RedactedSystem: A multi-stage, Python-embedded DSL for machine learning

Anonymous Authors[1]

Abstract

RedactedSystem is a multi-stage, Python-embedded domain-specific language for hardware-accelerated machine learning, suitable for both interactive research and production. TensorFlow, which RedactedSystem extends, requires users to represent computations as dataflow graphs; this permits compiler optimizations and simplifies deployment but hinders rapid prototyping and run-time dynamism. RedactedSystem eliminates these usability costs without sacrificing the benefits furnished by graphs: It provides an imperative front-end to TensorFlow that executes operations immediately and a JIT tracer that translates Python functions composed of TensorFlow operations into executable dataflow graphs. RedactedSystem thus offers a multi-stage programming model that makes it easy to interpolate between imperative and staged execution in a single package.

1 Introduction

Many contemporary libraries for machine learning share a similar structure: they provide suites of primitive operations and functions to automatically differentiate compositions thereof (see, e.g., Bergstra et al.; Tokui et al., 2015; Maclaurin et al., 2015; Chen et al., 2015; Abadi et al., 2016; Paszke et al., 2017; The Gluon Team, 2017; Neubig et al., 2017; Innes, 2018; Frostig et al., 2018). These software packages in fact more closely resemble domain-specific languages (DSLs) than libraries (Innes et al., 2017). Indeed, models written using automatic differentiation software are often referred to as *differentiable programs*.

DSLs for differentiable programming are usually embedded in a host language (for a reference on embedded DSLs, see Hudak, 1996), and they can be roughly classified as either *imperative* or *declarative*, in the programming languages sense. Programming in an imperative DSL for differentiable programming is like programming in an imperative programming language such as Python: the construction and execution of primitive operations are inextricably tied, with each operation returning concrete numerical data. While imperative DSLs provide a natural programming paradigm, when embedded in an interpreted language like Python—which is the case for popular DSLs like Chainer (Tokui et al., 2015) and PyTorch (Paszke et al., 2017)—performance is bottlenecked on the interpreter and serialization of models is difficult. To address these problems, declarative DSLs separate the specification of models from their execution. These "define-before-run" libraries require users to *stage* their models as dataflow graphs, permitting compiler optimizations and the exploitation of parallelism, and simplifying deployment, distribution, and code generation (see, e.g., Bergstra et al.; Abadi et al., 2016). But, because declarative DSLs prevent users from using arbitrary host-language constructs, they have steep learning curves and are not suitable for expressing models with data-dependent structures.

An ideal DSL would offer the flexibility and accessibility of imperative execution along with the many benefits of declarative programming, without either of their costs. It is with this motivation in mind that we present RedactedSystem, a Python-embedded DSL for differentiable programming that lets developers interpolate between imperative and staged computations in a single package. RedactedSystem offers a *multi-stage programming* model that lets users rapidly prototype programs and selectively stage parts that they wish to accelerate or serialize. It is implemented as an opt-in extension to TensorFlow, and it can be enabled by calling a single TensorFlow library function at program start-up.

To empower machine learning practitioners and researchers to be productive from the start, RedactedSystem executes imperatively by default. To reap the benefits of dataflow graphs, RedactedSystem provides a Python decorator that traces its Python function in a graph-building context, staging primitive operations to construct a dataflow graph with named inputs and outputs and returning an executable *graph function*. While invoking a graph function is syntactically equivalent to calling the Python function from which it was generated, the execution of graph functions bypasses

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Systems and Machine Learning (SysML) Conference. Do not distribute.

Python: they are executed using a C++ dataflow runtime or are compiled to generate optimized code for CPUs, GPUs, and ASICs. Graph functions and imperative code share a lexical environment, making it simple to invoke graph functions from imperative code, create graph functions that close over imperatively constructed data, and embed imperative code in graph functions via unstaging annotations.

Our contributions are two-fold:

- Our implementation is elegant. RedactedSystem can be viewed as a multi-stage front-end to TensorFlow. Imperative and staged RedactedSystem code share a single set of primitive operations, kernels, and user-visible APIs. Not only does this sharing result in an easy-to-maintain implementation, it also lets us present a single, coherent API surface to our users that is agnostic to execution mode and lets users enjoy the rich ecosystem of tools developed for TensorFlow.
- While we are not the first in the differentiable programming community to recognize the value in bridging imperative and declarative programming, we are among the first to present this line of work in the context of multi-stage programming. This contextualization is a contribution insofar as it clarifies discourse and connects two otherwise separate communities.

The remainder of this paper is structured as follows: section 2 surveys related work; §3 puts forth our design principles, which prioritize usability and researcher productivity; §4 presents our mutli-stage programming model, with details on automatic differentiation, state, hardware acceleration, distribution, staging, and unstaging; §5 discusses our implementation; and §6 provides a quantitative evaluation of the performance of RedactedSystem on machine learning models, demonstrating that imperative RedactedSystem can train a ResNet-50 on a single GPU just as quickly as TensorFlow can, staged RedactedSystem can train a ResNet-50 on a TPU much faster than imperative RedactedSystem can, and that staging yields significant speedups for models with small operations, all with minimal code changes.

2 Related Work

In RedactedSystem, users must manually stage computations, which might require refactoring code (see §4.1). An ideal framework for differentiable programming would automatically stage computations, without programmer intervention. One way to accomplish this is to embed the framework in a compiled procedural language and implement graph extraction and automatic differentiation as compiler rewrites; this is what, e.g., DLVM and Swift for TensorFlow do (Wei et al., 2017; Lattner & the Swift for TensorFlow Team, 2018). Python's flexibility makes it difficult for DSLs embedded in it to use such an approach. Some projects, like AutoGraph (Wiltschko et al., 2018) do operate on Python abstract syntax trees to rewrite imperative code to code that constructs dataflow graphs, but such techniques are out of the scope of this paper.

An alternative to staging computations as graphs for performance is to implement fused kernels. For example, NVIDIA provides fused CuDNN kernels for popular recurrent neural network operations that are dramatically faster than non-fused implementations (Chetlur et al., 2014). This approach, while useful, is difficult to scale, as it requires substantial programmer intervention.

RedactedSystem is not the first Python library to offer a multi-stage programming model, though we believe our paper is the first in the differentiable programming community to call it as such. JAX (Frostig et al., 2018), a tracing-JIT compiler that generates code for heterogeneous devices via XLA (The XLA team, 2017), provides a similar programming paradigm; MXNet and Gluon also let users interpolate between imperative and staged computations, but at a level of abstraction that is higher than ours (Chen et al., 2015; The Gluon Team, 2017); and PyTorch is implementing a staging tracer that is appears to be similar to ours (PyTorch team, 2018). Outside of differentiable programming, Terra is a Lua-embedded DSL that supports code generation, and the paper in which it was introduced presents a thorough treatment of multi-stage programming that is more formal than ours (DeVito et al., 2013); as another example, OptiML is a Scala-embedded DSL for machine learning with support for staging and code generation but without support for automatic differentiation (Sujeeth et al., 2011). Outside of DSLs, there are several projects that provide just-in-time (JIT) compilation for Python, of which Numba (Lam et al., 2015) and PyPy (Bolz et al., 2009) are two examples.

Multi-stage programming is a well-studied topic in programming languages; a good reference is (Taha, 2004), and a modern design from which we drew inspiration is Scala's lightweight modular staging (Rompf & Odersky, 2010). Multi-stage programming is related to staging transformations in compilers and partial evaluation in programming languages, for which (Jørring & Scherlis, 1986) and (Jones et al., 1993) are classic references, respectively.

3 Design Principles

Our design strives to satisfy two goals: RedactedSystem should be immediately recognizable to Python programmers—for example, users should feel at home exploring APIs and prototyping models in IPython notebooks—and it should also provide a smooth path to testing ideas at scale and deploying models for inference on heterogeneous devices. The first two of the following three principles are

in service of the former goal, while the third is in service of the latter.

Privilege imperative execution. Because Python is an imperative language, RedactedSystem operates in an imperative fashion by default; staged execution is opt-in and often unnecessary (see §4.1 and §6 for details).

Seamlessly embed into Python. Whereas writing TensorFlow code is an exercise in metaprogramming, imperative execution lets programmers enjoy the full extent of the host language: programmers write Pythonic code, complete with familiar language constructs like native control flow (e.g., Python `if` statements and `while` loops), recursion, arbitrary data structures, and even `pdb` breakpoints. And, because we implement automatic differentiation via tracing (§4.2), the programmer can differentiate through all these constructs. Host-language integration is more than just syntactic sugar—it greatly simplifies the implementation of data-dependent models like segmental recurrent neural networks and recursive neural networks (Kong et al., 2015; Socher et al., 2011).

Stage imperative code as dataflow graphs. To leverage the benefits of dataflow graphs, RedactedSystem provides a mechanism to trace Python functions and stage their operations as graph functions. The staging workflow is detailed in §4.1, and the mechanism is described in §4.6. TensorFlowgraphs come with their own set of design principles, which are presented in (Abadi et al., 2016).

4 Execution Model

This section presents the pillars of RedactedSystem's execution model. §4.1 describes imperative and staged execution, presenting a workflow that hybridizes the two; §4.2 describes our trace-based implementation of automatic differentiation; §4.3 specifies how we represent mutable state and how we support serialization; §4.4 details how RedactedSystem supports execution across heterogeneous devices; §4.5 presents mechanisms for distributed execution; §4.6 discusses our tracing JIT in detail; and §4.7 discusses mechanisms for escaping staged computations.

The following terminology will be used in the sequel: a *tensor* is a multi-dimensional, typed array, an *operation* is a primitive, possibly stateful function that takes tensors as inputs and produces tensors as outputs, a *kernel* is a device-specific implementation of an operation, and a *model* is a composition of primitive operations.

4.1 Multi-stage programming

RedactedSystem provides two ways of executing operations: imperatively or as part of a static dataflow graph. Both execution models have access to the same set of operations and kernels, but they differ in how they dispatch kernels.

Imperative execution. By default, RedactedSystem executes operations immediately—library functions such as `tf.matmul` construct operations and then immediately execute their kernels. Under this regime, RedactedSystem resembles a NumPy-like library for hardware-accelerated numerical computation and machine learning. Calling `.numpy()` on a tensor fetches a NumPy array storing the tensor's data, and tensors can be supplied to external libraries like matplotlib that expect NumPy arrays (for a reference on NumPy, see Oliphant, 2015). As an example,

```
import tensorflow as tf
tf.enable_redacted_system()

def select(vector):
  A = tf.constant([[1.0, 0.0]])
  return tf.matmul(A, vector)

x = tf.constant([[2.0], [-2.0]])
print(select(A, x))
```

prints

```
tf.Tensor(
[[ 2.]], shape=(1, 1), dtype=float32).
```

Staged execution. While imperative execution simplifies prototyping, the overhead of going back and forth into the Python interpreter limits its performance; representing computations as dataflow graphs before executing them not only removes this bottleneck but also allows for inter-op parallelism and optimizations like constant-folding and buffer reuse. Thus, RedactedSystem provides a mechanism to *stage* computations as dataflow graphs. In particular, we provide a decorator, `defun` (short for "define function"), that traces the execution of a Python function, recording all TensorFlow operations and the tensors flowing between them in a dataflow graph. `defun` can be thought of as an opt-in, JIT compiler that generates an optimized polymorphic function for a Python function, creating concrete functions backed by dataflow graphs via a straightforward binding-time analysis at run-time. The analogy to compilers is imperfect because the traces generated by `defun` only record TensorFlow operations and not arbitrary Python code, but it nonetheless provides an approximate mental model. One advantage of this tracing mechanism is that the underlying dataflow graph format does not need to support all the dynamism present in the Python code being traced; as long as the set of operations in the trace does not depend on Python state we can generate a correct trace.

Invoking a callable returned by `defun` will execute a dataflow graph instead of the corresponding Python function. In fact, graph functions are themselves executed by a primitive operation that takes tensors as inputs and a function name as an attribute, and these operations are automatically

constructed and executed for the user. For example, if the `select` function defined in the previous section were decorated with `@defun`, then `select(A, x)` would execute an operation that would in turn execute the appropriate graph function. The dataflow graph runtime, which is written in C++, automatically partitions subgraphs across devices and parallelizes operations when possible. Readers interested in the runtime should consult (Abadi et al., 2016) for more information.

`defun` supports code generation via XLA (The XLA team, 2017). RedactedSystem relies upon XLA to execute code on Tensor Processing Units (TPUs) (Sato et al., 2017) (see §4.4). In addition to performance and hardware acceleration, dataflow graphs simplify distribution (§4.5) and deployment. Details about the mechanism of `defun` are provided in §4.6.

A multi-stage workflow. Many users will find the performance of imperative execution sufficient. Purely imperative RedactedSystem can match the performance of graph execution when training models with sufficiently expensive kernels, like ResNet-50 (He et al., 2016) (see §6). But when imperative performance disappoints, we recommend the following multi-stage workflow, modeled after (Taha, 2004).

1. *Implementation.* Develop, debug, and test a single-stage imperative program.

2. *Analysis.* Using any profiling tool the user is familiar with, identify performance-critical blocks of operations, and express these blocks as staging-friendly Python functions or callable objects.

3. *Staging.* Decorate the functions identified in the previous step with `@defun`.

With respect to the analysis step, the key fact to keep in mind is that `defun` is *not* a compiler for arbitrary Python code. Rather, it is a JIT tracer that executes Python functions in a graph-building context and only records operations and tensors. In a graph-building context, operations return symbolic representations of values to be computed instead of concrete values, and non-TensorFlow Python code executes normally. Python functions that are amenable to staging are those that, when called in a graph-building context, generate a graph that encapsulates the computation of interest. This means that if a Python function executes non-TensorFlow code, then there might be semantic discrepancies between executing the Python function and executing the traced dataflow graph. For example, whereas the python function

```
def add_noise():
  eye = tf.eye(5)
  randn = np.random.randn(5, 5)
  return eye + randn
```

will return a different output every time it is invoked, the dataflow graph generated by `defun(add_noise)` will return the same value every time it is called, since a particular random offset generated by NumPy will be inserted into the graph as a constant. Note that if state is represented in terms of *operations* (e.g., if we replace the call to `np.random.randn` with `tf.random_normal`), we can preserve semantics under this tracing model. As a corollary, if a Python function `f` has Python side-effects (e.g., every call to it increments a global Python counter), then executing it multiple times will not necessarily be semantically equivalent to repeatedly executing the callable returned by `defun(f)`. Python functions must also be resilient to being executed multiple times, as the callable returned by `defun` might trace its Python function multiple times (see the discussion on polymorphism in §4.6).

Because `defun` generates graphs by tracing and not by source code analysis, it fully unrolls Python `for` and `while` loops, potentially creating large graphs. If that is a problem, the programmer might need to replace their loops with the equivalent TensorFlow control flow constructs. Similarly, the branches of `if` statements that are taken during tracing are baked into the emitted graphs. Conditionals that depend on the value of tensors will need to be written using `tf.cond`, and `while` loops that depend on tensor values will need to be rewritten in terms of `tf.while_loops`. Python functions that depend on the values of tensors in complicated ways (e.g., via data structures that depend on the values of tensors) might prove to be prohibitively difficult to stage correctly. In such cases, users might need to refactor their functions into staging-friendly and staging-unfriendly helper functions (see the discussion on escaping staged computations in §4.7 for other options).

Note that staging trades off imperative execution (and therefore interactivity) and Python integration (and therefore run-time dynamism) for performance. It is up to the programmer to decide when this trade-off is acceptable and to use staging annotations judiciously. This trade-off can be diminished by using tools like AutoGraph that operate on abstract syntax trees and rewrite Python control flow to dataflow control flow (Wiltschko et al., 2018).

4.2 Automatic differentiation

We implement a variant of tracing-based reverse-mode automatic differentiation (Baydin et al., 2018), with a few changes to better support partially staged computation. Our implementation is similar to the implementations of Chainer (Tokui et al., 2015), Autograd (Maclaurin et al., 2015), and PyTorch (Paszke et al., 2017), but our API allows for more fine-grained control over which computations are traced.

The main user-visible concept in the gradient API is a *tape*. If a tape *watches* a value, operations taking this value as

an input will be recorded. It is possible to differentiate any scalar that is computed while a tape is active with respect to any watched value. Tapes are composable data structures: multiple tapes can be active simultaneously, and higher-order gradients can computed by having one tape recording while another tape computes a gradient. Listing 1 gives an example of nesting tapes to compute a second derivative.

```
x = tf.constant(3.0)
with tf.GradientTape() as t1:
  with tf.GradientTape() as t2:
    t1.watch(x)
    t2.watch(x)
    y = x * x
  dy_dx = t2.gradient(y, x) # 6.0
d2y_dx2 = t1.gradient(dy_dx, x) # 2.0
```

Listing 1. Tapes can be nested to compute higher-order derivatives.

Exposing the tape directly (as opposed to high-level Autograd-like gradient functions) lets users control which parts of the computation are traced for automatic differentiation, which can help limit the run-time overhead incurred in the tracing process.

The tape is tightly integrated with the logic responsible for staging code. The first time a graph function is called when a tape is both active and watching one of its inputs, we build a "forward" version of this function that returns any intermediate values needed for the backward step, in addition to its named outputs. As such, there is no meaningful change in the amount of computation or memory needed in the backward pass by staging or unstaging a particular function, leading to more predictable performance. Moreover, this ensures that if a computation was staged in the forward pass, its corresponding backward pass will also be staged.

Note that gradient computation is itself expressed as a function which executes primitive operations, so it is possible to stage it or not.

4.3 State

Like TensorFlow, RedactedSystem keeps program state in *variables*, restoring a variable's value by assigning to it from a restore operation and periodically saving it to disk by sending its value to a save operation. Variables are useful when implementing models because accessing a variable's value automatically watches it on all active tapes, as shown in Listing 2.

```
x = tf.Variable(3.0)
with tf.GradientTape() as t1:
  with tf.GradientTape() as t2:
    y = x * x
  dy_dx = t2.gradient(y, x) # 6.0
d2y_dx2 = t1.gradient(dy_dx, x) # 2.0
```

Listing 2. Gradient tapes automatically watch variables; compare this code to Listing 1

In RedactedSystem, variables correspond to Python objects. Each variable object has its own unique storage that is deleted when Python deletes the object. This is true even for traced computations, where staged *read*, *write*, *save*, and *restore* operations may interact with variables. Staged computations reference variables by unique identifiers, which are no longer usable if the Python variable objects they reference do not exist. This correspondence ensures that RedactedSystem state conforms to programmer expectations, stored like any other Python state and accessible through Python identifiers.

One challenge when moving from purely staged computation to keeping state in Python objects is matching state between executions of the same program. TensorFlow uses unique names for each variable in a program, which relies on the user creating variables in a consistent order. For example creating two copies of the same model requires special consideration when restoring the second model. RedactedSystem uses a graph-based matching system, where a directed graph with named edges between objects is serialized along with the program state. On restore, a greedy matching determines a correspondence between serialized Python state and the objects being restored. This matching is local in that it depends only on the objects being saved and restored, not on other parts of the program. Listing 3 and Figure 1 contain a short example.

```
class Net(tf.keras.Model):
  def __init__(self):
    super(Net, self).__init__()
    self.v = tf.Variable(1.)
    self.out = tf.layers.Dense(1)

  def call(self, x):
    return self.out(
      tf.nn.softplus(x * self.v))
```

Listing 3. Model-building code which implicitly constructs a graph with named directed edges (from attribute names), used for state matching.

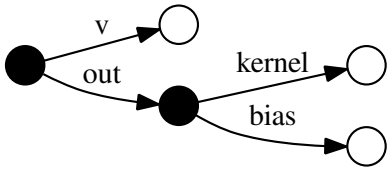


Figure 1. Visualization of the dependency graph for Listing 3, with filled-in intermediate nodes and nodes without fill containing state.

Variables are the most common type of state, but other state

is similarly scoped to a Python object and matched as part of a directed graph with named edges. Examples include an iterator over input data whose position in a dataset is serialized, mutable hash tables, and outside of traced code even miscellaneous Python state such as NumPy arrays can use graph-based state matching.

Staging enables serializing the program for use without a Python interpreter, as in TensorFlow. A typical development workflow involves using graph-based state matching while writing and tweaking a RedactedSystem program, then serializing a trace for use in a production environment that executes the trace using TensorFlow's C++ API.

4.4 Devices

RedactedSystem makes it simple to use a variety of devices, such as CPUs, GPUs, and TPUs. During program startup, the runtime detects the devices that are available to the machine, and makes it possible to both execute operations on them and store data on them. Imperative and staged computations use the same underlying `Device` abstraction, which makes it possible to both execute operations on devices and store data on them. A user-visible API endpoint `list_devices` is exposed which lists all devices that the runtime is aware of.

All tensors exposed to the user are handles to data stored on a particular device. The runtime is also aware of how to copy data between various types of devices, and exposes this functionality through API endpoints on tensor instances.

```
a = tf.constant(1.0) # stored on CPU
b = a.gpu() # stored on GPU
```

Listing 4. Tensor copies between CPU and GPU

When executing an operation, the runtime expects to have a specific device to run the operation on. RedactedSystem exposes a context manager, `device`, so that the user can control which device operations execute on. The user is not required to use this API, as the runtime is able to select a device based on the availability of kernels. When an operation has inputs on devices different from the device where the operation is executing, the runtime transparently copies the inputs to the correct device. This frees the user from having to explicitly copy tensors between various devices.

```
# stored on CPU
a = tf.constant(1.0)
b = tf.constant(2.0)

with tf.device("/gpu:0"):
  c = tf.add(a, b)

assert c.numpy() == 3.0
```

Listing 5. Executing a GPU operation with inputs on the CPU

Because graph functions are executed via a primitive operation, it is also possible to use the `device` context manager to run graph functions on various devices. If operations inside the graph function are explicitly placed on another device, they override the outer device context.

Graph functions can serve as a unit of compilation for accelerators; we use this to efficiently execute code on TPUs. When a staged computation is placed on a TPU, RedactedSystem automatically invokes XLA to compile the graph and produce a TPU-compatible executable. RedactedSystem does make it possible to execute code imperatively on TPUs, but the overhead of compiling operations for TPU and dispatching the generated code is significant. When amortized over a large graph function, this overhead becomes negligible (see §6 for a quantitative example). Note that this programming model is similar to JAX (Frostig et al., 2018), which provides a Python decorator that JIT-compiles functions via tracing and XLA. Finally, compiling staged computations through XLA provides us more opportunities for optimization, including layout optimization, instruction scheduling for concurrency, and operation fusion. Techniques like tensor re-materialization can make it possible to fit a staged model into TPU memory when it would be impossible to do so on an operation-by-operation basis.

4.5 Distribution

The current system supports distributed execution with a single central server running the main (typically Python) program and several worker servers running on remote hosts. Each worker server adds its locally available devices (for example, CPUs, GPUs, or TPUs) to the pool of devices available to the main program. The main program can then execute operations or whole graph functions on remote devices through the worker servers.

The remote devices are identified by application-level names. The names contain the job name, task inside the job, as well as the specific device available for the task. For example, "/job:training/task:2/device:GPU:0". When a server is brought up to be a part of a cluster, it is given the mapping from the application-level names to specific server instances identified by DNS names or IP addresses.

To run an operation on a remote device, the user uses the same syntax as for local devices (see 4.4) but uses a remote device name instead of the local device name. Tensors produced as result of running an operation on remote device stay on the remote device. Users can then either perform more operations on these tensors or copy them to the central server (e.g. to use their value in an `if` statement).

Some computations running on remote devices can directly communicate and synchronize between each other. In such cases, developers need to start these computations concur-

rently, e.g. using Python threads.

4.6 Staging computations

The particular type of staging that RedactedSystem supports is similar to lightweight modular staging (Rompf & Odersky, 2010), which in turn is a form of partial evaluation (Jones et al., 1993). As stated in §4.1, we expose a user-visible API endpoint named `defun` that takes a Python function and returns an object which, when called, executes a dataflow graph created by running the user-provided python function in a graph-building context. In this section, we discuss the implementation of `defun` in detail.

Polymorphism. All Python functions are polymorphic in their inputs. In contrast, graph functions are *not* polymorphic: their graphs have a fixed number of inputs, and all these inputs are statically typed. We bridge this semantic gap between Python functions and graph functions by implementing a trace cache, similar to the one described in JAX (Frostig et al., 2018). `F = defun(f)` maintains a cache mapping from inferred input signatures to concrete graph functions. In particular, each time `F` is invoked, its inputs are processed and their signature is inferred: tensors are represented as abstract types (numerical type and shape tuples), while non-tensor values are encoded by object identity. This input signature, coupled with a small amount of metadata about the surrounding program state such as the requested device, becomes a key into a cache of graph functions. A cache miss triggers a trace of `f` on the given inputs, while a cache hit results in the reuse of a previously created graph function. In this sense, `defun` provides ad hoc polymorphism (Strachey, 2000) or function overloading.

Not only is specializing functions on input types required for correctness, it also lets us generate optimized graphs optimized — this kind of optimization is well-known, and, indeed, one of the primary motivations for partial evaluation (Jones et al., 1993; Taha, 2004; Rompf & Odersky, 2010).

Like JAX, `defun` specializes on the run-time values of non-tensor arguments to let them parameterize the computation (`defun` specializes automatically, for convenience, whereas JAX makes this process a manual one). For example, it is common to write Python functions that take a boolean `is_training` argument that determines whether or not dropout is applied. Our implementation of binding-time analysis ensures that graph functions are specialized on the value of the boolean argument (see listing 6 for an example).

```
@redacted.defun
def lossy_matmul(W, x, training=True):
  outputs = tf.matmul(W, x)
  if training:
    outputs = tf.nn.dropout(outputs, 0.2)
  return outputs

W = tf.random_normal((3, 5))
x = tf.random_normal((5, 1))
# Executes a graph with dropout.
lossy_outputs = lossy_matmul(W, x,
  training=True)
# Executes a graph without dropout.
exact_outputs = lossy_matmul(W, x,
  training=False)
```

Listing 6. This code transparently makes two graph functions.

The user also has the option of specifying an input signature to eliminate input polymorphism. In this case, we guarantee that we only generate a single graph function using only the shape and numeric type information specified in the signature. This can be useful for serialization and error-checking, and for creating a single function that can handle arbitrary batch sizes or sequence lengths.

Lexical closure. `defun` is capable of tracing Python functions that lexically close over tensors or variables — these closed-over objects are treated as "captured" inputs that are silently passed to the graph function at call-time, without programmer intervention. Variables are captured by reference and not by value, which means that graph functions are free to mutate them. Listing 7 provides an example.

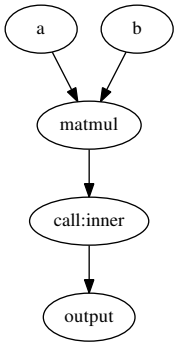
```
v = tf.Variable(0.0)

@redacted.defun
def mutate():
  v.assign_add(1.0)
  return v.read_value()

mutate()
assert float(v.read_value()) == 1.0
v.assign_add(1.0)
assert float(v.read_value()) == 2.0
mutate()
assert float(v.read_value()) == 3.0
```

Listing 7. `defun` transparently captures closed-over tensors and Variables, forwarding them to TensorFlow functions as inputs.

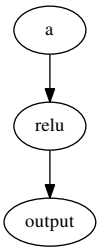
Composition. Because graph function execution is implemented as an operation, graph functions compose naturally: the graph of a function may include a function-call operation that executes another function. For example, consider the following code block:

```
@redacted.defun
def inner(a):
  return redacted.nn.relu(a)

@redacted.defun
def outer(a, b):
  return inner(redacted.matmul(a, b))

outer(tf.eye(3), tf.diag([-1.0, 1.0, 2.0]))
```

Listing 8. Graph functions can be nested.



(a) The graph generated for `outer`; note the `call` operation that executes `inner`'s graph function.



(b) The graph generated for `inner`.

Figure 2. `defun` composes; above, the graphs for Listing 8.

The call to `outer` will generate two graph functions, one for `inner`, and one for `outer` that contains a call to `inner`'s graph function. Figure 2 shows what their corresponding graphs look like.

4.7 Escaping staged computations

Embedding imperative code in graphs. As discussed in §4.1, staging computations requires the programmer to refactor the to-be-staged code into Python functions that, when traced, construct dataflow graphs. This process may at times seem prohibitively difficult, as it can require replacing complicated Python control flow with TensorFlow control flow or even implementing custom operations along with custom C++ kernels — indeed, this observation was one of the motivations for building RedactedSystem to begin with.

For concreteness, say that we have a Python function that we wish to stage, and say that the function is almost entirely staging-friendly with the exception of a call to a data-dependent recursive Python function that performs some operations on tensors. When presented with such an impasse, we have three options: we can refactor the function into three functions, staging the code before and after the recursive call and leaving the recursive call unstaged; we can give up on staging the function if refactoring proves too onerous; or we can stage the entire function while wrapping the recursive call in a `py_func`, an operation that takes a Python function as an attribute and executes it imperatively even in the context of staged code.

`py_func` executes its Python function under a gradient tape (see §4.2) and as such it is differentiable; it also has both CPU and GPU kernels. When executing in imperative mode, wrapping a Python function in a `py_func` has essentially no effect. But, in staged computations, i.e. in dataflow graphs, the `py_func` operation is a way to embed imperative, Pythonic code into a dataflow graph. Equivalently, `py_func` can be viewed as a way to quickly implement custom operations using Python instead of C++.

The benefit of `py_func` is that it makes it easier to decorate large Python functions with `@defun`. Disadvantages include a potential performance hit, as `py_func` returns control to a single-threaded Python interpreter, and the fact that graphs with `py_func`s are not in general serializable.

Escaping traces. When building machine learning models, it is common to write Python functions that special-case their first invocations. For example, the first call to the forward function of a model might create and initialize variables, like the following method:

```
def forward(model, args):
  if not model._built:
    model.W = tf.Variable(
      lambda: tf.random_normal((3, 4)))
    model._built = True
  ...
```

If `model._built` were `False` and `forward` were traced by `defun` in its entirety, the variable initialization operation would be encoded in the generated graph function, i.e., every call to the graph function would re-initialize `model.W`, which would not match the semantics of the Python function. To circumvent this problem, we provide a Python context manager, `tf.init_scope`, that pauses the trace and jumps into the imperative context. Wrapping variable construction in this scope escapes tracing, preventing the initialization operation from being added to the graph function. To simplify the user experience, our high-level APIs that use the create-state-on-first-call idiom automatically escape tracing when required for correctness.

5 Implementation

We have implemented the design presented in §4, and all of our code is open source. Because RedactedSystem was built as an extension to TensorFlow, the implementation is not large: staging is implemented in approximately 2000 lines of Python, automatic differentiation is split across 900 lines of Python and 600 lines of C, and the imperative runtime—i.e., the code responsible for constructing and executing operations—is implemented in approximately 4000 lines of C++. RedactedSystem also provides a lightweight C API that exposes our runtime, and several of our colleagues are using this API directly in their own projects.

RedactedSystem inherits the benefits of the TensorFlow's implementation. In particular, RedactedSystem is cross-platform, running on the Linux, Mac OS X, Windows, Android, and iOS operating systems, and various x86, ARM, and NVIDIA GPU architectures; it executes staged computations using a dataflow executor that can run over ten

thousand subgraphs in parallel and that runs kernels in parallel when possible, across multiple CPU cores or GPU streams; it provides high-level Python APIs for training models and C++ APIs for inference (see Abadi et al., 2016, §5). RedactedSystem also provides access to the over 900 primitive operations that TensorFlow offers.

RedactedSystem and TensorFlow differ slightly but significantly in their implementations of staged execution. In TensorFlow, the dataflow graph defines the union of all the computations that the author of the graph might be interested in; the actual computation to execute is defined when the programmer requests the runtime to fetch the concrete values of some set of tensors resident in the graph. This amounts to a discrepancy between what is expressed in Python and what is executed by the TensorFlow runtime. To provide a more Pythonic programming model, RedactedSystem represents each staged computation as a graph function, i.e., a graph with named inputs and outputs, representing the *exact* computation of interest. Note that this approach does not preclude the runtime from optimizing the computation before executing it: for example, non-stateful operations that are not reachable from the outputs of a function are pruned, just as in TensorFlow.

Graph functions provide benefits outside the realm of usability as well. Because graph functions are executed via an operation, we get function composition for free. In the context of single-coordinator distributed training, in which a single subgraph is executed by N workers, graph functions can reduce memory pressure on the coordinator: the coordinator only needs to own a graph function that contains N function-call operations (instead of N copies of a subgraph).

6 Evaluation

As described above, RedactedSystem considerably simplifies rapid prototyping of models. This at times trades off execution speed for development speed and ease. In this section, we present examples showing how we can leverage `defun` to recover the speed of TensorFlow.

ResNet-50. In Figure 3 we show the performance of training a ResNet-50 model on synthetic data, comparing RedactedSystem, RedactedSystem with the forward pass and gradient application staged with `defun`, and TensorFlow. The top chart shows the raw examples per second, and the bottom chart shows the improvement that RedactedSystem with `defun` and TensorFlow show over RedactedSystem. For smaller batch sizes, staging computations yields significant speed-ups. These speed-ups vanish as the batch size increases, since the ratio of the time spent in kernels over the time spent in Python increases. Additionally, training a ResNet doesn't benefit significantly from inter-op parallelization, so the staged computation is effectively as serial

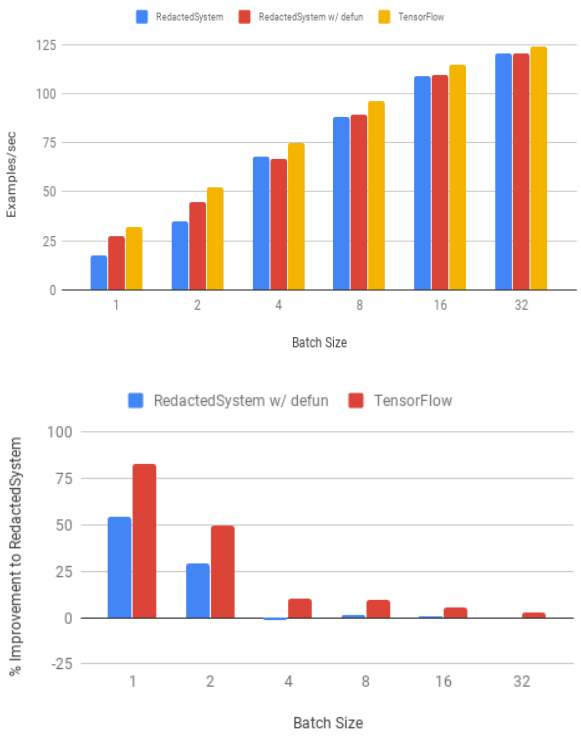


Figure 3. Comparing ResNet-50 running on a GPU with RedactedSystem with `defun` and TensorFlow

as the unstaged computation. These performance characteristics should hold true for other sufficiently large models, i.e., imperative performance will often be similar to staged performance. The code used to generate these benchmarks all rely on the same `Model` class; converting the code to use `defun` is simply a matter of decorating two functions.

ResNet-50 on TPU. It is possible to run single operations on a TPU using RedactedSystem. The performance of training ResNet-50 on ImageNet (Deng et al., 2009) using RedactedSystem versus RedactedSystem with `defun` is shown in Figure 4. Training the model in a per-operation fashion is slow, even at a batch size of 32; however, staging yields an order of magnitude improvement in examples per second.

An important caveat about the benchmarks presented is that they do not exploit the underlying hardware optimally. They are presented as illustrative of the multi-stage programming model that is presented above showing that the concept can be extended to various types of hardware, including TPUs, with practically no code changes. We don't present an accompanying TensorFlow benchmark for this reason.

L2HMC. In Figure 5 we show performance of an L2HMC (Levy et al., 2017) implementation, comparing RedactedSystem, RedactedSystem with `defun`, and TensorFlow on synthetic data running on the CPU. The benchmark samples

	1	2	4	8	16	32
RedactedSystem	1.06	1.99	4.3	8.4	16.6	30.3
RedactedSystem with `defun`	21.7	42.6	83.9	165.8	197.7	241.9

Figure 4. Examples/sec training ResNet-50 on a TPU.

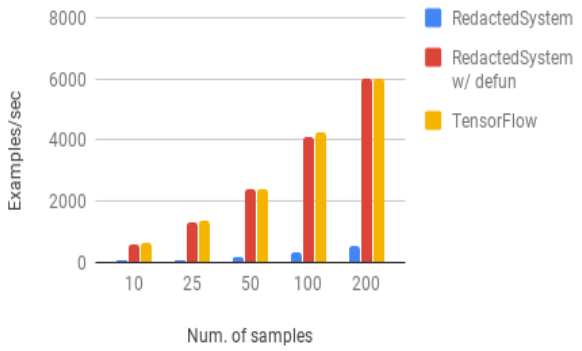


Figure 5. Examples/sec training L2HMC on a CPU.

from a 2-dimensional distribution, with 10 steps for the leapfrog integrator.

This example highlights the trade-off between debuggability and performance: by bypassing Python overheads and via buffer reuse and other static optimization, staging increasing examples per second by at least an order of magnitude. And while the trade-off exists, it is not particularly onerous here —- simply decorating a single function recovers the full performance of TensorFlow. This benchmark stages computation aggressively, essentially running the entire update as a graph function. Depending on the desired visibility into the model's execution during development, it is possible to stage less aggressively.

Note. These examples were chosen as they lie at opposite ends of the spectrum with regards to the tradeoff between execution speed and development speed. We expect most real-world models to fall somewhere between these two, and to be able to recover performance by making use of staged computations as required. RedactedSystem as presented is still an evolving technology, and closing the gap between the two ends of the spectrum is actively being worked upon.

Test setup. The benchmarks were run on a machine with Intel(R) Xeon(R) W-2135 CPU with 12 cores at 3.7GHz and 64GB of memory and a GTX 1080 GPU with 8GB of memory. The benchmarks were run within a docker container (tensorflow/tensorflow:1.11.0-rc2-devel-gpu). Every benchmark run was 10 iterations, and an average of 3 runs was reported. For staged computations, build and optimize times were not included in the benchmark runs as these are one-time costs that can be amortized over a number of runs.

7 Conclusion

We presented RedactedSystem, an extension to TensorFlow that makes what was once a declarative DSL for differentiable programming into a multi-stage, imperative-first one. RedactedSystem's imperative-by-default behavior makes it suitable for beginners and researchers alike, and the option to stage computations as graph functions lets users trade off the interactivity and seamless Python integration furnished by imperative execution for the benefits provided by static graphs, performance and ease of serialization among them.

The initial response to RedactedSystem is encouraging. Within our institution, dozens of scientists and students have adopted RedactedSystem; for example, some researchers use RedactedSystem to implement dynamic language models and reinforcement learning methods, and several internal workshops on RedactedSystem have been attended widely. Multiple groups are restructuring their machine learning frameworks to make RedactedSystem the default way of using them (examples include frameworks for probabilistic machine learning and reinforcement learning), and at least one large research group has engineers dedicated to supporting RedactedSystem. Externally, multiple university courses have included RedactedSystem as part of their curriculum. And, after attending our public summit for machine learning developers, 48 percent of survey respondents agreed with the statement, "[RedactedSystem] is important to me as an iterative development and debugging tool," 10 percent said it was not applicable to them, and the rest were neutral.

RedactedSystem is an evolving technology. While RedactedSystem is well-suited for research and pedagogy alike, we are still working to provide a performant, out-of-the-box solution for imperatively-driven distributed training. And, while multi-stage programming is powerful — we have found that wrapping large Python functions in `defun` often "does the right thing" — staging computations with dynamic control flow can require nontrivial programmer intervention; our colleagues are attempting to decrease this friction by providing tools that rewrite Python control flow to staged control flow and insert unstaging annotations as needed.

Acknowledgements

Omitted.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026899.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. URL http://jmlr.org/papers/v18/17-468.html.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: A CPU and GPU math compiler in Python.

Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pp. 18–25. ACM, 2009.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., and Vitek, J. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pp. 105–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462166. URL http://doi.acm.org/10.1145/2491956.2462166.

Frostig, R., Johnson, M. J., and Leary, C. Compiling machine learning programs via high-level tracing. In *SysML*, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hudak, P. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996. ISSN 0360-0300. doi: 10.1145/242224.242477. URL http://doi.acm.org/10.1145/242224.242477.

Innes, M. Flux: Elegant machine learning with Julia. *Journal of Open Source Software*, 2018. doi: 10.21105/joss.00602.

Innes, M., Barber, D., Besard, T., Bradbury, J., Churavy, V., Danisch, S., Edelman, A., Karpinski, S., Malmaud, J., Revels, J., Shah, V., Stenetorp, P., and Yuret, D. On machine learning and programming languages. https://julialang.org/blog/2017/12/ml&pl, 2017.

Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

Jørring, U. and Scherlis, W. L. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pp. 86–96, New York, NY, USA, 1986. ACM. doi: 10.1145/512644.512652. URL http://doi.acm.org/10.1145/512644.512652.

Kong, L., Dyer, C., and Smith, N. A. Segmental recurrent neural networks. *arXiv preprint arXiv:1511.06018*, 2015.

Lam, S. K., Pitrou, A., and Seibert, S. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pp. 7:1–7:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4005-2. doi: 10.1145/2833157.2833162. URL http://doi.acm.org/10.1145/2833157.2833162.

Lattner, C. and the Swift for TensorFlow Team. Swift for TensorFlow. https://github.com/tensorflow/swift, 2018.

Levy, D., Hoffman, M. D., and Sohl-Dickstein, J. Generalizing hamiltonian monte carlo with neural networks. *arXiv preprint arXiv:1711.09268*, 2017.

Maclaurin, D., Duvenaud, D., and Adams, R. P. Autograd: Effortless gradients in NumPy. In *ICML 2015 AutoML Workshop*, 2015.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. DyNet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

Oliphant, T. E. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015. ISBN 151730007X, 9781517300074.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

PyTorch team. Torch script. https://pytorch.org/docs/master/jit.html, 2018.

Rompf, T. and Odersky, M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Acm Sigplan Notices*, volume 46, pp. 127–136. ACM, 2010.

Sato, K., Young, C., and Patterson, D. An in-depth look at Googles first Tensor Processing Unit (TPU). https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2017.

Socher, R., Lin, C. C., Manning, C., and Ng, A. Y. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 129–136, 2011.

Strachey, C. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1-2): 11–49, 2000.

Sujeeth, A., Lee, H., Brown, K., Rompf, T., Chafi, H., Wu, M., Atreya, A., Odersky, M., and Olukotun, K. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 609–616, 2011.

Taha, W. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pp. 30–50. Springer, 2004.

The Gluon Team. Deep learning: The straight dope. https://gluon.mxnet.io/, 2017.

The XLA team. XLA - TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html, 2017.

Tokui, S., Oono, K., and Hido, S. Chainer: a next-generation open source framework for deep learning. 2015.

Wei, R., Adve, V., and Schwartz, L. DLVM: A modern compiler infrastructure for deep learning. *arXiv preprint arXiv:1711.03016*, 2017.

Wiltschko, A., Moldovan, D., and Dobson, W. AutoGraph converts Python into TensorFlow graphs. https://medium.com/tensorflow/autograph-converts-python-into-tensorflow-graphs-b2a871f87ec7, 2018.