

A HPX backend for TensorFlow

Lukas Troska

Institute for Numerical Simulation
University Bonn

April 5, 2017

Table of Contents

- 1 Introduction to TensorFlow
 - What is it?
 - Examples
- 2 The HPX backend
 - Sources of Parallelism
 - TensorFlow architecture
 - Key Steps
 - Examples
 - Benchmarks

Table of Contents

1 Introduction to TensorFlow

- What is it?
- Examples

2 The HPX backend

- Sources of Parallelism
- TensorFlow architecture
- Key Steps
- Examples
- Benchmarks

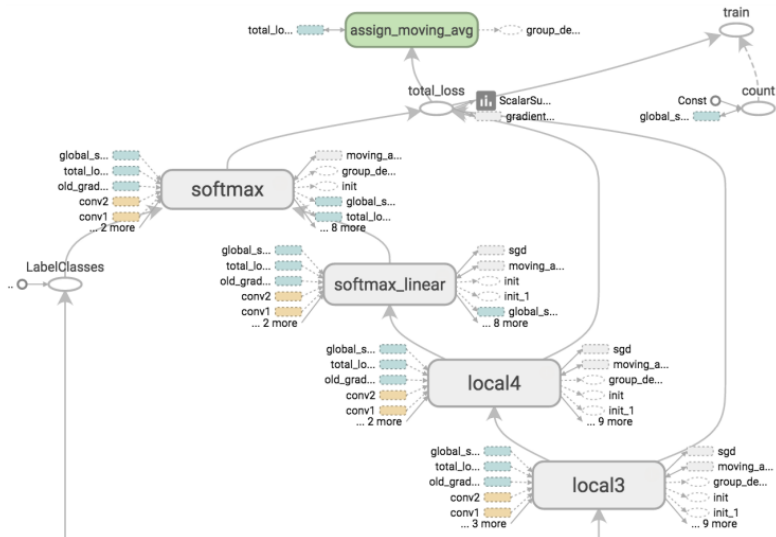
Introduction to TensorFlow: What is it?

"An open-source software library for Machine Intelligence"

- uses dataflow graphs to represent numerical computation
 - nodes \equiv computation ("operations")
 - edges \equiv flow of data ("tensors")
- common operations are already implemented (user can supply custom operations)
- backend written in C++ with python frontend
- seamless integration with numpy etc. for usability

- node \equiv graph node (operation)
- tensor \equiv data (\approx multidimensional array)
- session \equiv environment, in which a graph is executed
- server \equiv cluster node
- master \equiv part of the server responsible for creating sessions
- worker \equiv part of the server responsible for graph execution

Introduction to TensorFlow: What is it?



```
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(
    cross_entropy)

with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for _ 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}))
```

Table of Contents

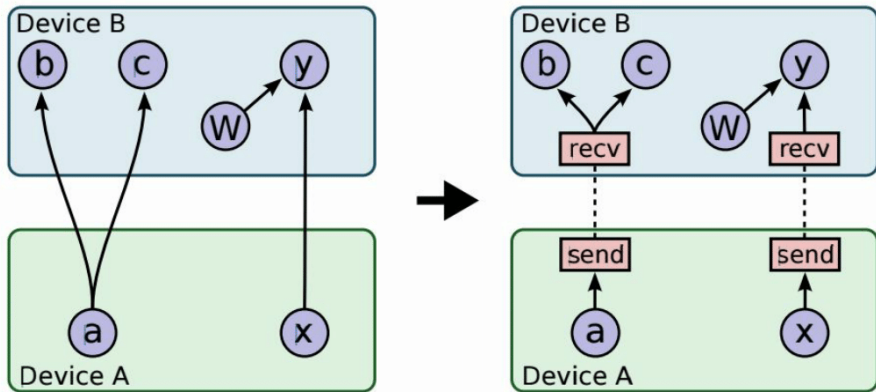
- 1 Introduction to TensorFlow
 - What is it?
 - Examples
- 2 The HPX backend
 - Sources of Parallelism
 - TensorFlow architecture
 - Key Steps
 - Examples
 - Benchmarks

The HPX backend: Sources of Parallelism

There are two (main) sources of parallelism

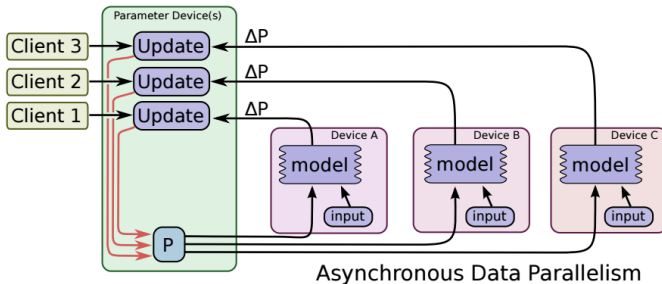
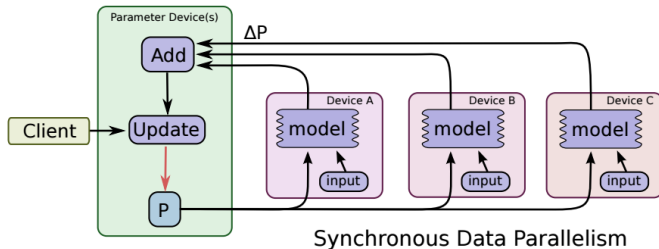
- **Graph Parallelism:** distributing the graph to multiple nodes
- **(a-)synchronous Data Parallelism:** each node has the same graph, but get different chunks of the input data
- mixing of these is possible

Graph Parallelism



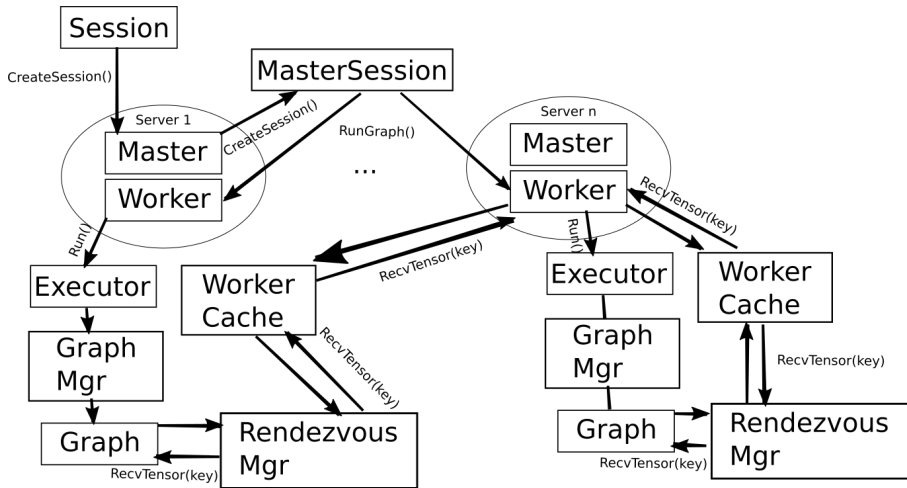
¹<http://muratbuffalo.blogspot.com/2016/06/tensorflow-system-for-large-scale.html>

Data Parallelism



¹<http://ischlag.github.io/2016/06/12/async-distributed-tensorflow/>

The HPX backend: TensorFlow architecture



The HPX backend: Key Steps

- TF currently uses the gRPC protocol ("Remote Procedure Call") and protocol buffers ("protobufs") for distributed work
- gRPC is used in the "Master", "Worker", "Rendezvous" and "Session" classes to communicate with other (remote) instances
⇒ replace with components and actions
- Communication of data through protobufs, which are basically POD-types with getter/setter
⇒ Serialize protobufs with HPX techniques
- Use dynamic connection for the HPX runtime

Wrapping TF into components/actions

The "easy" step first: replace gRPC

- base classes for "Master", "Worker" etc. already exist (for non distributed use)
 - ⇒ wrap into components
- allows reuse of code and (almost) seamless integration into the existing codebase, without changing the structure

Wrapping TF into components/actions

```
std::pair<Status, CreateSessionResponse>
HPXMasterServer::CreateSession(CreateSessionRequest const&
    request)
{
    CreateSessionResponse response;
    Status s = CallMasterSync(&Master::CreateSession, &request,
        &response);
    return std::make_pair(std::move(s), std::move(response));
}
HPX_DEFINE_COMPONENT_ACTION(HPXMasterServer,
                            CreateSession,
                            CreateSessionAction);
```

Serialization of messages

- protobufs are basically POD-types with getter/setter
- user provides a proto-file which the protobuf compiler compiles into a header

```
message RunGraphRequest {
  string graph_handle = 1;

  int64 step_id = 2;

  ExecutorOpts exec_opts = 5;

  repeated NamedTensorProto send = 3;
  repeated string recv_key = 4;

  bool is_partial = 6;
  bool is_last_partial_run = 7;
}
```

Serialization of messages

- protobufs provide methods for serialization to/from some classes (e.g. string, ostream)

```
inline void serialize(hpx::serialization::output_archive& ar,  
                    ::google::protobuf::Message& proto,  
                    unsigned v)  
{  
    std::string data;  
    proto.SerializeToString(&data);  
  
    ar << data;  
}
```

Serialization of Tensors

- structure of the serialization process:
class \implies proto \implies string \implies serialization \implies string \implies
proto \implies class
- works well for small protos, but expensive for protos containing a lot of data, e.g. large tensors due to unnecessary copies
 \implies treat tensors differently

Serialization of Tensors

```
inline void serialize(hpx::serialization::output_archive& ar,  
                    std::vector< ::grpc::Slice>& data,  
                    unsigned v)  
{  
    std::size_t num_slices = data.size();  
  
    ar << num_slices;  
  
    for (auto const& s : data)  
        ar << s.size() << hpx::serialization::make_array(s.begin  
                    (), s.size());  
}
```

Serialization of Tensors

```
inline void serialize(hpx::serialization::input_archive& ar,
                    std::vector< ::grpc::Slice>& buf) {
    std::size_t num_slices;
    ar >> num_slices;

    buf.reserve(num_slices);
    for (std::size_t i = 0; i < num_slices; ++i) {
        std::size_t slice_len;
        ar >> slice_len;

        grpc_slice s = grpc_slice_malloc(slice_len);
        hpx::serialization::array<uint8_t> array =
            hpx::serialization::make_array(GPR_SLICE_START_PTR(s)
            , slice_len);
        ar >> array;
        buf.push_back(::grpc::Slice(s, ::grpc::Slice::STEAL_REF))
            ;
    }
}
```

Replacing the Executor

- The executor computes a (sub-)graph
- each node should be computed as soon as its inputs become ready
⇒ encode the edges of the graph by unique keys and store the outputs of each node as futures ⇒ `hpx::dataflow`
- need to take into account loops, invariants etc.

MNIST Softmax (HPX)

```
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(
    cross_entropy)

with tf.Session('local_hpx') as sess:
    tf.global_variables_initializer().run()
    for _ 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}))
```

Distributed TF

```
cluster = tf.train.ClusterSpec({"local": ["localhost:2222", "
    localhost:2223"]})
server = tf.train.Server(cluster, job_name="local", task_index=
    task_number)
server.start()

if task_number == 1:
    x = tf.constant(2)
    with tf.device("/job:local/task:1"):
        y2 = x - 66
    with tf.device("/job:local/task:0"):
        y1 = x + 300
        y = y1 + y2

    with tf.Session("grpc://localhost:2222") as sess:
        print(sess.run(y))
else:
    server.join()
```

Distributed TF (HPX)

```
cluster = tf.train.ClusterSpec({"local": ["localhost:2222", "
    localhost:2223"], "root_hpx": ["localhost:2222"]})
server = tf.train.Server(cluster, job_name="local", task_index=
    task_number, protocol="hpx")
server.start()

if task_number == 1:
    x = tf.constant(2)
    with tf.device("/job:local/task:1"):
        y2 = x - 66
    with tf.device("/job:local/task:0"):
        y1 = x + 300
        y = y1 + y2

    with tf.Session("hpx://localhost:2223|localhost:2222") as
        sess:
            print (sess.run(y))
else:
    server.join()
```

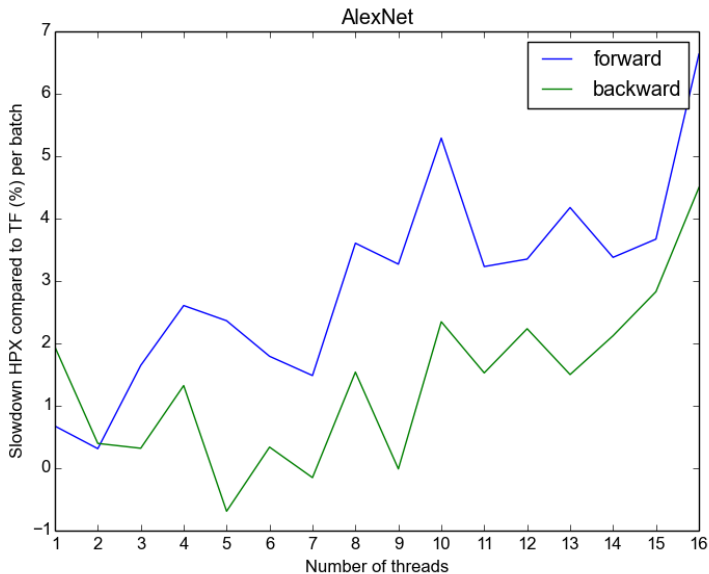
The HPX backend: Benchmarks

- benchmarking done with multiple different (well known) neural nets (local and distributed)
- **local:** AlexNet, GoogleNet, Overfeat (implementations taken from ¹)
- **distributed:** MNIST

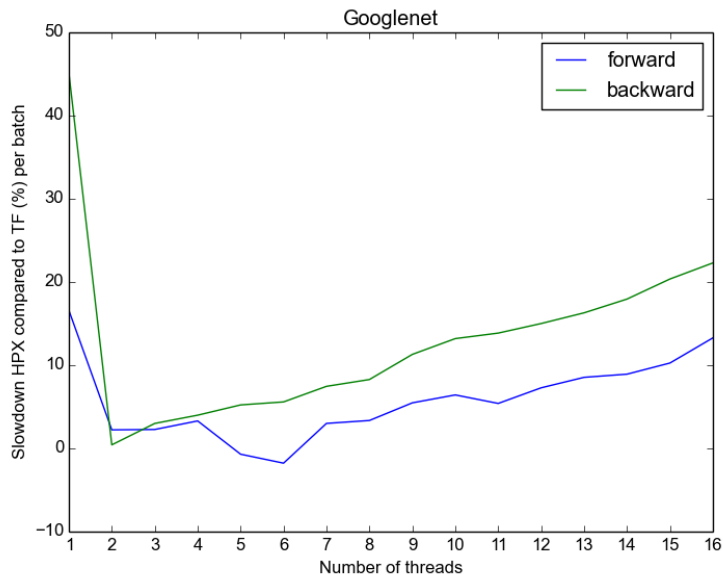
¹<https://github.com/soumith/convnet-benchmarks>

- winning model of the ILSVRC2012 (imagenet large scale visual recognition challenge)
- **Task:** classify \approx 1.3 million high resolution images into 1000 classes
- 5 convolutional layers (some with max-pooling and normalization layer after), two globally connected layers, final softmax layer
- 60 million parameters and 500,000 neuron

AlexNet

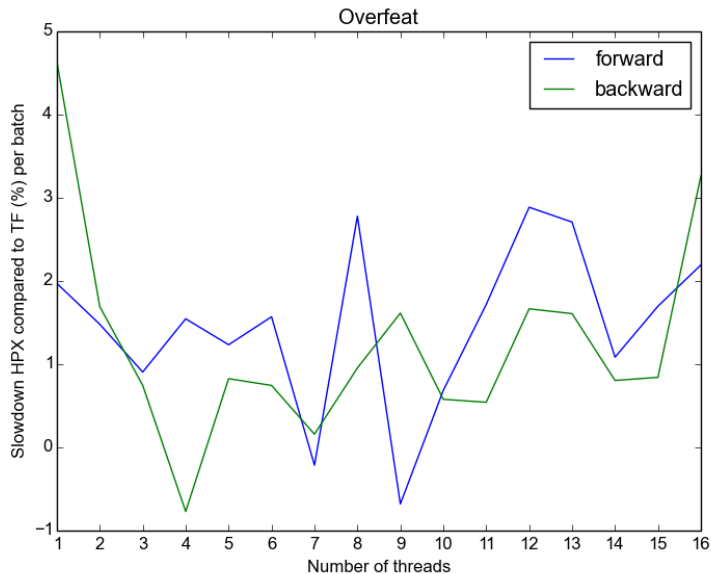


- winning model of the ILSVRC 2014 (classification + localization)
- **deep neural net:** 22 layers
- 12x fewer parameters than alexnet



- winning model of the ILSVRC2013 localization task
- 5 convolutional layers (some with max-pooling layer after), two globally connected layers
- similar to AlexNet

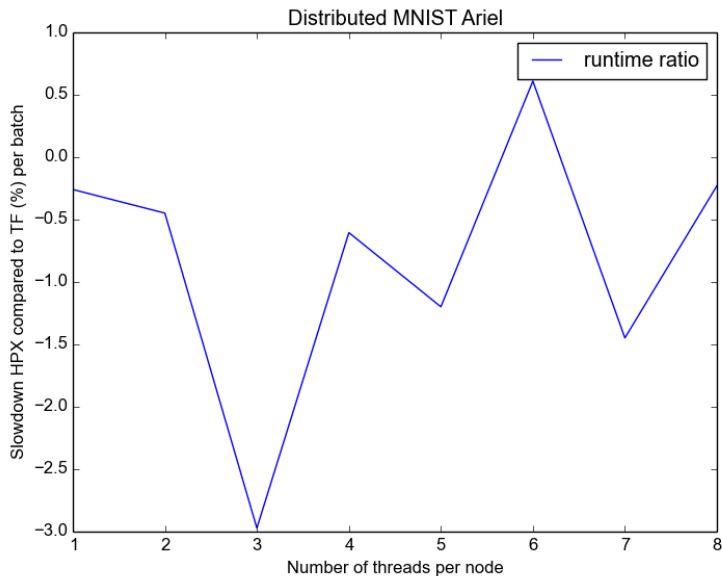
Overfeat



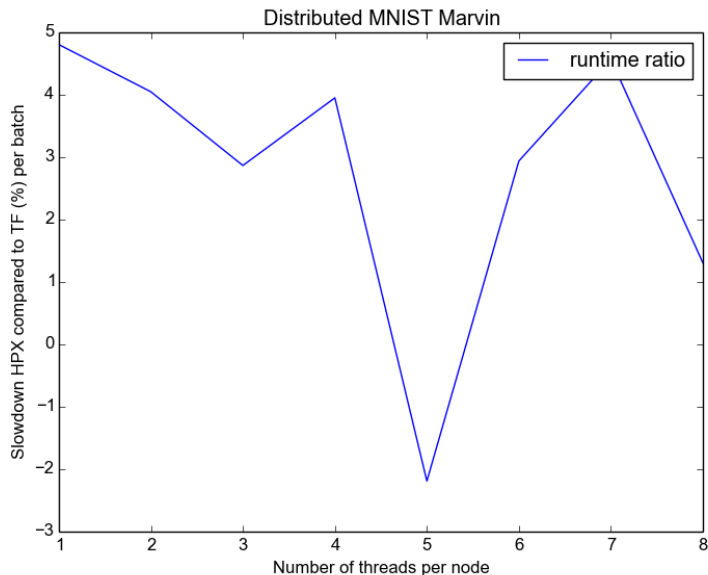
Distributed MNIST

- similar to earlier MNIST example, but distributed
- data parallelism model
- one parameter server, multiple workers on rostam
- parameter server stores the weights, and updates them using the new deltas from each worker

Distributed MNIST (Ariel)



Distributed MNIST (Marvin)



Further Reading I



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E

<http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>



Going Deeper with Convolutions

C. Szegedy et al

arXiv:1409.484



OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks

P. Sermanet et al

arXiv:1312.62292

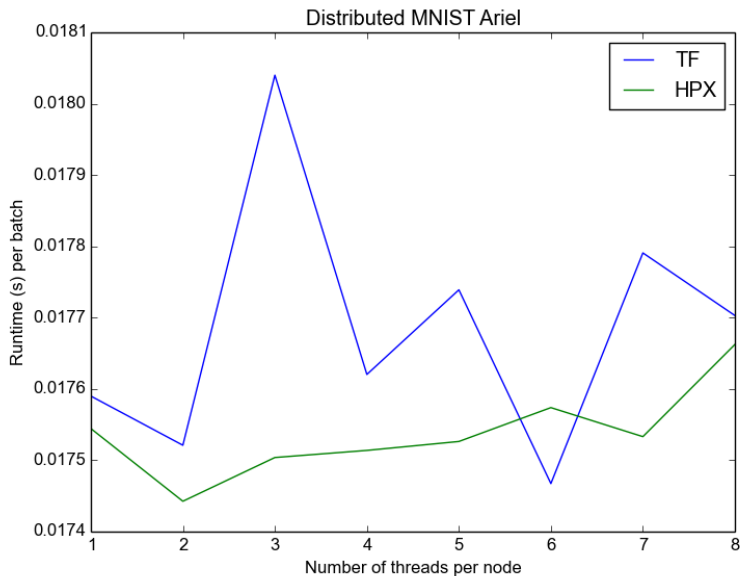


Very Deep Convolutional Networks for Large-Scale Image Recognition

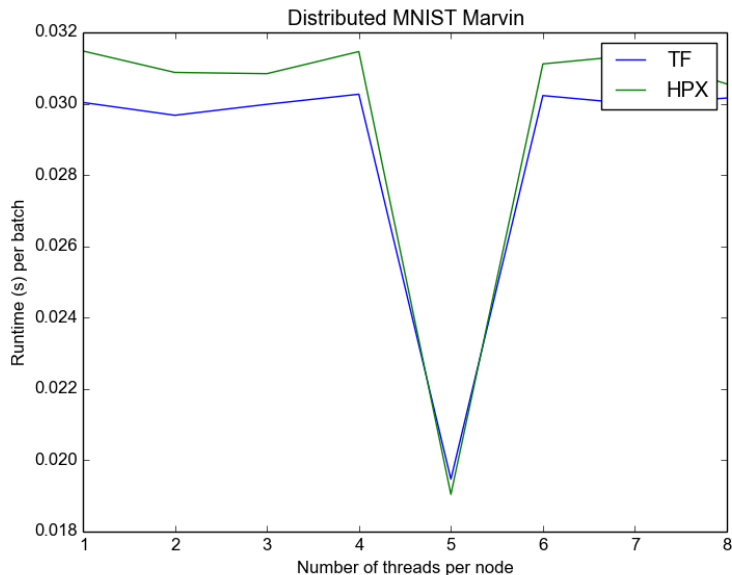
K. Simonyan, A. Zisserman

arXiv:1409.1556

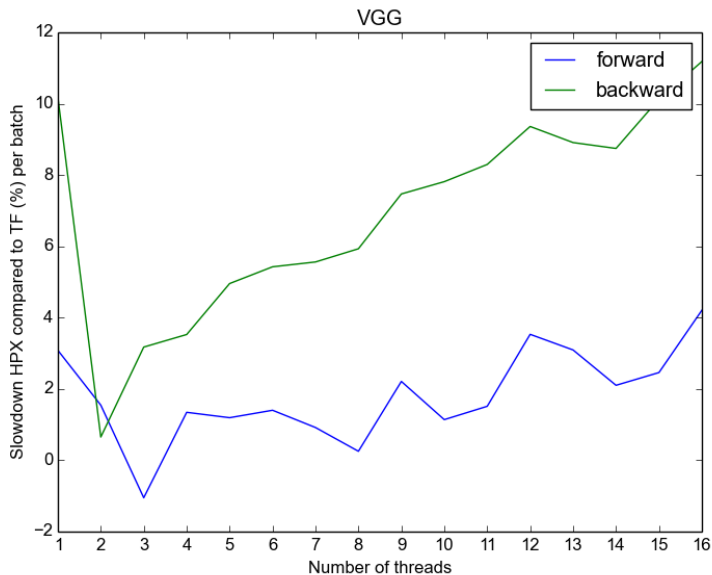
Distributed MNIST (Ariel)



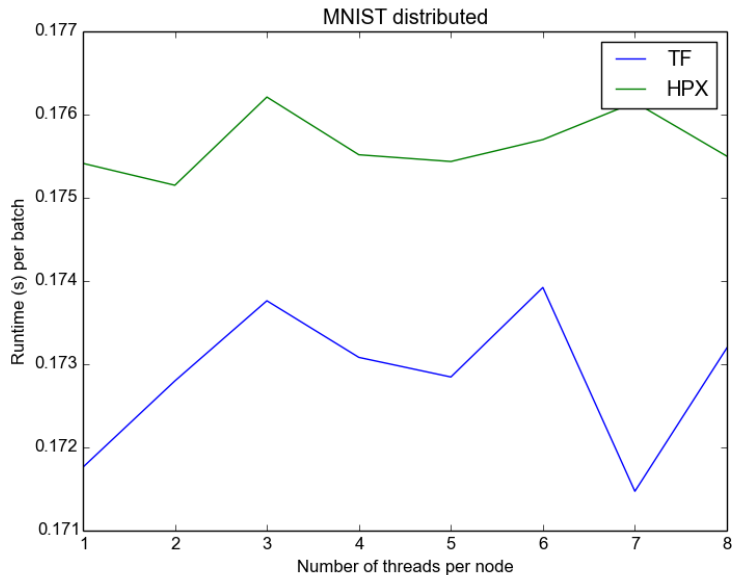
Distributed MNIST (Marvin)



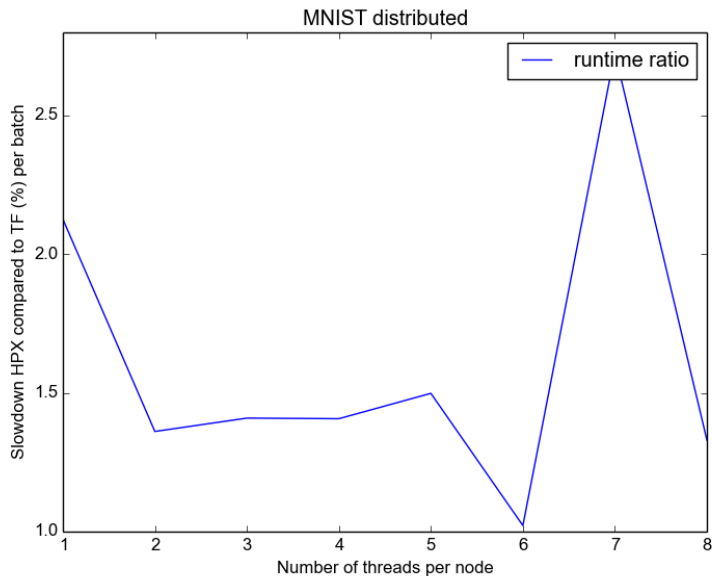
- participating model in ILSVRC2014
- 16 layers
- by the Visual Geometry Group at University of Oxford

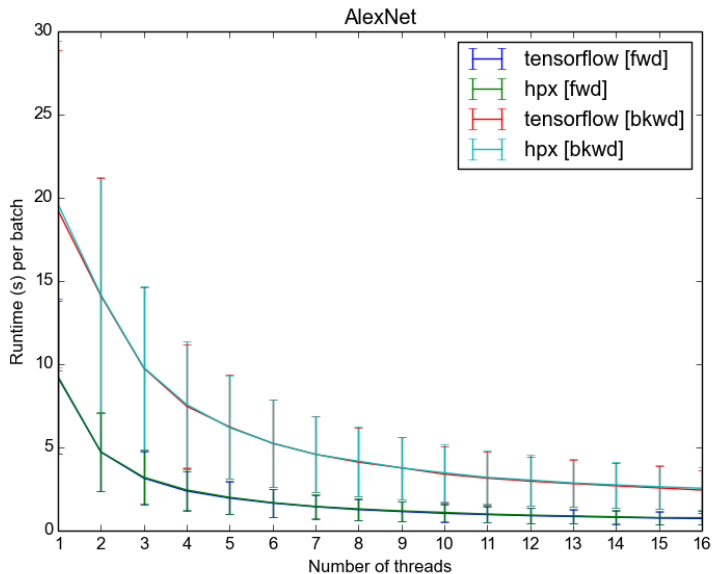


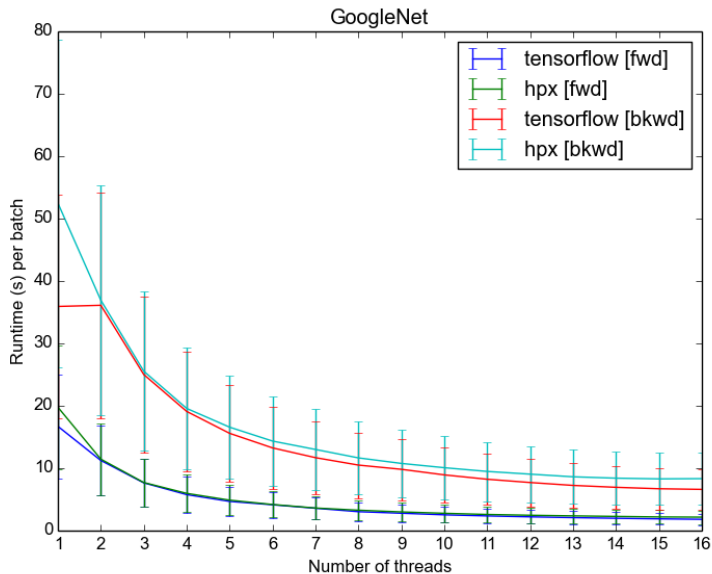
Distributed with TF Executor



Distributed with TF Executor







OverFeat

