Learning TensorFlow

This is study note of reading the whitepaper-TensorFlow:Large-Scale Machine Learning on Heterogeneous Distributed Systems (http://download.tensorflow.org/paper/whitepaper2015.pdf) Some notes are shamelessly copied from the whitepaper.

About TensorFlow

TensorFlow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms.

- A Framework that allows a developer to express his machine learning algorithm symbolically, performing compilation of these statements and executing them.
- A programming metaphor that requires the developer to model the machine learning algorithm as a computation graph.
    - Nodes in the graph represent mathematical operations
    - Edges in the graph represent the multidimensional data arrays (called tensors), which is used to communicate between nodes.
- A set of Python classes and methods that provide an API interface

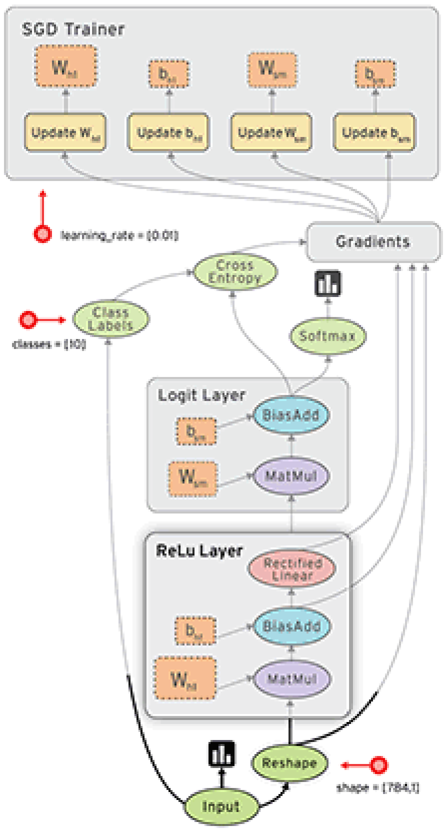
- A re-targetable system that can run on different hardware

TensorFlow supports the computation on one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. It can be deployed to a cloud (Google Cloud Platform (https://cloud.google.com/ml/), AWS (https://aws.amazon.com/marketplace/pp/B01AOE205O), ...) or to a container.

Google have open-sourced the TensorFlow API and a reference implementation under the Apache 2.0 license in November, 2015, available at www.tensorflow.org (https://www.tensorflow.org/).

Programming Model and Basic Concepts

A TensorFlow computation is described by a directed graph (https://en.wikipedia.org/wiki/Directed_acyclic_graph), which is composed of a set of nodes. The graph represents a dataflow computation, with extensions for allowing some kinds of nodes to maintain and update persistent state and for branching and looping control structures within the graph in a manner similar to Naiad (http://research.microsoft.com:8082/pubs/201100/naiad_sosp2013.pdf).



Clients typically construct a computational graph using one of the supported frontend languages (C++ or Python).

Operations and Kernels

An operation has a name and represents an abstract computation (e.g., “matrix multiply”, or “add”). An operation can have attributes, and all attributes must be provided or inferred at graph-construction time in order to instantiate a node to perform the operation.

A kernel is a particular implementation of an operation that can be run on a particular type of device (e.g., CPU or GPU).

Table 1 shows some of the kinds of operations built into the core TensorFlow library.

Operation Types

Table-1

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

Session

Clients programs interact with the TensorFlow system by creating a `Session`. The initial graph when a session is created is empty.

The Session interface supports an `Extend` method to augment the current graph managed by the session with additional nodes and edges.

The main operation is `Run`, which

- takes a set of output names that need to be computed, as well as an optional set of tensors to be fed into the graph in place of certain outputs of nodes.
- Using the arguments to Run, the TensorFlow implementation can compute the transitive closure of all nodes that must be executed in order to compute the outputs that were requested, and can then arrange to execute the appropriate nodes in an order that respects their dependencies.

Variable

is a special kind of **operation** that returns a handle to a persistent mutable tensor that survives across executions of a graph. Handles to these persistent mutable tensors can be passed to a handful of special operations, such as `Assign` and `AssignAdd` (equivalent to +=) that mutate the referenced tensor.

For machine learning applications of TensorFlow, the parameters of the model are typically stored in tensors held in variables, and are updated as part of the Run of the training graph for the model.

Work Flow

Build a Graph -> Initialize Sessions -> Run Sessions

1. Perform any pre-processing steps to set up the dataset for the classifier.
2. Set up the variables (such as the Weights and biases) and placeholders (such as the input vector) that are fed at runtime
3. Define the required computations. E.g. you may define the dot_product computation using the matmul method of tensorflow feeding it the weight matrix variable and the input vector (that may be defined as a placeholder).

```
    Till this point nothing is executed by Tensorflow
```

4. Start a session instance and initialize all variables
5. Run the session

In [16]:
```
# Example-1
import tensorflow as tf
import numpy as np
```

In [17]:
```
c1 = tf.constant([1,2,3,4])
c2 = tf.constant([-1,2,-3,4])
y = c1 + c2 # symbolic add
```

y is only a place holder, nothing happen at this point. If we `run` it, we will get value.

In [19]:
```
print(y)  # this gives us type information.
```

```
Tensor("add_1:0", shape=(4,), dtype=int32)
```

In [20]:
```
# create a session
sess = tf.Session()
```

In [21]:
```
y = sess.run(y)
```

In [22]:
```
print(y)  # we can get value now
```

```
[0 4 0 8]
```

Implementation

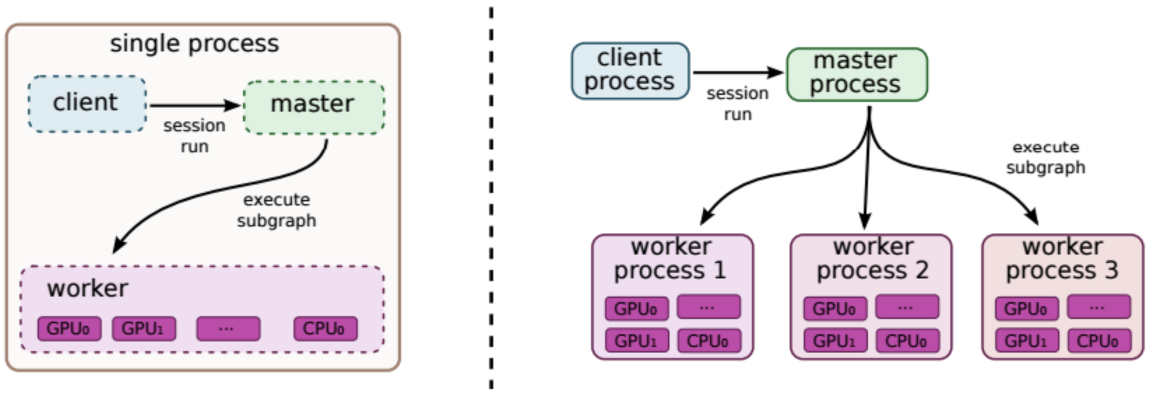
TensorFlow has two main components:

- client, which uses the Session interface to communicate with the master,
- and one or more worker processes, with each worker process responsible for arbitrating

access to one or more computational devices (such as CPU cores or GPU cards) and for executing graph nodes on those devices as instructed by the master.

Tensorflow can be implemented either local or distributed.

- Local: the client, the master, and the worker all run on a single machine
- Distributed: the client, the master, and the workers can all be in different processes on different machines



Devices

Each worker is responsible for one or more devices, and each device has a device type, and a name. Device names are composed of pieces that identify the device’s type, the device’s index within the worker, and, in our distributed setting, an identification of the job and task of the worker (or localhost for the case where the devices are local to the process).

name = type + distributed setting + index
type = [cpu | gpu]
index = 0, 1, ...
distributed setting = task:number

Example device names are "/job:localhost/device:cpu:0" or "/job:worker/task:17/device:gpu:3". We have implementations of our Device interface for CPUs and GPUs, and new device implementations for other device types can be provided via a registration mechanism. Each device object is responsible for managing allocation and deallocation of device memory, and for arranging for the execution of any kernels that are requested by higher levels in the TensorFlow implementation.

Tensors

A tensor is a typed, multidimensional array.

Supported types including

- signed and unsigned integers ranging in size from 8 bits to 64 bits,
- IEEE float and double types,
- a complex number type, and
- a string type (an arbitrary byte array).

Single Device Execution

The nodes of the graph are executed in an order that respects the dependencies between nodes. A node will track its dependencies, whether they have been executed or not. Once this dependency count drops to zero, the node is eligible for execution and is added to a ready-queue. The ready-queue is processed in some unspecified order, delegating execution of the kernel for a node to the device object. When a node has finished executing, the counts of all nodes that depend on the completed node are decremented.

Multiple Devices Execution

When a system has multiple devices, there are two more concerns:

- deciding which device to place the computation for each node in the graph, and then
- managing the required communication of data across device boundaries implied by these placement decisions.

Node Placement

For a given computation graph, one of the main responsibilities of the TensorFlow implementation is to map the computation onto the set of available devices.

One input to the placement algorithm is a cost model, which contains

- estimates of the sizes (in bytes) of the input and
- output tensors for each graph node, along with
- estimates of the computation time required for each node when presented with its input tensors.

This cost model is either statically estimated based on

- heuristics associated with different operation types, or is
- measured based on an actual set of placement decisions for earlier executions of the graph.

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

The placement algorithm starts with the sources of the computation graph, and simulates the activity on each device in the system as it progresses. For each node that is reached in this traversal, the set of feasible devices is considered (a device may not be feasible if the device does not provide a kernel that implements the particular operation).

For nodes with multiple feasible devices, the placement algorithm uses a greedy heuristic that examines the effects on the completion time of the node of placing the node on each possible device. This heuristic takes into account the estimated or measured execution time of the operation on that kind of device from the cost model, and also includes the costs of any communication that would be introduced in order to transmit inputs to this node from other devices to the considered device. The device where the node’s operation would finish the soonest is selected as the device for that operation, and the placement process then continues onwards to make placement decisions for other nodes in the graph, including downstream nodes that are now ready for their own simulated execution.

Distributed Execution

Send/Receive node pairs that communicate across worker processes use remote communication mechanisms such as TCP or RDMA to move data across machine boundaries.

Fault Tolerance Failures in a distributed execution can be detected in a variety of places. The main ones we rely on are (a) an error in a communication between a Send and Receive node pair, and (b) periodic health-checks from the master process to every worker process. When a failure is detected, the entire graph execution is aborted and restarted from scratch. Recall however that Variable nodes refer to tensors that persist across executions of the graph. We support consistent checkpointing and recovery of this state on a restart. In partcular, each Variable node is connected to a Save node. These Save nodes are executed periodically, say once every N iterations, or once every N seconds. When they execute, the contents of the variables are written to persistent storage, e.g., a distributed file system. Similarly each Variable is connected to a Restore node that is only enabled in the first iteration after a restart. See Section 4.2 for details on how some nodes can only be enabled on some executions of the graph.

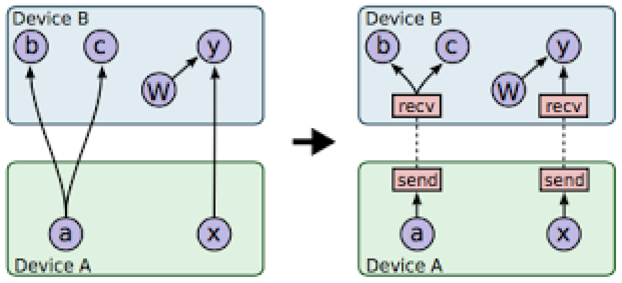
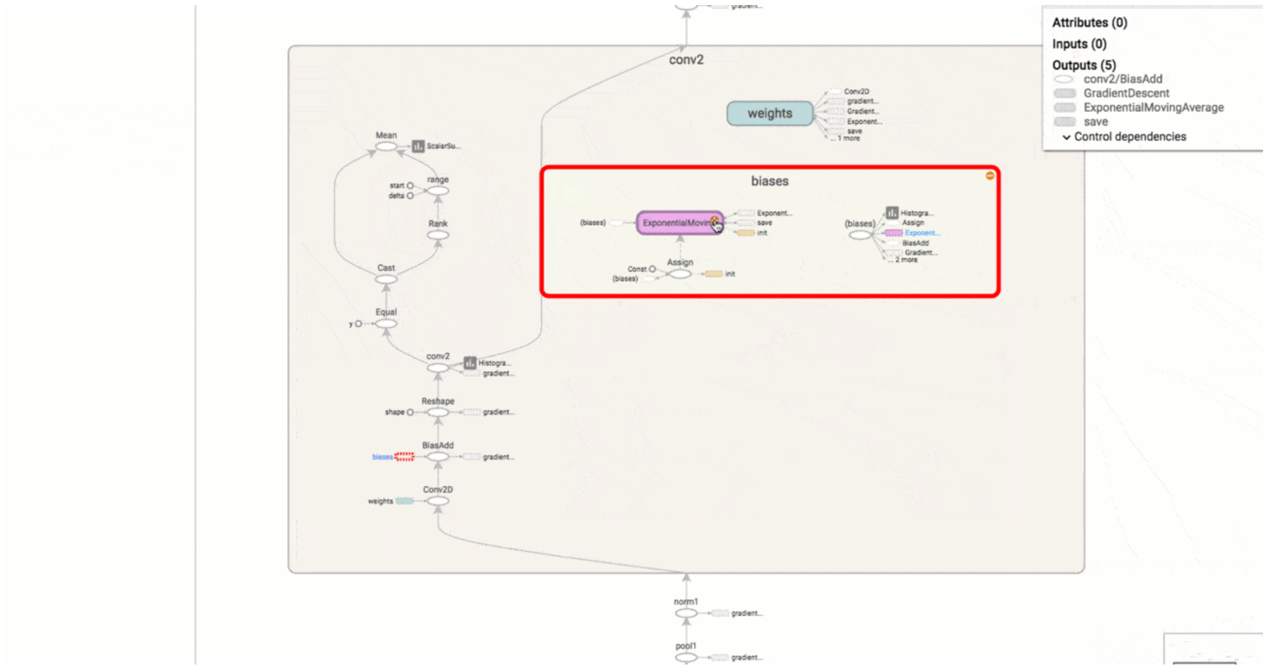


Figure 4: Before & after insertion of Send/Receive nodes

TensorBoard



Reference

[Whitepaper (http://download.tensorflow.org/paper/whitepaper2015.pdf)](http://download.tensorflow.org/paper/whitepaper2015.pdf)
[AI and Deep Machine Learning Primer (http://a16z.com/2016/06/10/ai-deep-learning-machines/)](http://a16z.com/2016/06/10/ai-deep-learning-machines/)

```
In [ ]:
```