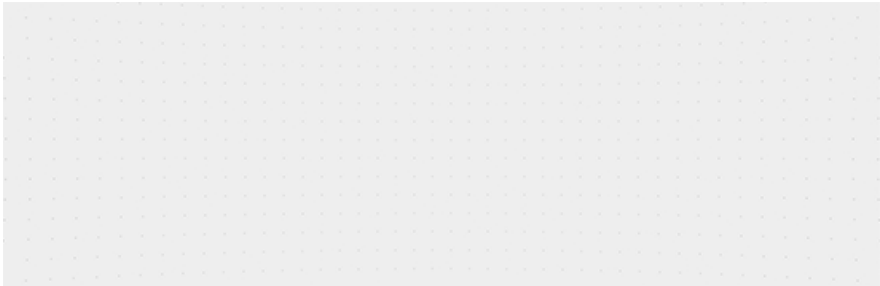


PyTorch 深度剖析：如何使用模型并行技术（Model Parallel）

原创 CV开发者都爱看的 极市平台 2021-11-18 22:00:00 手机阅读 跟

收录于话题
#pytorch

↑ 点击蓝字 关注极市平台



作者 | 科技猛兽
编辑 | 极市平台

极市导读

本文介绍了模型并行技术 Model Parallel，与 DataParallel 相反，Model Parallel 将一个单一的模型分割到不同的 GPU 上，而不是在每个 GPU 上复制整个模型。Pipeline Model Parallel 是一种进一步加速模型并行的策略。 >>加入极市CV技术交流群，走在计算机视觉的最前沿

0 背景

模型并行 (Single Machine Model Parallel) 在分布式训练技术中广泛使用。

它和 DataParallel 不同，DataParallel 是在多个GPU上训练神经网络，将同一个模型复制到所有 GPU 上，每个 GPU 消耗不同的输入数据分区。虽然 DataParallel 可以大大加快训练过程，但是，有的时候一些模型太大，没办法装入单个 GPU 的时候，就不适用了。

这篇文章展示了如何通过使用模型并行来解决这个问题，与 DataParallel 相反，Model Parallel 将一个单一的模型分割到不同的 GPU 上，而不是在每个 GPU 上复制整个模型。具体来说，比如一个模型 m 包含10层：当使用 DataParallel 时，每个 GPU 将有这10层的每个副本，而在两个 GPU 上使用模型并行时，每个 GPU 只需要承载5层。

Model Parallel 的 High-level 的理念是将模型的不同子网络放在不同的设备上，并相应地实现 forward() 方法，在 GPU 之间移动模型中间输出。每个模型被拆成了多块，只有一块在单独的设备上运行，所以，一组设备可以共同为一个更大的模型服务。

在这篇文章中，我们不会试图构建巨大的模型，并将其硬挤入数量有限的 GPU。相反，这篇文章的重点是展示 Model Parallel 的具体操作方法。

1 Model Parallel 基操

比如现在有一个包含2个 Linear layers 的模型，我们想在2块 GPU 上 run 它，办法可以是在每块 GPU 上放置1个 Linear layer，并且把得到的中间结果在 GPU 之间移动。代码可以是这样子：

```
import torch
import torch.nn as nn
```

壹伴图

极市平台
extreme

月发文数目： **
月平均阅读： **

文章工具

已发文

采集图文 合成多

采集样式 查看

```
import torch.optim as optim

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to('cuda:1')

    def forward(self, x):
        x = self.relu(self.net1(x.to('cuda:0')))
        return self.net2(x.to('cuda:1'))
```

注意，上述 ToyModel 看起来与在单个 GPU 上的实现方式**非常相似**，除了四个 **to(device)** 的调用，将 Linear layer 和张量放在适当的设备上。这是该模型中唯一需要改变的地方。backward() 和 torch.optim 将自动处理梯度问题，就像模型是在一个 GPU 上一样。

你只需要确保在**调用损失函数时**，**标签和输出是在同一个设备上**。像下面这样：

```
model = ToyModel()
loss_fn = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)

optimizer.zero_grad()
outputs = model(torch.randn(20, 10))
labels = torch.randn(20, 5).to('cuda:1')
loss_fn(outputs, labels).backward()
optimizer.step()
```

这里应该把标签 labels 放在1号 GPU 上面，因为模型的输出就在1号 GPU 上。

2 对已有的模块使用 Model Parallel

这段我们介绍如何在多个 GPU 上运行一个**现有的单 GPU 模块**，只需做几行修改即可。

下面的代码显示了如何将 torchvision.models.resnet50() 分解到两个 GPU。这个想法是继承现有的 ResNet 模块，并在构建过程中将各层分割到两个 GPU 上面。然后，overwrite forward() 方法，通过相应地移动中间输出来缝合两个子网络。

```
from torchvision.models.resnet import ResNet, Bottleneck

num_classes = 1000

class ModelParallelResNet50(ResNet):
    def __init__(self, *args, **kwargs):
        super(ModelParallelResNet50, self).__init__(
            Bottleneck, [3, 4, 6, 3], num_classes=num_classes, *args, **kwargs)

        self.seq1 = nn.Sequential(
            self.conv1,
            self.bn1,
            self.relu,
            self.maxpool,

            self.layer1,
            self.layer2
        ).to('cuda:0')

        self.seq2 = nn.Sequential(
            self.layer3,
```

```

        self.layer4,
        self.avgpool,
    ).to('cuda:1')

    self.fc.to('cuda:1')

    def forward(self, x):
        x = self.seq2(self.seq1(x).to('cuda:1'))
        return self.fc(x.view(x.size(0), -1))

```

上述实现解决了模型太大，无法装入单个GPU的情况下的问题。然而，对于运行速度而言，它将比在单个 GPU 上运行的速度要慢。这是因为，在任何时候，两个 GPU 中只有一个在工作，而另一个则是坐在那里啥也不干。由于中间输出需要在第二层和第三层之间从 cuda:0复制到 cuda:1，所以性能会进一步恶化。

下面我们通过一个实验，看看具体的程序运行时间的量化对比。在这个实验中，我们通过运行随机输入和标签来训练 ModelParallelResNet50 和现有的 torchvision.models.resnet50()。在训练之后，这些模型不会产生任何有用的预测，但我们可以对执行时间有一个合理的了解。

```

import torchvision.models as models

num_batches = 3
batch_size = 120
image_w = 128
image_h = 128

def train(model):
    model.train(True)
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.001)

    one_hot_indices = torch.LongTensor(batch_size) \
        .random_(0, num_classes) \
        .view(batch_size, 1)

    for _ in range(num_batches):
        # generate random inputs and labels
        inputs = torch.randn(batch_size, 3, image_w, image_h)
        labels = torch.zeros(batch_size, num_classes) \
            .scatter_(1, one_hot_indices, 1)

        # run forward pass
        optimizer.zero_grad()
        outputs = model(inputs.to('cuda:0'))

        # run backward pass
        labels = labels.to(outputs.device)
        loss_fn(outputs, labels).backward()
        optimizer.step()

```

上面的 train(model) 方法使用 nn.MSELoss 作为损失函数，optim.SGD 作为优化器。它模拟在 128 × 128 的图像上进行训练，这些图像被组织成 3 个 batches，每个批次包含 120 张图像。然后，我们使用 timeit 运行 train(model) 方法 10 次，并绘制执行时间的标准差，代码如下：

```

import matplotlib.pyplot as plt
plt.switch_backend('Agg')
import numpy as np
import timeit

num_repeat = 10

stmt = "train(model)"

```

```

setup = "model = ModelParallelResNet50()"
mp_run_times = timeit.repeat(
    stmt, setup, number=1, repeat=num_repeat, globals=globals())
mp_mean, mp_std = np.mean(mp_run_times), np.std(mp_run_times)

setup = "import torchvision.models as models;" + \
    "model = models.resnet50(num_classes=num_classes).to('cuda:0')"
rn_run_times = timeit.repeat(
    stmt, setup, number=1, repeat=num_repeat, globals=globals())
rn_mean, rn_std = np.mean(rn_run_times), np.std(rn_run_times)

def plot(means, stds, labels, fig_name):
    fig, ax = plt.subplots()
    ax.bar(np.arange(len(means)), means, yerr=stds,
           align='center', alpha=0.5, ecolor='red', capsize=10, width=0.6)
    ax.set_ylabel('ResNet50 Execution Time (Second)')
    ax.set_xticks(np.arange(len(means)))
    ax.set_xticklabels(labels)
    ax.yaxis.grid(True)
    plt.tight_layout()
    plt.savefig(fig_name)
    plt.close(fig)

plot([mp_mean, rn_mean],
     [mp_std, rn_std],
     ['Model Parallel', 'Single GPU'],
     'mp_vs_rn.png')

```

实验结果：

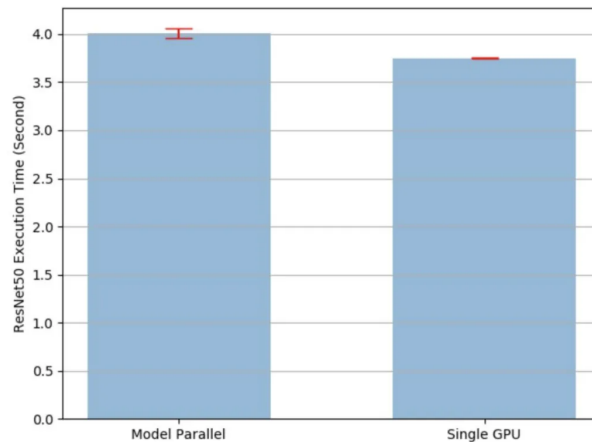


图1：Model Parallel 和 Single GPU 的运行时间比较

结果显示，Model Parallel 实现的执行时间比现有的 Single GPU 实现长 $4.02/3.75-1=7\%$ 。因此，我们可以得出结论，在 GPU 之间来回复制张量的开销大约是 7%。还有改进的余地，如何改进？

3 通过 Pipelining Inputs 加速模型并行

因为我们知道两个 GPU 中的一个在整个执行过程中是闲置的。一个选择是将每个批次的 images 进一步划分为一个个的 splits，这样当一个 split 到达第二个子网络时，下面的 split 可以被送入第一个子网络。通过这种方式，两个连续的 splits 可以在两个 GPU 上同时运行。

要理解这波操作，就得首先学习一个 torch.split 函数：

<https://pytorch.org/docs/stable/generated/torch.split.html>

`torch.split` (tensor, split_size_or_sections, dim=0)

Parameters

- **tensor** (Tensor) – tensor to split.
- **split_size_or_sections** (int) or (list(int)) – size of a single chunk or list of sizes for each chunk
- **dim** (int) – dimension along which to split the tensor.

tensor:<https://pytorch.org/docs/stable/tensors.html#torch.Tensor>

list:<https://docs.python.org/3/library/stdtypes.html#list>

int:<https://docs.python.org/3/library/functions.html#int>

它的作用的官方描述是：Splits the tensor into chunks. Each chunk is a view of the original tensor.

如果 `split_size_or_sections` 是一个整数类型，那么张量将被分割成同等大小的块。如果张量沿着给定的维度 `dim` 的大小不能被 `split_size` 整除，那么最后一个块会更小。

如果 `split_size_or_sections` 是一个列表，那么张量将被分割成 `len(split_size_or_sections)` 个小块，其大小与 `split_size_or_sections` 相同。

举例：

```
>>> a = torch.arange(10).reshape(5,2)
>>> a
tensor([[0, 1],
        [2, 3],
        [4, 5],
        [6, 7],
        [8, 9]])
>>> torch.split(a, 2)
(tensor([[0, 1],
        [2, 3]]),
 tensor([[4, 5],
        [6, 7]]),
 tensor([[8, 9]]))
>>> torch.split(a, [1,4])
(tensor([[0, 1]]),
 tensor([[2, 3],
        [4, 5],
        [6, 7],
        [8, 9]]))
```

接下来，我们回到 `PipelineParallelResNet50` 模型，进一步将每个 batch 的120张图片分成20张图片的 `split`，这步操作可以通过 `splits = iter(x.split(self.split_size, dim=0))` 来完成。

由于PyTorch是异步启动CUDA操作的，因此该实现不需要催生多个线程来实现并发。代码如下，简单梳理一下代码的含义：

在 **forward()** 函数里面做以下这些事情：

对输入的 `batch=120` 的图片分成相同大小为20的 `splits`：

```
splits = iter(x.split(self.split_size, dim=0))
```

从头开始，每次取出一个 `split`：

```
s_next = next(splits)
```

把第1个 split 通过第1段模型：

```
s_prev = self.seq1(s_next).to('cuda:1')
```

for 循环可以看做每次循环做2件事：

A. 前半段模型的输出传到后半段并前向传播：

```
s_prev = self.seq2(s_prev)
```

B. 下一个 split 输入前半段模型：

```
s_prev = self.seq1(s_next).to('cuda:1')
```

```
class PipelineParallelResNet50(ModelParallelResNet50):
    def __init__(self, split_size=20, *args, **kwargs):
        super(PipelineParallelResNet50, self).__init__(*args, **kwargs)
        self.split_size = split_size

    def forward(self, x):
        splits = iter(x.split(self.split_size, dim=0))
        s_next = next(splits)
        s_prev = self.seq1(s_next).to('cuda:1')
        ret = []

        for s_next in splits:
            # A. s_prev runs on cuda:1
            s_prev = self.seq2(s_prev)
            ret.append(self.fc(s_prev.view(s_prev.size(0), -1)))

            # B. s_next runs on cuda:0, which can run concurrently with A
            s_prev = self.seq1(s_next).to('cuda:1')

        s_prev = self.seq2(s_prev)
        ret.append(self.fc(s_prev.view(s_prev.size(0), -1)))

        return torch.cat(ret)

setup = "model = PipelineParallelResNet50()"
pp_run_times = timeit.repeat(
    stmt, setup, number=1, repeat=num_repeat, globals=globals())
pp_mean, pp_std = np.mean(pp_run_times), np.std(pp_run_times)

plot([mp_mean, rn_mean, pp_mean],
     [mp_std, rn_std, pp_std],
     ['Model Parallel', 'Single GPU', 'Pipelining Model Parallel'],
     'mp_vs_rn_vs_pp.png')
```

整个过程可以用下图2表示，其中数字1代表进入for循环前的第1步：`s_prev = self.seq1(s_next).to('cuda:1')`，2-6代表5次for循环，数字7代表最后一步：`s_prev = self.seq2(s_prev)`。

同一个 batch 的数据，有些提前进入后半段模型 Model Part 2，而不用等待全部数据走完前半段模型 Model Part 1之后再统一进入后半段模型 Model Part 2。这样子节约了运行时间

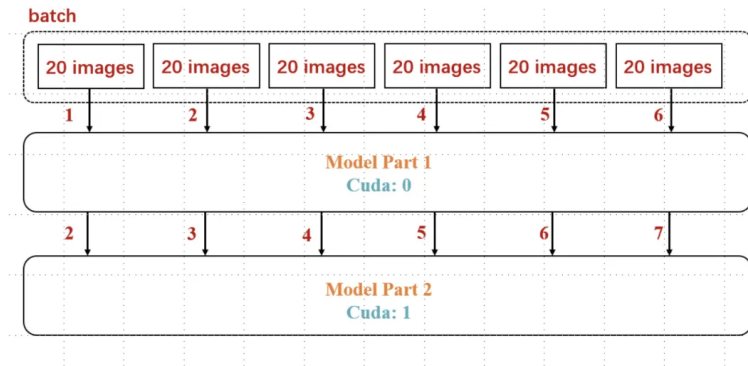


图2：Pipeline Model Parallel过程，同一个 batch 的数据，有些提前进入后半段模型 Model Part 2，而不用等待全部数据走完前半段模型 Model Part 1之后再统一进入后半段模型 Model Part 2。这样子节约了运行时间

实验结果：

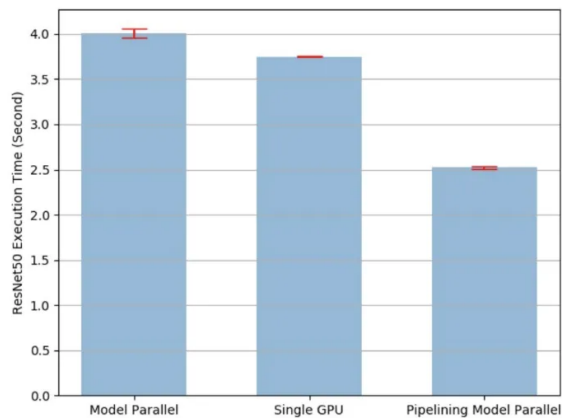


图3：Pipeline Model Parallel, Model Parallel 和 Single GPU 的运行时间比较

实验结果表明，通过流水线输入并行 ResNet50 模型，训练过程大约加快了 $3.75/2.51-1=49\%$ 。这与理想的 100% 的速度仍然相差甚远。由于我们在管道并行实现中引入了一个新的参数 `split_sizes`，目前还不清楚这个新参数对整个训练时间有什么影响。直观地说，使用小的 `split_size` 会导致许多微小的 CUDA 内核启动，而使用大的 `split_size` 会导致在第一次和最后一次分割时出现相对较长的空闲时间。两者都不是最优的。对于这个特定的实验来说，可能会有一个最佳的 `split_size` 配置。让我们通过使用几个不同的 `split_size` 值进行实验来找到它。代码如下：

```
means = []
stds = []
split_sizes = [1, 3, 5, 8, 10, 12, 20, 40, 60]

for split_size in split_sizes:
    setup = "model = PipelineParallelResNet50(split_size=%d)" % split_size
    pp_run_times = timeit.repeat(
        stmt, setup, number=1, repeat=num_repeat, globals=globals())
    means.append(np.mean(pp_run_times))
    stds.append(np.std(pp_run_times))

fig, ax = plt.subplots()
ax.plot(split_sizes, means)
ax.errorbar(split_sizes, means, yerr=stds, ecolor='red', fmt='ro')
ax.set_ylabel('ResNet50 Execution Time (Second)')
ax.set_xlabel('Pipeline Split Size')
ax.set_xticks(split_sizes)
ax.yaxis.grid(True)
plt.tight_layout()
plt.savefig("split_size_tradeoff.png")
plt.close(fig)
```

实验结果：

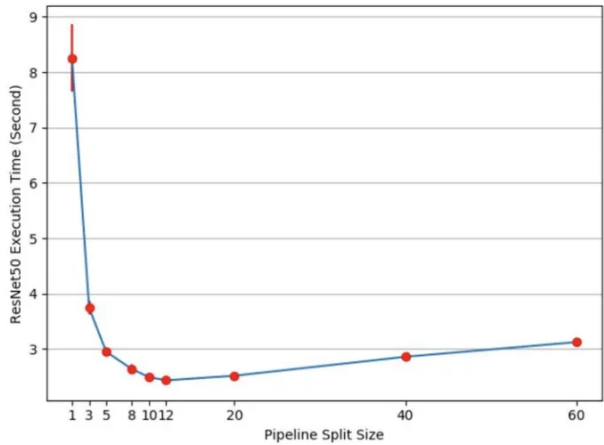



图4：Pipeline Split Size 对加速效果的影响

如上图所示，结果显示，将 split_size 设置为12可以达到最快的训练速度，实现了 $3.75/2.43-1=54\%$ 的速度提升。

总结

本文介绍了模型并行技术 Model Parallel，与 DataParallel 相反，Model Parallel 将一个单一的模型分割到不同的 GPU 上，而不是在每个 GPU 上复制整个模型。Pipeline Model Parallel 是一种进一步加速模型并行的策略。

如果觉得有用，就请分享到朋友圈吧！



极市平台
专注计算机视觉前沿资讯和技术干货，官网：www.cvmart.net
624篇原创内容

公众号

▲点击卡片关注极市平台，获取最新CV干货
公众号后台回复“transformer”获取最新Transformer综述论文下载～

极市干货

- 课程/比赛：珠港澳人工智能算法大赛 | 保姆级零基础人工智能教程
- 算法trick：目标检测比赛中的tricks集锦 | 从39个kaggle竞赛中总结出来的图像分割的Tips和Tricks
- 技术综述：一文弄懂各种loss function | 工业图像异常检测最新研究总结（2019-2020）



极市平台签约作者



科技猛兽

知乎：科技猛兽

清华大学自动化系19级硕士

研究领域：AI边缘计算 (Efficient AI with Tiny Resource)：专注模型压缩，搜索，量化，加速，加法网络，以及它们与其他任务的结合，更好地服务于端侧设备。

作品精选

搞懂 Vision Transformer 原理和代码，看这篇技术综述就够了

用Pytorch轻松实现28个视觉Transformer，开源库 timm 了解一下！（附代码解读）

轻量高效！清华智能计算实验室开源基于PyTorch的视频（图片）去模糊框架SimDeblur

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

年度回顾 | 从九大国际AI顶会接收论文一窥ML算法趋势（上）
当交通遇上机器学习

为迎接超大模型时代，Meta 想要打造“全球最快 AI 超算”
硅星人