**TensorFlow:**

**Large-Scale Machine Learning on Heterogeneous Distributed Systems**

**(Preliminary White Paper, November 9, 2015)**

Mart´ın 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 Vie´gas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng

Google Research*∗*

# Abstract

arXiv:1603.04467v2 [cs.DC] 16 Mar 2016

TensorFlow [[1]](#_bookmark30) is an interface for expressing machine learn- ing algorithms, and an implementation for executing such al- gorithms. A computation expressed using TensorFlow can be executed with little or no change on a wide variety of hetero- geneous systems, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of hundreds of machines and thousands of computational devices such as GPU cards. The system is flexible and can be used to express a wide variety of algorithms, including training and inference algorithms for deep neural network models, and it has been used for conducting research and for deploying machine learn- ing systems into production across more than a dozen areas of computer science and other fields, including speech recogni- tion, computer vision, robotics, information retrieval, natural language processing, geographic information extraction, and computational drug discovery. This paper describes the Ten- sorFlow interface and an implementation of that interface that we have built at Google. The TensorFlow API and a reference implementation were released as an open-source package under the Apache 2.0 license in November, 2015 and are available at [www.tensorflow.org.](http://www.tensorflow.org/)

# Introduction

The Google Brain project started in 2011 to explore the use of very-large-scale deep neural networks, both for research and for use in Google’s products. As part of the early work in this project, we built DistBelief, our first-generation scalable distributed training and infer- ence system [[14],](#_bookmark43) and this system has served us well. We and others at Google have performed a wide variety of re- search using DistBelief including work on unsupervised learning [[31],](#_bookmark60) language representation [[35,](#_bookmark64) [52],](#_bookmark81) models for image classification and object detection [[16,](#_bookmark45) [48],](#_bookmark77) video classification [[27],](#_bookmark56) speech recognition [[56,](#_bookmark85) [21,](#_bookmark50) [20],](#_bookmark49)

*∗*Corresponding authors: Jeffrey Dean and Rajat Monga:

*{*jeff,rajatmonga*}*@google.com

sequence prediction [[47],](#_bookmark75) move selection for Go [[34],](#_bookmark63) pedestrian detection [[2],](#_bookmark31) reinforcement learning [[38],](#_bookmark67) and other areas [[17,](#_bookmark46) [5].](#_bookmark34) In addition, often in close collab- oration with the Google Brain team, more than 50 teams at Google and other Alphabet companies have deployed deep neural networks using DistBelief in a wide variety of products, including Google Search [[11],](#_bookmark40) our advertis- ing products, our speech recognition systems [[50,](#_bookmark79) [6,](#_bookmark35) [46],](#_bookmark76) Google Photos [[43],](#_bookmark72) Google Maps and StreetView [[19],](#_bookmark48) Google Translate [[18],](#_bookmark47) YouTube, and many others.

Based on our experience with DistBelief and a more complete understanding of the desirable system proper- ties and requirements for training and using neural net- works, we have built TensorFlow, our second-generation system for the implementation and deployment of large- scale machine learning models. TensorFlow takes com- putations described using a dataflow-like model and maps them onto a wide variety of different hardware platforms, ranging from running inference on mobile device platforms such as Android and iOS to modest- sized training and inference systems using single ma- chines containing one or many GPU cards to large-scale training systems running on hundreds of specialized ma- chines with thousands of GPUs. Having a single system that can span such a broad range of platforms signifi- cantly simplifies the real-world use of machine learning system, as we have found that having separate systems for large-scale training and small-scale deployment leads to significant maintenance burdens and leaky abstrac- tions. TensorFlow computations are expressed as stateful dataflow graphs (described in more detail in Section [2),](#_bookmark1) and we have focused on making the system both flexible enough for quickly experimenting with new models for research purposes and sufficiently high performance and robust for production training and deployment of ma- chine learning models. For scaling neural network train- ing to larger deployments, TensorFlow allows clients to easily express various kinds of parallelism through repli- cation and parallel execution of a core model dataflow

graph, with many different computational devices all col- laborating to update a set of shared parameters or other state. Modest changes in the description of the com- putation allow a wide variety of different approaches to parallelism to be achieved and tried with low effort [[14,](#_bookmark43) [29,](#_bookmark58) [42].](#_bookmark71) Some TensorFlow uses allow some flexibil- ity in terms of the consistency of parameter updates, and we can easily express and take advantage of these relaxed synchronization requirements in some of our larger de- ployments. Compared to DistBelief, TensorFlow’s pro- gramming model is more flexible, its performance is sig- nificantly better, and it supports training and using a broader range of models on a wider variety of hetero- geneous hardware platforms.

Dozens of our internal clients of DistBelief have al- ready switched to TensorFlow. These clients rely on TensorFlow for research and production, with tasks as diverse as running inference for computer vision mod- els on mobile phones to large-scale training of deep neural networks with hundreds of billions of parame- ters on hundreds of billions of example records using many hundreds of machines [[11,](#_bookmark40) [47,](#_bookmark75) [48,](#_bookmark77) [18,](#_bookmark47) [53,](#_bookmark82) [41].](#_bookmark70) Although these applications have concentrated on ma- chine learning and deep neural networks in particular, we expect that TensorFlow’s abstractions will be useful in a variety of other domains, including other kinds of machine learning algorithms, and possibly other kinds of numerical computations. We have open-sourced the TensorFlow API and a reference implementation under the Apache 2.0 license in November, 2015, available at [www.tensorflow.org.](http://www.tensorflow.org/)

The rest of this paper describes TensorFlow in more detail. Section [2](#_bookmark1) describes the programming model and basic concepts of the TensorFlow interface, and Section [3](#_bookmark4) describes both our single machine and distributed imple- mentations. Section [4](#_bookmark11) describes several extensions to the basic programming model, and Section [5](#_bookmark15) describes several optimizations to the basic implementations. Sec- tion [6](#_bookmark16) describes some of our experiences in using Ten- sorFlow, Section [7](#_bookmark18) describes several programming id- ioms we have found helpful when using TensorFlow, and Section [9](#_bookmark21) describes several auxiliary tools we have built around the core TensorFlow system. Sections [10](#_bookmark24) and [11](#_bookmark25) discuss future and related work, respectively, and Sec- tion [12](#_bookmark29) offers concluding thoughts.

# Programming Model and Basic Concepts

A TensorFlow computation is described by a directed *graph*, which is composed of a set of *nodes*. The graph represents a dataflow computation, with extensions for allowing some kinds of nodes to maintain and update persistent state and for branching and looping control

structures within the graph in a manner similar to Naiad [[36].](#_bookmark65) Clients typically construct a computational graph using one of the supported frontend languages (C++ or Python). An example fragment to construct and then ex- ecute a TensorFlow graph using the Python front end is shown in Figure [1,](#_bookmark0) and the resulting computation graph in Figure [2.](#_bookmark2)

In a TensorFlow graph, each *node* has zero or more in- puts and zero or more outputs, and represents the instan- tiation of an *operation*. Values that flow along normal edges in the graph (from outputs to inputs) are *tensors*, arbitrary dimensionality arrays where the underlying el- ement type is specified or inferred at graph-construction time. Special edges, called *control dependencies*, can also exist in the graph: no data flows along such edges, but they indicate that the source node for the control de- pendence must finish executing before the destination node for the control dependence starts executing. Since our model includes mutable state, control dependencies can be used directly by clients to enforce happens before relationships. Our implementation also sometimes in- serts control dependencies to enforce orderings between otherwise independent operations as a way of, for exam- ple, controlling the peak memory usage.

## Operations and Kernels

An *operation* has a name and represents an abstract com- putation (e.g., “matrix multiply”, or “add”). An opera- tion can have *attributes*, and all attributes must be pro- vided or inferred at graph-construction time in order to instantiate a node to perform the operation. One com- mon use of attributes is to make operations polymorphic over different tensor element types (e.g., add of two ten- sors of type float versus add of two tensors of type int32). A *kernel* is a particular implementation of an operation that can be run on a particular type of device (e.g., CPU or GPU). A TensorFlow binary defines the sets of opera- tions and kernels available via a registration mechanism, and this set can be extended by linking in additional op- eration and/or kernel definitions/registrations. Table [1](#_bookmark3) shows some of the kinds of operations built into the core TensorFlow library.

## Sessions

Clients programs interact with the TensorFlow system by creating a *Session*. To create a computation graph, the Session interface supports an *Extend* method to augment the current graph managed by the session with additional nodes and edges (the initial graph when a session is cre- ated is empty). The other primary operation supported

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



C

...

ReLU

Add

b

MatMul

W

x

Figure 2: Corresponding computation graph for Figure [1](#_bookmark0)

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

Table 1: Example TensorFlow operation types

by the session interface is *Run*, which takes a set of out- put names that need to be computed, as well as an op- tional set of tensors to be fed into the graph in place of certain outputs of nodes. Using the arguments to Run, the TensorFlow implementation can compute the transi- tive closure of all nodes that must be executed in order to compute the outputs that were requested, and can then

arrange to execute the appropriate nodes in an order that respects their dependencies (as described in more detail in [3.1).](#_bookmark5) Most of our uses of TensorFlow set up a Session with a graph once, and then execute the full graph or a few distinct subgraphs thousands or millions of times via Run calls.

## Variables

In most computations a graph is executed multiple times. Most tensors do not survive past a single execution of the graph. However, a *Variable* is a special kind of opera- tion that returns a handle to a persistent mutable tensor that survives across executions of a graph. Handles to these persistent mutable tensors can be passed to a hand- ful of special operations, such as *Assign* and *AssignAdd* (equivalent to +=) that mutate the referenced tensor. For machine learning applications of TensorFlow, the param- eters of the model are typically stored in tensors held in variables, and are updated as part of the *Run* of the train- ing graph for the model.

# Implementation

The main components in a TensorFlow system are the *client*, which uses the Session interface to communicate with the *master*, and one or more *worker processes*, with each worker process responsible for arbitrating access to one or more computational *devices* (such as CPU cores or GPU cards) and for executing graph nodes on those devices as instructed by the master. We have both *lo- cal* and *distributed* implementations of the TensorFlow interface. The local implementation is used when the client, the master, and the worker all run on a single ma- chine in the context of a single operating system process (possibly with multiple devices, if for example, the ma- chine has many GPU cards installed). The distributed implementation shares most of the code with the local implementation, but extends it with support for an en- vironment where the client, the master, and the workers can all be in different processes on different machines. In our distributed environment, these different tasks are containers in jobs managed by a cluster scheduling sys- tem [[51].](#_bookmark80) These two different modes are illustrated in Figure [3.](#_bookmark7) Most of the rest of this section discusses is- sues that are common to both implementations, while Section [3.3](#_bookmark9) discusses some issues that are particular to the distributed implementation.

## Devices

Devices are the computational heart of TensorFlow. Each worker is responsible for one or more devices, and each device has a device type, and a name. Device names are composed of pieces that identify the de- vice’s type, the device’s index within the worker, and, in our distributed setting, an identification of the job and task of the worker (or localhost for the case where the devices are local to the process). Example device names are "/job:localhost/device:cpu:0" or "/job:worker/task:17/device:gpu:3". We

have implementations of our Device interface for CPUs and GPUs, and new device implementations for other de- vice types can be provided via a registration mechanism. Each device object is responsible for managing alloca- tion and deallocation of device memory, and for arrang- ing for the execution of any kernels that are requested by higher levels in the TensorFlow implementation.

## Tensors

A tensor in our implementation is a typed, multi- dimensional array. We support a variety of tensor ele- ment types, including signed and unsigned integers rang- ing in size from 8 bits to 64 bits, IEEE float and double types, a complex number type, and a string type (an ar- bitrary byte array). Backing store of the appropriate size is managed by an allocator that is specific to the device on which the tensor resides. Tensor backing store buffers are reference counted and are deallocated when no refer- ences remain.

# Single-Device Execution

Let’s first consider the simplest execution scenario: a sin- gle worker process with a single device. The nodes of the graph are executed in an order that respects the depen- dencies between nodes. In particular, we keep track of a count per node of the number of dependencies of that node that have not yet been executed. Once this count drops to zero, the node is eligible for execution and is added to a ready queue. The ready queue is processed in some unspecified order, delegating execution of the ker- nel for a node to the device object. When a node has finished executing, the counts of all nodes that depend on the completed node are decremented.

# Multi-Device Execution

Once a system has multiple devices, there are two main complications: deciding which device to place the com- putation for each node in the graph, and then managing the required communication of data across device bound- aries implied by these placement decisions. This subsec- tion discusses these two issues.

## Node Placement

Given a computation graph, one of the main responsi- bilities of the TensorFlow implementation is to map the computation onto the set of available devices. A sim- plified version of this algorithm is presented here. See Section [4.3](#_bookmark14) for extensions supported by this algorithm.

One input to the placement algorithm is a cost model, which contains estimates of the sizes (in bytes) of the



single process

client

master

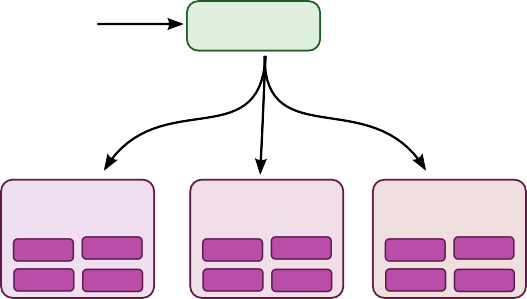
session run

execute subgraph

CPU0

worker

GPU0 GPU1 ...



client process

session run

master process

execute subgraph

worker process 1

worker process 2

worker process 3

GPU0

...

GPU0

...

GPU0

...

GPU1 CPU0

GPU1 CPU0

GPU1 CPU0

Figure 3: Single machine and distributed system structure

input and output tensors for each graph node, along with estimates of the computation time required for each node when presented with its input tensors. This cost model is either statically estimated based on heuristics associated with different operation types, or is measured based on an actual set of placement decisions for earlier execu- tions of the graph.

The placement algorithm first runs a simulated execu- tion of the graph. The simulation is described below and ends up picking a device for each node in the graph using greedy heuristics. The node to device placement gener- ated by this simulation is also used as the placement for the real execution.

The placement algorithm starts with the sources of the computation graph, and simulates the activity on each device in the system as it progresses. For each node that is reached in this traversal, the set of feasible devices is considered (a device may not be feasible if the device does not provide a kernel that implements the particular operation). For nodes with multiple feasible devices, the placement algorithm uses a greedy heuristic that exam- ines the effects on the completion time of the node of placing the node on each possible device. This heuristic takes into account the estimated or measured execution time of the operation on that kind of device from the cost model, and also includes the costs of any communica- tion that would be introduced in order to transmit inputs to this node from other devices to the considered device. The device where the node’s operation would finish the soonest is selected as the device for that operation, and the placement process then continues onwards to make placement decisions for other nodes in the graph, includ- ing downstream nodes that are now ready for their own simulated execution. Section [4.3](#_bookmark14) describes some exten- sions that allow users to provide hints and partial con- straints to guide the placement algorithm. The placement algorithm is an area of ongoing development within the system.

## Cross-Device Communication

Once the node placement has been computed, the graph is partitioned into a set of subgraphs, one per device. Any cross-device edge from **x** to **y** is removed and replaced by an edge from **x** to a new *Send* node in **x**’s subgraph and an edge from a corresponding *Receive* node to **y** in **y**’s subgraph. See Figure [4](#_bookmark8) for an example of this graph transformation.





Device B

b c

y

W

a

Device A

x



Device B

b c

y

recv

W

recv

a

Device A

x

send

send

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

At runtime, the implementations of the Send and Re- ceive nodes coordinate to transfer data across devices. This allows us to isolate all communication inside Send and Receive implementations, which simplifies the rest of the runtime.

When we insert Send and Receive nodes, we canoni- calize all users of a particular tensor on a particular de- vice to use a single Receive node, rather than one Re- ceive node per downstream user on a particular device. This ensures that the data for the needed tensor is only transmitted once between a source device destination device pair, and that memory for the tensor on the desti- nation device is only allocated once, rather than multiple times (e.g., see nodes **b** and **c** in Figure [4)](#_bookmark8)

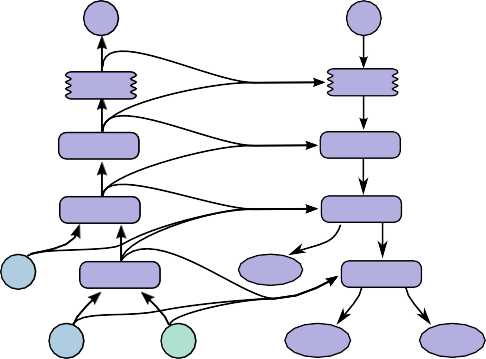
*→*

By handling communication in this manner, we also allow the scheduling of individual nodes of the graph on different devices to be decentralized into the work- ers: the Send and Receive nodes impart the necessary

synchronization between different workers and devices, and the master only needs to issue a single Run request per graph execution to each worker that has any nodes for the graph, rather than being involved in the scheduling of every node or every cross-device communication. This makes the system much more scalable and allows much finer-granularity node executions than if the scheduling were forced to be done by the master.

# Distributed Execution

Distributed execution of a graph is very similar to multi- device execution. After device placement, a subgraph is



C



...

...

ReLU

dReLU

Add

dAdd

b

MatMul

dCdb

dMatMul

W

x

dCdW

dCdx

Figure 5: Gradients computed for graph in Figure [2](#_bookmark2)

created per device. Send/Receive node pairs that com-

municate across worker processes use remote communi-

cation mechanisms such as TCP or RDMA to move data across machine boundaries.

## Fault Tolerance

Failures in a distributed execution can be detected in a variety of places. The main ones we rely on are (a) an error in a communication between a Send and Receive node pair, and (b) periodic health-checks from the master process to every worker process.

When a failure is detected, the entire graph execution is aborted and restarted from scratch. Recall however that Variable nodes refer to tensors that persist across ex- ecutions of the graph. We support consistent checkpoint- ing and recovery of this state on a restart. In partcular, each Variable node is connected to a Save node. These Save nodes are executed periodically, say once every N iterations, or once every N seconds. When they execute, the contents of the variables are written to persistent stor- age, e.g., a distributed file system. Similarly each Vari- able is connected to a Restore node that is only enabled in the first iteration after a restart. See Section [4.2](#_bookmark12) for details on how some nodes can only be enabled on some executions of the graph.

# Extensions

In this section we describe several more advanced fea- tures of the basic programming model that was intro- duced in Section [2.](#_bookmark1)

# Gradient Computation

Many optimization algorithms, including common ma- chine learning training algorithms like stochastic gradi- ent descent [[45],](#_bookmark74) compute the gradient of a cost function with respect to a set of inputs. Because this is such a

common need, TensorFlow has built-in support for au- tomatic gradient computation. If a tensor *C* in a Ten- sorFlow graph depends, perhaps through a complex sub- graph of operations, on some set of tensors *Xk* , then there is a built-in function that will return the tensors *dC/dXk* . Gradient tensors are computed, like other tensors, by extending the TensorFlow graph, using the following procedure.

When TensorFlow needs to compute the gradient of a tensor *C* with respect to some tensor *I* on which *C* depends, it first finds the path in the computation graph from *I* to *C*. Then it backtracks from *C* to *I*, and for each operation on the backward path it adds a node to the TensorFlow graph, composing the partial gradients along the backwards path using the chain rule. The newly added node computes the “gradient function” for the cor- responding operation in the forward path. A gradient function may be registered by any operation. This func- tion takes as input not only the partial gradients com- puted already along the backward path, but also, option- ally, the inputs and outputs of the forward operation. Fig- ure [5](#_bookmark10) shows gradients for a cost computed from the ex- ample of Figure [2.](#_bookmark2) Grey arrows show potential inputs to gradient functions that are not used for the particular operations shown. The addition needed to Figure [1](#_bookmark0) to compute these gradients is:

*{ }*

*{ }*

[db,dW,dx] = tf.gradients(C, [b,W,x])

In general an operation may have multiple outputs, and *C* may only depend on some of them. If, for example, operation *O* has two outputs *y*1 and *y*2, and *C* only de- pends on *y*2, then the first input to *O*’s gradient function is set to 0 since *dC/dy*1 = 0.

Automatic gradient computation complicates opti- mization, particularly of memory usage. When execut- ing “forward” computation subgraphs, i.e., those that are explicitly constructed by the user, a sensible heuristic breaks ties when deciding which node to execute next by observing the order in which the graph was constructed.

This generally means that temporary outputs are con- sumed soon after being constructed, so their memory can be reused quickly. When the heuristic is ineffective, the user can change the order of graph construction, or add control dependencies as described in Section [5.](#_bookmark15) When gradient nodes are automatically added to the graph, the user has less control, and the heuristics may break down. In particular, because gradients reverse the forward com- putation order, tensors that are used early in a graph’s execution are frequently needed again near the end of a gradient computation. Such tensors can hold on to a lot of scarce GPU memory and unnecessarily limit the size of computations. We are actively working on improve- ments to memory management to deal better with such cases. Options include using more sophisticated heuris- tics to determine the order of graph execution, recom- puting tensors instead of retaining them in memory, and swapping out long-lived tensors from GPU memory to more plentiful host CPU memory.

# Partial Execution

Often a client wants to execute just a subgraph of the entire execution graph. To support this, once the client has set up a computation graph in a Session, our Run method allows them to execute an arbitrary subgraph of the whole graph, and to inject arbitrary data along any edge in the graph, and to retrieve data flowing along any edge in the graph.

Each node in the graph has a name, and each output of a node is identified by the source node name and the out- put port from the node, numbered from 0 (e.g., “bar:0” refers to the 1st output of the “bar” node, while “bar:1” refers to the 2nd output).

Two arguments to the Run call help define the exact subgraph of the computation graph that will be executed. First, the Run call accepts inputs, an optional mapping of *name*:*port* names to “fed” tensors values. Second, the Run call accepts output names, a list of output *name*[:*port*] specifications indicating which nodes should be executed, and, if the port portion is present in a name, that that particular output tensor value for the node should be returned to the client if the Run call completes successfully.

The graph is transformed based on the values of in- puts and outputs. Each node:port specified in inputs is replaced with a **feed** node, which will pick up the pro- vided input tensor from specially-initialized entries in a Rendezvous object used for the Run call. Similarly, each output name with a port is connected to a special **fetch** node that arranges to save the output tensor and return it to the client when the Run call is complete. Finally, once the graph has been rewritten with the insertion of these

fetch



e

f

d

c

a

b



f

d

c

a

b

feed

Figure 6: Before and after graph transformation for par- tial execution

special **feed** and **fetch** nodes, the set of nodes to execute can be determined by starting at each of the nodes named by any output and working backwards in the graph using the graph dependencies to determine the full set of nodes that must be executed in the rewritten graph in order to compute the outputs. Figure [6](#_bookmark13) shows an original graph on the left, and the transformed graph that results when Run is invoked with inputs== **b** and outputs== **f:0** . Since we only need to compute the output of node **f**, we will not execute nodes **d** and **e**, since they have no con- tribution to the output of **f**.

# Device Constraints

*{ } { }*

TensorFlow clients can control the placement of nodes on devices by providing partial constraints for a node about which devices it can execute on. For ex- ample, *“only place this node on a device of type GPU”*, or *“this node can be placed on any device in*

*/job:worker/task:17”*, or *“Colocate this node with the node named variable13”*. Within the con- fines of these constraints, the placement algorithm is re- sponsible for choosing an assignment of nodes to de- vices that provides fast execution of the computation and also satisfies various constraints imposed by the devices themselves, such as limiting the total amount of memory needed on a device in order to execute its subset of graph nodes.

Supporting such constraints requires changes to the placement algorithm described in Section [3.2.1.](#_bookmark6) We first compute the feasible set of devices for each node, and then use union-find on the graph of colocation constraints to compute the graph components that must be placed together. For each such component, we compute the in- tersection of the feasible device sets. The computed fea- sible device set per node fits easily into the placement algorithm’s simulator.

# Control Flow

Although dataflow graphs without any explicit control flow are quite expressive, we have observed a number of cases where supporting conditionals and loops can lead to more concise and efficient representations of machine learning algorithms.

Much as in the dataflow-machine approach described by Arvind [[3],](#_bookmark32) we introduce a small set of primitive con- trol flow operators into TensorFlow and generalize Ten- sorFlow to handle cyclic dataflow graphs. The *Switch* and *Merge* operators allow us to skip the execution of an entire subgraph based on the value of a boolean ten- sor. The *Enter*, *Leave*, and *NextIteration* operators allow us to express iteration. High-level programming con- structs such as if-conditionals and while-loops can be easily compiled into dataflow graphs with these control flow operators.

The TensorFlow runtime implements a notion of tags and frames conceptually similar to the MIT Tagged- Token machine [[4].](#_bookmark33) Each iteration of a loop is uniquely identified by a tag, and its execution state is represented by a frame. An input can enter an iteration whenever it becomes available; thus, multiple iterations can be exe- cuted concurrently.

TensorFlow uses a distributed coordination mecha- nism to execute graphs with control flow. In general, a loop can contain nodes that are assigned to many dif- ferent devices. Therefore, managing the state of a loop becomes a problem of distributed termination detection. TensorFlow’s solution is based on graph rewriting. Dur- ing the graph partitioning, we automatically add control nodes to each partition. These nodes implement a small state machine that orchestrates the start and termination of each iteration, and decides the termination of the loop. For each iteration, the device that owns the loop termi- nation predicate sends a tiny control message to every participating device.

As explained above, we often train machine learning models by gradient descent, and represent gradient com- putations as part of dataflow graphs. When a model includes control-flow operations, we must account for them in the corresponding gradient computation. For ex- ample, the gradient computation for a model with an if- conditional will need to know which branch of the con- ditional was taken, then apply the gradient logic to this branch. Similarly, the gradient computation for a model with a while-loop will need to know how many iterations were taken, and will also rely on the intermediate values computed during those iterations. The basic technique is to rewrite the graph so to memorize the values needed for the gradient computation. We omit the somewhat intri- cate details of this encoding.

# Input Operations

Although input data can be provided to a computation via feed nodes, another common mechanism used for train- ing large-scale machine learning models is to have spe- cial input operation nodes in the graph, which are typi- cally configured with a set of filenames and which yield a tensor containing one or more examples from the data stored in that set of files each time they are executed. This allows data to be read directly from the underlying storage system into the memory of the machine that will perform subsequent processing on the data. In configura- tions where the client process is separate from the worker process, if the data were fed, it typically would require an extra network hop (from the storage system to the client and then from the client to the worker vs. directly from the storage system to ther worker when using an input node).

# Queues

Queues are a useful feature that we have added to Ten- sorFlow. They allow different portions of the graph to execute asynchronously, possibly at different candences, and to hand off data through Enqueue and Dequeue op- erations. Enqueue operations can block until space be- comes available in the queue, and Dequeue operations can block until a desired minimum number of elements are available in the queue. One use of queues is to allow input data to be prefetched from disk files while a previ- ous batch of data is still being processed by the compu- tational portion of a machine learning model. They can also be used for other kinds of grouping, including accu- mulating many gradients in order to compute some more complex combination of gradients over a larger batch, or to group different input sentences for recurrent lan- guage models into bins of sentences that are approxi- mately the same length, which can then be processed more efficiently.

In addition to normal FIFO queues, we have also im- plemented a shuffling queue, which randomly shuffles its elements within a large in-memory buffer. This shuffling functionality is useful for machine learning algorithms that want to randomize the order in which they process examples, for example.

# Containers

A *Container* is the mechanism within TensorFlow for managing longer-lived mutable state. The backing store for a *Variable* lives in a container. The default con- tainer is one that persists until the process terminates, but we also allow other named containers. A container

can be reset by clearing it of its contents entirely. Us- ing containers, it is possible to share state even across completely disjoint computation graphs associated with different Sessions.

# Optimizations

In this section, we describe some of the optimizations in the TensorFlow implementation that improve perfor- mance or resource usage of the system.

# Common Subexpression Elimination

Since the construction of computation graphs is often done by many different layers of abstractions in the client code, computation graphs can easily end up with redun- dant copies of the same computation. To handle this, we have implemented a common subexpression pass similar to the algorithm described by Click [[12]](#_bookmark41) that runs over the computation graph and canonicalizes multiple copies of operations with identical inputs and operation types to just a single one of these nodes, and redirects graph edges appropriately to reflect this canonicalization.

# Controlling Data Communication and Memory Usage

Careful scheduling of TensorFlow operations can result in better performance of the system, in particular with respect to data transfers and memory usage. Specifically, scheduling can reduce the time window during which intermediate results need to be kept in memory in be- tween operations and hence the peak memory consump- tion. This reduction is particularly important for GPU devices where memory is scarce. Furthermore, orches- trating the communication of data across devices can re- duce contention for network resources.

While there are many opportunities for scheduling op- timizations, here we focus on one that we found partic- ularly necessary and effective. It concerns the schedul- ing of Receive nodes for reading remote values. If no precautions are taken, these nodes may start much ear- lier than necessary, possibly all at once when execution starts. By performing an as-soon-as-possible/as-late-as- possible (ASAP/ALAP) calculation, of the kind common in operations research, we analyze the critical paths of graphs, in order to estimate when to start the Receive nodes. We then insert control edges with the aim of de- laying the start of these nodes until just before their re- sults are needed.

# Asynchronous Kernels

In addition to normal synchronous kernels that complete their execution at the end of the Compute method, our framework also supports non-blocking kernels. Such non-blocking kernels use a slightly different interface whereby the Compute method is passed a continuation that should be invoked when the kernel’s execution is complete. This is an optimization for environments where having many active threads is relatively expensive in terms of memory usage or other resources, and allows us to avoid tying up an execution thread for unbounded periods of time while waiting for I/O or other events to occur. Examples of asynchronous kernels include the **Receive** kernel, and the **Enqueue** and **Dequeue** kernels (which might need to block if queue space is not avail- able or if no data is available to be read, respectively).

# Optimized Libraries for Kernel Imple- mentations

We often make use of pre-existing highly-optimized nu- merical libraries to implement kernels for some opera- tions. For example, there are a number of optimized li- braries for performing matrix multiplies on different de- vices, including BLAS [[15]](#_bookmark44) and cuBLAS [[39],](#_bookmark68) or GPU libraries for convolutional kernels for deep neural nets such as cuda-convnet [[28]](#_bookmark57) and cuDNN [[9].](#_bookmark38) Many of our kernel implementations are relatively thin wrappers around such optimized libraries.

We make fairly extensive use of the open-source Eigen linear algebra library [[25]](#_bookmark54) for many of the kernel imple- mentations in the system. As one part of the develop- ment of TensorFlow, our team (primarily Benoit Steiner) has extended the open source Eigen library with support for arbitrary dimensionality tensor operations.

# Lossy Compression

Some machine learning algorithms, including those typ- ically used for training neural networks, are tolerant of noise and reduced precision arithmetic. In a manner sim- ilar to the DistBelief system [[14],](#_bookmark43) we often use lossy compression of higher precision internal representations when sending data between devices (sometimes within the same machine but especially across machine bound- aries). For example, we often insert special conversion nodes that convert 32-bit floating point representations into a 16-bit floating point representation (not the pro- posed IEEE 16-bit floating point standard, but rather just a 32-bit IEEE 794 float format, but with 16 bits less pre- cision in the mantissa), and then convert back to a 32- bit representation on the other side of the communica- tion channel (by just filling in zeroes for the lost portion

of the mantissa, since that’s less computationally expen- sive than doing the mathematically correct probabilistic rounding when doing this 32 16 32-bit conver- sion).

*→ →*

# Status and Experience

The TensorFlow interface and a reference implemen- tation have been open sourced under an Apache 2.0 license, and the system is available for download at [www.tensorflow.org.](http://www.tensorflow.org/) The system includes detailed docu- mentation, a number of tutorials, and a number of exam- ples demonstrating how to use the system for a variety of different machine learning tasks. The examples in- clude models for classifying hand-written digits from the MNIST dataset (the “hello world” of machine learning algorithms) [[32],](#_bookmark61) classifying images from the CIFAR- 10 dataset [[30],](#_bookmark59) doing language modeling using a recur- rent LSTM [[22]](#_bookmark51) network, training word embedding vec- tors [[35]](#_bookmark64) and more.

The system includes front-ends for specifying Tensor- Flow computations in Python and C++, and we expect other front-ends to be added over time in response to the desires of both internal Google users and the broader open-source community.

We have quite a few machine learning models in our previous DistBelief system [[14]](#_bookmark43) that we have migrated over to TensorFlow. The rest of this section discusses some lessons we have learned that are generalizable for any such migration of machine learning models from one system to another, and therefore may be valuable to oth- ers.

In particular, we focus on our lessons from porting a state-of-the-art convolutional neural network for image recognition termed *Inception* [[23].](#_bookmark52) This image recogni- tion system classifies 224 224 pixel images into one of 1000 labels (e.g., “cheetah”, “garbage truck”, etc.). Such a model comprises 13.6 million learnable parame- ters and 36,000 operations when expressed as a Tensor- Flow graph. Running inference on a single image re- quires 2 billion multiply-add operations.

*×*

After building all necessary mathematical operations in TensorFlow, assembling and debugging all 36,000 op- erations into the correct graph structure proved challeng- ing. Validating correctness is a difficult enterprise be- cause the system is inherently stochastic and only in- tended to behave in a certain way in expectation — po- tentially after hours of computation. Given these cir- cumstances, we found the following strategies critical for porting the Inception model to TensorFlow:

1. *Build tools to gain insight into the exact number of parameters in a given model.* Such tools demon-

strated subtle flaws in a complex network architec- ture specification. In particular we were able to identify operations and variables instantiated incor- rectly due to automatic broadcasting in a mathemat- ical operation across a dimension.

1. *Start small and scale up.* The first convolutional neural network that we ported from our previ- ous system was a small network employed on the CIFAR-10 data set [[30].](#_bookmark59) Debugging such a network elucidated subtle edge cases in individual opera- tions (e.g., max-pooling) within the machine learn- ing system that would have been practically indeci- pherable in more complex models.
2. *Always ensure that the objective (loss function) matches between machine learning systems when learning is turned off.* Setting the learning rate to be zero helped us identify unexpected behavior in how we had randomly initialized variables in a model. Such an error would have been difficult to identify in a dynamic, training network.
3. *Make a single machine implementation match be- fore debugging a distributed implementation.* This strategy helped us delineate and debug discrep- ancies in training performance between machine learning system. In particular, we identified bugs due to race conditions and non-atomic operations incorrectly assumed to be atomic.
4. *Guard against numerical errors.* Numerical li- braries are inconsistent in how they handle non- finite floating point values. Convolutional neu- ral networks are particularly susceptible to numer- ical instability and will tend to diverge quite regu- larly during experimentation and debugging phases. Guarding against this behavior by checking for non- finite floating point values allows one to detect er- rors in real time as opposed to identifying divergent behavior post-hoc.
5. *Analyze pieces of a network and understand the magnitude of numerical error.* Running subsec- tions of a neural network in parallel on two machine learning systems provides a precise method to en- sure that a numerical algorithm is identical across two systems. Given that such algorithms run with floating point precision, it is important to predict and understand the magnitude of expected numer- ical error in order to judge whether a given compo- nent is correctly implemented (e.g., distinguishing between *“within 1e-2, great!”* and *“within 1e-2: why is it so incorrect?!”*).



Parameter Device(s)

ΔP

Add

Device A

Device B

Device C

Client

Update

model

model

model

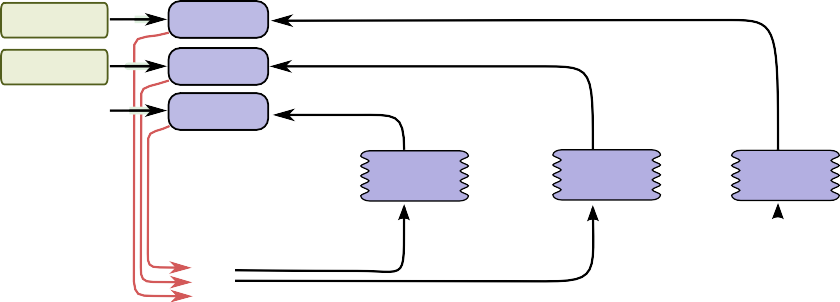
input

input

input

P

Synchronous Data Parallelism



Parameter Device(s)

Client 3

Client 2

Client 1

Update

Update Update

ΔP ΔP

ΔP

Device A

Device B

Device C

model model model

input

input

input

P

Asynchronous Data Parallelism

Figure 7: Synchronous and asynchronous data parallel training

Validating complex mathematical operations in the presence of an inherently stochastic system is quite chal- lenging. The strategies outlined above proved invaluable in gaining confidence in the system and ultimately in in- stantiating the Inception model in TensorFlow. The end result of these efforts resulted in a 6-fold speed improve- ment in training time versus our existing DistBelief im- plementation of the model and such speed gains proved indispensable in training a new class of larger-scale im- age recognition models.

# Common Programming Idioms

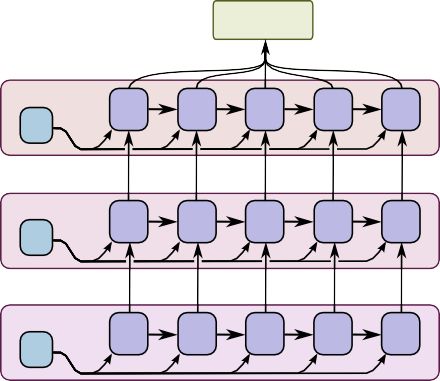
TensorFlow’s basic dataflow graph model can be used in a variety of ways for machine learning applications. One domain we care about is speeding up training of com- putationally intensive neural network models on large datasets. This section describes several techniques that we and others have developed in order to accomplish this, and illustrates how to use TensorFlow to realize these various approaches.

The approaches in this subsection assume that the model is being trained using stochastic gradient descent (SGD) with relatively modest-sized mini-batches of 100 to 1000 examples.

## Data Parallel Training

One simple technique for speeding up SGD is to paral- lelize the computation of the gradient for a mini-batch across mini-batch elements. For example, if we are us- ing a mini-batch size of 1000 elements, we can use 10 replicas of the model to each compute the gradient for 100 elements, and then combine the gradients and apply updates to the parameters synchronously, in order to be- have exactly as if we were running the sequential SGD algorithm with a batch size of 1000 elements. In this case, the TensorFlow graph simply has many replicas of the portion of the graph that does the bulk of the model computation, and a single client thread drives the entire training loop for this large graph. This is illustrated in the top portion of Figure [7.](#_bookmark17)

This approach can also be made asynchronous, where the TensorFlow graph has many replicas of the portion of the graph that does the bulk of the model computation, and each one of these replicas also applies the parame- ter updates to the model parameters asynchronously. In this configuration, there is one client thread for each of the graph replicas. This is illustrated in the bottom por- tion of Figure [7.](#_bookmark17) This asynchronous approach was also described in [[14].](#_bookmark43)



Client

Device 3

P3

C

C

C

C

C

Device 2

P2

B

B

B

B

B

Device 1

P1

A

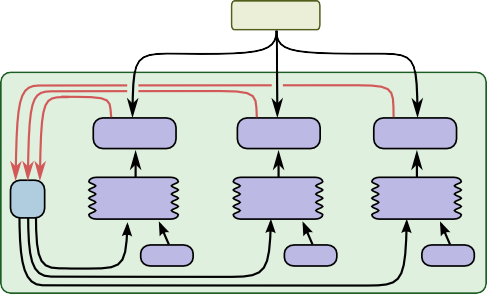
A

A

A

A

Figure 8: Model parallel training



Client

Update

Update

Update

P

model

model

model

input

input

input

Figure 9: Concurrent steps

## Model Parallel Training

Model parallel training, where different portions of the model computation are done on different computational devices simultaneously for the same batch of examples, is also easy to express in TensorFlow. Figure [8](#_bookmark19) shows an example of a recurrent, deep LSTM model used for sequence to sequence learning (see [[47]),](#_bookmark75) parallelized across three different devices.

## Concurrent Steps for Model Computation Pipelining

Another common way to get better utilization for train- ing deep neural networks is to pipeline the computation of the model within the same devices, by running a small number of concurrent steps within the same set of de- vices. This is shown in Figure [9.](#_bookmark20) It is somewhat similar to asynchronous data parallelism, except that the paral- lelism occurs within the same device(s), rather than repli- cating the computation graph on different devices. This allows “filling in the gaps” where computation of a sin- gle batch of examples might not be able to fully utilize the full parallelism on all devices at all times during a single step.

# Performance

*A future version of this white paper will have a compre- hensive performance evaluation section of both the sin- gle machine and distributed implementations.*

# Tools

This section describes some tools we have developed that sit alongside the core TensorFlow graph execution en- gine.

# TensorBoard: Visualization of graph structures and summary statistics

In order to help users understand the structure of their computation graphs and also to understand the overall behavior of machine learning models, we have built Ten- sorBoard, a companion visualization tool for TensorFlow that is included in the open source release.

## Visualization of Computation Graphs

Many of the computation graphs for deep neural net- works can be quite complex. For example, the computa- tion graph for training a model similar to Google’s Incep- tion model [[48],](#_bookmark77) a deep convolutional neural net that had the best classification performance in the ImageNet 2014 contest, has over 36,000 nodes in its TensorFlow compu- tation graph, and some deep recurrent LSTM models for language modeling have more than 15,000 nodes.

Due to the size and topology of these graphs, naive vi- sualization techniques often produce cluttered and over- whelming diagrams. To help users see the underlying organization of the graphs, the algorithms in Tensor- Board collapse nodes into high-level blocks, highlighting groups with identical structures. The system also sep- arates out high-degree nodes, which often serve book- keeping functions, into a separate area of the screen. Do- ing so reduces visual clutter and focuses attention on the core sections of the computation graph.

The entire visualization is interactive: users can pan, zoom, and expand grouped nodes to drill down for de- tails. An example of the visualization for the graph of a deep convolutional image model is shown in Figure [10.](#_bookmark22)

## Visualization of Summary Data

When training machine learning models, users often want to be able to examine the state of various aspects of the model, and how this state changes over time. To this end, TensorFlow supports a collection of different Summary operations that can be inserted into the graph,

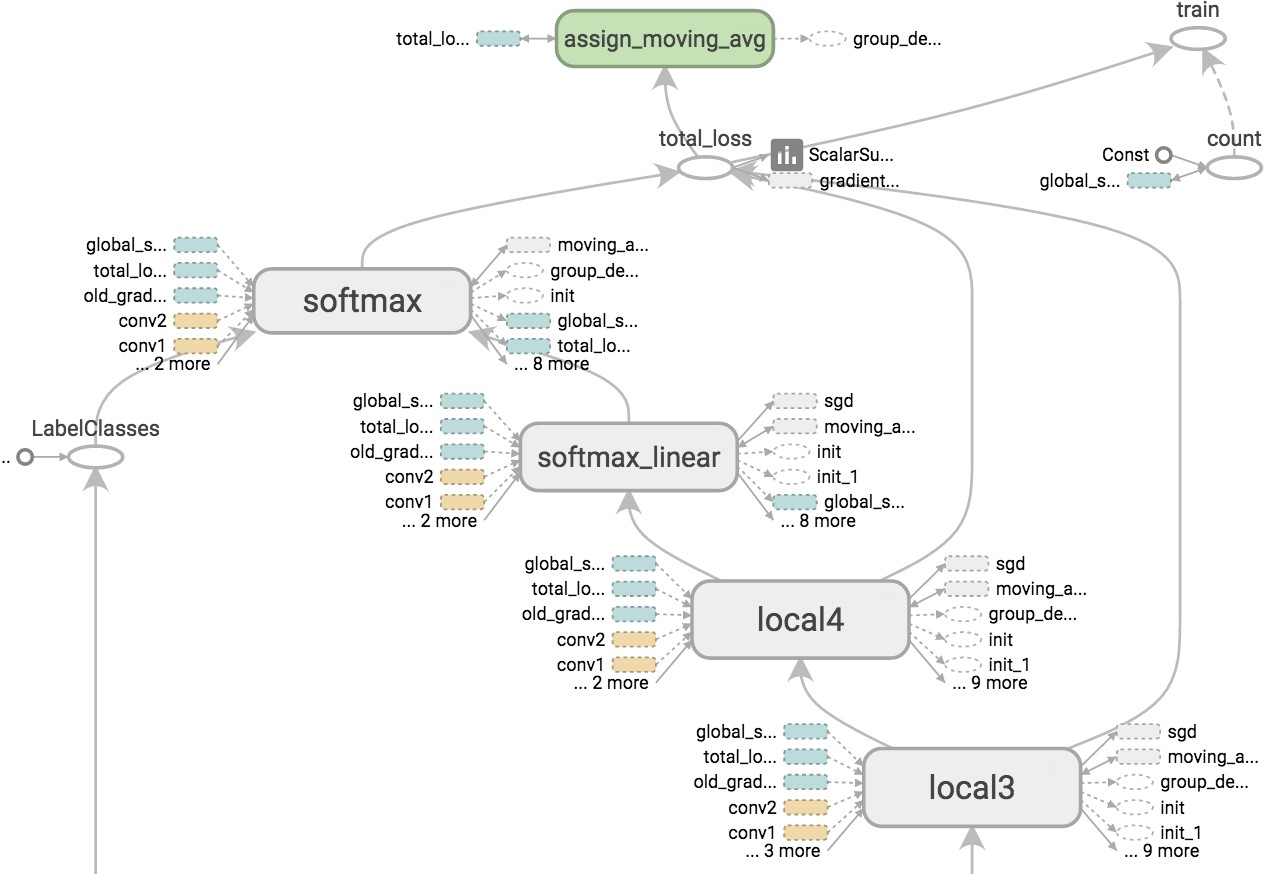


Figure 10: TensorBoard graph visualization of a convolutional neural network model

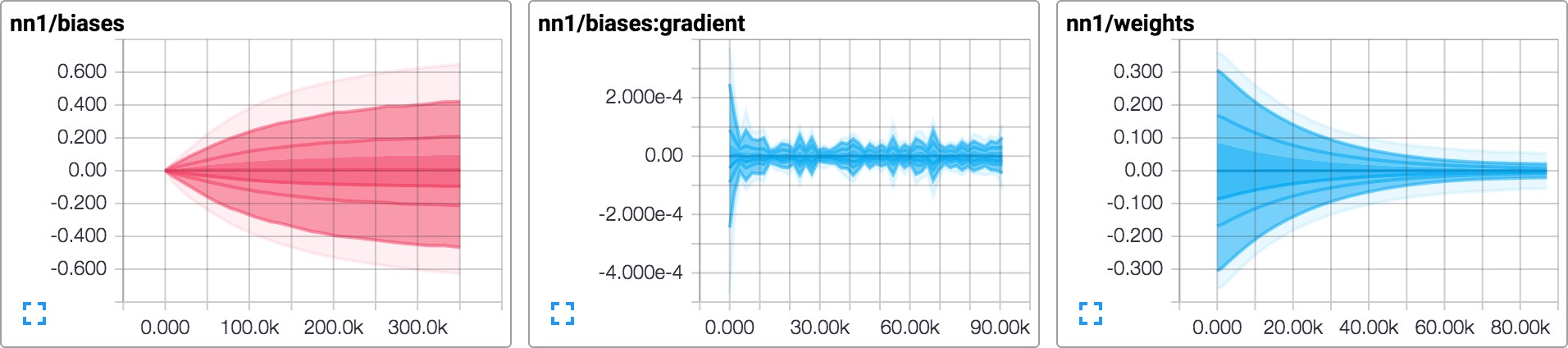


Figure 11: TensorBoard graphical display of model summary statistics time series data

including scalar summaries (e.g., for examining overall properties of the model, such as the value of the loss function averaged across a collection of examples, or the time taken to execute the computation graph), histogram- based summaries (e.g., the distribution of weight values in a neural network layer), or image-based summaries (e.g., a visualization of the filter weights learned in a convolutional neural network). Typically computation graphs are set up so that Summary nodes are included to monitor various interesting values, and every so often during execution of the training graph, the set of sum- mary nodes are also executed, in addition to the normal set of nodes that are executed, and the client driver pro- gram writes the summary data to a log file associated with the model training. The TensorBoard program is then configured to watch this log file for new summary

records, and can display this summary information and how it changes over time (with the ability to select the measurement of “time” to be relative wall time since the beginning of the execution of the TensorFlow pro- gram, absolute time, or “steps”, a numeric measure of the number of graph executions that have occurred since the beginning of execution of the TensorFlow program). A screen shot of the visualization of summary values in TensorBoard is shown in Figure [11.](#_bookmark23)

# Performance Tracing

We also have an internal tool called EEG (not included in the initial open source release in November, 2015) that we use to collect and visualize very fine-grained informa- tion about the exact ordering and performance character-

istics of the execution of TensorFlow graphs. This tool works in both our single machine and distributed imple- mentations, and is very useful for understanding the bot- tlenecks in the computation and communication patterns of a TensorFlow program.

Traces are collected simultaneously on each machine in the system from a variety of sources including Linux kernel ftrace, our own lightweight thread tracing tools and the CUDA Profiling Tools Interface (CUPTI). With these logs we can reconstruct the execution of a dis- tributed training step with microsecond-level details of every thread-switch, CUDA kernel launch and DMA op- eration.

Traces are combined in a visualization server which is designed to rapidly extract events in a specified timerange and summarize at appropriate detail level for the user-interface resolution. Any significant delays due to communication, synchronization or DMA-related stalls are identified and highlighted using arrows in the visualization. Initially the UI provides an overview of the entire trace, with only the most significant performance artifacts highlighted. As the user progressively zooms in, increasingly fine resolution details are rendered.

Figure [12](#_bookmark26) shows an example EEG visualization of a model being trained on a multi-core CPU platform. The top third of the screenshot shows TensorFlow operations being dispatched in parallel, according to the dataflow constraints. The bottom section of the trace shows how most operations are decomposed into multiple work- items which are executed concurrently in a thread pool. The diagonal arrows on the right hand size show where queueing delay is building up in the thread pool. Fig- ure [13](#_bookmark27) shows another EEG visualization with compu- tation mainly happening on the GPU. Host threads can be seen enqueuing TensorFlow GPU operations as they become runnable (the light blue thread pool), and back- ground housekeeping threads can be seen in other col- ors being migrated across processor cores. Once again, arrows show where threads are stalled on GPU to CPU transfers, or where ops experience significant queueing delay.

Finally, Figure [14](#_bookmark28) shows a more detailed view which allows us to examine how Tensorflow GPU operators are assigned to multiple GPU streams. Whenever the dataflow graph allows parallel execution or data trans- fer we endeavour to expose the ordering constraints to the GPU device using streams and stream dependency primitives.

# Future Work

We have several different directions for future work. We will continue to use TensorFlow to develop new and in-

teresting machine learning models for artificial intelli- gence, and in the course of doing this, we may discover ways in which we will need to extend the basic Ten- sorFlow system. The open source community may also come up with new and interesting directions for the Ten- sorFlow implementation.

One extension to the basic programming model that we are considering is a function mechanism, whereby a user can specify an entire subgraph of a TensorFlow computation to be a reusable component. In the imple- mentation we have designed, these functions can become reusable components even across different front-end lan- guages for TensorFlow, so that a user could define a func- tion using the Python front end, but then use that func- tion as a basic building block from within the C++ front- end. We are hopeful that this cross-language reusability will bootstrap a vibrant community of machine learning researchers publishing not just whole examples of their research, but also small reusable components from their work that can be reused in other contexts.

We also have a number of concrete directions to im- prove the performance of TensorFlow. One such direc- tion is our initial work on a just-in-time compiler that can take a subgraph of a TensorFlow execution, perhaps with some runtime profiling information about the typi- cal sizes and shapes of tensors, and can generate an op- timized routine for this subgraph. This compiler will un- derstand the semantics of perform a number of optimiza- tions such as loop fusion, blocking and tiling for locality, specialization for particular shapes and sizes, etc.

We also imagine that a significant area for future work will be in improving the placement and node scheduling algorithms used to decide where different nodes will exe- cute, and when they should start executing. We have cur- rently implemented a number of heuristics in these sub- systems, and we’d like to have the system instead learn to make good placement decisions (perhaps using a deep neural network, combined with a reinforcement learning objective function).

# Related Work

There are many other systems that are comparable in various ways with TensorFlow. Theano [[7],](#_bookmark36) Torch [[13],](#_bookmark42) Caffe [[26],](#_bookmark55) Chainer [[49]](#_bookmark78) and the Computational Network Toolkit [[54]](#_bookmark83) are a few systems designed primarily for the training of neural networks. Each of these systems maps the computation onto a single machine, unlike the dis- tributed TensorFlow implementation. Like Theano and Chainer, TensorFlow supports symbolic differentiation, thus making it easier to define and work with gradient- based optimization algorithms. Like Caffe, TensorFlow has a core written in C++, simplifying the deployment

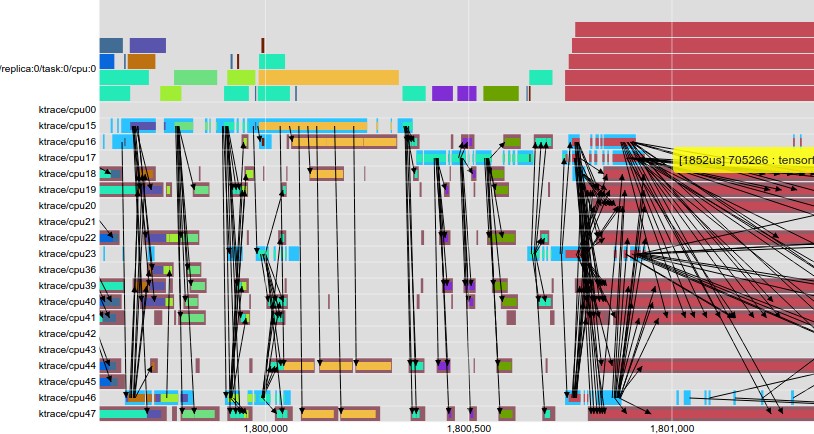


Figure 12: EEG visualization of multi-threaded CPU operations (x-axis is time in *µ*s).

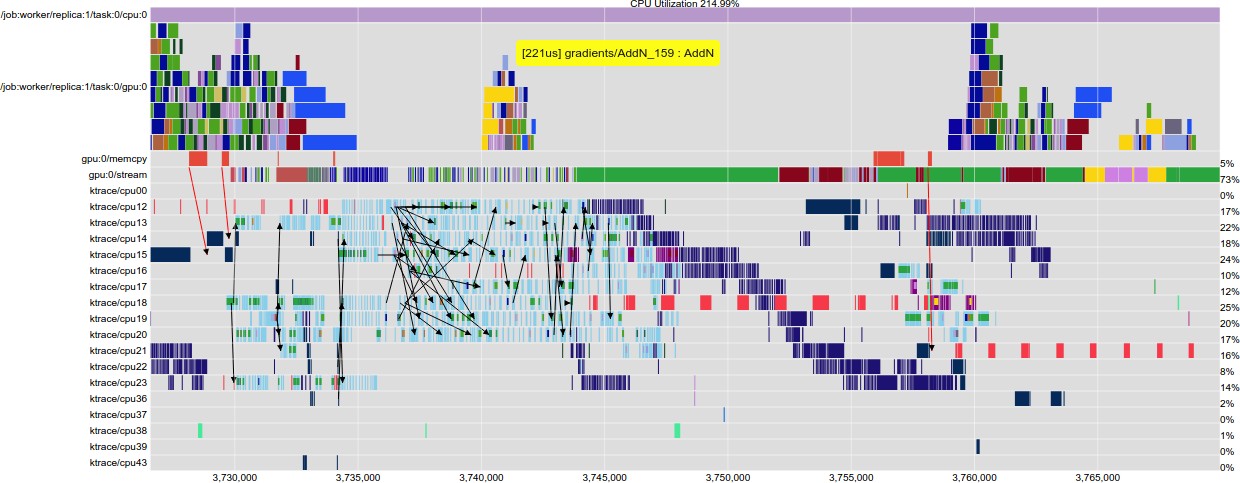


Figure 13: EEG visualization of Inception training showing CPU and GPU activity.

of trained models in a wide variety of production set- tings, including memory- and computation-constrained environments such as mobile devices.

The TensorFlow system shares some design charac- teristics with its predecessor system, DistBelief [[14],](#_bookmark43) and with later systems with similar designs like Project Adam [[10]](#_bookmark39) and the Parameter Server project [[33].](#_bookmark62) Like DistBelief and Project Adam, TensorFlow allows com- putations to be spread out across many computational de- vices across many machines, and allows users to specify

machine learning models using relatively high-level de- scriptions. Unlike DistBelief and Project Adam, though, the general-purpose dataflow graph model in TensorFlow is more flexible and more amenable to expressing a wider variety of machine learning models and optimization al- gorithms. It also permits a significant simplification by allowing the expression of stateful parameter nodes as variables, and variable update operations that are just additional nodes in the graph; in contrast, DistBelief, Project Adam and the Parameter Server systems all have

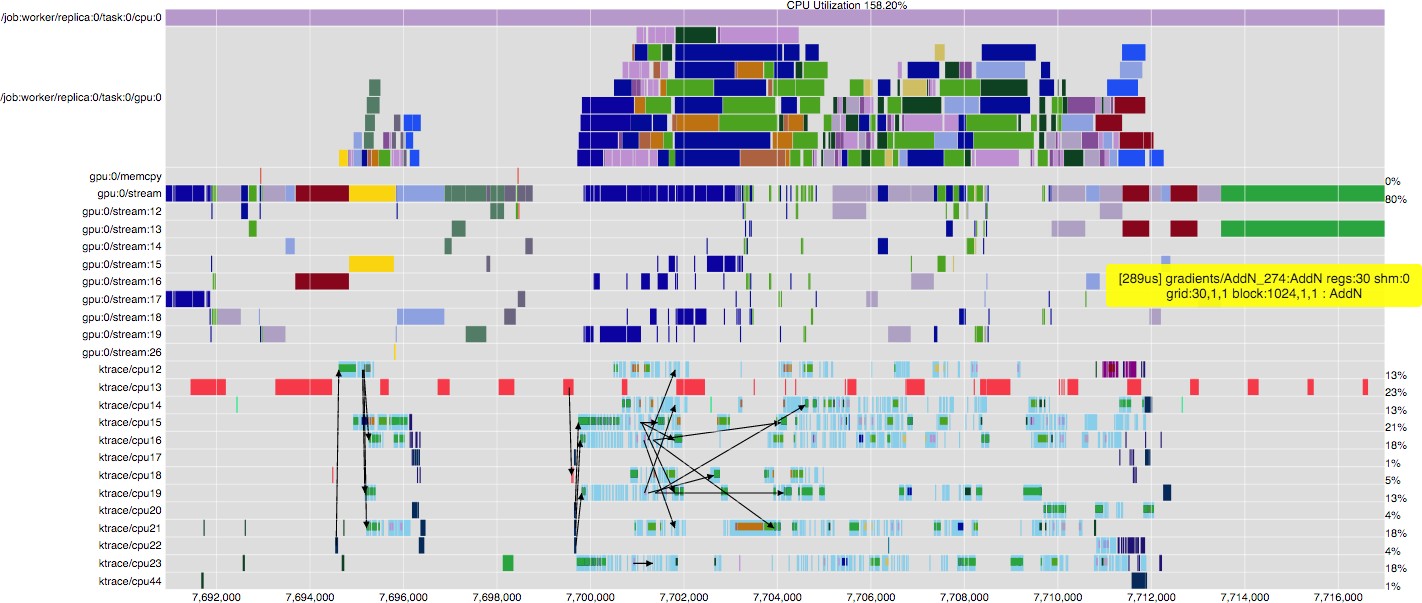


Figure 14: Timeline of multi-stream GPU execution.

whole separate parameter server subsystems devoted to communicating and updating parameter values.

The Halide system [[40]](#_bookmark69) for expressing image pro- cessing pipelines uses a similar intermediate represen- tation to the TensorFlow dataflow graph. Unlike Ten- sorFlow, though, the Halide system actually has higher- level knowledge of the semantics of its operations and uses this knowledge to generate highly optimized pieces of code that combine multiple operations, taking into ac- count parallelism and locality. Halide runs the resulting computations only on a single machine, and not in a dis- tributed setting. In future work we are hoping to extend TensorFlow with a similar cross-operation dynamic com- pilation framework.

Like TensorFlow, several other distributed systems have been developed for executing dataflow graphs across a cluster. Dryad [[24]](#_bookmark53) and Flume [[8]](#_bookmark37) demon- strate how a complex workflow can be represented as a dataflow graph. CIEL [[37]](#_bookmark66) and Naiad [[36]](#_bookmark65) introduce generic support for data-dependent control flow: CIEL represents iteration as a DAG that dynamically unfolds, whereas Naiad uses a static graph with cycles to support lower-latency iteration. Spark [[55]](#_bookmark84) is optimized for com- putations that access the same data repeatedly, using “re- silient distributed datasets” (RDDs), which are soft-state cached outputs of earlier computations. Dandelion [[44]](#_bookmark73) executes dataflow graphs across a cluster of heteroge- neous devices, including GPUs. TensorFlow uses a hy- brid dataflow model that borrows elements from each of these systems. Its dataflow scheduler, which is the component that chooses the next node to execute, uses the same basic algorithm as Dryad, Flume, CIEL, and Spark. Its distributed architecture is closest to Naiad, in

that the system uses a single, optimized dataflow graph to represent the entire computation, and caches information about that graph on each device to minimize coordination overhead. Like Spark and Naiad, TensorFlow works best when there is sufficient RAM in the cluster to hold the working set of the computation. Iteration in TensorFlow uses a hybrid approach: multiple replicas of the same dataflow graph may be executing at once, while sharing the same set of variables. Replicas can share data asyn- chronously through the variables, or use synchronization mechanisms in the graph, such as queues, to operate syn- chronously. TensorFlow also supports iteration within a graph, which is a hybrid of CIEL and Naiad: for simplic- ity, each node fires only when all of its inputs are ready (like CIEL); but for efficiency the graph is represented as a static, cyclic dataflow (like Naiad).

# Conclusions

We have described TensorFlow, a flexible data flow- based programming model, as well as single machine and distributed implementations of this programming model. The system is borne from real-world experience in conducting research and deploying more than one hun- dred machine learning projects throughout a wide range of Google products and services. We have open sourced a version of TensorFlow, and hope that a vibrant shared community develops around the use of TensorFlow. We are excited to see how others outside of Google make use of TensorFlow in their own work.

# Acknowledgements

The development of TensorFlow has benefitted enor- mously from the large and broad machine learning com- munity at Google, and in particular from the suggestions and contributions from rest of the Google Brain team and also from the hundreds of DistBelief and TensorFlow users within Google. Without a doubt, the usability and functionality of TensorFlow has been greatly expanded by listening to their feedback.

Many individuals have contributed to TensorFlow and to its open source release, including John Gian- nandrea (for creating a supportive research environ- ment), Irina Kofman and Phing Turner (project manage- ment), Bill Gruber and David Westbrook (technical writ- ing), Dave Andersen, Anelia Angelova, Yaroslav Bu- latov, Jianmin Chen, Jerjou Cheng, George Dahl, An- drew Dai, Lucy Gao, mig Gerard, Stephan Gouws, Naveen Kumar, Geoffrey Hinton, Mrinal Kalarishnan, Anjuli Kannan, Yutaka Leon-Suematsu, Frank Li, Pe- ter Liu, Xiaobing Liu, Nishant Patil, Pierre Sermanet, Noam Shazeer, Jascha Sohl-dickstein, Philip Tucker, Yonghui Wu, Ke Yang, and Cliff Young (general con- tributions), Doug Fritz, Patrick Hurst, Dilip Krish- nan, Daniel Smilkov, James Wexler, Jimbo Wilson, Kanit Ham Wongsuphasawat, Cassandra Xia, and the Big Picture team (graph visualization), Chris Leary, Robert Springer and the Stream Executor team, Kayur Patel, Michael Piatek, and the coLab team, and the many others who have contributed to the TensorFlow design and code base.

# References

1. Mart´ın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghe- mawat, Ian Goodfellow, Andrew Harp, Geoffrey Irv- ing, 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 Vie´gas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Soft- ware available from [tensorflow.org](http://tensorflow.org/).
2. Anelia Angelova, Alex Krizhevsky, and Vincent Van- houcke. Pedestrian detection with a large-field-of-view deep network. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 704–711. IEEE, 2015. [CalTech PDF](http://www.vision.caltech.edu/anelia/publications/Angelova15LFOV.pdf).
3. Arvind and David E. Culler. Annual review of computer science vol. 1, 1986. chapter

Dataflow Architectures, pages 225–253. 1986. [www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&](http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&amp;doc=GetTRDoc.pdf&amp;AD=ADA166235) [doc=GetTRDoc.pdf&AD=ADA166235](http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&amp;doc=GetTRDoc.pdf&amp;AD=ADA166235).

1. Arvind and Rishiyur S. Nikhil. Executing a pro- gram on the MIT tagged-token dataflow architec- ture. *IEEE Trans. Comput.*, 39(3):300–318, 1990. [dl.acm.org/citation.cfm?id=78583](http://dl.acm.org/citation.cfm?id=78583).
2. Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual atten- tion. *arXiv preprint arXiv:1412.7755*, 2014. [arxiv.org/abs/1412.7755](http://arxiv.org/abs/1412.7755).
3. Franc¸oise Beaufays. The neural networks behind Google Voice transcription, 2015.

[googleresearch.blogspot.com/2015/08/the-neural-](http://googleresearch.blogspot.com/2015/08/the-neural-networks-behind-google-voice.html) [networks-behind-google-voice.html](http://googleresearch.blogspot.com/2015/08/the-neural-networks-behind-google-voice.html).

1. James Bergstra, Olivier Breuleux, Fre´de´ric Bastien, Pas- cal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing con- ference (SciPy)*, volume 4, page 3. Austin, TX, 2010. [UMontreal PDF](http://www-etud.iro.umontreal.ca/%7Ewardefar/publications/theano_scipy2010.pdf).
2. Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, effi- cient data-parallel pipelines. In *ACM Sigplan No- tices*, volume 45, pages 363–375. ACM, 2010. [re- search.google.com/pubs/archive/35650.pdf](https://research.google.com/pubs/archive/35650.pdf).
3. Sharan Chetlur, Cliff Woolley, Philippe Vandermer- sch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. [arxiv.org/abs/1410.0759](https://arxiv.org/abs/1410.0759).
4. Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014. [www.usenix.org/system/files/conference/osdi14/osdi14- paper-chilimbi.pdf](https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-chilimbi.pdf).
5. Jack Clark. Google turning its lucrative web search over to AI machines, 2015.

[www.bloomberg.com/news/articles/2015-10-26/google-](http://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-ai-machines) [turning-its-lucrative-web-search-over-to-ai-machines](http://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-ai-machines).

1. Cliff Click. Global code motion/global value number- ing. In *ACM SIGPLAN Notices*, volume 30, pages 246–

257. ACM, 1995. [courses.cs.washington.edu/courses/ cse501/06wi/reading/click-pldi95.pdf](https://courses.cs.washington.edu/courses/cse501/06wi/reading/click-pldi95.pdf).

1. Ronan Collobert, Samy Bengio, and Johnny Marie´thoz. Torch: A modular machine learning software library. Technical report, IDIAP, 2002.

[infoscience.epfl.ch/record/82802/files/rr02-46.pdf](http://infoscience.epfl.ch/record/82802/files/rr02-46.pdf).

1. Jeffrey Dean, Gregory S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew Senior, Paul Tucker,

Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012. [Google Research PDF](http://research.google.com/archive/large_deep_networks_nips2012.pdf).

1. Jack J Dongarra, Jeremy Du Croz, Sven Hammar- ling, and Iain S Duff. A set of level 3 basic lin- ear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990. [www.maths.manchester.ac.uk/˜sven/pubs/Level3BLAS- 1-TOMS16-90.pdf](https://www.maths.manchester.ac.uk/%7Esven/pubs/Level3BLAS-1-TOMS16-90.pdf).
2. Andrea Frome, Greg S Corrado, Jonathon Shlens, Samy Bengio, Jeff Dean, Tomas Mikolov, et al. DeVISE: A deep visual-semantic embedding model. In *Advances in Neural Information Pro- cessing Systems*, pages 2121–2129, 2013. [re- search.google.com/pubs/archive/41473.pdf](http://research.google.com/pubs/archive/41473.pdf).
3. Javier Gonzalez-Dominguez, Ignacio Lopez-Moreno, Pe- dro J Moreno, and Joaquin Gonzalez-Rodriguez. Frame- by-frame language identification in short utterances using deep neural networks. *Neural Networks*, 64:49–58, 2015.
4. Otavio Good. How Google Translate squeezes deep learning onto a phone, 2015. [googleresearch.blogspot.com/2015/07/how-google- translate-squeezes-deep.html](http://googleresearch.blogspot.com/2015/07/how-google-translate-squeezes-deep.html).
5. Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from Street View imagery using deep convolutional neu- ral networks. In *International Conference on Learning Representations*, 2014. [arxiv.org/pdf/1312.6082](http://arxiv.org/pdf/1312.6082).
6. Georg Heigold, Vincent Vanhoucke, Alan Senior, Patrick Nguyen, Marc’Aurelio Ranzato, Matthieu Devin, and Jeffrey Dean. Multilingual acoustic models using dis- tributed deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE Interna- tional Conference on*, pages 8619–8623. IEEE, 2013. [re- search.google.com/pubs/archive/40807.pdf](https://research.google.com/pubs/archive/40807.pdf).
7. Geoffrey E. Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, An- drew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, 29(6):82– 97, 2012. [www.cs.toronto.edu/˜gdahl/papers/ deepSpeechReviewSPM2012.pdf](http://www.cs.toronto.edu/%7Egdahl/papers/deepSpeechReviewSPM2012.pdf).
8. Sepp Hochreiter and Ju¨rgen Schmidhuber. Long short- term memory. *Neural computation*, 9(8):1735–1780, 1997. [ftp.idsia.ch/pub/juergen/lstm.pdf](http://ftp.idsia.ch/pub/juergen/lstm.pdf).
9. Sergey Ioffe and Christian Szegedy. Batch normaliza- tion: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. [arxiv.org/abs/1502.03167](http://arxiv.org/abs/1502.03167).
10. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007. [www.michaelisard.com/pubs/eurosys07.pdf](http://www.michaelisard.com/pubs/eurosys07.pdf).
11. Benoˆıt Jacob, Gae¨l Guennebaud, et al. Eigen library for linear algebra. [eigen.tuxfamily.org](http://eigen.tuxfamily.org/).
12. Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadar- rama, and Trevor Darrell. Caffe: Convolutional archi- tecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014. [arxiv.org/pdf/1408.5093](http://arxiv.org/pdf/1408.5093).
13. Andrej Karpathy, George Toderici, Sachin Shetty, Tommy Leung, Rahul Sukthankar, and Li Fei- Fei. Large-scale video classification with con- volutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Con- ference on*, pages 1725–1732. IEEE, 2014. [re- search.google.com/pubs/archive/42455.pdf](https://research.google.com/pubs/archive/42455.pdf).
14. A Krizhevsky. Cuda-convnet, 2014.

[code.google.com/p/cuda-convnet/](https://code.google.com/p/cuda-convnet/).

1. Alex Krizhevsky. One weird trick for paralleliz- ing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014. [arxiv.org/abs/1404.5997](http://arxiv.org/abs/1404.5997).
2. Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. [www.cs.toronto.edu/˜kriz/cifar.html](http://www.cs.toronto.edu/%7Ekriz/cifar.html).
3. Quoc Le, Marc’Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsu- pervised learning. In *ICML’2012*, 2012. [Google Research PDF](http://research.google.com/archive/unsupervised_icml2012.pdf).
4. Yann LeCun, Corinna Cortes, and Christopher JC Burges. The MNIST database of handwritten digits, 1998. [yann.lecun.com/exdb/mnist/](http://yann.lecun.com/exdb/mnist/).
5. Mu Li, Dave Andersen, and Alex Smola. Parameter server. [parameterserver.org](http://parameterserver.org/).
6. Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in Go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*, 2014. [arxiv.org/abs/1412.6564](http://arxiv.org/abs/1412.6564).
7. Tomas Mikolov, Kai Chen, Greg Corrado, and Jef- frey Dean. Efficient estimation of word representa- tions in vector space. In *International Conference on Learning Representations: Workshops Track*, 2013. [arxiv.org/abs/1301.3781](http://arxiv.org/abs/1301.3781).
8. Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Mart´ın Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty- Fourth ACM Symposium on Operating Systems Princi- ples*, pages 439–455. ACM, 2013. Microsoft Research PDF.
9. Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smit, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for dis- tributed data-flow computing. In *Proceedings of the Ninth USENIX Symposium on Networked Systems Design and Implementation*, 2011. [Usenix PDF](https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Murray.pdf).
10. Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Ve- davyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel meth- ods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015. arxiv.org/abs/1507.04296.
11. CUDA Nvidia. Cublas library. *NVIDIA Corpo- ration, Santa Clara, California*, 15, 2008. [devel- oper.nvidia.com/cublas](https://developer.nvidia.com/cublas).
12. Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fre´do Durand, and Saman Ama- rasinghe. Halide: A language and compiler for optimiz- ing parallelism, locality, and recomputation in image pro- cessing pipelines. *ACM SIGPLAN Notices*, 48(6):519– 530, 2013. [people.csail.mit.edu/fredo/tmp/Halide- 5min.pdf](http://people.csail.mit.edu/fredo/tmp/Halide-5min.pdf).
13. Bharath Ramsundar, Steven Kearnes, Patrick Riley, Dale Webster, David Konerding, and Vijay Pande. Massively multitask networks for drug discovery. *arXiv preprint arXiv:1502.02072*, 2015. [arxiv.org/abs/1502.02072](http://arxiv.org/abs/1502.02072).
14. Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to paral- lelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011. [papers.nips.cc/paper/4390-hogwild-a-lock-free- approach-to-parallelizing-stochastic-gradient-descent](http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent).
15. Chuck Rosenberg. Improving Photo Search: A step across the semantic gap, 2013. [googleresearch.blogspot.com/2013/06/improving-](http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html)

[photo-search-step-across.html](http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html).

1. Christopher J Rossbach, Yuan Yu, Jon Currey, Jean- Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013. [research-srv.microsoft.com/pubs/201110/sosp13- dandelion-final.pdf](http://research-srv.microsoft.com/pubs/201110/sosp13-dandelion-final.pdf).
2. David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back- propagating errors. *Cognitive modeling*, 5:3, 1988. [www.cs.toronto.edu/ hinton/absps/naturebp.pdf](http://www.cs.toronto.edu/%7Ehinton/absps/naturebp.pdf).
3. Has¸im Sak, Andrew Senior, Kanishka Rao, Franc¸oise Beaufays, and Johan Schalkwyk. Google Voice Search: faster and more accurate, 2015. [googleresearch.blogspot.com/2015/09/google-voice-](http://googleresearch.blogspot.com/2015/09/google-voice-search-faster-and-more.html) [search-faster-and-more.html](http://googleresearch.blogspot.com/2015/09/google-voice-search-faster-and-more.html).
4. Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014. [papers.nips.cc/paper/5346-sequence-to-sequence- learning-with-neural](http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural).
5. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Ser- manet, Scott Reed, Dragomir Anguelov, Dumitru Er- han, Vincent Vanhoucke, and Andrew Rabinovich. Go- ing deeper with convolutions. In *CVPR’2015*, 2015. [arxiv.org/abs/1409.4842](http://arxiv.org/abs/1409.4842).
6. Seiya Tokui. Chainer: A powerful, flexible and intuitive framework of neural networks. [chainer.org](http://chainer.org/).
7. Vincent Vanhoucke. Speech recognition and deep learn- ing, 2015. [googleresearch.blogspot.com/2012/08/speech- recognition-and-deep-learning.html](http://googleresearch.blogspot.com/2012/08/speech-recognition-and-deep-learning.html).
8. Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015. [re- search.google.com/pubs/archive/43438.pdf](http://research.google.com/pubs/archive/43438.pdf).
9. O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and

G. Hinton. Grammar as a foreign language. Technical report, arXiv:1412.7449, 2014. [arxiv.org/abs/1412.7449](http://arxiv.org/abs/1412.7449).

1. Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *NIPS*, 2015. [arxiv.org/abs/1506.03134](http://arxiv.org/abs/1506.03134).
2. Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the com- putational network toolkit. Technical report, Tech. Rep. MSR, Microsoft Research, 2014, 2014. [re- search.microsoft.com/apps/pubs/?id=226641](http://research.microsoft.com/apps/pubs/?id=226641).
3. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems De- sign and Implementation*. USENIX Association, 2012. [www.usenix.org/system/files/conference/nsdi12/nsdi12- final138.pdf](https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf).
4. Matthew D. Zeiler, Marc’Aurelio Ranzato, Rajat Monga, Mark Mao, Ke Yang, Quoc Le, Patrick Nguyen, Andrew Senior, Vincent Vanhoucke, Jeff Dean, and Geoffrey E. Hinton. On rectified linear units for speech processing. In *ICASSP*, 2013. [re- search.google.com/pubs/archive/40811.pdf](http://research.google.com/pubs/archive/40811.pdf).