Implementation of Control Flow in TensorFlow

Original doc authored 2016/11/04

Overview

This document presents the current design and implementation of control flow operators in TensorFlow. This is a descriptive document based on the original design; please see the actual implementation for details.

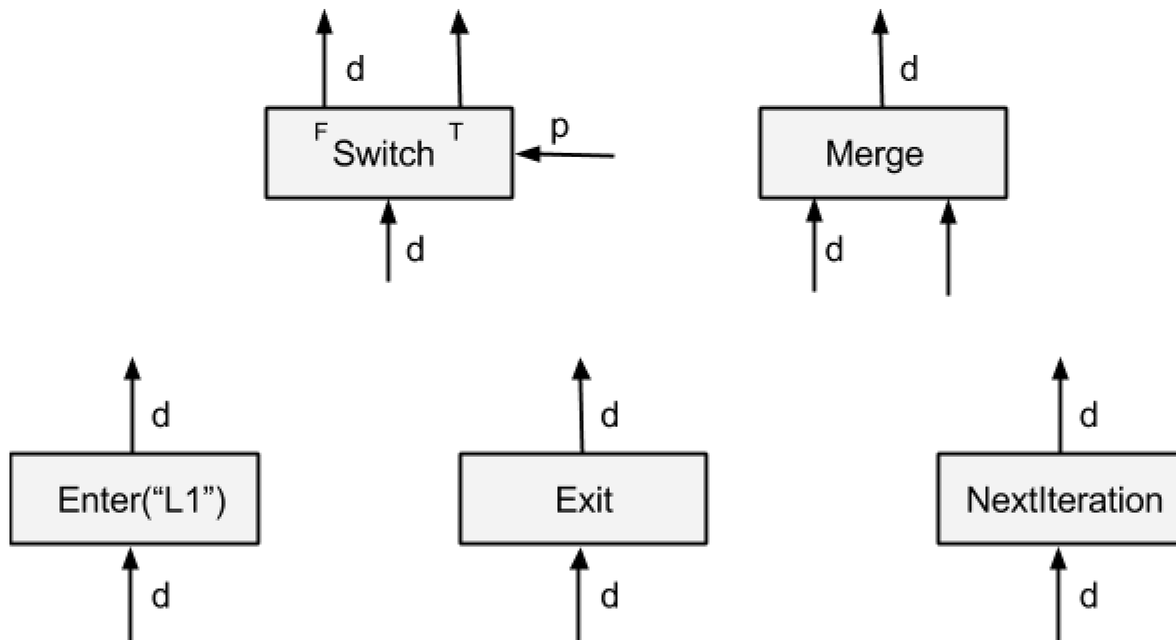
In this document we will:

- Introduce the five TensorFlow primitive operators that are added specifically to handle control flow,
- Show how the high-level control-flow constructs get compiled down to dataflow graphs that involve these five primitives,
- Explain how these dataflow graphs are executed by the TensorFlow runtime, including distributed execution on a set of hybrid devices (e.g., CPU, GPU, and TPU), and
- Describe how automatic differentiation works for the control-flow constructs.

Control-Flow Primitives

The basic design principle of control flow in TensorFlow is to introduce a very small set of simple, primitive operators that can be used to express complex flows of control for a wide range of TensorFlow applications. We want these primitives to be flexible and expressive, serving as a good compilation target for high-level domain specific languages (DSLs). They should fit well with the dataflow model of TensorFlow, and should be amenable to parallel and distributed execution and automatic differentiation. This section introduces these primitives. There are five control-flow primitive operators as shown below. They closely resemble the control-flow primitives introduced in the dataflow machines developed by Dennis and Arvind. The combination of Switch and Merge allows us to implement conditionals. All five primitives together allow us to implement while loops.

(c) 2017 TensorFlow Authors. TensorFlow design document, last updated 2017/11/1



In TensorFlow, every op is executed in an *execution frame*, and the control-flow primitives are responsible for creating and managing these execution frames. Intuitively, for each while loop, the TensorFlow runtime sets up an execution frame and runs all the ops belonging to the while loop inside the execution frame. Execution frames can be nested. Nested while loops run in nested execution frames. Ops from different execution frames can run in parallel as long as there is no data dependency between them.

Switch: A *Switch* operator forwards the input tensor d to one of its outputs depending on the boolean tensor of the control input p. A Switch is enabled for execution when both its inputs are available.

Merge: A *Merge* operator forwards one of its available inputs to its output. A Merge is enabled for execution when any of its inputs is available. It is unspecified which available input it outputs if there are multiple inputs available.

Enter(name): An *Enter* operator forwards its input to the execution frame that is uniquely identified by the given name. This Enter op is used to pass a tensor in one execution frame to a child execution frame. There can be multiple Enter ops to the same child execution frame, each making a tensor available (asynchronously) in that child execution frame. An Enter is enabled for execution when its input is available. A new execution frame is instantiated in the TensorFlow runtime when the first Enter op to that frame is executed.

Exit: An *Exit* operator forwards a value from an execution frame to its parent execution frame. This Exit op is used to return a tensor computed in a child execution frame back to its parent frame. There can be multiple Exit ops to the parent frame, each asynchronously passing a tensor back to the parent frame. An Exit is enabled when its input is available.

NextIteration: A *NextIteration* operator forwards its input to the next iteration in the current execution frame. The TensorFlow runtime keeps track of iterations in an execution frame. Any op executed in an execution frame has a unique iteration id, which allows us to uniquely identify different invocations of the same op in an iterative computation. Note that there can be multiple NextIteration ops in an execution frame. The TensorFlow runtime starts iteration N+1 when the first NextIteration op is executed at iteration N. As more tensors enter an iteration by executing NextIteration ops, more ops in that iteration will be ready for execution. A NextIteration is enabled when its input is available.

Compilation of Control-Flow Constructs

With the addition of these five control-flow primitives, high-level programming constructs such as cond and while_loop can now be compiled into dataflow graphs that can be executed by TensorFlow runtime. We now look at how cond and while_loop are implemented in TensorFlow.

The cond Operator

Below is the high-level pseudocode of building the dataflow graph of cond(pred, fn1, fn2). For simplicity, we ignore many important issues in real implementation. Readers may find the implementation in control_flow_ops.py.

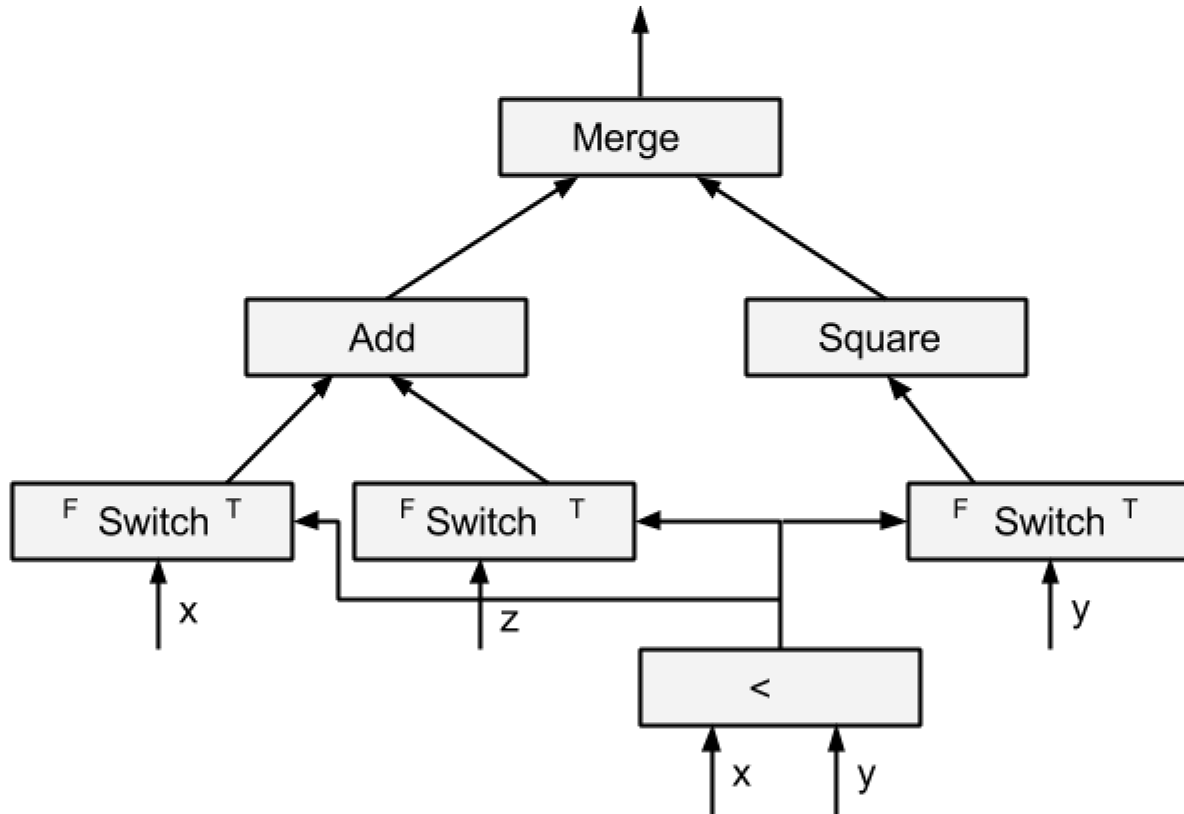
```
# Build the graph for the true branch
context_t = CondContext(pred, branch=1)
res_t = context_t.Call(fn1)

# Build the graph for the false branch
context_f = CondContext(pred, branch=0)
res_f = context_f.Call(fn2)

# Add the Merge nodes for the outputs
merges = [Merge([f, t]) for (f, t) in zip(res_f, res_t)]
return merges
```

For each branch of the cond, we create a new control-flow context for conditionals, and call its graph construction function (fn1 or fn2) inside the context. The conditional context allows us to capture any external tensors (not created in the context) and insert an appropriate Switch op to guard its entering into a branch. This ensures that any ops in a branch will only be executed when that branch is taken. Because of TensorFlow's async execution model, those external tensors may become available at very different times, so we use one Switch op for each external tensor to maximize parallelism.

Each branch returns a list of tensors as result (ref_t or res_f); we then add a list of Merge nodes to merge the the true and false values for each output, respectively. Again, the outputs may be

computed at very different times, so we use one Merge op for each output, which allows us to enable downstream computation as soon as possible.

As an example, let us look at a simple program.




```
tf.cond(x < y, lambda: tf.add(x, z), lambda: tf.square(y))
```

In the generated dataflow graph, the Switch ops are inserted to control the flow of tensors x, y, and z. On the true/false branch, only the true/false outputs of the Switch ops are used. Since the inputs of add are from the true output of the Switch ops, the Add op is only executed when x < y is true. Similarly, the Square op is only executed when x < y is false. The final Merge op emits either the result of the Add or the Square. If there are multiple outputs, there will be multiple Merge ops, one for each output.

There are multiple ways to encode cond using Switch and Merge. We choose the current encoding mainly because it makes automatic differentiation of cond simpler.

The while_loop Operator

Below is the high-level pseudocode of building the dataflow graph of while_loop(pred, body, loop_vars):

(c) 2017 TensorFlow Authors. TensorFlow design document, last updated 2017/11/1

```
while_context = WhileContext()
while_context.Enter()

# Add the Enter nodes for each loop variable.
enter_vars = [Enter(x, frame_name) for x in loop_vars]

# Add the Merge nodes. Note that input[1] will be updated later.
merge_vars = [Merge([x, x]) for x in enter_vars]

# Build the loop pred subgraph.
pred_result = pred(*merge_vars)

# Add the Switch nodes.
switch_vars = [Switch(x, pred_result) for x in merge_vars]

# Build the loop body subgraph.
body_result = body(*[x[1] for x in switch_vars])

# Add the NextIteration nodes.
next_vars = [NextIteration(x) for x in body_result]

# Form the cycles for the loop.
for m, v in zip(merge_vars, next_vars):
  m.op._update_input(1, v)

# Add the Exit nodes.
exit_vars = [Exit(x[0]) for x in switch_vars]
while_context.Exit()
return exit_vars
```

The entire while loop graph is created in a control-flow context for while loops. The basic idea here is quite simple.

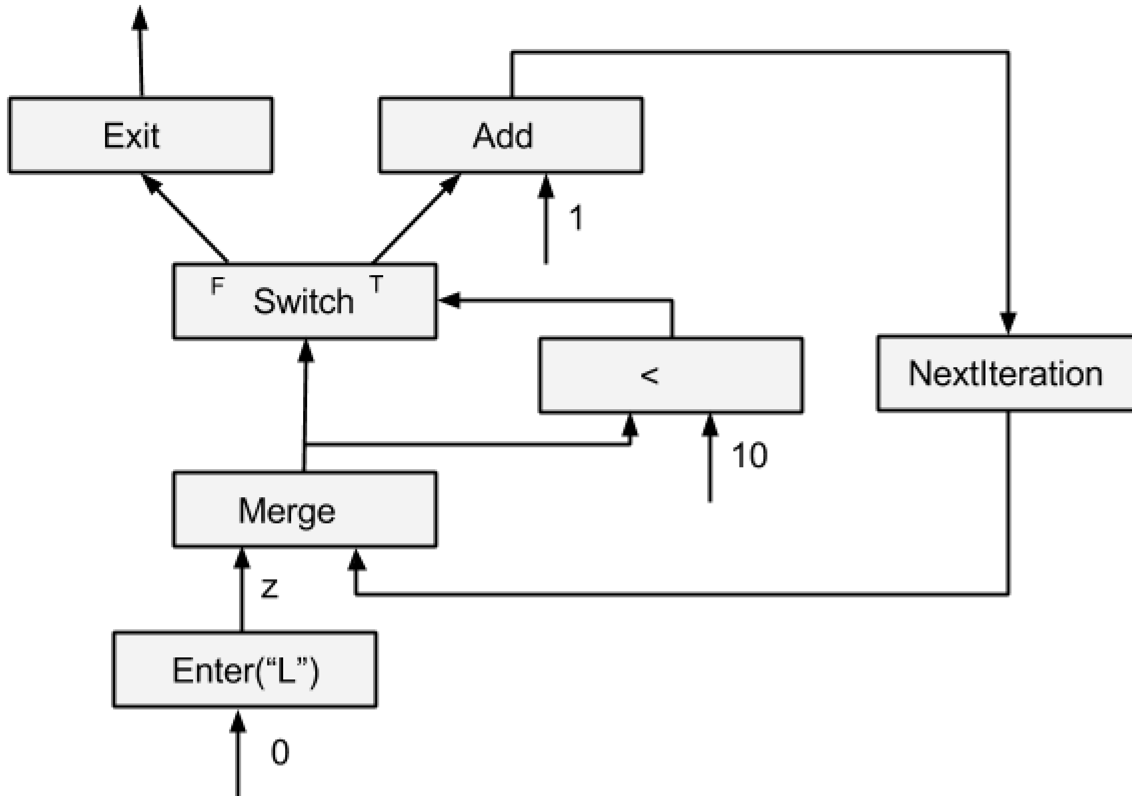
Starting from the loop variables, we add an Enter op and then a Merge op for each of them. We then use the result (merge_vars) to build the pred subgraph, which computes the loop termination condition.

After adding the Switch ops, we use the true outputs of the Switches to build the subgraph for the body of the while loop. The results of the loop body need to go into the next iteration, so we add the NextIteration ops and connect them back to the second inputs of the Merge ops. This forms cycles, allowing us to run the same op repeatedly many times when executing a graph.

The false outputs of the Switch ops are the outputs of the entire while loop, so we add the Exit ops to them and return the outputs of the Exit ops. Similar to cond, the while loop context is used to keep track of external tensors used in the pred and body lambdas. These external tensors are treated as loop constants, and we automatically insert an Enter op for each such

external tensor, making it accessible within the while loop context. Nested loops require adding nested Enter ops.

Again, let us look at the generated graph for one of a simple program.




```
tf.while_loop(lambda i: i < 10, lambda i: tf.add(i, 1), [0])
```

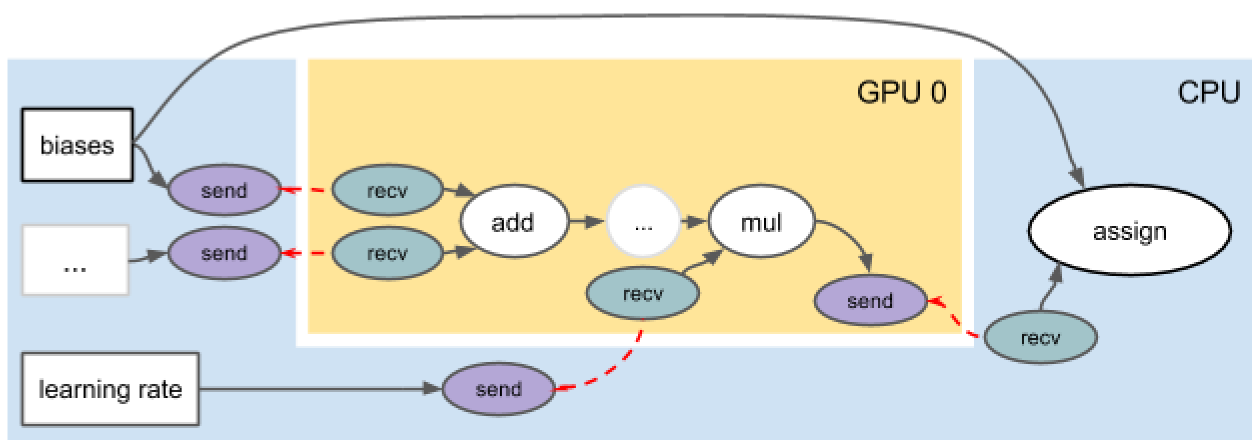
For this example, we have only one loop variable. If there are multiple loop variables, we will have multiple Enter, Merge, Switch, NextIteration, and Exit ops. This enables parallel executions across multiple loops and across multiple iterations within a loop. You may notice that we omit among other things to explain how the constants are handled in a while loop. Please look at the real code if you want to understand the next level of details.

This translation of cond and while_loop supports arbitrary nestings of conditionals and loops. For example, a loop body can call another while_loop, which will be translated recursively as a nested subgraph. The translation ensures that each loop is statically assigned a unique frame name.

Implementation

The TensorFlow runtime is responsible for the execution of the dataflow graphs. Let us start with a quick overview.

To run on multiple devices, TensorFlow automatically assigns the ops to the set of devices. Based on the device placement, TensorFlow automatically partitions the dataflow graph into a set of subgraphs, one per device. When an edge is broken by the partitioning, we automatically insert a pair of send and recv nodes for transporting tensors across devices. A pair of send and recv communicates with a unique key, and recv proactively pulls data from send. For example, the following is the result of partitioning a graph onto two devices. TensorFlow imposes no restrictions on partitioning: A node can be assigned to a device as long as the computation can be done on that device.



Graph after partitioning

The execution of a subgraph is managed by an executor local to the device the subgraph is assigned to. The executor starts from the source nodes and repeatedly executes the ready nodes. A node, with the exception of Merge node, becomes ready when all its inputs are available. Note that all recv nodes in a subgraph are considered to be source nodes.

Without control flow, graph execution is conceptually quite straightforward: Every node is executed exactly once and the execution is done when all nodes are executed. Control flow introduces quite a bit of complexity. A node now can be executed any number of times including 0. The executor needs to be able to manage the (possibly concurrent) execution of multiple instances of the same node, and to determine the completion of graph execution.

To keep track of the tensors generated during execution, tensors inside the executor are represented as a tuple d = (value, is_dead, tag), where value is the actual tensor, is_dead is a boolean indicating if the tensor is on an untaken branch of a conditional, and tag is a string uniquely identifying the tensor (and the execution instance of the node producing the tensor). Intuitively, the tag defines an execution context, and within an execution context a node is executed at most once. The tag is part of the communication key of a send/recv pair to distinguish multiple invocations of the same send/recv pair.

(c) 2017 TensorFlow Authors. TensorFlow design document, last updated 2017/11/1

The executor follows the following rules of execution *(Note: All the inputs of a node must have the same tag.)*

- Switch(p, d) = (r_1, r_2) :
  r_1 = (value(d), p || is_dead(d), tag(d))
  r_2 = (value(d), !p || is_dead(d), tag(d))

- Merge(d_1, d_2) = r :
  r = if is_dead(d_1) then d_2 else d_1

- Enter(d, frame_name) = r :
  value(r) = value(d)
  is_dead(r) = is_dead(d)
  tag(r) = tag(d)/frame_name/0

- Exit(d) = r :
  value(r) = value(d)
  is_dead(r) = is_dead(d)
  tag(r) = tag_1 where tag(d) = tag_1/frame_name/n

- NextIteration(d) = d_1 :
  value(d_1) = value(d)
  is_dead(d_1) = is_dead(d)
  tag(d_1) = tag_1/frame_name/(n+1) where tag(d) = tag_1/frame_name/n

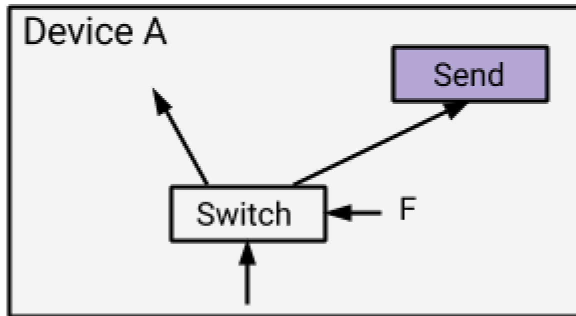
- Op(d_1, …, d_m) = (r_1, …, r_n) :
  value(r_i) = Op.Compute(value(d_1), …, value(d_m)) if !is_dead(r_i)
  is_dead(r_i) = any(is_dead(d_1), … is_dead(d_m)), for all i
  tag(r_i) = tag(d_1), for all i

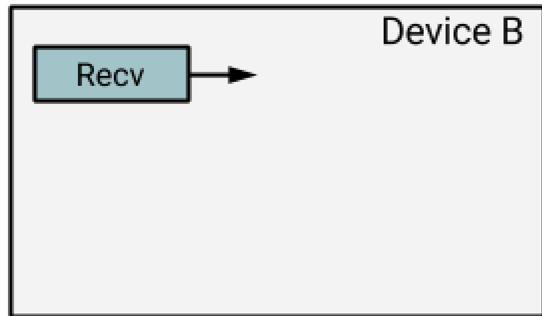
The last rule is for all non-control-flow nodes. Note that the actual computation is performed only when all the inputs are not dead. If there is a dead input, we will skip the computation and propagate a dead signal downstream. This propagation of deadness is used to support distributed execution of control flow.

Distributed Conditional Execution

For distributed execution, a cond can be partitioned onto multiple devices, as shown below.

(c) 2017 TensorFlow Authors. TensorFlow design document, last updated 2017/11/1

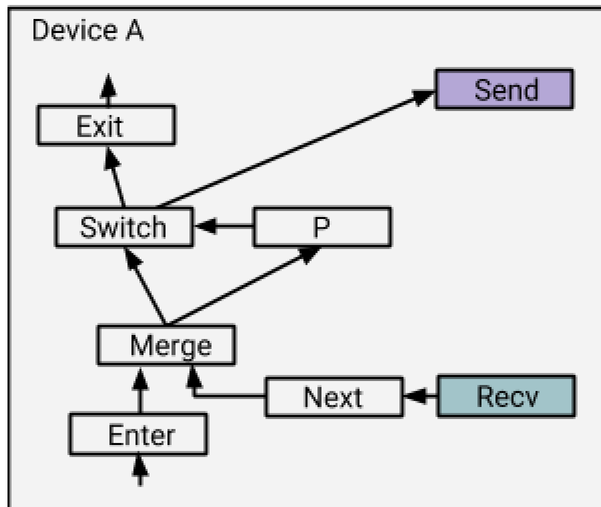


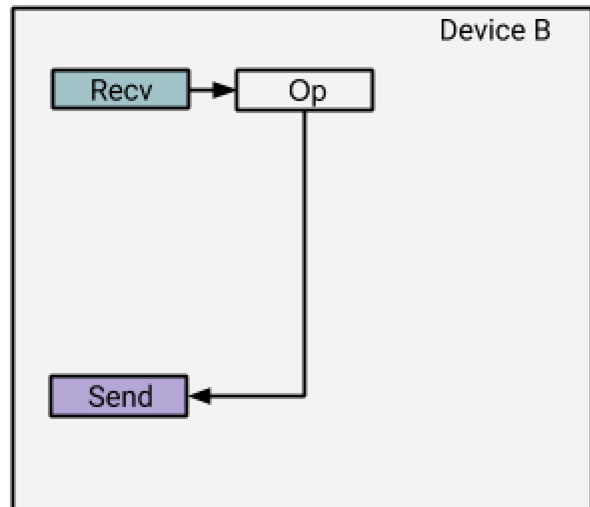


Since any recv node is a source node and can start unconditionally, the recv on device B can start even when it is on the untaken branch of the cond. In order to make the recvs on untaken branch happy, we propagate the is_dead flag across devices from the send to the recv. The propagation may continue on any number of devices. This simple propagation scheme handles distributed execution of nested conditionals, and interacts well with distributed execution of while loops.

Distributed While Loop

For distributed execution, a while loop, in particular the loop body, can be partitioned onto multiple devices. If we naively apply the partitioning scheme of adding send/recv nodes for cross-device edges, the local executors on the devices wouldn't have enough information to run the while loop correctly.



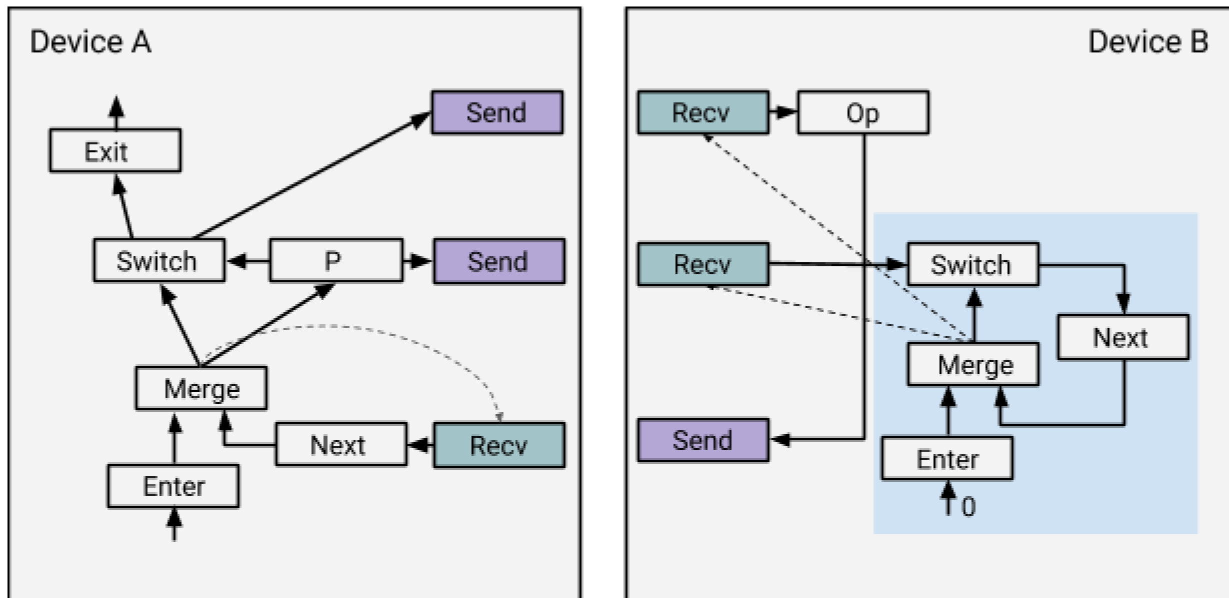


Naive partitioning breaks control flow

Let us use a simple example to illustrate the problems. In the example above, Op is in the loop body and is assigned to device B. A naive partitioning would simply break the edge from Switch to Op with a pair of send/recv nodes. However, this would not work since device B would not know the recv and Op nodes are part of a while loop and would terminate the execution after just one iteration. The solution is to rewrite the dataflow graph, adding a

(c) 2017 TensorFlow Authors. TensorFlow design document, last updated 2017/11/1

control-loop state machine (as shown below in the right bottom corner of device B) in every partition. A scalar tensor 0 is used as the input to the Enter node of a control-loop.



These control loops provide enough information to allow the executors on the devices to run independently as before, communicating with each other via the send/recv nodes. Note that the dotted lines are control edges.

In more detail, let's us first look at the base case that the while loop runs only 0 iteration:

- On device A, the execution starts with the nodes Enter, Merge, P, and Switch. Because P is false, the Send connected to Switch would propagate a dead signal to device B, and the Exit also runs, enabling concurrent execution of nodes outside the loop. The Send connected to P would send the boolean tensor False to device B. The Recv can also be executed, waiting for the value from device B.
- On device B, the execution starts with the nodes Enter and Merge. The execution of Merge enables the two Recvs. The Recv for Switch would receive False so Next would get a dead tensor. Next stops the propagation of deadness. The Recv for Op would get a dead tensor so the Send for Op would send a dead tensor back to device A. At this point, device B has no outstanding ops so the execution terminates.
- Back on device A, the Recv for Next gets a dead tensor. The Next runs, and since it stops the propagation of deadness, device A has no outstanding ops and the execution terminates.

Now suppose the while loop runs one or more iterations:

- On device A, since P is True at the first iteration, a real tensor is sent to device B. The Recv is executed, waiting for the value from device B.
- On device B, the control-loop state machine runs and enables the Recvs. The Recv for Op gets a real tensor from device A; the Op is performed and the Send sends a real

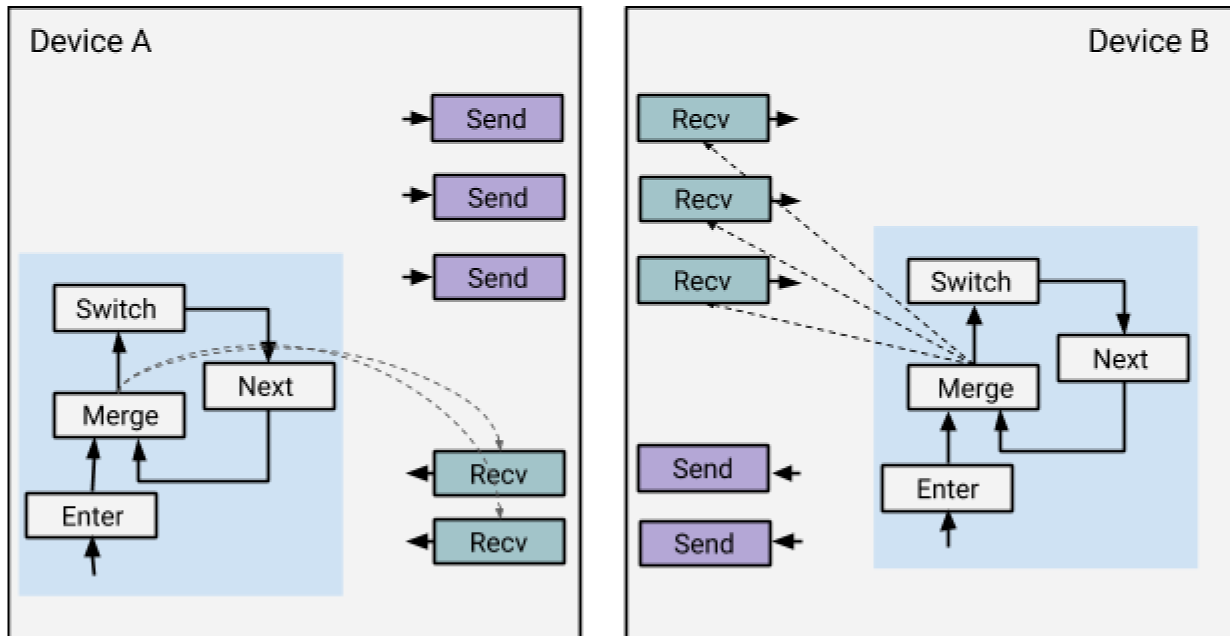
(c) 2017 TensorFlow Authors. TensorFlow design document, last updated 2017/11/1

tensor back to device A. The Recv for Switch gets the boolean tensor True. The Next and Merge are executed, further enabling the Recvs for the next iteration.
- Back on device A, the Recv gets a real tensor. The Next, Merge, and P are executed. Depending on the value of P, either the base case or a new iteration will be executed.

Note that there is a lot of parallelism in the execution. For example, device B can start the next iteration or exit once it receives the value of P. A participating device can have multiple iterations running in parallel, and two participating devices can work on different iterations of the same loop.

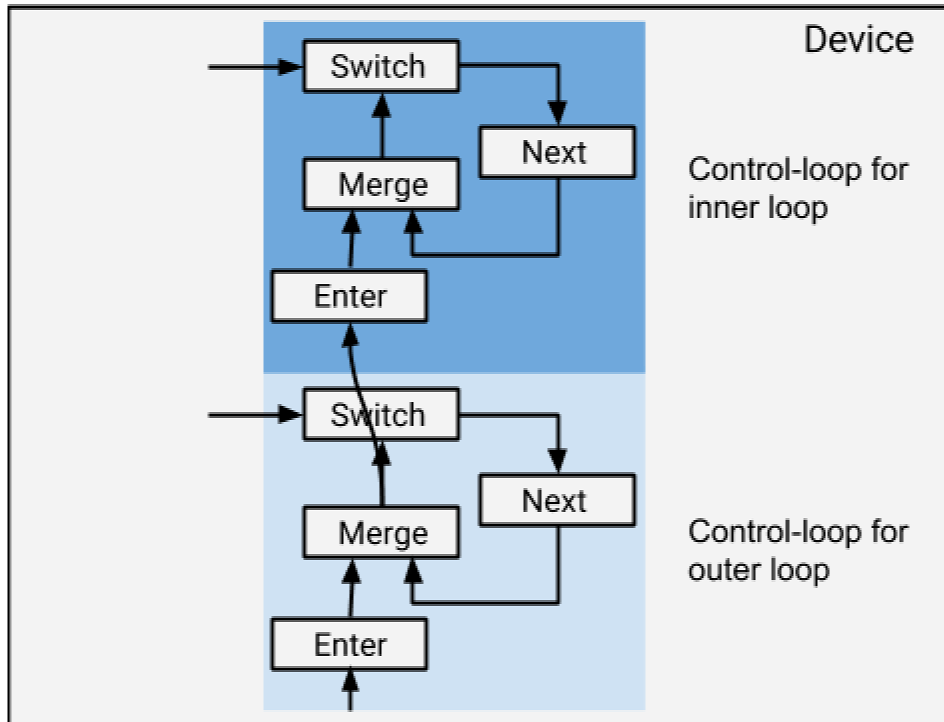
The overhead for the distributed execution of a while loop is that every participating device needs to receive a boolean tensor at each iteration from the device that produces P. Given the parallelism in the execution, the overhead should be largely hidden.

The following shows what the dataflow graph looks like when a while loop is partitioned across multiple devices. A control-loop is added to each partitions and controls the Recvs inside the while loop. The graph after rewriting is semantically equivalent to the original graph.



Graph partitions after rewriting

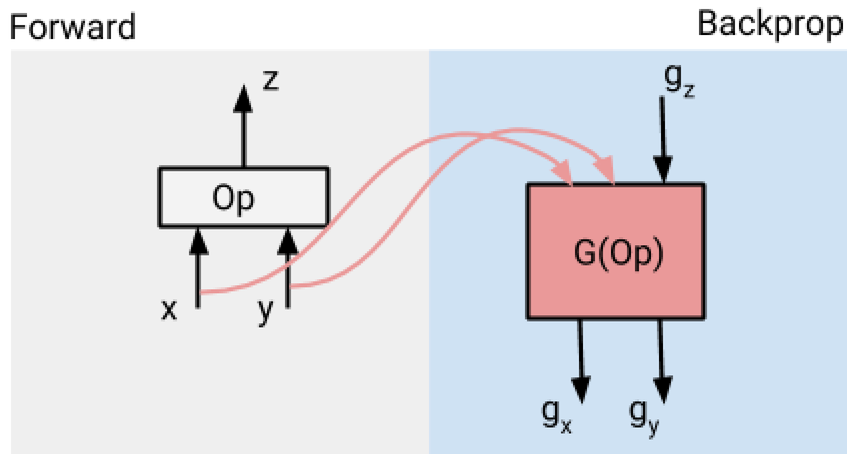
For nested while loops, we just stack the control-loops as follows. Note that if a device only has nodes of the outer loop, we don't add the control-loop for any inner loop on that device.



Automatic differentiation

TensorFlow supports automatic differentiation. A user can, for example, define a neural network with a loss function, and TensorFlow will automatically derive and build the backpropagation dataflow graph. This section explains how TensorFlow automatically builds the backpropagation graph in the presence of cond and while_loop. We assume that the reader has some understanding of how automatic backpropagation works. (See http://colah.github.io/posts/2015-08-Backprop/ for an excellent article on backpropagation.)

The backpropagation algorithm traverses the ops in the forward graph in reverse order, and constructs the gradient graph incrementally by calling the gradient functions of the ops. The gradient function of an op defines the subgraph that computes the symbolic gradient of the op. A gradient function may use the input/output values of the op, so some tensors produced in the forward computation will be kept around for awhile until it is used in the backprop. For example, the following shows a forward op and its gradient graph. G(Op) is the gradient subgraph of Op. The values of x and y will be kept in memory until G(Op) is executed.

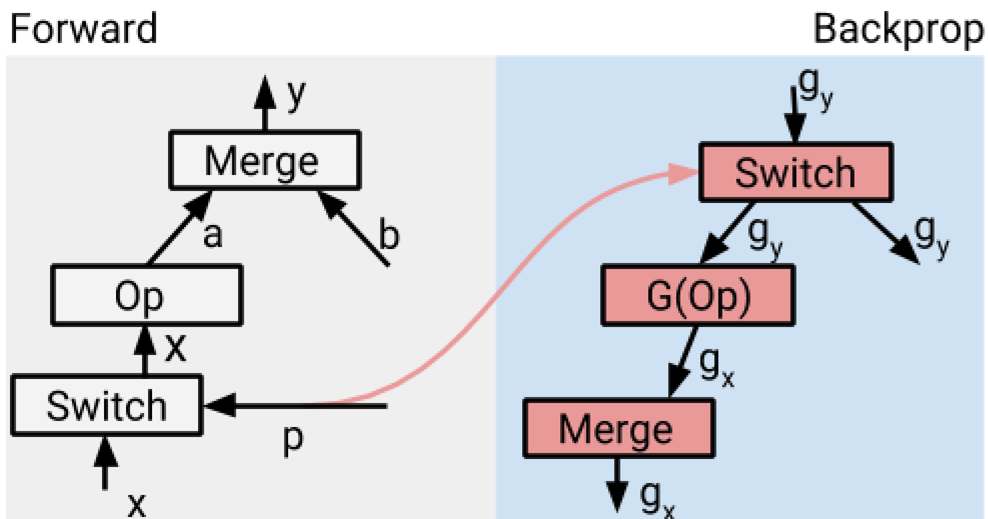


Once the entire dataflow graph is constructed, the TensorFlow runtime automatically partitions the graph and distributes the execution on multiple devices. So the gradient computation in TensorFlow will also be distributed to run on multiple devices.

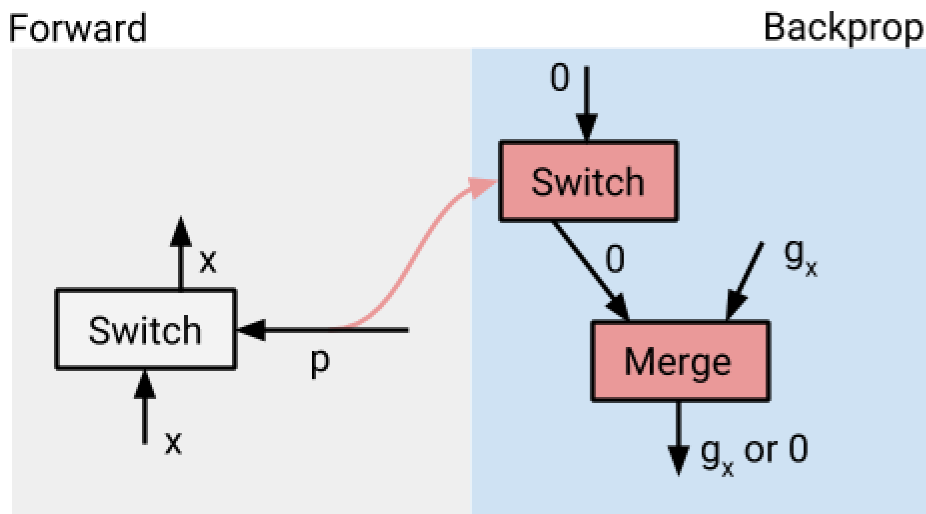
Intuitively, in the context of our high-level constructs of cond and while_loop, the backpropagation of the control flow operators is just to reverse the flow in the following way: The gradient of Exit is Enter; the gradient of Switch is either Merge (for cond) or NextIteration followed by Merge (for while_loop); the gradient of Merge is Switch; the gradient of NextIteration is Identity; and the gradient of Enter is Exit. TensorFlow supports the backpropagation of nested conditionals and while loops.

Backpropagation of Conditional

Intuitively, the gradient of `cond(p, fn1, fn2)` is `cond(p, g_fn1, g_fn2)` where `g_fn1` and `g_fn2` are the gradients of `fn1` and `fn2` respectively. The following shows the basic backpropagation of cond when cond is not nested in a while loop. We assume that Op is on the true branch of the cond. A cond nested in a while loop requires more work to remember the value of p for every iteration of the forward loop. We will get to it later when we look at the backprop of a while loop.



The forward Merge is turned into a Switch, which uses the same predicate p as the forward Switch. The gradient g_y is backpropped to both branches of the Switch. The forward Switch is turned into a Merge. If only one branch of a froward Switch is used in the forward, we add a zero, as shown below, to ensure that there is always a live gradient flowing through the Merge in the backprop. The zero is guarded by a Switch so it will only be sent to the Merge when p is false.

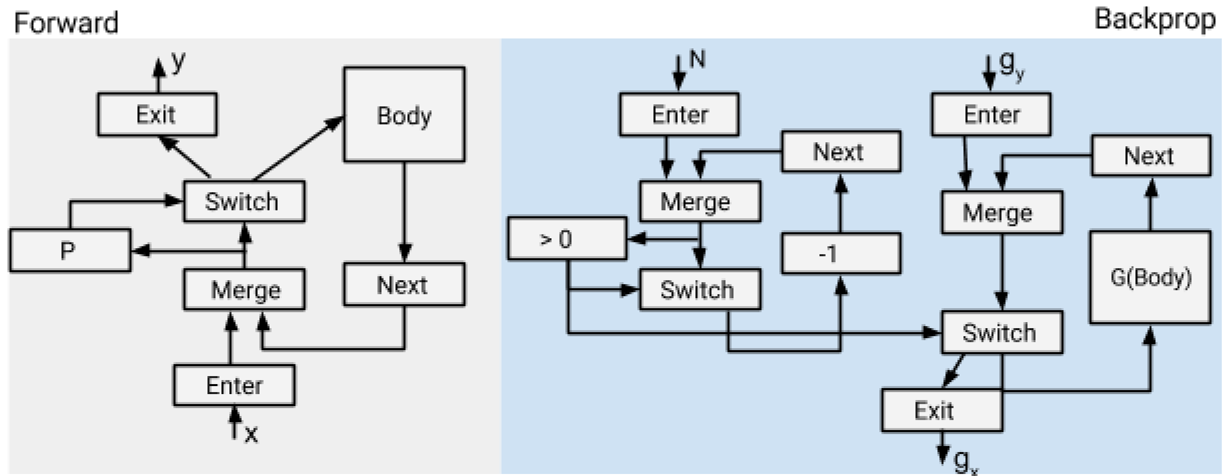


Backpropagation of While Loop

Intuitively, the gradient of `while_loop(pred, body)` is just a while loop of the form:

```
def pred(i, _): return i < N
while_loop(pred, g_body, [0] + g_vars)
```

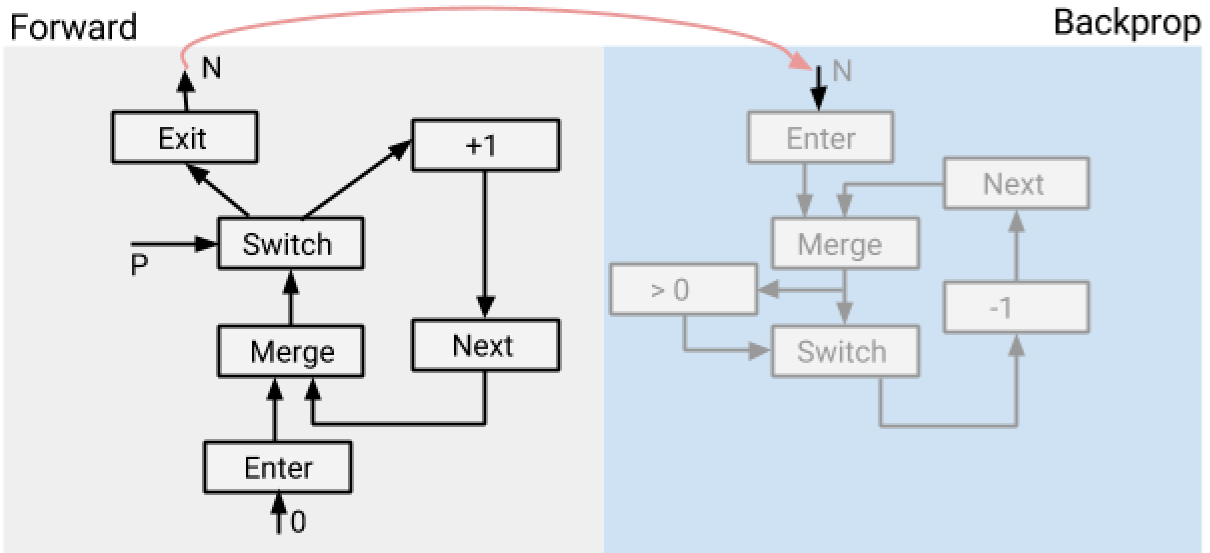
Where `N` is the number of iterations that the forward while loop runs, `g_body` is the gradient of the forward loop body, and `g_vars` is the initial values for the loop variables. As we will see later, `g_vars` includes the initial gradients for the loop variables of the forward while loop. The following is roughly what the graph of a while loop and its backprop while loop looks like:



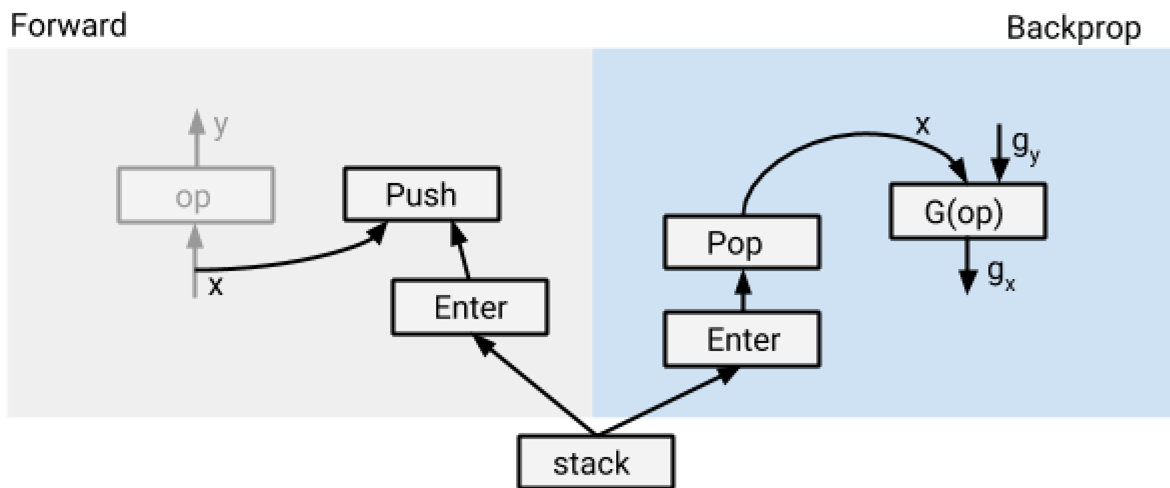
Note that the backprop loop is controlled by N, the number of iterations the forward loop will run. This means that we assume that `pred` is not trainable. G(Body) is the gradient of Body. Body may again contain while loops so this construction may occur recursively to handle nested while loops.

This description so far is rather a big oversimplification. For example, N is not known statically at the graph construction time. More importantly, G(Body) may use values produced by the forward loop body and we want to keep these values around so to avoid recomputing them in the backprop. The solution is to rewrite the graph of the forward while loop to add the logic of computing and/or saving the values needed in the backprop.

To compute N, we add the following subgraph into the forward while loop. So N will be dynamically computed by the forward loop and fed as the initial value of the count loop variable of the backprop loop.



To reuse forward values in backprop loop, we automatically detect, during the construction of the backprop while loop, the forward values that are needed in the backprop. For each such forward value x, we automatically introduce a *stack* and add nodes in the forward loop to save its value at each iteration to the stack. The backprop loop uses the values from the stack in the reverse order. The stack lives outside the forward and backprop loops and is shared by the two loops.

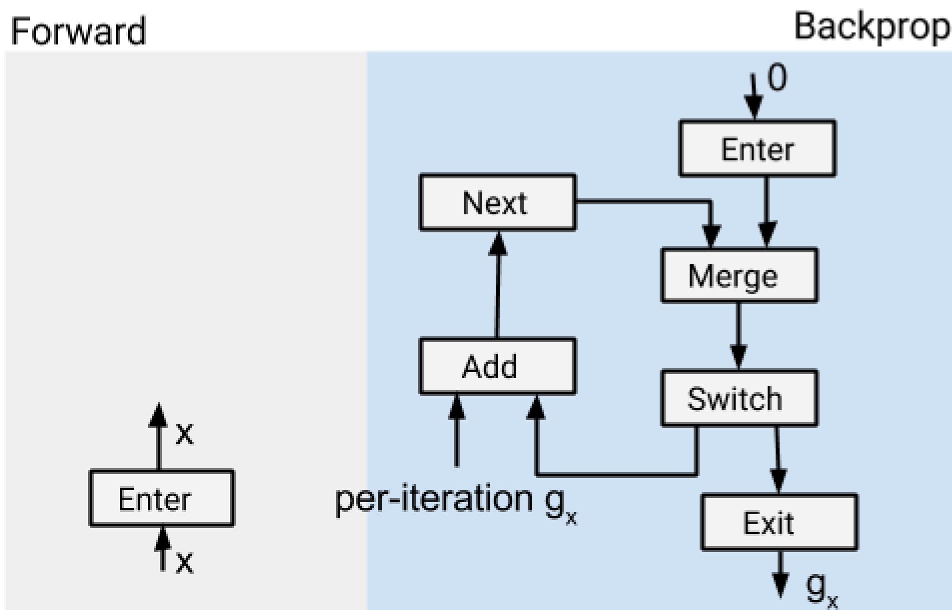


The actual graph construction is actually more subtle and complicated than this. For now let us ignore these details. Here are some of the problems:

- For correctness we ensure that the stack pushes and pops are ordered by the iterations of their respective loops. We also ensure the stack pushes in the forward are ordered before the stack pops in the backprop. The orderings are enforced using control edges.

- For performance we make the stack push and pop operations to be asynchronous so they could run in parallel with the actual computation. For example, op (and even future iterations) can run in parallel with Push.
- If the op is inside a cond nested inside the while loop, the push and pop operations must be guarded properly by the predicate of the cond.
- If the value is immediately reduced by a reduction op (e.g., Shape, Rank, or Size) in the backprop, we move the reduction op to the forward loop to reduce the memory usage.

As described before, the gradient of Enter is Exit. For loop variables, that is all it does. For loop constants, we also add a subgraph to accumulate their gradients, as shown below.



Suppose x is a loop constant in the forward. In the backprop, a partial gradient is generated for x at every iteration. So we add small accumulation subgraph in the backprop to add all of these partial gradients together. The final g_x at Exit is the sum of all the partial gradients. Note that the accumulation is done eagerly, bounded by the number of parallel iterations. This is different from static unrolling, where the use of AddN would require all the partial gradients alive at the same time.

This construction works for both nested conditionals and loops. For a cond nested in a while loop, we introduce a stack to save the value of the predicate at each forward iteration, and use the values (in the reverse order) from the stack in the backprop. For nested loops, this construction is called recursively when we encounter an inner while loop nested in the loop body.

One important optimization is memory swapping. As we have seen, for each forward value v that is needed in backprop, we save its values at all iterations v_1, …, v_N in a stack so we would reuse them in the backprop. This can be a limitation for training on devices such as GPUs

(c) 2017 TensorFlow Authors. TensorFlow design document, last updated 2017/11/1

where memory is limited. We use memory swapping to asynchronously move the values stored in the stacks from GPU to CPU, and move them back in GPU memory when they are needed in backprop.