A Quick Survey on Deep Learning Engines

Chensong Zhang

April 27, 2017

1 Introduction

1.1 Background

Unlike traditional numerical simulation, "ML gives computers the ability to learn without being explicitly programmed". As a research field, ML explores the study and construction of algorithms that can learn from and make predictions on data. General Tasks of ML include:

- Classification: Inputs are divided into two or more classes, and the learner must produce a model that assigns unseen inputs to one or more of these classes
- Regression: Similar to classification, but the outputs are continuous
- Clustering: Inputs are divided into several groups (Unlike in classification, the groups are not known beforehand, making this an unsupervised task)
- Density estimation, dimension reduction, ...

There are many machine learning engines available; see the following picture:

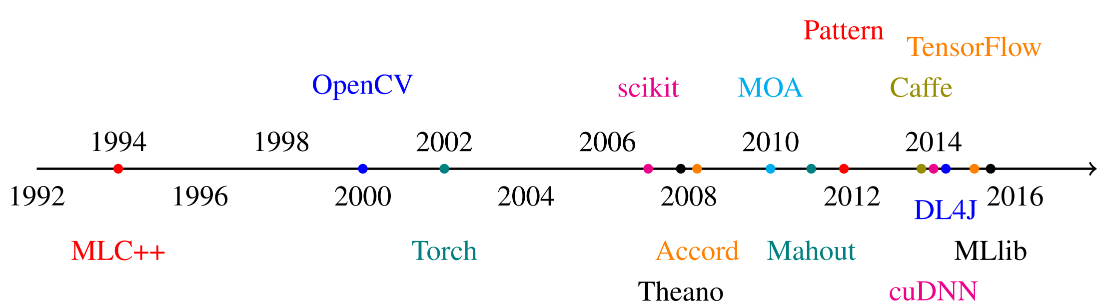


Figure 1: Refer to Tour of TensorFlow, by Peter Goldsborough, arXiv, 2016

Besides the packages mentioned in the above figure, there are a few other worth-noting ML (DL) packages:

- PyTorch (still in beta stage)
- Theano (Lasagne/Keras): Slow in graph compilation
- MXNet (DMLC): First released on Jan 2015, scalable distributed computing
- CNTK/DMTK (Microsoft): First released on April 2015
  - Windows/Linux, no official OS X support thought
  - C++/Python front-end
- Neon (Nervana & Intel): First released on May 2015, professional support
- Caffe2 (Google, Facebook, ...): Release on late 2016
- Digits (Nvidia, powered by Caffe): Web interface

1.2 DL engines

In this talk, we shall talk about the most popular DL engines that are freely available. So what do we want for a software package which are suitable for deep learning research?

- Easy for debugging (with good documentation, support, and active community)
- Good flexibility
  - Easy to add new tasks
  - Easy to build new network structures
- Good performance and scalability
  - Multicore CPU
  - Single or multiple GPU(s)
  - Cluster

Among dozens of available software packages, we mainly look at Torch, Caffe, TensorFlow, and MXNet; see Table 1 for the basic information about these four packages.

2 Comparison

2.1 Test setting

In order to compare performance of these software packages on CPU and GPU platforms, we look at the recent numerical study carried out in

Benchmarking State-of-the-Art Deep Learning Software Tools, by S.-H. Shi, et al., arXiv, 2017

Viewpoint	Torch	Caffe	TensorFlow	MXNet
First Released	2002	2013	2015	2015
Main Developers	Facebook, Twitter, Google, ...	BAIR BVLC	Google	DMLC
Core Languages	C/Lua	C++	C++ Python	C++
Supported Interface	Lua	C++/Python Matlab	Python/C++/R Java/Go/...	C++/Python/R Matlab/Julia/...
License	BSD	BSD	Apache	Apache
Pretrained Models	Yes	Yes	No	Yes
High-level Support	Good	Good	Good	Good
Low-level Operators	Good	Good	Fairly good	Increasing fast
Speed One-GPU	Great	Great	Good	Good
Memory Management	Great	Great	Not so good	Excellent
Parallel Support	Multi-GPU	Multi-GPU	Multi-GPU	Distributed
Coding Style	Imperative	Declarative	Declarative	Mixed
GitHub Watching	649/268	1856	4939	887

Table 1: Basic information on popular DL engines

Computational Unit	Cores	Memory	OS	CUDA
Intel CPU i7-3820	4	64 GB	Ubuntu 14.04	-
Intel CPU E5-2630x2	16	128 GB	CentOS 7.2	-
GTX 1080	2560	8 GB	Ubuntu 14.04	8
Telsa K80 GK210	2496	12 GB	CentOS 7.2	8

Figure 2: The experimental hardware settings for numerical tests

Networks		Input	Output	Layers	Parameters
FCN	FCN-S	26752	26752	5	~55 millions
FCN	FCN-R	784	10	5	~31 millions
CNN	AlexNet-S	150528	1000	4	~61 millions
CNN	AlexNet-R	3072	10	4	~81 thousands
RNN	LSTM	10000	10000	2	~13 millions

Figure 3: The experimental setup of neural networks for synthetic and real data

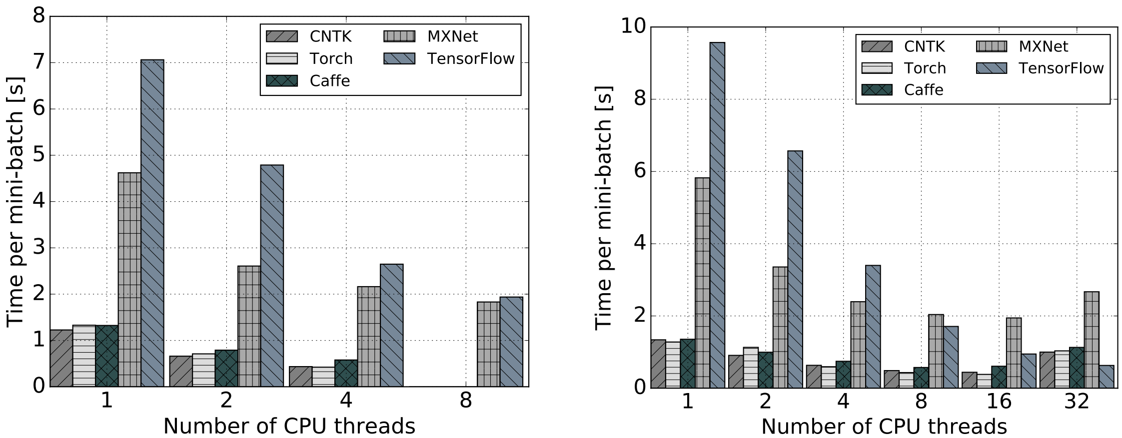
In this paper, different network models are tested on several consumer-class computing platforms; see Figures 2 and 3

- A large fully-connected neural network (FCN-S) with around 55 million parameters is used to evaluate the performance of FCN;
- The classical AlexNet (AlexNet-S) is used as an representative of CNN;
- A smaller FCN (FCN-R) is constructed for MNIST data set;
- An AlexNet (AlexNet-R) architecture is used for Cifar10 data set;
- For RNNs, considering that the main computation complexity is related to the length of input sequence, 2 LSTM layers with input length of 32.

2.2 CPU tests

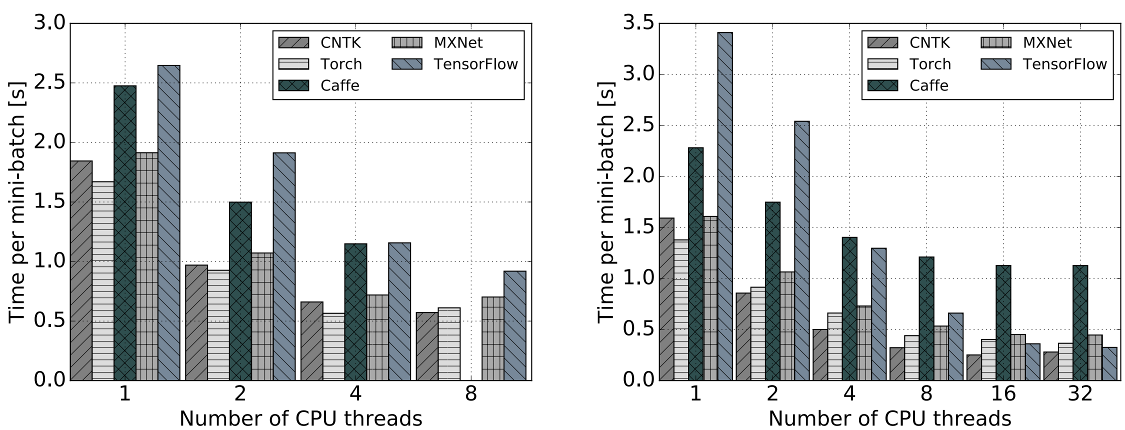
From the numerical results on CPUs, we have the following observations:

- CNTK/Torch/Caffe have similar CPU performance
- TensorFlow has excellent scalability
- All engines have good CPU performance
- TensorFlow has good scalability but considerably slower
- Caffe has best CNN performance as promised
- MXNet does not scale well for this test
- Good scalability of TensorFlow kicks in
- Caffe does not scale well on multicore CPUs
- Pay more attention to CNTK in the future
- Caffe/MXNet does not support LSTM on CPUs



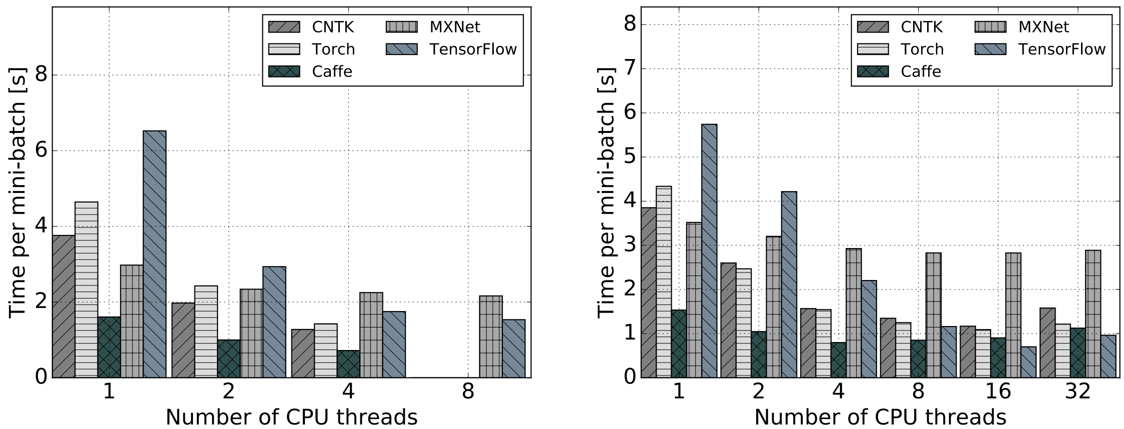
(a) Results on i7-3820. (b) Results on E5-2630.

Figure 4: FCN-S performance on CPU platform with a mini-batch size of 64



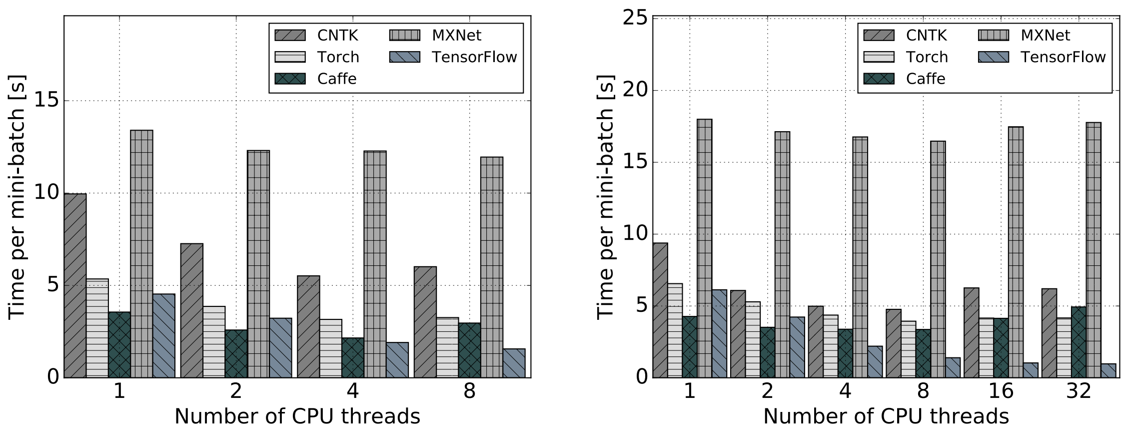
(a) Results on i7-3820. (b) Results on E5-2630.

Figure 5: The FCN-R performance on CPU platform with a mini-batch size of 1024



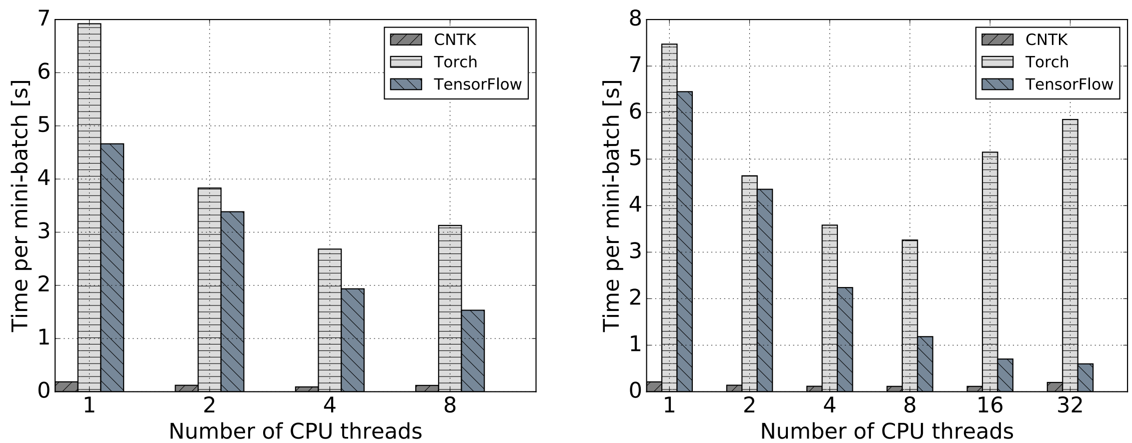
(a) Results on i7-3820. (b) Results on E5-2630.

Figure 6: AlexNet-S performance on CPU platform with a mini-batch size of 16



(a) Results on i7-3820. (b) Results on E5-2630.

Figure 7: AlexNet-R performance on CPU platform with a mini-batch size of 1024



(a) Results on i7-3820.

(b) Results on E5-2630.

Figure 8: LSTM performance on CPU platform with a mini-batch size of 256

2.3 GPU tests

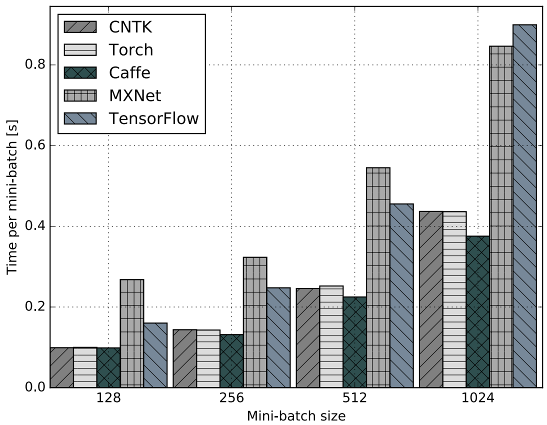
From the numerical results on GPUs, we have the following observations:

- CNTK/Torch/Caffe out-perform the others
- All packages have similar performance
- MXNet out-perform the others for CNN on GPUs
- TensorFlow does not have good GPU performance in general
- CNTK has excellent RNN performance both on CPU and GPU
- Multi-GPU can greatly boost the training process of network
- MXNet shows overwhelming advantage over the others
- TensorFlow does not scale well on multi-GPU platform

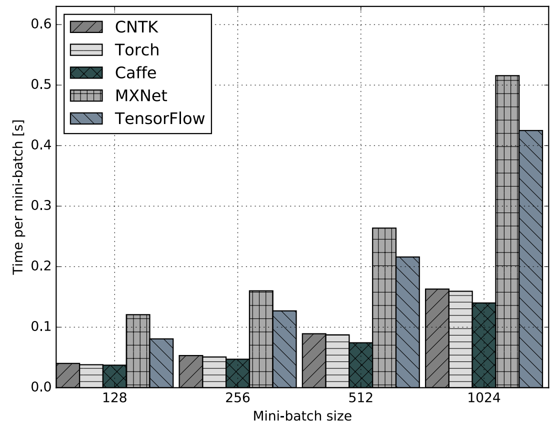
3 TensorFlow

3.1 Computational graph

- Graph: In TensorFlow, ML algorithms are represented as computational graph. A computational graph (data flow graph) is a form of directed graph where vertices (nodes) describe operations, while edges represent data flowing between these operations.

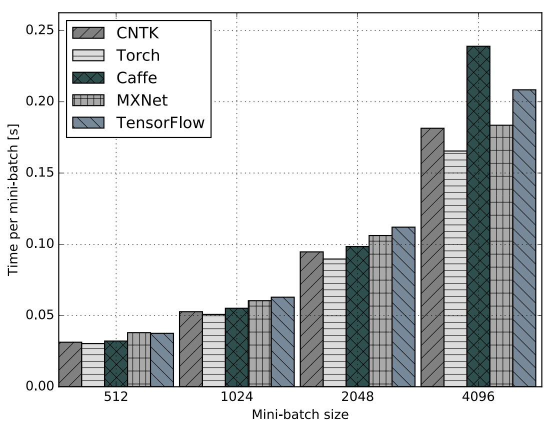


(a) Results on Tesla K80.

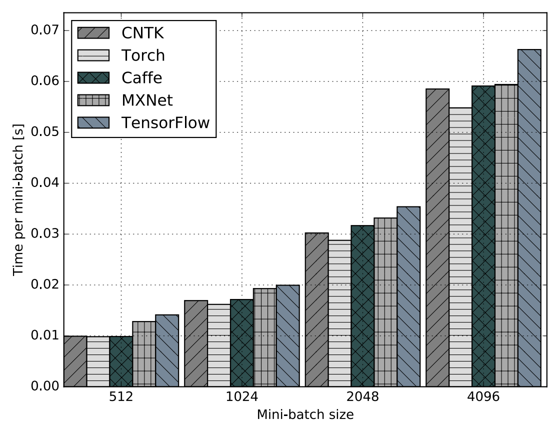


(b) Results on GTX1080.

Figure 9: The performance comparison of FCN-S on GPU platforms

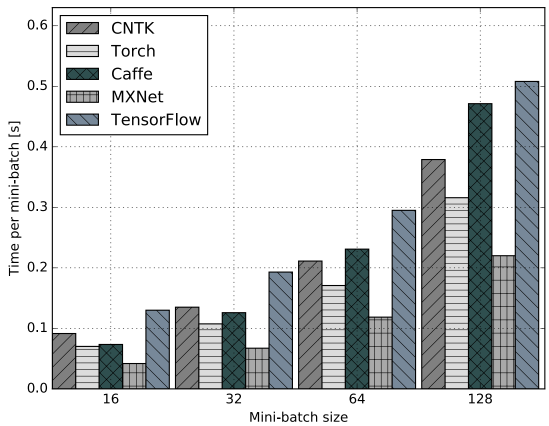


(a) Results on Tesla K80.

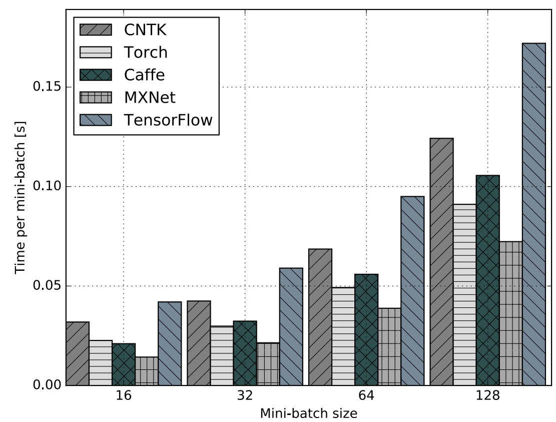


(b) Results on GTX1080.

Figure 10: The performance comparison of FCN-R on GPU platforms

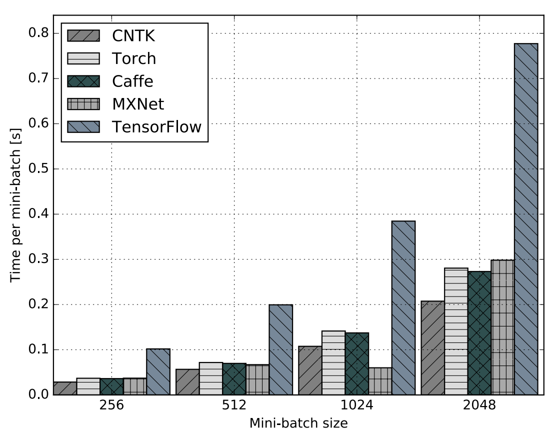


(a) Results on Tesla K80.

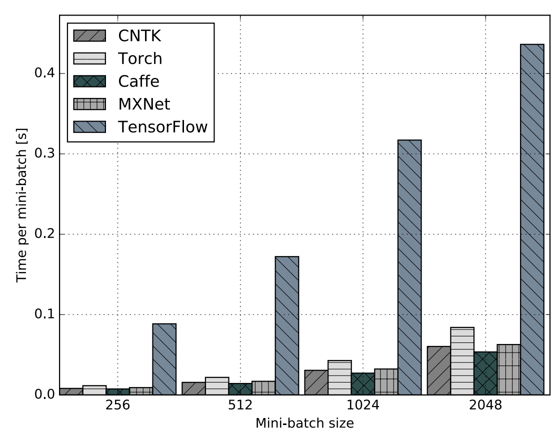


(b) Results on GTX1080.

Figure 11: The performance comparison of AlexNet-S on GPU platforms

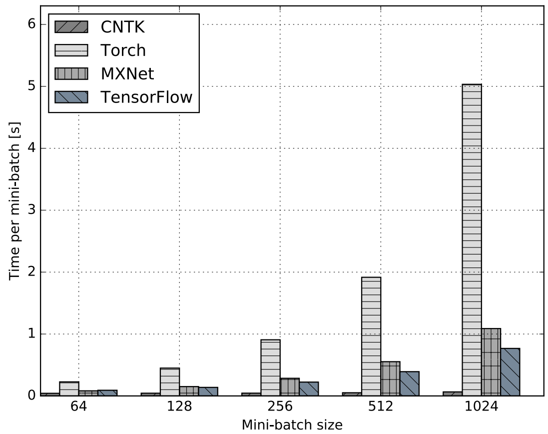


(a) Results on Tesla K80.

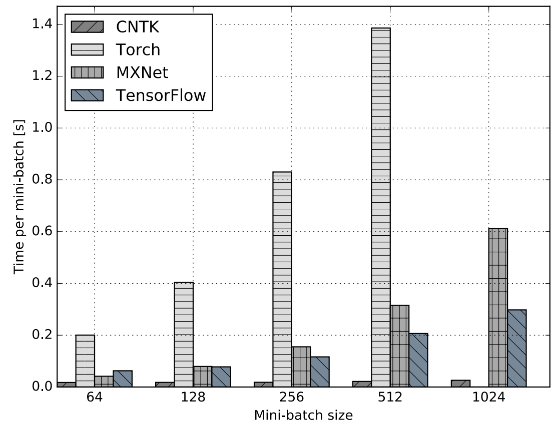


(b) Results on GTX1080.

Figure 12: The performance comparison of AlexNet-R on GPU platforms

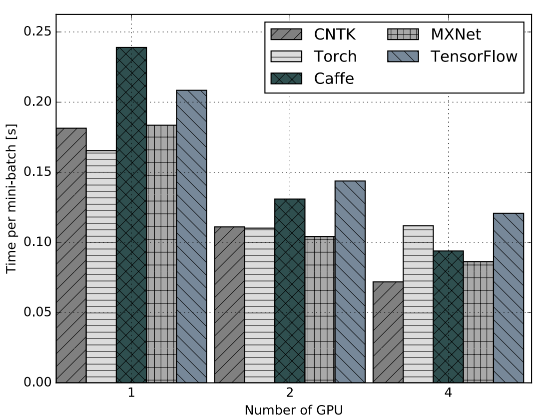


(a) Results on Tesla K80.

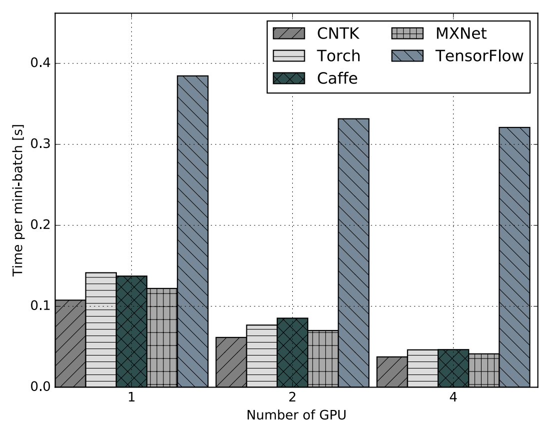


(b) Results on GTX1080.

Figure 13: The performance comparison of LSTM on GPU platforms



(a) FCN-R



(b) CNN-R

Figure 14: The scalability on a multi-GPU platform (2 $\times$ K80)

- Operation: An operation may represent a variable or constant, a control flow directive, a mathematical function, a file I/O, or a network communication port.
- Tensor: A tensor is a multi-dimensional collection of homogeneous values with a fixed static type.
- Variable: Variables can be described as persistent, mutable handles to in-memory buffers storing tensors.
- Session: In TensorFlow the execution of operations and evaluation of tensors may only be preformed in a special environment called session.

3.2 A simple example

```
import tensorflow as tf

# Import training and test data from MNIST
import tensorflow.examples.tutorials.mnist.input_data as input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

graph = tf.Graph()

with graph.as_default():

    # Nodes can be grouped into visual blocks for TensorBoard
    with tf.name_scope('input_features'):
        x = tf.placeholder(tf.float32, shape=[None, 784], name='input_x')

    with tf.name_scope('input_labels'):
        y_ = tf.placeholder(tf.float32, shape=[None, 10], name='labels')

    with tf.name_scope('parameters'):
        W = tf.Variable(tf.zeros([784, 10]), name='weights')
        b = tf.Variable(tf.zeros([10]), name='biases')
        tf.summary.histogram('WEIGHTS', W)
        tf.summary.histogram('BIASES', b)

    with tf.name_scope('use_softmax'):
        y = tf.nn.softmax(tf.matmul(x, W) + b)

    with tf.name_scope('train'):
        # Compute the cross entropy of real label y_ and prediction label y
        cross_entropy = -tf.reduce_sum(y_*tf.log(y))
        # Create a gradient-descent optimizer with learning rate = 0.01
        train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)

    with tf.name_scope('test'):
        correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
        # Track accuracy over time for TensorBoard
        tf.summary.scalar('Accuracy', accuracy)

    logpath = '/tmp/tensorboard'  # temporary path for storing TB summaries
    merged = tf.summary.merge_all()  # Merge all the summaries
    writer = tf.summary.FileWriter(logpath, graph)  # Write summaries

with tf.Session(graph=graph) as sess:
    # Initialize all variables
    tf.global_variables_initializer().run()

    for step in range(1,501):
        if (step%10) == 0:
            feed = {x: mnist.test.images, y_: mnist.test.labels}
            _, acc = sess.run([merged, accuracy], feed_dict=feed)
            print('Accuracy at %s step: %s' % (step, acc))
        else:
            batch_x, batch_y = mnist.train.next_batch(100)
            sess.run(train_step, feed_dict={x: batch_x, y_: batch_y})
```

```
            writer.add_summary(merged.eval(feed_dict={x: batch_x, y_: batch_y}),
                               global_step=step)

    writer.close()

print("Run the command line to start TensorBoard:\n" \
      "(TensorFlow) $ tensorboard --logdir=/tmp/tensorboard" \
      "\nThen open http://0.0.0.0:6006/ into your web browser")
```

4 MXNet

4.1 Programming interface

- Used to power Amazon Web Services (AWS)
- Support many different applications (e.g. computer vision, natural language processing, speech recognition, unsupervised machine learning, support embedded APIs, visualization)
- Mixed programming style: imperative and declarative
  - Light-weighted (around 50K lines of core code)
  - Data parallelism with multi-devices: 88% scalability with 256 GPUs
  - Support many front-ends, including JavaScript (run on web browsers)
  - Provide intermediate-level and high-level interface modules
  - Provide abundant IO functions
- Fully compatible with Torch: modules and operators
- Visualize neural network graphs
  - Call mx.viz.plot_network( )
- Not well documented, code not easy to read

4.2 A simple example

```
import os, gzip, struct
import numpy as np
import mxnet as mx

# Read data from the MNIST dataset
def read_data(label_url, image_url):
    with gzip.open(label_url) as flbl:
        magic, num = struct.unpack(">II", flbl.read(8))
        label = np.fromstring(flbl.read(), dtype=np.int8)
    with gzip.open(image_url, 'rb') as fimg:
        magic, num, row, col = struct.unpack(">IIII", fimg.read(16))
        image = np.fromstring(fimg.read(),dtype=np.uint8).reshape(len(label),row,col)
    return (label, image)

train_lbl_file = 'train-labels-idx1-ubyte.gz'
train_img_file = 'train-images-idx3-ubyte.gz'
train_lbl, train_img = read_data(train_lbl_file, train_img_file)
```

```
test_lbl_file = 't10k-labels-idx1-ubyte.gz'
test_img_file = 't10k-images-idx3-ubyte.gz'
test_lbl,  test_img  = read_data(test_lbl_file, test_img_file)

# Create data iterators for MXNet
def to4d(img):
    return img.reshape(img.shape[0], 1, 28, 28).astype(np.float32)/255

batch_size = 100
train_iter = mx.io.NDArrayIter(to4d(train_img), train_lbl, batch_size, shuffle=True)
test_iter  = mx.io.NDArrayIter(to4d(test_img), test_lbl, batch_size)

# Define the network
data = mx.sym.Variable('data') # Create a place holder variable for the input data
data = mx.sym.Flatten(data=data)# Flatten the data from 4-D shape into 2-D
fc1  = mx.sym.FullyConnected(data=data, name='fc1', num_hidden=64)
act1 = mx.sym.Activation(data=fc1, name='relu1', act_type="relu")
fc2  = mx.sym.FullyConnected(data=act1, name='fc2', num_hidden = 32)
act2 = mx.sym.Activation(data=fc2, name='relu2', act_type="relu")
fc3  = mx.sym.FullyConnected(data=act2, name='fc3', num_hidden=10)
out  = mx.sym.SoftmaxOutput(data=fc3, name='softmax')
mod  = mx.mod.Module(out)

# Plot the network graph
mx.viz.plot_network(symbol=out, shape={'data': (batch_size, 1, 28, 28)}).view()

# Prepare output log infomation
import logging
logging.getLogger().setLevel(logging.INFO)

# Train the network
mod.fit(train_data=train_iter, eval_data=test_iter, num_epoch=25)
```