TensorFlow: A System for Large-Scale Machine Learning
Background: Training deep neural networks

Limited by GPU memory using Nvidia GTX 580 (3GB RAM)

60M Parameters ~ 240 MB

Need to cache activation maps for backpropagation

- Batch size = 128
- $128 \times (227 \times 227 \times 3 + 55 \times 55 \times 96 \times 2 + 96 \times 27 \times 27 + 256 \times 27 \times 27 \times 2 + 256 \times 13 \times 13 + 13 \times 13 \times 384 + 384 \times 13 \times 13 + 256 \times 13 \times 13 + 4096 + 4096 + 1000)$ Parameters ~ 718MB

- That assuming no overhead and single precision values

Tuned splitting across GPUS to balance communication and computation
Background: Training deep neural networks

Limited by GPU memory using Nvidia GTX 580 (3GB RAM)
60M Parameters ~ 240 MB
Need to cache activation maps for backpropagation
  - Batch size = 128
  - 128 * (227*227*3 + 55*55*96*2 + 96*27*27 + 256*27*27*2 + 256*13*13 + 13*13*384 + 384*13*13 + 256*13*13 + 4096 + 4096 + 1000) Parameters ~ 718MB
  - That assuming no overhead and single precision values

Tuned splitting across GPUS to balance communication and computation

Too much manual effort!
Background: Training deep neural networks

**Trend towards distributed training for large-scale models**

**Parameter server: A shared key-value storage abstraction for distributed training**

E.g., DistBelief, Project Adam
Background: Training deep neural networks

**Hides details of distribution**

Trend towards distributed training for large-scale models

**But still difficult to reason about end-to-end structure**

Inflexible update mechanisms and state management

E.g., DistBelief, Project Adam
TensorFlow

Flexible dataflow-based programming model for machine learning

Dataflow captures natural structure of computation in both training and inference
# 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.AdamOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:
sess.run(tf.initialize_all_variables())  # Randomly initialize weights.
for step in range(NUM_STEPS):
x_data, y_data = ...
sess.run(train_op, {x: x_data, y: y_data})  # Load one batch of input data.
sess.run(train_op, {x: x_data, y: y_data})  # Perform one training step.
# 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:
sess.run(tf.initialize_all_variables())
for step in range(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.
# 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_1)  # 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.AdamOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:
sess.run(tf.initialize_all_variables())  # Randomly initialize weights.
for step in range(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.
# 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.
l_1 = tf.nn.relu(tf.matmul(x, W_1) + b_1)  # 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.
l_2 = tf.matmul(l_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(l_2, y)
train_op = tf.train.AdamOptimizer(0.01).minimize(loss)

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

Lack of a flexible programming model to build machine learning models

Prior approaches restricted innovation due to their inflexibility
E.g., parameter updates in parameter server-based approaches
Dataflow-based programming model

Computation structured as a dataflow graph

Nodes can be stateful
Captures accumulated state as part of the training process
E.g., parameter values

Graph elements

Tensors flow across edges between nodes
Operations are expressions over tensors (e.g., constants, matrix multiplication, add)
Variables can accumulate state
Queues provide explicit advanced coordination
TensorFlow

```python
# 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.AdamOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())         # Randomly initialize weights.
    for step in range(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.
```
TensorFlow

# 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_1)         # 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:
sess.run(tf.initialize_all_variables())            # Randomly initialize weights.
for step in range(NUM_STEPS):
    x_data, y_data = ...
    sess.run(train_op, {x: x_data, y: y_data})      # Load one batch of input data.
    sess.run(train_op)                              # Perform one training step.
What are the metrics of success?

Variety of specialized extensions built over the framework
“User level” code

Acceptable performance with respect to state-of-the-art
Extensibility

**Optimization algorithms**
Momentum, AdaGrad, AdaDelta, Adam
E.g., parameter updates in momentum are based on accumulated state over multiple iterations

**Difficult to implement extensible optimization algorithms in parameter servers**
Extensibility

**Sharding very large models**

*E.g., Sparse embedding layers*

Shard embedding layer across parameter server tasks
Encode incoming indices as tensors, and ship to the appropriate shard
Extensibility

Use queues to coordinate the execution of workers

Synchronous replication
Straggler mitigation with backup workers
Competitive training times on single node

<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>
Key results

Extensibility matters!

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).
Key results

Extensibility matters!

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.
Limitations and scope for improvement

**TF’s high-level programming model is tightly coupled with its execution model**

Translate TF programs to more efficient executables using compilation to hide translation

**TF dataflow graphs are static**

**Key runtime decisions, such as number of PS shards, seem to require manual specification**

Can these be automatically deduced based on workload characteristics?