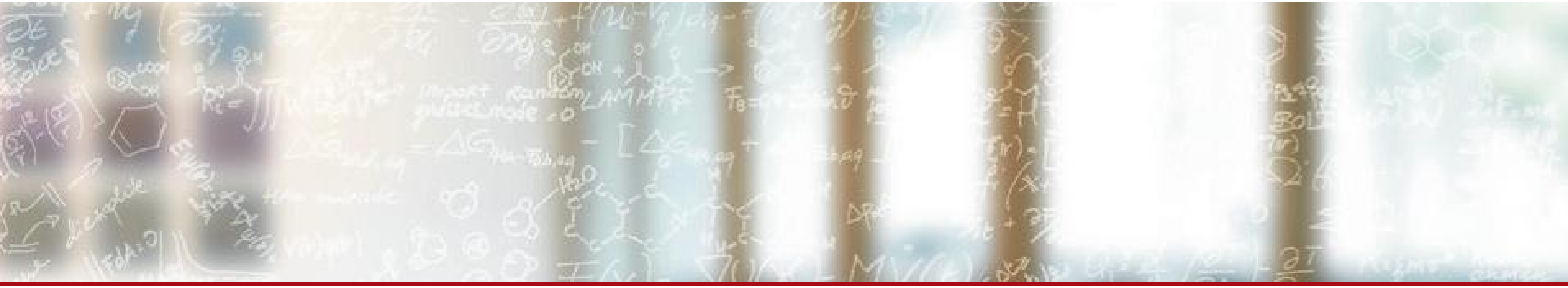
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Distributed TensorFlow: A performance evaluation

End-of-internship Seminar

Emanuele Bugliarello
emanuele.bugliarello@gmail.com

September 6, 2017

https://github.com/e-bug/distributed-tensorflow-benchmarks

Introduction

What is TensorFlow?

Google's open-source software library for Machine Learning

- Best-supported client language: Python
- Experimental interfaces for: C++, Java and Go

Why TensorFlow?

- Portable & flexible → popular in industries and in research communities
- Most CSCS clients choose TensorFlow as their Deep Learning library

Why distributed training?

Training a neural network can take an impractically long time on a single machine (even with a GPU)

Results

On 64 GPUs: ~*80%* scalability efficiency in Piz Daint & almost *90%* in 8 8-GPU nodes

ToC

Distributed training in TensorFlow (1)

TensorFlow is based on data flow graphs
- Nodes represent mathematical operations
- Tensors move across the edges between nodes

Writing a TensorFlow application
1. Build computation graph
2. Run instances of that graph

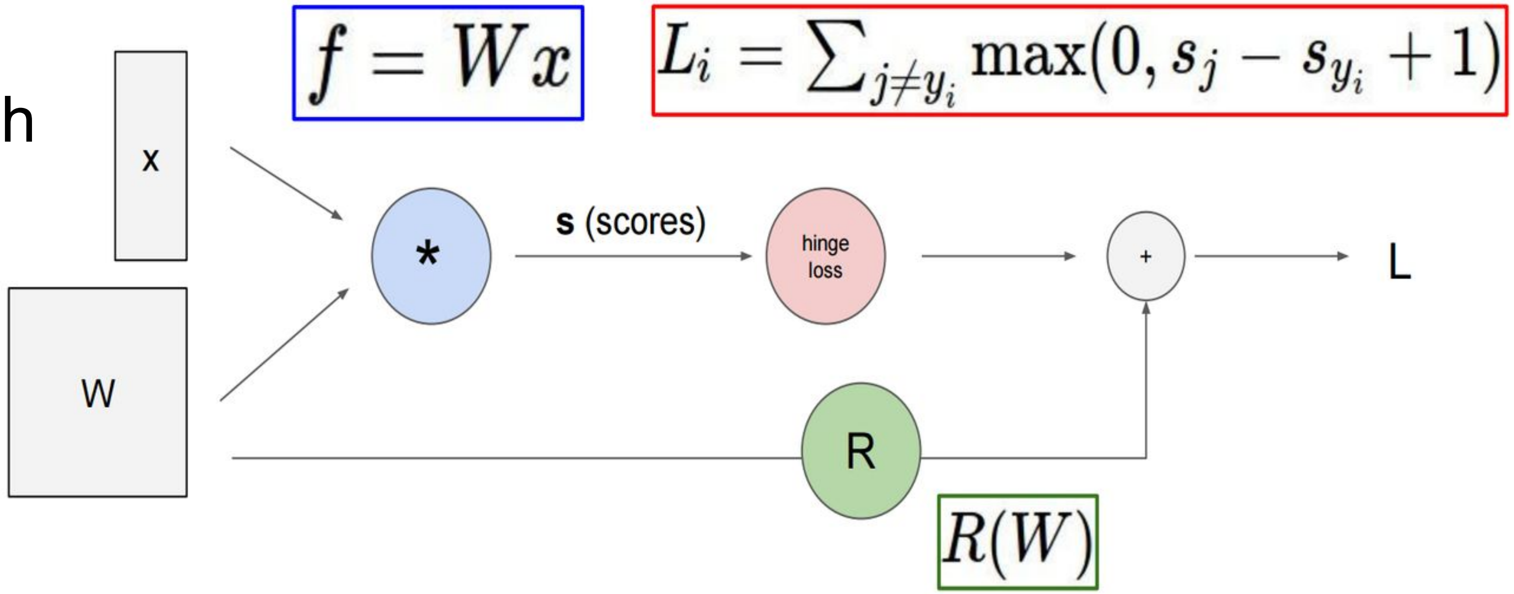
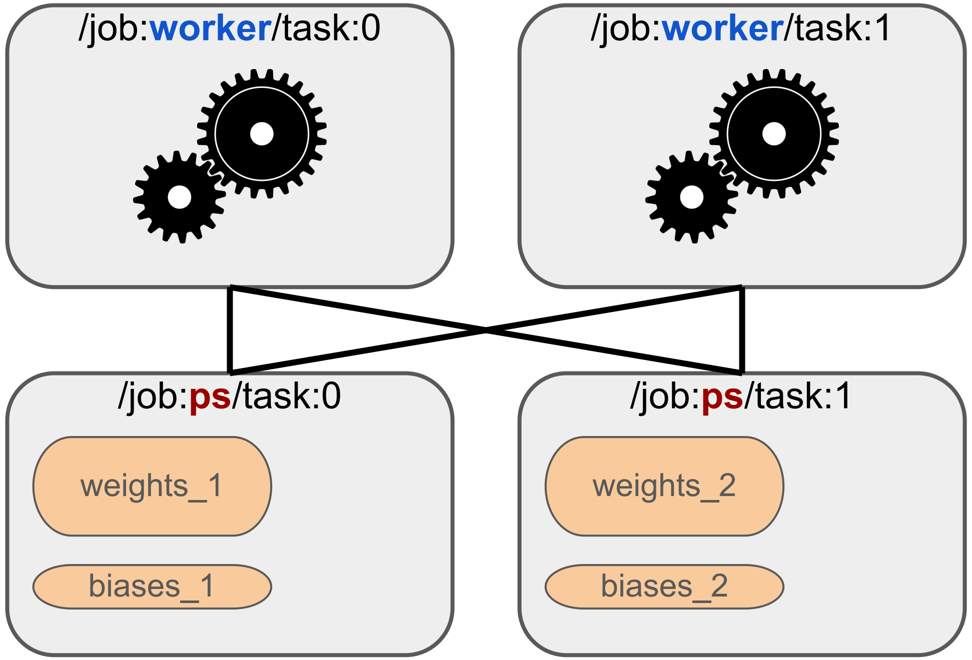


Figure 1: Computational graph for regularized Multiclass SVM loss (CS231N, Stanford University)

Distributed training in TensorFlow (2)

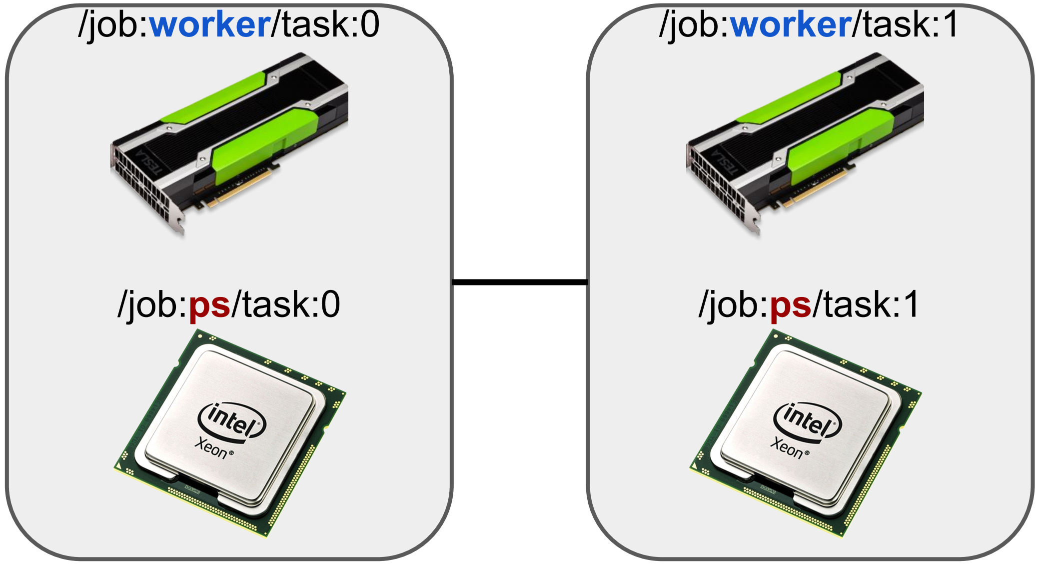
- Split the training of a neural network across multiple nodes
- Most common approach: data parallelism
  - Each node has an instance of the model and reads different training samples
  - Also known as “between-graph replication” in TensorFlow
- Processes can either be:
  - Worker
    - Runs the model
    - Sends its local gradients to the PSs
    - Receives updated variables back
  - Parameter Server (PS)
    - Hosts trainable variables
    - Updates them with values sent by the Workers
- PSs sum gradients to merge in one step what each Worker has learned to reduce the loss

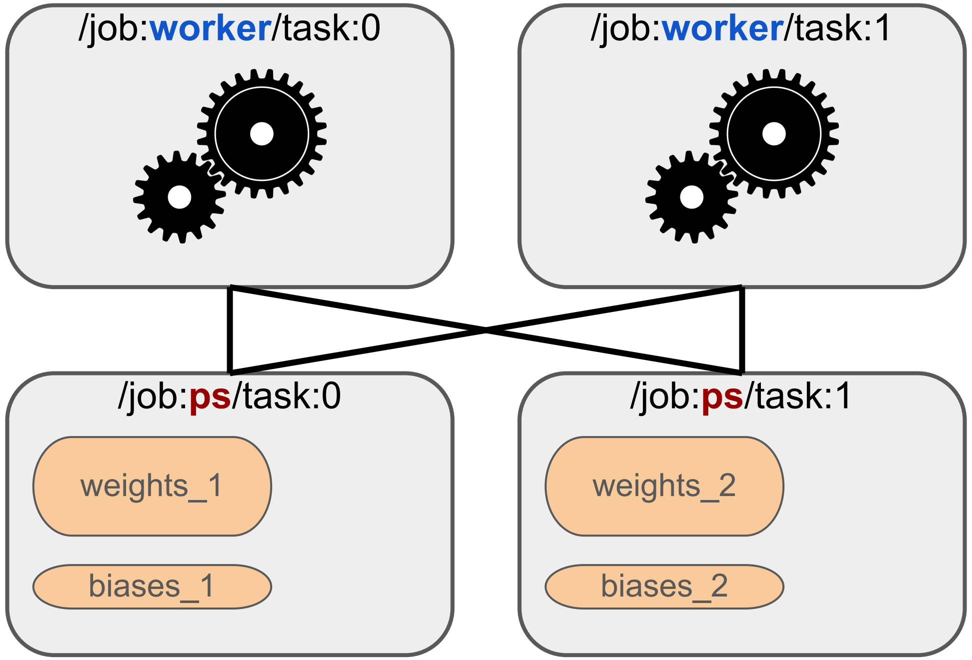


Distributed training in TensorFlow (3)

- Workers need to send their updates to the correct Parameter Servers
  - Use TensorFlow's replica_device_setter for a deterministic variable allocation

- Parameter Servers and Workers may coexist on the same machine
  - Recommended when Workers run on GPUs
  - Reduce the number of nodes
  - Minimize network communications





Distributed training in TensorFlow (4)

- Define cluster of nodes and the role of each of them (PS/Worker)
- The following code snippet (https://clindatsci.com/blog/2017/5/31/distributed-tensorflow) would be executed on each machine in the cluster, but with different arguments

```
import sys
import tensorflow as tf

# Specify the cluster's architecture
cluster = tf.train.ClusterSpec(
                    {'ps': ['192.168.1.1:1111'],
                     'worker': ['192.168.1.2:1111','192.168.1.3:1111']})

# Parse command-line to specify machine
job_type = sys.argv[1]  # job type: "worker" or "ps"
task_idx = sys.argv[2]  # index job in the worker or ps list
                        # as defined in the ClusterSpec

# Create TensorFlow Server.
# This is how the machines communicate.
server = tf.train.Server(cluster, job_name=job_type,
                         task_index=task_idx)
```

```
# Parameter server is updated by remote clients.
# Will not proceed beyond this if statement.
if job_type == 'ps':
   server.join()
else:
   # Workers only
   with tf.device(tf.train.replica_device_setter(
                  worker_device='/job:worker/task:'+task_idx,
                  cluster=cluster)):
      # Build your model here
      # as if you only were using a single machine

    with tf.Session(server.target):
       # Train your model here
```

Distributed training in TensorFlow (5)

Running distributed TensorFlow on Piz Daint

- Write a Python script (TF_SCRIPT) accepting job_name, task_index, ps_hosts and worker_hosts TensorFlow flags
- Write a Bash script like the following one; run_dist_tf_daint.sh will specify the cluster from allocated nodes

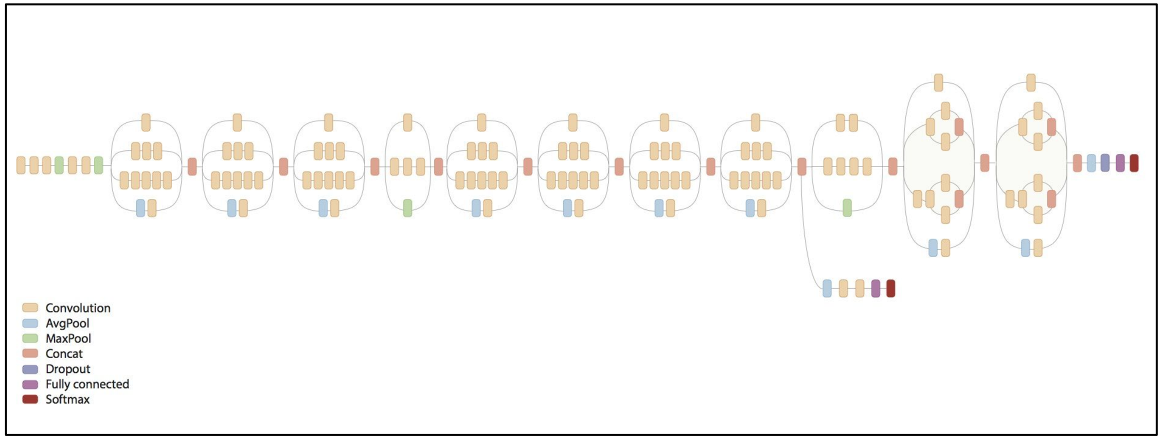
```
#!/bin/bash

#SBATCH --job-name=distributed_tf
#SBATCH --time=00:12:00
#SBATCH --nodes=8
#SBATCH --constraint=gpu
#SBATCH --output=distributed_tf.%j.log

# Arguments:
#   $1: TF_NUM_PS: number of parameter servers
#   $2: TF_NUM_WORKER: number of workers

# load modules
module load daint-gpu
module load TensorFlow
```

```
# set TensorFlow script parameters
export TF_SCRIPT="$HOME/project_dir/project_script.py"

export TF_FLAGS="--num_gpus=1 --batch_size=64
                 --num_batches=4 --data_format=NCHW"

# set TensorFlow distributed parameters
export TF_NUM_PS=$1 # 1
export TF_NUM_WORKERS=$2 # $SLURM_JOB_NUM_NODES
# export TF_WORKER_PER_NODE=1
# export TF_PS_PER_NODE=1
# export TF_PS_IN_WORKER=true

# run distributed TensorFlow
DIST_TF_LAUNCHER_DIR=$SCRATCH/run_dist_tf_daint_dir
cd $DIST_TF_LAUNCHER_DIR
./run_dist_tf_daint.sh
```

ToC

Benchmarks (1)

- Model
  - InceptionV3
    - Neural Network for 1000-class image classification (ImageNet competition)
  - Optimized code for benchmarks available from Google
- Data set
  - ImageNet: 1,280,000 images (144 GB)
- TensorFlow 1.1.0
- Performance metric
  - Number of trained images per second



Benchmarks (2)

- Methodology
  - For each configuration of number of Workers and number of nodes
    - Run with different number of Parameter Servers on synthetic data (no I/O access)
    - Report best setting of number of Workers and number of PSs
    - Run best setting on real data (with I/O access)
  - Results repeatability
    - Run each test 5 times and average times together (Google's approach)
      - Compare results with Google's
      - Limit impact on Piz Daint (200+ tests)
  - For each test
    - 10 warmup steps
    - Next 100 steps are averaged

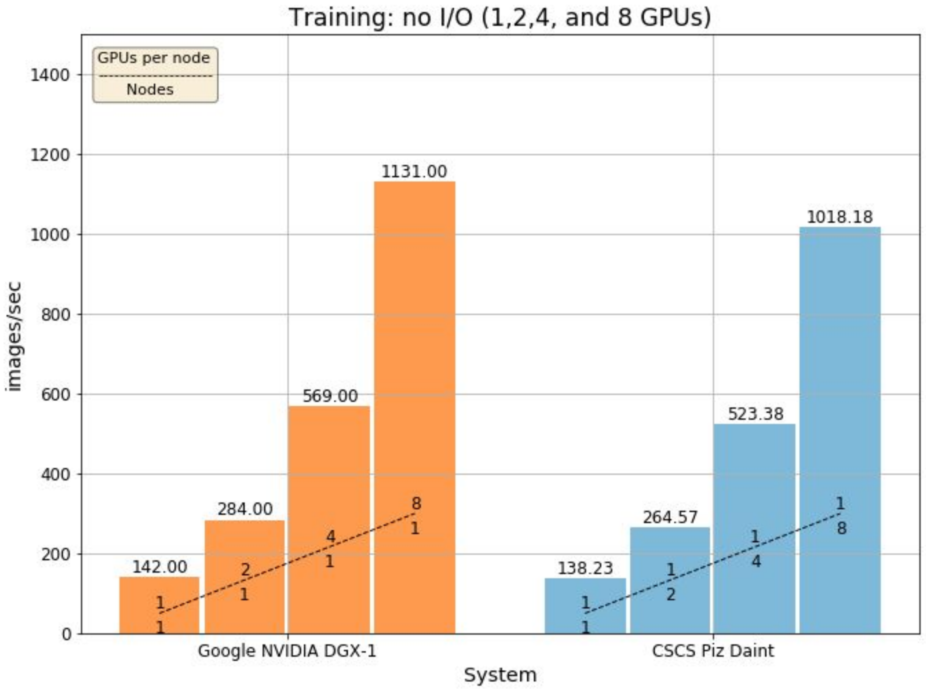
Benchmarks (3)

- Systems
  - Piz Daint (NVIDIA Tesla P100 - 1 GPU per node)
  - Amazon EC2 instances
    - p2.xlarge (NVIDIA Tesla K80 - 1 GPU per node)
    - p2.8xlarge (NVIDIA Tesla K80 - 8 GPUs per node)

- Benchmarks from Google available at https://www.tensorflow.org/performance/benchmarks
  - Google’s systems
    - NVIDIA DGX-1 (NVIDIA Tesla P100 - 8 GPUs per node)
    - Amazon p2.8xlarge (NVIDIA Tesla K80 - 8 GPUs per node)

Benchmarks (4)

NVIDIA Tesla P100 - synthetic data (no I/O) - up to 8 GPUs

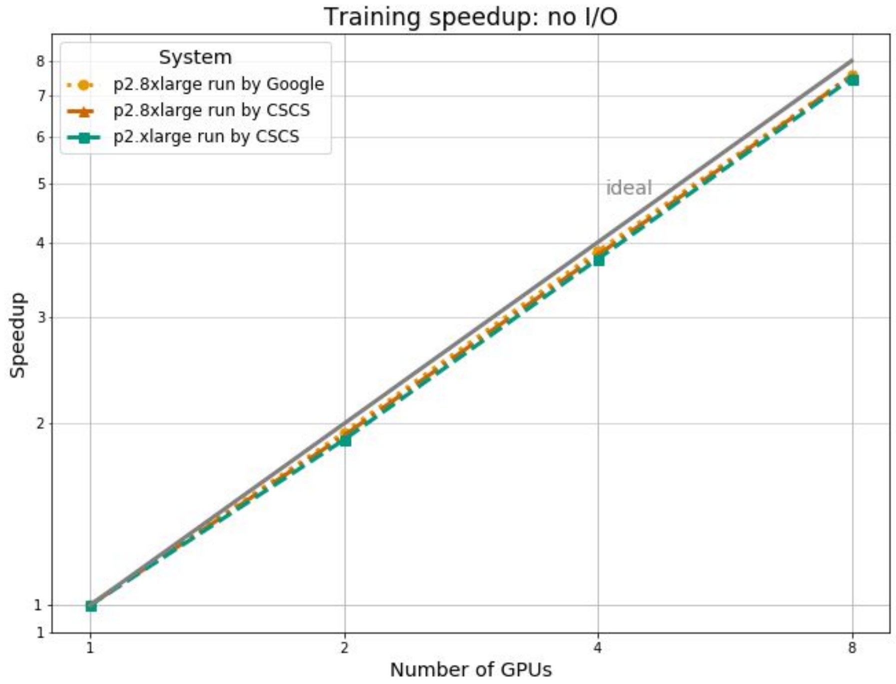


Scalability efficiency

- *99.56%* on 8 GPUs in NVIDIA DGX-1
- *92.07%* on 8 GPUs in Piz Daint
- 8 nodes in Piz Daint have similar performance as an NVIDIA DGX-1

Benchmarks (5)

NVIDIA Tesla K80 - synthetic data (no I/O) - up to 8 GPUs

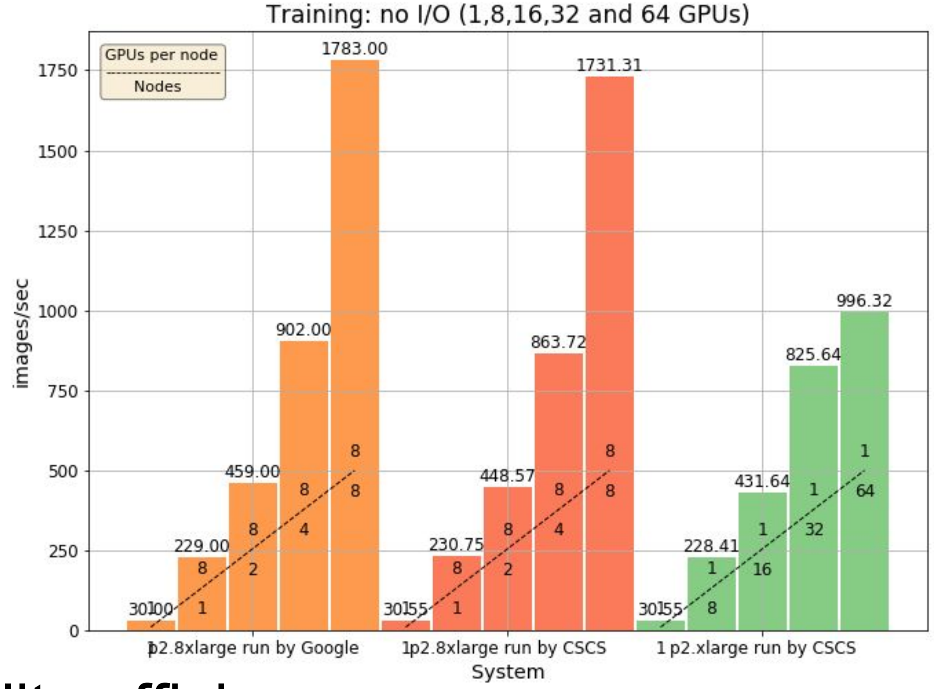


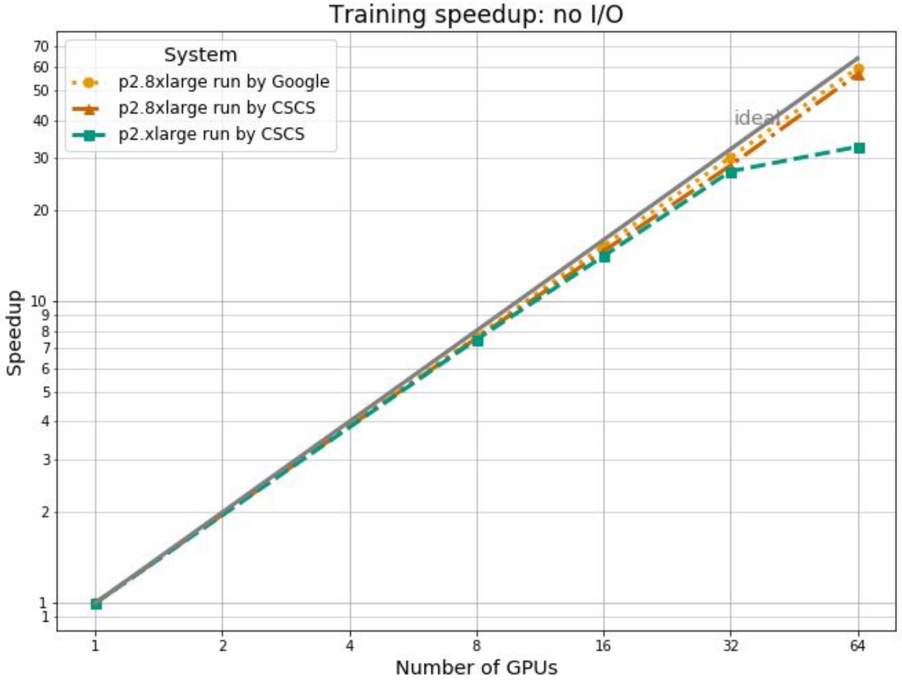
Scalability efficiency

- *94.58%* and *94.44%* on 8 GPUs in p2.8xlarge
- *93.45%* on 8 GPUs in p2.xlarge
- Up to 8 GPUs, compute bound application

Benchmarks (6)

NVIDIA Tesla K80 - synthetic data (no I/O) - up to 64 GPUs



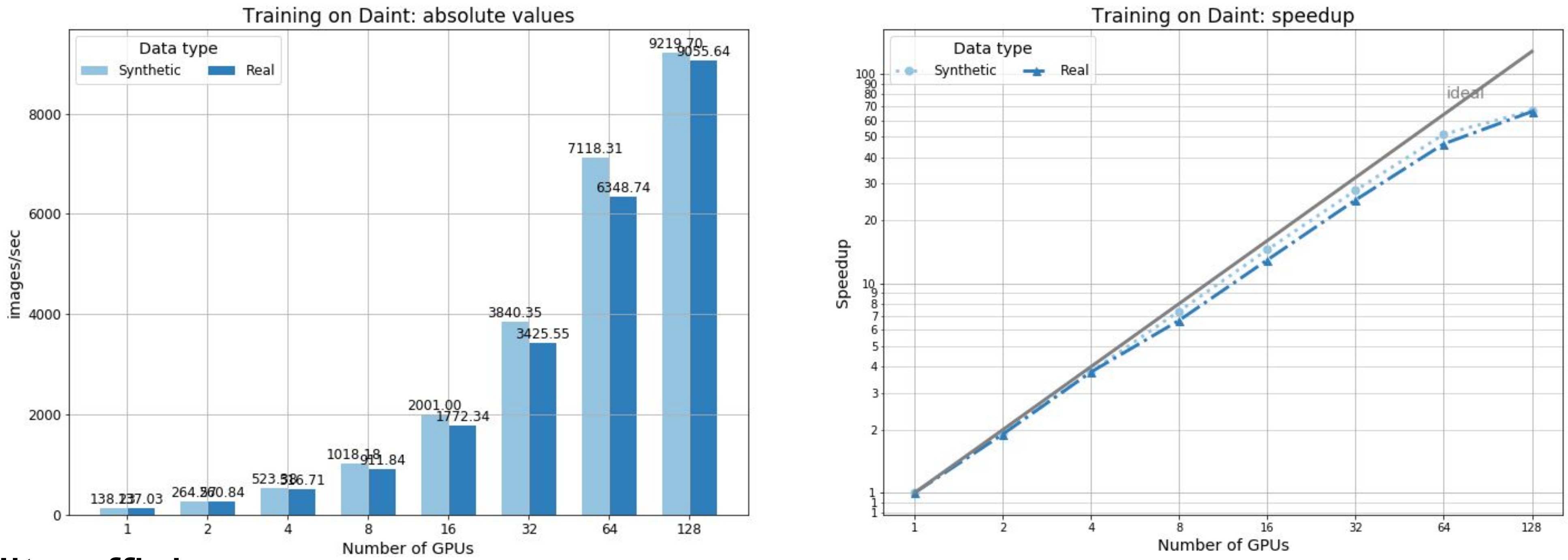


Scalability efficiency

- *92.86%* and *88.55%* on 64 GPUs in p2.8xlarge
- *50.96%* on 64 GPUs in p2.xlarge
- Intuition: inter-node network capacity reached with 64 GPUs in p2.xlarge

Benchmarks (7)

Piz Daint (NVIDIA Tesla P100) - synthetic and real data - up to 128 GPUs

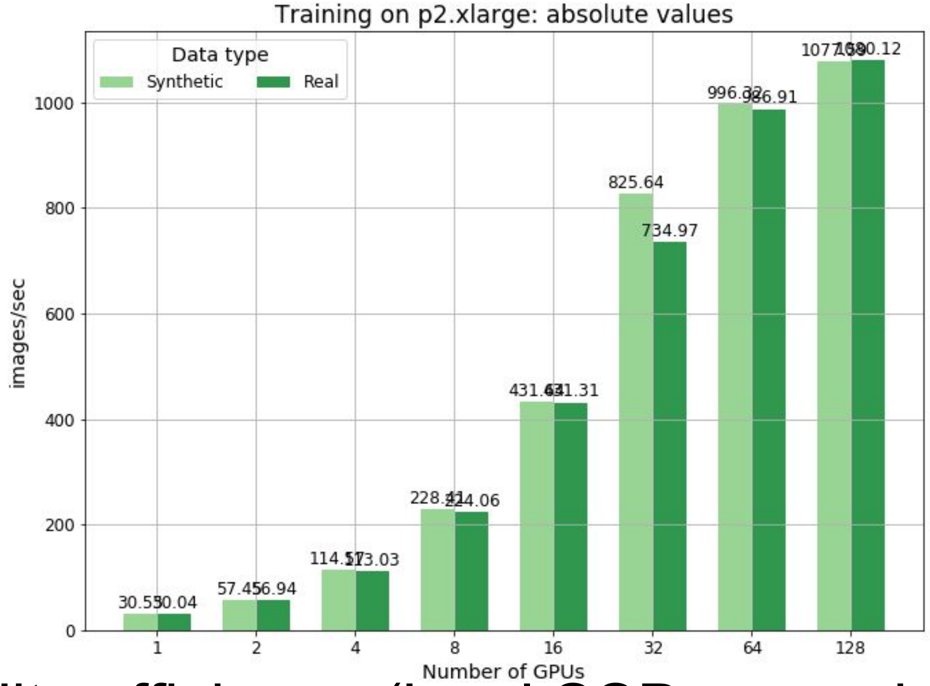


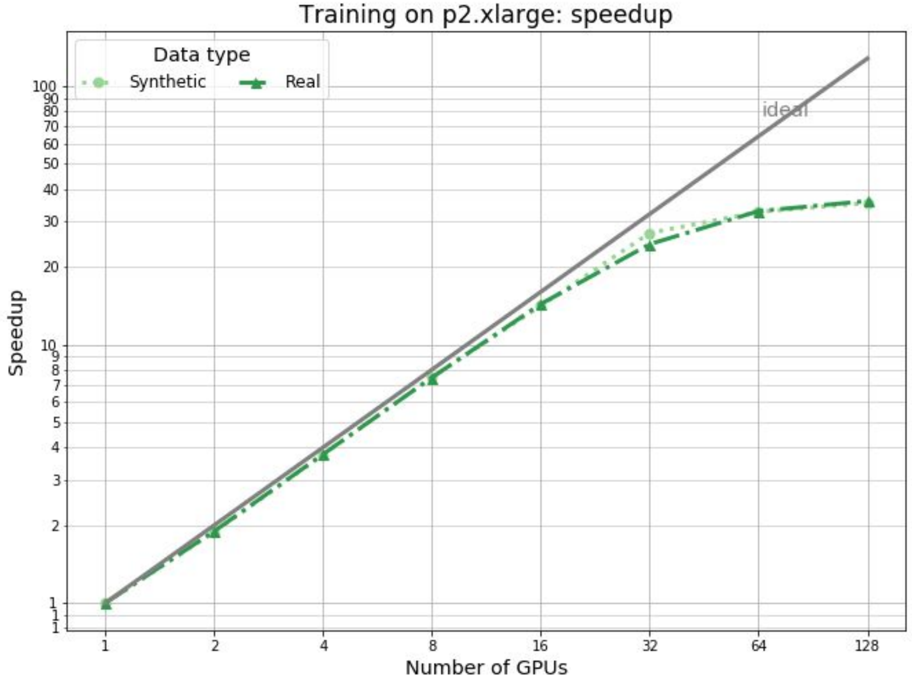
Scalability efficiency

- *80.46%* (synthetic) and *72.39%* (real) on 64 GPUs
- *52.11%* (synthetic) and *51.63%* (real) on 128 GPUs
- Intuition: inter-node network capacity reached with 128 nodes

Benchmarks (8)

p2.xlarge (NVIDIA Tesla K80) - synthetic and real data - up to 128 GPUs



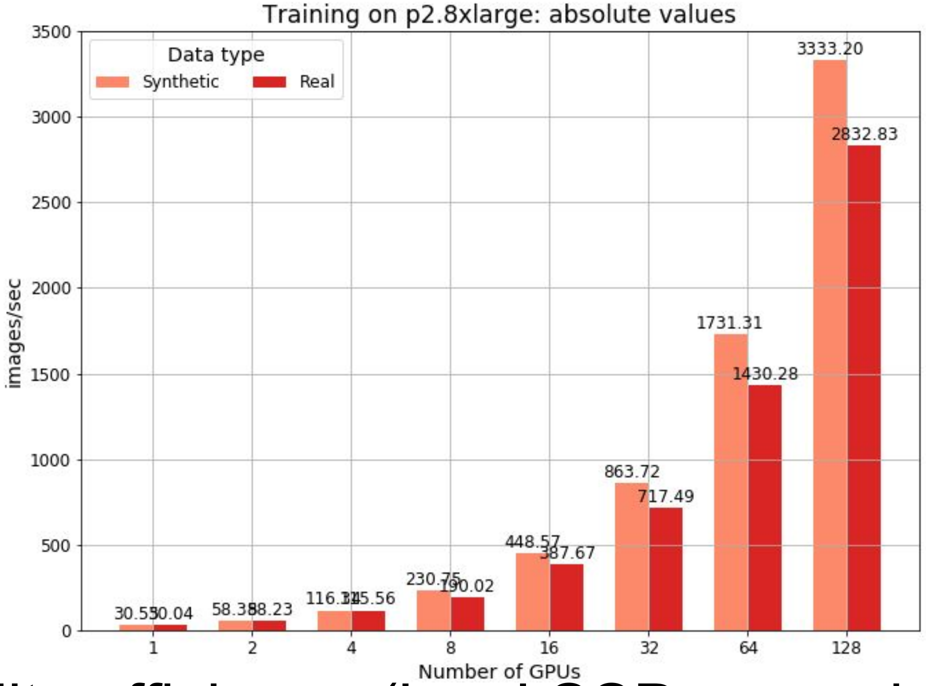


Scalability efficiency (local SSD on each node)

- *50.96%* (synthetic) and *51.33%* (real) on 64 GPUs
- *27.56%* (synthetic) and *28.09%* (real) on 128 GPUs
- Intuition: inter-node network capacity reached with 64 nodes

Benchmarks (9)

p2.8xlarge (8 * NVIDIA Tesla K80) - synthetic and real data - up to 128 GPUs

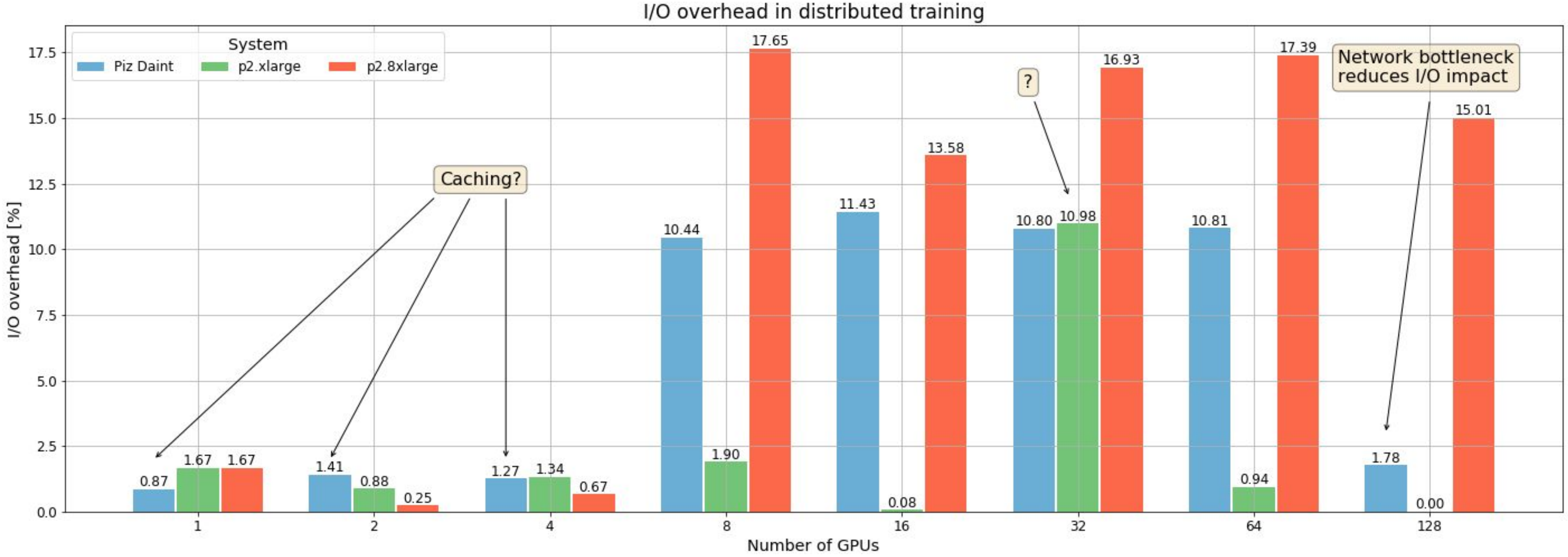


Scalability efficiency (local SSD on each node)

- *88.55%* (synthetic) and *74.39%* (real) on 64 GPUs
- *85.24%* (synthetic) and *73.67%* (real) on 128 GPUs
- Intuition: inter-node network capacity not reached (only 16 nodes for 128 GPUs)

Benchmarks (9)

I/O overhead



- *~17%* on p2.8xlarge when 8 GPUs per node are used
- *~1%* on p2.xlarge
- *~11%* on Piz Daint when 8 to 64 GPUs used, *~1.5%* otherwise

ToC

Conclusion

- 8 nodes in Piz Daint have similar performance to 1 NVIDIA DGX-1
- Scalability for InceptionV3 in TensorFlow
  - On Piz Daint
    - Supposedly inter-node bandwidth capacity reached after 64 nodes
    - I/O cost ~11%
  - On a multi-GPU system
    - Inter-node traffic algorithmically reduced by the number of GPUs per node (interconnect seems to have no real impact)
    - Using local SSDs and 8 GPUs per node adds a constant ~17% I/O overhead (PCIe traffic)
    - No benchmarks available for multiple NVIDIA DGX-1
  - ⇒ Estimation according to the examined use case: Similar performance between 64 nodes on Piz Daint and 8 NVIDIA DGX-1 connected by a reasonable inter-node network

Future Work

- Investigate impact of training accuracy in distributed setting (preliminary results)
- Profile TensorFlow communication patterns
- Analyze influence of number of PSs for single- and multi-GPU systems

Conclusion

- 8 nodes in Piz Daint have similar performance to 1 NVIDIA DGX-1
- Scalability for InceptionV3 in TensorFlow
  - On Piz Daint
    - Supposedly inter-node bandwidth capacity reached after 64 nodes
    - I/O cost ~11%
  - On a multi-GPU system
    - Inter-node traffic algorithmically reduced by the number of GPUs per node (interconnect seems to have no real impact)
    - Using local SSDs and 8 GPUs per node adds a constant ~17% I/O overhead (PCIe traffic)
    - No benchmarks available for multiple NVIDIA DGX-1
  - ⇒ Estimation according to the examined use case: Similar performance between 64 nodes on Piz Daint and 8 NVIDIA DGX-1 connected by a reasonable inter-node network

Thank you

Future Work

- Investigate impact of training accuracy in distributed setting (preliminary results)
- Profile TensorFlow communication patterns
- Analyze influence of number of PSs for single- and multi-GPU systems

Backup slides

TensorFlow overview (1)

TensorFlow is based on data flow graphs

- Nodes represent mathematical operations
- Tensors move across the edges between nodes

Writing a TensorFlow application

1. Build computation graph
2. Run instances of that graph

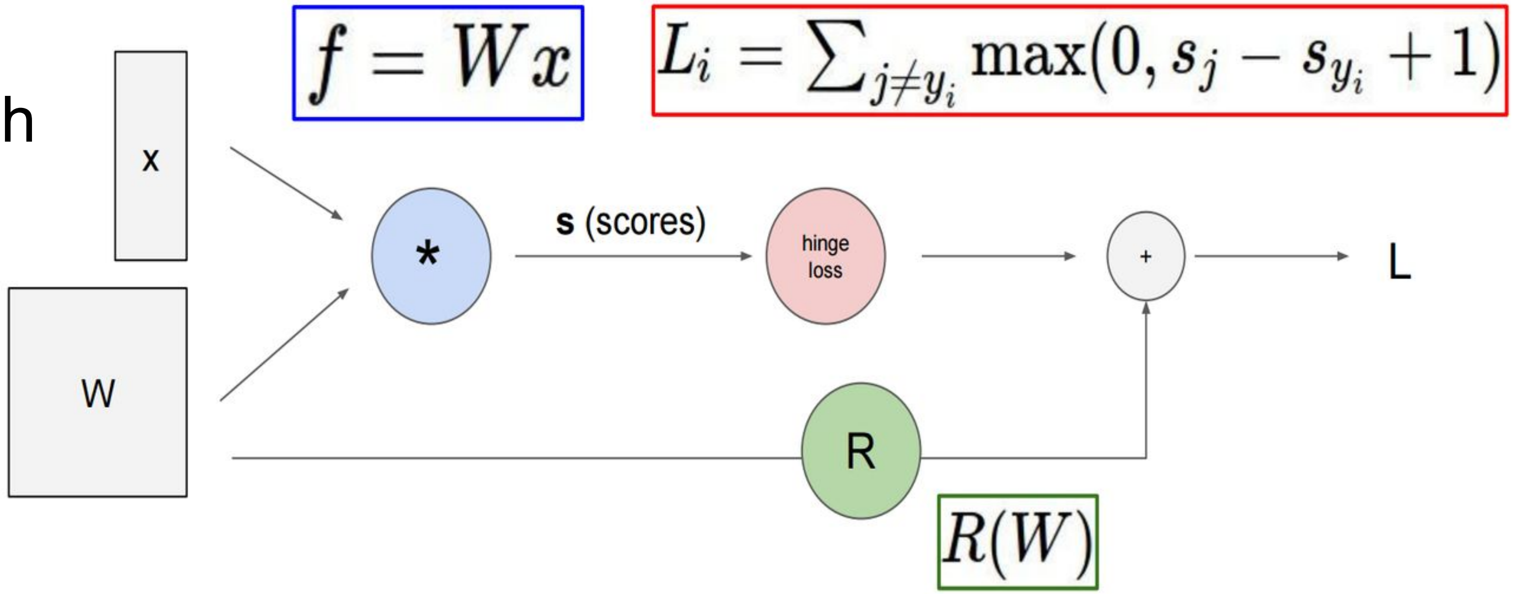


Figure 1: Computational graph for regularized Multiclass SVM loss (CS231N, Stanford University)

TensorFlow overview (2)

Example: Linear Regression in TensorFlow

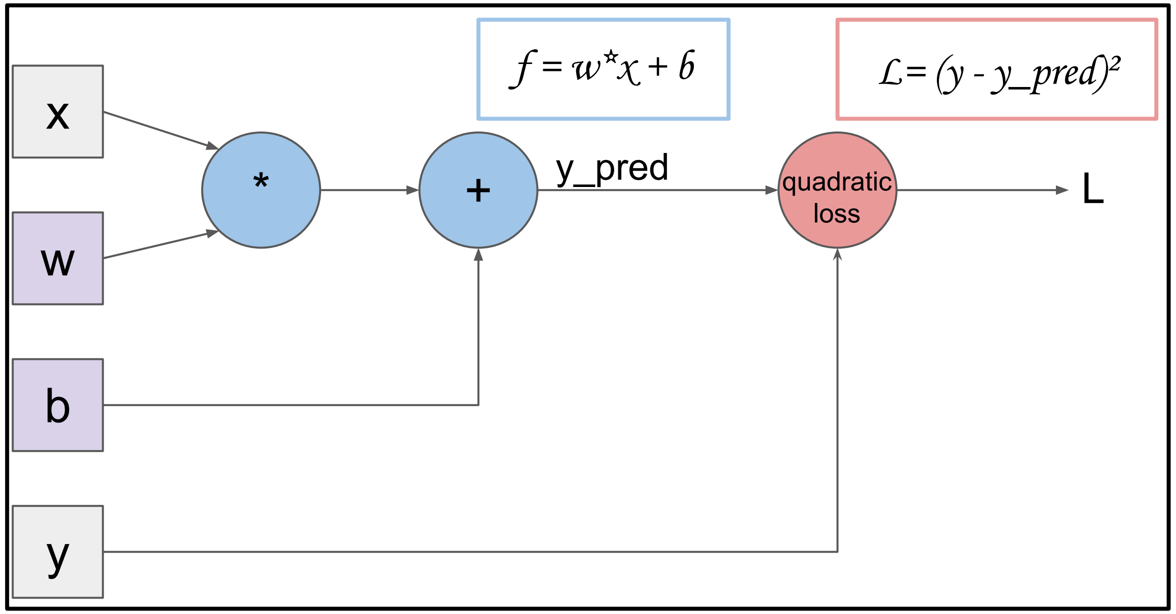


Figure 2: Computational graph for Linear Regression with squared loss

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

# ============================================= #
#                   LOAD DATA                   #
# ============================================= #
# Generate some data as y=3*x + noise
N_SAMPLES = 10
x_in = np.arange(N_SAMPLES)
y_in = 3*x_in + np.random.randn(N_SAMPLES)
data = list(zip(x_in, y_in))
```

TensorFlow overview (3)

Example: Linear Regression in TensorFlow

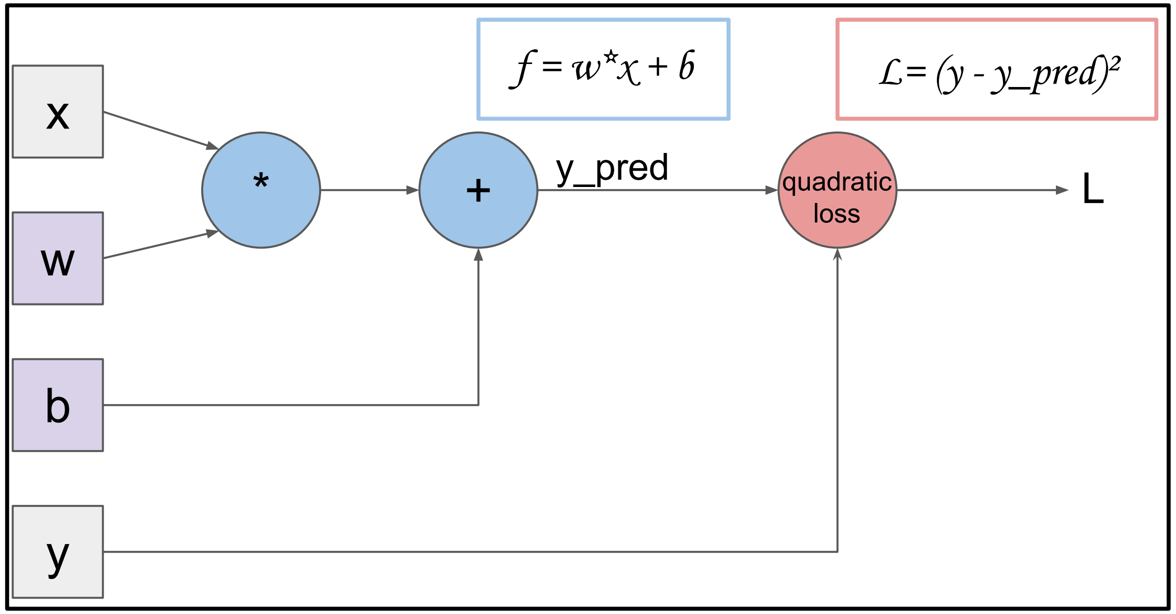


Figure 2: Computational graph for Linear Regression with squared loss

```
# ========================================================== #
#                         BUILD GRAPH                         #
# ========================================================== #
simple_graph = tf.Graph()
with simple_graph.as_default():
   # Generate placeholders for input x and output y
   x = tf.placeholder(tf.float32, name='x')
   y = tf.placeholder(tf.float32, name='y')

   # Create weight and bias, initialized to 0
   w = tf.Variable(0.0, name='weight')
   b  = tf.Variable(0.0, name='bias')

   # Build model to predict y
   y_predicted = x * w + b

   # Use the square error as the loss function
   loss = tf.square(y - y_predicted, name='loss')

   # Use gradient descent to minimize loss
   optimizer = tf.train.GradientDescentOptimizer(0.001)
   train = optimizer.minimize(loss)
```

TensorFlow overview (4)

Example: Linear Regression in TensorFlow

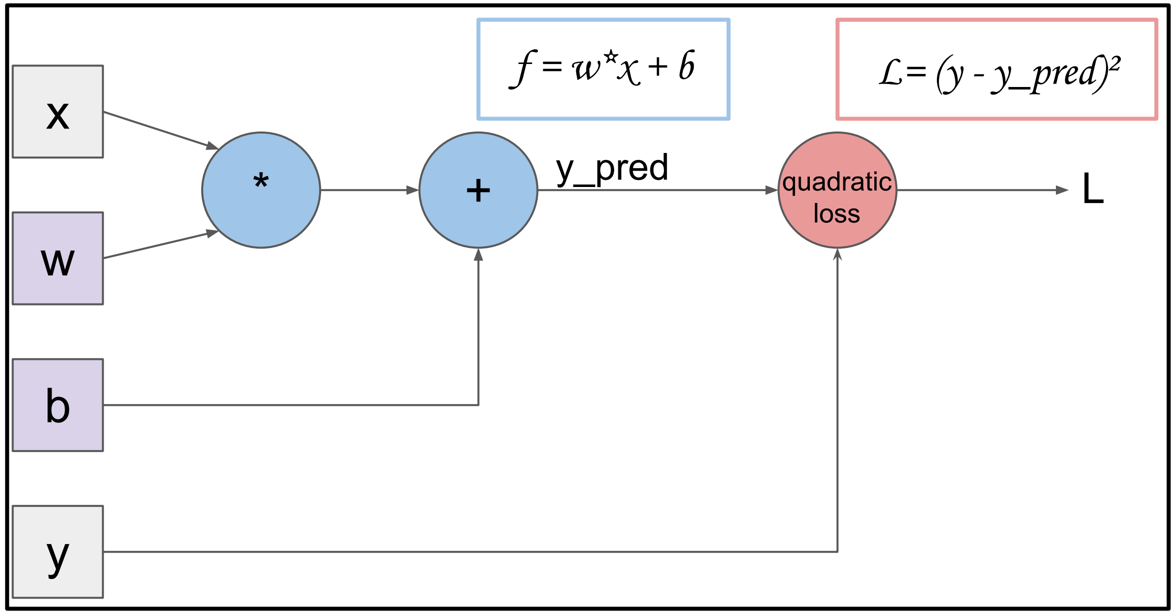


Figure 2: Computational graph for Linear Regression with squared loss

```
# ========================================================== #
#                       EXECUTE GRAPH                        #
# ========================================================== #
# Run training for N_EPOCHS epochs
N_EPOCHS = 5
with tf.Session(graph=simple_graph) as sess:
    # Initialize the necessary variables (w and b here)
    sess.run(tf.global_variables_initializer())

    # Train the model
    for i in range(N_EPOCHS):
        total_loss = 0
        for x_,y_ in data:
            # Session runs train operation and fetches values of loss
            _, l_value = sess.run([train, loss], feed_dict={x: x_, y: y_})
            total_loss += l_value
        print('Epoch {0}: {1}'.format(i, total_loss/N_SAMPLES))

    # Output final values of w and b
    w_value, b_value = sess.run([w, b])
```

TensorFlow overview (5)

Example: Linear Regression in TensorFlow

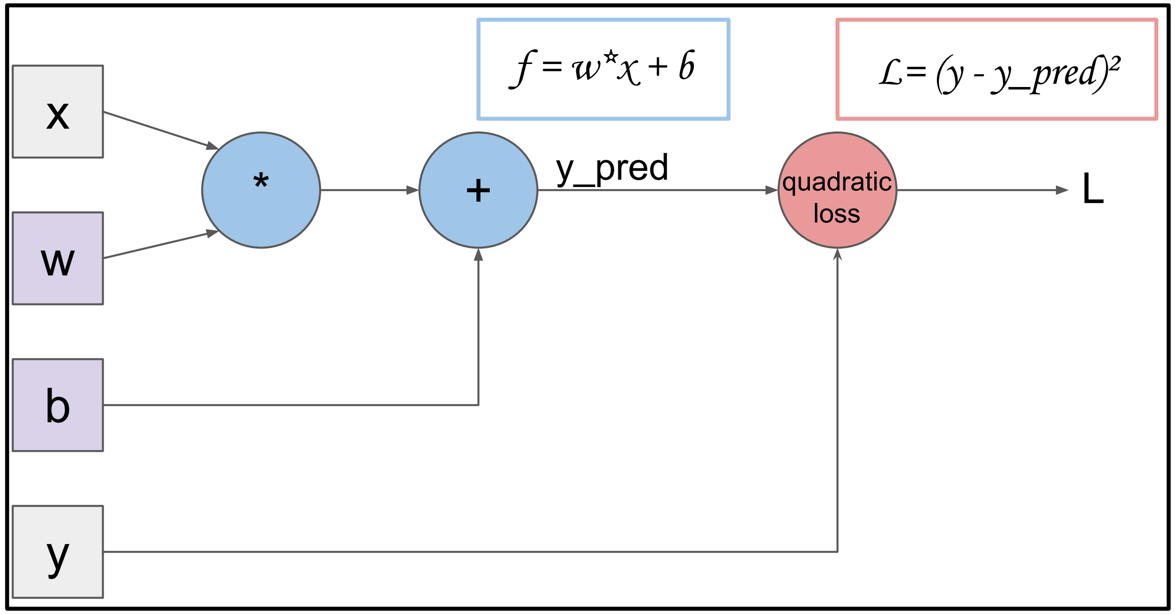


Figure 2: Computational graph for Linear Regression with squared loss

```
# ============================================== #
#                  PLOT RESULTS                  #
# ============================================== #
print(w_value, b_value) # 2.89, 0.45
plt.plot(x_in, y_in, 'bo', label='Real data')
plt.plot(x_in, x_in*w_value + b_value, 'orange',
         label='Predicted data')
plt.ylabel('y');plt.xlabel('x')
plt.title('Linear Regression')
plt.legend();plt.grid()
plt.show()
```

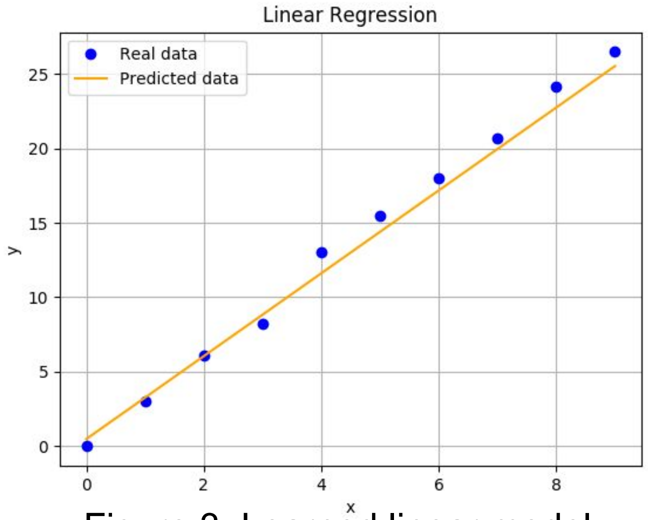
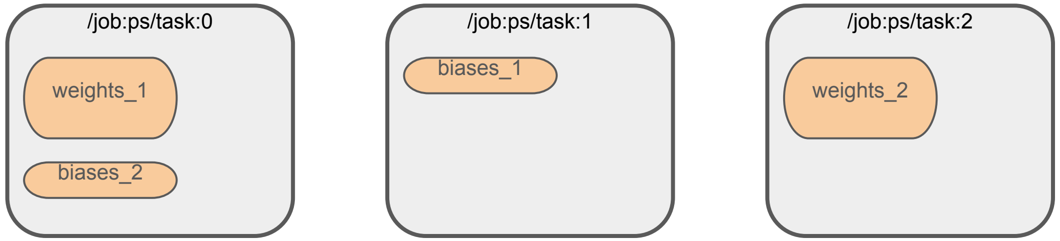


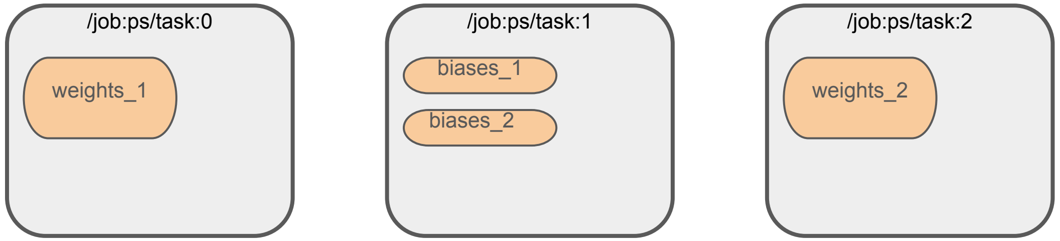
Figure 3: Learned linear model

Distributed training in TensorFlow (5)

Round-robin variables



Greedy load balancing variables



replica_device_setter provides two load balancing strategies

- Round-robin (default)
- Greedy load balancing

```
greedy =  tf.contrib.training.GreedyLoadBalancingStrategy(...)
with tf.device(
            tf.train.replica_device_setter(ps_tasks=3,
                                           ps_strategy=greedy)):
   weights_1 = tf.get_variable('weights_1', [784, 100])
   biases_1 = tf.get_variable('biases_1', [100])
   weights_2 = tf.get_variable('weights_2', [100, 10])
   biases_2 = tf.get_variable('biases_2', [10])
```