TensorFlow:大规模机器学习的异构分布式系统

本文作者均来自Google Research:Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng

摘要

TensorFlow[1]是一个表达机器学习算法的接口，并且是执行算法的实现框架。使用TensorFlow表示的计算可以在众多异构的系统上方便地移植，从移动设备如手机或者平板电脑到成千的GPU计算集群上都可以执行。该系统灵活，可以被用来表示很多的算法包括，深度神经网络的训练和推算算法，也已经被用作科研和应用机器学习系统在若干的计算机科学领域或者其他领域中，例如语言识别、计算机视觉、机器人、信息检索、自然语言理解、地理信息抽取和计算药物发现。该论文描述了TensorFlow的接口和我们在Google构建的结构实现。TensorFlow API和参考实现都已经作为开源项目按照Apache2.0协议在2015年11月发布，可以在这里查看。

1 引言

Google大脑项目开始于2011年，目的是探索在科研和Google的产品中超大规模深度神经网络的使用。作为这个项目的早期工作，我们构建了DistBelif——第一代的可扩展分布式训练和推断系统[14]，这个系统工作得很不错。我们和其他Google的同事使用DistBelief进行了广泛的研究包括非监督学习[31]、语言表示[35,52]、图像分类模型和目标检测[16,48]，视频分类[27]、语音识别[56,21,20]、序列预测[47]、Go 的移动选择[34]、行人检测[2]、强化学习[38] 等等。另外，和 Google Brain 团队合作中，超过 50 个 Google 内部的团队和其他 Alphabet 公司也已经部署了使用 DistBelief 的深度神经网络在众多产品中，包括 Google Search[11]、广告产品、语音识别系统[50,6,46]、Google Photos[43]、Google Maps 和 街景[19]、Google 翻译[18]、Youtube 和很多其他的产品。

基于我们使用 DistBelief 的经验和对于期望用来训练和使用神经网络的系统特性和需求更加完备地理解，我们构建了 TensorFlow——第二代大规模机器学习模型的实现和部署的系统。TensorFlow 使用通过类似数据流模型的计算，将这些计算映射到不同的硬件平台例如使用包含一个或者多个 GPU 显卡的装有 Android 和 iOS 的单个机器上进行推断，到运行在数百台包含数千个 GPU 的大规模系统训练和推断。拥有一个单一的系统可以扩展分布到众多的平台上可以大大简化真实场景中机器学习系统的使用，正如我们在用分离的系统进行大规模训练和小规模的部署，会产生巨大的维护代价和较差的抽象效果。TensorFlow 的计算被表示为含状态的数据流图（在第二节详细讲解），我们聚焦在让这个系统足够灵活能够快速地实验研究中产生的新模型，并同时充分地提升产品级训练的性能和部署机器学习模型健壮性。为扩展神经网络训练搞更大的部署环境，TensorFlow 允许 client 简单地表达不同类型的并行通过复制和并行执行一个核心模型数据流图，依赖不同计算设备合作更新一个共享的参数或者其他的状态。 对计算描述的微妙变动可以使用较低的代价来达到和尝试很多不同的并行的方法。一些 TensorFlow 的用途借助参数更新的一致性来实现灵活性，我们可以在一些更大的部署环境中轻易表达和利用这些同步上的松弛。对比 DistBelief，TensorFlow 的编程模型更加灵活，性能也更好，支持在大规模的异构硬件平台上训练和使用很多的模型。

DistBelief 的内部用户已经切换成 TensorFlow 了。这些客户依赖 TensorFlow 来研究和产品，执行诸如在移动电话计算机视觉模型的推断到使用数百台机器进行千亿级样本的千亿级参数的深度神经网络的训练[11,47,48,18,53,41]。尽管这些应用集中在机器学习和深度神经网络上，我们希望 TensorFlow 的抽象可以用在其他的领域中，例如其他的机器学习算法或者可能其他类型的数值计算。我们按照 Apache 2.0 协议在 2015 年 11 月

开源了 TensorFlow API，可以在 www.tensorflow.org 查看。

本文下面的部分更加细致地描述了 TensorFlow。第二节介绍编程模型和 TensorFlow 接口的基本概念，第三节介绍单机和分布式的实现 。第四节给出了基本编程模型的扩展，第五节介绍了一些基本实现的优化方法。第六节给出了一些使用 TensorFlow 的实验结果，第七节描述了一些使用 TensorFlow 编程的 idiom，第九节则是一些在 TensorFlow 核心外围的工具。第十节和第十一节分别讨论了未来和相关的工作，最后一节给出了总结性想法。

2 编程模型和基本概念

TensorFlow 的计算由一个有向图描述，这个图中由一个节点集合组成。该图表达了数据流计算，作出了一些类型的节点保持和更新持久状态和让分支及循环控制结构类似于 Naiad 的行为方式的扩展。客户一般都会使用 TensorFlow 支持的前端语言（C++或者Python）构建一个计算图。在图 1 中展示了一段样例使用 Python 构建并执行了一个 TensorFlow 的计算图，结构计算图在图 2 中展示。

```
import tensorflow as tf

b = tf.Variable(tf.zeros([100]))                    # 100-d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100],-1,1)) # 784x100 matrix w/rnd vals
x = tf.placeholder(name="x")                        # Placeholder for input
relu = tf.nn.relu(tf.matmul(W, x) + b)              # Relu(Wx+b)
C = [...]                                           # Cost computed as a function
                                                    # of Relu

s = tf.Session()
for step in xrange(0, 10):
  input = ...construct 100-D input array ...        # Create 100-d vector for input
  result = s.run(C, feed_dict={x: input})           # Fetch cost, feeding x=input
  print step, result
```

Figure 1: Example TensorFlow code fragment

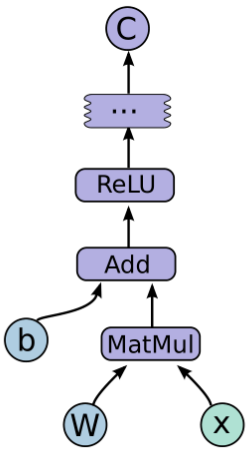

Figure 2: Corresponding computation graph for Figure 1

在一幅 TensorFlow 图中，每个节点（node）有一个或者多个输入和零个或者多个输出，表示一种操作（operation）的实例化。流过图中正常的边（输出到输入）的值都是张量（tensor），任意维度的数组其中基础元素类型是指定的或者在图的构造过程中自动推断出来的。特别的边，我们称之为控制依赖（control dependencies），同样也存在在图中：这类边上没有数据流过，但是他们表示源节点必须在目标节点的控制依赖开始执行前完成运行。因为我们的模型包括可变状态，控制依赖可以被直接用来强制在保证关系的发生。（Since our model includes mutable state, control dependencies can be used directly by clients to enforce happens before relationships.）我们的实现同样会插入控制依赖来确保独立操作之间的顺序，比如说作为控制内存使用最高峰值的方式。

操作和核(Kernel)

一个操作有一个名字。它表示一个抽象的计算（比如说，“矩阵相乘”或者“相加”）。一个操作可以有属性（attribute），所有的属性必须提供或者在图构造的过程中推断出以实例化一个节点来执行操作。属性通常的使用方式是让操作在不同的张量元素类型上多态（例如，两个 float 类型的张量和两个 int32 类型的张量）。核（kernel）是一种操作的特别实现，可以运行在一个特定类型的设备上（如 CPU 或者 GPU）。TensorFlow 的 binary 定义了可以通过注册（registration）机制实现的操作和核的集合上，这个集合可以通过连接额外的操作/核的定义/注册。表 1 展示了内置于 TensorFlow 核心库的一些操作类型。

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural-net building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

Table 1: Example TensorFlow operation types

会话(session)

客户端通过创建会话（session）和 TensorFlow 系统进行交互。为了创建一个计算图，会话接口支持外部（external）方法来提升当前由包含额外节点和边的会话的图（当会话创建时初始的图是空的）。另一个由会话接口提供的主要的操作就是 Run，以需要计算的输出名称和替换某些输出节点的张量的操作集合作为其参数输入。通过控制 Run 的参数，TensorFlow 的实现可以计算所有节点的必须执行传递闭包来计算需要的输出，然后安排执行合适节点来保证他们的依赖关系（在3.1小节详细讲解）。大多数 TensorFlow 的使用都是针对一个图启动一个会话，然后执行整个图或者通过 Run 调用来执行分离的子图数千或者数百万次。

变量(variable)

在大多数计算中，图都是执行多次的。大多数的张量在一次执行后不会存活。然而，变量（variable）是一种特别的操作可以返回一个在图执行若干次过程中存活的持久化的可变张量的句柄。这个句柄可以传递给一系列特定的操作，例如 Assign 和 AssignAdd（等同于 +=）就可以改变其引用的张量了。对应 TensorFlow 在机器学习中的应用，模型的参数典型地就存放在变量引用的张量中，并作为模型训练图的 Run 的一部分进行更新。

3 实现

TensorFlow 系统的主要部分就是客户端，它使用了会话接口来和 master 及一个或者多个的 worker processes 进行通信，每个 worker process 负责对一个或者多个计算设备（CPU 核或者 GPU card）的任意访问和在这些设备上进行图节点的计算按照 master 的要求执行。我们有本地和分布式实现的 TensorFlow 接口。本地实现通常是客户端、master 和 worker 都是在同一台机器上在一个单一的操作系统进程（可能包括多个设备，比如说装了多个 GPU card的设备）上运行。分布式实现采用了本地实现的很多的代码，但是扩展了对客户端、master 和 worker 可以在不同的机器的不同的进程上运行的场景支持。在我们的分布式环境中，这些不同的任务对应于 cluster 调度系统分配在 job 中的容器中[51]。这两种不同的模式在图 3 中进行的展示。本节剩下的部分讨论了在两种实现中遇到的问题，3.3 节讨论了针对分布式实现的一些问题。

设备

设备是 TensorFlow 的计算核心。每个 worker 负责一个或者多个设备，每个设备有一个设备类型和一个名字。设备名字由识别设备类型的部分，在 worker 中的设备索引，以及在分布式设定中，worker 的 job和任务（或者 localhost 当设备是和进程在同一机器时）的标志构成。一些例子如/job:localhost/device:cpu:0 或者 /job:worker/task:17/device:gpu:3。我们已实现了 CPU 和 GPU 的设备接口而其他的设备类型也有了通过注册机制完成的设备实现方式。每个设备对象负责管理分配和解除分配设备内存，对在 TensorFlow 实现中的更高层请求任意 kernel 的执行调度管理。

张量

实现中的张量是一种有类型的、多维度数组。我们支持若干张量元素类型，包含大小为从 8 bit 到 64 bit 的带符号和无符号整型，IEEE 浮点数和双精度类型、复数类型和字符串类型（任意长的字节数组）。合适大小的后台存储通过一个分配器进行管理，该分配器由张量所处的设备确定。张量的后端存储缓存是引用计数的并在没有引用存在时解除分配。

3.1 单设备执行

首先考虑最简单的执行场景：单一的worker进程运行在单一的设备上。图上的节点按照代表节点之间的顺序执行。特别地，我们会在每个节点上保持一个计数来记录这个节点上还没有执行的依赖。一旦这个计数变为 0，节点就可以被调度使用，并会加入到待续的队列中。待续队列按照某个非指定的顺序处理，指派节点执行的kernel 到设备对象上。当一个节点完成执行，所有依赖这个完成的节点的节点的计数都会增加。

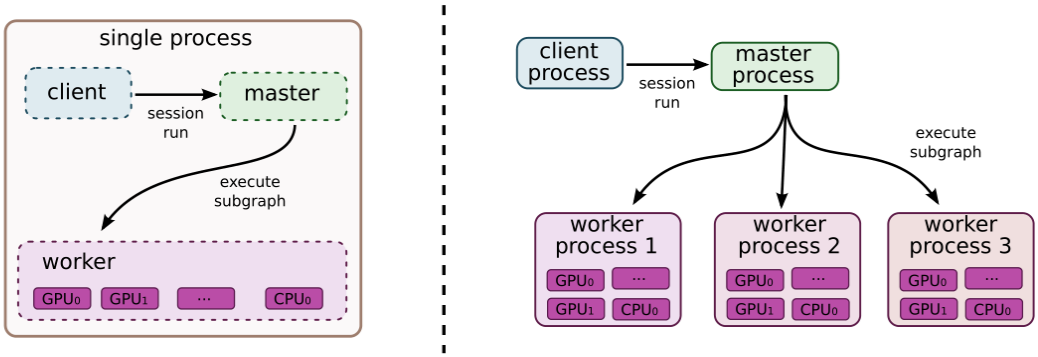

Figure 3: Single machine and distributed system structure

3.2 多设备执行

一旦系统有了多个设备，有两个主要的复杂情形出现：确定图中每个节点的计算所处的设备，然后管理由上一步确定的置放决定产生的设备间的所需的数据通信。后续部分讨论这两个问题。

3.2.1 节点的置放

给定计算图，TensorFlow 实现的主要责任之一就是将计算映射到可用的设备集合上。这个算法的简单版本下面给出。参见第 4.3 节有关该算法支持的扩展。

该置放算法的输入是一个代价模型，包括对每个图节点的输入和输出张亮的规模的估计，和对每个节点在给于其输入张量时的计算时间的。这个代价模型或者是基于关联不同操作类型的启发式规则的静态估计，或者基于实际的为更早的图的执行而做的置放决定集合衡量。

置放算法首先运行模拟的图的执行过程。模拟按照下面描述进行，对每个节点使用贪心策略选择一个设备。节点到设备的置放过程也是用作真实执行的置放。

置放算法从计算图的源点开始，在系统中的每个设备上模拟相应的活动。对每个在遍历中抵达的节点，可选 available 设备的集合会被考虑到（设备可能会由于其没能提供实现了特定操作的kernel而不可选）。对那些拥有多个可选设备的节点，置放算法使用一种贪心策略来检查在每个可能谁被上置放节点需要完成的时间的效果完成决策。这种启发式规则考虑了根据代价模型在那种设备上估计的和衡量的执行时间，还有任何用来从其他设备传输输入到该节点的通信的代价。其中节点的操作完成最快的设备会被选作该操作的设备，置放决策然后会继续针对图中其他的节点进行处理，包含那些已经做好模拟执行的下游节点。第 4.3 节描述了一些扩展，让用户可以提供提示和部分限制来指导置放算法。这个算法现在还在持续开发的过程中。

3.2.2 交叉设备通信

一旦置放节点被计算，图会被分割成一系列的子图，每个子图在分布在1台设备上。任何从x到y的交叉设备的边会被移除，然后被一个从x到新的发送节点的边代替。参见图 4 中所进行的变换。

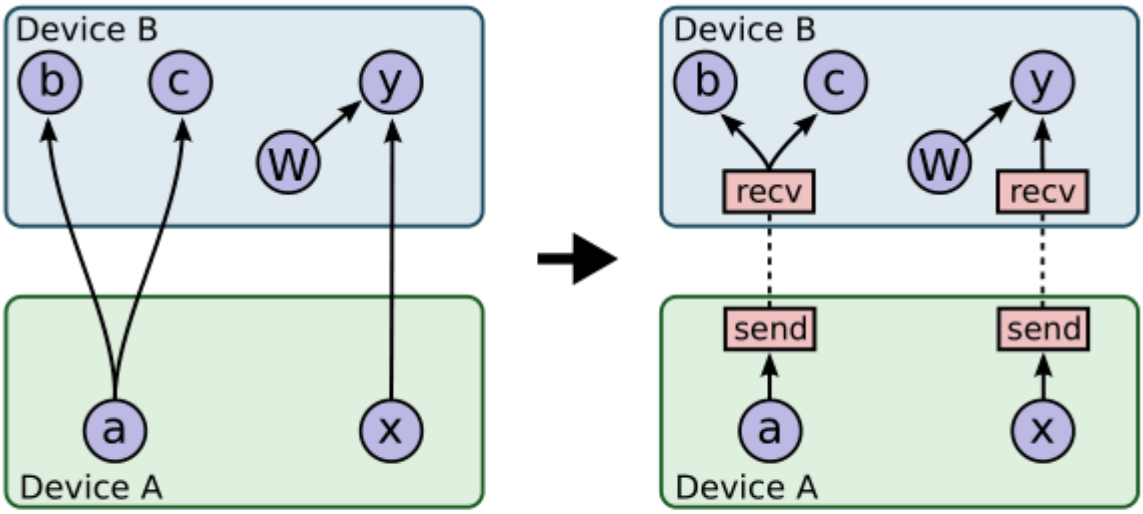

Figure 4: Before & after insertion of Send/Receive nodes

在运行时刻，Send 和 Receive 节点合作进行跨设备的数据交换。这使得我们可以隔离所有在 Send 和 Receive 内部实现的通信，这样简化了运行时刻剩下的部分工作。

当我们插入 Send 和 Receive 节点时，我们将在特定设备上的特定张量的所有使用者进行合并规整来使用单个 Receive 节点，而不是对特定设备上的每个下游使用者都给一个 Receive 节点。这确保了需要使用的张量数据仅仅会从源设备到目的设备传输一次，而在目的设备上的张量内存也只会分配一次（而非多次，参看图 4 的节点 b 和 c）。

通过这种方式处理通信，我们也允许了不同设备上的图中的个别节点调度可以被去中心化到 workers 上：Send 和 Receive 节点传达了在不同的 worker 和 设备间必要的同步信息，master 仅仅需要对每个图的执行给出一个 Run 请求给那些包含图中任意节点的 worker，而不是会对所有节点或者每个跨设备通信都进行调度。这也让系统更加可扩展，并允许比通过 master 来强制进行所有的调度更加精确的节点执行。

3.3 分布式执行

计算图的分布式执行非常类似于多设备执行。在设备置放后，子图会针对每个设备创建。用于 worker 进程之间的通信的 Send/Receive 节点对使用了诸如 TCP 或者 RDMA 这样的远程通信机制进行跨机器的数据迁移。

容错

分布式执行中的错误可以在很多地方进行检测。最主要的有 (a) 在 Send 和 Receive 节点对之间的通信错误，(b) 从 master 进程到每个 worker 进程的周期性的健康状态检测。

如果发现了错误，整个图的执行就会终止，并从头开始。但是回想之前变量节点对应于那些在执行过程中记忆持有（persist）的张量。我们支持在重启过程中的一致的检查点和状态恢复。特别是，每个变量节点连接在一个 Save 节点上。这些 Save 节点周期性地执行，比如说每 N 次迭代，或者每隔 N 秒。他们执行的时候，变量的内容被写到持久化的存储中，比如说，一个分布式的文件系统。类似地，每个变量连接在一个 Restore 节点上，只会在一次重启后的第一个迭代中启用。在 4.2 节有某些节点仅能够在某些图的执行中启用的细节。

4 扩展

本节我们描述在第 2 节给出的编程模型中几个更加高级的特性。

4.1 梯度计算

很多优化算法，包括常用的机器学习训练算法，如随机梯度下降 [45]，计算代价函数关于一个输入集合的梯度。由于该算法广泛的应用需求，TensorFlow 已经内置了对自动梯度计算的支持。如果张量 C 通过一个复杂的子图操作依赖于张量 {X_k} 集合，那么就有一个内置的函数可以返回张量 {dC/dX_k}。梯度张量如同其他张量一样通过扩展 TensorFlow 图使用下面的流程进行计算的。

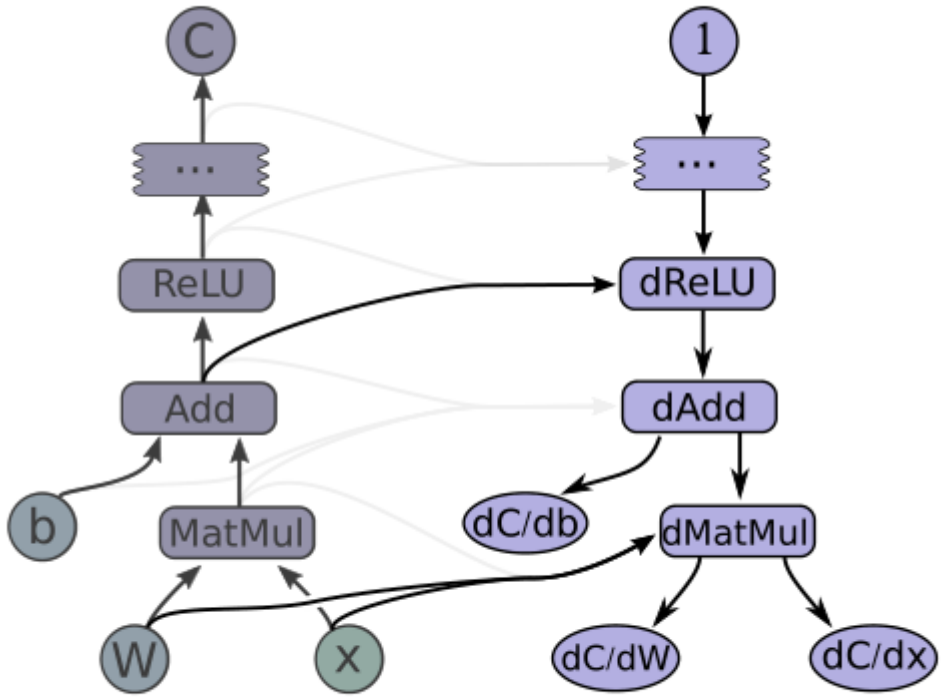

Figure 5: Gradients computed for graph in Figure 2

当 TensorFlow 需要计算一个张量 C 关于某个张量 I 时，首先找出计算图中从 I 到 C 的路径。然后从 C 回溯到 I，而对在回溯路径中的每个操作都会添加一个节点到 TensorFlow 的图上，包含回溯路径上使用链式法则的偏导数。新加的节点计算前向路径中相对应的操作的梯度函数。梯度函数可以是任何的操作。这个函数不但可以用在反向路径中计算出的偏导数作为输入，同样也能选择性地用前向操作的输入输出作为输入。图 5 展示了图 2 中例子的代价函数的梯度。灰色箭头代表梯度函数没有在特定操作中显示的潜在输入。图 1 所对应的是计算这些梯度：[db,dW,dx] = tf.gradients(C, [b,W,x])

通常操作可能会有多个输出，C 可能仅仅会依赖于其中一部分。例如，如果操作 O 有两个输出 y_1 和 y_2，C 仅仅依赖于 y_2，那么 O 的梯度函数的第一个输入就可以设置为 0 因为 dC/dy_1 = 0。

自动梯度计算让优化尤其是内存耗用变得复杂。在执行前向计算子图时，那些显式地由用户创建的操作，可能的启发式想法就是通过观察图被构建的次序来确定哪个节点执行下一步。

这通常指的是临时输出会在创建后不久就是用到，所以他们的内存可以很快重新用到。当这个启发式想法不适用时，用户可能会改变图构建的次序，或者增加在第 5 节将会介绍的控制依赖。当梯度节点自动加入到图中时，用户控制就会变弱，启发式想法就失效了。特别地，因为梯度逆转了前向计算的顺序，在图早期用到的张量会在梯度计算的结尾时重新被频繁用到。这样的张量会消耗很多的 GPU 内存，也就不必要地限制了计算的规模。我们正积极地提升内存关联的效果以更好地处理这个问题。是用更加精密的启发式想法来确定图执行的次序，重计算张量而不是存储在内存，将长效的张量从 GPU 内存中移到 CPU 内存中等等都是可以尝试的思路。

4.2 部分执行

常常有 client 希望执行整个图的子图。为了支持这一点，只要 client 在 Session 中构建出一个计算图，我们 Run 的方法运行他们执行整个计算图的任意的子图，并将任意的数据注入到图中的任何边上，以及获取流经任何边上的数据。

图中每个节点都有一个名字，一个节点的每个输出都通过源节点名和节点输出端口确定，从 0 开始计数（例如，“bar:0” 表示 “bar” 节点的第一个输出，而 “bar:1” 则是第二个输出）。Run 调用的两个参数可以帮助定义准确的将要执行的子图。第一，Run 调用接受输入，一个可选的映射 name:port 名来填充张量值。第二，Run 调用接受 output_names，输出 name[:port] 列表指定了需要执行的节点，并且，如果端口出现在名字中，那么对那个节点的特定输出张量值就应该返回给 client 如果 Run 调用成功完成。

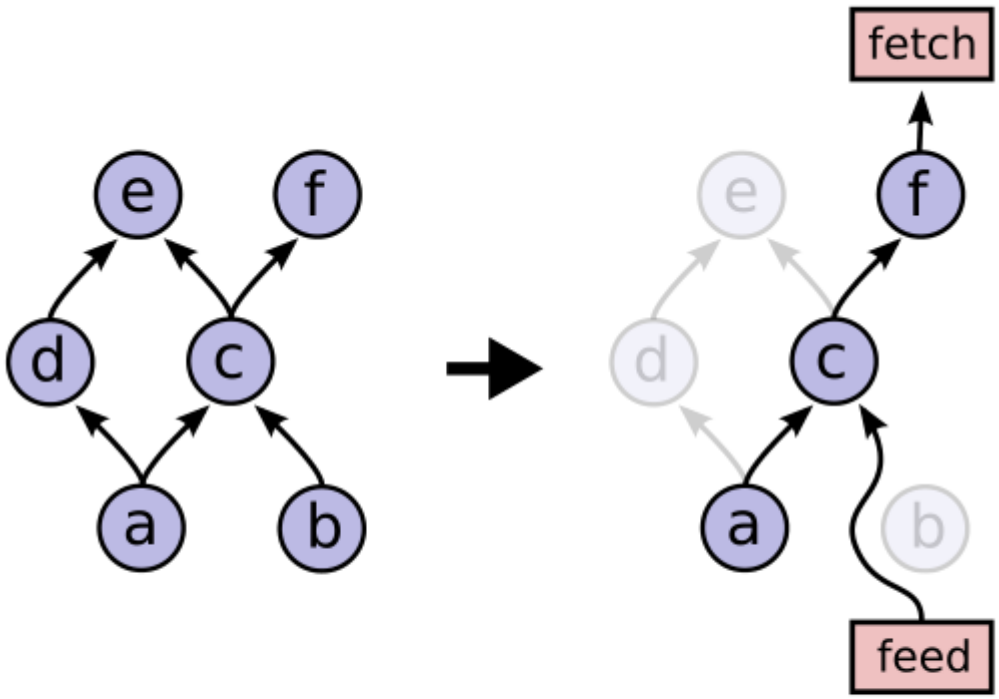

Figure 6: Before and after graph transformation for partial execution

计算图是基于输入输出的值进行变换的。每个在输入中指定的 node:port 使用 feed 节点替换，这个会选出从特定初始化的用来给 Run 调用的 Rensezvous 对象的入口选择出给定的输入向量。类似地，每个输出名字关联在一个特定的 fetch 节点上，可供输出张量的存储，并在 Run 调用完成时返回给客户端。最后，一旦图已经被这些特定的 fee 和 fetch 节点重写，将要执行的节点集合可以被从每个被输出命名的节点出发，然后使用图的依赖关系来确定必须在重写图中执行产生输出的整个节点的集合中进行反向传播。图 6 展示了左边的原始图，以及变换的图，其中 Run 被激活输入是 {b} 输出是 {f:0}。因为我们仅仅需要计算节点 f 的输出，我们不需要执行节点 d 和 e，因为他们没有对 f 做出贡献。

4.3 设备限制

TensorFlow 客户端可以通过为一个节点哪些设备可以在其上执行来提供部分的限制来控制节点在设备中的放置。例如，“仅仅放置这个节点在类型为 GPU 的设备上” 或者 “这个节点可以被放置在任何在 /job:worker/task:17 中的设备上”，或者 “共享这个节点和 variable13 节点”。按照这些限制，放置算法就可以负责选择节点的分配来提供快速执行，并且满足不同的设备本身的限制，例如内存的总量的限制来执行子图的计算。

支持这样的限制需要对 3.2.1 节介绍的放置算法进行调整。我们首先计算每个节点的可行设备集合，然后使用 union-find 算法来计算图中必须放在一起的图的组成部分。对每个这样的组成部分，我们计算可行设备集合的交集。计算出的每个节点可行设备和放置算法的模拟器很容易匹配。

4.4 控制流

尽管没有任何显式的控制流数据流图非常强大，但是我们发现了一些例子中支持条件和循环可以得到更加简洁而高效的机器学习算法的表示。

在 Arvind [3] 中描述的数据流机器观点，我们引入了一个小控制流操作的集合进入 TensorFlow 并且推广 TensorFlow 使之能够处理循环的数据流图。Switch 和 Merge 操作符可以让我们基于布尔值的张量来跳过整个子图的执行。Enter Leave NextIteration 操作符可以进行迭代。高级程序构造如 if-conditional 和 while-loop 可以很容易用这些控制流操作编译成数据流图。

TensorFlow 运行时刻实现了一个 tags 和 frames 概念，这与 MIT Tagged Token Machine 类似。每次迭代都使用了唯一一个 tag，这个执行的状态通过 frame 表示的。只要被使用，输入就可以进入一次迭代；因此，多次迭代可以被并发地执行。

TensorFlow 使用分布式协同机制来使用控制流进行图的执行。一般来说，循环可以包含被分配到多个不同的设备上的节点。因此，管理循环的状态成为了一个分布式终止检测的问题。TensorFlow 的解决方案基于图的重写。在图的划分（partitioning）过程中，我们自动地增加控制节点到每个划分（partition）上。这些节点实现了一个小的状态机，来管理每次迭代的开始和终止，确定整个循环的终止。

如上所示，我们通常通过随机梯度下降来训练机器学习模型，将梯度计算表示成数据流图的一部分。在模型包含控制流操作时，我们必须在对应的梯度计算中处理这些操作。例如，使用 if-conditional 来对模型梯度计算时，需要知道哪个分值会被选择，然后应用梯度逻辑到这个分支上。类似地，使用 while-loop 来对模型梯度计算时，需要知道多少次迭代，同样还依赖在这些迭代中出现的中间值。基本技术就是重写这些图来记住需要计算梯度计算的值。我们这里省略掉细节介绍。

4.5 输入操作

尽管输入数据可以被通过 feed 节点提供给计算，另一个用来训练大规模机器学习模型通常的机制是在图中采用特定的输入操作节点，这些节点一般来说会通过文件名配置，并产生一个包含一个或者多个样本的张量来自在每次执行时的那些文件集合中存放的数据。这使得数据可以被直接从底层存储系统读出到将要执行接下来处理的内存中。在配置中，从 worker 进程分隔开的客户端进程，如果数据给定，一般会需要一个额外的网络 hop（从存储系统到客户端然后从客户端到 worker vs. 使用一个输入节点时直接从存储系统到 worker）。

4.6 队列

队列是一个加入到 TensorFlow 中的有用的特性。他们允许图的不同部分来异步地执行，可能按照不同的节奏，来通过 Enqueue 和 Dequeue 操作来处理数据。在队列中空间尚存时，可以进行 Enqueue 操作；在指定的最小数量的元素可以取出时，可以进行 Dequeue 操作。队列的一个用途就是允许输入数据可以从磁盘文件中预先取出，这样可以同时进行前一批的数据的处理。同样还能用来进行其他类型的归类，包括聚合很多梯度来计算某种更加复杂的梯度组合，或者来组织不同的输入语句到递归语言模型到语句的近似同样长度的 bins 中，这样处理得更有效率。

在一般的 FIFO 队列上，我们还实现了一个 shuffling 队列，可以对内存内的缓冲区内的元素进行随机洗牌。洗牌功能在机器学习算法中比较有用，常常需要对样本进行随机化。

4.7 容器

容器 是用来管理长期存在的可变状态的机制。变量 Variable 反向存放在一个容器内。默认的容器是直到进程终止时都是存活的，但是我们也允许其他的容器。容器可以通过清除它的整个内容进行重置。使用容器，就可以分享状态在完全不相交的不同会话中的计算图中。

5 优化

本节，我们给出一些在 TensorFlow 实现中的优化，这些优化对系统性能和资源利用率起到了提升作用。

5.1 共同子表达式消除

因为计算图的构造通常是由很多不同层的抽象完成的，计算图一般都会包含冗余的计算过程的多个副本。为了解决这个问题，我们已经实现了一个共同子表达式过程，类似于 Click[12] 中给出的算法，运行在计算图上，并将操作的多重复制，用这些节点中的一个单独节点代替，并将图的边重定向以满足这个归一化。

5.2 控制数据通信和内存分配

小心规划的Tensorflow操作可以取得更好的效能。特别是在数据传输和内存使用上面。特别是在直接使用需要存储在内存的结果计划可以减少时间窗口，在操作之间和内存尖峰消耗的问题。这种减少对GPU设备特别重要。将来，计划的通讯也会减少设备间的网络资源的消耗。

在那么多可以优化和计划的地方，我们着重那些特别必要和效率的地方。它涉及调度接收节点读取远程值。如果没有采取任何预防措施，这些节点可能比必要时早得多启动，甚至可能在执行开始时就立即全部启动。通过执行计算常见的即时/尽可能快（ASAP / ALAP）的方式，我们分析图的关键路径，以估计何时启动接收节点。然后我们插入控制边缘，目的是延迟这些节点的启动，直到需要这些结果才启动。

5.3 异步 kernel

除了在Compute方法结束时完成其执行的普通同步内核之外，我们的框架还支持非阻塞内核。这样的非阻塞内核使用一个稍微不同的接口，从而Compute方法被传递一个继续命令，当内核的执行完成时应该调用它。 这是针对内存使用情况或其他资源方面有很多活动线程相对昂贵的环境的优化，并允许我们避免在无限期的时间内捆绑执行线程，同时等待I / O或其他事件发生。异步内核的例子包括Receive内核，Enqueue和Dequeue内核（如果队列空间不可用或者没有数据可分别读取，则可能需要阻塞）。

5.4 用于 kernel 实现的优化库

我们经常利用预先存在的高度优化的数学库来实现某些操作的内核。例如，有许多优化库用于在不同设备上执行矩阵乘法，包括BLAS [15]和cuBLAS [39]，或用于深度神经网络的卷积内核的GPU库，例如cuda-convnet [28]和cuDNN [9]。我们的许多内核实现都是围绕这种优化库的相对较薄的包装器。我们相当广泛地使用了开源的Eigen线性代数库[25]，用于系统中的许多内核实现。 作为TensorFlow开发的一部分，我们的团队（主要是Benoit Steiner）已经扩展了开源Eigen库，并支持任意维度张量操作。

5.5 有损压缩

一些机器学习算法，包括通常用于训练神经网络的机器学习算法，容忍噪声和降低的精度算术。以类似于DistBelief系统[14]的方式，当在设备之间发送数据时（有时在同一机器内但特别是在机器边界上），我们经常使用高精度内部表示的有损压缩。例如，我们经常插入特殊的转换节点，将32位浮点表示转换为16位浮点表示（不是提议的IEEE 16位浮点标准，而只是32位IEEE 794浮点格式，但在尾数处精度低16位），然后在通信信道的另一侧转换回32位表示（通过为尾数的丢失部分填充0，因为这比计算成本低得多在进行32→16→32位转换时，数学上正确的概率舍入）。

6 状态和经验

TensorFlow界面和一个参考实现已经在Apache 2.0许可下开源，系统可以从www.tensorflow.org下载。该系统包括详细的文档，许多教程和大量示例，演示如何使用该系统执行各种不同的机器学习任务。这些例子包括对来自MNIST数据集（“机器学习算法的hello world”）的手写数字进行分类的模型[32]，对来自CIFAR10数据集[30]的图像进行分类，使用循环的LSTM [22]网络进行语言建模，训练词嵌入向量[35]等等。该系统包括用于在Python和C ++中指定TensorFlow计算的前端，并且我们期望随着时间的推移会添加其他前端，以响应内部Google用户和更广泛的开源社区的需求。我们之前的DistBelief系统中有很多机器学习模型[14]，我们已经迁移到TensorFlow。本节的其余部分将讨论我们学到的一些经验教训，这些经验教训对于从一个系统到另一个系统的机器学习模型的任何这种迁移都是可以普遍适用的，因此可能对其他人有用。特别是，我们将重点放在我们的经验教训上，即将先进的卷积神经网络用于称为Inception的图像识别[23]。该图像识别系统将224×224像素图像分类为1000个标签之一（例如，“猎豹”，“垃圾车”等）。以TensorFlow图形表示时，这种模型包含1360万可学习参数和36000次操作。在单个图像上运行推理需要20亿次乘加运算。在TensorFlow中建立所有必要的数学运算之后，将所有36,000个操作组装并调试成正确的图形结构证明具有挑战性。验证正确性是一项困难的事情，因为系统本身就是随机的，并且只打算以期望的某种方式行事 - 可能需要几个小时的计算。鉴于这些情况，我们发现以下策略对于将Inception模型移植到TensorFlow中至关重要：

1. 构建工具以深入了解给定模型中参数的确切数量。这些工具在复杂的网络体系结构规范中表现出了微妙的缺陷。特别是，我们能够识别由于在维度上的数学运算中自动广播而导致实例化的操作和变量。
2. 从小处开始并扩大规模。 我们从以前的系统中移植出来的第一个卷积神经网络是一个在CIFAR-10数据集上使用的小型网络[30]。 调试这样的网络阐明了机器学习系统内单个操作（例如，最大化池）中的细微边缘情况，其在更复杂的模型中实际上难以解读。
3. 学习关闭时，务必确保机器学习系统之间的目标（损失功能）匹配。 将学习速率设置为零有助于我们识别模型中随机初始化变量的意外行为。 在一个动态的培训网络中，这样的错误很难找到。
4. 在调试分布式实现之前，使单个机器实现匹配目标。 该策略帮助我们描绘和调试机器学习系统之间的训练效果差异。 特别是，我们确定了导致的错误的原因是竞态条件和错误地假定为原子操作的非原子操作。
5. 防范数字错误。 数值库在处理非定值浮点值方面不一致。 卷积神经网络对数值不稳定性特别敏感，并且在实验和调试阶段往往会相当规律地发散。 通过检查非确定浮点值来防止这种行为，可以实时检测错误，而不是识别事后发散行为。
6. 分析一个网络的各个部分，了解数值误差的大小。 在两个机器学习系统上并行运行神经网络的子节点，提供了一种确保两个系统中的数值算法相同的精确方法。 鉴于这些算法是以浮点精度运行的，预测和理解预期数值误差的大小以判断给定组件是否正确实现（例如，区分“1e-2，好！”和“在1e-2之内：为什么它是不正确的？！”）。

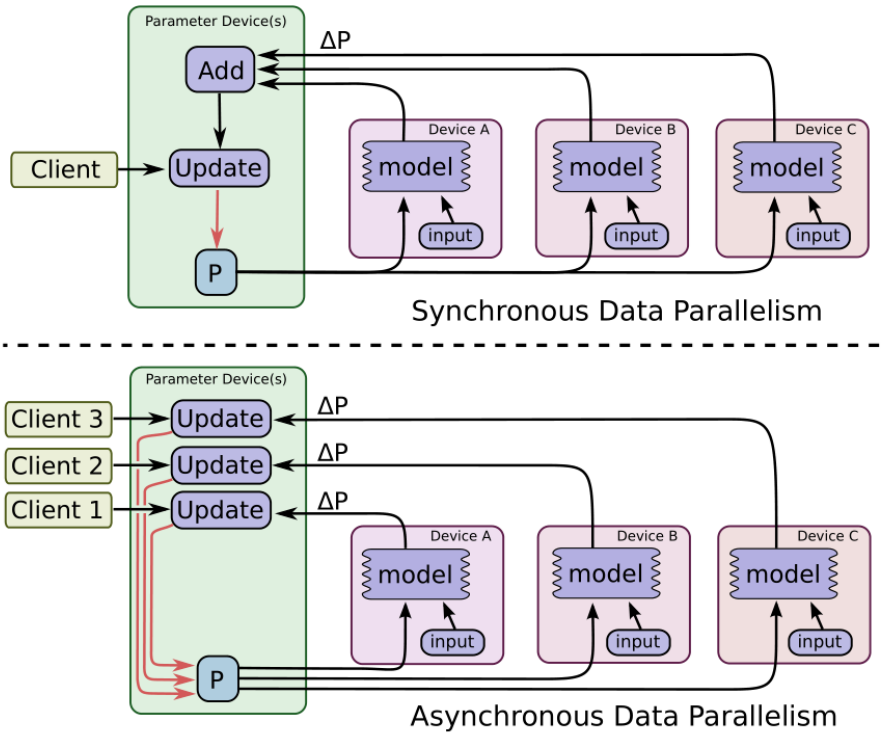

Figure 7: Synchronous and asynchronous data parallel training

在存在固有随机系统的情况下验证复杂的数学运算相当具有挑战性。 上面列出的策略证明在获得对系统的信心并最终实例化TensorFlow中的初始模型方面是非常宝贵的。 这些努力的最终结果是训练时间比我们现有的DistBelief模型实现速度提高了6倍，并且这种速度增益在训练一类新的大规模图像识别模型中不可或缺。

7 常用编程规范

TensorFlow的基本数据流图模型可以以多种方式用于机器学习应用程序。我们关心的一个领域是加速大型数据集上计算密集型神经网络模型的训练。本部分介绍了我们和其他人为实现此目的而开发的几种技术，并说明了如何使用TensorFlow来实现这些不同的方法。本小节中的方法假定模型正在使用随机梯度下降（SGD）进行训练，其中相对较小的小批量为100到1000个例子。

数据并行训练加速SGD的一个简单技术是并行化小批量元素中小批量的梯度计算。例如，如果我们使用1000个元素的小批量大小，我们可以使用模型的10个副本为每个元素计算100个元素的梯度，然后组合梯度并同步更新参数，以便表现就好像我们正在运行批量为1000个元素的顺序SGD算法。在这种情况下，TensorFlow图只需要复制大量模型计算的图形部分的副本，并且单个客户机线程驱动整个训练循环以获取此大图。这在图7的顶部部分中进行了说明。此方法也可以是异步的，其中TensorFlow图有许多模型计算大部分图的部分副本，并且这些副本中的每一个还应用参数更新为模型参数异步。在此配置中，每个图副本都有一个客户端线程。这在图7的底部进行了说明。这种异步方法也在[14]中进行了描述。

模型并行训练模型并行训练，其中模型计算的不同部分在同一批示例中同时在不同的计算设备上完成，在TensorFlow中也很容易表达。图8显示了用于序列到序列学习的循环，深度LSTM模型（参见[47]）的例子，并行化了三种不同的器件。模型计算流水线的并行步骤另一种更好地利用训练深度神经网络的常用方法是通过在同一组设备中运行少量并行步骤，在同一设备中管理模型的计算。这如图9所示。它与异步数据并行性有些类似，除了并行性发生在同一设备内，而不是在不同设备上复制计算图。这允许“填补空白”，在单个步骤中，单批示例的计算可能无法在所有时间内充分利用所有设备上的完全并行性。

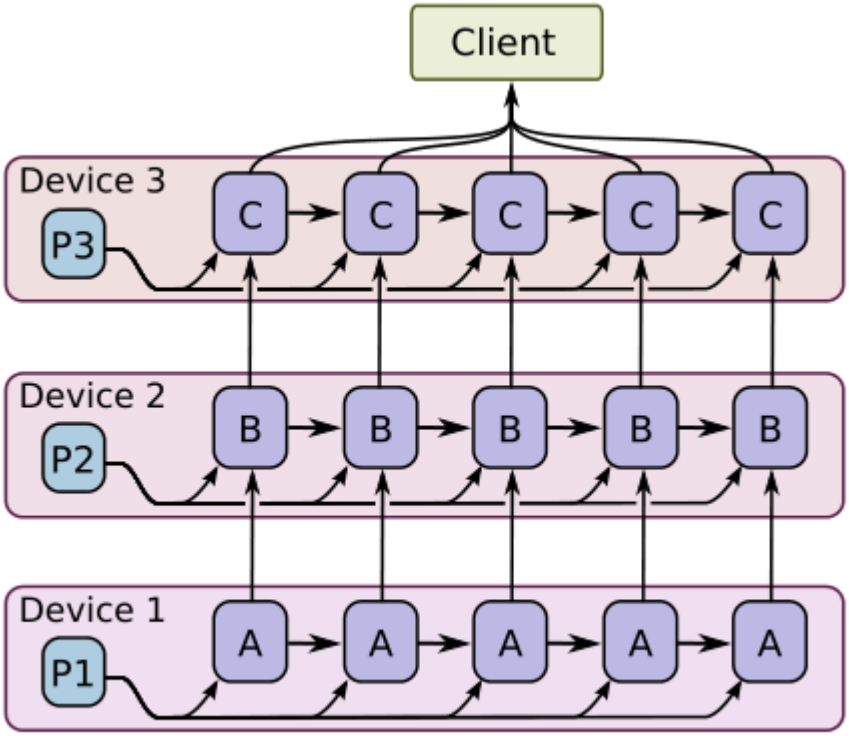

Figure 8: Model parallel training

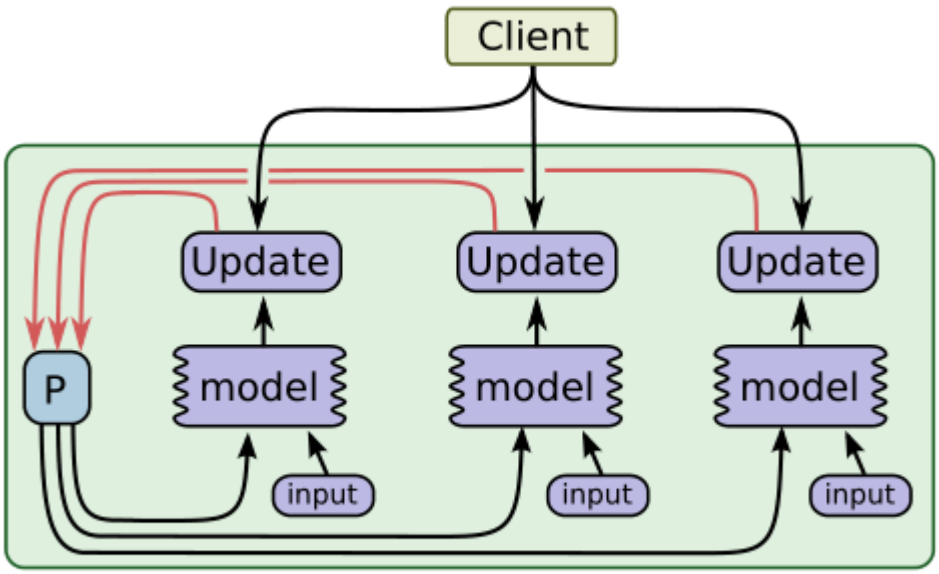

Figure 9: Concurrent steps

8 性能

本白皮书的未来版本将具有单机和分布式实施的综合性能评估部分。

9 工具

本节介绍一些我们开发的与TensorFlow核心图执行引擎并行的工具。

9.1 TensorBoard: 图结构和总结统计可视化

为了帮助用户理解其计算图的结构以及了解机器学习模型的整体行为，我们已经构建了TensorBoard，这是TensorFlow的配套可视化工具，包含在开源版本中。计算图的可视化深度神经网络的许多计算图可能非常复杂。例如，用于训练与Google的Inception模型类似的模型的计算图[48]，在ImageNet 2014比赛中具有最佳分类性能的深度卷积神经网络，其TensorFlow计算图中有超过36,000个节点，并且一些深度循环用于语言建模的LSTM模型拥有超过15,000个节点。由于这些图的大小和拓扑结构，幼稚的可视化技术通常会产生混乱的和压倒性的图。为帮助用户查看图形的基本组织，TensorBoard中的算法将节点折叠成高级块，突出显示具有相同结构的组。该系统还将经常提供簿记功能的高度节点分隔到屏幕的单独区域中。这样做可以减少视觉混乱，并将注意力集中在计算图的核心部分。整个可视化都是交互式的：用户可以平移，缩放和展开分组节点来深入了解细节。图10显示了深卷积图像模型图的可视化示例。

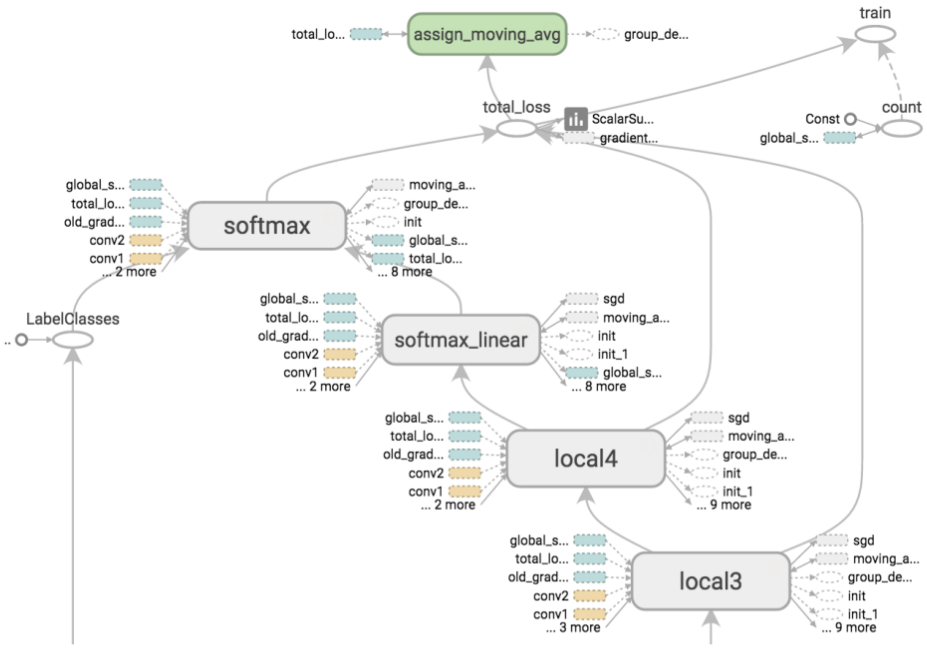

Figure 10: TensorBoard graph visualization of a convolutional neural network model

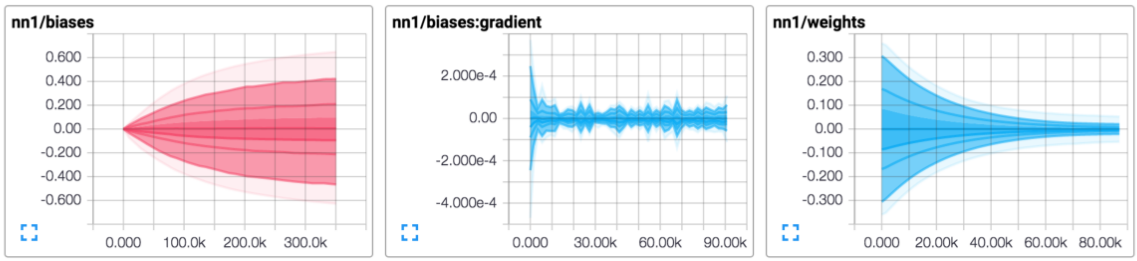

Figure 11: TensorBoard graphical display of model summary statistics time series data

汇总数据的可视化在训练机器学习模型时，用户通常希望能够检查模型各个方面的状态，以及这种状态随时间变化的情况。为此，TensorFlow支持可以插入到图形中的不同汇总操作的集合，包括标量汇总（例如，用于检查模型的整体属性，例如跨一系列示例的平均损失函数的值，或者执行计算图所花费的时间），基于组织格式的摘要（例如，神经网络层中的权值分布）或基于图像的摘要（例如卷积神经网络中学习的滤波器权重的可视化）。通常

设置计算图以便汇总节点被包括来监视各种感兴趣的值，并且在训练图的执行期间每隔一段时间，除了被执行的正常节点组之外，还执行汇总节点集合，并且客户机驱动程序将汇总数据写入与模型训练关联的日志文件。然后，TensorBoard程序被配置为观察该日志文件以获得新的汇总记录，并且可以显示该汇总信息及其随时间变化的方式（能够选择“时间”的测量值是自开始以来的相对保留时间执行TensorFlow程序，绝对时间或“步骤”，这是从TensorFlow程序执行开始以来发生的图形执行次数的数字度量）。图11显示了TensorBoard中汇总值的可视化屏幕截图。

9.2 性能追踪

我们还有一个名为EEG的内部工具（2015年11月最初的开放源代码版本中没有包含），用于收集和可视化关于TensorFlow图表执行的精确排序和性能特征的非常精细的信息。该工具适用于单机和分布式实现，对于理解TensorFlow程序的计算和通信模式中的瓶颈非常有用。系统中的每台计算机上都会同时从各种来源收集跟踪信息，包括Linux内核跟踪，我们自己的轻量级线程跟踪工具以及CUDA分析工具界面（CUPTI）。通过这些日志，我们可以重建每个线程切换的微秒级细节，CUDA内核启动和DMA操作的分布式训练步骤的执行。在可视化服务器中组合痕迹，该服务器旨在快速提取指定时间范围内的事件，并以适当的详细级别汇总用户界面的分辨率。由于通信，同步或与DMA有关的延迟而导致的任何重大延迟都会使用可视化中的箭头进行标识和突出显示。最初，UI提供了整个跟踪的概览，只显示了最重要的性能工件。随着用户渐进式放大，渲染的分辨率越来越高。图12显示了在多核CPU平台上训练的模型的示例EEG可视化。截图的前三分之一显示根据数据流限制，TensorFlow操作并行分派。跟踪的底部显示了大多数操作如何被分解为在线程池中并发执行的多个工作项目。右侧大小上的对角线箭头显示线程池中排队延迟的位置。图13显示了另一种主要在GPU上进行计算的EEG可视化。可以看到主机线程在TensorFlow GPU操作变为可运行时（淡蓝色线程池）排队，而后台看家线程可以在跨处理器内核迁移的其他颜色中看到。再一次，箭头显示线程在GPU到CPU的传输中停顿的位置，或者操作系统遇到严重的排队延迟。最后，图14显示了更详细的视图，它允许我们检查Tensorflow GPU运算符如何分配给多个GPU流。无论何时数据流图允许并行执行或数据传输，我们都努力使用流和流依赖原语向GPU设备公开排序约束。

10 未来工作

我们对未来的工作有几个不同的方向。我们将继续使用TensorFlow开发新的有趣的人工智能机器学习模型，并且在这个过程中，我们可能会发现我们需要扩展基本TensorFlow系统的方式。开源社区也可能会为TensorFlow实现提出新的有趣方向。我们正在考虑的基本编程模型的一个扩展是函数机制，用户可以将TensorFlow计算的整个子图指定为可重用组件。在我们设计的实现中，这些函数可以成为TensorFlow的不同前端语言的可重用组件，这样用户可以使用Python前端定义一个函数，然后将该函数用作基本构建块C ++前端。我们希望这种跨语言的可重用性能够引导一个充满活力的机器学习研究人员社区，他们不仅发布了他们研究的全部范例，而且还发布了可以在其他环境中重用的小型可重用组件。我们也有许多具体的方向来提高TensorFlow的性能。一个这样的方向是我们在即时编译器上的初始工作，该编译器可以获取TensorFlow执行的子图，可能带有一些关于张量的典型大小和形状的运行时分析信息，并且可以为该子图生成优化的例程。该编译器将理解执行许多优化的语义，例如循环融合，局部区域的阻塞和平铺，特定形状和大小的专门化等。我们还想象未来工作的一个重要领域将是改进布局和节点调度算法，用于决定不同节点的执行位置以及何时开始执行。我们目前在这些子系统中实施了一些启发式算法，我们希望让系统学会做出好的放置决策（可能使用深度神经网络，并结合强化学习目标函数）。

11 相关工作

还有很多其他系统可以通过各种方式与TensorFlow进行比较。 Theano [7]，Torch [13]，Caffe [26]，Chainer [49]和计算网络工具包[54]是一些主要用于训练神经网络的系统。与分布式TensorFlow实现不同，这些系统中的每一个都将计算映射到单个机器上。像Theano和Chainer一样，TensorFlow支持符号差异化，从而更容易定义和使用基于渐变的优化算法。像Caffe一样，TensorFlow有一个使用C ++编写的核心，简化了部署14图12：多线程CPU操作的EEG可视化（x轴是以μs为单位的时间）。图13：显示CPU和GPU活动的Inception培训的EEG可视化。

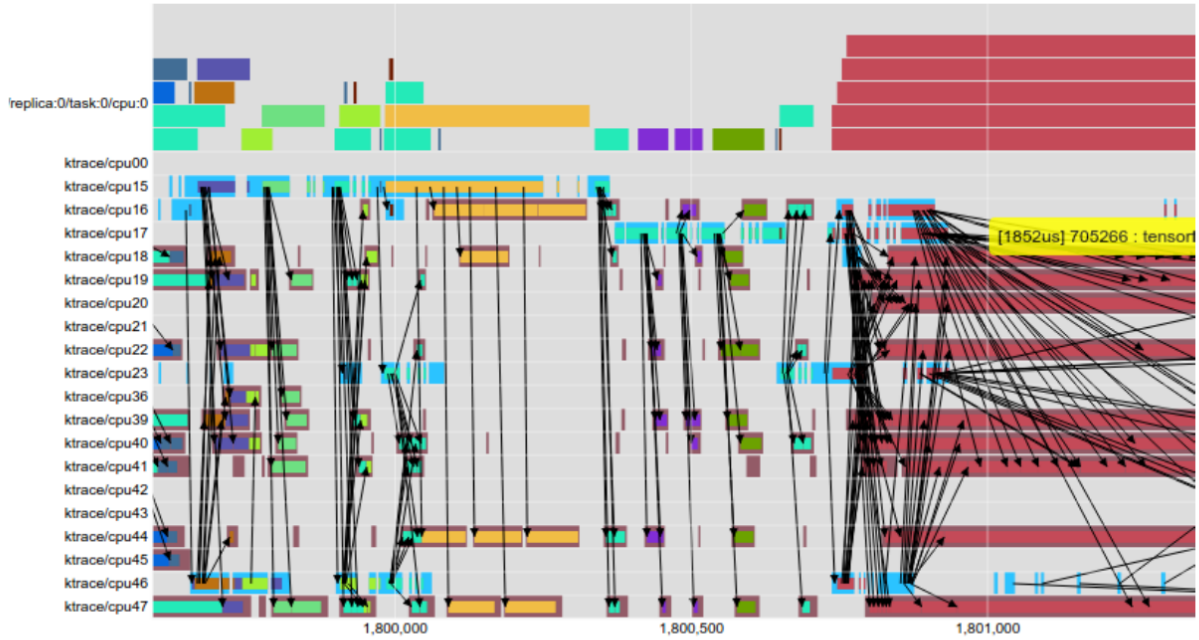

Figure 12: EEG visualization of multi-threaded CPU operations (x-axis is time in μs).

在各种各样的生产环境中训练有素的模型，包括存储和计算受限的环境，例如移动设备。 TensorFlow系统与其前身系统DistBelief [14]共享一些设计特征，以及具有Project Adam [10]和Parameter Server [33]等类似设计的后期系统。像DistBelief和Project Adam一样，TensorFlow允许计算跨越许多机器上的许多计算设备，并允许用户使用相对较高级的描述来指定机器学习模型。与DistBelief和Project Adam不同，TensorFlow中的通用数据流图模型更灵活，更适合表达更多种类的机器学习模型和优化算法。它还允许将有状态参数节点的表达式作为变量进行显着简化，并且可变更新操作只是图中的附加节点;相比之下，DistBelief，Project Adam和参数服务器系统都有15个图14：多流GPU执行时间线。

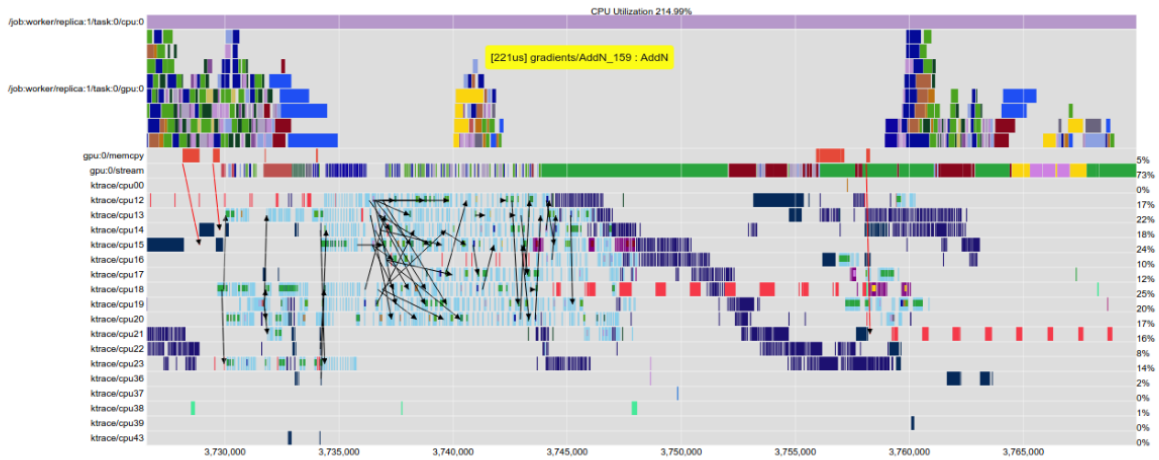

Figure 13: EEG visualization of Inception training showing CPU and GPU activity.

整个独立的参数服务器子系统致力于通信和更新参数值。用于表达图像处理流水线的Halide系统[40]对TensorFlow数据流图使用类似的中间表示。不过，与TensorFlow不同的是，Halide系统实际上对其操作的语义具有更高层次的知识，并利用这些知识生成高度优化的代码片段，结合了多个操作，并考虑到并行性和局部性。Halide只在一台机器上运行计算结果，而不是在分布式设置中运行。

在未来的工作中，我们希望用类似的跨操作动态编译框架来扩展TensorFlow。像TensorFlow一样，已经开发了几个其他分布式系统来执行跨集群的数据流图。 Dryad [24]和Flume [8]演示了如何将复杂的工作流程表示为数据流图。 CIEL [37]和耐德[36]引入了数据相关的控制流程的通用支持：CIEL表示迭代作为动态展开一个DAG，而耐德使用具有周期的静态图形，以支持低等待时间的迭代。 Spark [55]针对使用"弹性分布式数据集"（RDD）重复访问相同数据的计算进行了优化，RDD是早期计算的软状态高速缓存输出。蒲公英[44]在包括GPU的异构设备集群上执行数据流图。 TensorFlow使用混合数据流模型，借用每个系统的元素。它的数据流调度器（它是选择下一个要执行的节点的组件）使用与Dryad，Flume，CIEL和Spark相同的基本算法。其分布式架构最接近Naiad，因为系统使用单个优化的数据流图来表示整个计算，并在每个设备上缓存关于该图的信息，以最大限度地减少协调开销。像Spark和Naiad一样，当集群中有足够的RAM来保存计算的工作集时，TensorFlow的工作效果最佳。 TensorFlow中的迭代使用混合方法：同一个数据流图的多个副本可以一次执行，同时共享相同的一组变量。副本可以通过变量异步共享数据，或者使用图中的同步机制（如队列）同步操作。 TensorFlow还支持图形中的迭代，该图形是CIEL和Naiad的混合：为简单起见，每个节点只有在所有输入都准备就绪时才会触发（如CIEL）;但为了提高效率，图表被表示为静态的循环数据流（如Naiad）。

12 结论

我们已经描述了TensorFlow这一灵活的基于数据流的编程模型，以及这种编程模型的单机和分布式实现。该系统源自在实施研究和在各种Google产品和服务中部署超过100个机器学习项目的实际经验。我们已经开源了TensorFlow的一个版本，并希望一个充满活力的共享社区围绕TensorFlow的使用而开发。我们很高兴看到Google以外的其他人如何在自己的工作中使用TensorFlow。 16致谢TensorFlow的发展受益于谷歌庞大而广泛的机器学习社区，特别是Google Brain团队的其他成员以及谷歌的数百名DistBelief和TensorFlow用户的建议和贡献。毫无疑问，通过倾听他们的反馈意见，TensorFlow的可用性和功能得到了极大的扩展。许多人为TensorFlow及其开源版本做出了贡献，其中包括John Giannandrea（创建支持性研究环境），Irina Kofman和Phing Turner（项目管理），Bill Gruber和David Westbrook（技术写作），Dave Andersen，Anelia Angelova，Yaroslav Bulatov，Jianmin Chen，Jerjou Cheng，George Dahl，Andrew Dai，Lucy Gao，mig Gerard，Stephan Gouws，Naveen Kumar，Geoffrey Hinton，Mrinal Kalarishnan，Anjuli Kannan，Yutaka Leon-Suematsu，Frank Li，Peter Liu，刘小兵，Nishant Patil，Pierre Sermanet，Noam Shazeer，Jascha Sohl-dickstein，Philip Tucker，Yonghui Wu，Ke Yang和Cliff Young（总贡献），Doug Fritz，Patrick Hurst，Dilip Krishnan，Daniel Smilkov，James Wexler，Jimbo Wilson， Kanit Ham Wongsuphasawat，Cassandra Xia和Big Picture团队（图表可视化），Chris Leary，Robert Springer和Stream Executor团队，Kayur Patel，Michael Piatek和coLab团队以及许多其他贡献者他是TensorFlow设计和代码库。