A Computational Model for TensorFlow

An Introduction

Martín Abadi Michael Isard Derek G. Murray

Google Brain

Abstract

TensorFlow is a powerful, programmable system for machine learning. This paper aims to provide the basics of a conceptual framework for understanding the behavior of TensorFlow models during training and inference: it describes an operational semantics, of the kind common in the literature on programming languages. More broadly, the paper suggests that a programming-language perspective is fruitful in designing and in explaining systems such as TensorFlow.

CCS Concepts • **Theory of computation → Operational semantics**; • **Computing methodologies → Neural networks**; • **Software and its engineering → Data flow architectures**

Keywords Machine learning, TensorFlow

1. TensorFlow as a Programming Language

The TensorFlow system for machine learning [1, 2] largely owes its flexibility and generality to its programmability. TensorFlow models are assembled from primitive operations by function composition and other constructs familiar from programming languages. This approach supports a wide variety of machine learning applications, including training and inference with deep neural networks, on heterogeneous distributed systems. Aspects of the implementation of TensorFlow, for example its rewriting optimizations, also have roots in programming languages. It is therefore reasonable and fruitful to think of TensorFlow in programming-language terms.

At its core, TensorFlow relies on dataflow graphs with mutable state. This paper describes a semantics for these graphs. The semantics is an operational semantics of the kind common in the literature on programming languages. It is presented in English, for the sake of readability, but it is as detailed as a logical specification; indeed, it is largely a paraphrase of a logical formulation previously written in TLA [14]. It belongs in a long line of work on the semantics of dataflow systems (e.g., [12]). Mathematically, it is rather simple and elementary. The literature on dataflow systems includes much deeper and harder results, in particular connections between operational semantics and denotational semantics (e.g., [10, 11]).

The main goal of the semantics is to provide a conceptual framework for execution, as a starting point for thinking about the behavior of TensorFlow models. Thus, the semantics does not aim to account for implementation choices; it aims to say what outputs may be produced, not to say exactly how. A framework of this kind can sometimes be valuable to users as they develop their models. Indeed, some of our internal users have suggested that an operational semantics would help them in their work. A semantics can also provide guidance in the development of TensorFlow. Ongoing work on state encapsulation, mentioned below, illustrates this point. Finally, a semantics is essential in assessing the correctness of implementation decisions. The rewriting optimizations mentioned above constitute one class of examples. In our past work on timely dataflow [16], we found that having a semantics [3] was crucial for the design of correct techniques for fault-tolerance [4]; a semantics could play an analogous role in future work on TensorFlow.

We focus on central features of TensorFlow, and their default behavior in TensorFlow 1.0, but omit many details. In some cases, those details are straightforward. In others, they appear hard to model cleanly and simply, represent sources of possible confusion for users (see, for example, [9]), and may be best addressed through design improvements. We also omit other programming-language aspects of TensorFlow, such as the specifics of front-ends for defining graphs and the facilities for control flow.

The next section reviews the relevant aspects of TensorFlow. Section 3, which is the core of this paper, defines the semantics. Section 4 discusses further work, touching on other programming-language aspects of TensorFlow.

2. TensorFlow Review

TensorFlow represents computations by dataflow graphs. Although focused on machine-learning applications, TensorFlow is rather agnostic on the exact purpose of the computations. In particular, a computation may perform one or more steps of training for a machine-learning model, or it may be the application of a trained model. Thus, dataflow graphs support both training and inference.

A dataflow graph consists of nodes and edges, where each node represents an instantiation of an operation, and values flow along the edges. The operations are implemented by kernels that can be run on particular types of devices (for instance, CPUs or GPUs).

The main values of interest are tensors: arrays of arbitrary dimensionality where the underlying element type is specified or inferred at graph-construction time. Accordingly, many of the operations are mathematical functions such as matrix multiplication.

In addition, some of the operations may read or update state. In TensorFlow, a variable is a special kind of operation that returns a handle to a tensor. In this case, informally, we may say that the tensor is held in the variable, and we may conflate the variable operation and the resulting handle. Such a handle can be passed as argument to operations that read or update the corresponding tensor. For example, the tensor may contain the weights of a layer in a neural network, which are updated during the training process.

MAPL'17, June 18, 2017, Barcelona, Spain
ACM. 978-1-4503-5071-6/17/06...$15.00
http://dx.doi.org/10.1145/3088525.3088527

Moreover, in addition to edges for communicating tensors, a graph may include control edges that constrain the order of execution. This order can affect the observable semantics in the presence of mutable state, and it can also affect performance.

A client typically constructs a graph using a front-end language such as Python. Then the client can make a call to run the graph, specifying which inputs to "feed" and which outputs to "fetch". TensorFlow propagates the input values, repeatedly applying the operations prescribed by the graph, until no more nodes can fire. The order in which nodes fire is constrained by data dependencies and control edges, but is not necessarily unique. The execution ends with values on the graph's output edges. In these respects, TensorFlow graphs are similar to expressions in programming languages.

Often, a graph is executed multiple times. Most tensors do not survive past a single execution of the graph. However, mutable state does persist across executions. In a typical application, a graph represents a step of training for a machine-learning model, the parameters of the model are stored in tensors held in variables, and they are updated as part of running the graph.

3. A Core Computational Model

In this section we define the semantics described in Section 1. Some parts of its presentation aim to be quite detailed, and paraphrase logical formulas (originally written in TLA); accordingly, they may be a little dry. We also give small, self-contained examples.

3.1 Programs

As explained above, TensorFlow graphs consist of nodes and edges, where each node represents an instantiation of an operation, and values flow along the edges. So, as a starting point for our model, we first define the values, the operations, and the graphs of interest. We refer to them with names such as "TF values" and "TF operations", thus distinguishing them from similar concepts in the full TensorFlow and elsewhere.

3.1.1 Definitions

TF values include tensors, as expected, but also auxiliary values. Specifically, we assume:

- a set of values Tensors, to which we may refer as tensors;
- a set of variables Vars, which correspond to the handles discussed in Section 2;
- a constant GO that we use as a trigger; and
- a constant EMPTY that we use to indicate not-yet-produced or already-consumed data.

These are all disjoint.

Correspondingly, we distinguish three kinds of edges:

- tensor edges, which are used for conveying elements of Tensors;
- variable edges, which are used for conveying elements of Vars; and
- control edges, which are used only for GO signals.

Operations are of several kinds, too. A TF operation is one of:

- f for f a function in Tensors$^k \to$ Tensors^l, for some non-negative integers k and l;
- Var(x) for x in Vars;
- Read; and
- Assign-f for f a function in (Tensors $\times$ Tensors) $\to$ Tensors.

Next we describe the intended semantics of these operations, briefly and informally; a more detailed semantics is below.

- When f is a function in Tensors$^k \to$Tensors^l, the operation f simply applies the function to the operation's k inputs and returns its l results.
- When x is a variable (an element of Vars), the operation Var(x) simply outputs x.
- The operation Read outputs the current value of a variable; this variable is an input to the operation.
- Finally, when f is a function in (Tensors $\times$ Tensors) $\to$ Tensors, the operation Assign-f has as inputs a variable x and a tensor v; it reads the current value of x, applies f to this value and to v, and updates x to hold this result.[1]

As in this explanation, we generally ignore the possibility that operations may fail to terminate or may produce errors, for simplicity. Similarly, we require each operation to be deterministic.[2] TensorFlow is more general in these respects. We also omit control-flow constructs, which we discuss briefly in Section 4.

Other operations may be added, and indeed some are easy to define from the ones here. For example, an ordinary Write operation is a special case of Assign-f where f is the function Snd such that Snd(x,y) = y. In examples, we write Write as an abbreviation for Assign-Snd.

A TF program consists of a directed acyclic graph G, plus a mapping (a "labelling") L from nodes of G to TF operations. The labelling L must satisfy the following arity constraints for the tensor edges and variable edges:

- If L(n) is a function f in Tensors$^k \to$ Tensors^l then n has k incoming tensor edges (one for each argument of f) and l outgoing tensor edges (one for each result of f).

  We assume that the edges are ordered, so that the order of the arguments and the results of f is unambiguous.

  If several nodes downstream need to consume one of the results, the desired sharing can be implemented with explicit, additional Copy nodes, where Copy is the obvious function of type Tensors $\to$ (Tensors $\times$ Tensors). (Alternatively, the sharing could rely on connecting multiple outgoing edges to a single output "port"; the resulting definitions would be more complicated, though not particularly difficult.)
- If L(n) = Var(x) then n has one outgoing variable edge.
- If L(n) = Read then n has one incoming variable edge and one outgoing tensor edge.
- If L(n) = Assign-f then n has one incoming variable edge and one incoming tensor edge.

There are no other tensor or variable edges beyond those just indicated, but there may be control edges. We allow some edges to have no source or no destination, to model external inputs and outputs; we may think of them as the edges for "feeding" and "fetching", in TensorFlow parlance. We call them input and output edges. We require each node to have at least one incoming edge (possibly simply a control edge), for triggering execution; this

[1] The corresponding operation in the TensorFlow API also returns x, but this output is unimportant, so we omit it.

[2] In this and later footnotes, we indicate how to relax these restrictions.
As a first step in this direction, we may let Tensors include not only ordinary tensors but also distinct elements that represent various errors that may be detected in the course of computation, for example the result of reading a variable that has not been properly initialized or the result of attempting to add two tensors of incompatible shapes.
In the definitions of operations f and Assign-f, we could relax the requirements on f. For example, in order to model non-determinism or non-termination, we may allow f to be a relation or a partial function, respectively.

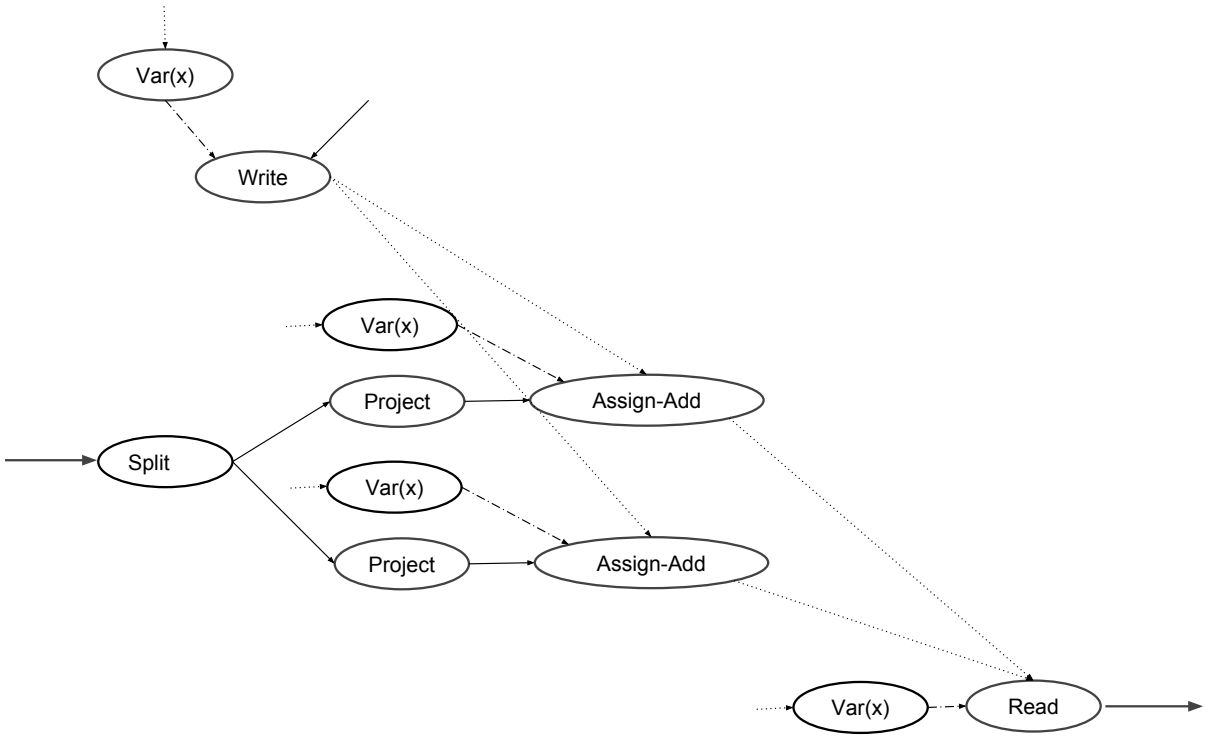
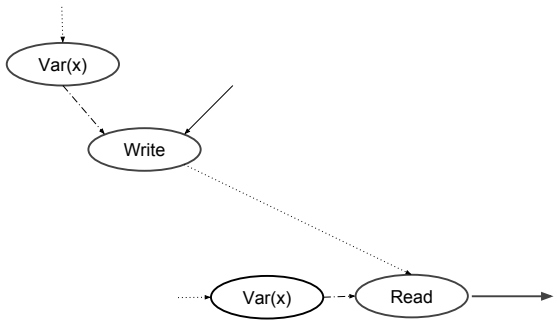


Figure 1. An example program

requirement makes it easy to ensure that each node fires exactly once (see Section 3.2).

3.1.2 Example Programs

The following is a first, easy example of a TF program:



Here, solid arrows represent tensor edges; dashed arrows represent variable edges; and dotted arrows represent control edges. While two nodes bear the label Var(x), both refer to the same underlying variable x. An input tensor edge provides a value for the variable x, and an output tensor edge transmits its final value. Thus, the graph consists of two subgraphs, one for setting x and the other for reading it. A control edge connects the node labelled Write to the node labelled Read, constraining the execution order.

A second, more complex example, in Figure 1, uses the Assign-Add operation, and also some functions from tensors to tensors, named Split and Project, whose details are unimportant. This example enriches the previous one with two subgraphs that look like replicas and that read and update the variable x, combining it with an external input.

In these examples, the arrows from the Var(x) nodes to other nodes indicate that these nodes consume the variable x (intuitively, as a handle or LValue). On the other hand, the flow of information from assignments (Write or Assign-Add) to the contents of x is not represented by an arrow. In this respect, TensorFlow differs from many traditional dataflow models. More generally, not all flows of information are apparent from the arrows in the graph. Thus, in the second example, one Assign-Add operation will "see" the value of x written by the other, although there are no paths between the two Assign-Add nodes.

3.2 Behaviors

Intuitively, a TF program starts with non-EMPTY input edges, consumes the values on those edges, and repeatedly propagates them, applying operations, until no more nodes can fire. In the course of such an execution, each node fires exactly once, and the execution ends with non-EMPTY output edges.[3] The order in which nodes fire is not necessarily unique, as mentioned above. Furthermore, the TF program executes in the context of an assignment of values to variables; these values can be read during the execution and can be updated repeatedly during the execution, under program control. In general, the resulting behavior may be non-deterministic. It is sometimes delicate but important to understand which behaviors are possible, and how much determinism, if any, is guaranteed for a given program. Note, however, that determinism is not always expected or desired. In particular, lock-free stochastic gra-

[3] Once control-flow constructs and other advanced features are added, it is no longer true that each node fires exactly once. We mention how to relax this property in Section 4.

dient descent is a common source of intentional race conditions (e.g., [2, 17]).

The following definitions explain this semantics precisely, but without the complications of an actual implementation.

3.2.1 Definitions

The semantics maps each program to a set of behaviors, where each behavior is a sequence of states. Intuitively, these are the behaviors that the program may produce.

A state consists of:

- a mapping VarValue from Vars to Tensors (intuitively, giving the values of all the variables), and
- a mapping InTransit from the edges of the TF graph to contents of the appropriate kind: an element of Tensors or EMPTY for tensor edges, an element of Var or EMPTY for variable edges, and GO or EMPTY for control edges (intuitively, indicating what each edge holds).

The initial state of each behavior must have these properties:

- for all input edges, InTransit(e) is not EMPTY,
- for all other edges e, InTransit(e) is EMPTY,
- for each variable x, VarValue(x) is in Tensors.[4]

Subsequently, each change of state in the behavior is caused by the execution (i.e., the firing) of exactly one node in the graph. A condition for whether a node n can cause a change from a state s to a state s' is that for all its incoming control edges d, InTransit(d) = GO in s and InTransit(d) = EMPTY in s', and for all its outgoing control edges e, InTransit(e) = GO in s'. Moreover, InTransit(d) must be the same in s and in s' for all edges d not incident on n, and VarValue(x) must be the same in s and in s' for all variables x, except in the case where L(n) = Assign-f for some function f. Additional conditions depend on n's label:

- If L(n) = f for f a function in Tensors^k $\rightarrow$ Tensors^l, then n's inputs must be some tensors $v_1, \ldots, v_k$, and n consumes those inputs (so the incoming edges are EMPTY afterwards), and produces the result of applying f to those tensors on its outgoing edges.[5]

  Formally, this means that if n has incoming tensor edges $d_1, \ldots, d_k$ and outgoing tensor edges $e_1, \ldots, e_l$ then

  1. for all d_i, InTransit(d_i) = v_i for some tensor v_i in s,
  2. for all d_i, InTransit(d_i) = EMPTY in s',
  3. for all e_j, InTransit(e_j) is the jth result of applying f to the values v_i in s'.
- If L(n) = Var(x), then n simply outputs the variable x.

  Formally, this means that InTransit(e) = x in s', where e is n's outgoing variable edge.
- If L(n) = Read, then n's input must be some variable x, and it consumes this input (so the incoming edge is EMPTY afterwards) and produces the current value of x as output.

  Formally, this means that, if d is n's incoming variable edge and e is its outgoing tensor edge, then

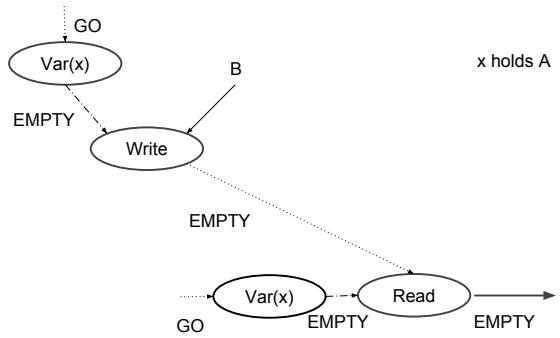
  1. InTransit(d) = x for some x in s,
  2. InTransit(d) = EMPTY in s',
  3. InTransit(e) = VarValue(x) in s'.
- If L(n) = Assign-f, then n's inputs must be some variable x and some tensor v, and it consumes these inputs (so the incoming edges are EMPTY afterwards), and updates the value of x to be the result of applying f to the current value of x and to v.

  Formally, this means that, if d_1 is n's incoming variable edge and d_2 is its incoming tensor edge, then

  1. InTransit(d_1) = x for some x in s,
  2. InTransit(d_1) = EMPTY in s',
  3. InTransit(d_2) = v for some tensor v in s,
  4. InTransit(d_2) = EMPTY in s',
  5. VarValue(x) = f(w,v) in s', where w = VarValue(x) in s, and
  6. VarValue(y) is the same in s and in s' for all other y.

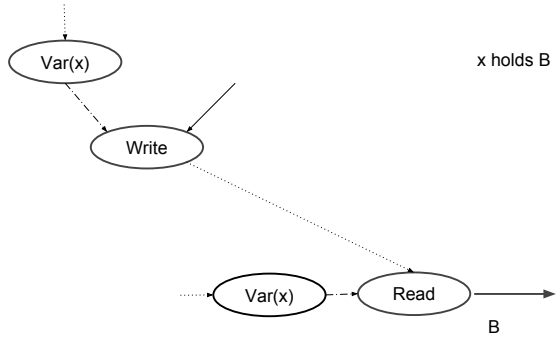
3.2.2 Example Behaviors

The following diagram illustrates a possible initial state of our first example, with a tensor A as the initial value of x (as noted on the side), and a tensor B in transit on the input tensor edge:[6]



(From now on, we omit the annotation EMPTY, in order to lighten the figures.)

From this initial state, a few steps yield a final state where B is produced as output on the output tensor edge, and no further progress is possible:



Several different sequences of steps lead to this final state. The sequences differ on when the node Var(x) at the bottom fires, but

[4] This definition may suggest that all variables are initialized. However, the set Tensors may contain distinguished elements to represent errors, including in particular an element that means "not properly initialized".

[5] In order to account for partiality, non-determinism, and probabilities, this definition and that for the case of Assign-f may be generalized. In particular, the semantics ought to ensure that errors propagate, but that Assign-f nodes do not have side effects when any of their inputs represent errors. The propagation of errors implies that errors are terminal, in the sense that they are not caught.

[6] In particular, A may be a special element of Tensor that means "not properly initialized". In this case, writing B is the actual initialization.

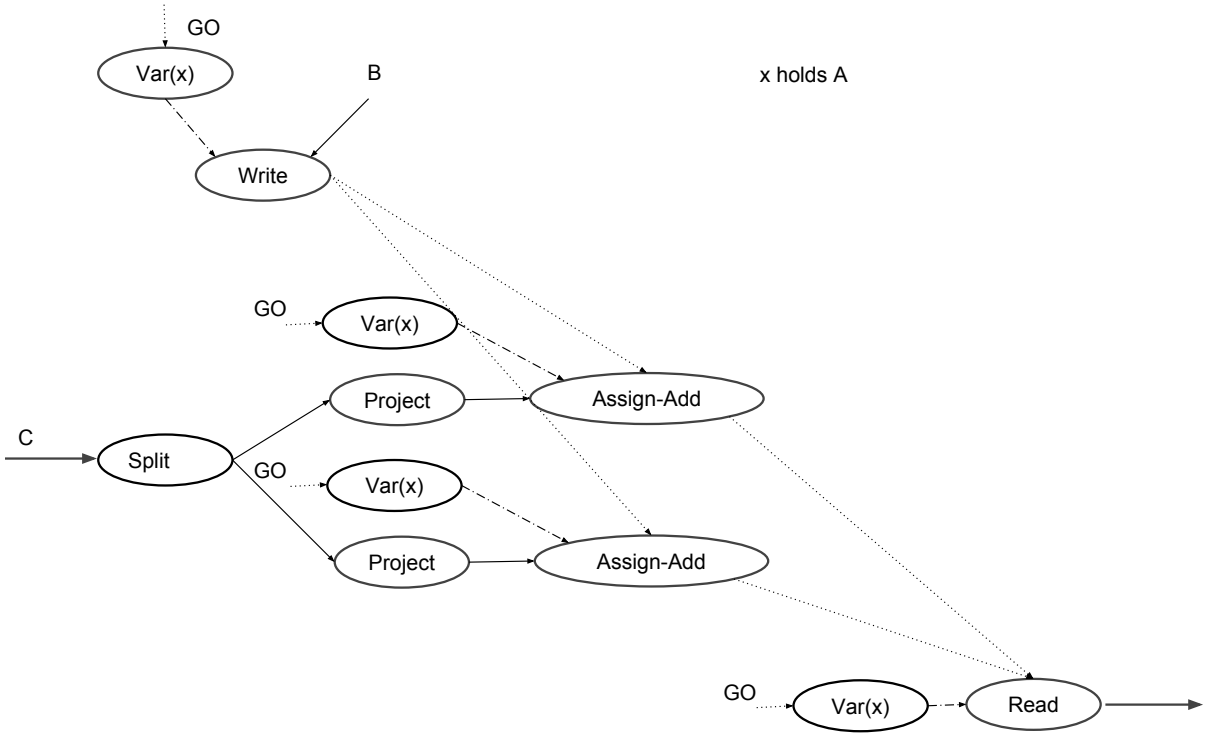


Figure 2. An example of an initial state

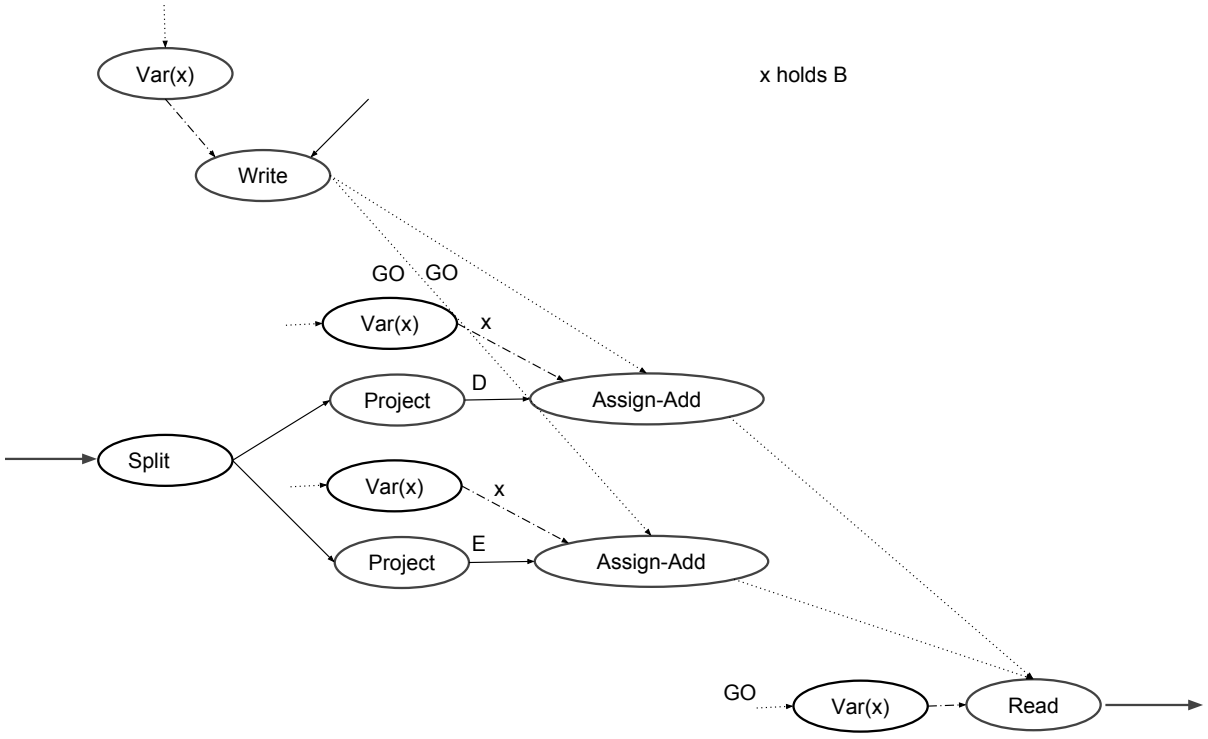


Figure 3. An example of an intermediate state

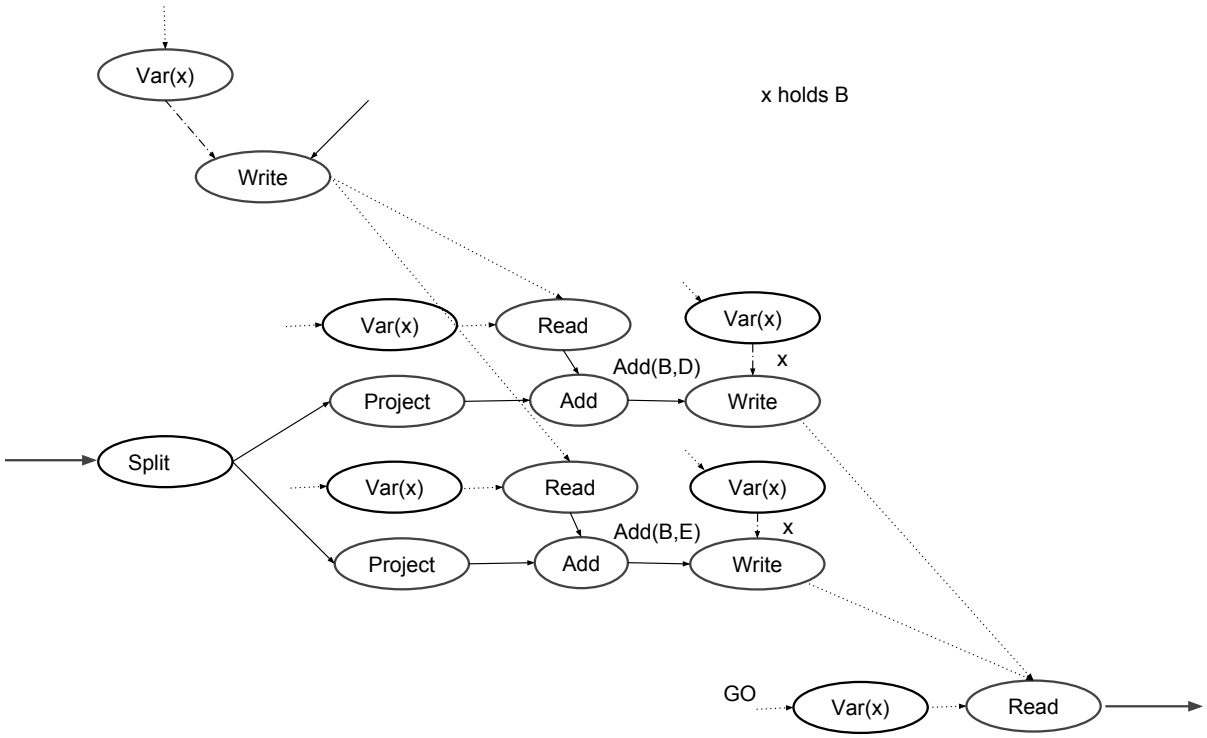


Figure 4. Another example of an intermediate state

yield the same final output B. On the other hand, if we had forgotten the control edge between the two subgraphs, the non-determinism would become visible, potentially: the final output could be A.[7]

The example program of Figure 1 can exhibit even more non-determinism. A behavior may start with the initial state depicted in Figure 2. After a few steps, the two Assign-Add nodes can fire in either order, as shown in Figure 3. With one order, the final output is Add(Add(B,D),E), where D and E are the tensors obtained by splitting C and projecting the results of this split. With the other order, the final output is Add(Add(B,E),D). The algebraic properties of Add should guarantee that these two outputs are the same. However, if Assign-Add was replaced with a different operation, such as Assign-Concat (where Concat is a function that concatenates its inputs), the non-determinism could lead to different final outputs.

It is also instructive to consider what happens if we replace each node Assign-Add with a little graph that comprises separate Read, Add, and Write nodes. In this case, the non-determinism leads to the possibility of interleavings where the two Add operations happen before either Write operation, yielding a state where two different values are in transit simultaneously towards the two Write nodes, respectively, as shown in Figure 4. The second Write operation will overwrite the effect of the first, which will thus be lost.

This example illustrates an important difference between a node Assign-Add and the combination of Read, Add, and Write nodes. More generally, the semantics says that Assign-f nodes are atomic, thus preventing certain race conditions. (Race conditions may also be avoided by other means, in particular by the explicit use of mutexes, which can be implemented in terms of other primitives.) In this respect, the semantics corresponds to reality, but only loosely. TensorFlow does not offer strong guarantees with respect to atomicity; it prevents some race conditions between updates, but allows others between reads and updates.

[7] In fact, if the control edge is omitted from the original graph, TensorFlow can "optimize away" the subgraph that sets x, and then the final output is definitely A. TensorFlow decides what parts of a graph to execute by working backwards from the values being "fetched". The semantics of this paper applies after the graph-pruning process.

4. Further Work

The semantics defined in this paper illustrates that ideas and techniques common in the programming-languages literature may be relevant to TensorFlow, and probably to other machine-learning systems, more broadly. (Conversely, machine learning appears increasingly relevant to programming languages—but that should be the subject of other papers.) We conclude by discussing a few other programming-language aspects of TensorFlow.

While the definitions above cover what we consider the core of TensorFlow, the full TensorFlow includes additional constructs, in some cases based on programming languages. For example, TensorFlow supports a richer system of types and tensor shapes than our core fragment; this is an area in which programming languages are obviously pertinent, and should inform further design. In addition, TensorFlow supports control flow through constructs such as conditionals and while loops, analogous but not identical to those of Theano [5, 8]. These constructs give rise to cyclic graphs. In the TensorFlow implementation, they are mapped to dataflow primitives, as in tagged dataflow architectures [6, 7]. Our semantics can be generalized to accommodate those primitives. In this generalization, it is no longer true that each node fires exactly once. However, each node fires at most once in each "execution context", where an "execution context" identifies an iteration of each (possibly nested) loop that contains the node.

Several front-end languages can be employed for constructing graphs, for setting up training loops, and more. In particular, "imperative mode" is an extension of TensorFlow that blurs the line between graph construction and graph execution [13]. TensorFlow Fold is a library that supports working with dynamic graphs; its design draws on techniques from functional programming such as parser combinators [15]. Further development of such languages and facilities seems worthwhile.

An important advantage of the TensorFlow graph representation is that the user offloads a large amount of well-defined computation to the runtime in a single invocation. Therefore, the system has an opportunity to optimize the code aggressively, and—since many computations are invoked repeatedly over a long time period—the system may profitably resort to expensive optimizations. The current program transformations include dead code elimination, common subexpression eliminination, constant folding, and fusion of consecutive operations. XLA, a recent domain-specific compiler for linear algebra [18], can perform such target-independent transformations and also maps programs in an intermediate representation to code for CPUs, GPUs, and other kinds of devices.

As the examples of Section 3.2.2 indicate, the use of variables is an area in which the TensorFlow design may benefit from refinement. In particular, we may reconsider the support for synchronization for computations that share state. We may also desire stronger encapsulation for state, with fewer possibilities of unintended sharing, thus avoiding errors and enabling more optimizations. Some current work goes in this direction, and is reflected in experimental features released in TensorFlow 1.0. Formal specifications (written with the same operational approach as the one of this paper, but much more detailed) played a role at the start of that work; we may revisit them in the future and use them as the basis for formal reasoning.

Acknowledgments

TensorFlow is joint work with many people in the Google Brain team and elsewhere. This paper benefits from their frequent explanations. In addition, Rajkishore Barik, Keith Bonawitz, Jeff Dean, Wolfgang Grieskamp, Peter Hawkins, Łukasz Kaiser, Dandelion Mané, Rajat Monga, Ramesh Viswanathan, and Yuan Yu made useful comments on drafts of this paper.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] 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, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, Proceedings*, pages 265–283, 2016.

[3] M. Abadi and M. Isard. Timely dataflow: A model. In *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Proceedings*, pages 131–145, 2015.

[4] M. Abadi and M. Isard. Timely rollback: Specification and verification. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods – 7th International Symposium, Proceedings*, pages 19–34. Springer, 2015.

[5] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, 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. Côté, M. Côté, A. C. 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. J. Goodfellow, M. Graham, Ç. Gülçehre, P. Hamel, I. Harlouchet, J. 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. Lefrançois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P. Manzagol, O. Mastropietro, R. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. J. 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. P. 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. *CoRR*, abs/1605.02688, 2016.

[6] Arvind and D. E. Culler. Dataflow architectures. In J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, editors, *Annual Review of Computer Science Vol. 1, 1986*, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[7] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, Mar. 1990.

[8] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio. Theano: new features and speed improvements. *CoRR*, abs/1211.5590, 2012.

[9] M. Giraud, D. G. Murray, and P. Tucker. control_dependencies and assign new shape not working (using validate_shape=false). Discussion at `https://github.com/tensorflow/tensorflow/issues/7782`, 2017.

[10] T. T. Hildebrandt, P. Panangaden, and G. Winskel. A relational model of non-deterministic dataflow. *Mathematical Structures in Computer Science*, 14(5):613–649, 2004.

[11] B. Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, 7(4):197–212, 1994.

[12] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[13] M. Kudlur. Imperative programming in TensorFlow. Code repository at `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/imperative`, 2017.

[14] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[15] M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig. Deep learning with dynamic computation graphs. *CoRR*, abs/1702.02181, 2017.

[16] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Communications of the ACM*, 59(10):75–83, 2016.

[17] B. Recht, C. Ré, S. J. Wright, and F. Niu. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *25th Annual Conference on Neural Information Processing Systems, Proceedings*, pages 693–701, 2011.

[18] The XLA Team. XLA – TensorFlow compiled. Post in the Google Developers Blog, at `https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html`, 2017.