A Tour of TensorFlow

Proseminar Data Mining

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

***Abstract*— Bar**

Foo

***Index Terms*— Artificial Intelligence, Machine Learning, Supervised learning, Unsupservised learning**

I. Introduction

Modern artificial-intelligence systems and machine-learning algorithms have revolutionized approaches to a myriad of scientific and technological challenges in a variety of fields. We can observe an immense proliferation in the quality of state-of-the-art computer-vision and image-recognition, natural language processing, speech recognition and other techniques. However, the major and most obvious benefactor of this revolution is mankind itself. Personalized digital assistants, recommendations on e-commerce platforms, financial fraud detection, customized web-search results and social-network feeds as well as novel breakthroughs 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 may be defined as 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 discovery of new algorithms such as the *rectified linear unit* (ReLU) [2] activation function or *dropout* as a regularization technique [3]; the greater availability of large data-sets, containing more training examples and lastly the efficient use of graphical processing units (GPUs) and massively parallel commodity hardware to train deep-learning models on these equally massive data-sets [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 follow. Section II will provide a brief overview and history of machine-learning software libraries, listing but not comparing projects similar to TensorFlow. Subsequently, Section III will discuss in depth the computational paradigms underlying TensorFlow. Section IV will then move to explaining the current implementation's programming interface in the various programming languages supported. In the following, Section VI provides a qualitative as well as quantitative comparison of TensorFlow and other *deep-learning* libraries. Before concluding our review in Section VIII, we also examine current real-world use-cases of and experiences with TensorFlow in Section VII.

II. History of Machine Learning Libraries

In this section, we aim to give a brief overview and key milestones in the history of machine-learning software libraries. We begin with a review of libraries suitable for a wide range of machine-learning and data-analysis purposes, reaching back more than 20 years. We then provide a more focused review of recent programming frameworks suited especially to the task of deep learning. Figure **??** visualized this section in a timeline. We wish to emphasize that this section does in no way compare TensorFlow, as we have dedicated Section IV for this specific purpose.

A. General Machine Learning

In the following paragraphs we aim to list and briefly review a small set of *general machine-learning libraries* in chronological order. With *general*, we mean to describe any particular library whose common use-cases in the machine-learning and data-science community include *but are not limited to* deep-learning. As such, these libraries may be used for statistical analysis, clustering, dimensionality reduction,

structured prediction, anomaly detection, shallow (as opposed to deep) neural networks and other interests.

We begin our review with a library released 21 years before TensorFlow: *MLC++* [9]. MLC++ is a software library developed in the C++ programming language providing algorithms alongside a comparison-framework for a number of of data-mining, statistical analysis as well as pattern-recognition techniques. It was originally developed at Stanford University in 1994 and is now owned and maintained by Silicon Graphics, Inc (SGI[1]). To the best of our knowledge, MLC++ is the oldest machine-learning library available today.

Following MLC++ in the chronological order, *OpenCV*[2] (**Open C**omputer-**V**ision) was released in the year 2000 by Bradski et al. [10]. It is aimed primarily at solving learning-tasks in the field of computer-vision and image-recognition, including a collection of algorithms for face-recognition, object-identification, 3D-model extraction and other purposes. It released under a BSD-license and provides interfaces in multiple programming languages such as C++, Python, MATLAB, among others.

Another machine-learning library we wish to mention is *scikit-learn*[3] [7]. The scikit-learn project was originally developed by David Cournapeu as part of the Google Summer of Code program[4] in 2008. It is an open-source machine-learning library written in Python, on top of the NumPy, SciPy and matplotlib Python-libraries and useful for a large class of both supervised and unsupervised learning problems.

The *Accord.NET*[5] library stands apart from the aforementioned examples in that it is written in the C# ("C-Sharp") programming language. Released in 2008, it is composed not only of a variety of machine-learning algorithms, but also signal-processing modules for speech and image recognition [11].

Massive Online Analysis[6] (MOA) is an open-source framework for online and offline analysis of massive, potentially infinite, data-*streams*. MOA includes a variety of tools for classification, regression, recommender systems and further domains. It is written in the Java programming language and is maintained by staff of the University of Waikato, New Zealand. It was conceived in 2010 [12].

The *Mahout*[7] project, part of Apache Software Foundation[8], is a Java programming environment for scalable machine-learning applications, built on top of the Apache Hadoop[9] platform. It allows for analysis of large datasets distributed in the Hadoop Distributed File System (HDFS) using the *MapReduce* programming paradigm and includes machine-learning algorithms for classification, clustering and filtering.

Pattern[10] is a Python machine-learning module we include in our list due to its rich set of *web-mining* facilities. It includes not only general machine learning algorithms (e.g. clustering, classification or nearest-neighbor search) and natural language processing methods (e.g. n-gram search or sentiment analysis), but also a web-crawler that can, for example, fetch Tweets and Wikipedia entries, facilitating quick data-analysis on these sources. It was published by the University of Antwerp in 2012 and is open-source.

Lastly, *Spark MLlib*[11] is an open-source machine-learning and data-analysis platform released in 2015 and built on top of the Apache Spark[12] project [13], a fast cluster-computing system. Similar to Apache Mahout, it aims to support processing of large-scale *distributed* datasets and training of machine-learning models across a cluster of commodity hardware. For this, it includes classification, regression, clustering and other machine-learning algorithms [14].

B. Deep Learning

While the software libraries mentioned in the previous section are useful for a great variety of different machine-learning and statistical analysis tasks, the following paragraphs list software frameworks especially effective in training deep-learning models.

The first and oldest framework in our list suited to the development and training of deep neural networks is *Torch*[13], released already in 2002 [6]. Torch consisted originally of a pure C++ implementation and interface. Today, its core is implemented in C/CUDA while it exposes an interface in the Lua[14] scripting language. For this, Torch makes use of a LuaJIT (just-in-time) compiler to connect Lua routines to the underlying C implementations. It includes, inter alia, numerical optimization routines, neural network models as well as general-purpose n-dimensional array (tensor) objects.

Theano[15], released in 2008 [5], is another noteworthy deep-learning library. We note that while Theano enjoys greatest popularity among the machine-learning community, it is, in essence, not a machine-learning library at all. Rather, it is a programming framework that allows users to declare mathematical expressions *symbolically*, as computational graphs which may be optimized, eventually compiled and finally executed in a fast and efficient manner on either CPU or GPU devices. As such, [5] labels Theano a "mathematical compiler".

Caffe[16] is an open-source deep-learning library maintained by the Berkeley Vision and Learning Center (BVLC). It was released in 2014 under a BSD-License [15]. Caffe implemented in C++ and uses neural-network layers as its basic computational building blocks (as opposed to Theano and

[1] https://www.sgi.com/tech/mlc/
[2] http://opencv.org
[3] http://scikit-learn.org/stable/
[4] https://summerofcode.withgoogle.com
[5] http://accord-framework.net/index.html
[6] http://moa.cms.waikato.ac.nz
[7] http://mahout.apache.org
[8] http://www.apache.org
[9] http://hadoop.apache.org
[10] http://www.clips.ua.ac.be/pages/pattern
[11] http://spark.apache.org/mllib
[12] http://spark.apache.org/
[13] http://torch.ch
[14] https://www.lua.org
[15] http://deeplearning.net/software/theano/
[16] http://caffe.berkeleyvision.org

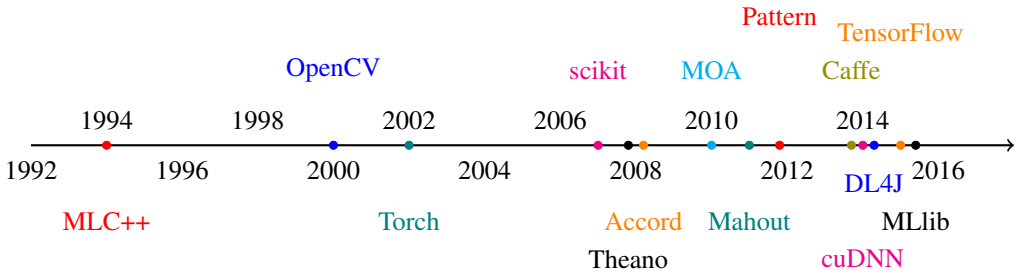


Fig. 1: A timeline showing the release of machine-learning libraries discussed in section I in the last 25 years.

others, where the user must define individual mathematical operations making up layers). A deep-learning model, consisting of many such layers, is stored in the Google Protocol Buffer data-representation format. While models may manually be defined in this Protocol Buffer "language", there exist bindings to Python and MATLAB to generate them programmatically. Caffe is especially well suited to the development and training of *convolutional neural networks* (CNNs or ConvNets), used extensively in the domain of image-recognition.

While the aforementioned machine-learning frameworks allowed for the definition of deep-learning models in Python, MATLAB and Lua, the *Deeplearning4J*[17] (DL4J) library enables also the Java programmer to create deep neural-networks. DL4J includes functionality to create Restricted Boltzmann machines, convolutional and recurrent neural networks, deep belief networks and other types of deep-learning models. Moreover, DL4J enables horizontal scalability using distributed computing platforms such as Apache Hadoop or Spark. It was released in 2014 by Adam Gibson, under an Apache 2.0 open-source license.

Lastly, we add the NVIDIA Deep Learning SDK[18] to to this list. Its main goal is to maximize the performance of deep-learning algorithms on (NVIDIA) graphical-processing-units (GPUs). The SDK consists of three core modules. The first, *cuDNN*, provides high-performance GPU implementations for deep-learning algorithms such as convolutions, activations functions and tensor transformations. The second is a linear-algebra library, *cuBLAS*, enabling GPU-accelerated mathematical operations on n-dimensional arrays. Lastly, *cuSPARSE* includes a set of routines for *sparse* matrices tuned for high efficiency on GPUs. While programming may be done in these libraries directly, there exist also bindings to other deep-learning libraries, such as Torch[19].

III. The TensorFlow Programming Model

In this section we aim to 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 models are assigned to available hardware processing 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 **??**. The following paragraphs discuss the principle elements of such a dataflow graph, namely *operations*, *variables*, *tensors* 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.

[17] http://deeplearning4j.org
[18] https://developer.nvidia.com/deep-learning-software
[19] https://github.com/soumith/cudnn.torch

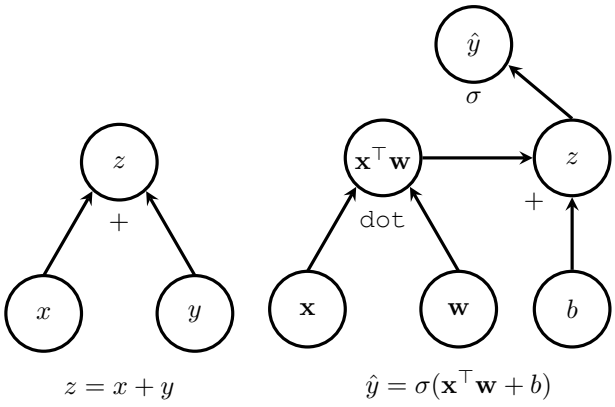


Fig. 2: Examples of computational graphs. The left graph displays a very simple computation, consisting of just an addition of the two input variables x and y. In this case, z is the result of the operation $+$, as the annotation suggests. The right graph shows a more complex example of computing a logistic regression variable $\hat{y}$ in dependence of some example vector $\mathbf{x}$, weight vector $\mathbf{w}$ as well as a scalar bias b. As can be seen in the graph, $\hat{y}$ is the result of the *sigmoid* function σ, also known as the *logistic* function.

Category	Examples
Elementwise 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].

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 homogenous 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 the size, i.e. the number of components, of the tensor 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. Most often, one wishes to update or *train* this shared state either manually or as part of the execution of the graph. 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 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.

When creating a variable node for a TensorFlow graph, one must always supply a tensor with which the variable is initialized upon graph execution. The shape and data-type of the variable is then deduced from this initializer. Interestingly, the variable iself does not store this initial tensor. Rather, constructing a variable results in the addition of *three* distinct nodes to the graph:

1) The actual variable node, holding the mutable state.
2) An operation producing the initial value, often a constant.
3) An *initializer* operation, that `assigns` the initial value to the variable tensor upon evaluation of the graph.

An example for this is given in Figure **??**.

4) Sessions: In TensorFlow, the execution of operations and the evalution 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 **??**, Moreover, as there exists the possibility to specify explicit orderings of node evaluations, called *control dependencies*,

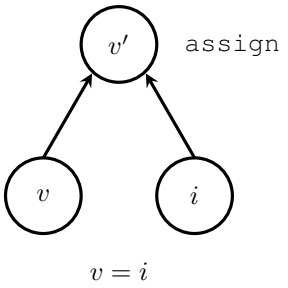


Fig. 3: The three nodes that are added to the computational graph for every variable definition. The first, v, is the variable operation that holds a mutable in-memory buffer containing the value-tensor of the variable. The second, i, is the node producing the initial value for the variable, which can be any tensor (that is, the result of any operation). Lastly, the `assign` node will set the variable to the initializer's value when executed. The `assign` node also produces a tensor referencing the initialized value v' of the variable, such that it may be connected to other nodes as necessary (e.g. when using a variable as the initializer for another variable).

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 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.

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 in a cluster. Figure **??** visualizes a possible distributed setup. Since 13 April 2016, the distributed version of TensorFlow is also part of its open-source distribution [16].

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 additional types of *physical execution units* by the user. For example, in May 2016, Google announced its *Tensor Processing Unit* (TPU), which is a custom-built ASIC (application-specific-integrated-circuit) optimized specifically for fast tensor-computations [**?**]. It is thus understandibly easy to integrate new device classes as novel hardware emerges.

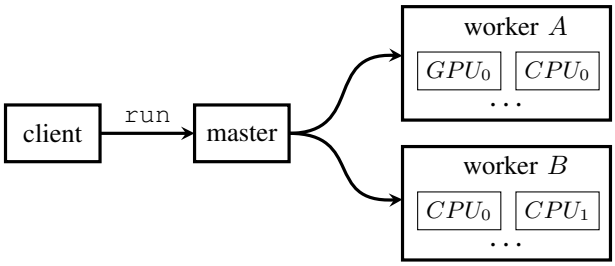


Fig. 4: A visualization of the different execution agents in a multi-machine, multi-device hardware configuration.

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 which 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) of the input and output tensors of the node.
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 provided more than one device, the placement algorithm will often have distributed the graph 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 exists cross-device edges $\nu \to \alpha$ and $\nu \to \beta$ from device A to device B. This is visualized in Figure **??**.

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 by three new nodes and an additional edge. On device A, a `send`-node is placed and connected to ν. In tandem, on device B, two `recv`-node are instantiated and connected to α and β, respectively. The `send` and `recv` nodes are then connected by new edges. This step is shown in Figure **??**. During execution of the graph, cross-device communication of data occurrs exclusively via these special nodes. If the devices are on the same machine, the transmission will most likely occur over a PCI Express bus. When the devices are located on separate machines, the 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 $(\texttt{send}, \texttt{receive})$ pairs. In the setup displayed in Figure **??**, 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 much more efficient transformation places only one `recv` node on device B and 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 **??**.

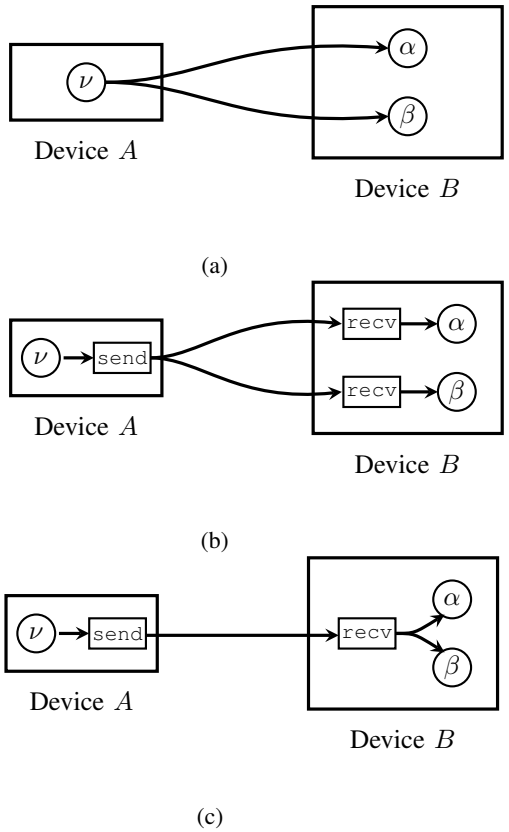


Fig. 5: The three stages of cross-device communication between graph nodes in TensorFlow. Figure **??** shows the initial, conceptual connections between nodes on different devices. Figure **??** gives a more practical overview of how data is actually transmitted across devices using `send` and `recv` nodes. Lastly, Figure **??** shows the final, canonicalized setup, where there is at most one `recv` node per destination device.

C. Optimizations

To ensure a maximum of efficiency and performance of the TensorFlow execution model, a number of optimizations are built into the library. In this subsection, we examine three such improvements: common subgraph elimination, execution scheduling and finally lossy compression.

1) Common-Subgraph Elimination: An optimzation performed by many modern compilers is *common subexpression elimination*, whereby a compiler may possibly replace the computation of an identical value two or more times by a single instance of that computation. The result is then stored in a temporary variable and reused where it was previously recalculated. Similarly, in a TensorFlow graph, it may occur that the same operation is performed on identical inputs more than once. This can be inefficient if the computation happens to be an expensive one. Moreover, it may incur a large memory overhead given that the result of that operation must be held in memory multiple times. Therefore, TensorFlow also employs a common-subexpression, or, more aptly put, common *subgraph* elimination pass prior to execution. For this, the computational graph is traversed and every time two or more operations of the same type (e.g. `MatMul`) receiving the same input tensors are encountered, they are canoncalized to only one such subgraph. The output tensor of that single operation is then redirected to all dependent nodes. Figure **??** gives an example of common subgraph elimination.

2) Scheduling: A simple yet powerful optimization is to schedule node execution as late as possible. Ensuring that the results of operations remain in memory only for the minimum required amount of time reduces peak memory consumption and can thus greatly improve the overall performance of the system. The authors of [8] note that this is especially vital on devices such as GPUs, where memory resources are scarce. Furthermore, careful scheduling also pertains to the activation of `send` and `recv` nodes, where not only memory but also network resources are contested.

3) Lossy Compression: One of the primary goals of many machine-learning algorithms used for classification, recognition or other tasks is to build robust models. With *robust* we mean that an optimally trained model should ideally not change its response if it is first fed a signal and then a noisy variaton of that signal. As such, these machine-learning algorithms typically do not require high precision arithmetic as provided by standard IEEE 754 32-bit floating point values. Rather, 16 bits of precision in the mantissa would do just as well. For this reason, another optimization performed by

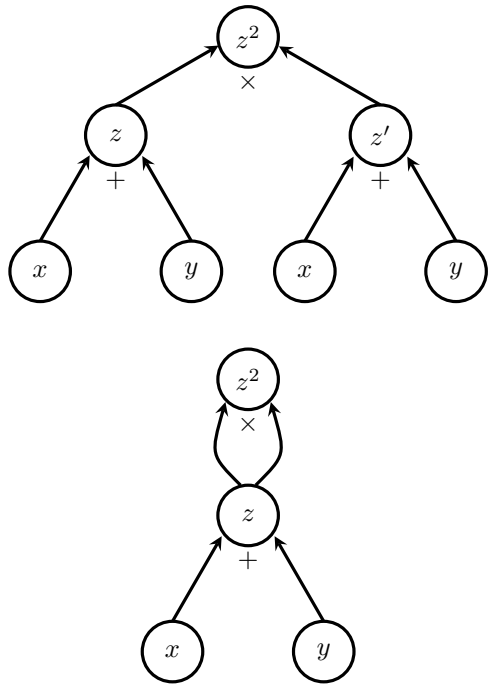


Fig. 6: An example of how common-subgraph elimination is used to transorm the equations $z = x + y$, $z' = x + y$, $z^2 = z \cdot z'$ to just two equations $z = x + y$ and $z^2 = z \cdot z$. This computation could theoretically be optimized further to a `square` operation requiring only one input (thus reducing the cost of data movement), though it is not known if TensorFlow employs such secondary canonicalization.

TensorFlow is the internal addition of conversion nodes to the computational graph, which convert such high-precision 32-bit floating-point values to truncated 16-bit representations when communicating across devices and across machines. On the receiving end, the truncated representation is converted back to 32 bits simply by filling in zeros, rather than rounding [8].

D. Additions to the Basic Programming Model

Having discussed the basic computation and execution model of TensorFlow, we will now review three more advanced topics that we deem highly relevant for anyone wishing to use TensorFlow to create machine-learning algorithms. First, we will discuss how TensorFlow handles *gradient backpropagation*, an essential concept for many deep-learning applications. Then, we will discuss in what way TensorFlow graphs support *control-flow*. Lastly, we briefly touch upon the topic of *checkpoints*, as they are very useful for maintenance of large models.

1) Back-Propagation Nodes: 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 and g, we can express the cost for that example as $c = (f \circ g)(x) = f(g(x))$. We would then typically wish to calculate the gradient dc/dx of that cost with respect to the input value x and use it to update certain internal parameters. Usually, we would do this by means of the *back-propagation* algorithm, which traverses the graph in reverse to compute the chain rule $[f(g(x))]' = f'(g(x)) \cdot g'(x)$.

In [**?**], 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 via the chain rule. Another approach, more relevant to TensorFlow, is what [**?**] 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 (that is, the tensor produced by that operation) 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 $(\nu \circ \ldots\ o \circ \ldots)(\alpha)$ 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 gradient $\frac{do}{d\alpha}$. At the end of the traversal, there will be a node providing a symbolic handle to the overall target derivative $\frac{d\nu}{d\alpha}$. It should now be clear that back-propagation in this symbol-to-symbol approach is implemented just like *any other computation* and requires no exceptional handling. Figure 7 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 is given in Equation 1 for example functions f, g and h. The second 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.

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

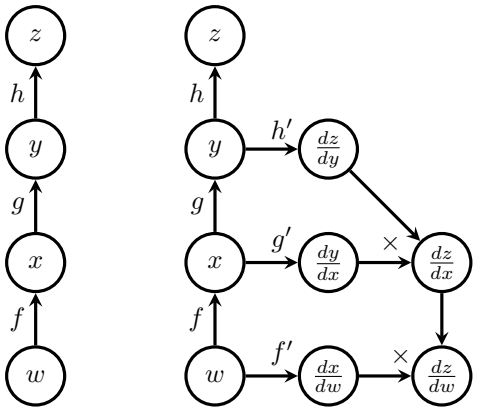


Fig. 7: A computational graph before (7a) and after (7b) gradient nodes are added. In this *symbol-to-symbol* approach, the gradient $\frac{dz}{dw}$ is just simply an operation like any other and therefore requires no special handling by the graph evaluation engine.

$$\frac{df}{dw} = f'(g(h(w))) \cdot g'(h(w)) \cdot h'(w) \tag{2}$$

According to [8], TensorFlow currently employs the first approach. Given that the inner-most functions must be re-computed for 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.

2) Control-Flow: Some machine-learning algorithms may benefit from being able to control the flow of their execution, performing certain steps only under a particular condition or repeating some computation a fixed or variable number of times. For this, TensorFlow provides a set of control-flow primitives including `if`-conditionals and `while`-loops. The possibility of loops is the reason why a TensorFlow computational graph is not necessarily *acyclic*. If the number of iterations for of a loop would be fixed and known at graph compile-time, its body could be *unrolled* into an acyclic sequence of computations, one per loop iteration [5]. However, to support a variable amount of iterations, TensorFlow is forced to jump through an additional set of hoops, as described in [8].

One aspect that must be especially cared for when introducing control-flow is back-propagation. In the case of a conditional, where an `if`-operation returns either one or the other tensor, its gradient node must be aware of which branch the node took during forward-propagation. Moreover, when a loop body (which may be a small graph) was executed a certain number of times, the gradient computation does not only need to know the number of iterations performed, but also requires access to each intermediary value produced. This technique of stepping through a loop in reverse to compute the gradients is referred to as *back-propagation through time* in [5].

3) Checkpoints: A small additional extension to TensorFlow's basic programming model is the notion of *checkpoints*, which allow for serialization of variables and storage on disk. It is possible to add `Store` nodes to the computational graph and connect them to variables whose tensors you wish to save. Furthermore, a variable may be connected to a `Restore` operation, which deserializes a stored tensor at a later point. This is especially useful when training a model over a long period of time to keep track of the model's performance as time progresses while reducing the risk of losing any progress made. Also, checkpoints are a vital element to ensuring fault-tolerance in a distributed environment [8].

IV. The TensorFlow Programming Interface

In this section we aim to concretize the abstract concepts of TensorFlow's computational model conveyed in Section III and speak about the TensorFlow software library's programming interface. We begin with a brief review of the available interfaces in subsection IV-A. Then, we provide a more hands-on look at TensorFlow's Python API by walking through a simple practical example. Lastly, we give insight into what higher-level abstractions exist for TensorFlow's API, which are especially suitable for rapid prototyping of machine-learning models.

A. Interfaces

There currently exist two programming interfaces, in C++ and Python, that allow interaction with the TensorFlow backend. The Python API boasts a very rich feature set allowing for creation and execution of computational graphs. As of this writing, the C++ interface (which is really just the core backend implementation) provides comparatively much more limited functionality, allowing only the execution of graphs built with Python and serialized to Google's Protocol Buffer[20] format. While there is experimental support for also building computational graphs in C++, this functionality is currently not as extensive as for the Python API.

It is noteworthy that the Python API integrates very well with NumPy[21], a popular open-source Python numeric and scientifc 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

[20] https://developers.google.com/protocol-buffers/
[21] http://www.numpy.org

(MLP) with one input and one output layer to classify handwritten digits in the MNIST[22] 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 grayscale 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`, that takes as arguments a path to extract and store the dataset. Moreover, we pass the keyword argument `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:

```
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)
```

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 $X \in \mathbb{R}^{n \times 784}$ containing n images, the learning task then applies an affine transformation $X \cdot W + b$, where W is a *weight matrix* $\in \mathbb{R}^{784 \times 10}$ and b a scalar *bias* $\in \mathbb{R}$. This yields a new matrix $Y \in \mathbb{R}^{n \times 10}$, 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 these logits into a valid probability distribution, giving the likelyhood $\Pr[x = i]$ 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 $\text{softmax}(X \cdot W + b)$, as shown below.

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

```
# 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)
```

Next, we compute our objective function, producing the error or *loss* of the model given its current trainable parameters W and b. We do this by calculating the *cross entropy* $H(L, Y)_i = -\sum_j L_{i,j} \cdot \log(Y_{i,j})$ between the probability distributions of our estimates Y and the one-hot-encoded labels 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 repedeatedly 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 opera-

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

tion 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 minibatch 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.

C. Abstractions

You may have observed how a relatively large amount of effort was required to create just a very simple two-layer neural-network. Given that *deep*-learning, by implication of its name, makes use of very *deep* neural-networks with many hidden layers, it may seem infeasible to each time create weight and bias variables, perform a matrix multiplication and addition and finally apply some non-linear activation function. When testing ideas for new deep-learning models, scientists often wish to rapidly prototype networks and quickly exchange layers. In that case, these many steps may seem very low-level, repetitive and generally cumbersome. For this reason, there exist a number of open-source libraries that *abstract* these concepts and provide higher-level building blocks, such as entire layers. We find *PrettyTensor*[23], *TFLearn*[24] and *Keras*[25] especially noteworthy. The following paragraphs give a brief overview of the first two abstraction libraries.

1) PrettyTensor: PrettyTensor is developed by Google and provides a high-level interface to the TensorFlow API via the *Builder* pattern. It allows the user to wrap TensorFlow operations and tensors into "pretty" versions and then quickly chain any number of layers operating on these tensors. For example, it is possible to feed an input tensor into a fully-connected ("dense") neural-network layer as we did in Subsection IV-B with just a single line of code. Shown below is an example use of PrettyTensor, where a standard TensorFlow placeholder is wrapped into a library-compatible object and then feed through three fully-connected layers to finally output a softmax distribution.

```
examples = tf.placeholder([None, 784], tf.float32)
softmax = (prettytensor.wrap(examples)
           .fully_connected(256, tf.nn.relu)
           .fully_connected(128, tf.sigmoid)
           .fully_connected(64, tf.tanh)
           .softmax(10))
```

2) TFLearn: TFLearn is another abstraction library built on top of TensorFlow that provides high-level building blocks to quickly construct TensorFlow graphs. It has a highly modular interface and allows for rapid chaining of neural-network layers, regularization functions, optimizers and other elements. Moreover, while PrettyTensor still relied on the standard `tf.Session` setup to train and evaluate a model, TFLearn adds functionality to easily train a model given an example batch and corresponding labels. As many TFLearn functions, such as those creating entire layers, return vanilla TensorFlow objects, the library is well suited to be mixed with existing TensorFlow code. For example, we could replace the entire setup for the output layer discussed in Subsection IV-B with just a single TFLearn method invocation, leaving the rest of our code base *untouched*. Furthermore, TFLearn handles everything related to visualization wtih TensorBoard, discussed in Section V, automatically. Shown below is how we can reproduce the full 65 lines of standard TensorFlow code given in Appendix I with *less than 10 lines of code* using TFLearn.

```
import tflearn
import tflearn.datasets.mnist as mnist

X, Y, validX, validY = mnist.load_data(one_hot=True)

# Building our neural network
input_layer = tflearn.input_data(shape=[None, 784])
output_layer = tflearn.fully_connected(input_layer,
    10, activation='softmax')

# Optimization
sgd = tflearn.SGD(learning_rate=0.5)
net = tflearn.regression(output_layer,
    optimizer=sgd)

# Training
model = tflearn.DNN(net)
model.fit(X, Y, validation_set=(validX, validY),
    show_metric=True)
```

V. Visualization of TensorFlow Graphs

High level discussion of what visualization is and why it is important

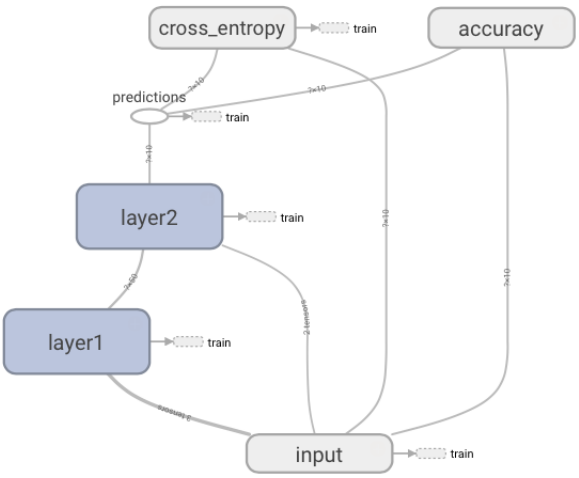
A. TensorBoard Features

Interesting are features:
1. 2. 3. 4. 5.

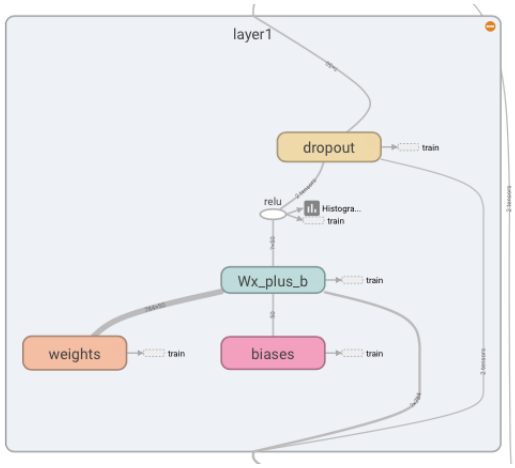
B. TensorBoard in Practice

To integrate, three elements are
1. name$_s copecontextmanager 2. scalar_s ummary 3. histogram_s umn$
Example

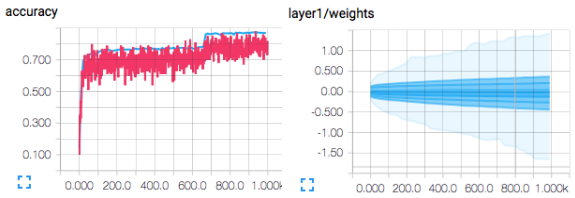
[23] https://github.com/google/prettytensor
[24] https://github.com/tflearn/tflearn
[25] http://keras.io



(a)



(b)



(c) (d)

Fig. 8: Foo

VI. Comparison With Other Deep-Learning Frameworks

VII. Use-Cases of TensorFlow Today

VIII. Conclusion

AppendixI

```
#!/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)

   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, 2016, accesse date: 22 May 2016.