A Tour of TensorFlow

Proseminar Data Mining

Peter Goldsborough
Fakultät für Informatik
Technische Universität München
Email: peter.goldsborough@in.tum.de

***Abstract*— Deep learning is a branch of artificial intelligence employing deep neural network architectures that has significantly advanced the state-of-the-art in computer vision, speech recognition, natural language processing and other domains. In November 2015, Google released *TensorFlow*, an open source deep learning software library for defining, training and deploying machine learning models. In this paper, we review TensorFlow and put it in context of modern deep learning concepts and software. We discuss its basic computational paradigms and distributed execution model, its programming interface as well as accompanying visualization toolkits. We then compare TensorFlow to alternative libraries such as Theano, Torch or Caffe on a quantitative basis and finally comment on observed use-cases of TensorFlow in academia and industry.**

***Index Terms*— Artificial Intelligence, Machine Learning, Neural Networks, Distributed Computing, Open source software, Software packages** TensorFlow

I. Introduction

Modern artificial intelligence systems and machine learning algorithms have revolutionized approaches to scientific and technological challenges in a variety of fields. We can observe remarkable improvements in the quality of state-of-the-art computer vision, natural language processing, speech recognition and other techniques. Moreover, the benefits of recent breakthroughs have trickled down to the individual, improving everyday life in numerous ways. Personalized digital assistants, recommendations on e-commerce platforms, financial fraud detection, customized web search results and social network feeds as well as novel discoveries in genomics have all been improved, if not enabled, by current machine learning methods.

A particular branch of machine learning, *deep learning*, has proven especially effective in recent years. Deep learning is a family of representation learning algorithms employing complex neural network architectures with a high number of hidden layers, each composed of simple but non-linear transformations to the input data. Given enough such transformation modules, very complex functions may be modeled to solve classification, regression, transcription and numerous other learning tasks [1].

For real world usage, deep learning algorithms must eventually be transcribed into a computer program. There exist a number of machine learning software libraries and frameworks for this purpose. Among these are Theano [2], Torch [3], scikit-learn [4] and many more. In November 2015, this list was extended by *TensorFlow*, a novel machine learning software library released by Google [5]. As per the initial publication, TensorFlow aims to be "an interface for expressing machine learning algorithms" in "large-scale [...] on heterogeneous distributed systems" [5]. direct quote is good

The remainder of this paper aims to give a thorough review of TensorFlow and is further structured as follows. Section II discusses in depth the computational paradigms underlying TensorFlow. In Section III we then explain the current programming interface. Tools allowing to visualize models built with TensorFlow are studied in Section IV. Subsequently, Section V compares TensorFlow to alternative deep learning libraries on a quantitative basis. Before concluding our review in Section VII, Section VI investigates current real world uses of TensorFlow in literature and industry.

II. The TensorFlow Programming Model

In this section we provide an in-depth discussion of the abstract computational principles underlying the TensorFlow software library. We begin with an examination of the basic structural and architectural decisions made by the TensorFlow development team. Thereafter, we study TensorFlow's local as well as distributed execution model. Lastly, we investigate how TensorFlow implements back-propagation, a crucial step for many machine learning algorithms.

A. Computational Graph Architecture

In TensorFlow, machine learning algorithms are represented as *computational graphs*. A computational or *dataflow* graph is a form of directed graph where *vertices* or *nodes* describe operations, while *edges* represent data flowing between these operations. If an output variable z is the result of applying a binary operation to two inputs x and y, then we draw directed edges from x and y to an output node representing z and annotate the vertex with a label describing the performed computation. Examples for computational graphs are given in Figure 1. The following paragraphs discuss the principle elements of such a dataflow graph, namely *operations*, *tensors*, *variables* and *sessions*. more citations

1) Operations: The major benefit of representing an algorithm in form of a graph is not only the intuitive (visual) expression of dependencies between units of a computational model, but also the fact that the definition of a *node* within the graph can be kept very general. In TensorFlow, nodes represent *operations*, which in turn express the combination or

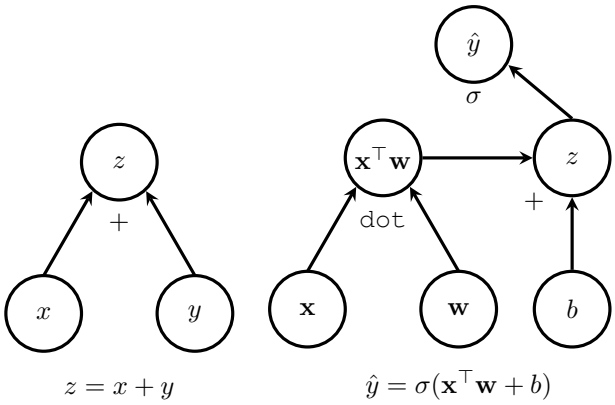


Fig. 1: Examples of computational graphs. The left graph performs the operation $+$ on inputs x and y to produce z. The right graph computes a logistic regression variable $\hat{y}$ for some example vector $\mathbf{x}$, weight vector $\mathbf{w}$ and scalar bias b. As shown in the graph, $\hat{y}$ is the result of the *sigmoid* function σ.

more citations

transformation of data flowing through the graph [5]. An operation can have *zero or more* inputs and produce *zero or more* outputs. As such, an operation may represent a mathematical equation, a variable or constant, a control flow directive, a file I/O operation or even a network communication port. Table I gives an overview of different kinds of operations that may be declared in a TensorFlow graph.

Any operation must be backed by an associated implementation. In [5] such an implementation is referred to as the operation's *kernel*. A particular kernel is always specifically built for execution on a certain kind of device, such as a CPU, GPU or other hardware units

2) Tensors: In TensorFlow, edges represent data flowing from one operation to another and are referred to as *tensors*. In the mathematical sense, a tensor is a multi-dimensional collection of homogeneous values with a fixed, static type. In terms of the computational graph, a tensor can be seen as a *symbolic handle* to one of the outputs of an operation. A tensor itself does not hold or store values in memory, but provides only an interface for retrieving the value referenced by the tensor.

more citations

3) Variables: Between two invocations of a computational graph, the majority of tensors in the graph are destroyed and do not persist. However, it is often necessary to maintain state across evaluations of a graph, such as for the weights and parameters of a neural network. For this purpose, there exist *variables* in TensorFlow, which can be described as persistent, mutable handles to in-memory buffers storing tensors. To manipulate and update variables, TensorFlow provides the `assign` family of graph operations.

4) Sessions: In TensorFlow, the execution of operations and evaluation of tensors may only be performed in a special environment referred to as a *session*. One of the responsibilities of a session is to encapsulate the allocation and management of resources such as variable buffers. Moreover, the `Session` interface of the TensorFlow library provides a

Category	Examples
Element-wise operations	`Add`, `Mul`, `Exp`
Matrix operations	`MatMul`, `MatrixInverse`
Value-producing operations	`Constant`, `Variable`
Neural network units	`SoftMax`, `ReLU`, `Conv2D`

TABLE I: Examples for TensorFlow operations [5].

`run` routine, which is the primary entry point for executing parts or the entirety of a computational graph. This method takes as input the nodes in the graph whose tensors should be computed and returned. Moreover, an optional mapping from arbitrary nodes in the graph to respective replacement values — referred to as *feed nodes* — may be supplied to `run` as well [5]. Upon invocation of `run`, TensorFlow will start at the requested output nodes and work backwards, examining the graph dependencies and computing the full transitive closure of all nodes that must be executed. These nodes are then assigned to one or many physical execution units (CPUs, GPUs etc.) on one or many machines.

more citations

B. Execution Model

's looks different to Session (styling error)

To execute computational graphs composed of the various elements just discussed, TensorFlow divides the tasks of its implementation among four distinct groups: the *client*, the *master*, a set of *workers* and lastly a number of *devices*. When the client requests evaluation of a TensorFlow graph via a `Session`'s `run` routine, this query is sent to the master process, which in turn delegates the task to one or more worker processes and coordinates their execution. Each worker is subsequently responsible for overseeing one or more devices, which are the physical processing units for which the kernels of an operation are implemented.

more citations

Within this model, there are two degrees of scalability. The first degree pertains to scaling the number of machines on which a graph is executed. The second degree refers to the fact that on each machine, there may then be more than one device, such as, for example, two GPUs and/or three CPUs. For this reason, there exist two "versions" of TensorFlow, one for local execution on a single machine (but possibly many devices), and one supporting a *distributed* implementation across many machines and many devices.

1) Devices: Devices are the smallest, most basic entities in the TensorFlow execution model. All nodes in the graph, that is, the kernel of each operation, must eventually be mapped to an available device to be executed. In practice, a device will most often be either a CPU or a GPU. However, TensorFlow supports registration of further kinds of physical execution units by the user. For example, in May 2016, Google announced its *Tensor Processing Unit* (TPU), a custom built ASIC optimized specifically for fast tensor computations [6]. It is thus understandably easy to integrate new device classes as novel hardware emerges.

2) Placement Algorithm: To determine what nodes to assign to which device, TensorFlow makes use of a *placement algorithm* [5]. The placement algorithm simulates the execution of the computational graph and traverses its nodes from

What is ASIC? It is not defined.

input tensors to output tensors. To decide on which of the available devices $\mathbb{D} = \{d_1, \ldots, d_n\}$ to place a given node ν encountered during this traversal, the algorithm consults a *cost model* $C_\nu(d)$. This cost model takes into account four pieces of information to determine the optimal device $\hat{d} = \arg\min_{d \in \mathbb{D}} C_\nu(d)$ on which to place the node during execution:

1) Whether or not there exists an implementation (kernel) for a node on the given device at all. For example, if there is no GPU kernel for a particular operation, any GPU device would automatically incur an infinite cost.
2) Estimates of the size (in bytes) for a node's input and output tensors.
3) The expected execution time for the kernel on the device.
4) A heuristic for the cost of cross-device (and possibly cross-machine) transmission of the input tensors to the operation, in the case that the input tensors were placed on nodes different from the one currently under consideration.

3) Cross-Device Execution: If the hardware configuration of the user's system provides more than one device, the placement algorithm will often distribute a graph's nodes among these devices. As a consequence, there may be cross-device dependencies between nodes that must be handled via a number of additional steps. Let us consider for this two devices A and B with particular focus on a node ν on device A. If ν's output tensor forms the input to some other operations α, β on device B, there conceptually exist cross-device edges $\nu \to \alpha$ and $\nu \to \beta$ from device A to device B. This is visualized in Figure 2a.

In practice, there must be some means of transmitting ν's output tensor from A, say a GPU device, to B — maybe a CPU device. For this reason, TensorFlow replaces the two edges $\nu \to \alpha$ and $\nu \to \beta$ by two new nodes. On device A, a `send` node is placed and connected to ν. In tandem, on device B, a `recv` node is instantiated and attached to α and β. The `send` and `recv` nodes are then connected by an additional edge. This is shown in Figure 2b. During execution of the graph, cross-device communication of data occurs exclusively via these special nodes.

C. Back-Propagation in TensorFlow

In a large number of deep learning and other machine learning algorithms, it is necessary to compute the gradients of particular nodes of the computational graph with respect to one or many other nodes. This is most often done via the *back-propagation* algorithm, which implements the chain rule. In [7], two approaches for back-propagating gradients through a computational graph are described. The first, which the authors refer to as *symbol-to-number differentiation*, receives a set of input values and then computes the *numerical values* of the gradients at those input values. It does so by explicitly traversing the graph first in the forward order (forward-propagation) to compute the cost, then in reverse order (back-propagation) to compute the gradients. Another approach, more relevant to TensorFlow, is what [7] calls *symbol-to-symbol derivatives*

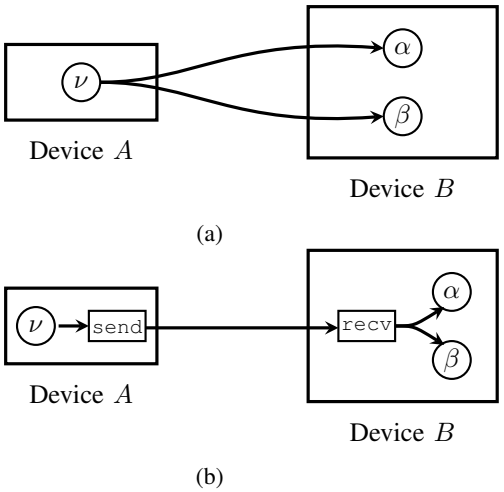


Fig. 2: A conceptual and practical visualization of cross-device communication. Figure 2a shows the conceptual connections between nodes on different devices. Figure 2b gives a more practical overview of how data is actually transmitted across devices using `send` and `recv` nodes.

and [5] terms *automatic gradient computation*. In this case, gradients are not computed by an explicit implementation of the back-propagation algorithm. Rather, special nodes are added to the computational graph that calculate the gradient of each operation and thus ultimately the chain rule. To perform back-propagation, these nodes must then simply be executed like any other nodes by the graph evaluation engine. As such, this approach does not produce the desired derivative as a numerical value, but only as a *symbolic handle* to calculate that value.

When TensorFlow needs to compute the gradient of a particular node ν with respect to some other tensor α, it traverses the graph in reverse order from ν to α. Each operation o encountered during this traversal represents a function depending on α and is one of the "links" in the chain $(\nu \circ \ldots \circ o \circ \ldots)(\alpha)$ producing the output tensor of the graph. Therefore, TensorFlow adds a *gradient node* for each such operation o that takes the gradient of the previous "link" and multiplies it with its own gradient. At the end of the traversal, there will be a node providing a symbolic handle to the overall target derivative $\frac{d\nu}{d\alpha}$, which *implicitly* implements the back-propagation algorithm. Figure 3 shows how a computational graph may look before and after gradient nodes are added.

III. The TensorFlow Programming Interface

Having conveyed the abstract concepts of TensorFlow's computational model in Section II, we will now concretize those ideas and speak to TensorFlow's programming interface. In the following paragraphs, we give a step-by-step walk-through of a practical, real-world example of TensorFlow's Python API. We will train a simple multi-layer perceptron (MLP) with one input and one output layer to classify

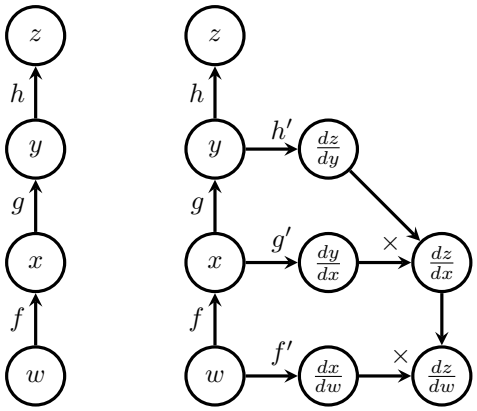
more citations

more citations

more citations

more citations in this whole section

It would be better if you could give a short definiton about what a gradient is, as it is important for the learning process.



(a) (b)

Fig. 3: A computational graph before (3a) and after (3b) gradient nodes are added. In this *symbol-to-symbol* approach, the gradient $\frac{dz}{dw}$ is just simply an operation like any other and therefore requires no special handling by the graph evaluation engine.

Use either "in" or the "element of" sign, but now both

handwritten digits in the MNIST[1] dataset. In this dataset, the examples are small images of 28×28 pixels depicting handwritten digits ~~in~~ $\in \{0, \ldots, 9\}$. We receive each such example as a flattened vector of $28 \times 28 = 784$ gray-scale pixel intensities. The label for each example is a *one-hot-encoded* vector $(d_1, \ldots, d_{10})^\top$ where the i-th component is set to one while all others are zero if the image shows the digit i.

We begin our walkthrough by importing the TensorFlow library:

```
import tensorflow as tf
```

Next, we create a new computational graph via the `tf.Graph` constructor. To add operations to this graph, we must register it as the *default graph*. The way the TensorFlow API is designed, library routines that create new operation nodes always attach these to the current default graph. We register our graph as the default by using it as a Python context manager in a `with-as` statement:

```
# Create a new graph
graph = tf.Graph()

# Register the graph as the default
with graph.as_default():
  # Add operations ...
```

We are now ready to populate our computational graph with operations. We begin by adding two *placeholder* nodes `examples` and `labels`. Placeholders are special variables that *must* be replaced with concrete tensors upon graph execution. That is, they must be supplied in the `feed_dict` argument to `Session.run()`, mapping tensors to replacement values. For each such placeholder, we specify a shape and data type. By specifying the keyword `None` for the first dimension of each placeholder shape, we can later feed a tensor of variable size in that dimension. For the column size of the `example` placeholder, we specify the number of features for each image, meaning the $28 \times 28 = 784$ pixels. The label placeholder should expect 10 columns, corresponding to the 10-dimensional one-hot-encoded vector for each label digit:

```
# Using 32-bit floating-point data type tf.float32
examples = tf.placeholder(tf.float32, [None, 784])
labels = tf.placeholder(tf.float32, [None, 10])
```

Given an example matrix $\mathbf{X} \in \mathbb{R}^{n \times 784}$ containing n images, the learning task then applies an affine transformation $\mathbf{X} \cdot \mathbf{W} + \mathbf{b}$, where $\mathbf{W}$ is a *weight* matrix $\in \mathbb{R}^{784 \times 10}$ and $\mathbf{b}$ a *bias* vector $\in \mathbb{R}^{10}$. This yields a new matrix $\mathbf{Y} \in \mathbb{R}^{n \times 10}$, containing the *scores* or *logits* of our model for each example and each possible digit. To transform the logits into a valid probability distribution, we apply the *softmax* function:

```
# Draw random weights for symmetry breaking
weights = tf.Variable(tf.random_uniform([784, 10]))
# Slightly positive initial bias
bias = tf.Variable(tf.constant(0.1, shape=[10]))
# tf.matmul performs the matrix multiplication XW
# Note how the + operator is overloaded for tensors
logits = tf.matmul(examples, weights) + bias
# Applies the operation element-wise on tensors
estimates = tf.nn.softmax(logits)
```

We then compute our objective function, producing the error or *loss* of the model given its current trainable parameters $\mathbf{W}$ and $\mathbf{b}$. We do this by calculating the *cross entropy* $H(\mathbf{L}, \mathbf{Y})_i = -\sum_j \mathbf{L}_{i,j} \cdot \log(\mathbf{Y}_{i,j})$ between the probability distributions of our estimates $\mathbf{Y}$ and the one-hot-encoded labels $\mathbf{L}$. More precisely, we consider the mean cross entropy over all examples as the loss:

```
# Computes the cross-entropy and sums the rows
cross_entropy = -tf.reduce_sum(
   labels * tf.log(estimates), [1])
loss = tf.reduce_mean(cross_entropy)
```

Now that we have an objective function, we can run (stochastic) gradient descent to update the weights of our model. For this, TensorFlow provides a `GradientDescentOptimizer` class. It is initialized with the learning rate of the algorithm and provides an operation `minimize`, to which we pass our `loss` tensor. This is the operation we will run repeatedly in a `Session` environment to train our model:

```
# We choose a learning rate of 0.5
gdo = tf.train.GradientDescentOptimizer(0.5)
optimizer = gdo.minimize(loss)
```

Specifiy what gets trained (which variables)

Finally, we can actually train our algorithm. For this, we enter a session environment using a `tf.Session` as a context manager. We pass our graph object to its constructor, so that it knows which graph to manage. To then execute nodes, we use `Session.run()` and pass a list of tensors we wish to compute. Before evaluating any other node, we must first initialize the variables in our graph. We do this by running the `tf.initialize_all_variables()` utility operation. Then, we perform a certain number of iterations of

styling error

[1] http://yann.lecun.com/exdb/mnist/

stochastic gradient descent, fetching an example and label mini-batch from the MNIST dataset each time and feeding it to the `run` routine. For this, we assume a utility object `mnist` which can be used as shown below. At the end, our loss will (hopefully) be small:

```
with tf.Session(graph=graph) as session:
    # Execute the operation directly
    tf.initialize_all_variables().run()
    for step in range(1000):
        # Fetch next 100 examples and labels
        x, y = mnist.train.next_batch(100)
        # Ignore the result of the optimizer (None)
        _, loss_value = session.run(
            [optimizer, loss],
            feed_dict={examples: x, labels: y})
        print('Loss at step {0}: {1}'
              .format(step, loss_value))
```

The full code listing for this example, along with some additional implementation to compute an accuracy metric at each time step, is given in Appendix I.

IV. Visualization of TensorFlow Graphs

Deep learning models often employ neural networks with a highly complex and intricate structure. For example, [5] reports of a deep convolutional network based on Google's *Inception* model, with more than 36,000 individual units. *TensorBoard*, a web interface for graph visualization built directly into TensorFlow, allows the user to maintain a clear overview of such complex structures and facilitates model debugging.

The core feature of TensorBoard is its lucid visualization of computational graphs, exemplified in Figure 4a. Graphs with intricate topologies and many layers can be displayed in a clear and organized manner, allowing the user to understand exactly how data flows through it. Especially useful is TensorBoard's notion of *name scopes*, whereby nodes or entire subgraphs may be grouped into one visual block, such as a single neural network layer. Such name scopes can then be expanded interactively to show the grouped units in more detail. Figure 4b shows the expansion of one the name scopes of the graph in Figure 4a.

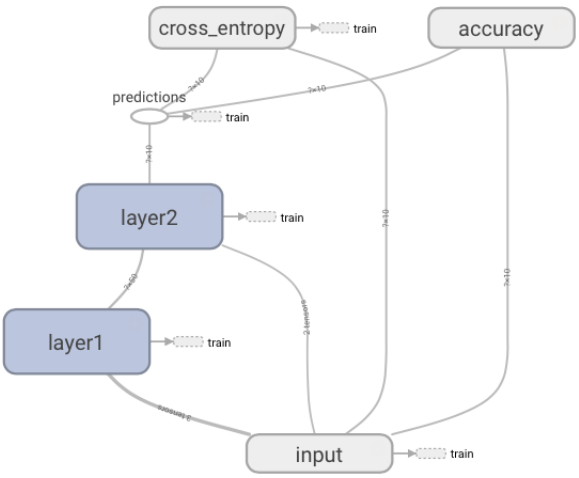
Furthermore, TensorBoard allows the user to track the development of individual tensor values over time. For this, you can attach two kinds of *summary operations* to nodes of the computational graph: *scalar summaries* and *histogram summaries*. Scalar summaries show the progression of a scalar tensor value, which can be sampled at certain iteration counts. In this way, you could, for example, observe the accuracy or loss of your model with time. Histogram summary nodes allow the user to track value *distributions*, such as those of neural network weights or the final softmax estimates. Figures 4c and 4d give examples of scalar and histogram summaries, respectively.

more citations

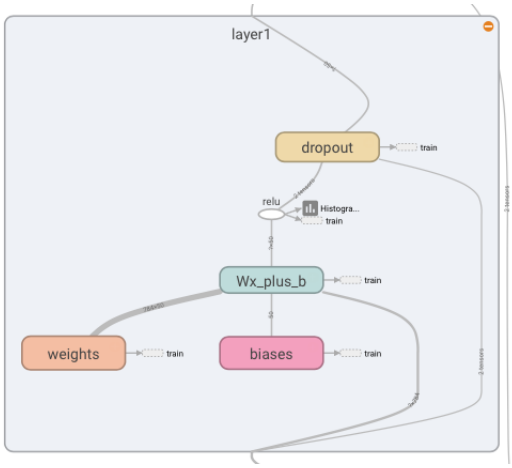
V. Comparison With Other Deep Learning Frameworks

Next to TensorFlow, there exist a number of other open source deep learning software libraries, the most popular being

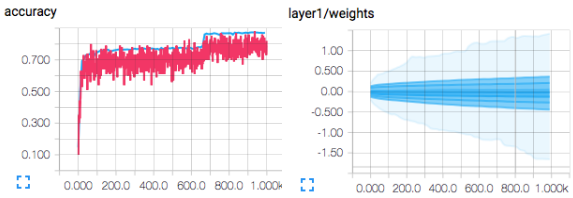
ones



(a)



(b)



(c) (d)

Fig. 4: A demonstration of TensorBoard's graph visualization features. Figure 4a shows the complete graph, while Figure 4b displays the expansion of the first layer. Figures 4c and 4d give examples for scalar and histogram summaries, respectively.

Theano, Torch and Caffe. In this section, we review two sources of quantitative comparisons between TensorFlow and these alternatives, providing a summary of the most important results of each work.

The first source in our collection is the *convnet-benchmarks* repository on GitHub by Soumith Chintala [9], a research

Library	Forward (ms)	Backward (ms)
TensorFlow	26	55
Torch	**25**	**46**
Caffe	121	203

TABLE II: The result of Soumith Chintala's benchmarks for TensorFlow, Torch and Caffe (not Theano) on an AlexNet ConvNet model [9], [10].

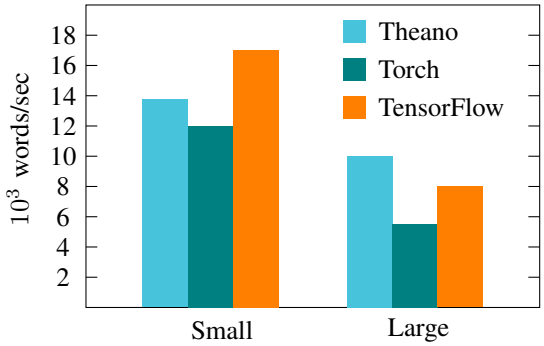


Fig. 5: The results of [2], comparing TensorFlow, Theano and Torch on an LSTM model for the Penn Treebank dataset [11]. On the left the authors tested a small model with a single hidden layer and 200 units; on the right they use two layers with 650 units each.

engineer at Facebook. The commit we reference[2] is dated April 25, 2016. Chintala provides an extensive set of benchmarks for a variety of convolutional network models and many libraries. Inter alia, Chintala gives the forward and backward-propagation time of TensorFlow, Torch and Caffe for the AlexNet CNN model [10]. Theano is not included in these benchmarks. The outcomes show that TensorFlow performs second-best in both measures behind Torch, with Caffe lagging relatively far behind. We reproduce the relevant results in Table II.

Further benchmarks were published by the Theano development team in [2] on May 9, 2016. We focus on their results for an LSTM network operating on the Penn Treebank dataset [11]. Their comparisons measure words processed per second for a small model consisting of a single 200-unit hidden layer and a large model with two 650-unit hidden layers. Tested libraries include Theano, Torch and TensorFlow, but not Caffe. In their benchmarks, TensorFlow performs best among all three for the small model, followed by Theano and then Torch. For the large model, TensorFlow is placed second behind Theano, while Torch remains in last place. Table 5 shows these outcomes. Figure

VI. Use Cases of TensorFlow Today

In this section, we investigate where TensorFlow is in use today. Given that TensorFlow was released only little over 6 months ago as of this writing, its adoption in academia and industry is not yet widespread. The one exception is, of course, Google, which has already employed TensorFlow for a variety of learning tasks [8], [12]–[14].

In [14], Ramsundar et al. discuss massively "multitask networks for drug discovery" in a joint collaboration work between Stanford University and Google, published in early 2016. In this paper, the authors employ deep neural networks developed with TensorFlow to perform virtual screening of potential drug candidates. This is intended to aid pharmaceutical companies and the scientific community in finding novel medication and treatments for human diseases.

August and Ni apply TensorFlow to create recurrent neural networks for optimizing dynamic decoupling, a technique for suppressing errors in quantum memory [15]. With this, the authors aim to preserve the coherence of quantum states, which is one of the primary requirements for building universal quantum computers.

Lastly, we make note of the decision of Google DeepMind, an AI division within Google, to move from Torch7 to TensorFlow [16]. A related source, [6], states that DeepMind made use of TensorFlow for its *AlphaGo*[3] model, alongside Google's newly developed Tensor Processing Unit (TPU), which was built to integrate especially well with TensorFlow. In a correspondence of the authors of this paper with a member of the Google DeepMind team, the following four reasons were revealed to us as to why TensorFlow is advantageous to DeepMind:

1) TensorFlow is included in the Google Cloud Platform[4], which enables easy replication of DeepMind's research.
2) TensorFlow's support for TPUs.
3) TensorFlow's main interface, Python, is one of the core languages at Google, implying a much greater internal tool set than for Lua.
4) The ability to run TensorFlow on many GPUs.

VII. Conclusion

We have discussed TensorFlow, a novel open source deep learning library based on computational graphs. Its ability to perform fast automatic gradient computation, its inherent support for distributed computation and specialized hardware as well as its powerful visualization tools make it a very welcome addition to the field of machine learning. We showed how TensorFlow's programming interface allowed for easy implementation of a simple machine learning task, forming the basis for more complex models. In the context of other deep learning toolkits such as Theano or Torch, TensorFlow adds new features and improves on others. On a qualitative basis, we gave evidence for the competitive performance of TensorFlow compared to alternative libraries.

TensorFlow has gained great popularity and strong support in the open-source community with many third-party contributions, making Google's move a sensible decision already. We believe, however, that it will not only benefit its parent company, but the greater scientific community as a whole; opening new doors to faster, larger-scale artificial intelligence.

[2]Commit hash: 84b5bb1785106e89691bc0625674b566a6c02147

[3]https://deepmind.com/alpha-go

[4]https://cloud.google.com/compute/

add whitespace

APPENDIX I

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" A one-hidden-layer-MLP MNIST-classifier. """

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# Import the training data (MNIST)
from tensorflow.examples.tutorials.mnist import
    input_data

import tensorflow as tf

# Possibly download and extract the MNIST data set.
# Retrieve the labels as one-hot-encoded vectors.
mnist = input_data.read_data_sets("/tmp/mnist",
    one_hot=True)

# Create a new graph
graph = tf.Graph()

# Set our graph as the one to add nodes to
with graph.as_default():

   # Placeholder for input examples (None =
       variable dimension)
   examples = tf.placeholder(shape=[None, 784],
       dtype=tf.float32)
   # Placeholder for labels
   labels = tf.placeholder(shape=[None, 10],
       dtype=tf.float32)

   # Draw the weights from a random uniform
       distribution for symmetry breaking
   weights =
       tf.Variable(tf.random_uniform(shape=[784,
       10]))
   # Slightly positive biases to avoid dead neurons
   bias = tf.Variable(tf.constant(0.1, shape=[10]))

   # Apply an affine transformation to the input
       features
   logits = tf.matmul(examples, weights) + bias
   estimates = tf.nn.softmax(logits)

   # Compute the cross-entropy
   cross_entropy = -tf.reduce_sum(labels *
       tf.log(estimates),
                          reduction_indices=[1])
   # And finally the loss
   loss = tf.reduce_mean(cross_entropy)

   # Create a gradient-descent optimizer that
       minimizes the loss.
   # We choose a learning rate of 0.5
   optimizer =
       tf.train.GradientDescentOptimizer(0.5).minimize(loss)

   # Find the indices where the predictions were
       correct
   correct_predictions = tf.equal(
      tf.argmax(estimates, dimension=1),
      tf.argmax(labels, dimension=1))
   accuracy =
       tf.reduce_mean(tf.cast(correct_predictions,
       tf.float32))

with tf.Session(graph=graph) as session:
   tf.initialize_all_variables().run()
   for step in range(1000):
      example_batch, label_batch =
          mnist.train.next_batch(100)
      feed_dict = {examples: example_batch, labels:
          label_batch}
      if step % 100 == 0:
         _, loss_value, accuracy_value =
             session.run(
            [optimizer, loss, accuracy],
            feed_dict=feed_dict
         )
         print("Loss at time {0}: {1}".format(step,
             loss_value))
         print("Accuracy at time {0}:
             {1}".format(step, accuracy_value))
      else:
         optimizer.run(feed_dict)
```

structurally error

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

[2] The Theano Development Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. Vlad Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. Ramana Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, "Theano: A Python framework for fast computation of mathematical expressions," *ArXiv e-prints*, May 2016.

[3] R. Collobert, S. Bengio, and J. Marithoz, "Torch: A modular machine learning software library," 2002.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078195

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[6] N. Jouppi, "Google supercharges machine learning tasks with tpu custom chip," Google Cloud Platform Blog, May 2016 (accessed May 22, 2016), https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[7] I. G. Y. Bengio and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *ArXiv e-prints*, Sep. 2014.

[9] S. Chintala, "convnet-benchmarks," GitHub, April 2016 (accessed May 24, 2016), https://github.com/soumith/convnet-benchmarks.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, p. 2012.

[11] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Comput. Linguist.*, vol. 19, no. 2, pp. 313–330, Jun. 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=972470.972475

[12] O. Good, "How google translate squeezes deep learning onto a phone," Google Research Blog, Jul. 2015 (accessed: May 25, 2016), http://googleresearch.blogspot.de/2015/07/how-google-translate-squeezes-deep.html.

[13] G. Corrado, "Computer, respond to this email." Google Research Blog, Nov. 2015 (accessed: May 25, 2016), http://googleresearch.blogspot.de/2015/11/computer-respond-to-this-email.html.

[14] B. Ramsundar, S. Kearnes, P. Riley, D. Webster, D. Konerding, and V. Pande, "Massively Multitask Networks for Drug Discovery," *ArXiv e-prints*, Feb. 2015.

[15] M. August and X. Ni, "Using recurrent neural networks to optimize dynamical decoupling for quantum memory," in *arXiv.org*, vol. quant-ph, no. arXiv:1604.00279. Technical University of Munich, Max Planck Institute for Quantum Optics, Apr. 2016. [Online]. Available: http://arxiv.org/pdf/1604.00279v1.pdf

[16] K. Kavukcuoglu, "Deepmind moves to tensorflow," Google Research Blog, Apr. 2016 (accessed May 24, 2016), http://googleresearch.blogspot.de/2016/04/deepmind-moves-to-tensorflow.html.