PyTorch 之 Checkpoint 机制解析

原创 CV开发者都爱看的 极市平台 2022-01-08 22:00:00 手机阅读

作者丨Lart

编辑丨极市平台

极市导读

PyTorch 提供了一种非常方便的节省显存的方式，就是 Checkpoint 机制。这篇文章的目的在于更透彻的了解其内在的机制。 >>加入极市CV技术交流群，走在计算机视觉的最前沿

Checkpoint 机制

该技术的核心是一种使用时间换空间的策略。在现有的许多方法中被大量使用，例如 DenseNet、Swin Transformer 源码中都可以看到它的身影。

为了了解它的工作原理，我们先得弄明白的一个问题是，PyTorch 模型在训练过程中显存占用主要是用来存储什么?

关于这一点，Connolly 的文章 《PyTorch 显存机制分析》（https://zhuanlan.zhihu.com/p/424512257） 介绍的非常详细：

> 开门见山的说，PyTorch 在进行深度学习训练的时候，有 4 大部分的显存开销，分别是**模型参数(parameters)**，**模型参数的梯度(gradients)**，**优化器状态(optimizer states)** 以及**中间激活值(intermediate activations)** 或者叫中间结果(intermediate results)。

而通过 Checkpoint 技术，我们可以通过一种取巧的方式，使用 PyTorch 提供的 “no-grad” （`no_grad()`）模式来避免将这部分运算被autograd记录到反向图“backward graph”中，从而避免了对于中间激活值的存储需求。

> 个人理解（欢迎指出错误）：
> 前向传播时 autograd 记录各个操作反向传播需要的一些信息和中间变量。反向传播之后，用于计算梯度的中间结果会被释放。也就是说，模型参数、优化器状态和参数梯度是始终在占用着存储空间的，中间激活值在反向传播之后就自动被清空了。这里我简单修改了 《PyTorch 显存机制分析》（https://zhuanlan.zhihu.com/p/424512257） 中给出的例子 进行了一下验证（https://github.com/lartpang/CodeForArticle/tree/main/CheckpointAndGPUUsage.PyTorch） 。
> 这里实际上会引申出另一个问题，**为什么自定义 Function 一般情况下会减少显存占用？**（在 Vision Longformer 中各种实现的对比里可以明显看到这一现象）
> 我觉得主要是因为自定义 Function 的时候，我们可以从一整个模块的角度来更有针对性的在 `ctx` 中存储中间变量，而自动求导引擎可能关注的太细了，导致存储许多不必要的中间变量。关于这一点暂时不知道如何验证。

这可以避免存储模型特定层中间运算结果，从而有效降低了前向传播中显存的占用。 这些中间结果会在反向传播的时候被即时重新计算一次。要注意，被 checkpoint 包裹的层反向传播时仍然会在第一次反向传播的时候开辟存储梯度的空间。

> 因为 checkpoint 是在 `torch.no_grad()` 模式下计算的目标操作的前向函数，这并不会修改原本的叶子结点的状态，有梯度的还会保持。只是关联这些叶子结点的临时生成的中间变量会被设置为不需要梯度，因此梯度链式关系会被断开。

通过这样的方式，虽然延长了反向传播的时间，但是却也在一定程度上缓解了存储大量中间变量带来的显存占用。

源码解析

以下代码来自 PyTorch v1.10.1 版本：https://github.com/pytorch/pytorch/blob/v1.10.1/torch/utils/checkpoint.py。最新的版本中补充了一些新的内容，待其最终发布后再说吧，下面的内容本身已经将 checkpoint 的核心介绍了。

辅助函数

这部分代码中首先构造了数个辅助函数，主要是用来做一些针对输入的检查和处理，同时也要处理好随机种子的问题。

```
def detach_variable(inputs: Tuple[Any, ...]) -> Tuple[torch.Tensor, ...]:
    if isinstance(inputs, tuple):
        out = []
        for inp in inputs:
            if not isinstance(inp, torch.Tensor):
                out.append(inp)
                continue

            # 直接detach()，从inp所在的计算图中剥离，默认会自动将requires_grad置为False
            x = inp.detach()
            # 但是这里的实际需求中，仍需要保持其自身的需要记录梯度的属性，且其梯度变为None
            x.requires_grad = inp.requires_grad
            # 因为只有需要保存梯度的参数才能够构建梯度的传播路径
            out.append(x)
        return tuple(out)
    else:
        raise RuntimeError(
            "Only tuple of tensors is supported. Got Unsupported input type: ", type(

def check_backward_validity(inputs: Iterable[Any]) -> None:
    """检查输入参数是否至少有一个需要记录梯度的Tensor，这样才能确保输出也有梯度。"""
    if not any(inp.requires_grad for inp in inputs if isinstance(inp, torch.Tensor)):
        warnings.warn("None of the inputs have requires_grad=True. Gradients will be
```

由于需要重复计算，所以随机状态的一致性是需要重视的。由于前向传播的部分在反向过程中仍会计算一次，所以如果不使用原始的随机状态的话，会导致重新计算和原本正常计算过程中的随机状态不同，而影响模型的行为。

另外在这段代码的注释中提到了一点有趣的地方：

> 由于无法获悉被 checkpoint 处理的操作是否在运算中间会将一些参数移动到不同的设备上，这可能需要手动保存这些设备对应的随机状态。当前的实现直接保存了所有可见设备上的随机状态，但是这样有时可能是不必要的，但是目前尚没有较好的解决策略。

所以按照文档的意思，就是在说如果没有这样的移动，那就可以不用保存随机状态咯？这一点其实有些令人疑惑。

```
    # We can't know if the run_fn will internally move some args to different devices,
    # which would require logic to preserve rng states for those devices as well.
    # We could paranoically stash and restore ALL the rng states for all visible devices,
    # but that seems very wasteful for most cases.  Compromise:  Stash the RNG state for
    # the device of all Tensor args.
    #
    # To consider:  maybe get_device_states and set_device_states should reside in torch/
    def get_device_states(*args) -> Tuple[List[int], List[torch.Tensor]]:
        """获取不同输入对应的GPU设备的随机数生成器的状态"""
        # This will not error out if "arg" is a CPU tensor or a non-tensor type because
        # the conditionals short-circuit.
        fwd_gpu_devices = list(set(arg.get_device() for arg in args
                                   if isinstance(arg, torch.Tensor) and arg.is_cuda))

        fwd_gpu_states = []
        for device in fwd_gpu_devices:
            with torch.cuda.device(device):
                fwd_gpu_states.append(torch.cuda.get_rng_state())

        return fwd_gpu_devices, fwd_gpu_states

    def set_device_states(devices, states) -> None:
        """针对不同的设备设置随机数生成器的状态"""
        for device, state in zip(devices, states):
            with torch.cuda.device(device):
                torch.cuda.set_rng_state(state)
```

核心 Function

可以看到，这里的 Checkpoint 本身就是基于 PyTorch 的 `Function` 实现的一个扩展算子，所以该部分代码也涉及到了 Function 的诸多功能。阅读它既可以帮助我们同时复习一下相关的知识，又能进一步了解更复杂的处理逻辑该如何搭建。

```
class CheckpointFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, run_function, preserve_rng_state, *args):
        check_backward_validity(args)
        # 暂存前向传播函数
        ctx.run_function = run_function
        ctx.preserve_rng_state = preserve_rng_state
```

```
        # 用来保存当前模型的混合精度的状态，以用在反向传播中
        ctx.had_autocast_in_fwd = torch.is_autocast_enabled()
        if preserve_rng_state:  # 保存目标模块前向传播之前，此时CPU和GPU的随机数生成器的状态
            ctx.fwd_cpu_state = torch.get_rng_state()
            # Don't eagerly initialize the cuda context by accident.
            # (If the user intends that the context is initialized later, within thei
            # run_function, we SHOULD actually stash the cuda state here.  Unfortunat
            # we have no way to anticipate this will happen before we run the functic
            ctx.had_cuda_in_fwd = False
            if torch.cuda._initialized:
                # PyTorch提供的一个内部变量，用于判定CUDA状态是否已经被初始化了
                # torch.cuda.is_initialized中就用到了该变量
                ctx.had_cuda_in_fwd = True
                # 保存输入变量涉及的各个GPU设备的随机状态
                ctx.fwd_gpu_devices, ctx.fwd_gpu_states = get_device_states(*args)

        # Save non-tensor inputs in ctx, keep a placeholder None for tensors
        # to be filled out during the backward.
        ctx.inputs = []
        ctx.tensor_indices = []
        tensor_inputs = []
        for i, arg in enumerate(args):
            if torch.is_tensor(arg):
                tensor_inputs.append(arg)
                ctx.tensor_indices.append(i)
                ctx.inputs.append(None)
            else:
                ctx.inputs.append(arg)

        # save_for_backward()中保存反向传播中需要用到的输入和输出tensor量。
        # 由于在反向传播中需要重新计算记录梯度的output，所以就不要保存output了。
        # 并且后面的计算也不需要在梯度模式下计算。
        ctx.save_for_backward(*tensor_inputs)

        with torch.no_grad():
            # 不保存梯度的前向传播操作，也就是说这里的output是不会记录中间变量，无法直接计算梯度的。
            outputs = run_function(*args)
        return outputs

    @staticmethod
    def backward(ctx, *args):
        if not torch.autograd._is_checkpoint_valid():
            raise RuntimeError(
                "Checkpointing is not compatible with .grad() or when an `inputs` par
```

```
            " is passed to .backward(). Please use .backward() and do not pass it
            " argument.")
    # Copy the list to avoid modifying original list.
    inputs = list(ctx.inputs)
    tensor_indices = ctx.tensor_indices
    tensors = ctx.saved_tensors # 获取前向传播中保存的输入tensor

    # Fill in inputs with appropriate saved tensors.
    for i, idx in enumerate(tensor_indices):
        inputs[idx] = tensors[i]

    # Stash the surrounding rng state, and mimic the state that was
    # present at this time during forward.  Restore the surrounding state
    # when we're done.
    rng_devices = []
    if ctx.preserve_rng_state and ctx.had_cuda_in_fwd:
        rng_devices = ctx.fwd_gpu_devices

    # 使用之前前向传播开始之前保存的随机数生成器的状态来进行一次一模一样的前向传播过程
    with torch.random.fork_rng(devices=rng_devices, enabled=ctx.preserve_rng_stat
        # 使用上下文管理器保护原始的随机数生成器的状态，内部处理后在进行复原
        if ctx.preserve_rng_state:
            torch.set_rng_state(ctx.fwd_cpu_state)
            if ctx.had_cuda_in_fwd:
                set_device_states(ctx.fwd_gpu_devices, ctx.fwd_gpu_states)
        # 这里将inputs从计算图中剥离开，但是其属性requires_grad和原来是一样的，这么做的目的是
        # 从整个操作目的来看，由于我们需要重新计算输出，并将梯度回传到输入上，所以输入本身需要可以
        # 但是这里的回传不可以影响到checkpoint之外更靠前的那些操作，
        # backward之后会将之前保存的中间变量释放掉，而我们仅仅是为了计算当前一小块结构，所以梯度
        detached_inputs = detach_variable(tuple(inputs))  # 会变成叶子结点，grad和gr
        # 处理完随机状态之后，就该准备着手重新前向传播了。
        # 这次前向传播是在梯度模式(torch.enable_grad())下执行的。此时会保存中间变量。
        with torch.enable_grad(), torch.cuda.amp.autocast(ctx.had_autocast_in_fwd
            outputs = ctx.run_function(*detached_inputs)

    if isinstance(outputs, torch.Tensor):
        outputs = (outputs,)

    # run backward() with only tensor that requires grad
    outputs_with_grad = []
    args_with_grad = []
    for i in range(len(outputs)):
        # 记录需要计算梯度的输出outputs[i]以及对应的回传回来的有效梯度args[i]
        if torch.is_tensor(outputs[i]) and outputs[i].requires_grad:
```

```
            outputs_with_grad.append(outputs[i])
            args_with_grad.append(args[i])
    # 检查需要计算梯度的输出，如果没有输出需要计算梯度，那么实际上就说明这个模块是不参与梯度计算的
    # 也就是说，该模块不需要使用checkpoint来调整。
    if len(outputs_with_grad) == 0:
        raise RuntimeError(
            "none of output has requires_grad=True,"
            " this checkpoint() is not necessary")
    # 该操作对被包裹的目标操作计算反向传播，即计算回传到输入detached_inputs上的梯度。
    # 由于输入的tensor已被从整体梯度图中剥离，所以可以看做是一个叶子结点，可以在反向传播之后获得其
    # 另外这里反传计算梯度也不会导致将更靠前的结构中暂时保存来计算梯度的参数给释放掉。
    torch.autograd.backward(outputs_with_grad, args_with_grad)
    # 如果前面不执行detach()，这里的inp.grad会被直接释放并置为None，这并不符合预期
    grads = tuple(inp.grad if isinstance(inp, torch.Tensor) else None
                  for inp in detached_inputs)

    # 这里返回的梯度与当前类的forward输入一一对应，
    # 由于这里的forward包含着本不需要梯度的两个参数run_function、preserve_rng_state，故对
    return (None, None) + grads
```

这里实际上就是在原始的操作和整体的计算图之间添加了一个中间层，用于信息的交互：

1. 原始模型的数据传输到被包裹的目标层的时候，数据进入 checkpoint 的 `forward()` 中，被 checkpoint 进行检查和记录后，再送入目标层中；
2. 目标层在非梯度模式下执行前向传播。该模式下，新创建的 tensor 都是不会记录梯度信息的；
3. 目标层的结果通过 checkpoint 的前向传播输出，送入模型后续的其他结构中；
4. 执行反向传播，损失求导，链式回传，计算梯度；
5. 回传回来的对应于 checkpoint 输出的梯度被送入其对应的反向传播函数，即 checkpoint 的 `backward()` 。
6. 梯度送入 checkpoint 中后，需要进一步将梯度回传到目标层的输入上。由于在 checkpoint 的 `forward` 中目标层本身前向传播是处于非梯度状态下，所以回传路径上缺少目标层中操作的梯度子图。于是为了获取这部分信息，需要先梯度状态下对目标层进行一次前向传播，通过将回传回来的梯度和目标层的输出一起执行 `torch.autograd.backward(outputs_with_grad, args_with_grad)` ，从而获得对应输入的梯度信息。

7. 将对应目标操作输入的梯度信息按照 checkpoint 本身 Function 的 backward 的需求，使用 None 对其他辅助参数的梯度占位后进行返回。
8. 返回的对应于其他模块的输出量的梯度，被沿着反向传播的路径送入对应操作的 backward 中，一层一层回传累加到各个叶子节点上。

定义好操作后，进行一个简单的包装，同时处理一下默认参数，补充了更细致的文档：

```
def checkpoint(function, *args, use_reentrant: bool = True, **kwargs):
    r"""Checkpoint a model or part of the model

    Checkpointing works by trading compute for memory. Rather than storing all
    intermediate activations of the entire computation graph for computing
    backward, the checkpointed part does **not** save intermediate activations,
    and instead recomputes them in backward pass. It can be applied on any part
    of a model.

    Specifically, in the forward pass, :attr:`function` will run in
    :func:`torch.no_grad` manner, i.e., not storing the intermediate
    activations. Instead, the forward pass saves the inputs tuple and the
    :attr:`function` parameter. In the backwards pass, the saved inputs and
    :attr:`function` is retrieved, and the forward pass is computed on
    :attr:`function` again, now tracking the intermediate activations, and then
    the gradients are calculated using these activation values.
    这一段详细介绍了checkpoint的核心技术，也就是在非梯度模式下执行目标操作的前向传播，只保留输入和结构

    The output of :attr:`function` can contain non-Tensor values and gradient
    recording is only performed for the Tensor values. Note that if the output
    consists of nested structures (ex: custom objects, lists, dicts etc.)
    consisting of Tensors, these Tensors nested in custom structures will not
    be considered as part of autograd.
    因为checkpoint的backward实现的逻辑中，直接遍历目标操作的输出（会被自定转换成元组类型）并确定那些

    .. warning::
        Checkpointing currently only supports :func:`torch.autograd.backward`
        and only if its `inputs` argument is not passed. :func:`torch.autograd.grad`
        is not supported.

    .. warning::
        If :attr:`function` invocation during backward does anything different
```

```
        than the one during forward, e.g., due to some global variable, the
        checkpointed version won't be equivalent, and unfortunately it can't be
        detected.
        尽量保证目标操作在反向计算期间和前向期间的操作的一致性。
        因为在checkpoint会在反向中重新计算一次前向，这可能会带来一些由于无法检测到的不确定因素而造成[

    .. warning::
        If checkpointed segment contains tensors detached from the computational
        graph by `detach()` or `torch.no_grad()`, the backward pass will raise an
        error. This is because `checkpoint` makes all the outputs require
        gradients which causes issues when a tensor is defined to have no
        gradient in the model. To circumvent this, detach the tensors outside of
        the `checkpoint` function.
        不要在目标操作中包含detach或者非梯度模式的处理。
        **在我的实际测试中似乎并没有这个问题？**或许这里应该看一下pytorch提供的测试案例。

    .. warning::
        At least one of the inputs needs to have :code:`requires_grad=True` if
        grads are needed for model inputs, otherwise the checkpointed part of the
        model won't have gradients. At least one of the outputs needs to have
        :code:`requires_grad=True` as well.
        要保证至少有一个输入是requires_grad的，这样才可以保证这部分操作可以被记录梯度。
        也要保证输出至少有一个需要计算梯度。

    Args:
        function: describes what to run in the forward pass of the model or
            part of the model. It should also know how to handle the inputs
            passed as the tuple. For example, in LSTM, if user passes
            ``(activation, hidden)``, :attr:`function` should correctly use the
            first input as ``activation`` and the second input as ``hidden``
        preserve_rng_state(bool, optional, default=True):  Omit stashing and restorin
            the RNG state during each checkpoint.
        args: tuple containing inputs to the :attr:`function`

    Returns:
        Output of running :attr:`function` on :attr:`*args`
    """
    # Hack to mix *args with **kwargs in a python 2.7-compliant way
    preserve = kwargs.pop('preserve_rng_state', True)
    if kwargs:
```

```
        raise ValueError("Unexpected keyword arguments: " + ",".join(arg for arg in k

    return CheckpointFunction.apply(function, preserve, *args)
```

应用案例

Checkpoint for Sequential

PyTorch 源码中给了一个很直接的应用案例，就是将 checkpoint 应用于 `Sequential` 搭建起来的模型。按照分段数 `segments` 指定的，将模型划分为多段。

```
def checkpoint_sequential(functions, segments, input, **kwargs):
    r"""A helper function for checkpointing sequential models.

    Sequential models execute a list of modules/functions in order
    (sequentially). Therefore, we can divide such a model in various segments
    and checkpoint each segment. All segments except the last will run in
    :func:`torch.no_grad` manner, i.e., not storing the intermediate
    activations. The inputs of each checkpointed segment will be saved for
    re-running the segment in the backward pass.

    See :func:`~torch.utils.checkpoint.checkpoint` on how checkpointing works.

    .. warning::
        Checkpointing currently only supports :func:`torch.autograd.backward`
        and only if its `inputs` argument is not passed. :func:`torch.autograd.grad`
        is not supported.

    .. warning:
        At least one of the inputs needs to have :code:`requires_grad=True` if
        grads are needed for model inputs, otherwise the checkpointed part of the
        model won't have gradients.

    .. warning:
        Since PyTorch 1.4, it allows only one Tensor as the input and
        intermediate outputs, just like :class:`torch.nn.Sequential`.

    Args:
        functions: A :class:`torch.nn.Sequential` or the list of modules or
```

```
            functions (comprising the model) to run sequentially.
        segments: Number of chunks to create in the model
        input: A Tensor that is input to :attr:`functions`
        preserve_rng_state(bool, optional, default=True):  Omit stashing and restorin
            the RNG state during each checkpoint.

    Returns:
        Output of running :attr:`functions` sequentially on :attr:`*inputs`

    Example:
        >>> model = nn.Sequential(...)
        >>> input_var = checkpoint_sequential(model, chunks, input_var)
    """
    # Hack for keyword-only parameter in a python 2.7-compliant way
    preserve = kwargs.pop('preserve_rng_state', True)
    if kwargs:
        raise ValueError("Unexpected keyword arguments: " + ",".join(arg for arg in k

    def run_function(start, end, functions):
        def forward(input):
            for j in range(start, end + 1):
                input = functions[j](input)
            return input
        return forward

    if isinstance(functions, torch.nn.Sequential):
        functions = list(functions.children())
        # 获取Sequential的子模块，这里使用children方法，仅获取最外层

    segment_size = len(functions) // segments
    # the last chunk has to be non-volatile （为什么？似乎加上也是可以的）
    end = -1
    for start in range(0, segment_size * (segments - 1), segment_size):
        end = start + segment_size - 1
        # 迭代式的将各个子模块集合使用checkpoint包装并前向传播。
        input = checkpoint(run_function(start, end, functions), input,
                           preserve_rng_state=preserve)
    # 剩余的结构不再使用checkpoint
    return run_function(end + 1, len(functions) - 1, functions)(input)
```

参考链接

- Checkpoint 源码：https://github.com/pytorch/pytorch/blob/master/torch/utils/checkpoint.py
- PyTorch 的 Autograd - xiaopl 的文章 - 知乎 https://zhuanlan.zhihu.com/p/69294347
- PyTorch 源码解读之 torch.autograd：梯度计算详解 - OpenMMLab 的文章 - 知乎 https://zhuanlan.zhihu.com/p/321449610
- 浅谈 PyTorch 中的 tensor 及使用 - xiaopl 的文章 - 知乎 https://zhuanlan.zhihu.com/p/67184419
- https://pytorch.org/docs/stable/notes/autograd.html#locally-disable-grad-doc
- https://pytorch.org/tutorials/beginner/introyt/autogradyt_tutorial.html

如果觉得有用，就请分享到朋友圈吧！

极市平台签约作者#

Lart

知乎：人民艺术家

CSDN：有为少年

大连理工大学在读博士

研究领域：主要方向为图像分割，但多从事于二值图像分割的研究。也会关注其他领域，例如分类和检测等方向的发展。

作品精选

实践教程 | PyTorch中相对位置编码的理解

实操教程 | 使用Docker为无网络环境搭建深度学习环境

实践教程 | 一文让你把Docker用起来!

投稿方式：

添加小编微信Fengcall（微信号：fengcall19），备注：**姓名-投稿**

△长按添加极市平台小编

觉得有用麻烦给个在看啦~

阅读原文

喜欢此内容的人还喜欢

Python实现替换照片人物背景，精细到头发丝（附代码）

Jack Cui

干货丨pytorch必须掌握的的4种学习率衰减策略

深度学习算法与计算机视觉

Python数据处理入门教程！

Datawhale