TensorFlow: A System for Large-Scale Machine Learning

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *Google Brain*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

TensorFlow: A system for large-scale machine learning

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng

Google Brain

Abstract

TensorFlow is a machine learning system that operates at large scale and in heterogeneous environments. TensorFlow uses dataflow graphs to represent computation, shared state, and the operations that mutate that state. It maps the nodes of a dataflow graph across many machines in a cluster, and within a machine across multiple computational devices, including multicore CPUs, general-purpose GPUs, and custom-designed ASICs known as Tensor Processing Units (TPUs). This architecture gives flexibility to the application developer: whereas in previous "parameter server" designs the management of shared state is built into the system, TensorFlow enables developers to experiment with novel optimizations and training algorithms. TensorFlow supports a variety of applications, with a focus on training and inference on deep neural networks. Several Google services use TensorFlow in production, we have released it as an open-source project, and it has become widely used for machine learning research. In this paper, we describe the TensorFlow dataflow model and demonstrate the compelling performance that TensorFlow achieves for several real-world applications.

1 Introduction

In recent years, machine learning has driven advances in many different fields [3, 5, 24, 25, 29, 31, 42, 47, 50, 52, 57, 67, 68, 72, 76]. We attribute this success to the invention of more sophisticated machine learning models [44, 54], the availability of large datasets for tackling problems in these fields [9, 64], and the development of software platforms that enable the easy use of large amounts of computational resources for training such models on these large datasets [14, 20].

We have developed the TensorFlow system for experimenting with new models, training them on large datasets, and moving them into production. We have based TensorFlow on many years of experience with our first-generation system, DistBelief [20], both simplifying and generalizing it to enable researchers to explore a wider variety of ideas with relative ease. TensorFlow supports both large-scale training and inference: it efficiently uses hundreds of powerful (GPU-enabled) servers for fast training, and it runs trained models for inference in production on various platforms, ranging from large distributed clusters in a datacenter, down to running locally on mobile devices. At the same time, it is flexible enough to support experimentation and research into new machine learning models and system-level optimizations.

TensorFlow uses a unified dataflow graph to represent both the computation in an algorithm *and* the state on which the algorithm operates. We draw inspiration from the high-level programming models of dataflow systems [2, 21, 34] and the low-level efficiency of *parameter servers* [14, 20, 49]. Unlike traditional dataflow systems, in which graph vertices represent functional computation on immutable data, TensorFlow allows vertices to represent computations that own or update mutable state. Edges carry *tensors* (multi-dimensional arrays) between nodes, and TensorFlow transparently inserts the appropriate communication between distributed subcomputations. By unifying the computation and state management in a single programming model, TensorFlow allows programmers to experiment with different parallelization schemes that, for example, offload computation onto the servers that hold the shared state to reduce the amount of network traffic. We have also built various coordination protocols, and achieved encouraging results with synchronous replication, echoing recent results [10, 18] that contradict the commonly held belief that asynchronous replication is required for scalable learning [14, 20, 49].

Over the past year, more than 150 teams at Google have used TensorFlow, and we have released the system as an

open-source project.[1] Thanks to our large community of users we have gained experience with many different machine learning applications. In this paper, we focus on neural network training as a challenging systems problem, and select two representative applications from this space: image classification and language modeling. These applications stress computational throughput and aggregate model size respectively, and we use them both to demonstrate the extensibility of TensorFlow, and to evaluate the efficiency and scalability of our present implementation.

2 Background & motivation

We begin by describing the limitations of our previous system (§2.1) and outlining the design principles that we used in the development of TensorFlow (§2.2).

2.1 Previous system: DistBelief

TensorFlow is the successor to DistBelief, which is the distributed system for training neural networks that Google has used since 2011 [20]. DistBelief uses the *parameter server* architecture, and here we criticize its limitations, but other systems based on this architecture have addressed these limitations in other ways [11, 14, 49]; we discuss those systems in Subsection 2.3.

In the parameter server architecture, a job comprises two disjoint sets of processes: stateless *worker* processes that perform the bulk of the computation when training a model, and stateful *parameter server* processes that maintain the current version of the model parameters. DistBelief's programming model is similar to Caffe's [38]: the user defines a neural network as a directed acyclic graph of *layers* that terminates with a *loss function*. A layer is a composition of mathematical operators: for example, a *fully connected* layer multiplies its input by a weight matrix, adds a bias vector, and applies a non-linear function (such as a sigmoid) to the result. A loss function is a scalar function that quantifies the difference between the predicted value (for a given input data point) and the ground truth. In a fully connected layer, the weight matrix and bias vector are *parameters*, which a learning algorithm will update in order to minimize the value of the loss function. DistBelief uses the DAG structure and knowledge of the layers' semantics to compute gradients for each of the model parameters, via backpropagation [63]. Because the parameter updates in many algorithms are commutative and have weak consistency requirements [61], the worker processes can compute updates independently and write back "delta" updates to each parameter server, which combines the updates with its current state.

Although DistBelief has enabled many Google products to use deep neural networks and formed the basis of many machine learning research projects, we soon began to feel its limitations. Its Python-based scripting interface for composing pre-defined layers was adequate for users with simple requirements, but our more advanced users sought three further kinds of flexibility:

Defining new layers For efficiency, we implemented DistBelief layers as C++ classes. Using a separate, less familiar programming language for implementing layers is a barrier for machine learning researchers who seek to experiment with new layer architectures, such as sampled softmax classifiers [37] and attention modules [53].

Refining the training algorithms Many neural networks are trained using stochastic gradient descent (SGD), which iteratively refines the parameters of the network by moving them in the direction that maximally decreases the value of the loss function. Several refinements to SGD accelerate convergence by changing the update rule [23, 66]. Researchers often want to experiment with new optimization methods, but doing that in DistBelief involves modifying the parameter server implementation. Moreover, the `get()` and `put()` interface for the parameter server is not ideal for all optimization methods: sometimes a set of related parameters must be updated atomically, and in many cases it would be more efficient to offload computation onto the parameter server, and thereby reduce the amount of network traffic.

Defining new training algorithms DistBelief workers follow a fixed execution pattern: read a batch of input data and the current parameter values, compute the loss function (a *forward* pass through the network), compute gradients for each of the parameter (a *backward* pass), and write the gradients back to the parameter server. This pattern works for training simple feed-forward neural networks, but fails for more advanced models, such as recurrent neural networks, which contain loops [39]; adversarial networks, in which two related networks are trained alternately [26]; and reinforcement learning models, where the loss function is computed by some agent in a separate system, such as a video game emulator [54]. Moreover, there are many other machine learning algorithms—such as expectation maximization, decision forest training, and latent Dirichlet allocation—that do not fit the same mold as neural network training, but could also benefit from a common, well-optimized distributed runtime.

In addition, we designed DistBelief with a single platform in mind: a large distributed cluster of multicore

[1] Software available from `https://tensorflow.org`.

```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```

Figure 1: An image classifier written using TensorFlow's Python API. This program is a simple solution to the MNIST digit classification problem [48], with 784-pixel images and 10 output classes.

servers [20]. We were able to add support for GPU acceleration, when it became clear that this acceleration would be crucial for executing convolutional kernels efficiently [44], but DistBelief remains a heavyweight system that is geared for training deep neural networks on huge datasets, and is difficult to scale *down* to other environments. In particular, many users want to hone their model locally on a GPU-powered workstation, before scaling the same code to train on a much larger dataset. After training a model on a cluster, the next step is to push the model into production, which might involve integrating the model into an online service, or deploying it onto a mobile device for offline execution. Each of these tasks has some common computational structure, but our colleagues found it necessary to use or create separate systems that satisfy the different performance and resource requirements of each platform. TensorFlow provides a single programming model and runtime system for all of these environments.

2.2 Design principles

We designed TensorFlow to be much more flexible than DistBelief, while retaining its ability to satisfy the demands of Google's production machine learning workloads. TensorFlow provides a simple dataflow-based programming abstraction that allows users to deploy applications on distributed clusters, local workstations, mobile devices, and custom-designed accelerators. A high-level scripting interface (Figure 1) wraps the construction of dataflow graphs and enables users to experiment with different model architectures and optimization algorithms without modifying the core system. In this subsection, we briefly highlight TensorFlow's core design principles:

Dataflow graphs of primitive operators Both TensorFlow and DistBelief use a dataflow representation for their models, but the most striking difference is that a DistBelief model comprises relatively few complex "layers", whereas the corresponding TensorFlow model represents individual mathematical operators (such as matrix multiplication, convolution, etc.) as nodes in the dataflow graph. This approach makes it easier for users to compose novel layers using a high-level scripting interface. Many optimization algorithms require each layer to have defined gradients, and building layers out of simple operators makes it easy to differentiate these models automatically (§4.1). In addition to the functional operators, we represent mutable state, and the operations that update it, as nodes in the dataflow graph, thus enabling experimentation with different update rules.

Deferred execution A typical TensorFlow application has two distinct phases: the first phase defines the program (e.g., a neural network to be trained and the update rules) as a symbolic dataflow graph with placeholders for

the input data and variables that represent the state; and the second phase executes an optimized version of the program on the set of available devices. By deferring the execution until the entire program is available, TensorFlow can optimize the execution phase by using global information about the computation. For example, TensorFlow achieves high GPU utilization by using the graph's dependency structure to issue a sequence of kernels to the GPU without waiting for intermediate results. While this design choice makes execution more efficient, we have had to push more complex features—such as dynamic control flow (§3.4)—into the dataflow graph, so that models using these features enjoy the same optimizations.

Common abstraction for heterogeneous accelerators In addition to general-purpose devices such as multicore CPUs and GPUs, special-purpose accelerators for deep learning can achieve significant performance improvements and power savings. At Google, our colleagues have built the Tensor Processing Unit (TPU) specifically for machine learning; TPUs yield an order of magnitude improvement in performance-per-watt compared to alternative state-of-the-art technology [40]. To support these accelerators in TensorFlow, we define a common abstraction for devices. At a minimum, a device must implement methods for (i) issuing a kernel for execution, (ii) allocating memory for inputs and outputs, and (iii) transferring buffers to and from host memory. Each operator (e.g., matrix multiplication) can have multiple specialized implementations for different devices. As a result, the same program can easily target GPUs, TPUs, or mobile CPUs as required for training, serving, and offline inference.

TensorFlow uses tensors of primitive values as a common interchange format that all devices understand. At the lowest level, all tensors in TensorFlow are dense; sparse tensors can be represented in terms of dense ones (§3.1). This decision ensures that the lowest levels of the system have simple implementations for memory allocation and serialization, thus reducing the framework overhead. Tensors also enable other optimizations for memory management and communication, such as RDMA and direct GPU-to-GPU transfer.

The main consequence of these principles is that in TensorFlow there is no such thing as a parameter server. On a cluster, we deploy TensorFlow as a set of *tasks* (named processes that can communicate over a network) that each export the same graph execution API and contain one or more devices. Typically a subset of those tasks assumes the role that a parameter server plays in other systems [11, 14, 20, 49], and we therefore call them *PS tasks*; the others are *worker tasks*. However, since a PS task is capable of running arbitrary TensorFlow graphs, it is more flexible than a conventional parameter server: users can program it with the same scripting interface that they use to define models. This flexibility is the key difference between TensorFlow and contemporary systems, and in the rest of the paper we will discuss some of the applications that this flexibility enables.

2.3 Related work

Single-machine frameworks Many machine learning researchers carry out their work on a single—often GPU-equipped—computer [43, 44], and several single-machine frameworks support this scenario. Caffe [38] is a high-performance framework for training declaratively specified neural networks on multicore CPUs and GPUs. As discussed above, its programming model is similar to DistBelief (§2.1), so it is easy to compose models from existing layers, but relatively difficult to add new layers or optimizers. Theano [2] allows programmers to express a model as a dataflow graph of primitive operators, and generates efficient compiled code for training that model. Its programming model is closest to TensorFlow, and it provides much of the same flexibility in a single machine.

Unlike Caffe, Theano, and TensorFlow, Torch [17] offers a powerful *imperative* programming model for scientific computation and machine learning. It allows fine-grained control over the execution order and memory utilization, which enables power users to optimize the performance of their programs. While this flexibility is useful for research, Torch lacks the advantages of a dataflow graph as a portable representation across small-scale experimentation, production training, and deployment.

Batch dataflow systems Starting with MapReduce [21], batch dataflow systems have been applied to a large number of machine learning algorithms [70], and more recent systems have focused on increasing expressivity and performance. DryadLINQ [74] adds a high-level query language that supports more sophisticated algorithms than MapReduce. Spark [75] extends DryadLINQ with the ability to cache previously computed datasets in memory, and is therefore better suited to iterative machine learning algorithms (such as k-means clustering and logistic regression) when the input data fit in memory. Dandelion extends DryadLINQ with code generation for GPUs [62] and FPGAs [16].

The principal limitation of a batch dataflow system is that it requires the input data to be immutable, and all of the subcomputations to be deterministic, so that the system can re-execute subcomputations when machines in the cluster fail. This feature—which is beneficial for many conventional workloads—makes updating a ma-

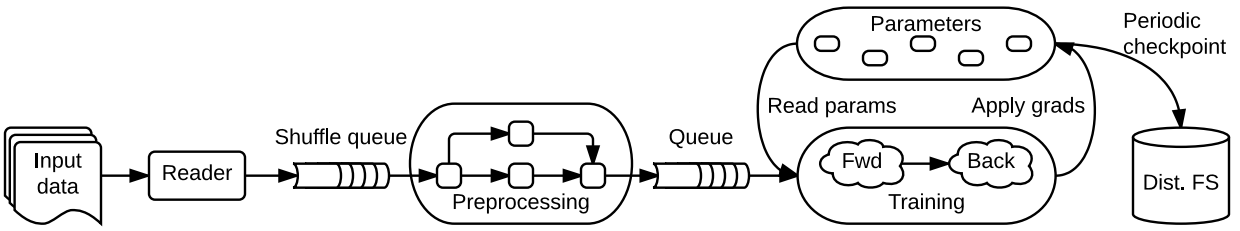

Figure 2: A schematic TensorFlow dataflow graph for a training pipeline, containing subgraphs for reading input data, preprocessing, training, and checkpointing state.

chine learning model an expensive operation. For example, the SparkNet system for training deep neural networks on Spark takes 20 seconds to broadcast weights and collect updates from five workers [55]. As a result, in these systems, each model update step must process larger batches, slowing convergence [8]. We show in Subsection 6.3 that TensorFlow can train larger models on larger clusters with step times as short as 2 seconds.

Parameter servers As we discuss in Subsection 2.1, a parameter server architecture uses a set of servers to manage shared state that is updated by a set of parallel workers. This architecture emerged in work on scalable topic modeling [65], and DistBelief showed how it can apply to deep neural network training. Project Adam [14] further applied this architecture for the efficient training of convolutional neural networks; and Li *et al.*'s "Parameter Server" [49] added innovations in consistency models, fault tolerance, and elastic rescaling. Despite earlier skepticism that parameter servers would be compatible with GPU acceleration [14], Cui *et al.* recently showed that a parameter server specialized for use with GPUs can achieve speedups on small clusters [18].

MXNet [11] is perhaps the closest system in design to TensorFlow. It uses a dataflow graph to represent the computation at each worker, and uses a parameter server to scale training across multiple machines. The MXNet parameter server exports a key-value store interface that supports aggregating updates sent from multiple devices in each worker, and using an arbitrary user-provided function to combine incoming updates with the current value. The MXNet key-value store interface [22] does not currently allow sparse gradient updates within a single value, which are crucial for the distributed training of large models (§4.2), and adding this feature would require modifications to the core system.

The parameter server architecture meets many of our requirements, and with sufficient engineering effort it would be possible to build most of the features that we describe in this paper into a parameter server. For TensorFlow we sought a high-level programming model that allows users to customize the code that runs in all parts of the system, so that the cost of experimentation with new optimization algorithms and model architectures is lower. In the next section, we describe the building blocks of a TensorFlow program in more detail.

3 TensorFlow execution model

TensorFlow uses a single dataflow graph to represent all computation and state in a machine learning algorithm, including the individual mathematical operations, the parameters and their update rules, and the input preprocessing (Figure 2). The dataflow graph expresses the communication between subcomputations explicitly, thus making it easy to execute independent computations in parallel and to partition computations across multiple devices. TensorFlow differs from batch dataflow systems (§2.3) in two respects:

- The model supports multiple concurrent executions on overlapping subgraphs of the overall graph.
- Individual vertices may have mutable state that can be shared between different executions of the graph.

The key observation in the parameter server architecture [14, 20, 49] is that mutable state is crucial when training very large models, because it becomes possible to make in-place updates to very large parameters, and propagate those updates to parallel training steps as quickly as possible. Dataflow with mutable state enables TensorFlow to mimic the functionality of a parameter server, but with additional flexibility, because it becomes possible to execute arbitrary dataflow subgraphs on the machines that host the shared model parameters. As a result, our users have been able to experiment with different optimization algorithms, consistency schemes, and parallelization strategies.

3.1 Dataflow graph elements

In a TensorFlow graph, each vertex represents a unit of local computation, and each edge represents the output from, or input to, a vertex. We refer to the computation at vertices as *operations*, and the values that flow along edges as *tensors*. In this subsection, we describe the common types of operations and tensors.

Tensors In TensorFlow, we model all data as tensors (n-dimensional arrays) with the elements having one of a small number of primitive types, such as `int32`, `float32`, or `string` (where `string` can represent arbitrary binary data). Tensors naturally represent the inputs to and results of the common mathematical operations in many machine learning algorithms: for example, a matrix multiplication takes two 2-D tensors and produces a 2-D tensor; and a batch 2-D convolution takes two 4-D tensors and produces another 4-D tensor.

At the lowest level, all TensorFlow tensors are dense, for the reasons we discuss in Subsection 2.2. TensorFlow offers two alternatives for representing sparse data: either encode the data into variable-length `string` elements of a dense tensor, or use a tuple of dense tensors (e.g., an n-D sparse tensor with m non-zero elements can be represented in coordinate-list format as an $m \times n$ matrix of coordinates and a length-m vector of values). The shape of a tensor can vary in one or more of its dimensions, which makes it possible to represent sparse tensors with differing numbers of elements.

Operations An operation takes $m \geq 0$ tensors as input and produces $n \geq 0$ tensors as output. An operation has a named "type" (such as `Const`, `MatMul`, or `Assign`) and may have zero or more compile-time attributes that determine its behavior. An operation can be polymorphic and variadic at compile-time: its attributes determine both the expected types and arity of its inputs and outputs.

For example, the simplest operation `Const` has no inputs and a single output; its value is a compile-time attribute. For example, `AddN` sums multiple tensors of the same element type, and it has a type attribute `T` and an integer attribute `N` that define its type signature.

Stateful operations: variables An operation can contain mutable state that is read and/or written each time it executes. A `Variable` operation owns a mutable buffer that may be used to store the shared parameters of a model as it is trained. A `Variable` has no inputs, and produces a *reference handle*, which acts as a typed capability for reading and writing the buffer. A `Read` operation takes a reference handle r as input, and outputs the value of the variable ($\mathrm{State}[r]$) as a dense tensor. Other operations modify the underlying buffer: for example, `AssignAdd` takes a reference handle r and a tensor value x, and when executed performs the update $\mathrm{State}'[r] \leftarrow \mathrm{State}[r] + x$. Subsequent `Read`$(r)$ operations produce the value $\mathrm{State}'[r]$.

Stateful operations: queues TensorFlow includes several queue implementations, which support more advanced forms of coordination. The simplest queue is `FIFOQueue`, which owns an internal queue of tensors, and allows concurrent access in first-in-first-out order. Other types of queues dequeue tensors in random and priority orders, which ensure that input data are sampled appropriately. Like a `Variable`, the `FIFOQueue` operation produces a reference handle that can be consumed by one of the standard queue operations, such as `Enqueue` and `Dequeue`. These operations push their input onto the tail of the queue and, respectively, pop the head element and output it. `Enqueue` will block if its given queue is full, and `Dequeue` will block if its given queue is empty. When queues are used in an input preprocessing pipeline, this blocking provides backpressure; it also supports synchronization (§4.4). The combination of queues and dynamic control flow (§3.4) can also implement a form of streaming computation between subgraphs.

3.2 Partial and concurrent execution

TensorFlow uses a dataflow graph to represent all possible computations in a particular application. The API for executing a graph allows the client to specify declaratively the *subgraph* that should be executed. The client selects zero or more edges to *feed* input tensors into the dataflow, and one or more edges to *fetch* output tensors from the dataflow; the runtime then prunes the graph to contain the necessary set of operations. Each invocation of the API is called a *step*, and TensorFlow supports multiple *concurrent steps* on the same graph. Stateful operations allow steps to share data and synchronize when necessary.

Figure 2 shows a typical training application, with multiple subgraphs that execute concurrently and interact through shared variables and queues. The core training subgraph depends on a set of model parameters and on input batches from a queue. Many concurrent steps of the training subgraph update the model based on different input batches, to implement data-parallel training. To fill the input queue, concurrent preprocessing steps transform individual input records (e.g., decoding images and applying random distortions), and a separate I/O subgraph reads records from a distributed file system. A checkpointing subgraph runs periodically for fault tolerance (§4.3).

Partial and concurrent execution is responsible for much of TensorFlow's flexibility. Adding mutable state

and coordination via queues makes it possible to specify a wide variety of model architectures in user-level code, which enables advanced users to experiment without modifying the internals of the TensorFlow runtime. By default, concurrent executions of a TensorFlow subgraph run asynchronously with respect to one another. This asynchrony makes it straightforward to implement machine learning algorithms with weak consistency requirements [61], which include many neural network training algorithms [20]. As we discuss later, TensorFlow also provides the primitives needed to synchronize workers during training (§4.4), which has led to promising results on some learning tasks (§6.3).

3.3 Distributed execution

Dataflow simplifies distributed execution, because it makes communication between subcomputations explicit. It enables the same TensorFlow program to be deployed to a cluster of GPUs for training, a cluster of TPUs for serving, and a cellphone for mobile inference.

Each operation resides on a particular *device*, such as a CPU or GPU in a particular *task*. A device is responsible for executing a *kernel* for each operation assigned to it. TensorFlow allows multiple kernels to be registered for a single operation, with specialized implementations for a particular device or data type (see §5 for details). For many operations, such as element-wise operators (`Add`, `Sub`, etc.), we can compile a single kernel implementation for CPU and GPU using different compilers.

The TensorFlow runtime places operations on devices, subject to implicit or explicit constraints in the graph. The placement algorithm computes a feasible set of devices for each operation, calculates the sets of operations that must be colocated, and selects a satisfying device for each colocation group. It respects implicit colocation constraints that arise because each stateful operation and its state must be placed on the same device. In addition, the user may specify partial device preferences such as "any device in a particular task", or "a GPU in any task", and the runtime will respect these constraints. A typical training application will use client-side programming constructs to add constraints such that, for example, parameters are distributed among a set of "PS" tasks (§4.2).

TensorFlow thus permits great flexibility in how operations in the dataflow graph are mapped to devices. While simple heuristics yield adequate performance for novice users, expert users can optimize performance by manually placing operations to balance the computation, memory, and network requirements across multiple tasks and multiple devices within those tasks. An open question is how

```
input = ...  # A sequence of tensors
state = 0    # Initial state
w = ...      # Trainable weights

for i in range(len(input)):
  state, out[i] = f(state, w, input[i])
```

Figure 3: Pseudocode for an abstract RNN (§3.4). The function `f` typically comprises differentiable operations such as matrix multiplications and convolutions [32]. TensorFlow implements the loop in its dataflow graph.

TensorFlow can automatically determine placements that achieve close to optimal performance on a given set of devices, thus freeing users from this concern. Even without such automation, it may be worthwhile to separate placement directives from other aspects of model definitions, so that, for example, it would be trivial to modify placements after a model has been trained.

Once the operations in a graph have been placed, and the partial subgraph has been computed for a step (§3.2), TensorFlow partitions the operations into per-device subgraphs. A per-device subgraph for device d contains all of the operations that were assigned to d, with additional `Send` and `Recv` operations that replace edges across device boundaries. `Send` transmits its single input to a specified device as soon as the tensor is available, using a *rendezvous key* to name the value. `Recv` has a single output, and blocks until the value for a specified rendezvous key is available locally, before producing that value. `Send` and `Recv` have specialized implementations for several device-type pairs; we describe some of these in Section 5.

We optimized TensorFlow for executing large subgraphs repeatedly with low latency. Once the graph for a step has been pruned, placed, and partitioned, its subgraphs are cached in their respective devices. A client *session* maintains the mapping from step definitions to cached subgraphs, so that a distributed step on a large graph can be initiated with one small message to each participating task. This model favors static, reusable graphs, but it can support dynamic computations using dynamic control flow, as the next subsection describes.

3.4 Dynamic control flow

TensorFlow supports advanced machine learning algorithms that contain conditional and iterative control flow. For example, a *recurrent neural network* (RNN) [39] such as an LSTM [32] can generate predictions from sequential data. Google's Neural Machine Translation system uses TensorFlow to train a deep LSTM that achieves state-of-

the-art performance on many translation tasks [73]. The core of an RNN is a recurrence relation, where the output for sequence element i is a function of some state that accumulates across the sequence (Figure 3). In this case, dynamic control flow enables iteration over sequences that have variable lengths, without unrolling the computation to the length of the longest sequence.

As we discussed in Subsection 2.2, TensorFlow uses deferred execution via the dataflow graph to offload larger chunks of work to accelerators. Therefore, to implement RNNs and other advanced algorithms, we add conditional (if statement) and iterative (while loop) programming constructs in the dataflow graph itself. We use these primitives to build higher-order constructs, such as `map()`, `fold()`, and `scan()` [2].

For this purpose, we borrow the `Switch` and `Merge` primitives from classic dynamic dataflow architectures [4]. `Switch` is a demultiplexer: it takes a data input and a control input, and uses the control input to select which of its two outputs should produce a value. The `Switch` output not taken receives a special *dead* value, which propagates recursively through the rest of the graph until it reaches a `Merge` operation. `Merge` is a multiplexer: it forwards at most one non-dead input to its output, or produces a dead output if both of its inputs are dead. The conditional operator uses `Switch` to execute one of two branches based on the runtime value of a boolean tensor, and `Merge` to combine the outputs of the branches. The while loop is more complicated, and uses `Enter`, `Exit`, and `NextIteration` operators to ensure that the loop is well-formed [56].

The execution of iterations can overlap, and TensorFlow can also partition conditional branches and loop bodies across multiple devices and processes. The partitioning step adds logic to coordinate the start and termination of each iteration on each device, and to decide the termination of the loop. As we will see in Subsection 4.1, TensorFlow also supports automatic differentiation of control flow constructs. Automatic differentiation adds the subgraphs for computing gradients to the dataflow graph, which TensorFlow partitions across potentially distributed devices to compute the gradients in parallel.

4 Extensibility case studies

By choosing a unified representation for all computation in TensorFlow, we enable users to experiment with features that were hard-coded into the DistBelief runtime. In this section, we discuss four extensions that we have built using dataflow primitives and "user-level" code.

4.1 Differentiation and optimization

Many learning algorithms train a set of parameters using some variant of SGD, which entails computing the *gradients* of a loss function with respect to those parameters, then updating the parameters based on those gradients. TensorFlow includes a user-level library that differentiates a symbolic expression for a loss function and produces a new symbolic expression representing the gradients. For example, given a neural network as a composition of layers and a loss function, the library will automatically derive the backpropagation code.

The differentiation algorithm performs breadth-first search to identify all of the backwards paths from the target operation (e.g., a loss function) to a set of parameters, and sums the partial gradients that each path contributes. Our users frequently specialize the gradients for some operations, and they have implemented optimizations like batch normalization [33] and gradient clipping [60] to accelerate training and make it more robust. We have extended the algorithm to differentiate conditional and iterative subcomputations (§3.4) by adding nodes to the graph that record the control flow decisions in the forward pass, and replaying those decisions in reverse during the backward pass. Differentiating iterative computations over long sequences can lead to a large amount of intermediate state being accumulated in memory, and we have developed techniques for managing limited GPU memory on these computations.

TensorFlow users can also experiment with a wide range of *optimization algorithms*, which compute new values for the parameters in each training step. SGD is easy to implement in a parameter server: for each parameter W, gradient $\partial L/\partial W$, and learning rate α, the update rule is $W' \leftarrow W - \alpha \times \partial L/\partial W$. A parameter server can implement SGD by using $-=$ as the write operation, and writing $\alpha \times \partial L/\partial W$ to each W after a training step.

However, there are many more advanced optimization schemes that are difficult to express as a single write operation. For example, the Momentum algorithm accumulates a "velocity" for each parameter based on its gradient over multiple iterations, then computes the parameter update from that accumulation; and many refinements to this algorithm have been proposed [66]. Implementing Momentum in DistBelief [20], required modifications to the parameter server implementation to change the representation of parameter data, and execute complex logic in the write operation; such modifications are challenging for many users. Optimization algorithms are the topic of active research, and researchers have implemented several on top of TensorFlow, including Momentum, AdaGrad, AdaDelta, RMSProp, Adam, and L-BFGS. These

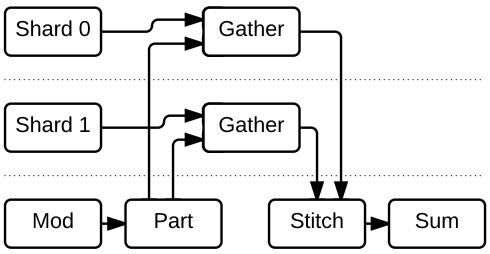

Figure 4: Schematic dataflow for an embedding layer (§4.2) with a two-way sharded embedding matrix.

can be built in TensorFlow using `Variable` operations and primitive mathematical operations without modifying the underlying system, so it is easy to experiment with new algorithms as they emerge.

4.2 Training very large models

To train a model on high-dimensional data, such as words in a corpus of text [7], it is common to use a *distributed representation*, which embeds a training example as a pattern of activity across several neurons, and which can be learned by backpropagation [30]. For example, in a language model, a training example might be a sparse vector with non-zero entries corresponding to the IDs of words in a vocabulary, and the distributed representation for each word will be a lower-dimensional vector [6]. "Wide and deep learning" creates distributed representations from cross-product transformations on categorical features, and the implementation on TensorFlow is used to power the Google Play app store recommender system [12].

Inference begins by multiplying a batch of b sparse vectors against an $n \times d$ *embedding matrix*, where n is the number of words in the vocabulary, and d is the desired dimensionality, to produce a much smaller $b \times d$ dense matrix representation; for training, most optimization algorithms modify only the rows of the embedding matrix that were read by the sparse multiplication. In TensorFlow models that process sparse data, $n \times d$ can amount to gigabytes of parameters: e.g., a large language model may use over 10^9 parameters with a vocabulary of 800,000 words [41], and we have experience with document models [19] where the parameters occupy several terabytes. Such models are too large to copy to a worker on every use, or even to store in RAM on a single host.

We implement sparse embedding layers in the TensorFlow graph as a composition of primitive operations. Figure 4 shows a simplified graph for an embedding layer that is split across two parameter server tasks. The core operation of this subgraph is `Gather`, which extracts a sparse set of rows from a tensor, and TensorFlow colocates this operation with the variable on which it operates. The dynamic partition (`Part`) operation divides the incoming indices into variable-sized tensors that contain the indices destined for each shard, and the dynamic stitching (`Stitch`) operation reassembles the partial results from each shard into a single result tensor. Each of these operations has a corresponding gradient, so it supports automatic differentiation (§4.1), and the result is a set of sparse update operations that act on just the values that were originally gathered from each of the shards.

Users writing a TensorFlow model typically do not construct graphs like Figure 4 manually. Instead TensorFlow includes libraries that expose the abstraction of a sharded parameter, and build appropriate graphs of primitive operations based on the desired degree of distribution.

While sparse reads and updates are possible in a parameter server [49], TensorFlow adds the flexibility to offload arbitrary computation onto the devices that host the shared parameters. For example, classification models typically use a softmax classifier that multiplies the final output by a weight matrix with c columns, where c is the number of possible classes; for a language model, c is the size of the vocabulary, which can be large. Our users have experimented with several schemes to accelerate the softmax calculation. The first is similar to an optimization in Project Adam [14], whereby the weights are sharded across several tasks, and the multiplication and gradient calculation are colocated with the shards. More efficient training is possible using a *sampled softmax* [37], which performs a sparse multiplication based on the true class for an example and a set of randomly sampled false classes. We compare the performance of these two schemes in §6.4.

4.3 Fault tolerance

Training a model can take several hours or days, even using a large number of machines [14, 20]. We often need to train a model using non-dedicated resources, for example using the Borg cluster manager [71], which does not guarantee availability of the same resources for the duration of the training process. Therefore, a long-running TensorFlow job is likely to experience failure or pre-emption, and we require some form of fault tolerance. It is unlikely that tasks will fail so often that individual operations need fault tolerance, so a mechanism like Spark's RDDs [75] would impose significant overhead for little benefit. There is no need to make every write to the parameter state durable, because we can recompute any update from the input data, and many learning algorithms do not require strong consistency [61].

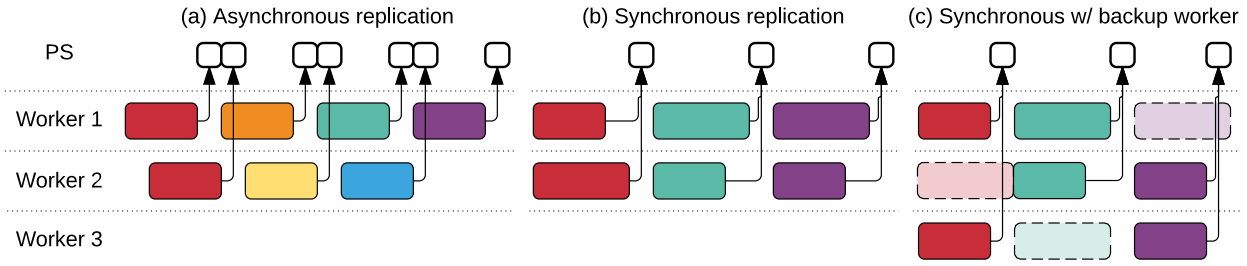

Figure 5: Three synchronization schemes for parallel SGD. Each color represents a different starting parameter value; a white square is a parameter update. In (c), a dashed rectangle represents a backup worker whose result is discarded.

We implement user-level checkpointing for fault tolerance, using two operations in the graph (Figure 2): `Save` writes one or more tensors to a checkpoint file, and `Restore` reads one or more tensors from a checkpoint file. Our typical configuration connects each `Variable` in a task to the same `Save` operation, with one `Save` per task, to maximize the I/O bandwidth to a distributed file system. The `Restore` operations read named tensors from a file, and a standard `Assign` stores the restored value in its respective variable. During training, a typical client runs all of the `Save` operations periodically to produce a new checkpoint; when the client starts up, it attempts to `Restore` the latest checkpoint.

TensorFlow includes a client library for constructing the appropriate graph structure and for invoking `Save` and `Restore` as necessary. This behavior is customizable: the user can apply different policies to subsets of the variables in a model, or customize the checkpoint retention scheme. For example, many users retain checkpoints with the highest score in a custom evaluation metric. The implementation is also reusable: it may be used for model fine-tuning and unsupervised pre-training [45, 47], which are forms of transfer learning, in which the parameters of a model trained on one task (e.g., recognizing general images) are used as the starting point for another task (e.g., recognizing breeds of dog). Having checkpoint and parameter management as programmable operations in the graph gives users the flexibility to implement schemes like these and others that we have not anticipated.

The checkpointing library does not attempt to produce consistent checkpoints: if training and checkpointing execute concurrently, the checkpoint may include none, all, or some of the updates from the training step. This behavior is compatible with the relaxed guarantees of asynchronous SGD [20]. Consistent checkpoints require additional synchronization to ensure that update operations do not interfere with checkpointing; if desired, one can use the scheme in the next subsection to take a checkpoint after the synchronous update step.

4.4 Synchronous replica coordination

SGD is robust to asynchrony [61], and many systems train deep neural networks using asynchronous parameter updates [14, 20], which are believed scalable because they maintain high throughput in the presence of stragglers. The increased throughput comes at the cost of using stale parameter values in training steps. Some have recently revisited the assumption that *synchronous* training does not scale [10, 18]. Since GPUs enable training with hundreds—rather than thousands [47]—of machines, synchronous training may be faster (in terms of time to quality) than asynchronous training on the same platform.

Though we originally designed TensorFlow for asynchronous training, we have begun experimenting with synchronous methods. The TensorFlow graph enables users to change how parameters are read and written when training a model, and we implement three alternatives. In the asynchronous case (Figure 5(a)), each worker reads the current values of parameters when each step begins, and applies its gradient to the (possibly different) current values at the end: this approach ensures high utilization, but the individual steps use stale parameter values, making each step less effective. We implement the synchronous version using queues (§3.1) to coordinate execution: a blocking queue acts as a barrier to ensure that all workers read the same parameter values, and a per-variable queue accumulates gradient updates from all workers in order to apply them atomically. The simple synchronous version (Figure 5(b)) accumulates updates from all workers before applying them, but slow workers limit overall throughput.

To mitigate stragglers, we implement *backup workers* (Figure 5(c), [10]), which are similar to MapReduce backup tasks [21]. Whereas MapReduce starts backup tasks reactively—after detecting a straggler—our backup workers run proactively, and the aggregation takes the first m of n updates produced. We exploit the fact that SGD samples training data randomly at each step, so each worker processes a different random batch, and it is not a

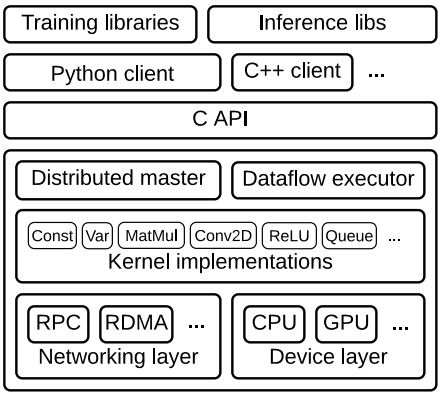

Figure 6: The layered TensorFlow architecture.

problem if a particular batch is ignored. In §6.3 we show how backup workers improve throughput by up to 10%.

5 Implementation

The TensorFlow runtime is a cross-platform library. Figure 6 illustrates its architecture: a C API separates user-level code in different languages from the core runtime.

The core TensorFlow library is implemented in C++ for portability and performance: it runs on several operating systems including Linux, Mac OS X, Windows, Android, and iOS; the x86 and various ARM-based CPU architectures; and NVIDIA's Kepler, Maxwell, and Pascal GPU microarchitectures. The implementation is open-source, and we have accepted several external contributions that enable TensorFlow to run on other architectures.

The *distributed master* translates user requests into execution across a set of tasks. Given a graph and a step definition, it prunes (§3.2) and partitions (§3.3) the graph to obtain subgraphs for each participating device, and caches these subgraphs so that they may be re-used in subsequent steps. Since the master sees the overall computation for a step, it applies standard optimizations such as common subexpression elimination and constant folding; pruning is a form of dead code elimination. It then coordinates execution of the optimized subgraphs across a set of tasks.

The *dataflow executor* in each task handles requests from the master, and schedules the execution of the kernels that comprise a local subgraph. We optimize the dataflow executor for running large graphs with low overhead. Our current implementation can execute 10,000 subgraphs per second (§6.2), which enables a large number of replicas to make rapid, fine-grained training steps. The dataflow executor dispatches kernels to local devices and runs kernels in parallel when possible, for example by using multiple CPU cores or GPU streams.

The runtime contains over 200 standard operations, including mathematical, array manipulation, control flow, and state management operations. Many of the operation kernels are implemented using Eigen::Tensor [36], which uses C++ templates to generate efficient parallel code for multicore CPUs and GPUs; however, we liberally use libraries like cuDNN [13] where a more efficient kernel implementation is possible. We have also implemented *quantization*, which enables faster inference in environments such as mobile devices and high-throughput datacenter applications, and use the `gemmlowp` low-precision matrix library [35] to accelerate quantized computation.

We specialize `Send` and `Recv` operations for each pair of source and destination device types. Transfers between local CPU and GPU devices use the `cudaMemcpyAsync()` API to overlap computation and data transfer; transfers between two local GPUs use DMA to relieve pressure on the host. For transfers between tasks, TensorFlow uses multiple protocols, including gRPC over TCP, and RDMA over Converged Ethernet. We are also investigating optimizations for GPU-to-GPU communication that use collective operations [59].

Section 4 describes features that we implement completely above the C API, in user-level code. Typically, users compose standard operations to build higher-level abstractions, such as neural network layers, optimization algorithms (§4.1), and sharded embedding computations (§4.2). TensorFlow supports multiple client languages, and we have prioritized Python and C++, because our internal users are most familiar with these languages. As features become more established, we typically port them to C++, so that users can access an optimized implementation from all client languages.

If it is difficult or inefficient to represent a subcomputation as a composition of operations, users can register additional kernels that provide an efficient implementation written in C++. We have found it profitable to hand-implement *fused kernels* for some performance critical operations, such as the ReLU and Sigmoid activation functions and their corresponding gradients. We are currently investigating automatic kernel fusion using a compilation-based approach.

In addition to the core runtime, our colleagues have built several tools that aid users of TensorFlow. These include serving infrastructure for inference in production [27], a visualization dashboard that enables users to follow the progress of a training run, a graph visualizer that helps users to understand the connections in a model, and a distributed profiler that traces the execution of a computation across multiple devices and tasks. We describe these tools in an extended whitepaper [1].

6 Evaluation

In this section, we evaluate the performance of TensorFlow on several synthetic and realistic workloads. Unless otherwise stated, we run all experiments on a shared production cluster, and all figures plot median values with error bars showing the 10th and 90th percentiles.

In this paper we focus on system performance metrics, rather than learning objectives like time to accuracy. TensorFlow is a system that allows machine learning practitioners and researchers to experiment with new techniques, and this evaluation demonstrates that the system (i) has little overhead, and (ii) can employ large amounts of computation to accelerate real-world applications. While techniques like synchronous replication can enable some models to converge in fewer steps overall, we defer the analysis of such improvements to other papers.

6.1 Single-machine benchmarks

Although TensorFlow is a system for "large-scale" machine learning, it is imperative that scalability does not mask poor performance at small scales [51]. Table 1 contains results from Chintala's benchmark of convolutional models on TensorFlow and three single-machine frameworks [15]. All frameworks use a six-core Intel Core i7-5930K CPU at 3.5 GHz and an NVIDIA Titan X GPU.

	Training step time (ms)			
Library	AlexNet	Overfeat	OxfordNet	GoogleNet
Caffe [38]	324	823	1068	1935
Neon [58]	87	**211**	**320**	**270**
Torch [17]	**81**	268	529	470
TensorFlow	**81**	279	540	445

Table 1: Step times for training four convolutional models with different libraries, using one GPU. All results are for training with 32-bit floats. The fastest time for each model is shown in bold.

Table 1 shows that TensorFlow achieves shorter step times than Caffe [38], and performance within 6% of the latest version of Torch [17]. We attribute the similar performance of TensorFlow and Torch to the fact that both use the same version of the cuDNN library [13], which implements the convolution and pooling operations on the critical path for training; Caffe uses open-source implementations for these operations that are simpler but less efficient than cuDNN. The Neon library [58] outperforms TensorFlow on three of the models, by using hand-optimized convolutional kernels [46] implemented in assembly language; in principle, we could follow the same approach in TensorFlow, but we have not yet done so.

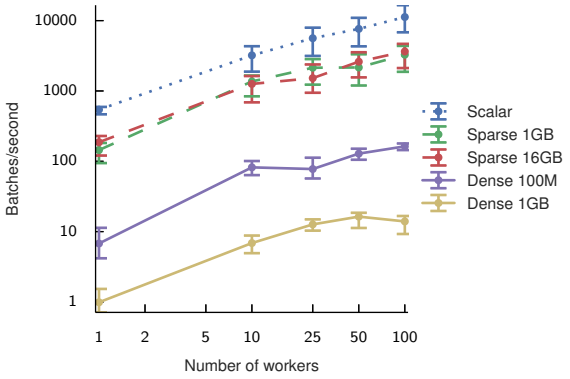

Figure 7: Baseline throughput for synchronous replication with a null model. Sparse accesses enable TensorFlow to handle larger models, such as embedding matrices (§4.2).

6.2 Synchronous replica microbenchmark

The performance of our coordination implementation (§4.4) is the main limiting factor for scaling with additional machines. Figure 7 shows that number of *null training steps* that TensorFlow performs per second for varying model sizes, and increasing numbers of *synchronous* workers. In a null training step, a worker fetches the shared model parameters from 16 PS tasks, performs a trivial computation, and sends updates to the parameters.

The *Scalar* curve in Figure 7 shows the best performance that we could expect for a synchronous training step, because only a single 4-byte value is fetched from each PS task. The median step time is 1.8 ms using a single worker, growing to 8.8 ms with 100 workers. These times measure the overhead of the synchronization mechanism, and capture some of the noise that we expect when running on a shared cluster.

The *Dense* curves show the performance of a null step when the worker fetches the entire model. We repeat the experiment with models of size 100 MB and 1 GB, with the parameters sharded equally over 16 PS tasks. The median step time for 100 MB increases from 147 ms with one worker to 613 ms with 100 workers. For 1 GB, it increases from 1.01 s with one worker to 7.16 s with 100 workers.

For large models, a typical training step accesses only a subset of the parameters, and the *Sparse* curves show the throughput of the embedding lookup operation from Subsection 4.2. Each worker reads 32 randomly selected entries from a large embedding matrix containing 1 GB or 16 GB of data. As expected, the step times do not vary with the size of the embedding, and TensorFlow achieves step times ranging from 5 to 20 ms.

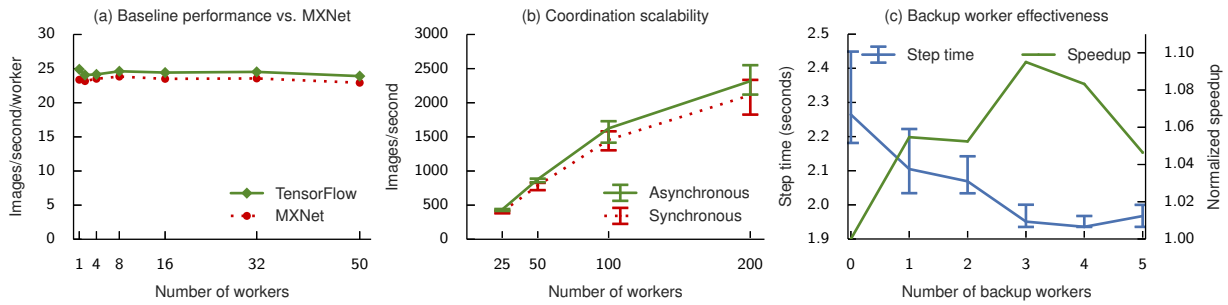

Figure 8: Results of the performance evaluation for Inception-v3 training (§6.3). (a) TensorFlow achieves slightly better throughput than MXNet for asynchronous training. (b) Asynchronous and synchronous training throughput increases with up to 200 workers. (c) Adding backup workers to a 50-worker training job can reduce the overall step time, and improve performance even when normalized for resource consumption.

6.3 Image classification

Deep neural networks have achieved breakthrough performance on computer vision tasks such as recognizing objects in photographs [44], and these tasks are a key application for TensorFlow at Google. Training a network to high accuracy requires a large amount of computation, and we use TensorFlow to scale out this computation across a cluster of GPU-enabled servers. In these experiments, we focus on Google's Inception-v3 model, which achieves 78.8% accuracy in the ILSVRC 2012 image classification challenge [69]; the same techniques apply to other deep convolutional models—such as ResNet [28]—implemented on TensorFlow. We investigate the scalability of training Inception-v3 using multiple replicas. We configure TensorFlow with 7 PS tasks, and vary the number of worker tasks using two different clusters.

For the first experiment, we compare the performance training Inception using asynchronous SGD on TensorFlow and MXNet, a contemporary system using a parameter server architecture. For this experiment we use Google Compute Engine virtual machines running on Intel Xeon E5 servers with NVIDIA K80 GPUs, configured with 8 vCPUs, 16Gbps of network bandwidth, and one GPU per VM. Both systems use 7 PS tasks running on separate VMs with no GPU. Figure 8(a) shows that TensorFlow achieves performance that is marginally better than MXNet. As expected, the results are largely determined by single-GPU performance, and both systems use cuDNN version 5.1, so they have access to the same optimized GPU kernels.

Using a larger internal cluster (with NVIDIA K40 GPUs, and a shared datacenter network), we investigate the effect of coordination (§4.4) on training performance. Ideally, with efficient synchronous training, a model such as Inception-v3 will train in fewer steps, and converge to a higher accuracy than with asynchronous training [10]. Training throughput improves to 2,300 images per second as we increase the number of workers to 200, but with diminishing returns (Figure 8(b)). As we add more workers, the step time increases, because there is more contention on the PS tasks, both at the network interface and in the aggregation of updates. As expected, for all configurations, synchronous steps are longer than asynchronous steps, because all workers must wait for the slowest worker to catch up before starting the next step. While the median synchronous step is approximately 10% longer than an asynchronous step with the same workers, above the 90th percentile the synchronous performance degrades sharply, because stragglers disproportionately impact tail latency.

To mitigate tail latency, we add backup workers so that a step completes when the first m of n tasks produce gradients. Figure 8(c) shows the effect of adding backup workers to a 50-worker Inception training job. Each additional backup worker up to and including the fourth reduces the median step time, because the probability of a straggler affecting the step decreases. Adding a fifth backup worker slightly degrades performance, because the 51st worker (i.e., the first whose result is discarded) is more likely to be a non-straggler that generates more incoming traffic for the PS tasks. Figure 8(c) also plots the *normalized speedup* for each configuration, defined as $t(b)/t(0) \times 50/(50+b)$ (where $t(b)$ is the median step time with b backup workers), and which discounts the speedup by the fraction of additional resources consumed. Although adding 4 backup workers achieves the shortest overall step time (1.93 s), adding 3 achieves the highest normalized speedup (9.5%), and hence uses less aggregate GPU-time to reach the same quality.

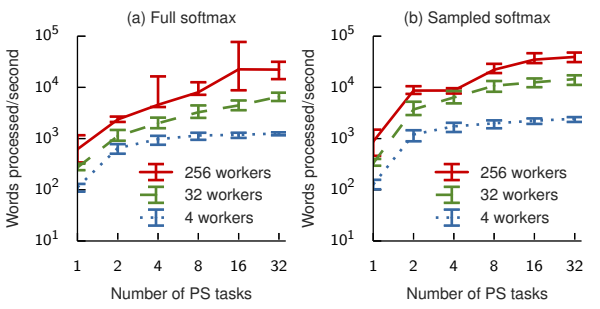

Figure 9: Increasing the number of PS tasks leads to increased throughput for language model training, by parallelizing the softmax computation. Sampled softmax increases throughput by performing less computation.

6.4 Language modeling

Given a sequence of words, a language model predicts the most probable next word [6]. Therefore, language models are integral to predictive text, speech recognition, and translation applications. In this experiment, we investigate how TensorFlow can train a recurrent neural network (viz. LSTM-512-512 [41]) to model the text in the One Billion Word Benchmark [9]. The vocabulary size $|V|$ limits the performance of training, because the final layer must decode the output state into probabilities for each of $|V|$ classes [37]. The resulting parameters can be large ($|V| \times d$ for output state dimension d) so we use the techniques for handling large models from Subsection 4.2. We use a restricted vocabulary of the most common 40,000 words—instead of the full 800,000 words [9]—in order to experiment with smaller configurations.

Figure 9 shows the training throughput, measured in words per second, for varying numbers of PS and worker tasks, and two softmax implementations. The *full* softmax (Figure 9(a)) multiplies each output by a 512 $\times$ 40,000 weight matrix sharded across the PS tasks. Adding more PS tasks increases the throughput, because TensorFlow can exploit distributed model parallelism [20, 43] and perform the multiplication and gradient calculation on the PS tasks, as in Project Adam [14]. Adding a second PS task is more effective than increasing from 4 to 32, or 32 to 256 workers. Eventually the throughput saturates, as the LSTM calculations dominate the training step.

The *sampled* softmax (Figure 9(b)) reduces the data transferred and the computation performed on the PS tasks [37]. Instead of a dense weight matrix, it multiplies the output by a random sparse matrix containing weights for the true class and a random sample of false classes. We sample 512 classes for each batch, thus reducing the softmax data transfer and computation by a factor of 78.

7 Conclusions

We have described the TensorFlow system and its programming model. TensorFlow's dataflow representation subsumes existing work on parameter server systems, and offers a set of uniform abstractions that allow users to harness large-scale heterogeneous systems, both for production tasks and for experimenting with new approaches. We have shown several examples of how the TensorFlow programming model facilitates experimentation (§4) and demonstrated that the resulting implementations are performant and scalable (§6).

Our initial experience with TensorFlow is encouraging. A large number of groups at Google have deployed TensorFlow in production, and TensorFlow is helping our research colleagues to make new advances in machine learning. Since we released TensorFlow as open-source software, more than 14,000 people have forked the source code repository, the binary distribution has been downloaded over one million times, and dozens of machine learning models that use TensorFlow have been published.

TensorFlow is a work in progress. Its flexible dataflow representation enables power users to achieve excellent performance, but we have not yet determined default policies that work well for all users. Further research on automatic optimization should bridge this gap. On the system level, we are actively developing algorithms for automatic placement, kernel fusion, memory management, and scheduling. While the current implementations of mutable state and fault tolerance suffice for applications with weak consistency requirements, we expect that some TensorFlow applications will require stronger consistency, and we are investigating how to build such policies at user-level. Finally, some users have begun to chafe at the limitations of a static dataflow graph, especially for algorithms like deep reinforcement learning [54]. Therefore, we face the intriguing problem of providing a system that transparently and efficiently uses distributed resources, even when the structure of the computation unfolds dynamically.

Acknowledgments

We gratefully acknowledge contributions from our colleagues within Google, and from members of the wider machine learning community. In particular, we appreciate the feedback we have received from the rest of the Google Brain team and the many users of DistBelief and TensorFlow. We thank the anonymous OSDI reviewers and our shepherd KyoungSoo Park for their suggestions, which greatly improved the presentation of this paper.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint*, 1603.04467, 2016. arxiv.org/abs/1603.04467. Software available from tensorflow.org.

[2] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, E. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint*, 1605.02688, 2016. arxiv.org/abs/1605.02688.

[3] A. Angelova, A. Krizhevsky, and V. Vanhoucke. Pedestrian detection with a large-field-of-view deep network. In *Proceedings of ICRA*, pages 704–711. IEEE, 2015. www.vision.caltech.edu/anelia/publications/Angelova15LFOV.pdf.

[4] Arvind and D. E. Culler. Dataflow architectures. In *Annual Review of Computer Science Vol. 1, 1986*, pages 225–253. Annual Reviews Inc., 1986. www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA166235.

[5] J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint*, 1412.7755, 2014. arxiv.org/abs/1412.7755.

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003. jmlr.org/papers/volume3/bengio03a/bengio03a.pdf.

[7] T. Brants and A. Franz. Web 1T 5-gram version 1, 2006. catalog.ldc.upenn.edu/LDC2006T13.

[8] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine learning. *Mathematical Programming*, 134(1):127–155, 2012. dx.doi.org/10.1007/s10107-012-0572-5.

[9] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, and P. Koehn. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint*, 1312.3005, 2013. arxiv.org/abs/1312.3005.

[10] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous SGD. In *Proceedings of ICLR Workshop Track*, 2016. arxiv.org/abs/1604.00981.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Proceedings of LearningSys*, 2015. www.cs.cmu.edu/~muli/file/mxnet-learning-sys.pdf.

[12] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide & deep learning for recommender systems. *arXiv preprint*, 1606.07792, 2016. arxiv.org/abs/1606.07792.

[13] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint*, 1410.0759, 2014. arxiv.org/abs/1410.0759.

[14] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *Proceedings of OSDI*, pages 571–582, 2014. www.usenix.org/system/files/conference/osdi14/osdi14-paper-chilimbi.pdf.

[15] S. Chintala. convnet-benchmarks, 2016. github.com/soumith/convnet-benchmarks.

[16] E. S. Chung, J. D. Davis, and J. Lee. LINQits: Big data on little clients. In *Proceedings of ISCA*, pages 261–272, 2013. www.microsoft.com/en-us/research/wp-content/uploads/2013/06/ISCA13_-linqits.pdf.

[17] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: A modular machine learning software library. Technical report, IDIAP, 2002. infoscience.epfl.ch/record/82802/files/rr02-46.pdf.

[18] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of EuroSys*, 2016. www.pdl.cmu.edu/PDL-FTP/CloudComputing/GeePS-cui-eurosys16.pdf.

[19] A. Dai, C. Olah, and Q. V. Le. Document embedding with paragraph vectors. *arXiv preprint*, 1507.07998, 2015. arxiv.org/abs/1507.07998.

[20] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Proceedings of NIPS*, pages 1232–1240, 2012. research.google.com/archive/large_deep_networks_nips2012.pdf.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–149, 2004. research.google.com/archive/mapreduce-osdi04.pdf.

[22] DMLC. MXNet for deep learning, 2016. github.com/dmlc/mxnet.

[23] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011. jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[24] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, T. Mikolov, et al. DeVISE: A deep visual-semantic embedding model. In *Proceedings of NIPS*, pages 2121–2129, 2013. research.google.com/pubs/archive/41473.pdf.

[25] J. Gonzalez-Dominguez, I. Lopez-Moreno, P. J. Moreno, and J. Gonzalez-Rodriguez. Frame-by-frame language identification in short utterances using deep neural networks. *Neural Networks*, 64:49–58, 2015. research.google.com/pubs/archive/42929.pdf.

[26] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of NIPS*, pages 2672–2680, 2014. papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[27] Google Research. Tensorflow serving, 2016. tensorflow.github.io/serving/.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of CVPR*, pages 770–778, 2016. arxiv.org/abs/1512.03385.

[29] G. Heigold, V. Vanhoucke, A. Senior, P. Nguyen, M. Ranzato, M. Devin, and J. Dean. Multilingual acoustic models using distributed deep neural networks. In *Proceedings of ICASSP*, pages 8619–8623, 2013. research.google.com/pubs/archive/40807.pdf.

[30] G. E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 1–12, 1986. www.cogsci.ucsd.edu/~ajyu/Teaching/Cogs202_-sp13/Readings/hinton86.pdf.

[31] G. E. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. 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.

[32] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf.

[33] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of ICML*, pages 448–456, 2015. jmlr.org/proceedings/papers/v37/ioffe15.pdf.

[34] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*, pages 59–72, 2007. www.microsoft.com/en-us/research/wp-content/uploads/2007/03/eurosys07.pdf.

[35] B. Jacob et al. gemmlowp: a small self-contained low-precision GEMM library, 2015. github.com/google/gemmlowp.

[36] B. Jacob, G. Guennebaud, et al. Eigen library for linear algebra. eigen.tuxfamily.org.

[37] S. Jean, K. Cho, R. Memisevic, and Y. Bengio. On using very large target vocabulary for neural machine translation. In *Proceedings of ACL-ICJNLP*, pages 1–10, July 2015. www.aclweb.org/anthology/P15-1001.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of ACM Multimedia*, pages 675–678, 2014. arxiv.org/abs/1408.5093.

[39] M. I. Jordan. Serial order: A parallel distributed processing approach. ICS report 8608, Institute for Cognitive Science, UCSD, La Jolla, 1986. cseweb.ucsd.edu/~gary/PAPER-SUGGESTIONS/Jordan-TR-8604.pdf.

[40] N. Jouppi. Google supercharges machine learning tasks with TPU custom chip, 2016. cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[41] R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint*, 1602.02410, 2016. arxiv.org/abs/1602.02410.

[42] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of CVPR*, pages 1725–1732, 2014. research.google.com/pubs/archive/42455.pdf.

[43] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint*, 1404.5997, 2014. arxiv.org/abs/1404.5997.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of NIPS*, pages 1106–1114, 2012. papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[45] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, 10:1–40, 2009. jmlr.org/papers/volume10/larochelle09a/larochelle09a.pdf.

[46] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *Proceedings of CVPR*, pages 4013–4021, 2016. arxiv.org/abs/1509.09308.

[47] Q. Le, M. Ranzato, R. Monga, M. Devin, G. Corrado, K. Chen, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. In *Proceedings of ICML*, pages 81–88, 2012. research.google.com/archive/unsupervised_icml2012.pdf.

[48] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits, 1998. yann.lecun.com/exdb/mnist/.

[49] M. Li, D. G. Andersen, J. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the Parameter Server. In *Proceedings of OSDI*, pages 583–598, 2014. www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf.

[50] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver. Move evaluation in Go using deep convolutional neural networks. *arXiv preprint*, 1412.6564, 2014. arxiv.org/abs/1412.6564.

[51] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *Proceedings of HotOS*, HOTOS'15, 2015. www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf.

[52] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Proceedings of ICLR Workshops Track*, 2013. arxiv.org/abs/1301.3781.

[53] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent models of visual attention. In *Proceedings of NIPS*, pages 2204–2212, 2014. papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.pdf.

[54] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. dx.doi.org/10.1038/nature14236.

[55] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training deep networks in Spark. In *Proceedings of ICLR*, 2016. arxiv.org/abs/1511.06051.

[56] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, Sept. 2016. dl.acm.org/citation.cfm?id=2983551.

[57] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint*, 1507.04296, 2015. arxiv.org/abs/1507.04296.

[58] Nervana Systems. Neon deep learning framework, 2016. github.com/NervanaSystems/neon.

[59] NVIDIA Corporation. NCCL: Optimized primitives for collective multi-GPU communication, 2016. github.com/NVIDIA/nccl.

[60] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of ICML*, pages 1310–1318, 2013. jmlr.org/proceedings/papers/v28/pascanu13.pdf.

[61] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of NIPS*, pages 693–701, 2011. papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf.

[62] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of SOSP*, pages 49–68, 2013. sigops.org/sosp/sosp13/papers/p49-rossbach.pdf.

[63] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. In *Cognitive modeling*, volume 5, pages 213–220. MIT Press, 1988. www.cs.toronto.edu/~hinton/absps/naturebp.pdf.

[64] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. arxiv.org/abs/1409.0575.

[65] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1–2):703–710, Sept. 2010. vldb.org/pvldb/vldb2010/papers/R63.pdf.

[66] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of ICML*, pages 1139–1147, 2013. jmlr.org/proceedings/papers/v28/sutskever13.pdf.

[67] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proceedings of NIPS*, pages 3104–3112, 2014. papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural.pdf.

[68] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of CVPR*, pages 1–9, 2015. arxiv.org/abs/1409.4842.

[69] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception architecture for computer vision. *arXiv preprint*, 1512.00567, 2015. arxiv.org/abs/1512.00567.

[70] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng. Map-reduce for machine learning on multicore. In *Proceedings of NIPS*, pages 281–288, 2007. papers.nips.cc/paper/3150-map-reduce-for-machine-learning-on-multicore.pdf.

[71] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of EuroSys*, 2015. research.google.com/pubs/archive/43438.pdf.

[72] O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton. Grammar as a foreign language. *arXiv preprint*, 2014. arxiv.org/abs/1412.7449.

[73] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's Neural Machine Translation system: Bridging the gap between human and machine translation. *arXiv preprint*, 1609.08144, 2016. arxiv.org/abs/1609.08144.

[74] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of OSDI*, pages 1–14, 2008. www.usenix.org/legacy/event/osdi08/tech/full_papers/yu_y/yu_y.pdf.

[75] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI*, pages 15–28, 2012. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[76] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton. On rectified linear units for speech processing. In *Proceedings of ICASSP*, pages 3517–3521, 2013. research.google.com/pubs/archive/40811.pdf.