Tensorflow : Large-Scale Machine Learning on Heterogeneous Distributed Systems

Mohammad Amin Samadi

Outline

1. *Abstract*
2. Programming Model and Basic Concepts
3. Implementation
4. Extensions
5. Optimizations
6. Common Programming Idioms
7. Tensorboard

1. Abstract

- TensorFlow is both an interface for expressing machine learning algorithms and an implementation to execute them
- Code can be transported across various machine architectures with little to no changes to the code
- Has been used at Google for all manner of machine learning tasks

Outline

1. Abstract
2. *Programming Model and Basic Concepts*
3. Implementation
4. Extensions
5. Optimizations
6. Common Programming Idioms

2. Programming Model and Basic Concepts

- TensorFlow computations are represented by *directed graphs*, which are composed of *nodes*
- *Values of 'normal' edges , connecting one node's output to another node's input , are tensors , n-dimensional arrays.*
- *There are special edges called control dependencies : no model data is transferred on these edges, rather they indicate that the source node must finish execution before the destination node begins execution.*

(cont) Programming Model

Operations and Kernels

- Operations have names and represent an abstract computation, such as "matrix multiply" or "add"

Sessions

- Clients interact with TensorFlow by creating a *Session*, which supports two main functions: *Extend* and *Run*
    - The Extend method adds additional nodes and edges to the existing dataflow model

Variables

- A *Variable* is a handle to a persistent and mutable tensor which survives each execution of a graph
- For ML tasks, learned parameters are usually held in TensorFlow Variables

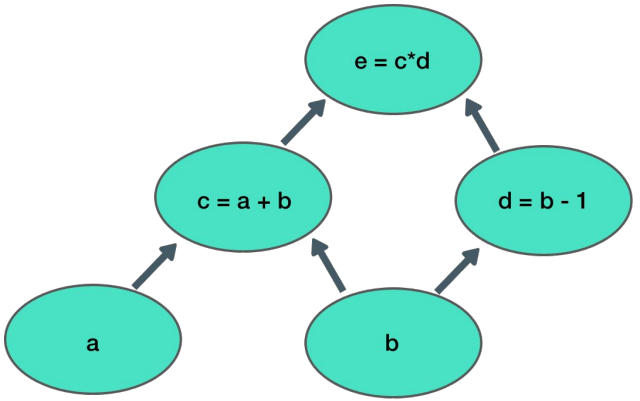
Computational Graph

```
import tensorflow as tf

a = tf.Variable(initial_value=10)
b = tf.Variable(initial_value=5)
c = tf.add(a , b)
d = tf.subtract(b , 1)
e = tf.multiply(c , d)

with tf.Session() as s:
   i = tf.global_variables_initializer()
   s.run(i)
   res = s.run(e)
   print(res)
```

>> res = 60



Outline

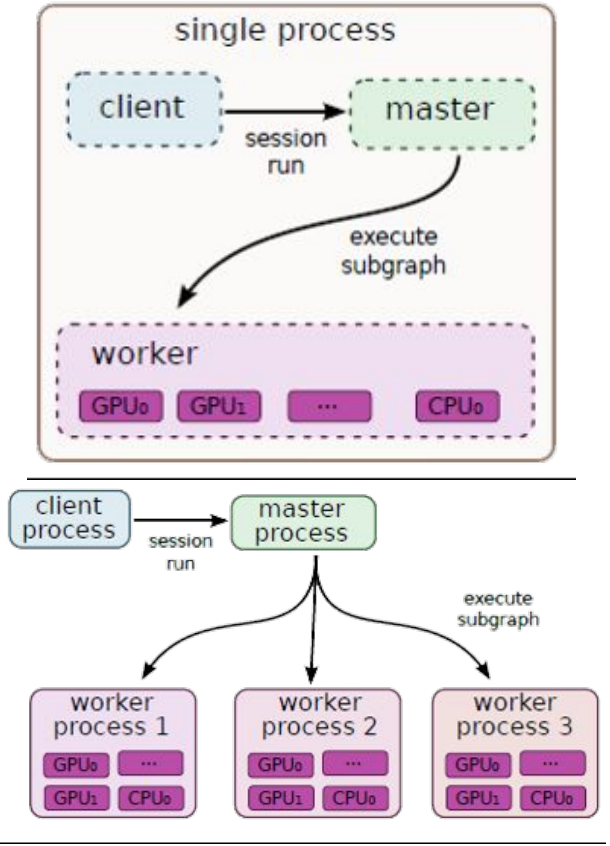
3. Implementation

There are three primary components in a TensorFlow system: the *client*, the *master*, and *worker processes*

The master schedules and coordinates worker processes and relays results back to the client.

The client uses a Session interface to communicate with the master.

Worker processes are responsible for maintaining access to devices such as CPU/GPU cores and execute graph nodes on their respective devices.



Devices

- Each device has both a device type and a name
  - Names are composed of the device's type, its index in a worker process, and (when used in a distributed setting) an identification of the job and task of the worker process
  - Example device names:
  - Local: /job:localhost/device:cpu:0
  - Distributed: /job:worker/task:17/device:gpu:3
- A device object manages its device's memory and executes kernels as requested

Tensors

- Typed, multi-dimensional array
- Memory management of tensors is handled automatically

3.1. Single-Device Execution

1. All nodes required to compute the desired output node(s) are determined
2. Each node is given a count of dependencies that need to be completed before it can begin execution
3. When a node's dependency count is zero, it is added to a ready queue
4. The ready queue delegates node kernel execution to device objects
5. When a node completes execution, the counts of all dependant nodes are decremented
6. Repeat steps 3-5 until the desired output is computed

3.2. Multi-Device Execution

There are two main challenges introduced when using multiple devices:

- Deciding which device should process each node
- Managing communication between devices as necessary after assigning nodes

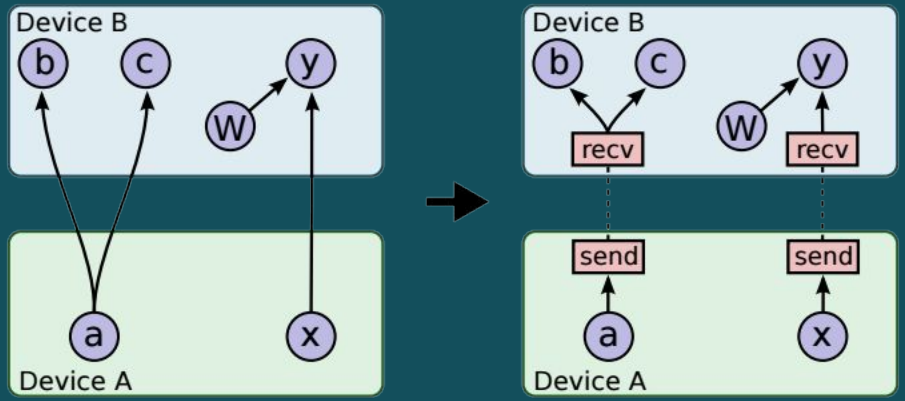
3.2.1. Node Placement

- Cost model is created, containing estimated computation and communication time.
- Every feasible (matching kernel) device for each node is identified and examined and the device finishing the operation sooner is selected (Greedy Algorithm) .

3.2.2. Cross-Device Communication

- After performing node placement the graph is divided into a set of subgraphs.
- Any edge between nodes on different devices is replaced by two new edges:
  - The outputing node will have an edge between it and a new *Send* node
  - The recieving node will have an edge between it and a new *Receive* node

- Memory allocation
- Synchronization
- Scalability
- Single run execution per graph



3.3. Distributed Execution

- Similar to multi-device execution. Send/Receive nodes that communicate across worker processes use mechanisms such as TCP or RDMA (remote direct memory access) to transmit data from machine to machine

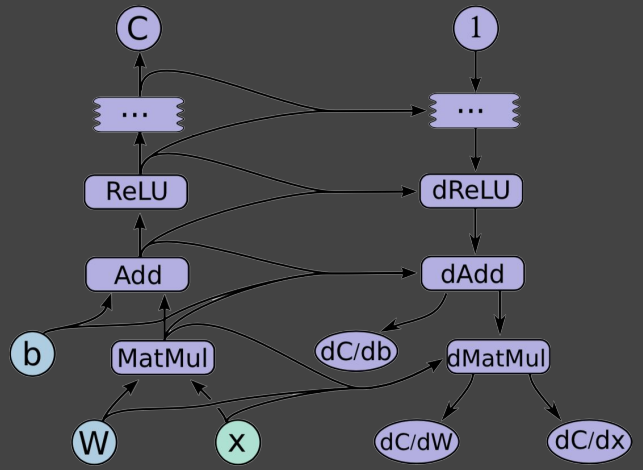
Fault Tolerance

- Main failures are communication errors
- When a failure is detected the operations are all aborted.
- There are some mechanisms for checkpointing and recovery after a restart.

Outline

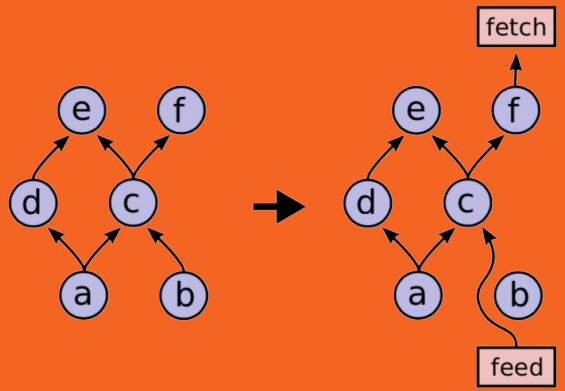
4.1. Gradient Computation

- TensorFlow has built-in gradient computation
- If a tensor, *C*, depends on a set of previous nodes, the gradient of *C* with respect to those previous nodes can be automatically computed with a built-in function, even if there are many layers in between them
- Derivations are computed using the *chain rule*.
- The area of improvement for automatic gradient computation is memory management.



4.2. Partial Execution

- Every tensor is defined using name : port
- We specify the exact subgraph using *fetch* and *feed* arguments.
- Once the fetch and feed nodes have been inserted, TensorFlow determines which nodes need to be executed. It moves backwards, starting at the fetch nodes, and uses the dependencies of the graph to determine all nodes that must be executed in the modified graph in order to compute the desired outputs



4.3. Device Constraints

- Users can provide partial constraints on nodes about which devices they can run on
- Examples: only allowing a node to run on GPUs, specifying a specific worker process/task, or ensuring that it is grouped with specific nodes
- Then constraints are added to the node placement Algorithm (3.2.1) .

4.4. Control Flow

- TensorFlow incorporates a few primitive control flow operators which allow for the skipping of subgraph execution and the expression of iteration. Using these primitive operators, higher-level constructs such as *if* and *while* statements can be compiled into TensorFlow graphs.
- The *Enter*, *Leave*, and *NextIteration* operators allow us to express High-level programming blocks!

4.5. Input Operations

- There is also the possibility to provide the input using files.
- Using this feature can reduce data transfer overhead when using TensorFlow on a distributed implementation (specifically when the client is on a different machine from the worker process):
  - Using feed_dict will cause data to first be sent from the storage system to the client, and then from client to the worker process.
  - Reading from the file will cause data to be sent directly from the storage system to the worker process.
- Ex : tf.TextLineReader

4.6. Queues

- Queues allows us to prefetch next batch of data while the previous batch is being processed.
- It supports FIFO and shuffling queue that can be useful for machine learning algorithms.

Outline

5.1. Common Subexpression Elimination

- Before execution, TensorFlow does a pass over the computation graph and reduces nodes with identical inputs and operations down to a single node.
- This prevents redundant execution of the same computation.

5.2. Controlling Data Communication and Memory Usage

- Proper scheduling of operations can create dramatic improvements on data transfer and memory usage rates by reducing the amount of time intermediate data needs to be stored in memory
- One particular example used in the implementation TensorFlow is the scheduling of Receive nodes (see "3.2 Cross Device Communication")

5.3. Lossy Compression

- Converts 32-bit floating point representation to 16-bit representation and then converts it back to 32-bit and replaces the lost parts with zeros.

Outline

7.1. Data Parallel Training

Users can parallelize the computation of the gradient, separating mini-batch elements onto different devices

- For example, a mini-batch size of 1000 elements can be split into 10 smaller, parallel computations of 100 elements. After they all finish running, their results can be combined to achieve the same result as if the entire calculation was performed in a single, sequential computation
- Can be implemented both with 1 master or multiple masters

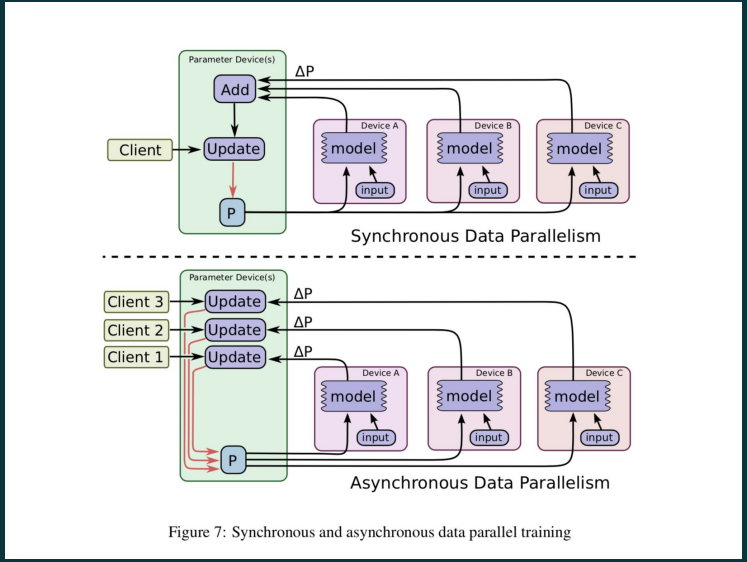


Figure 7: Synchronous and asynchronous data parallel training

7.2. Model Parallel Training

- Can run separate portions of the computation graph on different devices simultaneously on the same batch of examples
- Can also run a small set of concurrent training steps on a single device
    - This can "fill in the gaps" of device utilization, when parallel execution on all devices might not make full use of computation cycles

Thanks for your attention