TensorFlow: A System for Large-Scale Machine Learning

Abadi et al. 2015

Rohit Agrawal
CS598APK, Fall 2018
What is Tensorflow?

- An open source Machine Learning system operating at large scale and in heterogeneous environments.
- Uses dataflow graphs to represent:
  - Computations
  - Shared state
- Unifies computation and state management.
  - Provides flexibility to support experimentation into new ML models and system-level optimizations.
- C++ backend.
- C++ and Python frontend.
- Focused on training and inference of neural networks.
Neural Network

• A directed graph of *layers*.
• A layer can be different composition of mathematical operators.
  • Fully connected layer – $Ax + b$
  • Non-linearity – Element-wise non-linear function such as sigmoid, ReLu etc.
  • Pooling – Max, Min, Avg etc.
• Typically terminates with a loss function.
  • Quantifies the difference between ground truth and predicted value.
• Parameters of different layers are *learnt* during training.
  • A learning algorithm updates the parameters to minimize the loss.
• Learnt parameters are used to perform *inference* from unknown data points.
A brief history: DistBelief (2011)

- Predecessor to Tensorflow.
- Uses parameter server architecture.
- A stateless worker process – performs computations.
- A stateful parameter server – maintains current version of model parameters.
- Worker process compute gradient independently and write back delta updates to each parameter server which combines the updates with its current state.
Limitations of DistBelief

- Pre-defined layers in C++ and Python based scripting interface.
  - Difficult to experiment with new layer architectures in less familiar C++ language.
- Experimenting with new optimization methods apart from vanilla SGD required modifying the parameter server implementation.
- Execution pattern of DistBelief fails for RNNs (contains loops), Adversarial Networks, Expectation Maximization and other traditional ML algorithms.
Design Principles

• Dataflow based programming abstraction.

• Represents individual mathematical operators as nodes.
  • Easier to define new layers as a combination of these nodes.
  • Eg: Matrix Multiplication, convolution etc.

• Deferred execution in two phases:
  • Define the program as a dataflow graph with placeholders for input and states.
  • Optimize the graph according to available devices and defer the execution until entire program is available.

• A common abstraction to support CPUs, GPUs and custom ASICs (TPUs).
Programming Model

• A single dataflow graph to represent all states and computations including parameters, their update rules, mathematical operations, and input processing.

• Supports multiple concurrent executions on overlapping subgraphs of the overall graph.

• Mutable state of individual vertices can be shared between different execution of the graph.
  • makes in place update of large parameters possible so that the updates can be propagated as soon as possible.
  • parameter server with additional flexibility – execute arbitrary dataflow subgraphs on machines with shared model parameters.
Variables

- No inputs.
- Owns a mutable buffer that stores a shared parameter.
- Produces a reference handle to read and write the buffer.

```python
b = tf.Variable(tf.zeros([100]))  # 100-d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100],-1,1))  # 784x100 matrix w/rnd vals
x = tf.placeholder(name="x")  # Placeholder for input
```
Tensors

- N-dimensional array or list of int32, float32, double, string etc.
- Represents inputs to and results of the mathematical operators.
- All tensors are dense at the lowest level.

```python
b = tf.Variable(tf.zeros([100]))  # 100-d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100],-1,1))  # 784x100 matrix w/rnd vals
x = tf.placeholder(name="x")  # Placeholder for input
```
Operations and Kernels

• Takes $\geq 0$ tensors as input and produces $\geq 0$ tensors as outputs.
• Polymorphic – supports multiple data types.
• Kernel – An implementation of an operation that can be run on a device.

\[
\text{ReLU} = \text{tf.nn.relu} (\text{tf.matmul}(W, x) + b) \quad \# \text{ReLU}(Wx+b)
\]
Sessions

• A way for programs to interact with tensorflow system.
• Session creates an empty graph.
• Nodes and edges can be added with an Extend method.
• Output nodes can be computed by the Run operation.

```
s = tf.Session()                      # Creates a session
    ....
result = s.run(<some graph>)
```
Example

ReLu(Wx + b)

import tensorflow as tf

b = tf.Variable(tf.zeros([100]))

W = tf.Variable(tf.random_uniform([784, 100], -1, 1))

x = tf.placeholder(name="x")

relu = tf.nn.relu(tf.matmul(W, x) + b)

C = [...]

s = tf.Session()

for step in xrange(0, 10):
    input = ...construct 100-D input array ...
    result = s.run(C, feed_dict={x: input})
    print step, result
Example continued...

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

- Built-in support for automatic gradient computation.
- BFS to find all backward paths from target (loss function) to parameter and add a node corresponding to each operation.
- Sum partial gradient along each path.
- Extensive memory usage due to re-use data for gradient computations.
- Active area of improvement.
Implementation

- Runs on windows, linux, Mac OS X, Android and iOS, and x86, ARM CPUs and Kepler, Maxwell and Pascal GPUs.
- C API separates user-level code from core runtime.
- *Distributed master* translates user requests into execution across a set of tasks.
- *Dataflow executor* in each task handles requests from *master* and schedules execution of kernels to local devices.
Single-device Execution

- Order of execution respects dependencies.
- Count of dependencies per node is kept.
- Ready nodes are pushed into a *Ready Queue* and are processed in some unspecified order.
Multi-device Execution

- Two important decisions
  - Node placement
  - Cross-device communication
Multi-device Execution: Node placement

- Cost model based on input and output size and computation time for a device.
- Can be heuristic or based on measured time for previous placement decisions.
- Device for a node is selected greedily.
- Load balancing might be a problem in synchronous execution.
- Placement algorithm is an area of ongoing development.
Multi-device Execution: Cross-device Communication

• Communication using explicit `send` and `receive` nodes.
• Single receive node for all users of a tensor – avoids multiple transmission.
• Takes care of scheduling and makes the system scalable.

![Diagram showing multi-device execution process]
Partial Execution

- Running just a subgraph of the entire execution graph.
- Exact subgraph can be run using the name of input and output nodes.
- Graph is changed to add *feed* and *fetch* nodes for input and output respectively.
Control Flow

• Primitives similar to [1] to handle control flow.
• Switch (multiplixer) and Merge (demultiplexer) allow conditional execution of entire subgraph.
• Enter, Leave and NextIteration for expressing iterations.
• Tags and Frames similar to MIT Tagged Token Machine [2] to allow simultaneous execution of multiple iterations.

Training

• Asynchronous
  • Each worker updates the parameters asynchronously.
  • High throughput.

• Synchronous
  • A blocking queue for workers to read same parameters.
  • Slow workers limit overall throughput.

• Synchronous w/ backup
  • Aggregates first $m$ of $n$ updates produced.
  • Improves throughput by 10% in image classification.
Evaluation
Single Machine Execution

- Performs within 6% of latest version of Torch – both use same cuDNN library.
- Neon outperforms due to hand optimized kernels written in assembly.

<table>
<thead>
<tr>
<th>Library</th>
<th>AlexNet</th>
<th>Overfeat</th>
<th>OxfordNet</th>
<th>GoogleNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe [38]</td>
<td>324</td>
<td>823</td>
<td>1068</td>
<td>1935</td>
</tr>
<tr>
<td>Neon [58]</td>
<td>87</td>
<td>211</td>
<td>320</td>
<td>270</td>
</tr>
<tr>
<td>Torch [17]</td>
<td>81</td>
<td>268</td>
<td>529</td>
<td>470</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>81</td>
<td>279</td>
<td>540</td>
<td>445</td>
</tr>
</tbody>
</table>

Chintala benchmark of convolutional model on Intel Core i7-5930K CPU at 3.5GHz and Nvidia Titan X GPU
Image Classification

• Inception-v3 model with 7 PS tasks and varying worker tasks.
• Intel Xeon E5 servers with Nvidia K80 GPUs.
Language Modeling

- LSTM-512-512 on Billion Word benchmark.
- Vocabulary size limits training performance – limited to 40k words instead of 800k.
Conclusions

• Offers a set of uniform abstractions for harnessing large-scale heterogeneous systems for both production and experimentation.
• Enables *power* users to achieve excellent performance.
• Default policies that work for *all* users yet to be determined.
• Placement algorithm, memory management and scheduling are being actively improved.
• Static dataflow graph doesn’t work for deep reinforcement learning.
What Next? Tensorflow 2.0!

• Eager execution as the default execution mode.
  • Should make Tensorflow easy to learn and apply.
  • Primarily to compete with PyTorch.

• Cleanup.
  • Deprecated APIs removed.
  • Duplication reduced significantly.

• Heavy reliability on Keras API.
  • More object-oriented and Python-like.
  • Makes reusability of variables easier.
  • Simpler code.
References


• Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.