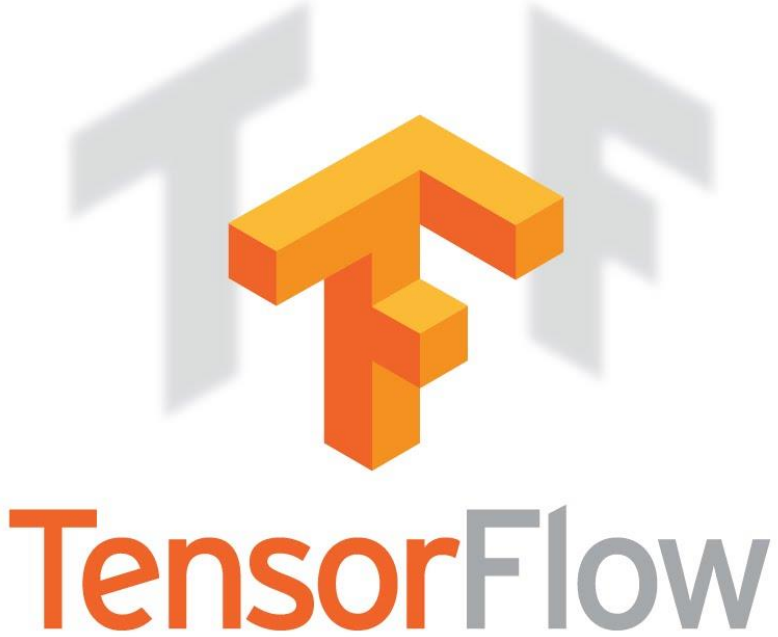
TensorFlow: A System for Large-Scale Machine Learning



Nathan Smith, Razieh Nabi, Alex Gain

Presentation Plan

- On Monday, we covered Spark and SystemML, big data systems that optimize the manipulation of big data
- We can now store and perform computations on large datasets, using things like MapReduce, Spark, Hadoop, SystemML, HBase, Cassandra, and more
- Today, we'll continue with that topic, with a greater focus on how we **understand** that data

Understanding?

Understanding?

- Query: car parts for sale
- Document 1:
  - ...**car** parking available **for** a small fee.
  - ...**parts** of our floor model inventory **for sale**
- Document 2:
  - Selling all kinds of automobile and pickup truck **parts**, engines, and transmissions.

What do you want in a machine learning system?

- Ease of expression: for lots of crazy ML ideas/algorithms
- Scalability: can run experiments quickly
- Portability: can run on wide variety of platforms
- Reproducibility: easy to share and reproduce research
- Production readiness

Machine Learning Requirements

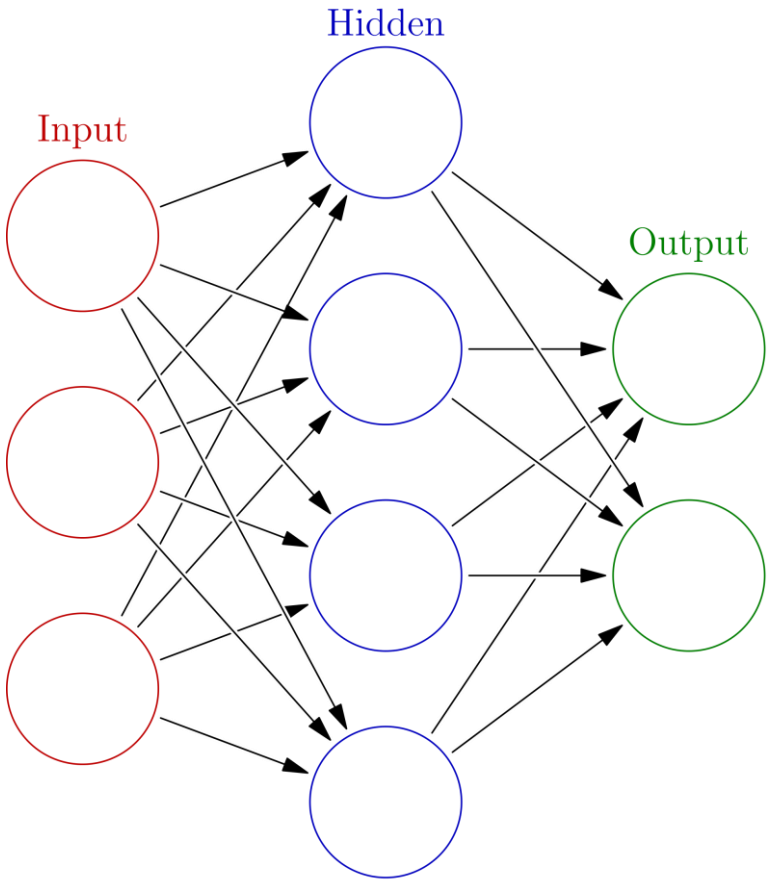
- Increase in training data generally improves algorithms
- Parallel training: workers process different subsets of data in parallel
- Most models have a large number of parameters, which can be efficiently divided with a distributed system
- Expensive computations: matrix multiplication, convolution, can be parallelized

Queries of the Future

- Which of these eye images shows symptoms of diabetic retinopathy?
- Find me all rooftops in North America
- Describe this video in Spanish
- Find me all documents relevant to reinforcement learning for robotics and summarize them in German

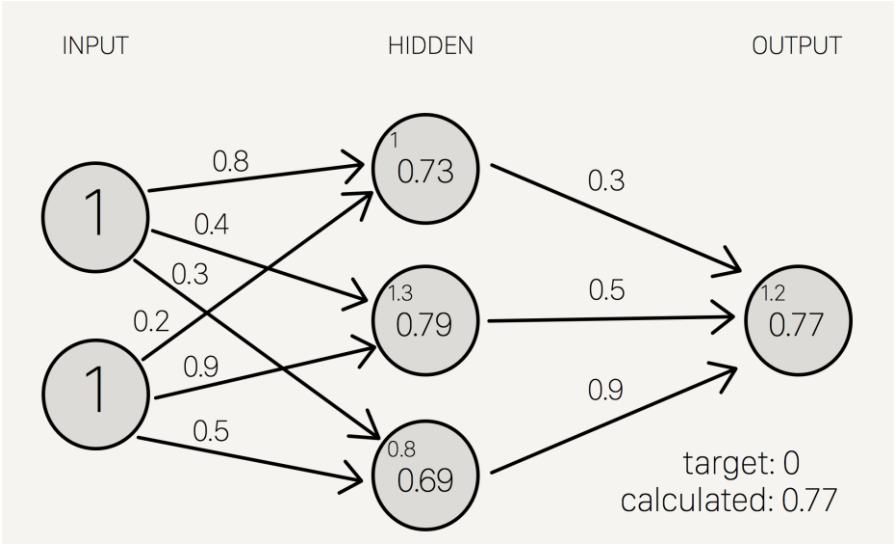
Neural Networks

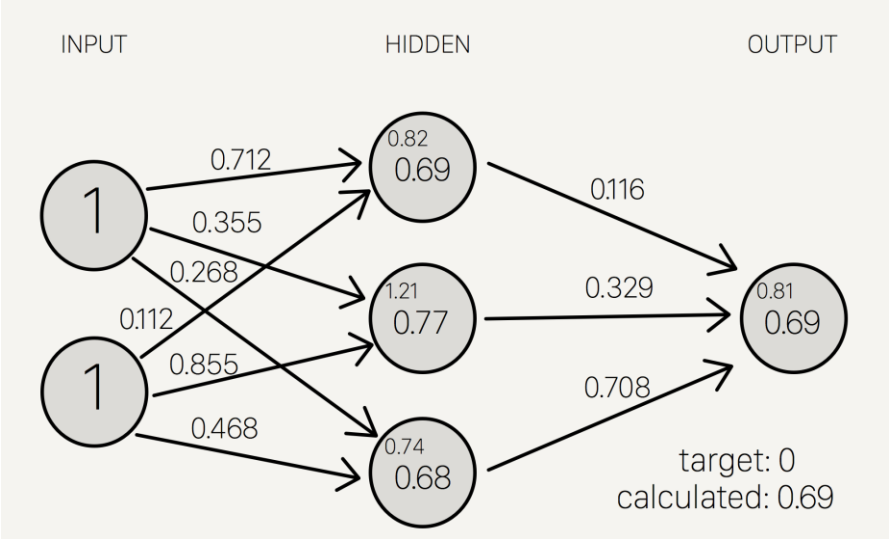
- A machine learning network inspired by biological neural networks
- Used to estimate or approximate functions that can depend on a large number of generally unknown inputs
- Contains a set of adaptive weights that are tuned by a learning algorithm



Training a Neural Network

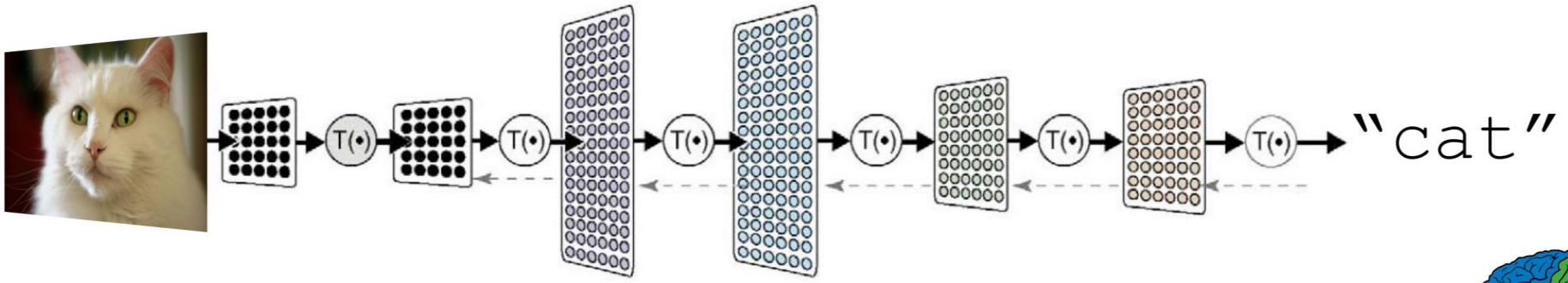
- In forward propagation, we apply a set of weights to the input data and calculate an output
- For the first forward propagation, the set of weights is selected randomly
- In back propagation, we measure the margin of error of the output and adjust the weights using partial derivatives and following gradients to decrease the error

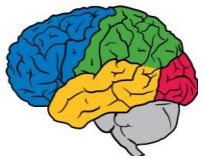




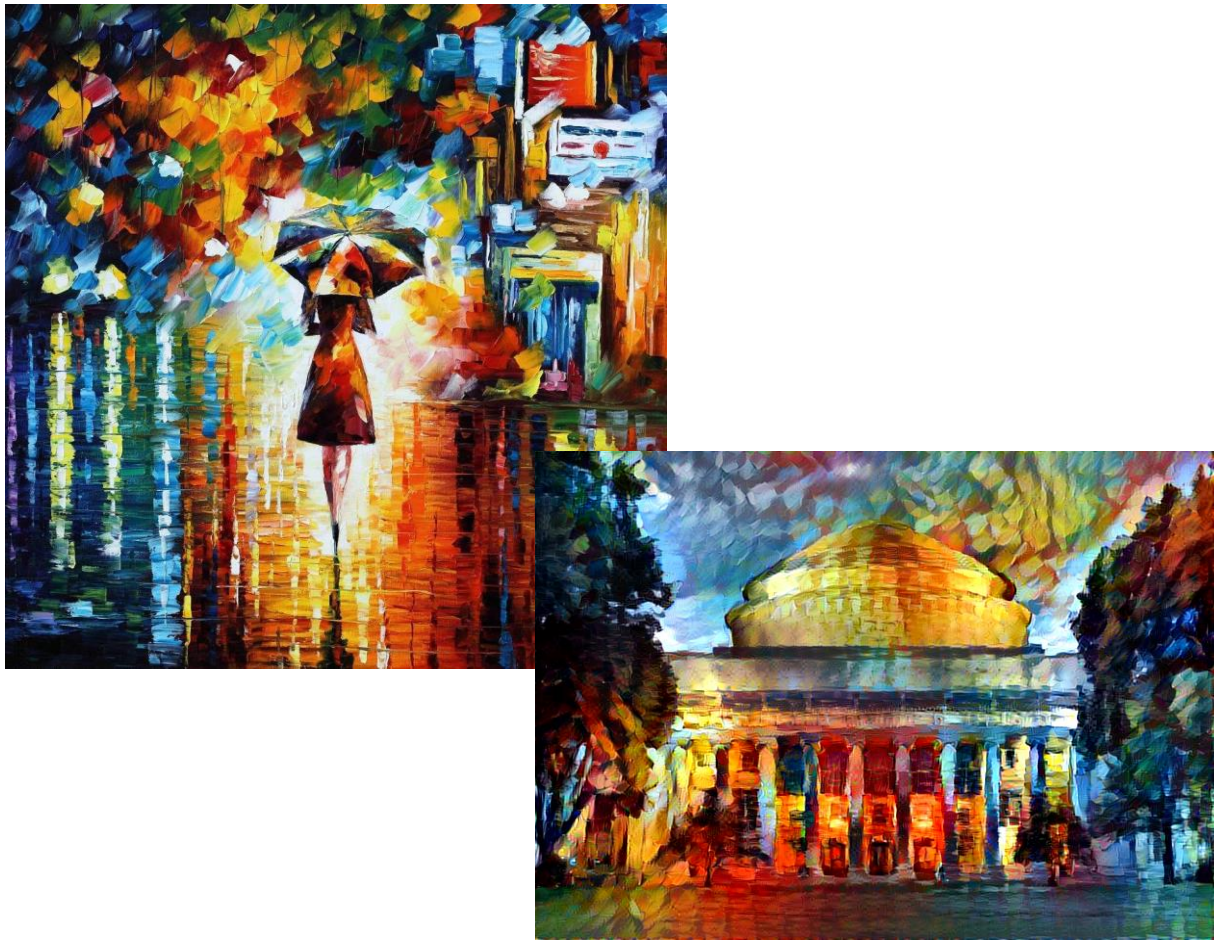
Deep Learning

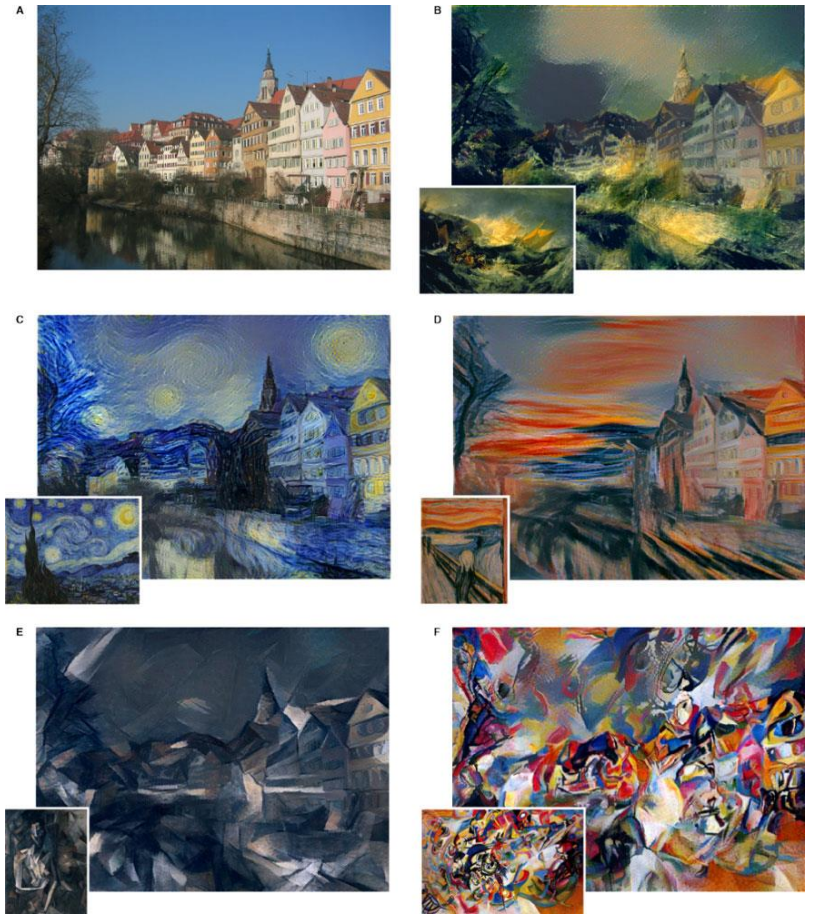
- "Deep learning" is simply achieved by adding additional hidden layers to a neural network
- Learns a complicated function from data
- Each layer is a trainable mathematical function





Endless Possibilities





Endless Possibilities

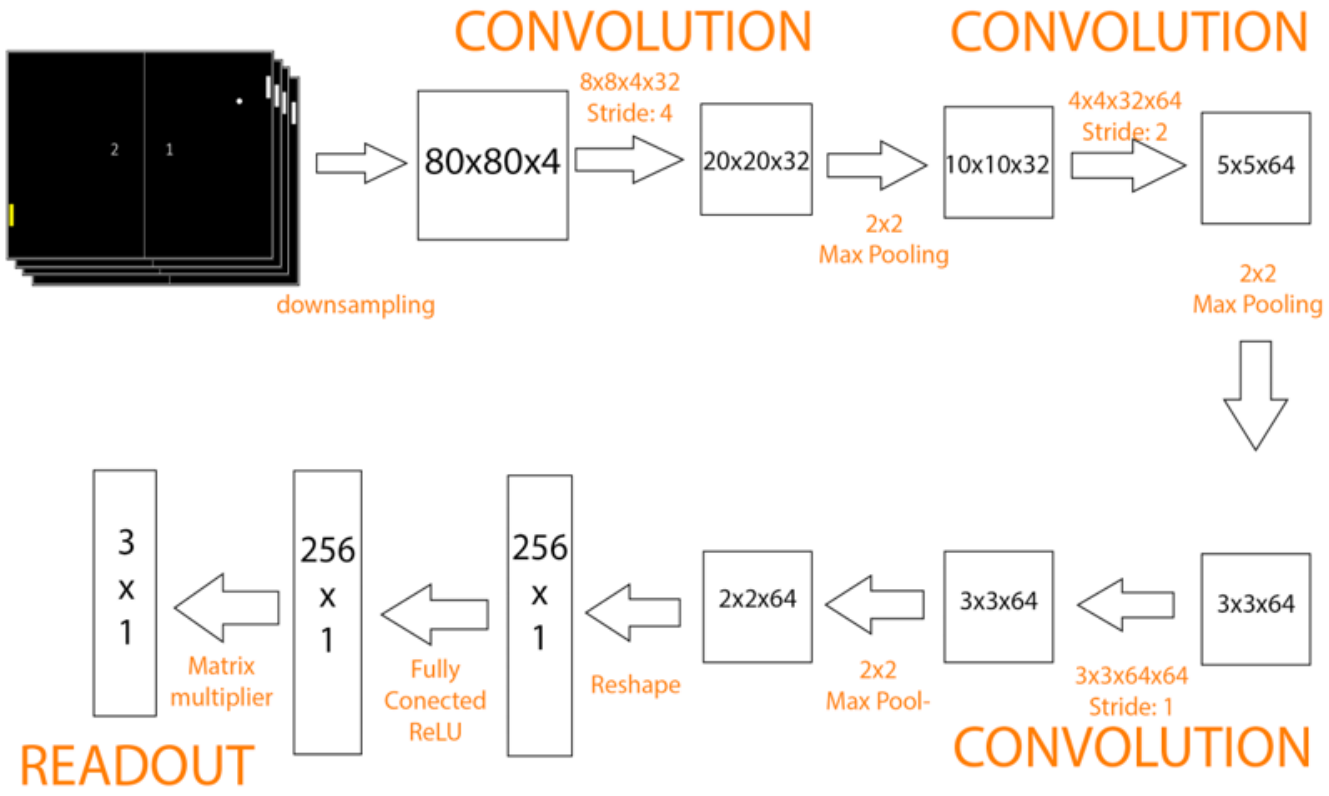


Figure 1: Deep Q Network architecture diagram. Note that unlike Minh et al [1], we perform max pooling on the convolutions. In the case of Pong, shown here, the output layer is three dimensional, as there are three valid actions: move up, move down, and do nothing.

Neural Networks Aren't New

- Many of the techniques that are successful now were developed 20-30 years ago
- What changed? Now we have:
  - Sufficient computational resources
  - Large enough interesting datasets
- Use of large-scale parallelism lets us look ahead many generations of hardware improvements, as well
- There are many network varieties, including recurrent networks where neuron connections form a directed cycle, and convolutional neural networks that use many identical copies of the same neuron

Data Distribution

- Neural network results typically improve with more data, bigger models, and more computation
- Recent image classification models have used the ImageNet dataset, containing 136GB of digital images
- Data sizes motivate a data parallel approach:
  - Distributed file system holds the data
  - A set of worker nodes processes different subsets of the data in parallel
  - Can shard large models across many processes, to increase the availability of network bandwidth when many workers are reading and updating simultaneously

Accelerator Support

- Machine learning algorithms often involve expensive computations with data dependencies
- General purpose GPUs are becoming more available: NVIDIA Titan X, now 1070/1080, Radeon 480
- Nvidia's cuDNN library for GPU-based neural network training accelerates popular image models 2-4x
- Special purpose accelerators also improve performance, including the Google Tensor Processing Unity

Trend: Much More Heterogenous Hardware

- General purpose CPU scaling has slowed significantly
- Specialization of hardware for certain workloads will be more important

Inference

- Using previously trained parameters to classify, recognize, and process unknown inputs
- Scalable and high-performance inference is a requirement for using models in production
- May need to produce results with low latency in an interactive service, or on a disconnected mobile device
- For large models, multiple nodes may participate in each inference computation, requiring distributed support
- We want a common, well-optimized system for both training and inference

Single-Machine Frameworks

- Machine learning algorithms often run on a single computer
- Caffe: framework for training declaratively specified convolutional neural networks that runs on multicore C/GPUs
- Theano: express a model as a dataflow graph, similar to TensorFlow as we'll touch on soon

Batch Dataflow Systems

- Spark extends MapReduce with the ability to cache previously computed datasets in memory, making it better suited for iterative machine learning algorithms
- However, it requires input data to be immutable, and for subcomputations to be deterministic
- This makes updating a machine learning model a heavy operation

Parameter Servers

- Unlike a key-value store, the write operation in a parameter server is specialized for parameter updates
- Can be efficient for training neural networks
- TensorFlow focuses on a high-level programming model so users can customize the code that runs in all parts of the system

History

- In 2011, Google Brain built DistBelief as their first proprietary machine learning system
- Focus on deep learning neural networks for use in Google Search, Photos, Maps, and more
- Codebase was refactored into a faster library, becoming TensorFlow
- TensorFlow was released open source on November 9, 2015
- As of June 2016, there were over 1500 Github repositories mentioning Tensorflow, only 5 of which were from Google

What is TensorFlow?

- TensorFlow is a machine learning system built to operate in heterogeneous environments
- Uses dataflow graphs to represent computation, shared state, and operations that mutate that state
- Maps nodes of a dataflow graph across many machines in a cluster, or within a machine across computational devices
- Models all data as tensors, dense n-dimensional arrays
- Features a Python and C++ frontend

Customizable

- Developed with the understanding that many researchers run smaller machine-learning tasks on single nodes with powerful GPUs
- Allows developers to experiment with node optimization for their particular use case
- Scales from large distributed clusters in a datacenter down to independent mobile devices

Runs on Variety of Platforms

phones

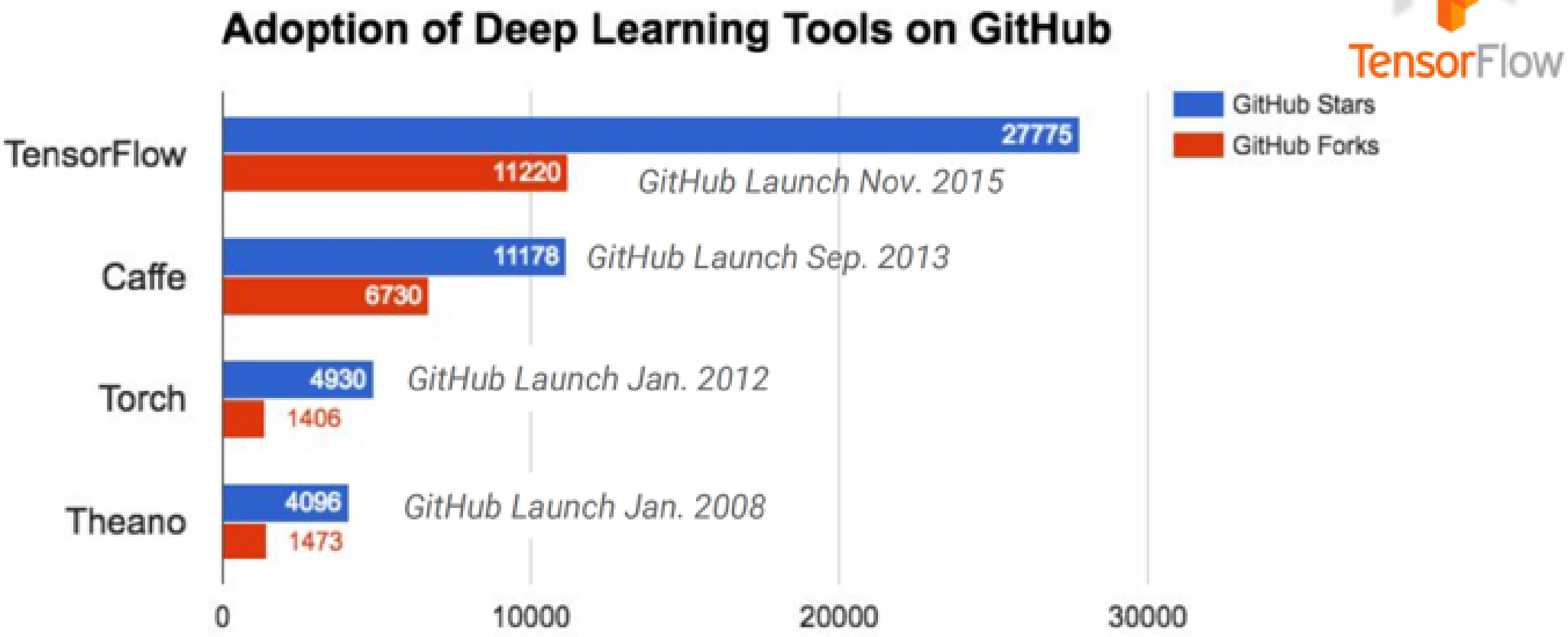
distributed systems of 100s of machines and/or GPU cards

single machines (CPU and/or GPUs) …

custom ML hardware

Fathom

Strong External Adoption



50,000+ binary installs in 72 hours, 500,000+ since November, 2015

Dataflow Programming

- Programming paradigm that models a program as a directed graph of the data flowing between operations
- Neural networks perform operations on multidimensional arrays referred to as tensors

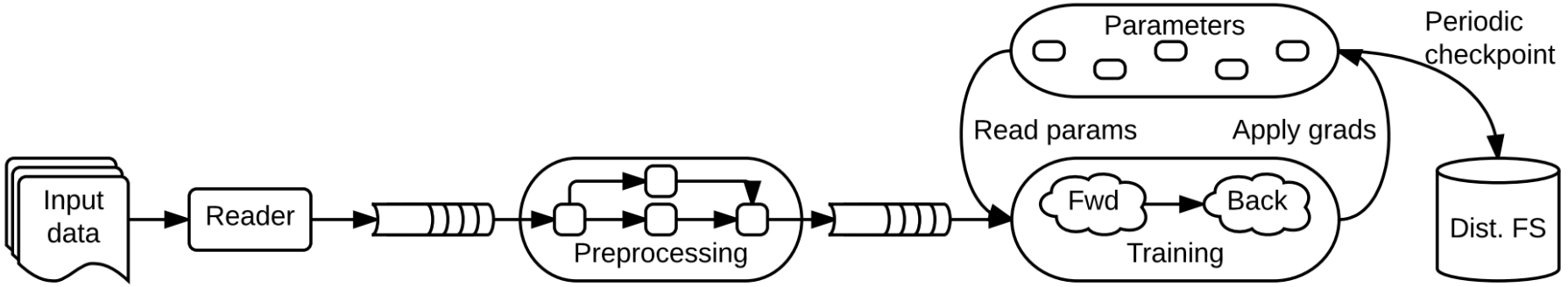


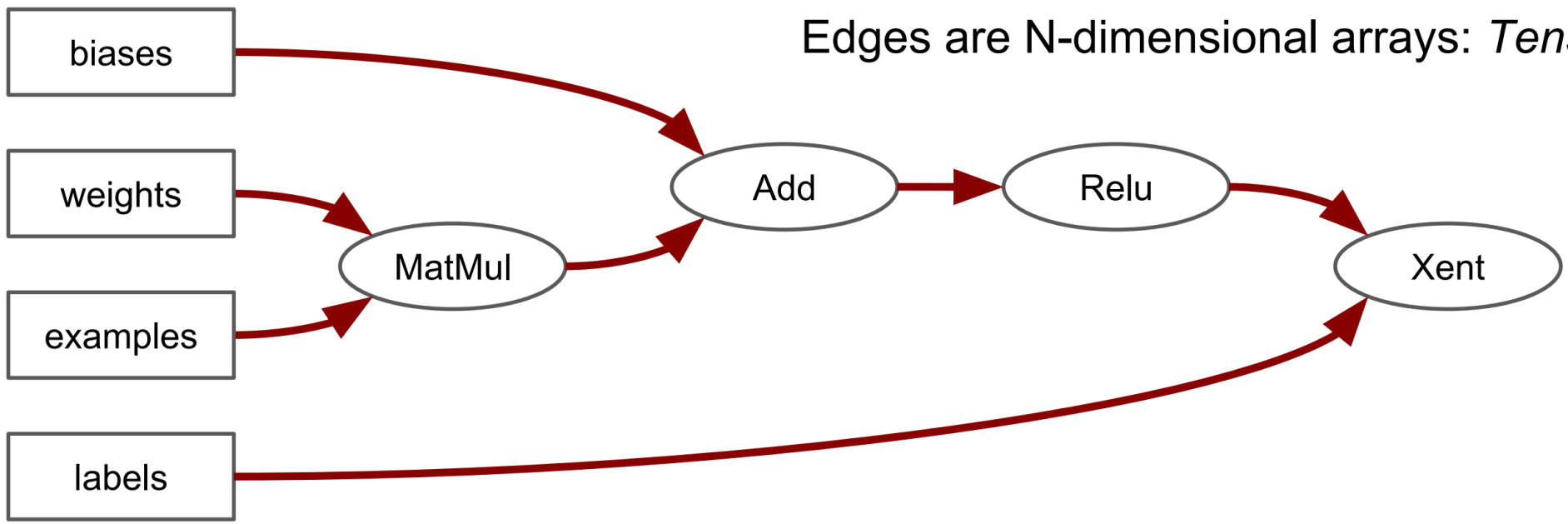
Figure 1: A schematic TensorFlow dataflow graph for a training pipeline contains subgraphs for reading input data, preprocessing, training, and checkpointing state.

Dataflow TensorFlow

- Differs from batch dataflow systems 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
- Mutable state is crucial when training very large models, so updates can be propagated to parallel training steps as quickly as possible

Computation is a dataflow graph

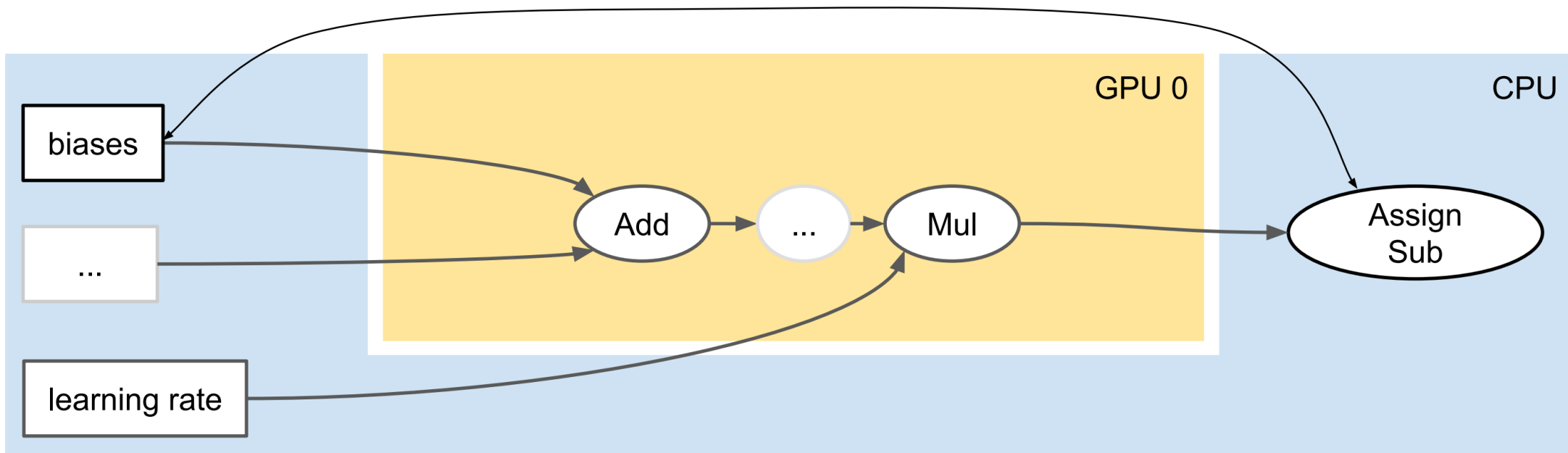
with tensors



Edges are N-dimensional arrays: *Tensors*

Computation is a dataflow graph

distributed



Tensors

- Dense n-dimensional arrays of primitive types
- All tensors are dense, ensuring the lower levels of the system can have simple implementations for memory allocation and serialization
- Sparse tensors are either encoded into variable-length string elements of a dense tensor, or a tuple of dense tensors

Operations

- Take in m many tensors as input, and output n tensors
- Has a type, such as Const or Assign
- The Const operation has no inputs and a single output
  - Const has an attribute T that determines the type of its output
  - An Attribute Value that determines the value that it produces

Stateful Operations

- A Variable operation owns a mutable buffer that is used to store the shared parameters of a model as it is trained
- A Variable has no inputs, and produces a reference handle, acting as a typed capability for reading and writing the buffer
- Includes several Queue operations including FIFOQueue, producing a reference handle that may be consumed by standard queue operations

Concurrent Execution

- TensorFlow's dataflow graph represents all possible computations in an application
- The user selects which subgraphs to execute, specifying which tensors to feed and fetch from the dataflow
- Each invocation of the API is a step, and concurrent steps on the same graph may occur where stateful operations enable coordination between steps

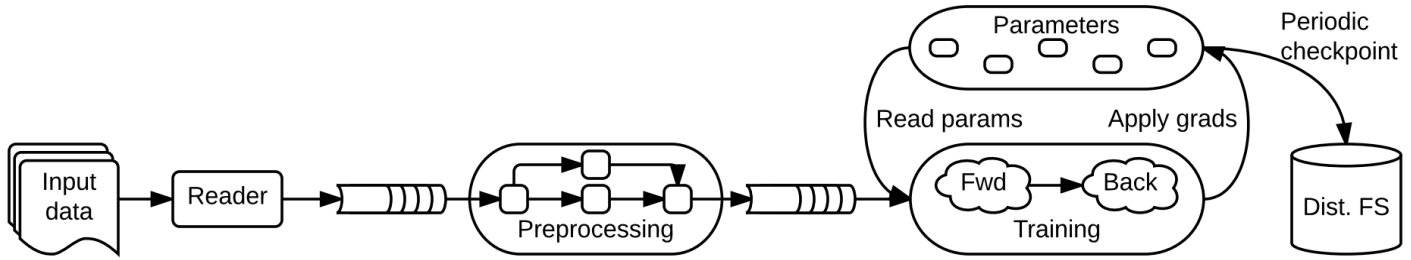


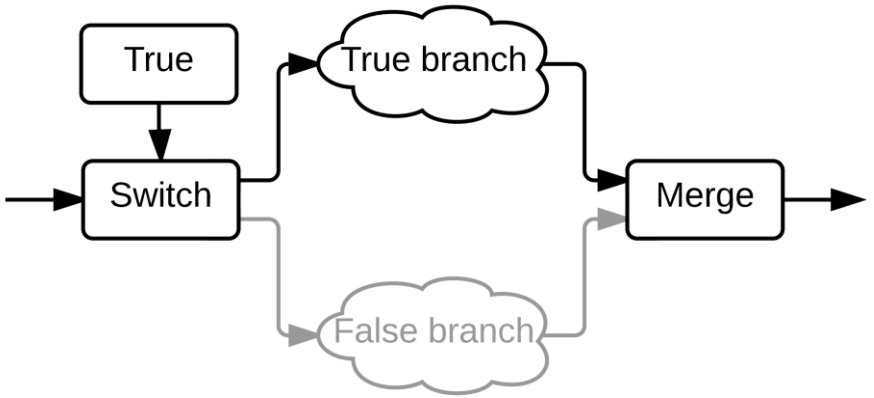
Figure 1: A schematic TensorFlow dataflow graph for a training pipeline contains subgraphs for reading input data, preprocessing, training, and checkpointing state.

Distributed Execution

- 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
- The TensorFlow runtime places operations on runtimes, and users may specify device preferences
- Once a subgraph has been computed for a step, operations are partitioned into per-device subgraphs, with send and receive operations connecting the device boundaries

Dynamic Control Flow

- Most TensorFlow evaluations are strict, where all inputs to an operation must be computed before running the operation
- It also supports conditional control flow using:
  - Switch: takes a data and control input, and uses the control input to select which of its two inputs should produce a value, giving the input not taken a dead value
  - Merge: forwards at most one non-dead input to its output, or produces a dead output if both of its inputs are dead

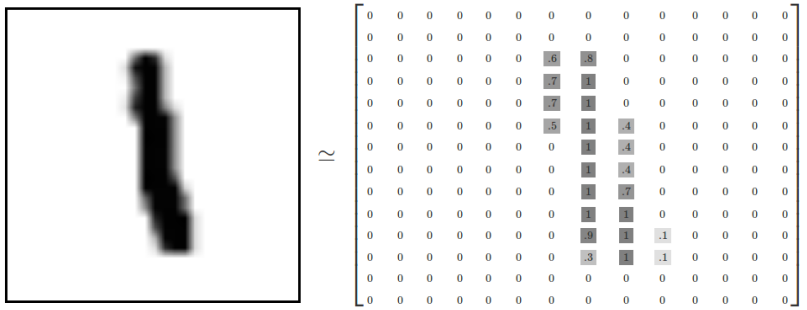


MNIST Tutorial Walkthrough

- MNIST is a simple computer vision dataset, with images of handwritten digits and labels for each image
- We want to create a function that is a model for recognizing digits, based on looking at every pixel in the image
- We can train the model by having it look at thousands of examples

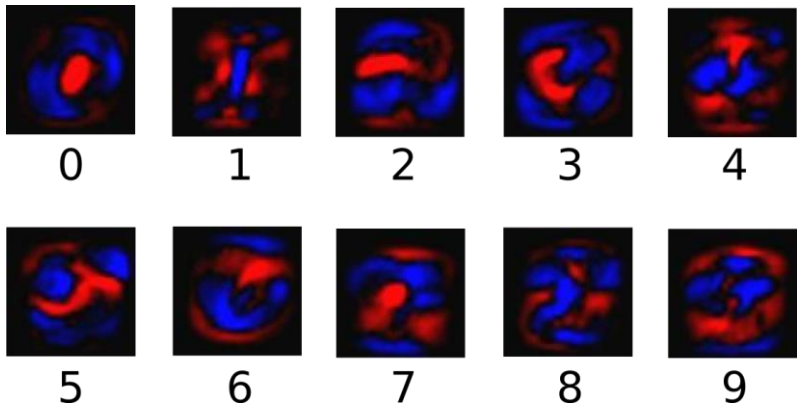
MNIST Walkthrough

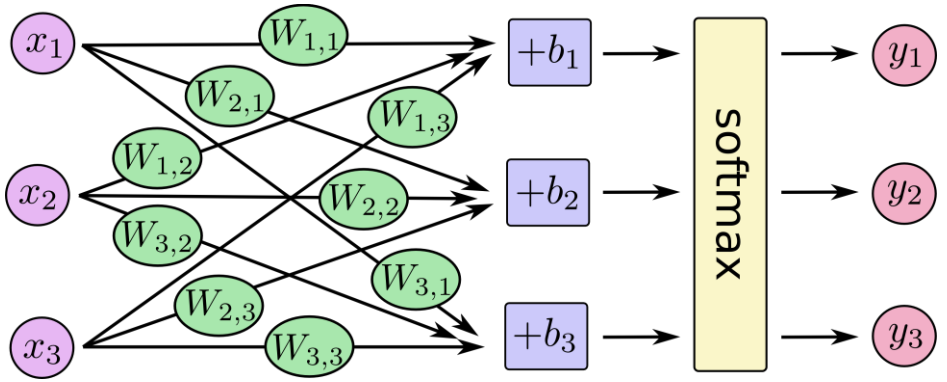
- Each image is 28x28 pixels, which we can interpret as a big array of numbers
- We flatten this array into a vector of 784 numbers
- Resultant mnist.train.images is a tensor with a shape of [55000, 784], the first index selecting the specific image
- Each label will be represented as a one-hot vector, so 3 would be [0,0,0,1,0...], thus mnist.train.labels is a [55000, 10] array



MNIST Walkthrough

- We want to look at an image and give the probabilities for it being each digit
- With softmax regression, we add up the evidence of our input being certain classes through a weighted sum of the pixel intensities, and converting that evidence into probabilities





Let's Implement!

- To do efficient computing in Python, you typically use libraries like NumPy that do expensive operations using highly efficient code implemented in another language
- TensorFlow avoids overhead by letting us describe a graph of interacting operations that run entirely outside Python

```
import tensorflow as tf
```

```
x = tf.placeholder(tf.float32, [None, 784])
```

```
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

X is a placeholder, a value we'll input when asking TF to run a computation.
We could imagine treating weights and biases as additional inputs, but TF can use Variables, modifiable tensors living in TF's graph of interacting operations.

Let's Implement!

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

```
y_ = tf.placeholder(tf.float32, [None, 10])
```

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

```
init = tf.initialize_all_variables()
```

```
sess = tf.Session()
sess.run(init)
```

- Our model only takes one line to define
- Cross-entropy is used to determine the loss of the model
- To implement this, we need a new placeholder to input the correct answers
- Tensorflow knows the entire graph of your computations, so it can automatically use backpropagation to determine how your variables affect your loss minimization

Training and evaluation

```
for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

- Each step of the loop, we get a "batch" of one hundred random data points from our training set
- Using small batches of random data is called stochastic training, using a different subset each time is less expensive and has much of the same benefit
- Check which number received the highest probability estimate and compare it to the truth
- Determine the mean accuracy in print
- For this program, taking only 16 lines of code, we get an accuracy of 92%

Extensibility case studies

Four extensions to TensorFlow built using simple dataflow primitives and “user-level” code:

- Differentiation and Optimization
- Handling very large models
- Fault tolerance
- Synchronous replica coordination

Differentiation and Optimization

- SGD: compute the gradients of a cost function with respect to some parameters, then updating the parameters based on those gradients.
- User-level library for TensorFlow that automatically differentiates expressions (provide model and loss function)
- Many implemented optimizations like batch normalization, and gradient clipping to accelerate training and make it more robust
- TensorFlow users can also experiment with a wide range of *optimization algorithms*

Auto-differentiation and TensorFlow Gradient Computation

- tf.train.Optimizer creates an optimizer.
- tf.train.Optimizer.minimize(loss, var_list)

  - Base class for optimizers to train a model.
  - Adds optimization operation to computation graph
  - Calling minimize() takes care of both computing the gradients and applying them to the variables.
  - Many subclasses as in the pic

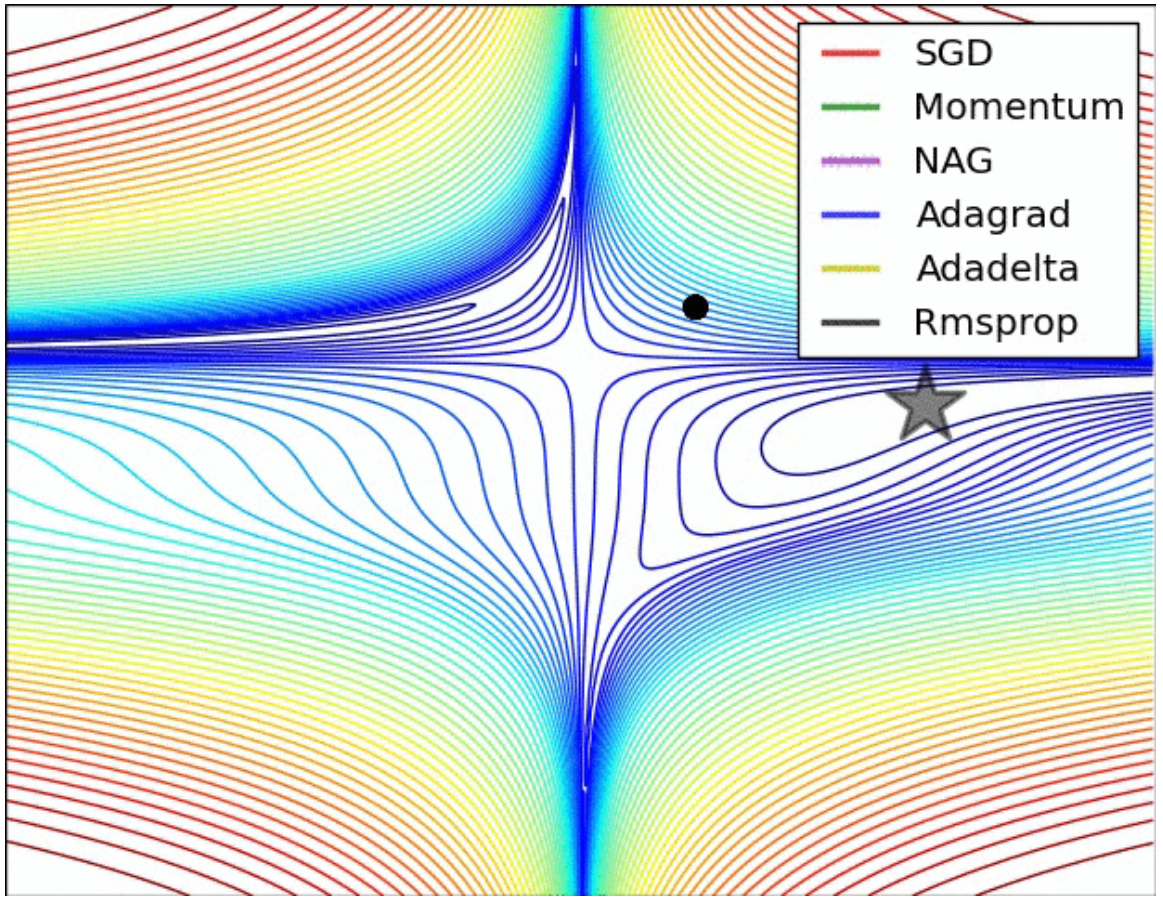
- class tf.train.GradientDescentOptimizer
- class tf.train.AdadeltaOptimizer
- class tf.train.AdagradOptimizer
- class tf.train.AdagradDAOptimizer
- class tf.train.MomentumOptimizer
- class tf.train.AdamOptimizer
- class tf.train.FtrlOptimizer
- class tf.train.RMSPropOptimizer

Optimization Algorithms

- **SGD**
- **Momentum** (accumulates a “velocity” for each parameter based on its gradient over multiple iterations, then computes the parameter up- date from that accumulation)
- **NAG** (Nesterov accelerated gradient)
- **Adagrad**
- **Adadelta**
- **Rmsprop**

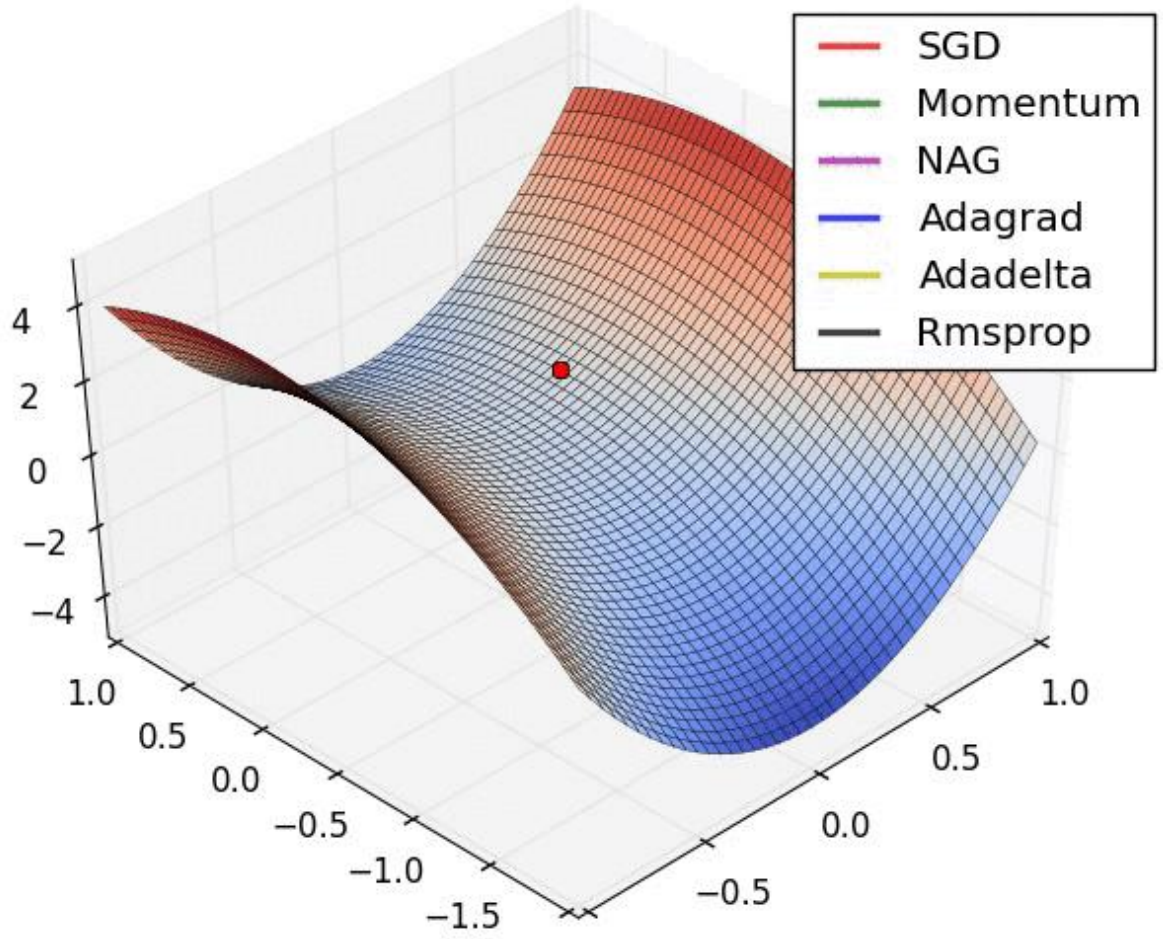
- Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge similarly fast
- Momentum and NAG are led off-track, evoking the image of a ball rolling down the hill.
- NAG, however, is quickly able to correct its course due to its increased responsiveness by looking ahead and heads to the minimum.

Behavior on the contours of a loss surface over time



Behavior of the algorithms at a saddle point

- Saddle point, i.e. a point where one dimension has a positive slope, while the other dimension has a negative slope
- Pose a difficulty for **SGD**
- **SGD, Momentum,** and **NAG** find it difficulty to break symmetry
- **Momentum** and **NAG** eventually manage to escape the saddle point
- **Adagrad, RMSprop,** and **Adadelta** quickly head down the negative slope



Handling very large models

- ***Distributed representation****:* to train a high-dimensional data (embeds a training example as a pattern of activity across several neurons which can be learned by backpropagation)
- **Example**: language model
  - Training example: a sparse vector with non-zero entries corresponding to the IDs of words in a vocabulary
  - Distributed representation for each word: a lower-dimensional vector
  - Inference: multiplying a batch of **b** sparse vectors against an **n × d** *embedding matrix*,

  (**n** is the number of words in the vocabulary,

  b is the desired dimensionality, to produce a much smaller **b × d** dense matrix representation)

Sparse embedding layers in the TensorFlow graph

- Graph: an embedding layer that is split across two parameter server task
- **Gather**: extracts a sparse set of rows from a tensor, and TensorFlow colocates this operation with the variable on which it operates
- **Part**: divides the incoming indices into variable-sized tensors that contain the indices destined for each shard
- **Stich**: 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
  - The result is a set of sparse update operations that act on just the values that were originally gathered from each of the shards

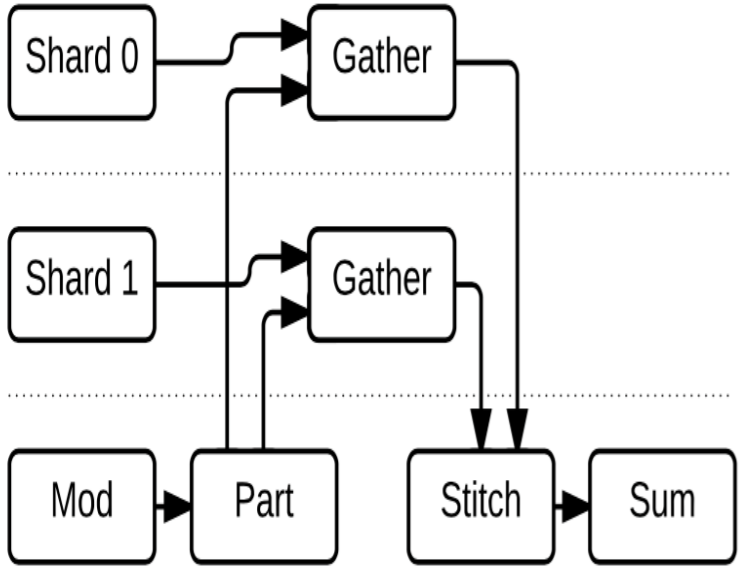


Figure 3: Schematic dataflow graph for a sparse embedding layer containing a two-way sharded embedding matrix.

Fault tolerance

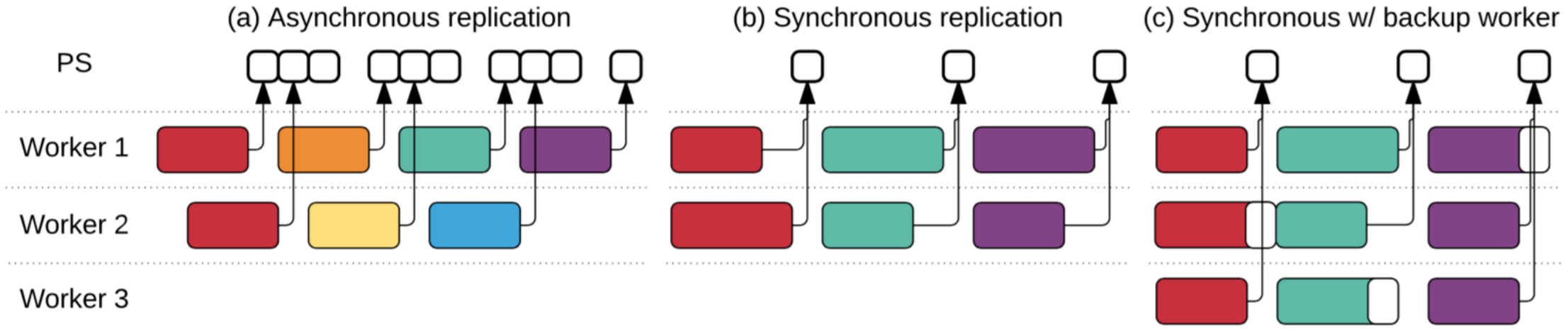
- TensorFlow job is likely to experience failure during the training process
- Require some form of fault tolerance.
- However, failures are unlikely to be so common that individual operations need fault tolerance, so a mechanism like Spark's RDDs would impose significant overhead for little benefit.
- 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.

User-level checkpointing for fault tolerance

- **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.

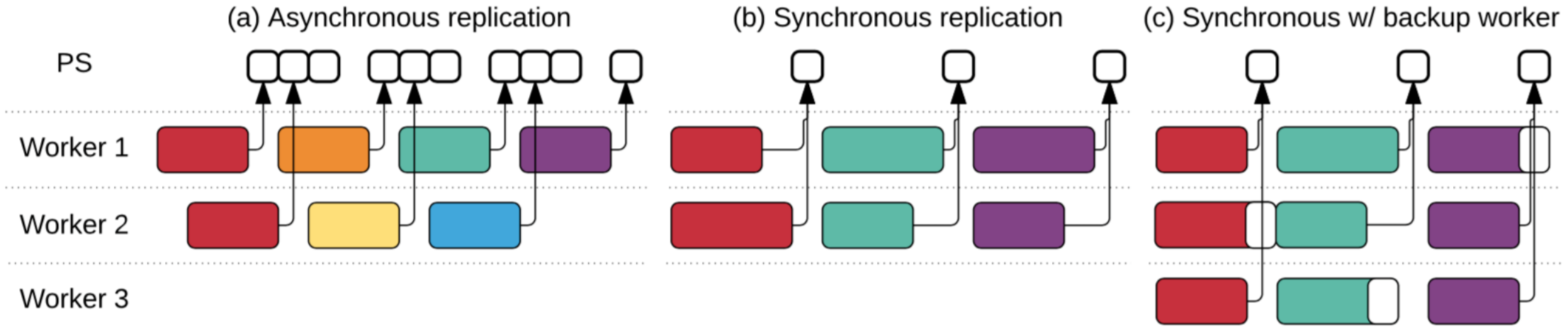
Synchronous replica coordination

- Revise the assumption that *synchronous* training does not scale
- Since GPUs enable training with hundreds of machines, it may be possible to train a model synchronously in less time than asynchronous training on the same machines.
- TensorFlow is designed for asynchronous training, experimenting with synchronous methods has begun.
- The TensorFlow graph enables users to change how parameters are read and written when training a model
- Three alternatives are implemented.



(a)

- Each worker reads the current value when the step begins, and applies its gradient to the different current value at the end
- This ensures high utilization, but the individual steps use stale information, making each step less effective.
- A blocking queue acts as a barrier to ensure that all workers read the same parameter version, and a second queue accumulates multiple gradient updates in order to apply them atomically



(b)

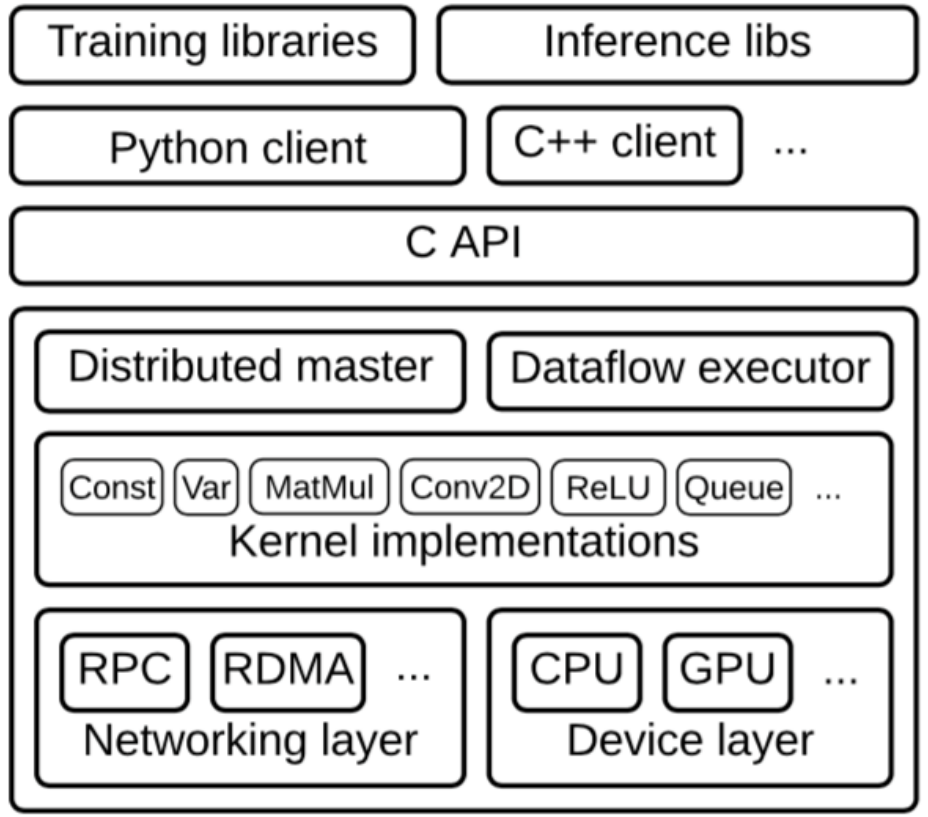
- Accumulates updates from all workers before applying them, but slow workers limit overall throughput.

(C)

- To mitigate stragglers, we implement *backup workers*, which are similar to MapReduce backup tasks.
- 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, so each worker processes a different random batch.

Implementation

- Core TensorFlow library is implemented in C++
- It runs on several operating systems including Linux, Mac OS X, Android, and iOS
- The implementation is open-source
- ***Distributed master*** translates user requests into execution across a set of tasks
- ***Dataflow executor:***
  - In each task, handles requests from the master, and schedules the execution of the kernels that comprise a local subgraph.
  - Dispatches kernels to local devices and runs kernels in parallel when possible: e.g., by using multiple cores in a CPU device, or multiple streams on a GPU



Evaluation Outline

- Training step times for convolutional models compared across libraries (Caffe, Neon, Torch, and TensorFlow)
- Synchronous replication benchmarks with respect to the number of workers and size of models
- Image classification: Training throughput with respect to asynchronous and synchronous replication
- Text language modeling throughput with an LSTM network

Single-machine Benchmarks

- Table: Step times for training four convolutional models with different libraries, using one GPU
- Specs: A six-core Intel Core i7-5930K CPU at 3.5 GHz and an NVIDIA Titan X GPU
- TensorFlow achieves shorter step times than Caffe, and performance within 6% of the latest version of Torch
- TF and Torch implement convolution and pooling operations similarly
- Neon uses using hand-optimized convolutional kernels implemented in assembly language

	Training step time (ms)			
Library	AlexNet	Overfeat	OxfordNet	GoogleNet
Caffe [36]	324	823	1068	1935
Neon [56]	87	**211**	**320**	**270**
Torch [17]	**81**	268	529	470
TensorFlow	**81**	279	540	445

Synchronous replica benchmark

- Coordination implementation is the limiting factor for scaling with more machines
- Figure: Baseline training step throughput for synchronous replication with null updates
- 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
- For the scalar curve, only a single 4-byte value is fetched from each PS task
- Dense curves show the performance of a null step when the worker fetches the entire model
- The Sparse curves show the throughput of the embedding lookup operation: Each worker reads 32 randomly selected entries from a large embedding matrix
- Sparse accesses allow handling of larger models

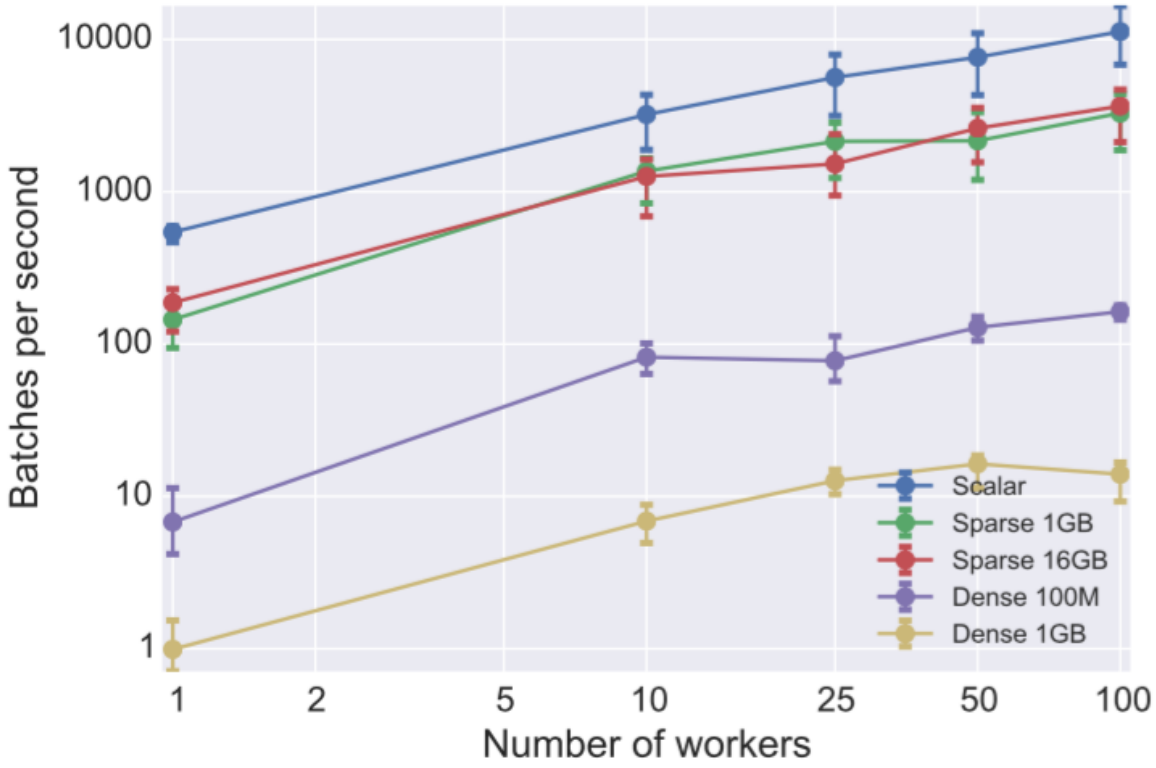
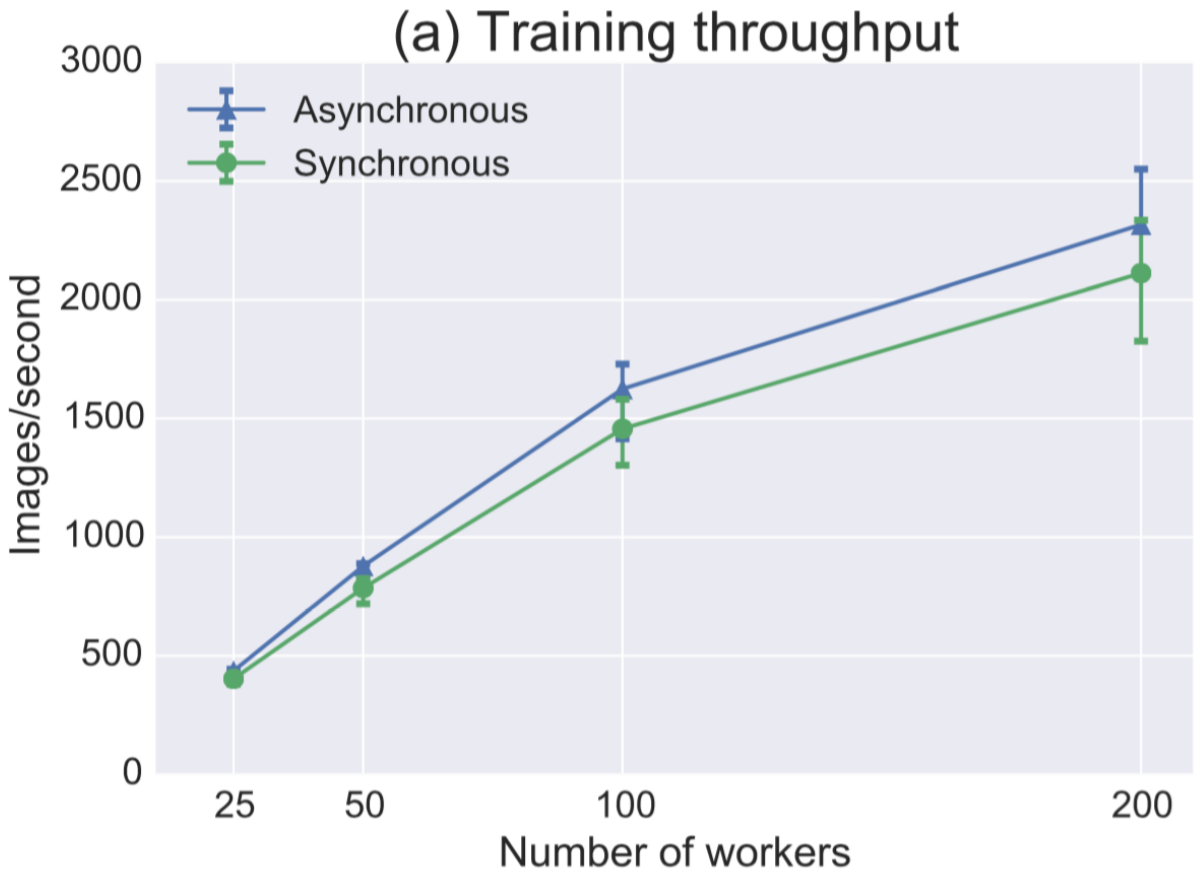
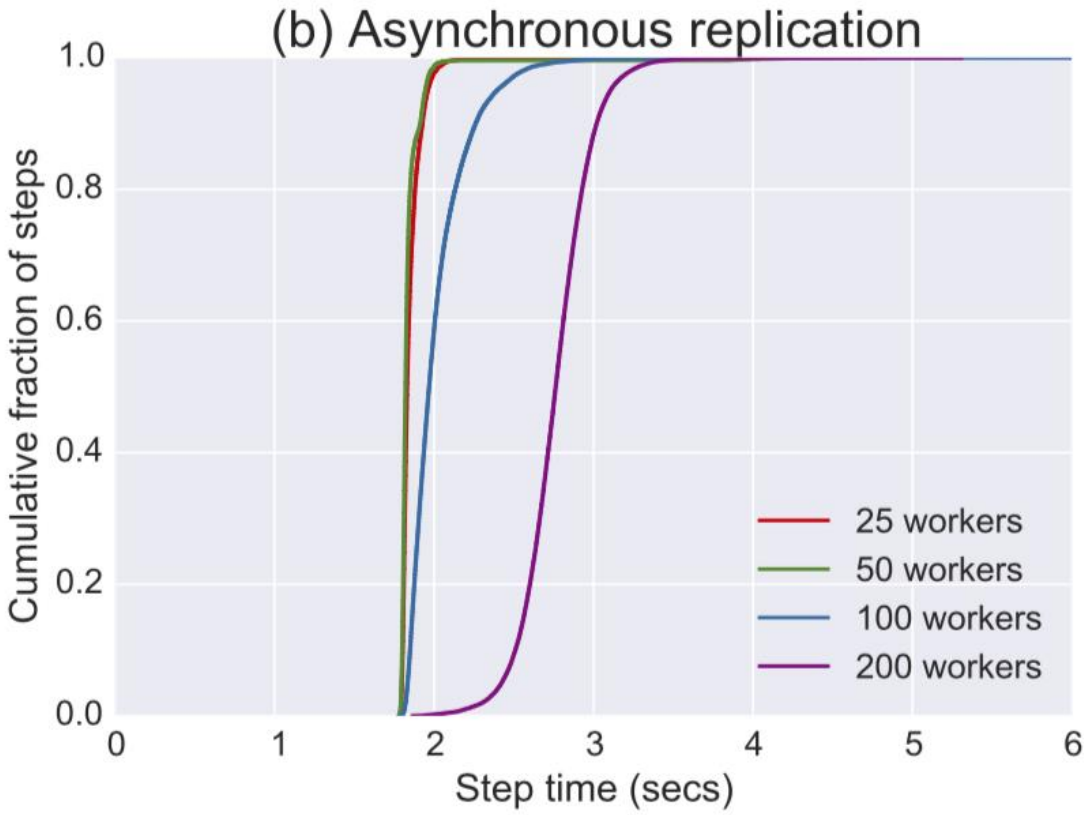
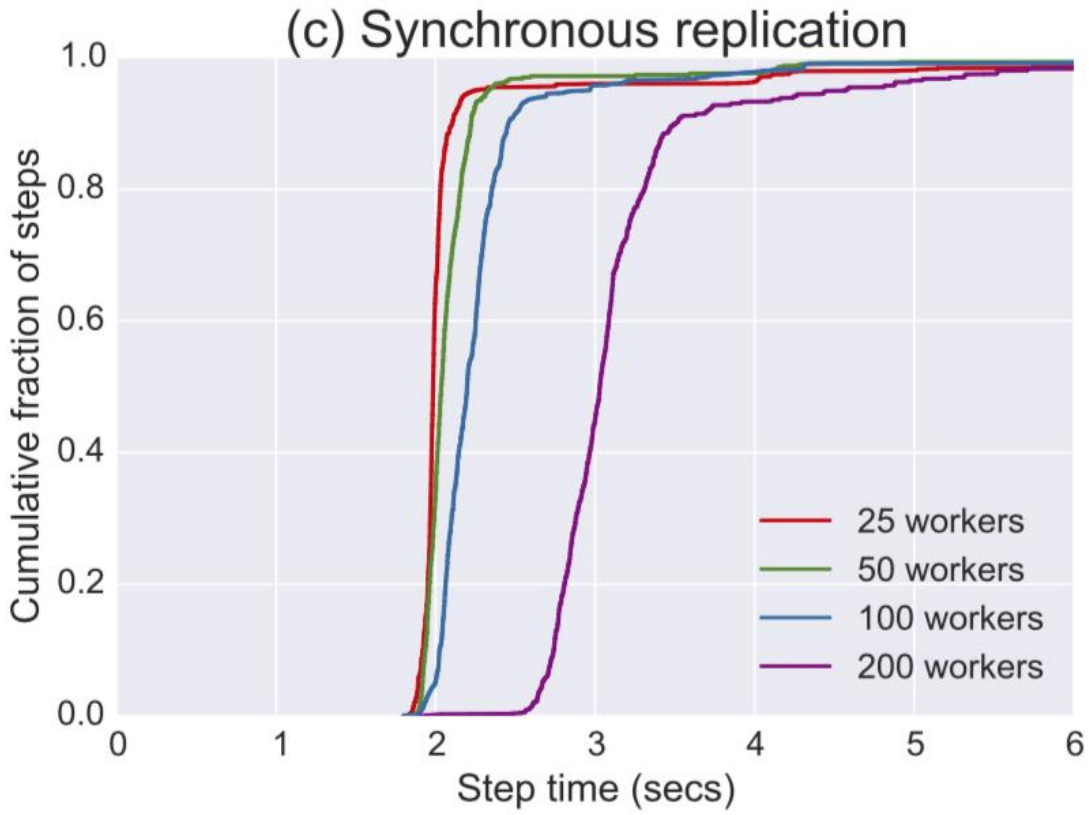


Image Classification

- Google's Inception-v3 model
- 78.8% on ILSVRC 2012
- Assessing the effect of synchronous vs asynchronous coordination on training
- Throughput increases with up to 200 workers
- Diminishing returns because of PS task contention







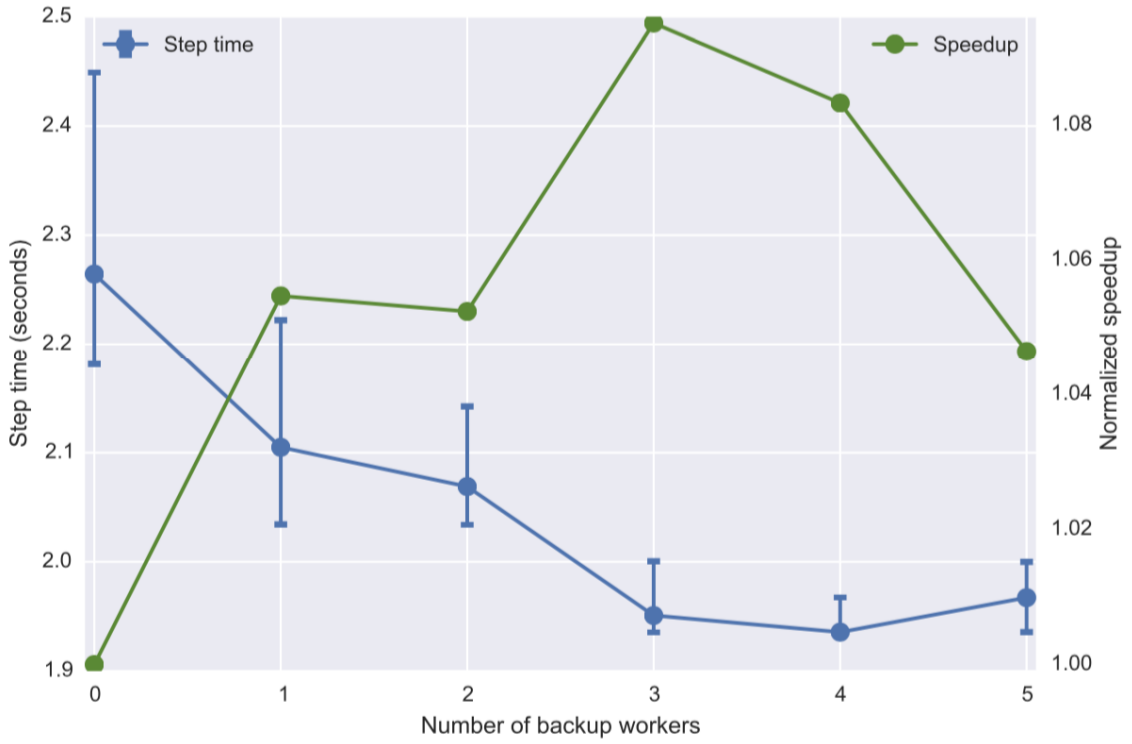
- However, adding more workers gets diminishing returns because of step time increases
- Adding workers increases step time because there is more contention on the PS tasks

Asynchronous vs Synchronous

- Synchronous steps are always longer than asynchronous steps, because all workers must wait for the slowest worker to catch up
- Slowest synchronous steps are way longer than asynchronous steps
- Stragglers disproportionately impact the tail
- To mitigate this:
  - Add backup workers
  - Step completes when the first m of n tasks produce gradients

Adding Backup workers

- Figure: # of Backup workers vs Step time and speedup normalized w.r.t. the total resources used
- 4 backup workers give shortest step time
- 3 backup workers are most efficient with respect to the total resources used
- Adding a fifth backup worker slightly degrades performance, because the 51st worker (i.e., the first whose result is dis- carded) is more likely to be a non-straggler that generates more incoming traffic for the PS tasks

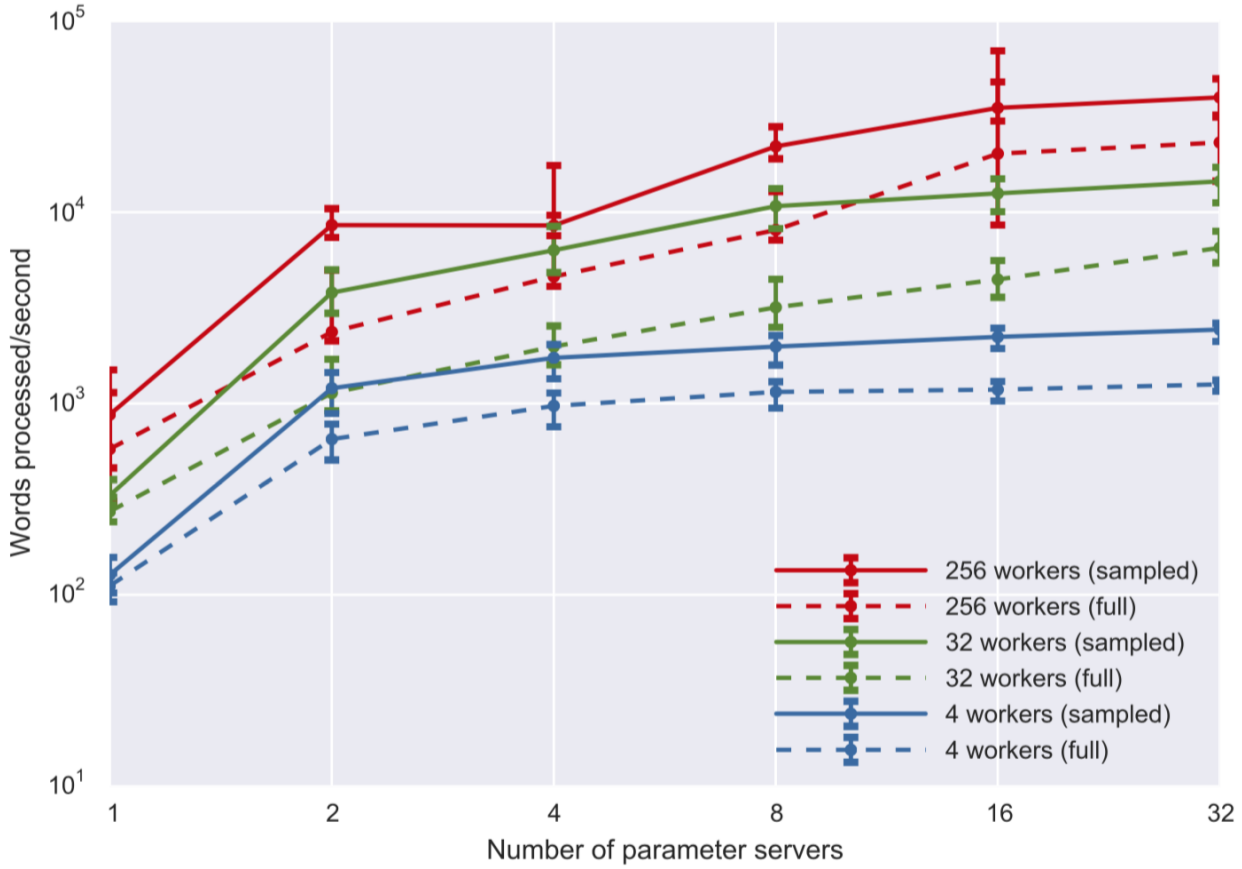


Language Modeling

- Given a sequence of words, a language model predicts the most probable next word
- Here, an LSTM network is trained on text in the One Billion Word benchmark
- Vocabulary size limits training performance (makes number of parameters large)
- Use a restricted vocabulary, 40,000 common words

PS tasks vs Throughput

- Increasing the number of PS tasks leads to increased throughput for language model training, by parallelizing the softmax computation
- Sampled softmax multiplies the output by a random sparse matrix containing weights for the true class and a random sample of false classes
  - Increases throughput by performing less computation
- Adding a second PS task is more effective than increasing from 4 to 32, or 32 to 256 workers



Conclusions

- Allows users to harness large-scale heterogeneous systems
- Good for both experimentation and production
- Large amount of use and adoption (8,000 forks and 500,000 downloads)
- Flexible dataflow representation, excellent performance
- Looking to improve implementations for mutable state and fault tolerance for applications that require stronger consistency
- Looking to develop a dynamic dataflow graph (to use in e.g. reinforcement learning)