运用分治原则提升机器学习推理性能

Alex Kogan
Oracle Labs alex。
kogan@oracle。com

arXiv:2301.05099v2 [cs.LG] 1 Mar 2023

摘要

许多广受欢迎的机器学习模型在 CPU 上部署时扩展性不佳。本文探讨了其中的原因，并提出一种基于著名分治原则的简单而有效的方法，以解决这一具有重要实际意义的问题。对于一项推理任务，我们不再使用所有可用计算资源（即 CPU 核心）来运行它，而是将任务拆分为可独立执行的部分，并根据每部分预期的计算成本分配相应数量的核心。我们在广受欢迎的 OnnxRuntime 框架中实现了这一方法，并通过多个用例评估其有效性，其中包括用于光学字符识别的知名模型（PaddleOCR）和用于自然语言处理的知名模型（BERT）。

1 引言

我们正处于一个研究人员和从业者都前所未有地关注机器学习（ML）的时代。几乎每天都会有涵盖各种领域（或模态，如视频、图像和文本）的新机器学习模型问世；模型规模不断扩大，日益复杂，其组成部分也在持续改进，以在各项任务中取得更高的准确率。尽管人们高度关注训练效率和预测准确率，但似乎较少有人致力于确保这些模型在实际部署时表现良好，也就是在推理阶段 [11]。正如本文所展示的，在基于 CPU 的部署中，随着可用核心数量增加，有些模型的扩展性并不理想（有时甚至会变得更差！）。

为什么 CPU 上的推理无法扩展？原因有很多，本文专门用一整节探讨其中一些。简而言之，原因既可能来自微观层面，例如机器学习架构中使用了不可扩展运算符；也可能来自宏观层面，例如采用了以迭代方式处理输入的机器学习架构。

为缓解这些可扩展性方面的挑战，可以考虑重新设计机器学习架构，或以更高效的实现替换其中无法扩展的操作。然而，这些方法要么需要深厚的机器学习领域专业知识、出色的工程技能以及对用于推理的机器学习框架的熟悉程度，要么需要大量投入（例如重新训练新模型，并可能面临准确率指标下降的风险），也可能两者都需要。

本文采用不同的方法，提出利用机器学习模型可扩展性较差这一特点，应用分治原则——这是一种计算机科学领域中广为人知的算法设计技术 [8]。具体而言，我们不再将所有可用计算资源（CPU 核心）分配给整个问题，而是将问题划分为更小的块[1]，，让框架决定如何在这些块之间分配计算资源，然后并行执行各自的计算。我们认为，在许多使用场景中，这种划分十分自然，只需对用户代码进行极少的修改。我们还介绍了一种简单的机制，可根据每个块预期的计算强度（或权重）分配计算资源。

例如，考虑一个用于解决自然语言处理（NLP）任务（如推文分类）的模型。我们的方法能够高效地对不同大小的推理请求进行批处理，无需填充（这是处理不同大小请求批次时常见但浪费资源的做法），并使框架能够根据每个序列的长度按比例分配计算资源。我们在 OnnxRuntime [24]，（一个广受欢迎的机器学习模型训练与推理框架）中实现了上述分配机制，并扩展其推理 API，使用户代码能够对多个输入发起并行推理。我们通过多个用例展示了该方法的有效性，其中包括广受欢迎的图像处理模型（PaddleOCR [14]）和自然语言处理任务模型（BERT [10]）。

本文其余部分安排如下。第 2 节详细阐述了推理（在 CPU 上）通常无法良好扩展的各种原因。接着，我们在第 3 节介绍分治原则在推理中的概念与实现细节。随后，我们在第 4 节展示该原则可应用于机器学习模型的若干用例，并评估其带来的性能收益。第 5 节讨论相关工作，并在第 6 节总结全文。

[1]我们注意到，与经典分治原理不同， [8], 我们只对问题进行一次划分，不过在某些情况下，也可以递归地将问题划分为越来越小的块，每个块由一个线程执行。

2 为什么推理速度慢?

造成可扩展性不足的原因有很多。本节将概述其中一些原因。

2.1 工作量“不够”

其中一个原因很简单：模型在推理期间所需的计算量“不够”，无法实现高效的并行化。正如 Aminabadi 等人 [3],所指出的，各种机器学习运算的内核实现通常针对训练进行了优化，而训练往往包含大量批次的大型输入（例如，包含512 个词元的句子）。然而，在推理期间，批次往往小得多，而且通常只包含一个输入（例如，实时/交互式推理）。此外，输入本身也可能很短，例如一条推文或仅由几个词组成的聊天机器人交互。

例如，自然语言处理任务中广受欢迎的 Transformer 模型 [28]，如 BERT [10] 或 GPT-3 [5],，主要（但并非完全）依赖矩阵乘法这类基本运算。这些基本运算已知在处理大型矩阵时具有良好的扩展性 [11, 22, 30]。然而，当推理期间模型的实际输入较短时，矩阵乘法所涉及的矩阵会更小，因此不太适合进行高效并行化 [11, 23, 30]。

2.2 不可扩展运算符

一些机器学习模型可扩展性较差的另一个原因，是其架构中使用了不可扩展（且通常为顺序执行）的算子。通常，与模型中其他可扩展性更强的部分相比，这些算子的开销可以忽略不计。然而，随着核心数量增加，根据阿姆达尔定律 [2], ，不可扩展算子的负面影响会随之增大，从而降低整体推理性能。再次以此前提到的基于 Transformer 的 [28] 模型为例，Dice 和 Kogan 观察到，矩阵乘法至少在输入较长时具有良好的可扩展性，而层归一化和 Softmax 等其他操作则不然，导致这些模型整体可扩展性较差 [11]。本文考察了一种基于视觉的模型，其中用于内部格式转换的函数采用顺序方式实现，同样导致整个模型无法扩展。

我们注意到，其中一些情况可以视为底层机器学习框架中的性能缺陷，可通过用更高效（且可并行）的替代方案重新实现相应算子来修复。不过，这需要投入大量工程工作，包括性能分析以及深入理解相应框架的实现细节。此外，某些机器学习算子（例如层归一化 [4], ）需要计算线程之间进行谨慎协调（例如，计算某一层所有隐藏单元的方差和标准差，然后利用这些统计量对各单元的值进行归一化），因此难以自然地实现高效并行化。

2.3 框架开销

与前一点略有关联的是，机器学习框架在调用模型算子时可能会增加少量但可测量的开销。大多数流行的机器学习框架，例如 PyTorch、TensorFlow 或 OnnxRuntime，都支持多个用于执行机器学习运算的后端，以适配不同的硬件架构（CPU、GPU、TPU），并使用不同的 BLAS 库（MKL、OpenBLAS、oneDNN 等）、不同的线程基础设施（Intel TBB、pthreads、自定义实现等）。为每个算子分派合适的内核（实现）虽然高效，但这一过程是顺序执行的，且需要相当多的工作，尤其是在模型以交互式 [11] 方式执行时（这是 PyTorch 的默认执行模式）。随着核心数量增加，内核的实际执行时间缩短，这种开销便会变得显著。

除此之外，各种内核可能要求其输入参数（张量）采用特定的内存布局，而框架会添加适当的虚拟算子来执行输入/输出转换或数据准备 [30]。正如我们将在本文后文中展示的那样，这些算子也可能带来显著的开销。

2.4 模型架构

机器学习模型本身的高层架构往往会导致推理无法实现良好的扩展。例如，一些机器学习模型，尤其是用于视频和图像处理的模型（例如 [14, 21, 29]），采用多阶段流水线构成。流水线的第一阶段通常会识别输入中的感兴趣点（例如图像中的文本框或视频中的运动物体），而后续阶段则会处理这些点（逐个处理或按批次处理），以解决预先定义的问题（例如识别文本框中的文本，或对视频中的运动物体进行分类）。这类模型的推理延迟可能会随着第一阶段识别出的物体数量线性增长。此外，如果流水线中哪怕只有一个阶段无法良好扩展，整个流水线的可扩展性都会受到影响。

2.5 填充

将多个输入组成批次并一次性处理，是提升推理吞吐量的一种广为人知的方法 [1, 3, 9,15, 32]。事实上，许多机器学习模型服务系统（如 TensorFlow Serving [16] 或 TorchServe [7]）都包含可调参数，用于配置推理服务器等待的时长，以便尽可能多地对输入请求进行批处理。然而，当一个批次中的输入形状不完全相同时，就需要对其进行填充，才能高效处理，因为底层内核通常要求批次中的输入彼此同质。填充会导致

计算效率降低，因为内核会像处理输入的其余部分一样处理这些填充内容，尽管模型生成的相应输出会被丢弃。

3 将分治原则应用于推理

本节从概念层面介绍如何将分治原则 [8] 应用于机器学习模型的推理，并说明如何通过在 OnnxRuntime 框架中实现该原则，将其具体落地。需要指出的是，应用这一原则并不能直接解决上一节详述的可扩展性不佳问题。事实上，我们方法的优势在于，无需识别和／或修复模型中的任何可扩展性瓶颈，就能从其核心思想中获益。

3.1 概念

基本思想相当简单——考虑一个计算任务 J，它可以拆分为 k 个相互独立的部分 j_1、j_2、……、 j_k，并行执行。假设有一个预言机，为各部分分配相对权重 $w_i \in (0, 1]$ 例如，可根据所需浮点运算次数（FLOPs）或该计算任务部分的单线程延迟来确定 j_i）。最后，假设有 C 个可用计算核心。我们力求根据各部分的权重为其分配相应数量的核心，也就是说，为部分 j_i 分配 $c_i = max\{1, \lfloor w_i * C \rfloor\}$ 个核心。这实际上意味着为 j_i 分配 c_i 个工作线程，因为我们之后会为每个核心创建一个工作线程（这在包括 OnnxRuntime 在内的机器学习框架中很常见）。

请注意，$\sum_{i=1}^{k} c_i$ 可能大于 C。当作业部分的数量 k 大于 C 时，这显而易见；但即使 $k \leq C$，也有可能出现这种情况。除了意味着某些作业部分要等其他作业部分完成后才能运行（而不是让它们全部并行运行）之外，这不会造成其他问题。同时，由于向下取整（floor）函数旨在降低上述超额分配的可能性，因此可能会有一些未分配的核心剩余。为避免浪费可用资源，我们按剩余未分配权重对所有作业部分排序，即按 $w_i * C - \lfloor w_i * C \rfloor$ 排序，然后按降序依次为每个部分分配一个核心，直至分配完所有核心。整个算法的类 C++伪代码见清单 1。

自然，上述思路引出了一个问题：如何为作业部分 j_i 分配相对权重。在第 4 节讨论的所有情况中，权重都只是按输入张量的大小成比例设置。具体而言，设 s_i 为作业部分 j_i 的输入张量大小。我们将 w_i 设为 $\frac{s_i}{\sum_{i=1}^{k} s_i}$，这实质上是假设计算量（以所需浮点运算次数表示）大致随输入张量的大小线性增长。不过，一般而言，可以借助性能分析阶段和轻量级分类机制来分配权重；该机制会将形状相同（或相似）的作业部分关联起来（例如当前遇到的

```
1 向量<整型> 分配(向量<张量> 输入, 整型 核心数) { 2 向量<整型> 线程分配; 3 向量<元组<整型, 浮点数>> 线程未分配权重; 4 整型 输入数量 =输入.大小(); 5 整型 已分配核心数 = 0; 6 整型 索引 = 0; 7 整型 总大小 = 0; 8 对于 (自动 j_i : 输入) 总大小 += j_i.大小() 9 对于 (自动 j_i : 输入) { 10 整型 使用线程数 = 1; 11 如果 (输入数量 <=核心数) { 12 整型 大小 =j_i.大小(); 13 浮点数 w_i =((浮点数)大小) / 总大小; 14 使用线程数 = 向下取整(w_i * 核心数); 15 // 这可能是由于向下取整而发生 16 如果 (使用线程数 < 1) 使用线程数 = 1; 17 未分配权重.添加( 18 创建_元组(索引, w_i * 核心数 - 使用线程数)); 19 } 20 线程分配.添加(使用线程数); 21 已分配核心数 +=使用线程数; 22 索引++; 23 } 24 如果 (已分配核心数 <核心数) { 25 // 按降序对向量排序， 26 // 比较每个元组中的第二字段 27 排序(未分配权重, 按第二字段); 28 整型 下一个待调整项 = 0; 29 当(已分配核心数 < 核心数) { 30 // 获取 `下一个待调整项` 元组中的第一个字段 31 索引 =32 未分配权重[下一个待调整项 % 输入数量].获取(0); 32 线程分配[索引]++; 33 已分配核心数++; 34 下一个待调整项++; 35 } 36 } 37 返回 线程分配; 38 }
```

清单 1. T hread all分配 a l算法hm

（在性能分析阶段）与性能分析期间获得的相对权重。

3.2 实现细节

我们为 On- nxRuntime 的 InferenceSession 类扩展了应用程序接口，新增了一种修剪方法。该方法基于

```
1 类 TextRecognizer(object)： 2 def __init__(self, args)： 3 ... 4 self.predictor =ort.InferenceSession(args.file_path) 5 self.postprocess_op =build_post_process(args) 6 ... 7 def __call__ (self, img_list)： 8 img_num = len(img_list) 9 forbeg_img_no in range(0, img_num, 批次_num)： 10 end_img_no = min(img_ num, beg_img_no +批次_num) 11 输入 =prepare(img_list, beg_ img_no, end_img_no) 12 outputs =self.predictor.run(输入) 13 preds =outputs[0]14 rec_result =self.postprocess_op(preds) 15 all_results.add(rec_result) 16 return all_results
```

清单 2. 原始（为清晰起见，已缩短并编辑）PaddleOCR 中的 TextRecognizer 类实现

```
1 类 TextRecognizer(object): 2 def __init__(self, args): 3 ... 4 self.predictor =ort.InferenceSession(args.file_path) 5 self.postprocess_op =build_post_process(args) 6 ... 7 def __call__ (self, img_list): 8 img_num =len(img_list) 9 for beg_img_noin range(0, img_num, batch_num): 10 end_img_no = min(img_ num, beg_img_no +batch_num) 11 inputs =prepare(img_list, beg_img_no, end_img_no) 12 all_inputs.append(inputs) 13 all_ outputs =self.predictor.prun(all_inputs) 14 for outputs in all_ outputs: 15 preds =outputs[0]16 rec_result =self.postprocess_ op(preds) 17 all_results.add(rec_result) 18 return all_results
```

清单 3. 修改后的`TextRecognizer` 类实现（使用 `prun`）。新增或修改的行以红色标示

在现有的 run 方法之后，该方法用于运行推理时作为主要入口点。主要区别在于，prun 接受一个输入列表（而不是单个输入），并返回一个输出列表。

在内部，prun 的实现会遍历输入列表，计算每个输入的大小（在验证其为张量之后）及相应的相对权重，并应用清单 1 中所述的分配算法，为每个输入（作业部分）分配工作线程数量。随后，该实现会为每个输入创建一个工作线程，并让它们并行运行。每个工作线程又会创建一个大小由分配算法计算得出的线程池（线程池包含该工作线程本身），并使用该线程池调用 InferenceSession 对象的 run 方法。为实现 prun 功能及进行其他细微的内部更改（例如让 run 方法接受线程池作为可选参数，而不是始终使用默认线程池），OnnxRuntime 代码库的整个补丁约有 200 行代码。

在用户端，代码也需要进行修改，以便使用新的 prun API。不过，这些修改相当简单。用户不再需要为每个作业都调用 run，而是创建一个作业部分列表并调用 prun。此外，用户还需要调整后处理代码，使其遍历 prun 的结果，并对每个返回的输出（对象）应用相应的后处理。为说明用户代码需要做哪些改动，我们展示了 PaddleOCR 中 TextRecognizer 类的原始 Python 代码（为简洁明了起见做了编辑），（清单 2），以及使用新的 prun API 的修改版本（清单 3）。

4 个用例

在详细介绍分治原则有益的用例并报告我们的性能发现之前，我们先简要总结评估环境和方法。所有实验均在 Oracle Cloud 中一台基于 AMD 的 16 核 VM 上运行（即 OCI VM.Standard.E3.Flex）。（我们也在较新的 E4 实例规格上运行了一些实验，但未发现显著差异。）为降低性能波动，尤其是针对使用 prun 的变体创建独立线程池时，我们对所有评估变体都采用线程绑定（固定线程）。每项实验均重复 5 次，并报告平均值。我们注意到，除下文讨论的一个特定情况外，所有报告结果的标准差都极低（通常低于均值的 1%）。实验中，我们使用了相应软件截至本文撰写之日的最新发布版本，具体为 OnnxRuntime v1.11.1 和 PaddleOCR v2.5。

4.1 顺序流水线

第一个能充分体现分治原则极为有用的例子是 PaddleOCR [14]。PaddleOCR 是一个轻量级 OCR 系统，由三个部分组成：文本检测、文本分类（在 [14] 中称为检测框校正）和文本识别。这些部分分别对应一个独立的机器学习模型。

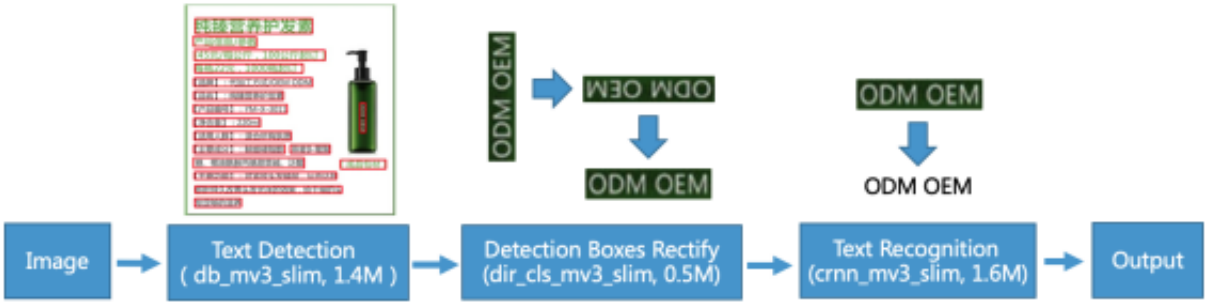


图 1. PaddleOCR 三阶段流水线（根据 [14]中的图 2 修改）。

OCR 流水线接收一个图像文件，并首先将其传入文本检测阶段，该阶段旨在定位图像中的文本区域。此阶段的输出是一个潜在文本框坐标列表。接下来，系统遍历该列表，并将其中的每一项（即一个文本框）发送给文本分类模型，由该模型判断是否需要先将文本框转换为水平矩形框，再进行实际的文本识别。系统根据分类器的判断，分别调整每个文本框。最后，系统再次遍历该列表，并将每一项发送给文本识别模型进行推理，由该模型识别给定文本框中的文本，并根据提供的字符字典生成实际的字符序列。此过程如图 1 所示，该图是对 [14] 中图 2 的删节版本。

在使用 PaddleOCR 进行的实验中，我们发现，随着可用核心数量增加，系统的扩展性并不理想。图 2 展示了推理延迟随可用核心数量变化的情况（可用核心数量直接对应运行时使用的工作线程数量）。本节中的所有实验（包括图 2 中的实验）均使用从 OpenI mages 数据集中选取的图像子集 [17], , 其筛选标准将在下文说明。

在图 2 中，我们将总延迟拆分为与上文讨论的 OCR 流水线三个阶段相对应的时间段。可以看到，平均推理延迟从 1 个线程时的 554 ms 降至 4 个核心时的 364 ms，随后又在 16 个核心时回升至 435 ms。有趣的是，文本分类阶段表现出负可扩展性：使用 1 个线程处理一张图像平均需要 27 ms，而使用 16 个线程处理同一图像则需要 38 ms，速度降低了 1.4 倍。这表明，系统在超过某个限度后，增加线程不仅无益，反而会降低性能。探讨这些特定模型缺乏可扩展性的具体原因不在本文讨论范围之内。不过，对于感兴趣的读者，我们注意到，内置的 OnnxRuntime 性能分析工具显示，输出重排序算子的执行时间偏高（这些算子由框架与输入重排序算子一同插入，用于转换各种内核的输入参数的内存布局）。

我们将分治原则应用于 OCR 流水线的最后两个阶段，即文本分类和识别。为此，我们不再针对文本检测生成的每个文本框分别调用相应模型，而是将所有文本框发送给运行时（通过调用 prun API），由运行时根据每个文本框的相对大小，决定为其分配多少核心／工作线程。清单 3 展示了在文本识别阶段实现此功能所需的更改；文本分类阶段的更改与此类似。

在性能评估中，我们将第 3 节讨论（并在清单 1 中展示）的 prun 实现（图表中记为 prun-def）与几个简单的变体进行比较。第一个变体记为 prun-1，它为传递给 prun 的列表中的每个输入分配一个工作线程。第二个变体记为 prun-eq，它为每个输入分配相同数量的核心（但至少为一个），即设置 $c_i = max\{1, \lfloor k/C \rfloor\}$。我们的目的是说明，简单的解决方案在某些情况下也可能有用（如下文所述），但与 prun-def 相比，它们往往表现较差。

我们注意到，在此用例中，只有在文本检测阶段识别出至少两个文本框时，prun 才能带来收益。否则，另外两个阶段将不会被使用（未检测到文本框时），或者 prun-def 变体将使用与 base（未修改）版本相同的（最大）核心数（仅检测到一个文本框时）。因此，本节用于性能评估的图像子集包括至少包含两个已识别文本框的图像。图3中的饼图展示了整个数据集在 OCR 流水线第一阶段实际检测到的文本框数量分布。该数据集共包含 500 张图像——选择这一数量是为了使评估时间保持在合理范围内。（我们注意到，我们还在一个包含少于两个文本框的图像的大型数据集上进行了评估，并确认在这些情况下使用 prun 不会产生任何开销。）

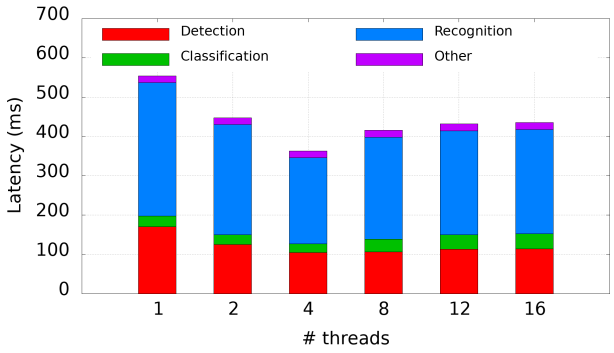


图 2。PaddleOCR 在不同线程数下的推理延迟，按流水线的三个阶段进行细分。

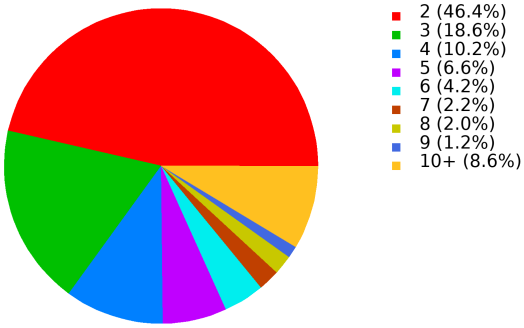


图3。 输入数据集中检测到的文本框数量分布。

鉴于上述讨论，我们按照检测框数量对延迟结果进行细分比较，如图4所示。该图中的延迟数据是在使用 16 个核心时收集的；我们将在后文讨论整体可扩展性趋势。我们还对使用 prun 的两个阶段的性能进行了细分，即文本分类（图4 (a)）和识别（图4 (b)）。

结合图4中的结果可以发现，正如预期，检测到的文本框越多，prun 带来的收益越大。例如，考察端到端总延迟（图4 (c)）时，只有两个检测框的情况下，prun-def 的表现比 base 快 1。28 倍。然而，在有 9 个和10+ 个检测框时，prun-def 的表现分别比 base 快 2。33 倍和 1。81 倍。

将 prun-def 与其他基于 prun 的变体进行比较很有意思。如图4 (a) 所示，当检测框数量较少时，prun-1 变体的延迟最低。事实上，在这种情况下，base 变体的表现也优于 prun-def。我们认为这主要有两个原因。首先，流水线的这一特定阶段表现出负可扩展性，这一点在图 2 中也可以看到。因此，在该阶段，每个检测框使用较少线程时性能最佳，而 prun-1 实际上正是如此。其次，在这些情况下，prun-def（以及 prun-eq）创建和销毁的线程比 prun-1 更多，因为每次 prun 调用都会创建包含更多线程的线程池。考虑到该阶段的执行时间很短，这会带来少量但不可忽视的开销。未来，我们计划尝试在多次 prun 调用之间复用线程池。然而，随着检测框数量增加，所有 prun 变体为每个检测框分配的线程都会减少（甚至只分配 1 个），且为线程池分配的线程数也趋于相近，因此与 prun-1 变体之间的差距逐渐缩小。

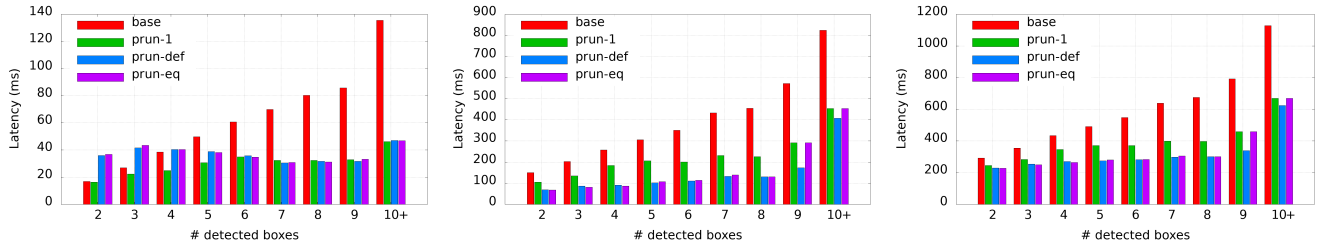
不过，就文本识别阶段而言（参见图4（b）），从图 2 中显然可以看出，使用多个线程可以降低其延迟。值得注意的是，从量化结果来看，该阶段的占比也远高于文本分类阶段。在这里，对于所有检测框数量，prun-def 的表现都达到或接近最佳，因此其端到端推理性能整体上极具竞争力（参见图4（c））。总体而言，图4中的结果表明需要一种动态机制，根据给定的工作负载和可用资源选择最佳线程分配策略。设计并试验此类策略将留待未来工作。

最后，我们通过图5进一步展示使用 prun 后可扩展性的提升情况。在图中，我们改变了 OnnxRuntime 可用的核心数量（因此也改变了工作线程总数）。与前面一样，我们列出了 PaddleOCR 最后两个阶段的延迟（分别记为 Rec，即文本识别，以及 Cls，即文本分类），以及端到端（Total）延迟。为便于说明，我们仅列出 base 和 prun-def 变体的结果（在图5中统称为 prun）。

总体而言，可以观察到与上文讨论相似的趋势。在 base 版本中，文本识别阶段的性能确实会随着线程数增加而提升，直到 4 个线程；但此后，随着线程数继续增加，性能反而会下降。prun 变体避免了这种性能退化，实际上，在最多 16 个线程时仍能持续提升性能。事实上，仅考虑文本识别阶段时，prun 变体在 16 个线程下的表现比 base 高出 2.4 倍。不过，由于两个变体的文本检测阶段完全相同，而且根据图 2，该阶段占总延迟的相当大一部分，因此 prun 在 16 个线程下的端到端加速仅为 1.5 倍。

4.2 异构输入的批处理

下一个例子涉及 Transformer 架构 [28], 。该架构于 2017年 推出，彻底改变了自然语言处理领域，此后也被应用于其他领域（例如， [12, 18]）。这一架构由多层堆叠而成，每层都由一个自注意力模块和一个全连接网络组成 [28]。以往研究表明



(a) 文本分类 **(b)**文本识别 **(c)** 端到端推理

图4. 在 PaddleOCR 中使用 prun 的影响。

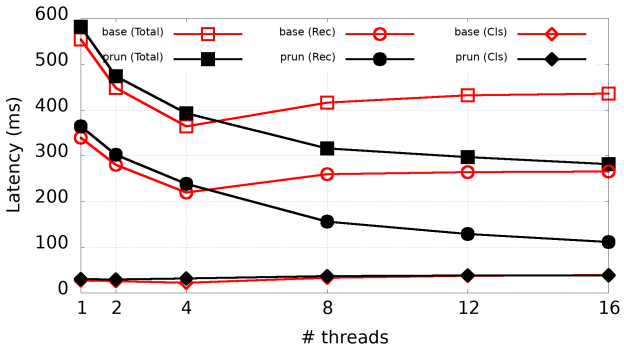


图5. PaddleOCR 在不同线程数下的总（端到端）推理延迟。图中还展示了文本分类（Cls）和文本识别（Rec）阶段的延迟

Transformer 中的大部分计算周期都用于（可扩展的）矩阵乘法操作，但仍有多达三分之一的周期用于其他操作（即可扩展性较差的操作） [11]。

众所周知，提高 Transformer 推理性能（尤其是吞吐量）的一种方法是输入批处理 [3, 15, 30]。不过，当输入长度相同时，这种策略效果很好。否则，要么放弃批处理，要么将输入填充到相同长度。后者会导致计算周期浪费，因为架构会将特殊填充标记与输入词元完全同等处理，并在计算结束时将其丢弃。

这种情况非常适合应用分治原则。我们无需将批次中长度各异的输入填充到最长输入的长度，而是可以使用 prun API 对这些输入（保持原样，不进行填充）运行推理，并由运行时决定处理每个输入时应使用多少个核心。我们修改了 OnnxRuntime [25] 内置的 Transformer 基准测试，以实现这一策略。

为了评估上述方法的有效性，我们设计了一项实验，生成长度从范围 $[16, 512]$ 中均匀随机选取的输入 X。随后，我们比较填充批次版本与该版本会将所有 X 输入填充至给定批次中的最长长度；而 prun 版本则会对批次中的所有输入调用 prun 进行推理。我们展示了广受欢迎的 BERT 模型 [10] （具体来说，是“bert-based-uncased”）的结果。我们还使用其他基于 Transformer 的模型（例如“bert-large-uncased”或“roberta-base”）进行了实验，测得了类似的定性结果。

需要注意的是，这项实验存在固有的随机性——抽取一批短句子与抽取一批长句子的可能性相同。为了尽量降低预期中较高的结果方差，我们选择重复实验 1000 次，因此，对于每个 X，每个数据点都是 1000 次结果的平均值。图 6 展示了不同批次大小下的吞吐量结果（即 X 从 2 变化到 8），误差线表示报告均值的标准差。尽管 prun 在所有批次大小下的表现都优于填充批处理版本，但测量结果的方差仍然异常高。

因此，我们以更可控的方式设置了另外两个实验，以期获得更稳定的结果。在第一个实验中，我们只需预先设定每个批次中各个序列的长度。例如，标记为“16-64-256”的批次包含三个句子，长度分别为16、64和256个词元。我们在图7中展示了该实验的结果。在这里，prun 版本轻松胜过填充批处理版本，后者必须将所有序列填充到批次中最长序列的长度。正如预期的那样，批次中的句子数量越多，使用 prun 的收益越大，因为该版本消除了与填充相关的所有冗余工作。

在第二个实验中，我们使用一个包含1个长句子（256个词元）和 X 个长度均为16个词元的短序列的批次，并将 X 的取值设为0到15。我们在图7中展示了该实验的吞吐量结果，同时还绘制了一条曲线，表示 prun 为批次中的长序列分配的线程数。

这里可以得出几个有趣的观察结果。首先，当 X=0，也就是说，批次中只有一个长句子时，两个变体都会使用所有可用核心来处理该批次，产生相似的结果。这表明

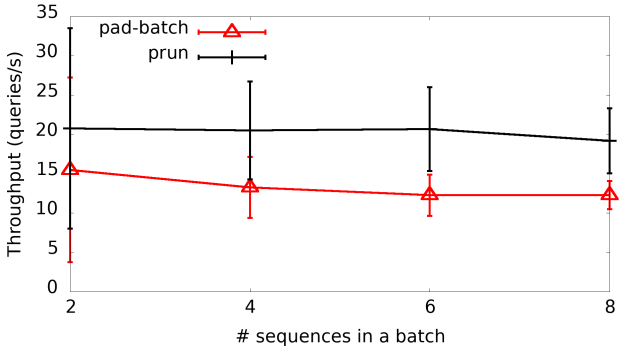


图6. 在批次中对序列进行推理时，BERT 的吞吐量；序列大小从范围 $[16, 512]$ 中随机选取

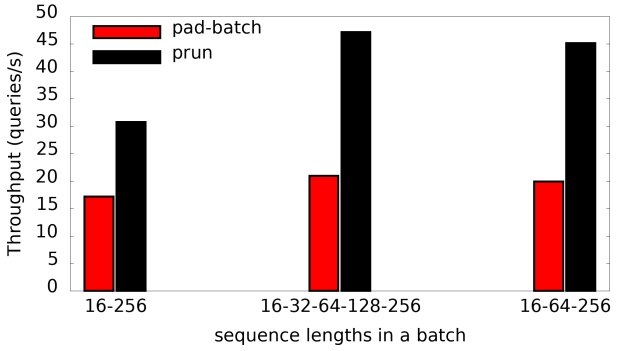


图7. 在批次中对不同预设大小的序列进行推理时，BERT 的吞吐量

当输入只有一个块时，使用 prun 的开销可以忽略不计。其次，随着短序列数量增加，pad-batch 版本的吞吐量有所提升，但幅度不大。这是因为，如上所述，较大的（填充后）序列批次有助于 Transformer 实现更高的吞吐量。与此同时，prun 的吞吐量在批次中短序列数量达到 3 个之前增长得更为显著，之后有所下降，但仍远高于 pad-batch 的吞吐量。这两种现象都可以这样解释：使用 16 个线程推理一个包含 256 个词元的序列，所需时间与使用 13 个线程时大致相同。因此，在批次中加入几个短序列，每个序列只分配 1 个线程（因为它们的相对权重较小），对延迟的影响可以忽略不计，却能提高吞吐量。当批次中的短序列更多时，分配给长序列的线程就会减少（如图7所示），其推理延迟也会增加。这会导致整体吞吐量下降。

4.3 同质输入的批处理

我们的最后一个例子直接承接第 2 节中关于机器学习模型缺乏可扩展性的讨论。正如前面所提到的，尽管 Transformer 模型大量使用

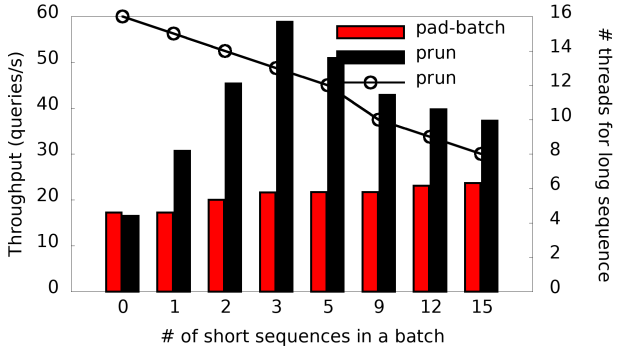


图8. 在一个批次中对 BERT 进行推理的吞吐量：该批次包含一个含 256 个词元的长句子，以及 X 个各含 16 个词元的短序列，其中 X 从 0 变化到 15。此外，我们展示了在 prun 变体中，有多少线程专门用于对批次中的一个长句子进行推理

可扩展的矩阵乘法操作，但它们也会采用可扩展性较差的操作。随着核心数量增加，后者的影响也会加大。因此，即使批次中的输入长度相同，将分治原则应用于 Transformer也可能带来益处。

举一个具体例子，考虑由两个输入组成的批次。我们不会使用所有可用核心来处理该批次，而是将一半核心分配给每个输入。直观而言，使用较少核心且输入序列较短（即词元数量为整个批次的一半）时，可扩展性较差的算子产生的相对开销会更小。

图9通过长度相同的输入批次展示了这一效果。除了填充批处理版本（由于无需填充，我们在此简称为批次）和 prun 之外，我们还加入了无批次变体，它会依次对给定批次中的每个序列进行推理。请注意，加入后一变体只是为了展示批处理的普遍优势，并印证先前的研究结果 [3, 15, 30]。图9中的每组柱状图对应一个由4个句子组成的批次，其句子长度各不相同（从64个词元到512个词元）。总体而言，与第 4.2 节中输入不同质的情况相比，prun 版本相较于批次带来的加速幅度较小（但仍不可忽视）。这是预期之中的，因为在这种情况下，相较于批次，可改进的空间不包括与填充相关的冗余计算。

5 相关工作

如引言中所述，机器学习社区的主要关注点一直是提高所提出模型的准确率和训练性能，而高效地对这些模型进行推理和服务则相对较少受到关注。不过，也有一些值得关注的研究专门聚焦于推理性能，

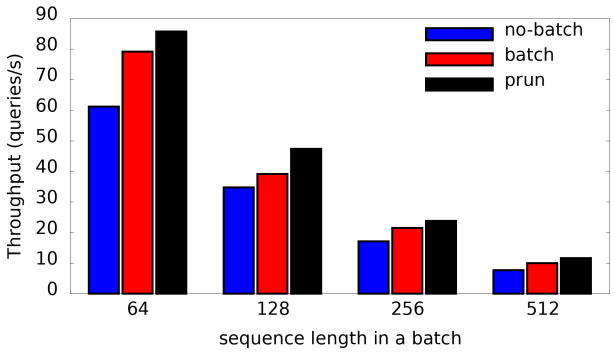


图 9. 以每批 4 个等长序列进行 BERT 推理的吞吐量

下面我们将综述其中最相关的成果。顺便指出，以下许多成果来自各家公司发布的非正式博客文章，这凸显了高效推理在实践中的重要性。

Wang 等人 [30] 探讨了影响 TensorFlow 推理性能的各种因素，包括特定数学库的选择、线程池库、是否支持单指令多数据（SIMD）等。他们指出，数据准备是导致小型矩阵乘法操作可扩展性不佳的原因之一；我们在第 2 节中则将此类问题更普遍地归因于框架开销。他们提出了一系列指南，可用于调整 TensorFlow 设置，使其性能优于采用 TensorFlow 作者或英特尔推荐设置时的表现。

随着 Transformer 日益普及，许多论文和博客文章都开始关注其推理性能。Dice 和 Kogan 研究了 Transformer 在 CPU 上的推理性能 [11]。他们的分析表明，大多数推理计算周期都耗费在矩阵乘法操作上。因此，他们提出了自适应矩阵乘法优化方案，旨在降低这些操作的延迟，进而提升整体推理性能。英特尔工程师介绍了使用 GluonNLP 工具包在 Apache MXNet 中优化 BERT 推理性能的工作，其中一种思路是对模型进行量化，以较低精度换取更高性能 [31]。Roblox 也采用了类似的量化方法（以及蒸馏——另一种常见的模型压缩方法 [27]），以加速其在 CPU 上部署 BERT [19]。同一篇博客文章还提到，消除输入句子的填充可以提升性能（不过作者仅针对输入数量为 1 的批次进行了处理）。微软团队 [26]介绍了通过 OnnxRuntime 加速 BERT 的工作：借助算子融合，减少逐个调用每个内核时产生的开销（例如内存复制）。

近期有一些论文和项目研究了异构输入的填充效率问题。Fang 等人 [15]提出了序列长度感知批次调度器，旨在将大小相近的请求组成批次，从而降低将所有请求通过零填充合并到一个批次中的成本。该方法需要一个性能分析阶段，在此期间收集各种批次的推理成本。Du 等人 [13] 提出对 Transformer 使用的 GPU 内核进行精心重新设计，以消除与零填充相关的大部分冗余计算。字节跳动的 Effective Transformer 项目 [6]旨在不同计算阶段动态移除和恢复填充。所有这些工作都专门针对 GPU 上 Transformer 的推理，目前尚不清楚它们在 CPU 和/或其他架构上的效率如何。

除 Transformer 之外，Liu 等人 [20]介绍了 NeoCPU，这是一种用于优化 CPU 上卷积神经网络推理的方法。NeoCPU 提出了一种可配置的高效卷积运算设计，可针对主流 CPU 进行高效调优。该设计还结合了一种方案，用于为卷积神经网络模型中的不同操作确定最佳内存布局，以尽量减少在各个操作之间转换数据所产生的开销。

6 讨论

本文讨论了机器学习模型推理缺乏可扩展性的各种原因。这些原因涉及从微观到宏观的不同层面，但共同点是，现有机器学习框架的设计侧重于实现高训练性能。这一点体现在：常见操作的内核通常针对大批次和长输入进行优化，而忽略了框架各个部分中相对较小的开销，因为这些开销对整体训练性能无关紧要。然而，在推理过程中，批次往往小得多，输入也更短，因此这些开销会更加显著。Aminabadi 等人也提出了某种类似的观察结果 [3]。

我们利用这种较差的可扩展性，提出了一种简单而有效的方法：将给定输入拆分为多个块，并并行处理每个块，而不是将所有可用资源用于处理整个输入。正如我们通过几个知名模型所展示的那样，这种方法能够提升推理可扩展性，最终可使延迟和吞吐量提升两倍以上。

这项工作为未来研究提供了几个方向。首先，我们希望探索更动态的线程分配策略，例如能够更好地适应以下情况的策略：工作块的权重与其大小并非线性相关，和/或底层模型在单线程运行时表现最佳。其次，我们希望找到自动将输入拆分为可并行处理的分块的方法，进一步降低使用 prun 的成本（以用户代码改动量计）。最后，我们希望

探索其他适合使用 prun 的场景，包括采用基于流水线架构的其他机器学习模型（例如，[21, 29]）。

致谢

作者谨此感谢 Dave Dice 对本文早期草稿提出的宝贵意见。

参考文献

[1] Ahsan Ali、Riccardo Pinciroli、Feng Yan 和 Evgenia Smirni。Batch：采用自适应批次处理的无服务器平台机器学习推理服务。载于高性能计算、网络、存储与分析国际会议（*SC*）论文集，2020年。[2] Gene M. Amdahl。单处理器方法在实现大规模计算能力方面的有效性。载于春季联合计算机会议（*AFIPS*）论文集，第483–485页，1967年。[3] Reza Yazdani Aminabadi、Samyam Rajbhandari、Minjia Zhang、Ammar Ahmad Awan、Cheng Li、Du Li、Elton Zheng、Jeff Rasley、Shaden Smith、Olatunji Ruwase 和 Yuxiong He。DeepSpeed 推理：实现前所未有规模的 Transformer 模型高效推理。*CoRR*，abs/2207.00032，2022年。[4] Jimmy Lei Ba、Jamie Ryan Kiros 和 Geoffrey E. Hinton。层归一化。*CoRR*，abs/1607.06450，2016年。[5] Tom B. Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell、Sandhini Agarwal、Ariel Herbert-Voss、Gretchen Krueger、Tom Henighan、Rewon Child、Aditya Ramesh、Daniel M. Ziegler、Jeffrey Wu、Clemens Winter、Christopher Hesse、Mark Chen、Eric Sigler、Mateusz Litwin、Scott Gray、Benjamin Chess、Jack Clark、Christopher Berner、Sam McCandlish、Alec Radford、Ilya Sutskever 和 Dario Amodei。语言模型是少样本学习器。载于神经信息处理系统进展（*NeurIPS*），2020年。[6] 字节跳动。Effective Transformer。https://github.com/bytedance/effective transformer。访问日期：07-29-22。_[7] PyTorch Serve 贡献者。TorchServe。https://pytorch.org/serve。访问日期：07-28-22。[8] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein。算法导论，第*3*版。麻省理工学院出版社，2009年。[9] Daniel Crankshaw、Xin Wang、Giulio Zhou、Michael J. Franklin、Joseph E. Gonzalez 和 Ion Stoica。Clipper：低延迟在线预测服务系统。载于*USENIX* 网络系统设计与实现研讨会（*NSDI*），第613–627页，2017年。[10] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。BERT：用于语言理解的深度双向 Transformer 预训练。载于*NAACL-HLT* 论文集，第4171–4186页，2019年。[11] Dave Dice 和 Alex Kogan。优化 CPU 上 Transformer 的推理性能。载于机器学习与系统研讨会（*EuroMLSys*），2021年。[12] Alexey Dosovitskiy、Lucas Beyer、Alexander Kolesnikov、Dirk Weissenborn、Xiaohua Zhai、Thomas Unterthiner、Mostafa Dehghani、Matthias Minderer、Georg Heigold、Sylvain Gelly、Jakob Uszkoreit 和 Neil Houlsby。一张图像胜过 16×16 个词：大规模图像识别的 Transformer。*ICLR*，2021年。[13] Jiangsu Du、Jiazhi Jiang、Yang You、Dan Huang 和 Yutong Lu。处理 GPU 上 Transformer 推理的重尾输入。载于*ACM* 超级计算国际会议（*ICS*）论文集，2022年。[14] Yuning Du、Chenxia Li、Ruoyu Guo、Xiaoting Yin、Weiwei Liu、Jun Zhou、Yifan Bai、Zilin Yu、Yehua Yang、Qingqing Dang 和 Haoshuang Wang。PP-OCR：实用的超轻量 OCR 系统。*CoRR*，abs/2009.09941，2020年。[15] Jiarui Fang、Yang Yu、Chengduo Zhao 和 Jie Zhou。TurboTransformers：面向 Transformer 模型的高效 GPU 服务系统。载于论文集。*ACM SIGPLAN PPoPP*，第 389–402 页，2021 年。[16] 谷歌。Tensorflow Serving。https://www.tensorflow.org/tfx/guide/ serving。访问日期：07-28-22。[17] Ivan Krasin、Tom Duerig、Neil Alldrin、Vittorio Ferrari、Sami Abu- El-Haija、Alina Kuznetsova、Hassan Rom、Jasper Uijlings、Stefan Popov、Shahab Kamali、Matteo Malloci、Jordi Pont-Tuset、Andreas Veit、Serge Belongie、Victor Gomes、Abhinav Gupta、Chen Sun、Gal Chechik、David Cai、Zheyun Feng、Dhyanesh Narayanan 和 Kevin Murphy。OpenImages：用于大规模多标签和多类别图像分类的公共数据集。数据集可从 *https://storage.googleapis.com/openimages/web/index.html* 获取，2017年。[18] Hang Le、Juan Miguel Pino、Changhan Wang、Jiatao Gu、Didier Schwab 和 Laurent Besacier。用于联合自动语音识别和多语言语音翻译的双解码器 Transformer。载于国际计算语言学会议（*COLING*）论文集，第 3520–3533 页，2020 年。[19] Quoc N. Le 和 Kip Kaehler。我们如何扩展 BERT，以便在 CPU 上处理 1+ 每天十亿次请求。https://blog.roblox.com/2020/05/scaled-bert-serve-1-billion-daily-requests-cpus。发布于：05-27-20，访问日期：08-02-22。[20] Yizhi Liu、Yao Wang、Ruofei Yu、Mu Li、Vin Sharma 和 Yida Wang。优化 CPU 上的卷积神经网络模型推理。载于*USENIX* 年度技术会议（*ATC*）论文集，第 1025–1040 页，2019 年。[21] Matteo Maggioni、Yibin Huang、Cheng Li、Shuai Xiao、Zhongqian Fu 和 Fenglong Song。通过循环时空融合实现高效的多阶段视频去噪。载于*IEEE* 计算机视觉与模式识别会议，*CVPR*，第 3466–3475 页，2021 年。[22] Bryan Marker、Field G. Van Zee、Kazushige Goto、Gregorio Quintana- Ortí 和 Robert A. van de Geijn。面向多线程架构的可扩展矩阵乘法。载于国际并行处理会议（*EuroPar*）论文集，第 748–757 页，2007 年。[23] Ian Masliah、Ahmad Abdelfattah、A. Haidar、S. Tomov、Marc Baboulin、J. Falcou 和 J. Dongarra。超小矩阵的高性能矩阵乘法。载于国际并行处理会议（*EuroPar*）论文集，第 659–671 页，2016 年。[24] 微软。OnnxRuntime。https://onnxruntime.ai。访问日期：08-02-22。[25] 微软。Transformer 模型优化工具概述。https://github.com/microsoft/onnxruntime/tree/master/onnxruntime/python/tools/transformers。访问日期：08-02-22。[26] Emma Ning、Nathan Yan、Jeffrey Zhu 和 Jason Li。微软开源 Transformer 在 GPU 和 CPU 上推理的突破性优化。https://cloudblogs.microsoft.com/opensource/2020/01/21/microsoft-onnx-open-source-optimizations-transformer-inference-gpu-cpu/。发布于：01-20-20，访问日期：01-06-21。[27] Victor Sanh、Lysandre Debut、Julien Chaumond 和 Thomas Wolf。Distilbert, BERT 的精简版本：更小、更快、更便宜、更轻。*CoRR*，abs/1910.01108，2019 年。[28] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Lukasz Kaiser 和 Illia Polosukhin。注意力就是你所需要的一切。载于*NeurIPS* 论文集，第 5998–6008 页，2017年。[29] Li Wang 和 Dennis Sng。深度学习算法在智慧城市视频分析中的应用：综述。*CoRR*，abs/1512 03131，2015 年。[30] Yu Emma Wang、Carole-Jean Wu、Xiaodong Wang、Kim Hazelwood 和 David Brooks。利用深度学习框架中的并行化机会。*ACM* 架构、代码优化汇刊，18(1)，2021 年。[31] Shufan Wu、Tao Lv、Pengxin Yuan、Patric Zhao、Jason Ye 和 Haibin Lin。CPU 上 BERT 推理性能优化。https://medium.com/apache-mxnet/optimization-for-bert-inference-performance-on-cpu-3bb2413d376c。发布于：09-12-19，访问日期：08-02-22。[32] Zhe Zhou、Xuechao Wei、Jiejing Zhang 和 Guangyu Sun。PetS：用于参数高效 Transformer 服务的统一框架。载于 *USENIX* 年度技术会议（*ATC*），2022 年。