

Recursive Function Definitions in Static Dataflow Graphs and their Implementation in TensorFlow

Kelly Kostopoulou
Columbia University
kelkost@cs.columbia.edu

Angelos Charalambidis
Harokopio University of Athens
acharal@hua.gr

Panos Rondogiannis
University of Athens
prondo@di.uoa.gr

Abstract

Modern machine learning systems represent their computations as dataflow graphs. The increasingly complex neural network architectures crave for more powerful yet efficient programming abstractions. In this paper we propose an efficient technique for supporting recursive function definitions in dataflow-based systems such as TensorFlow. The proposed approach transforms the given recursive definitions into a static dataflow graph that is enriched with two simple yet powerful dataflow operations. Since static graphs do not change during execution, they can be easily partitioned and executed efficiently in distributed and heterogeneous environments. The proposed technique makes heavy use of the idea of *tagging*, which was one of the cornerstones of dataflow systems since their inception. We demonstrate that our technique is compatible with the idea of *automatic differentiation*, a notion that is crucial for dataflow systems that focus on deep learning applications. We describe the principles of an actual implementation of the technique in the TensorFlow framework, and present experimental results that demonstrate that the use of tagging is of paramount importance for developing efficient high-level abstractions for modern dataflow systems.

CCS Concepts • **Software and its engineering** → **Data flow languages; Recursion; Distributed programming languages; Control structures**; • **Computing methodologies** → Neural networks;

Keywords Tagged dataflow, recursive functions, TensorFlow

1 Introduction

The interest in the dataflow model has been recently renewed by the latest developments in scalable machine learning systems. Modern dataflow systems express computations as dataflow graphs of processing nodes that can work independently and in parallel. In such a formalism, the data dependencies between the nodes are explicit and consequently individual nodes can be distributed in multiple machines and hardware accelerators. These characteristics render the dataflow model highly desirable for expressing scalable machine learning tasks.

The popularity of deep learning and its recent successes in solving challenging problems in various domains, lead to proposals of more complex neural network architectures.

The Recurrent Neural Networks (RNNs) [13] is a prominent example that uses an elaborate architecture in order to exhibit temporal dynamic behavior. RNNs benefit from the ability of the machine learning framework to efficiently implement control-flow decisions, such as conditionals and iteration, in terms of dataflow graphs. However, more complex neural network architectures require dataflow systems that can support more demanding control-flow, such as for example recursion.

1.1 Implementations of Control Flow

Current dataflow frameworks implement iteration and recursion using two main approaches. In the simpler approach, usually coined as the “out-of-graph” approach [20, 25], the program is expressed in a client programming language such as Python and the framework exploits the control-flow support of the host language. Even though this approach admits a relatively simple implementation, it requires a continuous interaction between the client and the dataflow system, resulting in reduced performance. Moreover, in this approach the dataflow graphs are usually small parts of the whole program, and the optimizations that can be applied are limited. In order to remedy these limitations, some machine learning frameworks employ the “in-graph” approach in which the control-flow operations are expressed directly in the dataflow graph. In this approach the dataflow execution engine is responsible for implementing the control-flow decisions.

There are two distinct proposals to implement control-flow constructs in the “in-graph” approach. The first, relatively straightforward, way suggests that the dataflow graph is allowed to change at runtime. In this approach the control-flow operations are represented as nodes in the dataflow graph and when executed they effectively replace themselves with new nodes of the graph. In this way one can simulate a dynamic unfolding of the graph. The second, more elaborate way is to assume that the dataflow graph is fixed, ie., it can not change at runtime. As it turns out, this *fixed-graph* approach is preferred by the established deep learning frameworks such as TensorFlow [29]. There are good reasons for choosing the fixed-graph idea. For example, an immediate consequence of knowing the graph ahead of time is that aggressive optimizations and distributed device placement algorithms can be performed at compile time. Therefore, it is highly desirable, whenever possible, to express computations as a fixed dataflow graph.

However, it is not always apparent how iteration or recursion can be expressed in a dataflow graph that is not allowed to expand at runtime. The main concern that immediately arises is that the nodes of the fixed graph must be reactivated with different inputs and in different points in time during the execution. In order to not mix different reactivations of the nodes, the simple dataflow model has to be extended with the notion of *tags*, i.e., labels that travel with the data and essentially indicate the *context* that the data should be evaluated under. Additionally, these tagged dataflow graphs must include “special” operators that can appropriately manipulate such tags. TensorFlow [1] implements iteration using the aforementioned tagging technique [29]. However, when it comes to recursion, TensorFlow resorts to using expandable dataflow graphs, a choice which diminishes the advantages of the fixed-graph approach.

1.2 Contributions

In this paper we consider the problem of efficiently implementing recursive function definitions using dataflow graphs that remain fixed at runtime. We assume that the initial program is a set of dataflow graphs each representing a different function and we allow nodes that correspond to a defined function to occur as nodes in the graphs. We base our work on the *tagged dataflow* model [3, 4, 26]. The initial program is transformed into a single graph by replacing the operators with the graphs that represent functions. We also use tags to distinguish between different function calls. The tags, in our proposal, are lists of labels and each label indicates a different call of a function. We demonstrate that our proposal can relatively easily be embedded in TensorFlow’s runtime, and we demonstrate that the resulting system is efficient and appropriate for deep learning applications. The contributions of the paper can be summarized as follows:

- We devise a purely tagged approach for implementing recursive functions in a dataflow environment. Our approach transforms a set of dynamic (i.e., expandable) dataflow graphs, into a single static (i.e., non-expandable) graph which uses two simple (yet powerful) dataflow operations. As a result, our technique ensures that dataflow graphs remain fixed during runtime. The conceptual origins of our work can be traced back to techniques that were developed in the 80s, namely [3, 4, 12, 21, 26, 28], but our approach has distinguishing characteristics from each one of them. A detailed comparison with the aforementioned techniques is given in Section 7.
- We demonstrate that our technique is compatible with the idea of *automatic differentiation* [6], a notion that is crucial for dataflow systems that focus on deep learning applications. In particular we demonstrate how we can efficiently support automatic differentiation in recursive dataflow graphs that have been obtained using the proposed tagging approach.

- We describe an actual implementation of our approach under the TensorFlow framework. In particular, we discuss the extensions that are required to the computational model of TensorFlow in order to execute dataflow graphs that contain the new dataflow operators required for handling recursion. The most demanding changes to the computational model concern the distributed execution of these static dataflow graphs.
- We present experimental evidence which confirms that the tagged execution outperforms the dynamic (i.e., graph expanding) execution of recursion in TensorFlow. Moreover, we present experiments regarding the efficiency of automatic differentiation which is a key component of deep learning applications.

As an overall remark, our work suggests that the extensive use of *tags* is a promising direction that current implementations of dataflow systems should follow. Tags offer increased parallelism and may prove to offer viable solutions in cases where current technologies appear to bottleneck.

1.3 Structure of the paper

The structure of the remaining part of the paper has as follows. Section 2 presents background on tagged-dataflow; most of the ideas in this section can be traced back to the early steps of dataflow systems. Section 3 presents the proposed transformation algorithm from dynamic dataflow graphs to static ones that contain two new dataflow operations. Section 4 presents how automatic differentiation can be implemented efficiently in the static graphs that result from our transformation. Section 5 describes an implementation of the proposed transformation in the TensorFlow framework. Section 6 presents a performance evaluation of our implementation. Finally, Section 7 compares our approach with related work and Section 8 concludes the paper giving pointers to future work.

2 Tagged Dataflow

The *dataflow model of computation* [9, 10] was developed more than forty years ago, as an alternative to the classical “von-Neumann” computing model. The key motivation was the creation of architectures and programming languages that would exploit the massive parallelism that is inherent in many applications. A dataflow program is essentially a directed graph in which vertices correspond to processing elements and edges correspond to channels. The data that need to be processed start “flowing” inside the channels; when they reach a node they are being processed and the data produced are fed to the output channels of the node. Since various parts of the dataflow graph can be working concurrently, the parallel nature of the model should be apparent. Moreover, this processing of data “while in motion” comes in sharp contrast with the traditional “von-Neumann” model in which data wait passively in memory until they are fetched

by the central processing unit of the (sequential) computer in order to be processed.

2.1 Dataflow Graphs, Tags, and Tokens

A key notion in our discussion is that of a *dataflow graph*:

Definition 2.1. A *dataflow graph* (or *dataflow network*) is a directed graph $G = (V, E)$, where V is the finite set of *nodes* of the graph and E is the set of edges connecting elements of V . The set V is partitioned into disjoint subsets V_I (*input nodes*), V_O (*output nodes*) and V_P (*processing nodes*), subject to the following restrictions:

- Every input node has no incoming edges and has one outgoing edge towards a processing node.
- Every output node has no outgoing edges and has one incoming edge from a processing node.
- Every processing node has incoming edges (at least one) from input nodes and/or from other processing nodes and outgoing edges (at least one) to output nodes and/or other processing nodes.

Intuitively, input nodes provide the input data to a dataflow graph, processing nodes perform the processing of data, and output nodes collect the output data produced by the network.

In the initial dataflow model, channels were assumed to be unbounded FIFO queues, i.e., the data were assumed to flow in a specific order inside the channels. However, it soon became apparent that a model that would not impose any particular temporal ordering of the data would be much more general. This resulted in the so-called *tagged dataflow* model [3, 4, 26]. The basic idea behind tagged dataflow is that data can flow inside a network accompanied by *tags* (i.e., labels). The tags can also carry essential information that can be used in order to implement iterative and recursive algorithms.

Intuitively, edges of our dataflow networks carry tuples of the form $\langle t, d \rangle$ where d is an element of a data domain D and t is an element of a set of tags T . The set T may be quite involved; in its simplest form it can be a set of natural numbers, or in more demanding cases it can be the set of lists of natural numbers, etc. Pairs of the form $\langle t, d \rangle \in T \times D$ are usually referred in the dataflow literature as *tokens*.

2.2 Tagged Dataflow Operations

We now describe the operation of the nodes of a dataflow graph. We start with the simplest case, namely input nodes. Such nodes carry a constant value which they pass to their output. For example, an input node labeled with the natural number 3, passes in its output the value 3. Since we have adopted the convention that edges carry tokens, the value 3 originating from an input node, can be interpreted as a tuple $\langle t, 3 \rangle$, for all possible tags t .

The operation of processing nodes (such as for example, nodes that perform addition, multiplication, and so on), is straightforward (see [4]): every such operator can *fire* when

in all of its inputs there exist tokens that have exactly the same tag. When the operator fires, it produces a token that has exactly the same tag as the input tokens that it has “consumed”. As an example, the operation of binary addition can be described as follows:

$$[[+]] (\langle t, a \rangle, \langle t, b \rangle) = \langle t, a + b \rangle$$

There exist operators that do not need data in all their input tokens in order to produce output or that do not produce data in all their output tokens. We will use the symbol $*$ to denote the absence of data. Historically, $*$ was named a “hiaton” in [27] where it is mentioned that “a hiaton can be thought as the ‘output’ of a process which at a particular point in time has no ‘genuine’ data to send on”. In the TensorFlow terminology, a hiaton is referred as a “dead” data item [1]. Using the concept of hiaton, we can define the semantics of two common dataflow operators:

$$[[\text{SWITCH}]] (\langle t, \text{true} \rangle, \langle t, d \rangle) = (\langle t, d \rangle, \langle t, * \rangle)$$

$$[[\text{SWITCH}]] (\langle t, \text{false} \rangle, \langle t, d \rangle) = (\langle t, * \rangle, \langle t, d \rangle)$$

$$[[\text{MERGE}]] (\langle t, d \rangle, \langle t, * \rangle) = \langle t, d \rangle$$

$$[[\text{MERGE}]] (\langle t, * \rangle, \langle t, d \rangle) = \langle t, d \rangle$$

We assume that all the operators we will be using, with the only exception of MERGE, have the following property: in order to fire, the tokens in their inputs must contain data values that are not hiatons. If some of their data inputs are hiatons then the hiatons are propagated to their outputs. The propagation of hiatons, also referred as “deadness propagation” [1], is also useful in practice and especially in distributed execution (see Section 5.2).

2.3 Tagged Iteration

Iterative algorithms can be represented as cyclic dataflow graphs where the output of one iteration becomes input of the next one. In order to increase parallelism, we would like to allow the nodes of our dataflow graphs to process concurrently data that belong to different iteration cycles. The tags can be used to ensure the implementation of asynchronous iteration in an elegant and effective way. The main idea is precisely defined in the following excerpt from [12]:

Each separate (loop) iteration reuses the same code but with different data. To avoid any confusion of operands from the different iterations, each data value is tagged with a unique identifier known as the iteration level that indicates its specific iteration.

TensorFlow implements in a simple way the above idea. A tag in TensorFlow is a list of natural numbers. The length of the list corresponds to the nesting of the iteration. For example, if we have two nested while-loops, a list of length 2 signifies that we are inside the inner “while”. Each item of the list corresponds to the iteration counter inside a particular loop. So, for example, a list of the form [2,3] means that

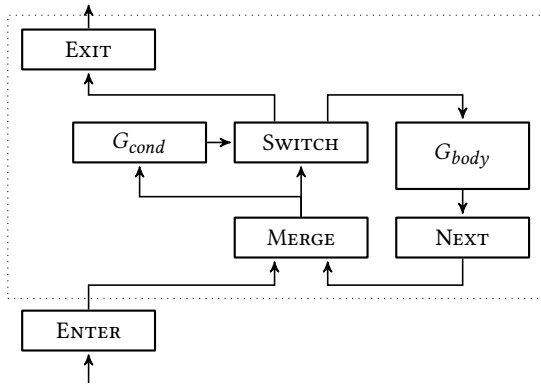


Figure 1. Dataflow implementation of a while-loop

we are inside two nested loops, the outer loop has executed 3 times and the inner loop 2 times. In order to manage these tags, TensorFlow supports the following operators:¹

$$\begin{aligned} \llbracket \text{ENTER} \rrbracket \langle t, d \rangle &= \langle 0 : t, d \rangle \\ \llbracket \text{EXIT} \rrbracket \langle n : t, d \rangle &= \langle t, d \rangle \\ \llbracket \text{NEXT} \rrbracket \langle n : t, d \rangle &= \langle (n + 1) : t, d \rangle \end{aligned}$$

Intuitively, the ENTER operator is used when we enter a new while-loop. It simply adds a new element in the list tag, representing in this way that the level of nesting has been increased by 1. Symmetrically, EXIT is used when we exit a while-loop, and it obviously signals that the level of nesting has just been decreased by 1. Finally, NEXT is used to increase the iteration counter for the while-loop we are currently in. Figure 1 depicts a dataflow graph that uses these operators in order to implement a simple while-loop.

2.4 Functions and Recursion

The introduction of functions and recursion in the tagged dataflow framework is more demanding than that of iteration. Supporting a set of (possibly recursive) function definitions, implies that we have to use a *set* of dataflow graphs. As an example, consider the following simple recursive program computing the factorial of a given number, consisting of two definitions. We use Haskell-like functional notation:

```
result = fact(3)+5
fact(n) = if (n==1) then n else n*fact(n-1)
```

In order to represent this program as a dataflow graph, we need to create two graphs, one for the variable `result` and another for the function `fact`. These two graphs are depicted in Figure 2. In order to represent recursion, each separate graph is assigned a name (`result` and `fact` in our

¹We use the notation $\langle n : t \rangle$ to represent a list where n is its first element (the *head* of the list) and t is also a list (the *tail* of the list $\langle n : t \rangle$).

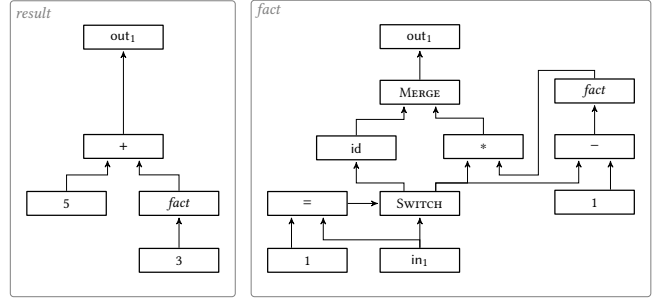


Figure 2. A simple program that computes the factorial expressed as a set of dataflow graphs

example), which is the name of the function that it implements. The dataflow graphs can then use nodes that contain these names, the intuition being that these nodes correspond to recursive calls of the corresponding functions. One can think of each such node as a “black box” that hides inside it a nested graph of the function, which may itself hide inside it another graph, and so on. This intuition seems to suggest that this view of recursion requires a different graph structure than the one implied by Definition 2.1, namely a possibly infinite graph, or a dynamically expanding one. If one wants to remain faithful to the “fixed-graph” approach of Definition 2.1, a different approach for handling recursion must be employed. Such an approach is introduced in the next section.

3 A Tagged Implementation of Recursion

In this section we demonstrate how first-order recursive functions can be implemented in a purely tagged manner. Our construction builds on previous work that was developed several years ago [3, 4, 12, 21, 26, 28], but has distinguishing characteristics from each one of them. A detailed comparison of our approach with the aforementioned ones, is given in Section 7.

Assume we are given a set of dataflow graphs representing a program consisting of (possibly) recursively defined functions. In this section we demonstrate how this set of graphs can be transformed into a single *fixed* graph that can be executed in a tagged way (namely, without the need of dynamic expansion). The key idea is that whenever a function is called, we create a unique new tag that characterizes this specific function invocation; as soon as we return from this function call, we restore the tag to its previous state.

Before we present the algorithm in a formal way, we motivate it through a simple example. Consider the two graphs of the factorial example depicted in Figure 2. In the following, we present each step of the algorithm, together with the intuition behind each step:

- There are two nodes labeled with `fact` in Figure 2. We replace one of them with a node labeled `CALL0` and

the other one with a node labeled $CALL_1$. It is not important which node receives which operator, as long as each different node receives a $CALL$ operator with a different subscript. Intuitively, each $CALL_i$ operator creates a unique new tag that characterizes the specific function call that it has replaced.

- The input node of the fact graph is replaced by a $MERGE$ node. The output edges of $CALL_0$ and $CALL_1$ become the two inputs of the $MERGE$ node. Intuitively, this $MERGE$ node gathers together all the different “entries” to the body of the function fact (avoiding in this way to have different “copies” of the body of fact).
- The output node of the fact graph is connected with two new nodes labeled as $RETURN_0$ and $RETURN_1$. The output edge of each $RETURN_i$ is connected to the node where the output of the corresponding fact node was directed in the initial graphs. Intuitively, each $RETURN_i$ operator restores the tag to the state it had just before the corresponding $CALL_i$.
- We add an extra output edge to each $CALL_i$ connecting it with its corresponding $RETURN_i$ node. Intuitively, this edge expresses the requirement that in order for a $RETURN_i$ node to fire, a $CALL_i$ must have fired at some previous time point². Notice that this is a “synchronizing” edge, i.e., the data value that it carries is irrelevant. Such edges are termed *control edges* in the TensorFlow literature and are usually denoted with dashed lines.

The transformation described above, produces a single dataflow graph depicted in Figure 3. It remains to specify the intuitive and formal meaning of the $CALL_i$ and $RETURN_i$ operators. As we have already discussed, the underlying idea behind the tagged implementation of recursion, is to represent with a unique tag the data that belong to a specific invocation of a function. In this way we avoid mixing data that belong to different invocations. When the operator $CALL_i$ takes as input a token $\langle t, d \rangle$, it changes the tag t into a new one that represents the function call that is being invoked. This is being done by prefixing t with i , namely:

$$[[CALL_i]] \langle t, d \rangle = \langle i : t, d \rangle$$

Intuitively, the list $i : t$ identifies the position in the recursion tree where execution currently is. When the function returns, we must restore the tag to the state it had before the function was called. This is performed by the operator $RETURN_i$ which performs the converse operation from that of $CALL_i$:

$$[[RETURN_i]] \langle i : t, d \rangle = \langle t, d \rangle$$

Using the above semantic equations, one can easily “run” the resulting graph depicted in Figure 3 and verify that it computes the desired output.

²The necessity of adding these edges will be explained in Section 5.2.1.

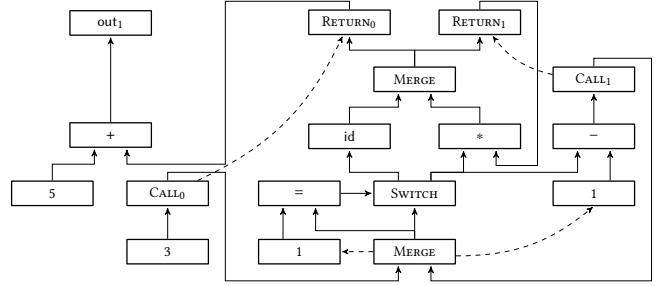


Figure 3. The transformed graph of *fact*

The transformation described for the fact program, can be easily generalized to programs with many different functions. Assume we are given a set S of dataflow graphs representing a set of (possibly recursively) defined functions. We assume that one of these function definitions is for a variable result (intuitively, the output of the program). The description is slightly more complicated because we now consider functions that may have more than one formal parameters. The transformation proceeds as follows:

- Let f be a function defined in S having m formal parameters. Assume there are n nodes in S labeled with the name of f (each such node corresponds to a different call to f). Number these nodes starting from 0 up to $n - 1$ (it is not important which node of f receives which number, as long as each different node receives a different number).
- Replace the i 'th node of f with m identical nodes all labeled with $CALL_i$, each one of them corresponding to one of the m different formal parameters of f . Each one of these $CALL_i$ nodes has only one input, namely the input to the original f node that corresponded to the specific formal parameter of f .
- In the initial graph of f , there exist m input nodes in_0, \dots, in_{m-1} , each one corresponding to a different formal parameter of f . Replace each such input node by a $MERGE$ node. Direct the output edge of each $CALL_i$ node corresponding to a specific formal parameter of f , to the input of the corresponding $MERGE$ node.
- The output of f 's graph is connected with n new nodes labeled $RETURN_0, \dots, RETURN_{n-1}$. The output edge of each $RETURN_i$ is connected to the node where the output of the i 'th node of f was connected in the initial version of the graph.
- We add an extra output edge to each $CALL_i$ connecting it with its corresponding $RETURN_i$ node.

The above algorithm transforms the given set S into a new graph; the output of this graph is the output node of the graph corresponding to the variable result.

4 Automatic Differentiation

Neural networks are typically trained using the backpropagation algorithm [23]. This involves the minimization of a loss function that naturally depends on gradient-based computations. Dataflow systems that focus on deep learning applications typically implement the backpropagation algorithm via a systematic way called reverse-mode automatic differentiation [6]. Roughly speaking, automatic differentiation derives automatically the operations that need to take place in order to compute the gradient of the function that is encoded as a dataflow graph. In this section we describe how we support automatic differentiation in recursive dataflow graphs that have been transformed using the tagging approach.

The automatic differentiation algorithm uses the chain rule to connect newly introduced gradient operators (i.e., operators that compute the gradients) to the graph. More specifically, for a simple tagless dataflow graph, for each operator $(y_1, \dots, y_m) = \text{op}_i(x_1, \dots, x_n)$ the algorithm adds a corresponding gradient operator $\text{op}_i^{\text{grad}}$ that computes the gradients dx_1, \dots, dx_n of the inputs with respect to the gradients dy_1, \dots, dy_m of the outputs of op_i . Note that in general $\text{op}_i^{\text{grad}}$ also requires the original inputs x_1, \dots, x_n of op_i in order to operate. Hence, $\text{op}_i^{\text{grad}}$ is an operator that accepts $n + m$ inputs and produces n outputs. This process results into a dataflow graph that is naturally activated in two phases: the forward and backward phase. The forward phase involves the activation of the original operators of the graph that compute the original outputs (also called forward values) of the dataflow. When all the input values of the last operator are available the backward phase can start to compute the gradient values. Note that, since the gradient operators need the input values of the original operator, the values produced in the forward phase are usually retained in memory during the whole execution to avoid unnecessary recomputations.

In the simple case we have discussed so far the graph consists of primitive operators only. The corresponding gradient operators are predefined and the mapping between the operator and its gradient counterpart is typically maintained in a catalogue for the purpose of the automatic differentiation. However, if we want to consider dataflow graphs that contain user-defined functions as operators, we also need to consider how to generate the appropriate gradient operators of these functions. Recall that, in this setting, each function is defined as a separate dataflow graph and as such can be automatically differentiated. Therefore, the straightforward approach is to derive f^{grad} by applying the automatic differentiation algorithm to the dataflow graph of f . This will produce a new graph that effectively computes the gradient of f . The derived graph can then be added as an ordinary function definition f^{grad} and used as the gradient

operator of f . Figure 4 depicts a set of dataflow graphs consisting of the main graph named *result* that uses a recursive function f . The corresponding f^{grad} is itself recursive, and is produced by automatically differentiating f . Also note that, as expected, the gradient computation of *result* uses both f and f^{grad} .

It is easy to see that the resulting set of dataflow graphs in Figure 4 can be transformed to a single static dataflow graph using the transformation algorithm introduced in Section 3. However, this approach would be highly inefficient: the invocation of f^{grad} triggers the evaluation of the forward values of f that have also been computed some time in the past by the invocation of f . In the specific example, the operators op_1 and op_2 will execute in both invocations of f and f^{grad} in *result*^{grad}. The main reason is that, using this naive process for automatic differentiation, we treat the gradient computation as a separate function and therefore we lose the relationship between the original function invocation and its corresponding gradient computation.

In order to remedy this inefficiency we have devised a more sophisticated way of supporting automatic differentiation. The key idea is to identify each function invocation f and its counterpart invocation f^{grad} in the graph and replace them simultaneously (i.e., assign to them the same label i for $\text{CALL}_i\text{-RETURN}_i$) with a single dataflow graph that computes both the forward values and the gradients. This subgraph will have $n + m$ inputs, corresponding to n inputs of f and m gradient inputs of f^{grad} , and $m + n$ outputs, corresponding to m outputs of f and n gradient outputs of f^{grad} . For example, the transformed graph of Figure 4 is shown in Figure 5. There are two distinguished labels, one corresponding to the invocations in *result*^{grad} and the other to the invocations in f^{grad} .

An interesting observation is that, using this approach, the forward values produced by the operators inside f are communicated to their gradient counterparts which naturally reside in f^{grad} . The use of the same tag for both the original operators and the gradient operators, will ensure that the correct forward values will be matched with the correct invocations of the gradient operators.

5 Implementation in TensorFlow

In this section we discuss the implementation issues that arise in the tagged approach to recursion introduced in the previous sections. As it turns out, the existing infrastructure of TensorFlow requires relatively small changes in order to incorporate the new ideas. In particular, it suffices to extend the computational model of TensorFlow to accommodate the execution of dataflow graphs that contain CALL_i and RETURN_i operators. In the following, we discuss the implementation details that arise in both the single-machine and the distributed execution of these extended dataflow graphs.

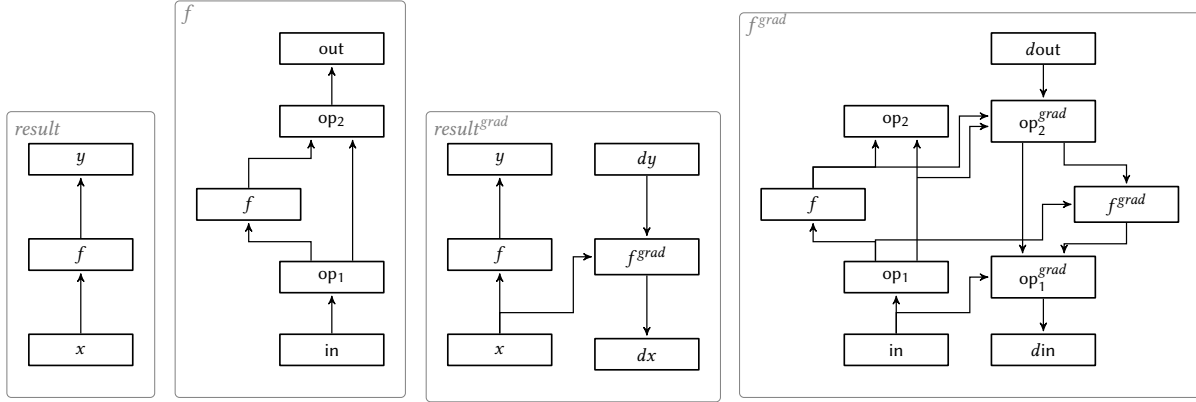


Figure 4. The set of dataflow graphs that compute the gradient of result.

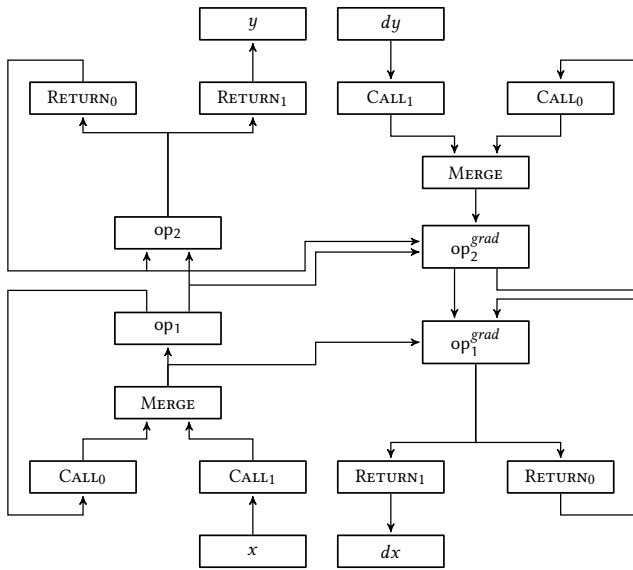


Figure 5. The transformed dataflow graph that computes the gradient of result.

5.1 Local Execution

TensorFlow executes a tagless dataflow graph in a simple way: it places in a “ready” queue those nodes that have all their inputs available, and the remaining nodes in a “pending” queue. It starts by executing (possibly in parallel) the nodes in the “ready” queue. The nodes from the “pending” queue are moved to the “ready” queue when all their inputs become available. The overall computation terminates when both queues become empty.

For dataflow graphs that use tags, TensorFlow allows multiple copies of a single node to appear in the queues, each copy corresponding to a different tag (that the inputs of the node must have in order for the node to fire). Execution proceeds as in the case of tagless graphs. Tags are implemented in TensorFlow with the help of *frames*. A frame conceptually captures a specific scope of execution (e.g., an iteration of a

while-body). Frames are organized in stacks, that is, each frame has a reference to the previous scope. Tags, are essentially pairs of an iteration number and a pointer to that global stack. The ENTER node allocates a new frame initializing its value to 0; the EXIT node deallocates a frame³; the NEXT node increases the value of the current frame by 1.

In our implementation we reuse the frame stack that already corresponds to the static nesting scopes of while-loops to also incorporate the (dynamic) scope of functions. The new primitive operators that we have added, namely $CALL_i$ and $RETURN_i$, actually extend the already available operations ENTER and EXIT of TensorFlow. Therefore, the required changes in the runtime of TensorFlow, are small. More specifically, we extend frames to include a variable in which the label i of $CALL_i$ is stored. Every time a $CALL_i$ node is executed, a new frame is allocated and this local variable of the frame is set to the value i . Similarly, when $RETURN_i$ is executed, the local variable of the current frame is first checked; if it is equal to i , then the frame is deallocated from the stack, otherwise no output is produced.

It is easy to see why the scopes of the functions and the scopes of the while-loops can be fused into a single stack, without adding any extra tagging machinery. Imagine, for example, a program that has a while-loop inside of which a function is called. When the function is called, a new frame is allocated and the dataflow subgraph corresponding to this function call is executed with respect to this frame. When the execution of the function returns, the stack frame is deallocated, and the execution of the dataflow graph corresponding to the while-loop continues with respect to the tag that existed before the function was invoked. In this way, recursive functions and iteration are supported by the same simple tagging mechanism.

³Actually, the operations performed by ENTER and EXIT are more complicated than this, but for the purposes of our discussion, a more precise description is unnecessary.

5.2 Distributed Execution

Distributed execution can be achieved by partitioning the dataflow graph into subgraphs and assigning them to different machines. Each machine then executes the subgraph locally, using the same process described in Section 5.1. The distributed computation completes when all the machines finish.

TensorFlow imposes no restriction to the partitioning of the graph and as a result parts of the same function may reside on different machines. In order not to limit parallelism, TensorFlow employs a design that has no centralized coordination of the different machines. Instead, the subgraphs that reside on the different machines are extended with `SEND` and `RCV` nodes that enable direct communication between the machines. In order to support recursion, we have extended this framework as described below.

5.2.1 Deadness Propagation

When all the results of a distributed computation become available, one has to ensure that all the machines that have participated in the computation, will terminate. As an example where such a need arises, consider the case of the subgraph of a conditional branch executing in a separate machine waiting for input via a `RCV` node. If, at runtime, this branch is not selected, then the machine will wait indefinitely and the execution will not terminate without some intervention. This issue is tackled in TensorFlow by sending a “dead” token to explicitly signify the absence of data.

In our extended framework, we must ensure the correct deadness propagation in the presence of the two new operators `CALLi` and `RETURNi`. In particular, we ensure that the dead token received in a `CALLi` will be propagated to its corresponding `RETURNi`, by adding a control edge from the former node to the latter. In this way every `RETURNi` will receive a “dead” token, which it will also propagate to its successor nodes. This process ensures the termination of all processes that are engaged in a distributed computation.

5.2.2 Tag Tracking

Tag tracking is an issue that arises from the fact that nodes that operate on the same tagged data can be distributed in several machines and connect directly through a `SEND-RCV` pair. Recall that in a local execution setting a node is scheduled accompanied with a specific tag, that is the tag that the data should match. An immediate consequence is that `RCV` nodes should also be scheduled with respect to a certain tag that may have originally been generated in a different machine (that is, the tag-generating nodes reside in a different machine). This observation makes it apparent that a mechanism should be established in order to track among the relevant machines which tags have been generated. TensorFlow tackles this issue by creating a “state machine” in every relevant machine that is essentially the backbone of the graph that includes the tag-generating nodes (e.g. `ENTER`, `NEXT`).

This apparatus simulates the decisions that occur distributively in the graph (that is, which branch in `SWITCH` nodes is active) and replays the tag creation process with dummy input data to all the machines that do not originally have those nodes.

In the case of the tagged dataflow graphs with recursion, the form of the “state machine” is slightly more complex but similar. In order to extract the correct form of the state machines we do the following:

- We traverse the graph and capture the occurrences of the tag-generating nodes. As tag-generating we consider `CALLi`, `ENTER` and `NEXT`.
- We also capture the occurrences of the control-flow decision nodes, namely the `SWITCH` and `MERGE` nodes.
- We generate a new graph using copies of the captured nodes. The edges of the graph are inferred by the original graph if we suppress the remaining nodes.
- We introduce a control edge between a tag-generating node n and the `RCV` nodes that connect nodes after node n and before the next tag-generating node. Intuitively these `RCV` nodes should wait for data with tags that n generates.
- Partitions that have `RCV` nodes that are connected in the previous step get a copy of this graph.

Deadness propagation and tag tracking, as outlined above, were the two most important issues we had to tackle when extending TensorFlow’s distributed execution for tagged recursion.

6 Evaluation

In this section we evaluate our approach and implementation in terms of performance. In particular, we first test our approach against some classic microbenchmarks designed to stress recursive calls. Moreover, since TensorFlow is primarily focused on machine learning tasks we test our approach in a recursive neural network architecture.

6.1 Functional Microbenchmarks

We used a variety of recursive functions that due to their definitions in terms of heavy use of recursion and demanding number of computations are regularly used as benchmarks for recursion optimization tasks. We compare the execution time of the tagged approach against the one that is already implemented in TensorFlow (i.e., the expanding graph).

Table 1 presents the average execution times of different execution of these microbenchmarks. The two implementations are essentially comparable in the single-machine setting as expected. Both implementations have their shortcomings in terms of imposed overhead in the execution engine. On one hand, the dynamic implementation suffers from an overhead due to the repetitive creation and initialization of the execution environment. This overhead seems to be analogous to the number of invocations and it is evidenced

Function	Static (sec)	Dynamic (sec)	Speed-up
fib(24)	0,812	0,990	18,00%
fib(25)	1,259	1,629	22,31%
fib(26)	2,127	2,572	17,31%
fib(27)	3,295	4,178	21,15%
fib(28)	5,338	6,739	20,79%
fib(29)	8,614	10,937	21,24%
fib(30)	13,968	17,801	21,53%
fib(31)	22,717	28,610	20,60%
fib(32)	37,446	46,855	20,08%
fib(33)	61,287	75,646	18,98%
ack(3, 3)	0,089	0,118	24,76%
ack(3, 4)	0,348	0,503	30,74%
ack(3, 5)	1,511	2,095	27,88%
ack(3, 6)	6,165	8,323	25,92%
ack(3, 7)	25,489	33,147	23,10%
ack(3, 8)	100,882	128,955	21,77%
tak(24, 16, 8)	18,881	16,075	-17,46%
tak(25, 16, 8)	29,545	25,394	-16,36%
tak(26, 16, 8)	45,218	38,722	-16,78%
tak(27, 16, 8)	67,757	57,876	-17,07%
tak(27, 17, 8)	188,506	159,778	-17,98%
primes(7500)	13,502	13,458	-0,33%
primes(8000)	14,836	14,784	-0,35%
primes(8500)	16,268	16,076	-1,19%
primes(9000)	17,556	17,348	-1,20%
primes(9500)	18,868	18,912	0,23%
primes(10000)	20,636	20,378	-1,27%

Table 1. Comparison of execution times (in seconds) in various invocations of well-known recursive function definitions. In particular, fib computes the Fibonacci sequence, ack is the Ackermann function, tak is the Takeuchi function and primes is the prime-counting function.

by the constant speed-up factor on the execution of the function fib (i.e., the recursive implementation of Fibonacci sequence). On the other hand, the tagged implementation is slower in the tak (i.e., the Takeuchi function). This is due to the inefficient manipulation and management of the frames in the execution engine. In particular, there is an overhead for concurrently accessing the global frame stack that limits the parallelism. This phenomenon is more evident for functions with many arguments. Recall that the frame stack was initially used by TensorFlow’s engine to store static scopes of while-loops. We believe that this inefficiency can be lifted by carefully redesigning the execution engine

We generally believe that functions with definitions that comprise a large number of operations tend to behave worse

Method	Training	Inference
Unrolling	0.99	1.74
Iteration	84.56	138.8
Recursion	143.33	161.29

Table 2. Comparison of three different implementations of a TreeRNN machine learning task. The measurements are throughput (instances per second) for both training and inference.

Batch Size	Iteration	Recursion
1	84.56	143.33
5	186.96	103.06
10	228.60	101.27
25	235.69	110.72

Table 3. Comparison of the throughput (instances per second) for two different implementations of training a TreeRNN model with different batch sizes.

in terms of performance when executed by the dynamic approach because of the repetitive copying of their large encoded graphs at runtime whereas functions with smaller definitions do not suffer as much by this particular overhead. However, a static approach may have the potential to impose an even greater challenge to the current implementation of TensorFlow, given that there still lies some space for further internal optimizations regarding the management of frames and tags.

6.2 Machine Learning Tasks

As a real-life example we experimented with the simplest model that belongs in the greater family of Recursive Neural Networks [24] also known as TreeRNN. In particular, we consider the task of predicting the sentiment of English phrases using a snapshot of the Stanford Sentiment Treebank dataset for training our model. In this dataset every data instance is a fully labeled parsed tree encoding the implicit tree-like structure of a certain sentence. While this task is recursive in nature the training can proceed bottom-up and as a result can be computed iteratively.

We compare the recursive implementation with two other implementations. First, the most straightforward implementation is to statically unroll the computation graph during its construction so that its form resembles the tree structure of the given data instance. This method generates a separate dataflow graph per input that will be executed only once throughout the overall training. The second implementation is using an iterative mechanism as a means to process the nodes of a particular tree in a bottom-up fashion.

Table 2 presents the performance of the three implementations of TreeRNN. We measure the throughput (i.e., instances per second) for both training and inference. We train on a set of 700 examples in total. As expected, the static unrolling approach has the worst throughput of the three. The iterative method performs also worse than the recursive. The parallelism in the iterative method is limited since it processes each level of the tree sequentially. On the other hand, the recursive method can process different levels of the tree concurrently.

We also experimented with different batch sizes during training to observe how this affects the throughput. Table 3 compares the iterative and recursive method under training sessions with different batch sizes. The iterative method seems to benefit from batching and improves as batch size increases. On the other hand, the recursive method does not seem to benefit from the increase of the batch size. This is expected since each instance produces different invocations of the operators. However, we believe that we can optimize the execution of recursion in order to take into consideration the batch execution of the same operator scheduled under different tags. This optimization was out of scope of this work and left as future direction.

As an overall remark, the recursive method can express computations that can make more dynamic decisions during runtime. We choose to evaluate on the simplest recursive neural network in order to have the chance to compare with different approaches. However, it should be noted that more dynamic neural networks, such as Top-down TreeLSTM [30], cannot be expressed using iteration and therefore could not be used in this evaluation experiment. We should note here that there is a recent recursive proposal based on the ideas of expanding graphs [14] (a more detailed discussion can be found in Section 7) that support automatic differentiation and could be possibly used as a comparison on machine learning tasks. However, we could not find any working implementation of that approach.

7 Related Work

In this section we present a comparison of the proposed technique initially with approaches that were developed in the early days of dataflow, and then with more contemporary systems.

7.1 Tagged Dataflow Systems

The implementation of recursive functions using a tagged approach, has been explored even from the early days of dataflow. In the Manchester prototype dataflow computer [12], in order to implement first-order recursive functions, tags are extended with an *activation name* which is used to distinguish different function invocations. In the MIT tagged-token architecture [4], a similar approach is used: each token is tagged with a *context identifier* that specifies the activation to which the token belongs. In this way, tokens

corresponding to different activations may flow simultaneously through the dataflow graph, offering increased parallelism. As it is noted in [4] the context identifier is the dataflow analogue of the “frame pointer” in traditional activation records. Both of the above works were groundbreaking and created the conceptual basis for the computational paradigm of tagged dataflow. The present work differs from both of the above techniques: we propose an algorithm for transforming any set of dataflow graphs corresponding to recursive functions, into a single static dataflow graph. Both [12] and [4] describe execution mechanisms for recursive functions, without explicitly describing an algorithm for deriving a static dataflow graph. The static graphs we obtain are of paramount importance in our more contemporary framework: they helped us in extending the distributed computational model of TensorFlow to handle recursion, and they guided us in obtaining an efficient procedure for automatic differentiation. Both of these issues are extremely important for modern deep learning applications.

In [28], a framework is developed for formalizing the implementation of first-order recursive functions in a tagged manner. More specifically, it is demonstrated that such an implementation can be achieved by transforming the source recursive programs into simple *intensional* definitions which can then be executed in a demand-driven dataflow approach. This transformation introduced in [28], is further examined and proven correct in [21]. Finally, in [22] it is demonstrated that a large class of *higher-order* recursive functions can also be implemented in a tagged manner. These approaches have been later used in order to derive an implementation of a general-purpose lazy functional language [7, 11]. In these approaches the tags are also represented as a global stack similarly to the implementation presented in this paper. The major difference, however, is that the execution proceeds in a demand-driven instead of a data-driven manner. The graph is not explicitly constructed during execution and no offloading to specialized computational units was considered. Moreover, none of these implementations address distributed execution nor automatic differentiation which is essential for machine learning applications. However, it would be interesting to investigate how the technique developed in [21] can be adapted to data-driven execution and then to examine how the resulting transformation is semantically related to the graphs that result from the proposed transformation algorithm.

Naiad [17], a more recent distributed dataflow system focused on data processing, also makes heavy use of tags. Naiad implements the differential dataflow model [16] that employs tags to support incremental and iterative computations. However, Naiad does not support recursive dataflows.

7.2 Machine Learning Dataflow Systems

Machine learning frameworks that follow the “in-graph” approach, such as TensorFlow [1] and Theano [2], encode dynamic control-flow decisions in the graph. However, user-defined functions are not fully supported by these two systems. More specifically, both systems support non-recursive function definitions. TensorFlow applies an optimization that inlines the body of a non-recursive function in the main dataflow graph. On the other hand, Theano does not support recursive definitions while TensorFlow supports them only partially without supporting automatic differentiation on them. TensorFlow executes a recursive function by dynamically expanding the graph at runtime when a recursive call is encountered. In order to support algorithms that recursively traverse tree-like data structures, a transformation [15] must be applied that transforms recursion to iteration that TensorFlow supports. Transforming recursion to efficient iteration is not generally a straightforward task.

A recent related work that tries to improve recursion support in TensorFlow [14] employs the same approach of the dynamically expanding graph to execute recursive functions and defines a recursive automatic differentiation technique of such functions. However, that work considers only a single-machine execution. The main disadvantage of this particular approach is that a dynamic graph cannot contain the subgraphs that correspond to the recursive callee functions, thus, they cannot be distributed without central coordination. Every special node representing the calling of a function resides on one partition and initiates the execution of the function by just one worker. That means, that the execution of functions cannot be easily shared amongst multiple machines.

Other machine learning frameworks, such as Torch [8], PyTorch [20], DyNet [19] and Chainer [25] follow the “out-of-graph” approach, that is they rely on the host language capabilities to encode control-flow decisions. This lacks efficiency because the ability to perform optimizations on the overall graphs is significantly limited.

7.3 Other Dataflow Systems

CIEL [18] represents a program as an unstructured dynamic task graph in which tasks can tail-recursively spawn other tasks. CIEL supports distributed recursive algorithms by transforming them into a continuation passing style. Contrary to the current work, CIEL makes use of a master node that maintains a list of pending tasks and acts as a task dispatcher to the workers. The major drawback of unstructured dynamic graphs is that they are much more challenging to optimize holistically.

8 Conclusions and Future Work

In this paper we proposed an algorithm for transforming a set of dynamic (ie., expandable) dataflow graphs into a

single static (ie., non-expandable) graph. Our technique can be extended to support automatic differentiation, something really important for machine learning applications. All the proposed ideas have been implemented in the TensorFlow framework.

We believe that *tagging* is a very promising direction that current implementations of dataflow systems should further exploit. Tags offer increased parallelism and may prove to offer viable solutions in cases where current technologies appear to bottleneck. As possible future directions, we would like to investigate the possibility of implementing *higher-order* recursive functions under the TensorFlow framework. In [22] it was suggested that this can be achieved by using more complicated tags, an idea that is worth further consideration. Additionally, we would like to investigate the possibility of implementing user-defined data structures in a purely tagged manner. Finally, we would like to investigate the relationship between *multidimensional programming* [5] (a form of programming that was proposed more than 25 years ago as an extension of classical dataflow programming) and the use of tensors in modern dataflow systems. More specifically, tensors appear to be closely related to the multidimensional streams introduced in [5], and therefore the techniques introduced in [5] may be applicable to modern tensor-based systems. In conclusion, we believe that tagged dataflow is a very interesting computational paradigm, and its resurgence through the appearance of modern dataflow systems, may lead to exciting new developments both in foundational as-well-as practical issues.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283.
- [2] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *CoRR* abs/1605.02688 (2016). arXiv:1605.02688
- [3] Arvind and D. Culler. 1983. *The tagged token dataflow architecture (preliminary version)*. Technical Report. Laboratory for Computer Science, MIT, Cambridge, MA.
- [4] Arvind and Rishiyur S. Nikhil. 1990. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Computers* 39, 3 (1990), 300–318. <https://doi.org/10.1109/12.48862>
- [5] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. 1995. *Multidimensional Programming*. Oxford University Press, Inc., New York, NY, USA.
- [6] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: a Survey. *J. Mach. Learn. Res.* 18 (2017), 153:1–153:43.
- [7] Angelos Charalambidis, Athanasios Grivas, Nikolaos S. Papaspyrou, and Panos Rondogiannis. 2008. Efficient Intensional Implementation for Lazy Functional Languages. *Mathematics in Computer Science* 2, 1 (2008), 123–141. <https://doi.org/10.1007/s11786-008-0047-5>

- [8] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: a modular machine learning software library*. Technical Report. Idiap.
- [9] A. L. Davis. 1978. The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine. In *ISCA*, Edward J. McCluskey, John F. Wakerly, E. David Crockett, Thomas E. Bredt, David J. Lu, William M. van Cleemput, Susan S. Owicki, Roy C. Ogus, Ravi Apte, M. Danielle Beurdy, and Jacques Losq (Eds.). ACM, 210–215.
- [10] Jack B. Dennis and David Misunas. 1974. A Preliminary Architecture for a Basic Data Flow Processor. In *ISCA*, Willis K. King and Oscar N. Garcia (Eds.). ACM, 126–132.
- [11] Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis. 2013. The Generalized Intensional Transformation for Implementing Lazy Functional Languages. In *Practical Aspects of Declarative Languages - 15th International Symposium, PADL 2013, Rome, Italy, January 21-22, 2013. Proceedings (Lecture Notes in Computer Science)*, Konstantinos Sagonas (Ed.), Vol. 7752. Springer, 157–172. https://doi.org/10.1007/978-3-642-45284-0_11
- [12] John R. Gurd, Chris C. Kirkham, and Ian Watson. 1985. The Manchester Prototype Dataflow Computer. *Commun. ACM* 28, 1 (1985), 34–52. <https://doi.org/10.1145/2465.2468>
- [13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>
- [14] Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byung-Gon Chun. 2018. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 19:1–19:13. <https://doi.org/10.1145/3190508.3190530>
- [15] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. *CoRR* abs/1702.02181 (2017). arXiv:1702.02181
- [16] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org.
- [17] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. <https://doi.org/10.1145/2517349.2522738>
- [18] Derek Gordon Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association.
- [19] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. *CoRR* abs/1701.03980 (2017). arXiv:1701.03980
- [20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
- [21] Panos Rondogiannis and William W. Wadge. 1997. First-Order Functional Languages and Intensional Logic. *J. Funct. Program.* 7, 1 (1997), 73–101.
- [22] Panos Rondogiannis and William W. Wadge. 1999. Higher-Order Functional Languages and Intensional Logic. *J. Funct. Program.* 9, 5 (1999), 527–564.
- [23] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323 (1986), 533–536.
- [24] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, 1631–1642. <https://www.aclweb.org/anthology/D13-1170/>
- [25] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a Next-Generation Open Source Framework for Deep Learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*.
- [26] I. Watson and J. R. Gurd. 1979. A prototype data flow computer with token labelling. In *Proceedings of the National Computer Conference*. 623–628.
- [27] Edward A. Ashcroft William W. Wadge. 1985. *Lucid, the Dataflow Programming Language*. Academic Press.
- [28] A. A. Yaghi. 1984. *The Intensional Implementation Technique for Functional Languages*. Ph.D. Dissertation. Department of Computer Science, University of Warwick, Coventry, UK.
- [29] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, et al. 2018. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 18:1–18:15. <https://doi.org/10.1145/3190508.3190551>
- [30] Xingxing Zhang, Liang Lu, and Mirella Lapata. 2016. Top-down Tree Long Short-Term Memory Networks. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, Kevin Knight, Ani Nenkova, and Owen Rambow (Eds.). The Association for Computational Linguistics, 310–320.