Introduction to TensorFlow

Proseminar Data Mining

Emre Kavak
Fakultät für Informatik
Technische Universität München
Email: emre.kavak@tum.de

***Abstract*— TensorFlow, released and open sourced in November 2015 by Google, is a machine learning framework that is able to execute its models on large-scale distributed settings. This paper illustrates how TensorFlow accomplishes efficient distributions by analyzing the underlying architecture and elucidating what the programming paradigm contributes to this property. Furthermore, this paper mentions different optimization strategies that are used to improve the model execution performance. In a comparison with Theano, Torch and Caffe it is presented how TensorFlow distinguishes itself from other frameworks. Moreover, a benchmark in the same section shows what factors influence the execution performance. The last part of the paper demonstrates some research papers and industrial applications that report having used TensorFlow successfully in their works.**

***Index Terms*— Artificial intelligence, Application software, Machine learning, Neural Networks, Open source software, Software packages**

I. Introduction

Machine learning methods are frequently used to solve different kinds of problems in various areas. The applications reach from autonomous driving to the use of learning models in the area of quantum computing [1]. But regardless of how substantially the areas of application differ, the means to express such models is always the same. Computers are utilized to perform these tasks in order to generate the expected results.

This paper introduces *TensorFlow* that enables the user to express such machine learning models as a computer program and to execute it subsequently. TensorFlow was initially built by the *Google Brain* team using insights gained by the predecessor system *DistBelief* [2], [3]. DistBelief was a proprietary system with the aim to solve Google internal tasks that needed machine learning approaches. TensorFlow, however, is now not only a Google domestic tool but also an open source project with a continuously growing community.

Although TensorFlow is designed for general machine learning algorithms, many sources entitle it a deep learning framework. Several reasons may accompany this notion.

One of them is that TensorFlow uses advanced visualization and reporting techniques that make the whole computational system transparent to the user. Since deep networks tend to be complex and obscure, the TensorBoard visualizer comes remarkably handy by allowing to draw computations as a graph and to add histograms, tables and other kinds of plots to track certain properties [4], [5].

Another key feature of TensorFlow is that it allows the execution of distributed machine learning models. There are only a few frameworks other than TensorFlow that have this trait. Especially noteworthy is the modularity in which TensorFlow achieves distribution since it allows even the use of mobile devices and more. This leads to the fact that large models can be parallelly trained using TensorFlow which again is definitely required in deep learning settings. The multitude of layers and neurons included in such deep models demand high computational power that can be achieved by dividing the labor on distinct machines.

Additionally, TensorFlow provides the feature of tailored code for different computational devices. Amongst them are the ordinary *CPU* and the more specialized *graphics processing unit (GPU)*.

Lastly, possibly the most obvious reason why TensorFlow is called a deep learning framework, stems from its origin as a Google internal tool that served mainly deep learning purposes in several different fields. Thus, there are already pre-defined and optimized solutions for various kinds of neural networks. The library extension *contrib.rnn* for example serves the purpose of providing *recurrent neural network* implementations [6]. Whilst there are several kinds of contributions to general machine learning algorithms, one can detect a slight focus on neural networks in the recent version of TensorFlow.

The aim of this paper is to illustrate how TensorFlow works in detail by giving simple and generic examples to make it easy to comprehend the outlined points, regardless of machine learning expertise.

Section II is completely dedicated to the inner workings of TensorFlow and how it handles technical issues enabling the user to solely focus on machine learning algorithms. Afterwards, in section III, TensorFlow is compared to other frameworks that are frequently used in science and industry. Moreover, this section will include a short benchmark on different machine learning models and data sets comparing various frameworks. To get in touch with real world applications, this paper reveals in section IV where TensorFlow is currently used. Section V, the concluding chapter, reviews some of the key features of TensorFlow and provides some thoughts on future work.

II. How does TensorFlow work?

TensorFlow separates the definition of a computation from its actual execution. The user defines a program using

symbolic handles and describes what the program aims to accomplish. TensorFlow represents this computation as a *graph* consisting of nodes and edges that are linked together.

In order to execute the computational graph, the user has to create a *session* that is responsible for resource allocation and storage of intermediate results. One can interpret the session as an environment where a certain machine learning model runs on and utilizes the resources provided by it. By invoking the run method on the session, the user transfers responsibility to the *master process* that orchestrates the graph execution. To be more precise, the master partitions the graph and distributes the corresponding tasks to different *worker processes* as Figure 1 depicts.

This section elucidates the mentioned procedure in more detail. First of all, it is discussed how a client defines a computation model in TensorFlow. Afterwards, this section illustrates the main components on the execution side and reveals how distribution works in TensorFlow. The final part of this section includes some optimization features that aim to accelerate computations.

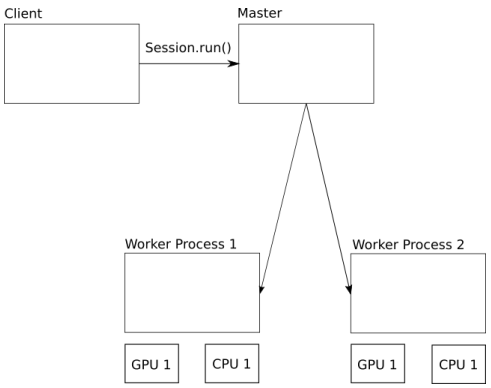


Fig. 1: The client invokes a run command through a session. The master process partitions the data flow graph and assigns tasks to the workers. In this illustrative example, the worker machines have both a CPU and a GPU [4].

A. Building a computational graph

To model a machine learning problem, the user can select a variety of possible *application programming interfaces* (APIs). Python is considered the main API since many data scientist and other types of researchers are familiar with this intuitive and easy to learn programming language. This leads to frequent contributions to the Python API by a large community. Important to mention here is that the modularity of TensorFlow allows a continuous extension of different interfaces and new features without impeding existing parts of the system. TensorFlow r1.0, released in February 2017, initially introduced a Java API [7]. In addition to these two APIs, there also exists the possibility to define an algorithm in Go or C++.

Ordinary programming, also known as the *imperative* paradigm, aims to describe how a program behaves step-by-step. TensorFlow, however, builds a symbolic graph with nodes and edges. In other words, it specifies a program *declaratively* as a computational graph, telling it what to accomplish but not how to attain it in detail. For instance, a TensorFlow client-side model does not make any suggestions in which order to execute the code.

Nodes represent operations and are linked by edges. Edges are symbolic placeholders for data flows between different operations. These abstractions help to make the environment indifferent of execution details. For example, the session will not care how a certain node is implemented. It rather will pass it to the master process as in figure 1. The master again will treat nodes equally and distribute them on devices abstracting from implementation details.

Moreover, it has been proven useful in machine learning to represent operations as nodes. They can be interpreted as functions in a mathematical way and composed as such. This again leads to the helpful fact that automatic gradient computations can be performed on them [4].

Operations and kernels Operations embody the first building blocks of the TensorFlow model and are represented as nodes in a graph. The intuition behind this analogy is that operations require inputs and create outputs after performing certain transformations. In a graph, such an operation can thereby be mirrored as a node with incoming and outgoing edges. Furthermore, an operation has a unique name and can be identified by it. Table I shows some of the standard built-in operations in TensorFlow.

Nodes do not contain any information about their real implementations. The actual manifestation in form of code is stored in so called *kernels*. If we consider matrix multiplication as an example, then there exist at least two versions. CPU kernels are programmed in C++ and GPU instructions in *CUDA* [8]. The goal is to obtain efficient implementations tailored for given devices.

If the user wants to implement novel operations, the first approach is to compose existing functions on the high-level API to achieve the desired result. However, there might be new operations not representable as compositions of existing ones due to efficiency concerns or plainly because they cannot be assembled using current parts. In that case, a new definition of kernels is required.

Tensors To stay with the interpretation of a graph for a TensorFlow program, the next step is to define what edges are. Edges are represented by *tensors*. Tensors, again, can be characterized as multidimensional arrays that hold data for computation models. They are defined implicitly by

Category	Examples
Element-wise operations	Add, Sub, Mul
Matrix operations	MatMul, MatrixInverse
Control flow operations	Constant, Variable
Neural network units	SoftMax, ReLU, Conv2D

TABLE I: Examples of some TensorFlow operations [4].

annotating a variable or constant with a data type such that the session knows what kind of array it should use to represent intermediate results. Noteworthy here is that tensors are stateless and cannot be accessed somewhere else in the computation. Their only purpose is to connect related edges to pass information amongst them.

The transition from the symbolic representation of the model to the actual compiled program is invoked by the run command provided by the session interface. The session represents a bridge from the client to the master by computing the transitive closure of the graph inspecting all dependencies between them [4]. This allows the master to compute the nodes in an appropriate order enabling an efficient memory handling and timing of execution.

B. Running a TensorFlow model

Being invoked by the run method, the master has the task to prune the computational graph and mark specified subgraphs for execution. The result is a sequence of operations. Moreover, it will partition the subgraphs and assign them to different worker processes that shall execute these. The decision how to distribute the partitions is handled by the *placement algorithm*. Having received information of which parts they are responsible for, the workers perform computations based on kernel definitions utilizing their devices (e.g. CPU or GPU). To combine different intermediate results across machine or device borders, various communication protocols are provided to guarantee efficient data transfer. This section sheds light on these mentioned steps in detail.

The placement algorithm The Google Brain team published a first version of the node placement algorithm emphasizing that they steadily keep developing it further [4]. Its task is to distribute the graph partitions alongside all available devices trying to achieve a near optimal solution.

The node placement is based on the principle of *Greedy algorithms*. The intuition behind this type of algorithm is to choose always a possibility that has the most benefit for a partial problem and to combine each of these solutions afterwards.

In the context of node placement, the greedy approach puts a given node on a device such that the execution of this operation is as fast as possible. Following this step, the algorithm decides for the next node on the same principle, thus, trying to maximize the profit of that one again.

It is important to see that the combination of optimal partial solutions does not necessarily lead to an overall optimal solution. But a Greedy approach is sufficient in most cases [9]. The following metrics and heuristics are included in the placement decision [4]:

1) The time to compute node n on device d. The higher the duration, the more will the cost increase. If d does not support given n, then d is excluded from the placement decision for n.

2) Communication delays caused by distribution. Such postponements increase the cost function.

3) Input and output sizes of tensors that need to flow through the node. Larger data sizes need more time and thus increase the cost.

The overall approach is to select a node and the suiting device that minimizes the cost function. This procedure is repeated until there is no unprocessed node left.

The device chosen in each step by this cost calculation is also the one that will execute the corresponding node.

Devices The entities that perform operations are named *devices*. Usually, CPUs and GPUs are used for various computations. The latter ones became notably popular in the field of deep learning. They outperform CPUs on certain repetitive matrix calculations due to their extremely high amount of small cores. In particular, GPUs are often explicitly adjusted for certain tasks while this may not be the case for many CPU architectures. [10]–[12]. TensorFlow goes even a step further. Any kind of user is always allowed to register novel devices if preferred. Google's *tensor processing units* (TPUs) are optimized computational units for deep learning calculations in the TensorFlow environment [13].

Distributed communication These devices can be dispersed on several different machines. Communication between different workers is handled by the open-source gRPC framework over TCP and by using RDMA over Converged Ethernet [14]. In order to attain the information where the tensors have to go in a distributed setting, special nodes are introduced to handle it. Correspondingly, these nodes are named *Send* and *Receive*. Figure 2 shows how these nodes are successively added and canonicalized.

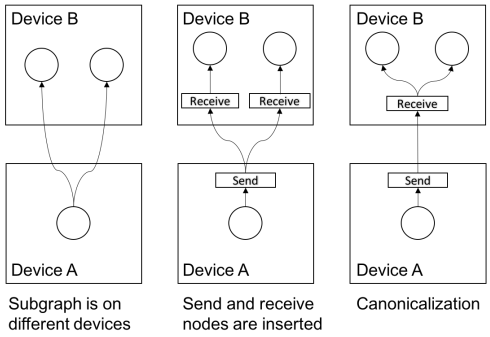


Fig. 2: In cross-device execution, the system needs to handle tensors that flow from devices A to B. New nodes are inserted for this purpose. If two nodes in a device expect the same input, they can be canonicalized as presented here. [4].

The last optimization step pools redundant senders and receivers into one. Benefits of this procedure are [4]:

1) Communication cost: identical tensors that are sent twice cause unnecessary communication overheads, possibly slowing down other data transfers taking place on the same device.

2) Storage: tensors that are sent more than once will allocate

memory multiple times (as often as they are sent).

In general, these nodes also carry the benefit of being able to handle all scheduling arrangements between different tasks and devices by encapsulating the complete communication details in them. Thus, a worker is responsible for its own data exchange and synchronizes with other machines implicitly through these nodes.

If these nodes were not included, the master process would have to define each data transfer constraint explicitly, leading to bad scaling abilities. Outstandingly adverse would be the situation when a model was executed on plenty of machines and devices, and thus making TensorFlow, contrary to its objectives, less scalable.

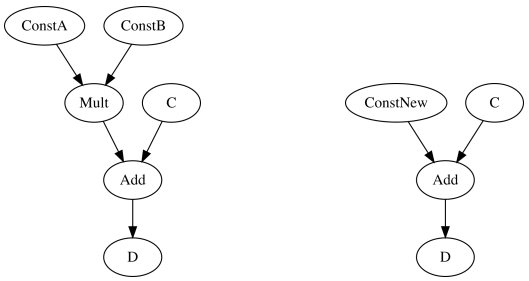
C. Optimizations

A key concern in machine learning frameworks is how fast programs are performed. Research and engineering teams devote much time to performance tuning with the aim to accelerate program execution.

Such optimizations can be carried out on different levels. This subsection presents two techniques that are typical for compilers and therefore belong to a set of optimizations performed prior to execution.

Before the partitions are distributed on different worker processes and their devices, the graph is simplified and improved first [4], [14].

1) *constant folding:* applying multiple operations consequently to variables that hold constant values is a case for *constant folding*. Instead of saving all intermediate results and passing them around, the system consolidates the multitude of steps to just one constant by applying all required operations on it as in Figure 3.

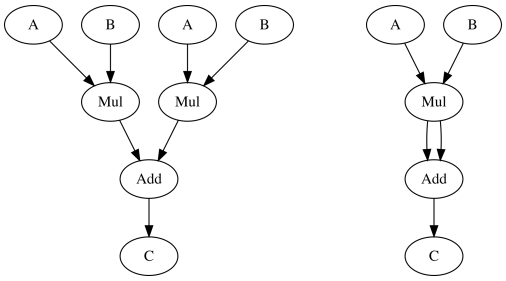


(a) pre-folding (b) folded graph

Fig. 3: The compiler processes the graph in order to fold successive operations into one. Note that ConstA and ConstB are placeholders that are initialized with constant values. ConstNew is the resulting placeholder carrying the sum of them. C is a variable with unknown reference and thus cannot be folded by the compiler at this point.

2) *common subgraph elimination:* another technique that helps to employ a graph more efficiently is to reuse parts of it that are needed multiple times. This procedure is called *canonicalization* as described in [15]. Figure 4 shows how the program defines the same operation, having identical inputs, in two distinct locations of the graph. But instead of computing common subgraphs more than once, the elimination technique detects redundant parts and deletes them before they are executed.

The result is that this certain subgraph only occurs once in the entire computation. This procedure will reduce both unnecessary computation costs and redundant storage of affected variables that need to be held in memory.



(a) Unoptimized version (b) Optimized version

Fig. 4: Multiplication of A and B are processed twice. Optimization saves storage since the intermediate results must be held in memory. Furthermore, computation time is saved due to omitting redundant steps.

III. TensorFlow in relation to other frameworks

This section provides some comparisons between the most significant frameworks that are used in research and industry. The decision fell on Theano, Caffe, and Torch since these are frameworks with the largest communities. Additionally, these systems are accompanied with much more scientific papers and support than other frameworks.

A. General Overview

1) *Theano*: Being developed by researchers at Université de Montréal, this framework is supported intensively by several scientists. It is the library with most commonalities to TensorFlow in this comparison. Both, TensorFlow and Theano, declare their algorithms as computational graphs.

Theano solely has a Python front-end, whereas TensorFlow has a wider range of programming languages (Java, C++, Go) to describe models. A more significant difference lies in the execution phase. Whilst TensorFlow has pre-compiled kernels for different operations, Theano is more flexible by always compiling the computation graph dynamically into more efficient C++ (CPU) and CUDA (GPU) code. This leads to more opportunities for complex optimizations and more specialized low-level code [16]–[18].

The disadvantages are, however, the immensely longer transition times when Theano is processing the graph and generating low-level instructions out of it. TensorFlow just uses its pre-compiled kernels without the need to translate code. Furthermore, contrary to TensorFlow, there is no standard and stable solution to distribute machine learning models in Theano. Also in terms of visualization, TensorFlow has a clear advantage with its TensorBoard feature [5], [19].

2) *Caffe*: In this comparison, Caffe is the framework that entirely focuses on deep learning. It was created by the

Vision and Learning Center at UC Berkeley. The community is mainly placed in the area of computer vision and similar institutions.

Using protobuf to define a neural network, it is quite different to TensorFlow and Theano. The latter ones define a layer by a multitude of nodes (operations). Caffe's smallest part, on the contrary, is a layer, making it more rigid and restricted. Models are defined in separate configuration files and isolated from its execution as it is the case with Theano and TensorFlow.

Similar to Theano, Caffe does not support distribution either and has only simple methods to visualize the computation model [20]–[22].

3) *Torch*: The last framework in this comparison is Torch. It uses LuaJIT as its frontend language and accesses CPU and GPU by using C and CUDA directly. The community is significant and contains names such as Facebook, NYU, Twitter and more.

The programming paradigm is imperative, thus, the code will be executed sequentially while in TensorFlow and Theano the whole graph is observed first before executing the model. Therefore, the other two mentioned frameworks can optimize their resource usage since they know in advance for how long a variable is needed and can deallocate them on time.

However, the speed of the LuaJIT compiler combined with the fact that there is no pre-compiler phase, lead to significantly lower transition times from a model to its execution. Finally, there is no solution for Torch to distribute computations on different machines so far and it also lacks visualization methods. [23].

B. Performance on deep learning models

Having seen a general overview of three additional frameworks, it is also interesting to know how they perform on real data sets. In order to attain a reliable view on the runtimes of these frameworks, benchmarks that were performed or at least recreated by other researchers will be reviewed here. This paragraph does not aim to give a full and detailed benchmark. It rather shows how fast benchmark results change over time and how different hardware settings and device accelerating libraries influence the throughput of frameworks.

The first benchmark we preview here compares Theano, Torch and TensorFlow [16]. The given sets of different machine learning models are executed on an Intel Core i7-5930K CPU @ 3.50GHz and are supported by the NVIDIA Titan X GPU. More details on AlexNet, OverFeat and GoogleLeNet can be found in the following papers [24]–[26]. The actual benchmark was performed and shared by the Facebook engineer Soumith Chintala [27].

It is important to see that TensorFlow and Theano use the cudNN v4 library from Nvidia that accelerates CUDA programming for deep learning tasks in this benchmark [28]. For Torch, this paper contains a version both with and without the cudNN v4 support. Caffe, on the contrary, is only represented with its native configuration lacking the cudNN acceleration, although it also has a binding to it and can access performance gains through that library.

The idea behind this constellation is, on the one hand, to convey their performances on three well-known data sets and, on the other hand, to show the effects of seemingly small modifications, such as using a GPU-accelerating library for neural networks, that has an extraordinarily high impact on their data processing speed. Figure 5 displays that the frameworks using cudNN v4 are comparably similar in performance, with Torch and TensorFlow leading slightly and alternately. The difference becomes truly immense when one compares how Torch performs when it disables its usage of the cudNN library and experiences immense performance losses.

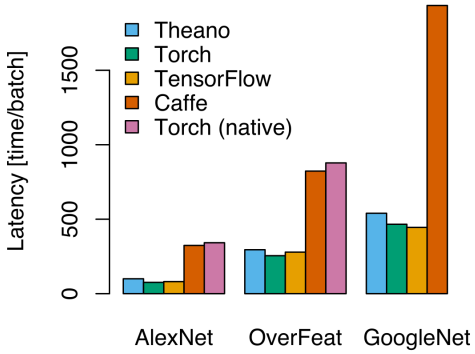


Fig. 5: TensorFlow, Theano, and Torch utilize the cudNN v4 framework for GPU computations while Caffe is used in its native variant here [16], [27].

The second benchmark is from a paper studying different deep learning frameworks and conducting a performance analysis across various models and settings [21]. The research team of Robert Bosch LLC compares the frameworks on different hardware and sequentially adjusts features, e.g. thread count, GPU support, and batch sizes.

As Table II indicates, TensorFlow is the second fastest when it comes to running the LeNet [29] data set and model with a batch size of 64. However, allowing to use a GPU changes the distribution of processing duration per batch entirely, which can be observed in Table III. Similarly, it is crucial to notice that Caffe, Torch, and Theano used cudNN v3 while TensorFlow had only cudNN v2 support to that time [21].

The overall goal of this subsection is to present that it is not trivial to claim which framework outperforms others. During the time of the results of [21], TensorFlow was clearly behind other frameworks in terms of efficiency. [16], [27] show us, nevertheless, that a few months later the entire performance landscape can look completely different.

A crucial factor, however, is the continuous development

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

TABLE II: Performing on 12 kernel CPU [21]

Framework	Gradient(ms)	Forward(ms)
TensorFlow with cudNN v2	14.6	4.5
Theano with cudNN v3	**1.4**	**0.5**
Torch with cudNN v3	1.7	**0.5**
Caffe with cudNN v3	1.9	0.8

TABLE III: Performing on GPU [21]

and consequently the community behind a framework. Seemingly small changes as renewing the GPU acceleration can cause a software library to be placed in the top tier to a given date. It is important to realize that benchmarks are just a screenshot in the dimension of time and change rapidly in the dynamic field of machine and deep learning. Looking at the patch notes and planned features, it becomes obvious that all frameworks have the aim to grow further and adjust to shifting requirements.

Nevertheless, one should not underestimate the overall potential of benchmarks. They are essential by helping to figure out the efficacy of further development processes relative to competing frameworks. Development teams and other open source contributors can see where they need to put effort into to achieve a more efficient system. Changes between different benchmark releases might happen due to insights some developers gain through them.

None of the research-based comparisons dealt with multi-machine executions of models. Hence TensorFlow's focus is on distribution, a key feature is entirely neglected in these measurements. It is going to be interesting how TensorFlow will perform under distributed settings in contrast to its competition in the future.

IV. TensorFlow applications and use cases

Since the first release of TensorFlow, the community has grown steadily and many people are contributing to its further development. Event though it is a comparably fresh framework, a relatively high number of research papers mention having used TensorFlow for their studies. This subsection will illustrate how TensorFlow is applied in science and industry to give the reader an impression of its usefulness. Papers [1], [30]–[33] report to use TensorFlow in their research works. A few of them are mentioned in the following paragraph.

One of the more recent publications is [32]. The research team reports having successfully used machine learning methods in their investigations on Alzheimer's and Parkinson's diseases (AD and PD). Their approach is based on collecting neuroimaging data in order to run a *principal component analysis* on them. Their result is to be able to distinguish healthy patients from AD and PD patients by using TensorFlow.

A further application in the area of medical research is conducted in [33]. They used neural networks to predict whether certain drugs are able to bind a given target. In different words, the neural network classified and decided whether a molecule is effective in fulfilling a certain task. They used TensorFlow to implement their models and test their efficacy.

Lastly, an example from the agriculture is from Kussul et al. They present in [31] how they used TensorFlow to evaluate freely available remote sensing data. Constructing a deep network of four layers and feeding satellite images into it, they were able to detect different crop types and label them accordingly.

Known industrial applications are the support of AI for the Google search engine. The famous *page rank algorithm* [34] is extended by neural networks, resulting in the *rankbrain* algorithm now. According to [35], this novel approach helped the Google engineers to deliver more accurate search results with less manual fine-tuning efforts.

There are several other applications of TensorFlow spread all over the Google products. The application Google Photos uses TensorFlow to recognize faces and groups them accordingly in different folders. The Google Cloud Speech is just another service enabled by TensorFlow. This service is able to provide an API that can be used to transform recorded speech into textual form.

More important is the fact that the whole *DeepMind* team of Google switched from Torch to TensorFlow [36]. DeepMind is a big artificial intelligence team that does intense research in machine learning. Having contributed a lot to the Torch framework, the expectations are also high for future contributions to the TensorFlow environment.

The near future will reveal how TensorFlow's community will develop and how the reputation in science will evolve. The current situation seems to be for the benefit of TensorFlow. Especially the advanced visualization methods paired with the distribution factor make this framework special in the deep learning scene.

V. Conclusions

This paper illustrates how TensorFlow defines a model, optimizes it, distributes it, and executes the data flow graph. Future work is mainly based on performance tuning in different areas.

The placement algorithm, for instance, is still not entirely implemented as described here in section II. Moreover, technology is changing fast, thus, TensorFlow needs to adjust frequently to fluctuating requirements on machine learning frameworks. Thanks to its modular construction it is able to replace and tailor certain software parts comparably easily.

TensorFlow's place among other frameworks is currently special because of its ability to perform distributed tasks on heterogeneous machines and to visualize entire executions. Additionally, one also has to consider the fact that Google still stands behind the whole framework, promoting it actively and successfully which obviously will benefit its future evolution.

REFERENCES

[1] M. August and X. Ni, "Using Recurrent Neural Networks to Optimize Dynamical Decoupling for Quantum Memory," *Arxiv*, 2016. [Online]. Available: http://arxiv.org/abs/1604.00279

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, and Q. V. Le, "Large scale distributed deep networks," *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999271

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, and G. Brain, "TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning," *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 265–284, 2016. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[4] 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. Mane, 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. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[5] TensorFlow, "TensorBoard," 2015, last visited 2017-06-01. [Online]. Available: https://www.tensorflow.org/get_started/summaries_and_tensorboard

[6] ——, "TensorFlowAPI," 2015, last visited 2017-06-06.

[7] Y. Feng, "TensorFlow - GitHub Releases," 2017. [Online]. Available: https://github.com/tensorflow/tensorflow/releases?after=v1.0.0

[8] J. Sanders and E. Kandrot, *CUDA by example : an introduction to general-purpose GPU programming*. Addison-Wesley, 2011.

[9] C. C. E. Leiserson, R. R. L. Rivest, C. Stein, and T. H. Cormen, *Introduction to Algorithms, Third Edition*, 2009, vol. 7.

[10] V. W. Lee and Others, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *Isca*, vol. 38, no. 3, pp. 451–460, 2010.

[11] N. Fujimoto, "Faster matrix-vector multiplication on GeForce 8800GTX," in *IPDPS Miami 2008 - Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, Program and CD-ROM*, 2008. [Online]. Available: http://www.nvidia.ca/docs/IO/47905/fujimoto_lspp2008.pdf

[12] V. Mnih, "Cudamat: a CUDA-based matrix class for python," *Department of Computer Science, University of Toronto ...*, 2009.

[13] G. Thorson, C. Clark, and D. Luu, "Vector computation unit in a neural network processor," Nov. 24 2016, uS Patent App. 14/845,117. [Online]. Available: http://www.google.com/patents/US20160342889

[14] TensorFlow, "TensorFlow Architecture," 2015, last visited 2017-06-06.

[15] C. Click, "Global code motion/global value numbering," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 246–257, 1995.

[16] 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. B. 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. E. 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. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. 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," 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[17] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, and Y. Bengio, "Theano: Deep Learning on GPUs with Python," *Journal of Machine Learning Research*, vol. 1, pp. 1–48, 2011. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.678.1889

[18] J. Bergstra, O. Breuleux, F. F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math compiler in Python," *Proceedings of the Python for Scientific Computing Conference (SciPy)*, no. Scipy, pp. 1–7, 2010. [Online]. Available: http://www-etud.iro.umontreal.ca/ wardefar/publications/theano_scipy2010.pdf

[19] L. lab., "Drawing Theano Graphs," last visited 2017-06-06. [Online]. Available: http://deeplearning.net/software/theano/tutorial/printing_drawing.html

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," 2014. [Online]. Available: http://arxiv.org/abs/1408.5093

[21] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative Study of Deep Learning Software Frameworks," 2015. [Online]. Available: http://arxiv.org/abs/1511.06435

[22] S. Dasgupta, "Netscope," last visited 2017-06-06. [Online]. Available: http://ethereon.github.io/netscope/quickstart.html

[23] R. Collobert, S. Bengio, and J. Mariethoz, "Torch: A Modular Machine Learning Software Library," p. 7, 2002. [Online]. Available: https://infoscience.epfl.ch/record/82802/files/rr02-46.pdf

[24] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014. [Online]. Available: http://arxiv.org/abs/1404.5997

[25] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," 2013. [Online]. Available: http://arxiv.org/abs/1312.6229

[26] C. Szegedy, W. Liu, Y. Jia, and P. Sermanet, "Going deeper with convolutions," *arXiv preprint arXiv: 1409.4842*, pp. 1–9, 2014.

[27] Soumith Chintala, "convnet-benchmarks," last visited 2017-06-01. [Online]. Available: https://github.com/soumith/convnet-benchmarks

[28] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[29] 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. 2278–2323, 1998. [Online]. Available: http://ieeexplore.ieee.org/document/726791/

[30] G. Barzdins, S. Renals, and D. Gosko, "Character-Level Neural Translation for Multilingual Media Monitoring in the SUMMA Project," *Arxiv*, no. February, pp. 1789–1793, 2016. [Online]. Available: https://arxiv.org/abs/1604.01221

[31] N. Kussul, M. Lavreniuk, S. Skakun, and A. Shelestov, "Deep Learning Classification of Land Cover and Crop Types Using Remote Sensing Data," *IEEE Geoscience and Remote Sensing Letters*, vol. 14, no. 5, pp. 778–782, may 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7891032/

[32] F. Segovia, M. García-Pérez, J. Górriz, J. Ramírez, and F. Martínez-Murcia, "Assisting the diagnosis of neurodegenerative disorders using principal component analysis and tensorflow," in *Advances in Intelligent Systems and Computing*, 2017, vol. 527, pp. 43–52. [Online]. Available: http://link.springer.com/10.1007/978-3-319-47364-2_5

[33] B. Ramsundar, S. Kearnes, P. Riley, D. Webster, D. Konerding, and V. Pande, "Massively Multitask Networks for Drug Discovery," 2015. [Online]. Available: https://arxiv.org/pdf/1502.02072.pdf http://arxiv.org/abs/1502.02072

[34] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," *World Wide Web Internet And Web Information Systems*, vol. 54, no. 1999-66, pp. 1–17, 1998. [Online]. Available: http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

[35] J. Clark, "Google Turning Its Lucrative Web Search Over to AI Machines," 2015, last visited 2017-05-02. [Online]. Available: https://www.bloomberg.com/news/articles/2015-

10-26/google-turning-its-lucrative-web-search-over-to-ai-machines%0Ahttp://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-ai-machines

[36] Koray Kavukcuoglu, "Research Blog: DeepMind moves to TensorFlow," last visited 2017-06-06. [Online]. Available: https://research.googleblog.com/2016/04/deepmind-moves-to-tensorflow.html