TensorFlow Debugger: Debugging Dataflow Graphs for Machine Learning

Shanqing Cai, Eric Breck, Eric Nielsen, Michael Salib, D. Sculley
Google, Inc.
{cais, ebreck, nielsene, msalib, dsculley}@google.com

Abstract

Debuggability is important in the development of machine-learning (ML) systems. Several widely-used ML libraries, such as TensorFlow and Theano, are based on dataflow graphs. While offering important benefits such as facilitating distributed training, the dataflow graph paradigm makes the debugging of model issues more challenging compared to debugging in the more conventional procedural paradigm. In this paper, we present the design of the TensorFlow Debugger (`tfdbg`), a specialized debugger for ML models written in TensorFlow. `tfdbg` provides features to inspect runtime dataflow graphs and the state of the intermediate graph elements ("tensors"), as well as simulating stepping on the graph. We will discuss the application of this debugger in development and testing use cases.

1 Introduction

Machine learning (ML) is becoming increasingly popular in academic and industrial settings. Several popular open-source ML libraries, such as TensorFlow [1] and Theano [8], are based on dataflow graphs [2]. Graph-based computation offers great flexibility in the placement of computational operations on devices to facilitate large-scale, distributed model training and inference.

However, the dataflow graph paradigm introduces a challenge: debuggability. Diagnosing and resolving software bugs is an inescapable part of software development and productization cycles. The efficiency with which debugging activities can be performed limits the speed of development and production issue resolution. ML systems are software systems, and hence are no exceptions in this regard. Moreover, the complexity and data-driven nature of ML systems lead to less deterministic and more opaque behavior in ML systems than in most non-ML systems, which puts greater emphasis on debuggability [3, 4, 5, 6]. From our interaction with a wide variety of TensorFlow users, a specialized debugging tool is among the most frequently requested features. Developers of TensorFlow ML models often ask questions such as:

- Where in my model do problematic numerical values such as infinities arise?
- Why is my graph not pre-processing the input data as expected?
- What issues keep my model from converging during training?

Designing a debugger for TensorFlow is not trivial, with the challenge coming from the fact that the standard API encapsulates computation on the graph as a black-box function call, which not only abstracts away the graph-internal details, but also involves parallel and potentially distributed execution. A tailored debugger for ML dataflow graphs needs to organize the intermediate and internal states and present them in a clear and understandable fashion, to help users locate the source of issues such as unexpected graph structure and graph element properties, problematic numerical values, along with data input pipeline and training issues. In this paper, we present `tfdbg`, an open-source TensorFlow Debugger, specialized for debugging ML models and dataflow graphs.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

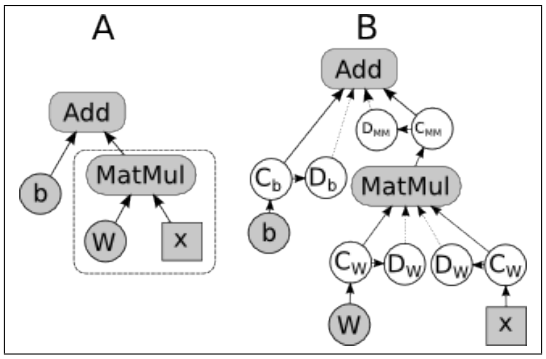

Figure 1: **A.** An example dataflow graph in TensorFlow. Nodes *Add* and *MatMul* are computation nodes. *W* and *b* are Variables. *x* is an Placeholder node. The dashed box provides an example for `tfdbg` NodeStepper's stepping through a graph. Suppose the nodes in the dashed box have been executed in a previous *continue* call, a subsequent *continue* call on the Add node need not recompute those nodes, but can use the cached tensor for the *MatMul* node. **B.** Debug nodes inserted by `tfdbg` are shown as white circles. Each node in the original graph is watched by a pair of nodes, a *Copy (C)* node and a *Debug (D)* node that exports the value copy to a configurable location.

2 Functional Design

Overview of Dataflow Graph Execution in TensorFlow. As detailed in [1], a dataflow graph in TensorFlow is a directed graph consisting of nodes and edges connecting them. Most nodes represent a computational operation, such as matrix multiplication. Inputs and outputs of the nodes, i.e., the data passed on the edges, are called *tensors*, which are typed multi-dimensional arrays. In addition to nodes representing computation, several special types of node exist, including, 1) Variables, which hold model parameters mutated during training of the model, and 2) Placeholders, which are the locations in the graph where external data can be entered (See Figure 1a).

A *session* in TensorFlow is an interface with which a graph can be executed. Using the *run* method of a session, clients can specify a node or a set of nodes to execute (called *fetches*), along with any tensor values to be fed into the graph. On receiving the *run* call, the TensorFlow runtime underlying the session rewrites the graph for performance optimization and node placement on devices and machines. Then the runtime calculates the fetched outputs in a multi-threaded fashion. In typical TensorFlow-based ML model training, the session's *run* method is invoked repeatedly to perform iterative training (e.g., mini-batch stochastic gradient descent).

The *run* method returns only the final fetch result, which makes the execution model a black box: the intermediate tensors generated during the execution are used during the run, but are not visible to the client. Standard debugging tools are not useful: Python's `pdb` cannot access information at the level of the runtime (which is written in C++), while `gdb` is not capable of presenting the information in a way relevant to TensorFlow constructs such as nodes and tensors. In addition, it is not possible for the client to control the execution process (e.g., pausing it at an intermediate state to observe and manipulate intermediate tensors). This is where `tfdbg` aims to provide visibility and controllability to the graph execution process.

2.1 Architecture of the TensorFlow Debugger

`tfdbg` consists of three main components, namely the Analyzer, the NodeStepper and the RunStepper. The Analyzer adds observability to the graph execution process, while the NodeStepper and RunStepper focus on the controllability of TensorFlow code on two different levels described below.

The Analyzer. It makes the structure and intermediate state of the runtime graph visible. The Analyzer is designed to be applicable to not only a TensorFlow runtime running on a single machine, but also a distributed runtime spanning multiple machines.

The NodeStepper. It is reminiscent of the sequential debuggers such as `gdb` [7]. Using the NodeStepper, clients can pause at a given node of the graph, inspect the current state of the graph's

nodes, and optionally make certain modifications to the tensors' states before resuming execution on the graph. The NodeStepper is currently limited to runs on a single machine.

The Analyzer and NodeStepper are not mutually exclusive and can work together, e.g., during a stepping debugging session in which the client desires visibility into intermediate tensors generated by the continuation calls.

The third module of `tfdbg`, which we call the **RunStepper**, achieves controllability on a higher level. This has to do with an interesting aspect about how model training happens in TensorFlow: breakpointing can occur at two different levels. On the lower level, one can pause at a given node in a graph, in the middle of a session *run* call. This is what the aforementioned NodeStepper implements. On a higher level, one can pause between successive *run* calls, e.g., between iterations of model training.

Debugging is usually an interactive process. A command-line interface of `tfdbg` provides human-friendly access to all three components mentioned above.

3 Detailed Design

Design of the Analyzer. The Analyzer inserts special-purpose debugging nodes into the runtime graph itself and lets those nodes export the data. This approach provides flexibility in expanding or changing the functionality of the data-exporting debug nodes and greater portability among different runtime environments.

As Figure 1b shows, to watch a node's output, a pair of *Copy* (C) and *Debug* (D) nodes are inserted between the output slot and its outgoing targets (recipients). The Copy node copies the watched tensor, to guard against updates to the tensor value by concurrent threads in a race condition. The debug ops support dumping the values of the watched tensors to the file system and sending them remotely through remote procedural calls (RPC). The file dumping mode is used for debugging on a single machine, where the total size of the intermediate tensors in a single run or across multiple runs can exceed the RAM of the machine. The RPC mode is used during distributed training, where a centralized debugging server receives the debug data and may store them to the file system for subsequent analysis. In addition to dumping the intermediate tensors, the Analyzer also exposes the runtime graphs, which generally differ from the graph set up by the client in a front-end language, e.g., Python.

The Analyzer provides options to watch a subset of the nodes on the graph, e.g., by filtering node names with a given regular-expression pattern. The Analyzer also has a limited amount of built-in intelligence to analyze common model issues, such as highlighting the backward path in a training graph and determining the source node of bad numerical values.

Design of the NodeStepper. The primary functionality provided by the NodeStepper is to let the client execute part of the transitive closure of a fetched node, thereby turning a session's *run* call from a monolithic operation into an incremental and controllable process. We refer to this incremental action as the *continue* method. While performing the incremental *continue* calls on the graph, there are three types of states that the NodeStepper needs to keep track of.

- What previous *continue* calls have occurred: the stepper should cache outputs from the previous *continue* calls and use them in subsequent *continue* calls (dashed box in Figure 1a).
- Whether previous *continue* calls have mutated states of relevant Variable nodes. Such Variable value updates should have invalidated the cached tensors that receive inputs from the variables transitively. In addition, the NodeStepper provides options to restore the previous values of the Variables between *continue* calls.
- The NodeStepper allows users to override the value of any intermediate tensor. Such value overriding can be useful when debugging issues at the node level.

Based on the *continue* method, the stepper can achieve other modes of incremental debugging. For example, by topologically sorting the nodes in the transitive closure of the fetched nodes, the NodeStepper can actually step through the nodes in a linear fashion. Based on this stepping, it can break at a given node, as specified by the node's name or type, or when a certain predicate on the

output tensor value is satisfied. This can be regarded as conditional breakpoints at the node level. Time-travel debugging (i.e., re-executing to already-executed nodes or jumping ahead to nodes whose input nodes have not been executed yet) is also possible, as the NodeStepper is not constrained by the actual node execution order in a non-debugging session and can set up any nodes' inputs at will.

Design of the RunStepper. While the Analyzer operates on the level of graph nodes, the *RunStepper* operates at a higher level, i.e., the level of the Session *run* calls. The RunStepper provides the ability to pause between runs, based on any specified predicates on the run history. The simplest mode is to pause at every *run*. Unlike general debuggers such as `pdb`, the RunStepper can access graph-internal states such as intermediate tensor values and hence is capable of more sophisticated debugging modes. For example, it can break conditionally at *run* calls when any intermediate tensor in the graph satisfies a certain configurable predicate, such as containing infinities or NaNs. A more powerful type of predicate examines the history of the graph state across previous runs. For example, the *RunStepper* can be configured to pause when a node in the graph outputting a loss function value starts to show an upward trend.

4 Use Cases

White-box Testing of ML Models. In an ML system under active development, feature development and code refactoring can sometimes lead to unforeseen changes in the structure and behavior of an ML model. Such changes can also arise as a result of changes in the underlying ML library itself. If left untested, these low-level changes can lead to subtle issues in production that are difficult to observe and debug. The above-described *Analyzer* module of `tfdbg` makes it possible to make assertions about a TensorFlow model in unit tests.

Two types of assertions can be made based on `tfdbg`'s Analyzer: 1) structural assertions: the structure of a TensorFlow graph, the nodes and their attributes 2) functional assertions: the intermediate tensor values in the graph under a given set of inputs. Structural and functional assertions should focus on the critical parts of the model, such as output of a neural-network layer or an embedding lookup result, and ignore noncritical parts, in order to avoid being sensitive to unimportant changes caused by refactoring or library changes.

Debugging Problematic Numerical Values. A type of frequently encountered problem in TensorFlow ML model training is bad numerical values, e.g., infinities and NaNs, which arise due to various reasons such as numerical overflow and underflow, logarithm of and division by zero. In a large ML model with thousands of nodes, it can be hard to find the node at which this first emerged and started propagating through the graph. With `tfdbg`, the user can specify a breaking predicate in the RunStepper to let runs break when any intermediate tensors in the model first show infinities or NaNs and drop into the Analyzer UI to identify the first-offending node. By examining the type of node and its inputs using the Analyzer UI, the user can obtain useful information about why these values occur, which often leads to a fix to the issue such as applying value clipping to the problematic node.

5 Conclusion and Future Directions

The TensorFlow Debugger (`tfdbg`) provides visibility and controllability into the execution of models written in TensorFlow, an ML library based on dataflow graphs. It enhances the unit-testability and debuggability of TensorFlow ML models. In the future, we plan to implement additional features in `tfdbg`, including integration with TensorBoard [1] for visual debugging and improved support for the distributed sessions of TensorFlow, including replaying of multi-machine graph execution.

Open source. `tfdbg` is a part of open-source TensorFlow. Its README file is at: `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/debug/examples`.

6 Acknowledgments

This work would not be possible without advice and support from Yuan Yu, Michael Isard, Sherry Moore, Josh Levenberg and other members of the TensorFlow team.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Arvind and David E. Culler. Dataflow architectures. *Annual Review of Computer Science*, 1:225–253, 1986.

[3] Roger B Grosse and David K Duvenaud. Testing mcmc code. *arXiv preprint arXiv:1412.5218*, 2014.

[4] Richard Mann. Rethinking retractions. `http://prawnsandprobability.blogspot.co.uk/2013/03/rethinking-retractions.html`, 2013.

[5] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?": Explaining the predictions of any classifier. *arXiv preprint arXiv:1602.04938*, 2016.

[6] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[7] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002.

[8] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.