查点和重新计算。

Disable debugging APIs

许多 PyTorch API 用于调试，应在常规训练运行时禁用：

- anomaly detection: torch.autograd.detect_anomaly or torch.autograd.set_detect_anomaly(True)
- profiler related: torch.autograd.profiler.emit_nvtx, torch.autograd.profiler.profile
- autograd gradcheck: torch.autograd.gradcheck or torch.autograd.gradgradcheck

CPU specific optimizations

Utilize Non-Uniform Memory Access (NUMA，非统一的内存访问) Controls

NUMA 或非统一内存访问是一种用于数据中心机器的内存布局设计，旨在利用具有多个内存控制器和块的多插槽机器中的内存局部性。一般来说，所有深度学习工作负载（训练或推理）都可以获得更好的性能，而无需跨 NUMA 节点访问硬件资源。因此，推理可以在多个实例上运行，每个实例在一个套接字上运行，以提高吞吐量。对于单节点训练任务，推荐分布式训练，让每个训练过程运行在一个socket上。

在一般情况下，以下命令只在第N个节点的核心上执行PyTorch脚本，并避免跨插槽内存访问，以减少内存访问开销。

```
# numactl --cpunodebind=N --membind=N python <pytorch_script>
```

More detailed descriptions can be found here.

Utilize OpenMP

OpenMP被用来为并行计算任务带来更好的性能。OMP_NUM_THREADS是一个最简单的开关，可以用来加速计算。它决定了用于OpenMP计算的线程数量。CPU亲和力设置控制工作负载如何在多个核心上分配。它影响到通信开销、缓存行无效开销或页面抖动，因此适当设置CPU亲和力会带来性能上的好处。GOMP_CPU_AFFINITY或KMP_AFFINITY决定了如何将OpenMP*线程与物理处理单元绑定。详细的信息可以在 这里 找到。

通过以下命令，PyTorch在N个OpenMP线程上运行任务。

```
# export OMP_NUM_THREADS=N
```

通常情况下，以下环境变量用于设置GNU OpenMP实现的CPU亲和性。OMP_PROC_BIND指定线程是否可以在处理器之间移动。将其设置为CLOSE可以使OpenMP线程在连续的地方分区中靠近主线程。OMP_SCHEDULE决定了OpenMP线程如何被调度。GOMP_CPU_AFFINITY将线程绑定到特定的CPU上。

```
# export OMP_SCHEDULE=STATIC
# export OMP_PROC_BIND=CLOSE
# export GOMP_CPU_AFFINITY="N-M"
```

Intel OpenMP Runtime Library (libiomp)

默认情况下，PyTorch使用GNU OpenMP（GNU libgomp）进行并行计算。在英特尔平台上，英特尔OpenMP运行时库（libiomp）提供了OpenMP API规范支持。与libgomp相比，它有时会带来更多的性能优势。利用环境变量LD_PRELOAD可以将OpenMP库切换到libiomp。

```
# export LD_PRELOAD=<path>/libiomp5.so:$LD_PRELOAD
```

类似于GNU OpenMP中的CPU亲和力设置，libiomp中提供了环境变量来控制CPU亲和力设置。KMP_AFFINITY将OpenMP线程与物理处理单元绑定。KMP_BLOCKTIME设置线程在完成一个并行区域的执行后，在睡眠前应该等待的时间，单位是毫秒。在大多数情况下，将KMP_BLOCKTIME设置为1或0会产生良好的性能。下面的命令显示了英特尔OpenMP Runtime Library的一个常见设置。

```
# export KMP_AFFINITY=granularity=fine,compact,1,0
# export KMP_BLOCKTIME=1
```

Switch Memory allocator

对于深度学习的工作负载，Jemalloc或TCMalloc可以通过尽可能地重用内存来获得比默认malloc功能更好的性能。Jemalloc是一个通用的malloc实现，强调避免碎片化和可扩展的并发支持。TCMalloc还具有一些优化功能，以加快程序的执行速度。其中之一是在缓存中保留内存，以加快对常用对象的访问。即使在去分配之后，保留这样的缓存也有助于避免在以后重新分配这些内存时进行昂贵的系统调用。使用环境变量LD_PRELOAD来利用其中的一个优势。

```
# export LD_PRELOAD=<jemalloc.so/tcmalloc.so>:$LD_PRELOAD
```

Use oneDNN Graph with TorchScript for inference

oneDNN Graph 可以显着提高推理性能。它将一些计算密集型操作（例如卷积、matmul）与其相邻操作融合在一起。目前，它作为 Float32 数据类型的实验性功能受到支持。 oneDNN Graph 接收模型的图，并根据示例输入的形状识别算子融合的候选者。应使用示例输入对模型进行 JIT 跟踪。在对与示例输入具有相同形状的输入进行几次热身迭代后，将观察到加速。下面的示例代码片段适用于 resnet50，但它们也可以很好地扩展到使用带有自定义模型的 oneDNN Graph。

```
# Only this extra line of code is required to use oneDNN Graph
torch.jit.enable_onednn_fusion(True)
```

使用 oneDNN Graph API 只需要多写一行代码。如果你使用 oneDNN Graph，请避免调用 `torch.jit.optimize_for_inference`。

```
# sample input should be of the same shape as expected inputs
sample_input = [torch.rand(32, 3, 224, 224)]
# Using resnet50 from TorchVision in this example for illustrative purposes,
# but the line below can indeed be modified to use custom models as well.
model = getattr(torchvision.models, "resnet50")().eval()
# Tracing the model with example input
traced_model = torch.jit.trace(model, sample_input)
# Invoking torch.jit.freeze
traced_model = torch.jit.freeze(traced_model)
```

使用样本输入对模型进行 JIT 跟踪后，可以在几次热身运行后将其用于推理。

```
with torch.no_grad():
    # a couple of warmup runs
    traced_model(*sample_input)
    traced_model(*sample_input)
    # speedup would be observed after warmup runs
    traced_model(*sample_input)
```

Train a model on CPU with PyTorch DistributedDataParallel(DDP) functionality

对于小规模的模型或受内存约束的模型，如DLRM，在CPU上训练也是一个不错的选择。在有多个套接字的机器上，分布式训练带来了高效的硬件资源使用，加速了训练过程。Torch-ccl通过英特尔(R)oneCCL（集体通信库）进行了优化，实现了allreduce、allgather、alltoall等集体通信，实现了PyTorch C10D ProcessGroup API，可以作为外部ProcessGroup动态加载。在PyTorch DDP模型中实现的优化后，torch-ccl可以加速通信操作。除了对通信内核进行优化外，torch-ccl还具有同步计算-通信功能。

GPU specific optimizations

Enable cuDNN auto-tuner

NVIDIA cuDNN 支持许多算法来计算卷积。 Autotuner 运行一个简短的基准测试并选择在给定硬件上针对给定输入大小具有最佳性能的内核。

对于卷积网络（目前不支持其他类型），在启动训练循环之前通过设置启用cuDNN自动调谐器。

```
torch.backends.cudnn.benchmark = True
```

- 自动调谐器的决定可能是非决定性的；不同的运行可能会选择不同的算法。For more details see PyTorch: Reproducibility.
- 在一些罕见的情况下，比如输入规模高度可变的情况下，最好在禁用自动调谐器的情况下运行卷积网络，以避免与每个输入规模的算法选择有关的开销。

Avoid unnecessary CPU-GPU synchronization

避免不必要的同步，尽可能让CPU运行在加速器前面，以确保加速器的工作队列包含许多操作。

在可能的情况下，避免需要同步的操作，比如说：

- `print(cuda_tensor)`
- `cuda_tensor.item()`
- memory copies : `tensor.cuda()` , `cuda_tensor.cpu()` and equivalent `tensor.to(device)` calls
- `cuda_tensor.nonzero()`
- python control flow which depends on results of operations performed on cuda tensors e.g. `if (cuda_tensor != 0).all()`

Create tensors directly on the target device

与其调用 `torch.rand(size).cuda()` 来生成随机张量，不如直接在目标设备上产生输出：`torch.rand(size, device=torch.device('cuda'))`。

这适用于所有创建新张量并接受设备参数的函数： torch.rand(), torch.zeros(), torch.full() and similar.

Use mixed precision and AMP

混合精度利用Tensor Cores，在Volta和较新的GPU架构上提供高达3倍的整体速度提升。要使用张量核心，应启用AMP，矩阵/张量尺寸应满足调用使用张量核心的内核的要求。

要使用张量核心：

- 将大小设置为 8 的倍数（映射到张量核心的维度）
    - 有关特定于层类型的更多详细信息和指南，请参阅深度学习性能文档

    - 如果层的大小来自于其他参数而不是固定的，它仍然可以被明确地填充，例如NLP模型中的词汇量。
- enable AMP
    - 混合精度训练和 AMP 简介：video, slides
    - 从 PyTorch 1.6 开始可以使用本机 PyTorch AMP： documentation, examples, tutorial

Pre-allocate memory in case of variable input length

语音识别或NLP的模型往往是在序列长度可变的输入张量上训练的。对于PyTorch的缓存分配器来说，可变长度可能是个问题，会导致性能下降或出现意外的内存不足错误。如果一个序列长度较短的批次之后是另一个序列长度较长的批次，那么PyTorch就不得不释放之前迭代的中间缓冲区并重新分配新的缓冲区。这个过程很耗时，而且会造成缓存分配器的碎片化，可能会导致内存不足的错误。

一个典型的解决方案是实现预分配。它由以下步骤组成：

1. 产生一批具有最大序列长度的输入（通常是随机的）（与训练数据集中的最大长度相对应或与一些预定义的阈值相对应），
2. 对生成的批处理进行前向和后向处理，不执行优化器或学习率调度器，该步骤预先分配最大尺寸的缓冲区，可在随后的训练迭代中重复使用，
3. 将梯度归零，
4. 进行定期训练

Distributed optimizations

Use efficient data-parallel backend

PyTorch 有两种方式来实现数据并行训练：

- torch.nn.DataParallel
- torch.nn.parallel.DistributedDataParallel

分布式数据并行（`DistributedDataParallel`）提供了更好的性能，并可扩展到多个GPU。更多信息请参考PyTorch文档中的 CUDA最佳实践的相关章节。

Skip unnecessary all-reduce if training with DistributedDataParallel and gradient accumulation

默认情况下，torch.nn.parallel.DistributedDataParallel 会在每次向后传递后执行梯度全还原，以计算所有参与训练的工作者的平均梯度。如果训练使用N个步骤的梯度累积，那么在每个训练步骤之后都不需要进行all-reduce，只需要在最后一次调用backward之后，在执行优化器之前进行all-reduce。

DistributedDataParallel 提供 no_sync() 上下文管理器，它为特定迭代禁用梯度全归约。no_sync() 应应用于梯度累积的前 N-1 次迭代，最后一次迭代应遵循默认执行并执行所需的梯度 all-reduce。

Match the order of layers in constructors and during the execution if using DistributedDataParallel(find_unused_parameters=True)

带有 `find_unused_parameters=True` 的 torch.nn.parallel.DistributedDataParallel 使用模型构造函数中的层顺序和参数来构建用于 `DistributedDataParallel` 梯度全归约的存储桶。`DistributedDataParallel` 将 all-reduce 与反向传递重叠。仅当给定存储桶中参数的所有梯度都可用时，才会异步触发特定存储桶的 All-reduce。

为了使重叠量最大化，模型构造器中的顺序应该与执行过程中的顺序大致相符。如果顺序不匹配，那么整个桶的全还原就会等待最后到达的梯度，这可能会减少后向传递和全还原之间的重叠，全还原最终可能会被暴露，从而减慢训练速度。

带有 `find_unused_parameters=False`（这是默认设置）的 `DistributedDataParallel`依赖于根据后向传递过程中遇到的操作顺序来自动形成桶。在 `find_unused_parameters=False` 的情况下，没有必要对层或参数进行重新排序以达到最佳性能。

Load-balance workload in a distributed setting

负载不平衡通常可能发生在处理序列数据的模型中（语音识别、翻译、语言模型等）。如果一个设备收到一批数据，其序列长度长于其余设备的序列长度，那么所有设备都会等待最后完成的工作者。在带有DistributedDataParallel后端的分布式环境中，后向传递的功能是一个隐式同步点。

有多种方法来解决负载平衡问题。其核心思想是在每个全局批次内尽可能均匀地将工作负荷分配给所有工作者。例如，Transformer通过形成具有近似恒定数量的标记（和一个批次中的可变序列数量）的批次来解决不平衡问题，其他模型通过对具有相似序列长度的样本进行分桶或甚至通过对数据集进行序列长度排序来解决不平衡问题。