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 qualitative as well as 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**

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].

It is noteworthy that the rise in popularity of deep learning can be traced back to only the last few years, enabled primarily by the greater availability of large data sets, containing more training examples; the efficient use of graphical processing units (GPUs) and massively parallel commodity hardware to train deep learning models on these equally massive data sets as well as the discovery of new methods such as the *rectified linear unit* (ReLU) activation function or *dropout* as a regularization technique [1]–[4].

While deep learning algorithms and individual architectural components such as representation transformations, activation functions or regularization methods may initially be expressed in mathematical notation, they must eventually be transcribed into a computer program for real world usage. For this purpose, there exist a number of open source as well as commercial machine learning software libraries and frameworks. Among these are Theano [5], Torch [6], scikit-learn [7] and many more, which we review in further detail in Section II of this paper. In November 2015, this list was extended by *TensorFlow*, a novel machine learning software library released by Google [8]. As per the initial publication, TensorFlow aims to be "an interface for expressing machine learning algorithms" in "large-scale [. . .] on heterogeneous distributed systems" [8].

The remainder of this paper aims to give a thorough review of TensorFlow and put it in context of the current state of machine learning. In detail, the paper is further structured as follows. Section **??** will provide a brief overview and history of machine learning software libraries, listing but not comparing projects similar to TensorFlow. Subsequently, Section II discusses in depth the computational paradigms underlying TensorFlow. In Section III we explain the current programming interface in the various supported languages. To inspect and debug models expressed in TensorFlow, there exist powerful visualization tools, which we examine in Section IV. Section V then gives a comparison of TensorFlow and alternative deep learning libraries on a qualitative as well as quantitative basis. Before concluding our review in Section VII, Section VI studies current real world use cases 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 a thorough examination of the basic structural and architectural decisions made by the TensorFlow development team and explain how machine learning algorithms may be expressed in its dataflow graph language. Subsequently, we study TensorFlow's execution model and provide insight into the way TensorFlow graphs are assigned

$$z = x + y \qquad\qquad \hat{y} = \sigma(\mathbf{x}^\top \mathbf{w} + b)$$
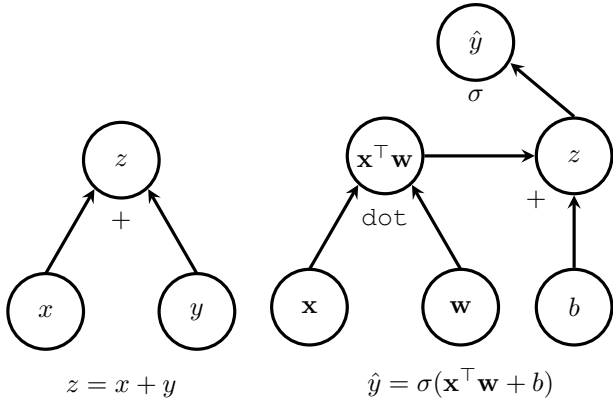
Fig. 1: Examples of computational graphs. The left graph displays a very simple computation, consisting of just an addition of the two input variables x and y. In this case, z is the result of the operation $+$, as the annotation suggests. The right graph gives a more complex example of computing a logistic regression variable $\hat{y}$ in for some example vector $\mathbf{x}$, weight vector $\mathbf{w}$ as well as a scalar bias b. As shown in the graph, $\hat{y}$ is the result of the *sigmoid* or *logistic* function σ.

to available hardware units in a local as well as distributed environment. Then, we investigate the various optimizations incorporated into TensorFlow, targeted at improving both software and hardware efficiency. Lastly, we list extensions to the basic programming model that aid the user in both computational as well as logistical aspects of training a machine learning model with TensorFlow.

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* or *arcs* 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* in further detail.

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 transformation of data flowing through the graph [8]. 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. It may seem unintuitive that an operation, which the reader

may associate with a *function* in the mathematical sense, can represent a constant or variable. However, a constant may be thought of as an operation that takes no inputs and always produces the same output corresponding to the constant it represents. Analogously, a variable is really just an operation taking no input and producing the current state or value of that variable. Table **??** 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 [8] 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 unit.

2) Tensors: In TensorFlow, edges represent data flowing from one operation to another and are referred to as *tensors*. A tensor is a multi-dimensional collection of homogeneous values with a fixed, static type. The number of dimensions of a tensor is termed its *rank*. A tensor's *shape* is the tuple describing its size, i.e. the number of components, in each dimension. In the mathematical sense, a tensor is the generalization of two-dimensional matrices, one-dimensional vectors and also scalars, which are simply tensors of rank zero.

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. When creating an operation in the TensorFlow programming environment, such as for the expression $x + y$, a tensor object is returned. This tensor may then be supplied as input to other computations, thereby connecting the source and destination operations with an edge. By these means, data flows through a TensorFlow graph.

Next to regular tensors, TensorFlow also provides a `SparseTensor` data structure, allowing for a more space-efficient dictionary-like representation of *sparse tensors* with only few non-zeros entries.

3) Variables: In a typical situation, such as when performing stochastic gradient descent (SGD), the graph of a machine learning model is executed from start to end multiple times for a single experiment. Between two such invocations, the majority of tensors in the graph are destroyed and do not persist. However, it is often necessary to maintain state across evaluations of the graph, such as for the weights and parameters of a neural network. For this purpose, there exist *variables* in TensorFlow, which are simply special operations that can be added to the computational graph.

In detail, variables can be described as persistent, mutable

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

TABLE I: Examples for TensorFlow operations [8].

handles to in-memory buffers storing tensors. As such, variables are characterized by a certain shape and a fixed type. To manipulate and update variables manually (as opposed to implicitly by certain library routines), 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 *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 `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 [8].

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 may then be assigned to one or many physical execution units (CPUs, GPUs etc.) on one or many machines. The rules by which this assignment takes place are determined by TensorFlow's *placement algorithm*, discussed in detail in Subsection **??**. Furthermore, as there exists the possibility to specify explicit orderings of node evaluations, called *control dependencies*, the execution algorithm will ensure that these dependencies are maintained.

B. Execution Model

To execute computational graphs composed of the various elements just discussed, TensorFlow divides the tasks for 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.

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, five independent 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. Figure 3 visualizes a possible distributed setup. While the initial release of TensorFlow supported only single-machine execution, the distributed version was open-sourced on April 13, 2016 [16].
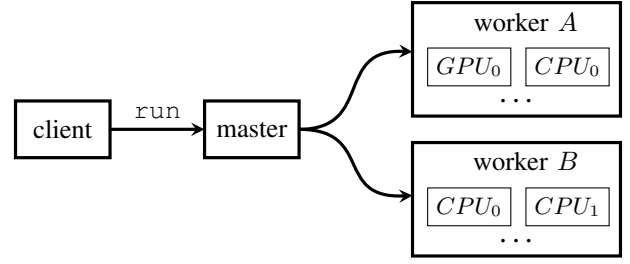

Fig. 2: A visualization of the different execution agents in a multi-machine, multi-device hardware configuration.

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 (application-specific-integrated-circuit) optimized specifically for fast tensor computations [17]. It is thus understandably easy to integrate new device classes as novel hardware emerges.

To oversee the evaluation of nodes on a device, a worker process is spawned by the master. As a worker process may manage one or many devices on a single machine, a device is identified not only by a name, but also an index for its worker group. For example, the first CPU in a particular group may be identified by the string "/cpu:0".

2) Placement Algorithm: To determine what nodes to assign to which device, TensorFlow makes use of a *placement algorithm*. The placement algorithm simulates the execution of the computational graph and traverses its nodes from 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 have been 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. This can be seen as partitioning the set of nodes into classes, one per device. 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 initially exist cross-device edges $\nu \to \alpha$ and $\nu \to \beta$ from device A to device B. This is visualized in Figure 4a.

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 initially replaces the two edges $\nu \to \alpha$ and $\nu \to \beta$ by three new nodes. On device A, a `send` node is placed and connected to ν. In tandem, on device B, two `recv` nodes are instantiated and attached to α and β, respectively. The `send` and `recv` nodes are then connected by two additional edges. This step is shown in Figure 4b. During execution of the graph, cross-device communication of data occurs exclusively via these special nodes. When the devices are located on separate machines, transmission between the worker processes on these machines may involve remote communication protocols such as TCP or RDMA.

Finally, an important optimization made by TensorFlow at this step is "canonicalization" of (`send`, `receive`) pairs. In the setup displayed in Figure 4b, the existence of each `recv` node on device B would imply allocation and management of a separate buffer to store ν's output tensor, so that it may then be fed to nodes α and β, respectively. However, an equivalent and more efficient transformation places only one `recv` node on device B, streams all output from ν to this single node, and then to the two dependent nodes α and β. This last and final evolution is given in Figure 4c.

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. For example, in a neural network, we may compute the cost c of the model for a given example x by passing that example through a series of non-linear transformations. If the neural network consists of two hidden layers represented by functions $f(x; w) = f_x(w)$ and $g(x; w) = g_x(w)$ with internal weights w, we can express the cost for that example as $c = (f_x \circ g_x)(w) = f_x(g_x(w))$. We would then typically calculate the gradient dc/dw of that cost with respect to the weights and use it to update w. Often, this is done by means of the *back-propagation* algorithm, which traverses the graph in reverse to compute the chain rule $[f_x(g_x(w))]' = f'_x(g_x(w)) \cdot g'_x(w)$.

In [18], 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
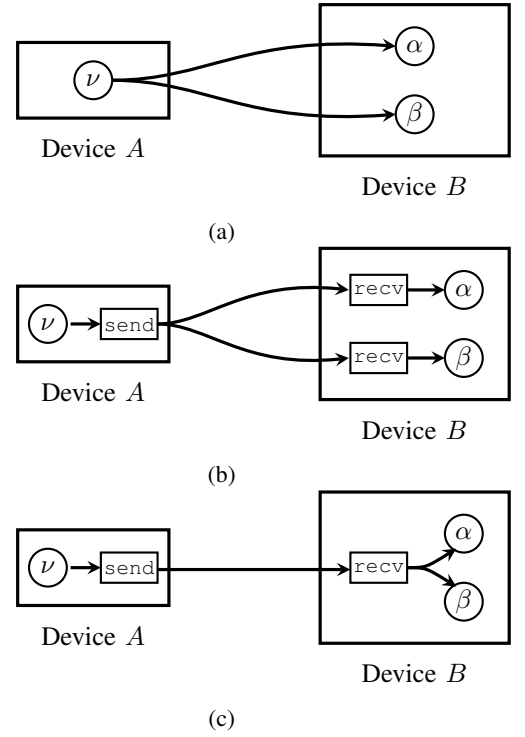

Fig. 3: The three stages of cross-device communication between graph nodes in TensorFlow. Figure 4a shows the initial, conceptual connections between nodes on different devices. Figure 4b gives a more practical overview of how data is actually transmitted across devices using `send` and `recv` nodes. Lastly, Figure 4c shows the final, canonicalized setup, where there is at most one `recv` node per destination device.

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 via the chain rule. Another approach, more relevant to TensorFlow, is what [18] calls *symbol-to-symbol derivatives* and [8] 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 derivatives as a numeric value, but only as a *symbolic handle* to compute those values.

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 (the outer function) and multiplies it with its own gradient. At the
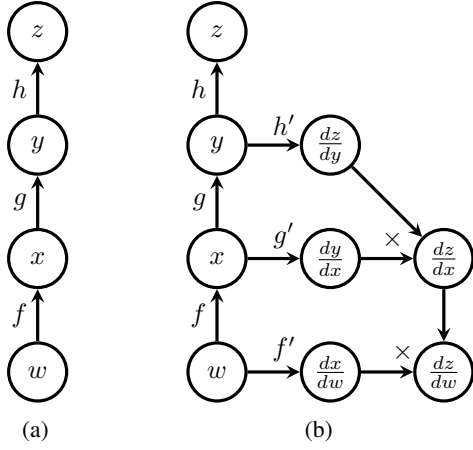
Fig. 4: A computational graph before (5a) and after (5b) gradient nodes are added. In this *symbol-to-symbol* approach, the gradient $\frac{dz}{dw}$ is just simply an operation like any other and therefore requires no special handling by the graph evaluation engine.

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. It should now be clear that back-propagation in this symbol-to-symbol approach is *just another operation*, requiring no exceptional handling. Figure 5 shows how a computational graph may look before and after gradient nodes are added.

In [8] it is noted that symbol-to-symbol derivatives may incur a considerable performance cost and especially result in increased memory overhead. To see why, it is important to understand that there exist two equivalent formulations of the chain rule. The first reuses previous computations and therefore requires them to be stored longer than strictly necessary for forward-propagation. For arbitrary functions f, g and h it is given in Equation 1:

$$\frac{\mathrm{d}f}{\mathrm{d}w} = f'(y) \cdot g'(x) \cdot h'(w) \text{ with } y = g(x), x = h(w) \quad (1)$$

The second possibility for computing the chain rule was already shown, where each function recomputes all of its arguments and invokes every function it depends on. It is given in Equation 2 for reference:

$$\frac{\mathrm{d}f}{\mathrm{d}w} = f'(g(h(w))) \cdot g'(h(w)) \cdot h'(w) \quad (2)$$

According to [8], TensorFlow currently employs the first approach. Given that the inner-most functions must be recomputed for almost every link of the chain if this approach is not employed, and taking into consideration that this chain may consist of many hundreds or thousands of operations, this choice seems sensible. However, on the flip side, keeping tensors in memory for long periods of time is also not optimal, especially on devices like GPUs where memory resources are scarce. For Equation 2, memory held by tensors could in theory be freed as soon as it has been processed by its graph dependencies. For this reason, in [8] the development team

of TensorFlow states that recomputing certain tensors rather than keeping them in memory may be a possible performance improvement for the future.

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. We begin with a brief discussion of the available language interfaces. Then, we provide a more hands-on look at TensorFlow's Python API by walking through a simple practical example.

A. Interfaces

There currently exist two programming interfaces, in C++ and Python, that permit interaction with the TensorFlow backend. The Python API boasts a very rich feature set for creation and execution of computational graphs. As of this writing, the C++ interface (which is really just the core backend implementation) provides a comparatively much more limited API, allowing only to execute graphs built with Python and serialized to Google's Protocol Buffer[1] format. While there is experimental support for also building computational graphs in C++, this functionality is currently not as extensive as in Python.

It is noteworthy that the Python API integrates very well with NumPy[2], a popular open source Python numeric and scientific programming library. As such, TensorFlow tensors may be interchanged with NumPy `ndarrays` in many places.

B. Walkthrough

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 hand-written digits in the MNIST[3] 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 784 gray-scale pixel intensities. The label for each example is the digit it is supposed to represent.

We begin our walkthrough by importing the TensorFlow library and reading the MNIST dataset into memory. For this we assume a utility module `mnist_data` with a method `read` which expects a path to extract and store the dataset. Moreover, we pass the parameter `one_hot=True` to specify that each label be given to us as a *one-hot-encoded* vector $(d_1, \ldots, d_{10})^\top$ where all but the i-th component are set to zero if an example represents the digit i:

```python
import tensorflow as tf

# Download and extract the MNIST data set.
# Retrieve the labels as one-hot-encoded vectors.
mnist = mnist_data.read("/tmp/mnist", one_hot=True)
```

[1]https://developers.google.com/protocol-buffers/
[2]http://www.numpy.org
[3]http://yann.lecun.com/exdb/mnist/

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 one to add nodes
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. An interesting feature of TensorFlow at this point is that we may specify the Python keyword `None` for the first dimension of each placeholder shape. This allows us to later on 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 a 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. These scores are more or less arbitrary values and not a probability distribution, i.e. they need neither be $\in [0, 1]$ nor sum to one. To transform the logits into a valid probability distribution, giving the likelihood $\Pr[x = i]$ that the x-th example represents the digit i, we make use of the softmax function, given in Equation 3. Our final estimates are thus calculated by $\mathrm{softmax}(\mathbf{X} \cdot \mathbf{W} + \mathbf{b})$, as shown below:

```
# 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)
```

$$\mathrm{softmax}(\mathbf{x})_i = \frac{\exp(\mathbf{x}_i)}{\sum_j \exp(\mathbf{x}_j)} \qquad (3)$$

Next, we 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)
```

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 have several options. The most general way is to call `Session.run()` and pass a list of tensors we wish to compute. Alternatively, we may call `eval()` on tensors and `run()` on operations directly. Before evaluating any other node, we must first ensure that the variables in our graph are initialized. Theoretically, we could `run` the `Variable.initializer` operation for each variable. However, one most often just uses the `tf.initialize_all_variables()` utility operation provided by TensorFlow, which in turn executes the `initializer` operation for each `Variable` in the graph. Then, we can perform a certain number of iterations of stochastic gradient descent, fetching an example and label mini-batch from the MNIST dataset each time and feeding it to the `run` function of our optimizer. 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, [19] reports of deep convolutional network based on the Google *Inception* model with more than 36,000 individual units, while [8] states that certain long short-term memory (LSTM) architectures can span over 15,000 nodes. To maintain a clear overview of such complex networks, facilitate model debugging and allow inspection of values on various levels of detail, powerful visualization tools are required. *TensorBoard*, a web interface for graph visualization and manipulation built directly into TensorFlow, is an example for such a tool. In this section, we first list a number of noteworthy features of TensorBoard and then discuss how it is used from TensorFlow's programming interface.

A. TensorBoard Features

The core feature of TensorBoard is the lucid visualization of computational graphs, exemplified in Figure 6a. Graphs with complex 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 6b shows the expansion of one the name scopes of Figure 6a.

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 6c and 6d give examples of scalar and histogram summaries, respectively. Lastly, TensorBoard also allows visualization of images. This can be useful to show the images sampled for each mini-batch of an image classification task, or to visualize the kernel filters of a convolutional neural network [8].

We note especially how interactive the TensorBoard web interface is. Once your computational graph is uploaded, you can pan and zoom the model as well as expand or contract individual name scopes. A demonstration of TensorBoard is available at *https://www.tensorflow.org/tensorboard/index.html*.

B. TensorBoard in Practice

To integrate TensorBoard into your TensorFlow code, three steps are required. Firstly, it is wise to group nodes into *name scopes*. Then, you may add scalar and histogram summaries to you operations. Finally, you must instantiate a
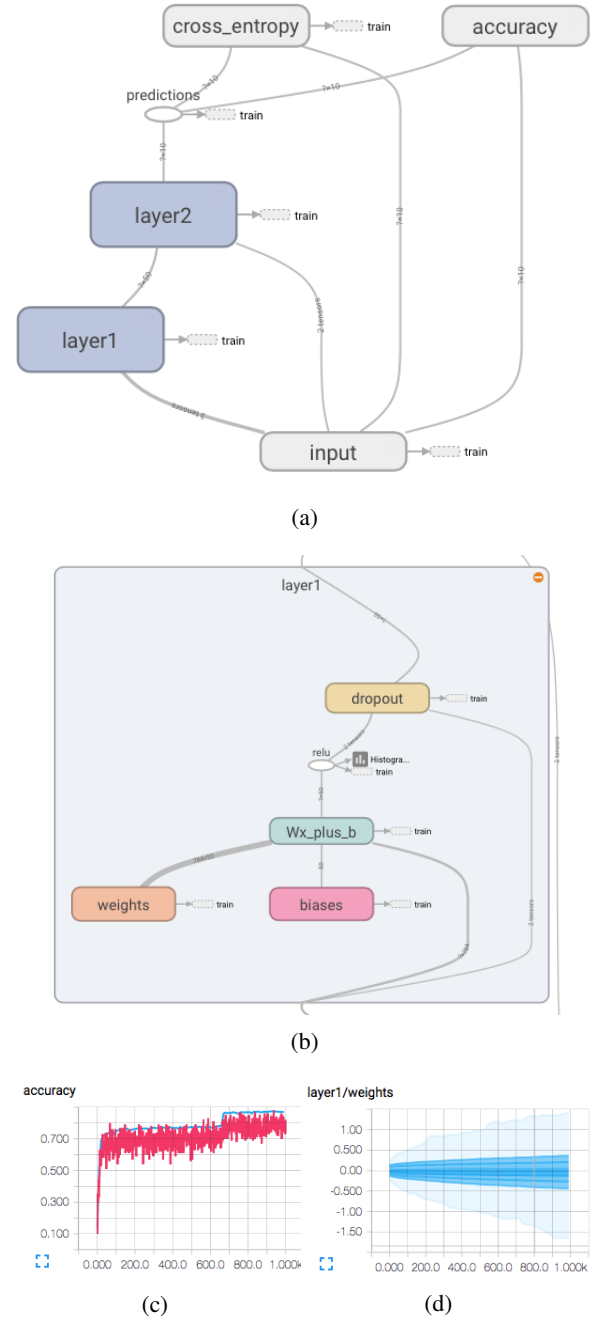

(a)

(b)

(c)

(d)

Fig. 5: Foo

SummaryWriter object and hand it the tensors produced by the summary nodes in a session context whenever you wish to store new summaries. Rather than fetching individual summaries, it is also possible to combine all summary nodes into one via the `tf.merge_all_summaries()` operation.

```
with tf.name_scope('Variables'):
    x = tf.constant(1.0)
    y = tf.constant(2.0)
    tf.scalar_summary('z', x + y)

merged = tf.merge_all_summaries()
writer = tf.train.SummaryWriter('/tmp/log', graph)
```

```
with tf.Session(graph=graph):
  for step in range(1000):
    writer.add_summary(
      merged.eval(), global_step=step)
```

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 Theano, Torch and Caffe. In this section, we review three sources of quantitative comparisons between TensorFlow and other deep learning libraries, providing a summary of the most important results of each work. Furthermore, we will briefly discuss the overall trend of these benchmarks.

The first work, [20], authored by the Bosch Research and Technology Center in late March 2016, compares the performance of TensorFlow, Torch, Theano and Caffe (among others) with respect to various neural network architectures. Their setup involves Ubuntu 14.04 running on an Intel Xeon E5-1650 v2 CPU @ 3.50 GHz and an NVIDIA GeForce GTX Titan X/PCIe/SSE2 GPU. One benchmark we find noteworthy tests the relative performance of each library on a slightly modified reproduction of the LeNet CNN model [21]. More specifically, the authors measure the forward-propagation time, which they deem relevant for model deployment, and the back-propagation time, important for model training. We have reproduced an excerpt of their results in Table II, where we show their outcomes on (a) a CPU running 12 threads and (b) a GPU. Interestingly, for (a), TensorFlow ranks second behind Torch in both the forward and backward measure while in (b) TensorFlow's performance drops significantly, placing it *last* in both categories. The authors of [20] note that one reason for this may be that they used the NVIDIA cuDNN *v2* library for their GPU implementation with TensorFlow while using cuDNN *v3* for the others. They state that as of their writing, this was the recommended configuration for TensorFlow[4].

The second source in our collection is the *convnet-benchmarks* repository on GitHub by Soumith Chintala [22], an artificial intelligence research engineer at Facebook. The commit we reference[5] is dated April 25, 2016. Chintala provides an extensive set of benchmarks for a variety of convolutional network models and includes many libraries, including TensorFlow, Torch and Caffe in his measurements. Theano is not present in all tests, so we will not review its performance in this benchmark suite. The author's hardware configuration is a 6-core Intel Core i7-5930K CPU @ 3.50GHz and an NVIDIA Titan X graphics chip running on Ubuntu 14.04. Inter alia, Chintala gives the forward and backward-propagation time of TensorFlow, Torch and Caffe for the AlexNet CNN model [23]. In these benchmarks, TensorFlow performs second-best in both measures behind Torch, with

[4]As of TensorFlow 0.8, released in April 2016 and thus after the publication of [20], TensorFlow now supports cuDNN v4, which promises better performance on GPUs than cuDNN v3 and especially cuDNN v2.

[5]Commit sha1 hash: 84b5bb1785106e89691bc0625674b566a6c02147

Library	Forward (ms)	Backward (ms)
TensorFlow	16.4	50.1
Torch	**4.6**	**16.5**
Caffe	33.7	66.4
Theano	78.7	204.3

(a) CPU (12 threads)

Library	Forward (ms)	Backward (ms)
TensorFlow	4.5	14.6
Torch	**0.5**	1.7
Caffe	0.8	1.9
Theano	**0.5**	**1.4**

(b) GPU

TABLE II: This table shows the benchmarks performed by [20], where TensorFlow, Torch, Caffe and Theano are compared on a LeNet model reproduction [21]. IIa shows the results performed with 12 threads each on a CPU, while IIb gives the outcomes on a graphics chips.

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

TABLE III: The result of Soumith Chintala's benchmarks for TensorFlow, Torch and Caffe (not Theano) on an AlexNet ConvNet model [22], [23].

Caffe lagging relatively far behind. We reproduce the relevant results in Table III.

Lastly, we review the results of [5], published by the Theano development team on May 9, 2016. Next to a set of benchmarks for four popular CNN models, including the aforementioned AlexNet architecture, the work also includes results for an LSTM network operating on the Penn Treebank dataset [24]. Their benchmarks measure words processed per second for a small model consisting of a single 200-unit hidden layer with sequence length 20, and a large model with two 650-unit hidden layers and a sequence length of 50. In [5] also a medium-sized model is tested, which we ignore for our review. The authors state a hardware configuration consisting of an NVIDIA Digits DevBox with 4 Titan X GPUs and an Intel Core i7-5930K CPU. Moreover, they used cuDNN v4 for all libraries included in their benchmarks, which are TensorFlow, Torch and Theano. Results for Caffe are not given. 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 7 shows these results, taken from [5].

When TensorFlow was first released, it performed poor on benchmarks, causing disappointment within the deep learning community. Since then, new releases of TensorFlow have emerged, bringing with them improving results. This is reflected in our selection of works. The earliest of the three
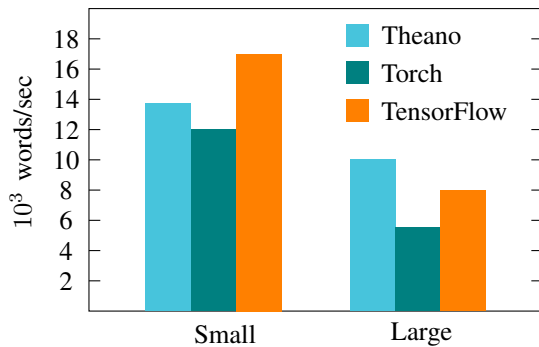
Fig. 6: The results of [5], showing a comparison of Tensor-Flow, Theano and Torch on an LSTM model for the Penn Treebank dataset [24]. On the left the authors tested a small model with a single hidden layer of 200 units; on the right they use two layers with 650 units each.

sources, [20], published in late March 2016, ranks TensorFlow consistently uncompetitive compared to Theano, Torch and Caffe. Released almost two months later, [22] ranks Tensor-Flow comparatively better. The latest work reviewed, [5], then places TensorFlow in first or second place for LSTMs and also other architectures discussed by the authors. We state that one reason for this upward trend is that [5] uses TensorFlow with cuDNN v4 for its GPU experiments, whereas [20] still used cuDNN v2. While we predict that TensorFlow will improve its performance on measurements similar to the ones discussed in the future, we believe that these benchmarks — also today — do not make full use of TensorFlow's potential. The reason for this is that all tests were performed on a *single* machine. As we reviewed in depth in section II-B, TensorFlow was built with massively parallel distributed computing in mind. This ability is currently unique to TensorFlow among the popular deep learning libraries and we estimate that it would be advantageous to its performance, particularly for large-scale models. We thus hope to see more benchmarks in literature in the future, making better use of TensorFlow's many-machine, many-device capabilities.

VI. Use Cases of TensorFlow Today

In this section, we investigate where TensorFlow is already 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. Migration from an existing system based on some other library within small and large organizations necessarily takes time and consideration, so this is not unexpected. The one exception is, of course, Google, which has already deployed TensorFlow for a variety of learning tasks [19], [25]–[28]. We begin with a review of selected mentions of TensorFlow in literature. Then, we discuss where and how TensorFlow is used in industry.

A. In Literature

The first noteworthy mention of TensorFlow is [29], published by Szegedy, Ioffe and Vanhoucke of the Google Brain Team in February 2016. In their work, the authors use TensorFlow to improve on the *Inception* model [19], which achieved best performance at the 2014 ImageNet classification challenge. The authors report a 3.08% top-5 error on the ImageNet test set.

In [25], 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 [30]. With this, the authors aim to preserve the coherence of quantum states, which is one of the primary requirements for building universal quantum computers.

B. In Industry

Adoption of TensorFlow in industry is currently limited only to Google, at least to the extent that is publicly known. We have found no evidence of any other small or large corporation stating its use of TensorFlow. As mentioned, we link this to TensorFlow's late release. Moreover, it is obvious that many companies would not make their machine learning methods public even if they do use TensorFlow. For this reason, we will review uses of TensorFlow only within Google, Inc.

Another area where Google applies deep learning with TensorFlow is smart email replies [27]. Google has investigated and already deployed a feature whereby its email service *Inbox* suggests possible replies to received email. The system uses recurrent neural networks and in particular LSTM modules for sequence-to-sequence learning and natural language understanding. An encoder maps a corpus of text to a "thought vector" while a decoder synthesizes syntactically and semantically correct replies from it, of which a selection is presented to the user.

In [26] it is reported how Google employs convolutional neural networks for image recognition and automatic text translation. As a feature integrated into its Google Translate mobile app, text in a language foreign to the user is first recognized, then translated and finally rendered on top of the original image. In this way, for example, street signs can be translated.

Lastly, we make note of the decision of Google DeepMind, an AI division within Google, to move from Torch7 to TensorFlow [28]. A related source, [17], states that DeepMind made use of TensorFlow for its *AlphaGo*[6] 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

---

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

were revealed to us as to why TensorFlow is advantageous to DeepMind:

1) TensorFlow is included in the Google Cloud Platform[7], 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, which implies 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. Its low-level programming interface gives fine-grained control for neural net construction, while abstraction libraries such as TFLearn allow for rapid prototyping with TensorFlow. In the context of other deep learning toolkits such as Theano or Torch, TensorFlow adds new features and improves on others. Its performance was inferior in comparison at first, but is improving with new releases of the library.

We note that very little investigation has been done in literature to evaluate TensorFlow's qualities with respect to distributed execution. We esteem this one of its principle strong points and thus encourage in-depth study by the academic community in the future.

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.

Appendix I

```python
#!/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()
```

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

```python
# 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)

    weights =
        tf.Variable(tf.truncated_normal(shape=[784,
        10], stddev=0.1))
    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 0.01
    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(1001):
        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)
```

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] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, J. FÃijrnkranz and T. Joachims, Eds. Omnipress, 2010, pp. 807–814. [Online]. Available: http://www.icml2010.org/papers/432.pdf

[3] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[4] L. Rampasek and A. Goldenberg, "Tensorflow: Biology's gateway to deep learning?" *Cell Systems*, vol. 2, no. 1, pp. 12–14, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.cels.2016.01.009

[5] 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.

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

[7] 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

[8] 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/

[9] R. Kohavi, D. Sommerfield, and J. Dougherty, "Data mining using mscr; lscr; cscr;++ a machine learning library in c++," in *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, Nov 1996, pp. 234–245.

[10] G. Bradski, "The opencv library," *Doctor Dobbs Journal*, vol. 25, no. 11, pp. 120–126, 2000.

[11] C. R. de Souza, "A tutorial on principal component analysis with the accord.net framework," *CoRR*, vol. abs/1210.7463, 2012. [Online]. Available: http://arxiv.org/abs/1210.7463

[12] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl, "Moa: Massive online analysis, a framework for stream classification and clustering," in *Journal of Machine Learning Research (JMLR) Workshop and Conference Proceedings, Volume 11: Workshop on Applications of Pattern Analysis*. Journal of Machine Learning Research, 2010, pp. 44–50.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[14] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *CoRR*, vol. abs/1505.06807, 2015. [Online]. Available: http://arxiv.org/abs/1505.06807

[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *CoRR*, vol. abs/1408.5093, 2014. [Online]. Available: http://arxiv.org/abs/1408.5093

[16] D. Murray, "Announcing tensorflow 0.8 âĂŞ now with distributed computing support!" Google Research Blog, April 2016 (accessed May 22, 2016), http://googleresearch.blogspot.de/2016/04/announcing-tensorflow-08-now-with.html.

[17] 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.

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

[19] 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.

[20] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative Study of Deep Learning Software Frameworks," *ArXiv e-prints*, Nov. 2015.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

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

[23] 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.

[24] 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

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

[26] 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.

[27] 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.

[28] 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.

[29] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *ArXiv e-prints*, Feb. 2016.

[30] 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

[31] G. Barzdins, S. Renals, and D. Gosko, "Character-Level Neural Translation for Multilingual Media Monitoring in the SUMMA Project," *ArXiv e-prints*, Apr. 2016.

[32] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[33] J. Clark, "Google turning its lucrative web search over to ai machines," Bloomberg Technology, Oct. 2015 (accessed: May 25, 2016), http://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-ai-machines.