AUTOGRAPH: IMPERATIVE-STYLE CODING WITH GRAPH-BASED PERFORMANCE

Dan Moldovan [1]   James M Decker [2]   Fei Wang [2]   Andrew A Johnson [1]   Brian K Lee [1]   Zachary Nado [1]   D Sculley [1]
Tiark Rompf [2]   Alexander B Wiltschko [1]

ABSTRACT

There is a perceived trade-off between machine learning code that is easy to write, and machine learning code that is scalable or fast to execute. In machine learning, *imperative* style libraries like Autograd and PyTorch are easy to write, but suffer from high interpretive overhead and are not easily deployable in production or mobile settings. *Graph-based* libraries like TensorFlow and Theano benefit from whole-program optimization and can be deployed broadly, but make expressing complex models more cumbersome. We describe how the use of staged programming in Python, via source code transformation, offers a midpoint between these two library design patterns, capturing the benefits of both. A key insight is to delay all type-dependent decisions until runtime, similar to dynamic dispatch. We instantiate these principles in AutoGraph, a software system that improves the programming experience of the TensorFlow library, and demonstrate usability improvements with no loss in performance compared to native TensorFlow graphs. We also show that our system is backend agnostic, targeting an alternate IR with characteristics not found in TensorFlow graphs.

1 PROGRAMMING PARADIGMS FOR MACHINE LEARNING

Programming platforms specialized for machine learning (ML) are undergoing widespread adoption, as ML models such as neural networks demonstrate state-of-the-art performance on many important industrial problems like translation and image recognition. In order to support this proliferation of use, there has been rapid development of platforms for building new ML models. These platforms follow two main paradigms, *graph-based* programming and *imperative* programming. These have also been labeled *Define-and-run* and *Define-by-run* (Tokui et al., 2015b).

Graph-based systems like TensorFlow and Theano use a high-level language (typically Python) to metaprogram a lower-level intermediate representation (IR) of computation (Abadi et al., 2016; Al-Rfou et al., 2016). In TensorFlow's case, this IR provides a representation that can then be automatically distributed across a datacenter, executed on accelerator hardware like GPUs or TPUs, deployed to mobile devices or web servers, and can benefit from whole-program optimization. The computational gains are significant, but come at the cost of additional cognitive load for developers.

Imperative programming systems like PyTorch and Autograd (Paszke et al., 2017; Maclaurin et al., 2015) run user

code directly, building up a representation of the user's program incrementally for either automatic differentiation or compilation. TensorFlow also supports imperative-style coding via "eager execution", where user-written Python code immediately executes TensorFlow kernels, without a graph being built. Such systems allow the user to enjoy the benefit of traditional imperative coding, but have reduced opportunities for program optimization, scalable computation and portability.

The differences between these approaches are especially apparent for models that require data-dependent control flow, such as conditionals or loops, which are important for state of the art methods in Reinforcement Learning, sequence-based models, and many other emerging research areas. *Imperative* platforms allow a user to write idiomatic and native Python control flow, using traditional syntax for data-dependent control flow operations such as conditionals and loops. However, this approach reduces opportunities for whole-program optimization and requires retracing on every execution for automatic differentiation. *Graph-based* platforms avoid this issue, but do not allow traditional Python syntax for data-dependent control flown, and instead require any data-dependent control flow to be expressed in a functional form. This is required because Python does not natively support deferring the execution of control flow.

While *graph-based* and *imperative* programming are often presented as orthogonal and independent programming paradigms, we provide an approach that offers the best of both, retaining *imperative* usability benefits while still yielding *graph-based* performance and portability benefits. We

---

[1]Google Brain, Cambridge, MA [2]Purdue University. Correspondence to: Tiark Rompf <tiark@purdue.edu>, Alexander B Wiltschko <alexbw@google.com>.

note that this approach assumes the ability to transform code into a specialized IR, and that this IR confers real benefits to the programmer such as speed, memory and numerical stability optimizations, as well as deployability to a variety of platforms. However, like many IRs, we also assume that it is cumbersome to program directly. Due to its widespread usage and robust IR, we focus much of our discussion on TensorFlow graphs, but show in our evaluation (Section 9.1) that this approach is completely independent of any back-end, and indeed, we can represent programs not easily expressible in TensorFlow's IR by selecting a different back-end for our code generation engine to target.

The contributions of this paper are as follows:

- We propose a new methodology that provides users the expressive power of *imperative* ML systems, while retaining the performance and portability of *graph-based* systems.
- We demonstrate this methodology in Python using static analyses and *source code transformations* (SCT).
- Using these analyses and code transforms, we enable staged programming in Python dispatching on runtime type information, in most cases requiring no additional annotations.
- We use our system, called AutoGraph, to convert idiomatic Python into TensorFlow Graph IR. We show that AutoGraph generalizes to target other back-ends, and can convert Python code into the Lantern IR, which supports features absent from the TensorFlow Graph IR, such as re-entrant function calls.
- We demonstrate that our system allows a user to easily express complex ML programs that lower to an optimized IR, and run as fast as hand-written alternatives.

2 RELATED WORK

A number of existing systems and approaches also aim to provide an easy-to-use programming interface for defining ML models without degrading performance. One such example is the Open Neural-Network eXchange (ONNX) format (ONNX Contributors, 2018), which provides an IR with APIs for many high-level front-ends that can target a number of popular back-ends focused on optimization and high-performance computing. This IR is exhibited as a computation graph, generated through the use of tracing, as in many *imperative* systems. ONNX provides insight into the ability to use an IR as the broker between *imperative* and *graph-based* systems, though extracting graphs via tracing may yield a loss of control flow information due to the inability to capture data-dependent control flow.

Another recent approach is that of PyTorch's Torch Script framework (PyTorch Contributors, 2018). While based on Python AST translation similar to AutoGraph, there are a number of important differences, most notably the lack of staging beyond shape propagation on a dynamically-shaped graph. A more complete comparison of Torch Script and AutoGraph can be found in Section 10. The Myia system (van Merrienboer et al., 2018) provides a similar facility Torch Script, where the user expresses numeric code in Python which is then parsed into a graph-based IR distinct from the Python AST. JANUS (Jeong et al., 2019) operates like a JIT compiler from Python bytecode to TensorFlow graph code, modifying the Python interpreter. In contrast, AutoGraph works as a stand-alone library performing source-to-source transformations.

Providing easier deferred execution using staged programming or multiple dispatch has a long history. Notable examples include Lightweight Modular Staging's type-based deferred execution model (Rompf & Odersky, 2010), the paired use of Lua and Terra to stage high-performance numerical code (DeVito et al., 2013), and Julia's multiple dispatch system (Bezanson et al., 2012). Libraries implementing or using code rewriting in Python have been in limited use, including the privacy- and confidentiality-aware Jeeves system (Yang et al., 2016), which relies on MacroPy (Haoyi et al.), as well as the Hy system, a Lisp dialect embedded in Python (Hy Contributers, 2018). However, each of these approaches alone, without substantial modification, are inappropriate for the Python language.

Other efforts contributed a variety of ML frameworks with different features. Lantern (Wang & Rompf, 2018; Wang et al., 2018) applied concepts of programming languages research (delimited continuations and multi-stage programming) to implement an expressive *graph-based* ML framework. Tangent (van Merrinboer et al., 2017) performs automatic differentiation using SCT. Dynet (Neubig et al., 2017) is a define-by-run system with a dynamic batching runtime for automated batching computations. MXNet (Chen et al., 2015) offers both options of *define-by-run* and *graph-based* through the use of different syntax. Both chainer (Tokui et al., 2015a) and torch-autograd, a Lua port of the Autograd library (Torch Autograd Contributors, 2018) are pure define-by-run systems. Numba (Lam et al., 2015) translates annotated Python functions to machine code at runtime.

3 PROGRAMMING TENSORFLOW

The TensorFlow software programming system has become popular for ML practitioners, particularly those focusing on large-scale training and deployment (Hale, 2018). ML programs naturally execute in separate stages, as model architecture and data examples become available at different points in a program's lifecycle, and TensorFlow makes these stages explicit. A TensorFlow user must first build up a representation of the computation to be run, and then later in their program, specify that the computation should

be executed. Dataflow graphs are used for this representation, because they can be readily optimized, distributed and deployed. This programming model is sometimes non-obvious, leading to difficult usability issues and bugs, and is particularly acute in the case of specifying control flow. For example, some control flow constructs should be included in the lowered IR, while others are meant to specify whether or not computation should be staged into the IR. A common coding pattern is to conditionally stage computation using model hyperparameters:

```python
# Conditional on bool not added to graph
if HParams.nonlin == 'relu':
    x = tf.nn.relu(x)
else:
    x = tf.nn.tanh(x)
```

However, other uses of control flow are meant to be executed in a data-dependent manner:

```python
# Conditional on Tensor added to graph
x = tf.cond(tf.reduce_sum(x) > 0,
  lambda: x * x, lambda: x)
```

In the code above, the conditional statement is expressed in a functional style so that it can be executed in-graph in a data-dependent manner. However, this clashes aesthetically and pragmatically with the imperative style of Python. This difficulty is exacerbated when the user needs to nest control flow, or use other Python idioms like `continue` and `break`. We would instead prefer to write

```python
# Conditional on Tensor - staged
if tf.reduce_sum(x) > 0:
  x = x * x
```

and have it be automatically converted into the functional style. We want this conversion to only occur for expressions using numeric types. Conditionals switching on plain Python booleans (e.g., the hyperparameter example above) should be executed imperatively, without staging.

4 EXTENDING OPERATOR OVERLOADING

In the case of TensorFlow, metaprogramming dataflow graphs can be difficult for complex programs, but it is made easier via *operator overloading*. For example, the user does not need to type out `tf.add(a, b)`, but instead can simply use `a + b`. This is possible due to Python's ability to allow the programmer to overload a subset of the language. Python's approach to operator overloading allows custom classes, like the `Tensor` type in TensorFlow, to override some default functionality, like their behavior when used in binary operators (e.g. `+,*,-,\%,/,^,~`) or item access.[1]

---

[1]See Python Language Reference (https://docs.python.org/3/reference/), Section 3.3.

```python
# Because Python lets us write this ...
class Tensor(_TensorLike):
  def __add__(self, right):
    return tf.add(self, right)

# ... we can write this
import tensorflow as tf
a = tf.constant(3)
b = tf.constant(4)
c = a + b
```

This is a powerful facility in the Python language, but it unfortunately only extends to methods of objects or classes, and does not include programming constructs required to build modern ML models. For example, the behavior of conditionals cannot be overloaded in Python.

```python
# We can write if statements...
if cond:
  ans = true_fn()
else:
  ans = false_fn()

# ... but we cannot overload them
def __if__(self, cond, true_fn, false_fn):
  if cond:
    return true_fn()
  else:
    return false_fn()
```

If overloading control flow syntax were possible, *imperative* programs would be able to generate full representations of user code, including previously-invisible loop and conditional statements. *Graph-based* programs would not need to require users to write their program control flow in a cumbersome functional form, because they could provide non-standard overrides of `__if__`, `__for__` and `__while__` and other useful parts of the Python language.

To circumvent this limitation, we use SCT on whole functions to enable overloading non-local parts of the Python language. We describe a specific instantiation of this system, called *AutoGraph* which uses SCT to allow users to target a lower-level IR while still writing idiomatic Python.

5 STAGED PROGRAMMING FOR REAL-WORLD ML SYSTEMS

Using the ability to overload arbitrary Python syntax, we built a staged programming system called AutoGraph for improving the performance of *imperative*-style ML programs and conversely, the simplicity of *graph-based* ML programs.

AutoGraph allows users to program using idiomatic and imperative-style Python, but still benefit from the advantages of TensorFlow graphs, and is exposed to users via a single-function API, as a Python function decorator as seen in Listing 1.

```python
import autograph as ag

# AutoGraph converts whole
# functions via a decorator...
@ag.convert()
def f(x):
if x > 0:
  x = x * x
return x

# ... into a form where control flow
# and other idioms are overloadable
def new_f(x):
  def if_true():
    x_1 = x
    x_1 = x_1 * x_1
    return x_1
  def if_false():
    return x
  x = ag.if_stmt(
      ag.gt_(x, 0), if_true, if_false)
    return x
```

Listing 1: AutoGraph automatically converts the code on the top into the code on the bottom (simplified example).

```python
def if_stmt(cond, body, orelse):
  if is_tensor(cond):
    return tf.cond(cond, body, orelse)
  elif cond:
    return body()
  else:
    return orelse()
```

Listing 2: Simplified version of AutoGraph's conditional statement override.

AutoGraph works with control flow, such as **if** , **for** and **while** statements, even if they are arbitrarily nested or contain **break** and **continue** statements.

The AutoGraph system can overload conditionals and loops via SCT, allowing us to deviate from Python's default behavior. Note that, using the same style of SCT, we may choose to overload some statements while preserving Python semantics for others. Because of this, we anticipate that this might be a tool of general interest to Python developers, or a feature that new language implementations might want to consider including. In order to transparently support control flow that is meant to either be staged or unstaged in TensorFlow, as in the conditional examples in Section 3, we must change the behavior of **if** statements based on the type of the boolean predicate.

6 "DYNAMIC DISPATCH" ENABLES STAGED PROGRAMMING IN PYTHON

Given that we can enable overloadable control flow in Python, we can redefine its default behavior by writing a non-default implementation of `ag.if_stmt` . In the case that a Python boolean is used as the predicate of a conditional, we would want to execute the conditional with normal semantics. However, if a TensorFlow Tensor is supplied, or some other specialized numeric type, we would want to stage more specialized code. A simplified version of `ag.if_stmt` is shown in Listing 2.

We use the term *dynamic dispatch* to describe this runtime

decision making, as it is analogous to dynamic *method* dispatch common in object oriented programming. Dynamic dispatch critically allows us to seamlessly switch between two common uses of control flow in ML code – a "macro-programming" mode that switches or loops on the value of hyperparameters, and a data-dependent mode, where the control flow is lowered into the target IR.

The same logic is applied to **for** and **while** loops in the equivalent of `ag.for_stmt` and `ag.while_stmt` functions. We also provide functionality for overriding the **print** statement, which is ordinarily incompatible with TensorFlow graphs, since **print** would log information immediately, and we instead want to log values at graph runtime.

Note that some native Python constructs, like **break** and **continue** statements have no direct representation in TensorFlow. This requires code transformations which entirely remove these statements without affecting program semantics. This is achieved by lowering the respective statements into equivalent TensorFlow constructs. For example, **continue** is lowered using extra variables and conditionals.

The dynamic dispatch approach incurs extra runtime overhead. Indeed, if AutoGraph was used to perform normal unstaged Python computation, it would be slower. However, because we target a lower-level IR that can be executed separately from the Python runtime, this overhead is amortized.

General Approach The conversion of a function proceeds as follows:

1. Read the source code of the function and obtain its closure variables, if they are available.
2. Parse the source code into a Python AST, abstracting away small differences between Python versions.
3. Transform the source code in multiple passes, with each pass consisting of two major steps:
   (a) Static analysis, detailed below. The AST is annotated with additional information that the actual transformation may use.
   (b) AST transformations, where each transform handles a specific Python idiom. The specific trans-

formations are detailed below.

4. Serialize the final AST into output code.
5. Load the new output code in as a Python function, and dynamically attach symbols corresponding to the original function's closure variables.

Comparison with Static Methods It is possible to extract computation graphs from Python code statically, but doing so requires a strict set of constraints. Systems like Torch Script (PyTorch Contributors, 2018) elect to impose these constraints in the form of a DSL which is a limited subset of Python. A major design decision in AutoGraph, however, is to allow users access to as much of the original Python interface as is possible (we discuss limitations to this in Section 10). Furthermore, due to binding-time analysis, relying wholly on static methods disallows staged programming in Python without requiring some form of an ersatz static type system (e.g., static type annotations). While enabling staged programming in a dynamic setting for arbitrary types does require careful consideration (Decker et al., 2019), our decision to primarily target TensorFlow as a back-end significantly alleviates some of the implementation pains due to the central focus of an array-based type (tensor). We discuss this in detail in Section 7.

7   CODE ANALYSES AND CONVERSION

Only a subset of Python can be trivially converted, and substantial rewriting of user-provided code is necessary to enable the overloading required for staged programming. For example, loops and conditionals need to be rewritten in functional form; nonlocal control flow statements need to be lowered. We perform these rewrites with the aid of dataflow and other analyses of program structure. We also separate these rewrites into multiple specialized passes.

7.1   Dataflow Analysis

Each specialized pass is preceded by several dataflow analysis passes. These are described below, in the order that they are run.

Control Flow Graph Construction A standard intra-procedural control flow graph (CFG) supports several static analyses.

Qualified Name Resolution We create the abstraction of qualified names to extend the notion of symbols to include compound names such as `a.b`. For example, the qualified name `a.b` roughly corresponds to the AST: `Attribute(name=Name('a'), attr='b')`.

Activity Analysis Here we annotate AST nodes with the list of symbols read and modified by the respective state-

ment. Only direct modifications are considered writes. For example, in the statement `a.b = c`, `a.b` is considered to be modified, but `a` is not. The activity analysis also keeps track of lexical scopes, their nesting relationships (e.g. the parent scope) and the symbols they include.

Reaching Definitions Analysis This standard dataflow analysis annotates help identify the definition that reaches each name. Additionally, the list of symbols defined on entry of certain statements is also annotated.

Liveness Analysis This standard dataflow analysis identifies symbols that are live upon entry into or exit from certain statements, including compound statements like conditionals.

7.2   Code Conversion Passes

AutoGraph performs code conversion using an extensible system of multiple, typically independent, AST conversion passes. For example, one conversion pass rewrites the `if` statements into an overloadable functional form. Another pass lowers the `break` statements into new loop predicates and extra conditionals. This mechanism facilitates adding support for more Python idioms in time.

Currently, the transformations include the following, in order of application:

Directives Identifies calls to specific functions that serve as AutoGraph compilation directives and annotates the relevant AST nodes. An example of such a directive is `ag.set_loop_options`.

Break, Continue and Return Statements These are actually three separate passes, but are very similar in nature. In each case, the corresponding statement is lowered into conditionals or expanded loop conditions.

```python
# Before conversion
if cond:
  return f(x)
return g(x)

# After conversion
if cond:
  return_value = f(x)
else:
  return_value = g(x)
return return_value
```

Assert Statements These are converted in-place to overloadable functional form.

Lists List idioms, including list literals and the `append` and `pop` function calls are overloaded with custom func-

tions (e.g. `ag.list_append` and `ag.list_pop`) that allow staging the respective operation.

Array computations require an additional idiom not present in the standard Python library: the stack operation. AutoGraph provides the `ag.stack` function which can be overloaded in a manner consistent with the other overloads. Note that list access (e.g. `l[i]`) and mutation are deferred to a separate conversion pass which covers slice operators.

Slices   Python does allow overloading the slice operators (`__setitem__`, `__getitem__`) in user classes. However, the slice write operation has the semantic of mutating the target.   We rewrite slice writes to use value semantics as currently required by Tensor-Flow. For instance, `x[i] = y` is converted in-place to `x = ag.setitem(x, i, y)`. Slice read operations are converted mechanically.

Function Calls   All function calls are overloaded. The overload will either dynamically convert the target function, call it as-is or replace it with a new function, depending on the characteristics of the function being called and the configuration of the conversion. For example, the built-in function **print** may be converted to `tf.`**print** (see Appendix E for details).

```python
# Before conversion
def f(a, x):
  return a(x)

# After conversion (simplified)
def f(a, x):
  return ag.converted_call(a, x)
```

Control Flow   This conversion pass replaces all local control flow with an overloadable equivalent functional form.

The **if** statement is stateless, therefore its functional form can be expressed using niladic functions that return all the variables modified inside the statement.

```python
# Before conversion
if x > 0:
  x = x * x

# After conversion (simplified)
def true_fn():
  return x * x
def false_fn():
  return x
x = ag.if_stmt(x > 0, true_fn, false_fn)
```

Note that Python allows to define (i.e., assign for the first time) symbols inside the body of control flow statements and use them later. It is possible to write code where symbols may be undefined based on whether the branch of a conditional executed or not. However, the functional version

of the conditional operators always sets the symbols that the conditional may modify in either branch. To simulate the undefined semantics, we use a special value to reify the "undefined" state of a variable. This currently deviates from Python semantics, but we plan to remedy this by verifying and explicitly deleting "undefined" symbols before they are used.

The **while** and **for** loops are stateful, and their functional form requires functions whose arguments and return values represent the variables modified inside the loop (its state).

```python
# Before conversion
while x > eps:
  x = f(x)

# After conversion (simplified)
def loop_test(x):
  return x > eps
def loop_body(x):
  return f(x)
x = ag.while_stmt(
  loop_test, loop_body, (x,))
```

The **for** statement is handled similarly. Similar to **if** statements, **while** and **for** loops may define symbols inside their body. If the loop body never executes, those symbols will remain undefined. This is also handled by using special "undefined" values for the symbols that are not defined (as identified by liveness analysis) upon entry into the loop.

The overloaded control flow uses dynamic dispatch (see Appendix E).

Ternary Conditional Expressions   The ternary operator `x` **if** `cond` **else** `y` is converted inline to the functional form `ag.if_stmt(cond, x, y)`.

Logical Expressions   Binary and unary logical expressions can be handled using traditional operator overloading (e.g. `__lt__` for the `<` operator). However, `Tensor` does not support all operators for compatibility reasons (for example, `__eq__` is not supported). Therefore we replace certain binary and unary operators inline with overloadable functional forms. For example, `a` **and** `b` is replaced with `ag.and_(a, b)`.

Function Wrappers   This conversion pass wraps the entire block of functions with additional boilerplate code. This accommodates for examples the necessary calls to create a TensorFlow *name scope*, which improves the readability of the rendered graph. In addition, the function wrappers contain specialized error handlers that intercept certain errors to improve usability.

6

8 BEYOND TENSORFLOW: ALTERNATE BACK-ENDS

If TensorFlow is the only back-end of this code transformation, then the limitations of TensorFlow must also apply to AutoGraph. However, due to the nature of metaprogramming, the SCT in AutoGraph can easily be used to target a variety of back-ends. As previously discussed, one shortcoming of TensorFlow is the inability to handle re-entrant in-graph functions, and by extension, recursive models. In order to showcase the utility of a general purpose SCT methodology as implemented by AutoGraph, we elect to target a new ML framework prototype called Lantern (Wang & Rompf, 2018; Wang et al., 2018), which is capable of generating graphs describing recursive models.

The Lantern IR The Lantern back-end converts Lisp-like S-expressions describing numeric operations into efficient C++ code. Critically, Lantern supports programming features absent in the TensorFlow graph specification, like function recursion and in-line function definitions, which are essential in some state-of-the-art ML language models. We demonstrate the generality of AutoGraph by targeting the Lantern S-expression IR, which is supported by additional code conversion passes.

Staging Functions and Recursion In order to deal with functions in our model, we introduce two new functions: `__def_staging(function, *args)` and `__call_staging(function, *args)`. These emit a function definition or call, respectively, in the generated S-Expression. Due to the deferred API presented by AutoGraph, we have the ability to specialize the generated functions in the S-Expression IR with respect to known parameters. Note that this specialization in function calls/definitions requires no additional modifications, as it is handled using the existing dispatching and overloading mechanisms present in AutoGraph. With the ability to define and call functions in the generated computation graph, this provides the interface necessary for defining and running recursive models.

To demonstrate this, we provide an end-to-end example of Python → S-Expr → C++. We first examine a recursive function in Python, as follows:

```python
@ag.convert()
def tree_prod(base, tree):
    if not tree.is_empty:
        l = tree_prod(base, tree.left)
        r = tree_prod(base, tree.right)
        return l * r * tree.value
    else:
        return base
```

With the modifications in place which allow us to target Lantern, this will generate the following Python code (simplified for presentation):

```python
def run(base, tree):
    def tree_prod(base, tree):
        def true_fn():
            return base

        def false_fn():
            l = __call_staged(tree_prod,
                    base, tree.left)
            r = __call_staged(tree_prod,
                    base, tree.right)
            return l * r * tree.value
        ag.if_stmt(tree.is_empty,
                true_fn, false_fn)
    __def_staged(tree_prod, base, tree)
    return __call_staged(tree_prod, base,
                    tree)
```

Note that in order to correctly generate the staged function, `__def_staged` must be passed the arguments which will eventually be passed to the function being defined. Running this generates S-Expression code, which is then fed as input to Lantern, which performs some internal computations and eventually generates and executes the following C++ code:

```cpp
double Snippet(double base, Tree tree) {
  auto rec = [&](Tree tree,
  function<double(double)> cont,
  double base) {
    double grad = 0.0;
    if (!tree.is_empty) {
      auto cont_l = [&](double x1) {
        double sub_grad = 0.0;
        auto cont_r = [&](double x2) {
          double x3 = tree.value;
          double x4 = cont(x1 * x2 * x3);
          double x5 = x3 * x4;
          sub_grad += x2 * x5;
          return x1 * x5;
        };
        grad += rec(tree.R, cont_r, base);
        return sub_grad;
      };
      grad += rec(tree.L, cont_l, base);
    } else
      grad += cont(base);
    return grad;
  };
  return rec(tree,
    [&](auto x){return 1.0;}, base);
}
```

As shown, staging a recursive function requires that the generated C++ code also be recursive (as noted by the `rec` function). We note that the generated C++ code looks fairly complicated, due to the handling of back-propagation. Back-propagation is implemented via callbacks (seen as continuations, noted by `cont`, `cont_l`, and `cont_r` in the

code), the details of which can be referenced in Wang & Rompf (2018); Wang et al. (2018).

9 EVALUATION

We tested the utility of AutoGraph on several axes. First, we asked whether AutoGraph could improve the readability of ML code that relied on data-dependent control flow without incurring a performance penalty. Second, we tested if AutoGraph could be used to move computation usually left outside of the TensorFlow graph, such as the entire training process of stochastic gradient descent (SGD), inside the graph IR. Third, we tested if AutoGraph could be used to produce performant code using features not supported in the TensorFlow graph by targeting alternative IRs. We also prepared additional samples of more complex algorithms, including Neural Model Translation with Attention, Sequence-to-sequence, MAML metalearning and L-BFGS optimizations. These can be found in Appendix D.

RNN cells The code snippet below is an implementation of an RNN model that on simple inputs produces results identical to TensorFlow's built-in `tf.dynamic_rnn` function and runs at similar speed.

```
def dynamic_rnn(rnn_cell, input_data,
  initial_state, sequence_len=None):
  input_data = tf.transpose(input_data,
    (1, 0, 2))
  outputs = []
  ag.set_element_type(outputs, tf.float32)
  state = initial_state
  if sequence_length is None:
    max_len = tf.shape(input_data)[0]
  else:
    max_len = tf.reduce_max(sequence_len)
  for i in tf.range(max_len):
    prev_state = state
    output, state = rnn_cell(input_data[i],
      state)
    state = tf.where(
        i < sequence_len,
        state,
        prev_state)
    outputs.append(output)
  outputs = ag.stack(outputs)
  outputs = tf.transpose(outputs,
    (1, 0, 2))
  return outputs, state
```

Compare this terse and readable implementation to the equivalent graph version in Appendix A.

We compared TensorFlow's official implementation of `tf.dynamic_rnn` with both a hand-written, graph-based implementation, and the code snippet above converted into graphs via AutoGraph. Each run consisted of an execution of an RNN having hidden size 256, while varying batch

Table 1. RNN Cell Performance (1K examples/sec)

Sequence Size	Seq Size: 64			Seq Size: 128		
Batch Size	32	64	128	32	64	128
Eager	0.82 ± 0.08	1.57 ± 0.13	2.04 ± 0.14	0.43 ± 0.03	0.76 ± 0.05	1.04 ± 0.06
Official	2.88 ± 0.11	3.63 ± 0.13	5.13 ± 0.15	1.44 ± 0.04	1.91 ± 0.06	2.61 ± 0.05
Handwritten	2.95 ± 0.13	3.71 ± 0.15	5.24 ± 0.11	1.52 ± 0.06	1.96 ± 0.07	2.68 ± 0.03
AutoGraph	2.72 ± 0.09	3.61 ± 0.12	5.05 ± 0.10	1.37 ± 0.04	1.86 ± 0.06	2.59 ± 0.04

sizes and the sequence length. Five warm-up runs were executed, and the mean and standard deviation of the 100 following runs are reported. For all examples, each run is executed as one `tf.Session.run()` call. All benchmarks were run on a dual-threaded 6-core Intel Xeon E5-1650 CPU. The use of AutoGraph improves the readability of the code and has a very minor effect on performance.

In-Graph Training Typically, a TensorFlow graph representing a single training step is executed repeatedly in a Python training loop outside of TensorFlow. This method is used because of the difficulty of using control flow operators within TensorFlow graphs, and incurs additional computational overhead. Here, we use AutoGraph to demonstrate a training loop that is implemented entirely as a computation graph. We trained a single linear layer on MNIST with stochastic gradient descent (SGD), and compared its performance with several other implementations. The first approach was TensorFlow Eager, an imperative execution mode for TensorFlow similar to NumPy and PyTorch. The second approach we tested was a traditional TensorFlow training process. The third approach was an in-graph training loop implemented using the TensorFlow `while_loop` API.

Table 2. Model and Training Loop

	SGD Steps / sec
Eager	274.1 ± 3.6
Model In Graph, Loop In Python	484.1 ± 7.7
Model And Loop In Graph	646.5 ± 14.1
Model And Loop In AutoGraph	623.5 ± 13.5

Each run consisted of 1000 training steps with a batch size of 200. One warm-up run was executed, and the mean and standard deviation of the 10 following runs are reported. For the in-graph training loop examples, the entire set of 1000 training steps is executed in one `tf.Session.run()` call. For the other examples, each training step is run as a separate `tf.Session.run()` call. Executing a single-training-step graph repeatedly in a Python loop (the traditional approach) is faster than the eager-style code by 75%. Moving the entire training process into a TensorFlow graph further yielded a roughly 30% speedup.

9.1 AutoGraph + Lantern: TreeLSTM

We evaluated a model of TreeLSTM for Sentiment Classification running on the dataset of the Stanford Sentiment (Socher et al., 2013), following the work of (Tai et al., 2015). The model embeds sentence parse-trees by recursively embedding the left/right sub-trees and combining the embedding vectors via BiLSTM core. The embedding of whole sentences is then passed to MLP for sentiment prediction. The model can be easily expressed in PyTorch using recursive functions, or in AutoGraph targeting recursive functions in Python. The final generated C++ code was compared against the PyTorch implementation in terms of training efficiency. To approximate a "real-world" running time, this experiment was run using a single thread on a laptop with a dual-core AMD A9-9410 Radeon CPU @ 1.70GHz and 8GB of SODIMM Synchronous 2400 MHz RAM, with Ubuntu 16.04.

Our AutoGraph implementation of TreeLSTM targeting Lantern yielded performance approximately 2.38 times faster than that of the PyTorch implementation. Our system achieved approximately 36.75 SGD steps per second, compared with the 15.41 steps per second using the PyTorch implementation. We note that we used a batch size of 1 for both systems due to difficulty in batching recursive models.

Table 3. TreeLSTM Targeting Lantern

Moved to separate files.	SGD Steps / sec
Loop and Model in PyTorch	15.41
Loop and Model in AutoGraph/Lantern	36.75

10 DISCUSSION

Developing a source code transformation methodology is far from mechanical. There exist a number of design decisions which may ultimately yield different results in terms of expressiveness, performance and portability. In this section, we discuss some of these decisions and provide insight regarding how they shaped the current state of AutoGraph, including its current limitations. We provide a detailed discussion of error handling in AutoGraph in the Appendix B.

Engineering Practices as a Feature The code conversion passes we implement in AutoGraph are non-local, and can interact with each other in complicated ways. For instance, converting deeply-nested `for` loops and `if` statements exposes dataflow interactions between each level of nesting. In order to build a reliable system, we made extensive use of engineering best-practices. For instance, all static analyses, code transforms, and utility functions are extensively unit tested (>50% of the 22k LOC in Auto-Graph is tests). Further, interactions between features are tested in end-to-end reference tests. Any changes to the AutoGraph system require that all unit and reference tests pass, and all code is manually reviewed by at least one engineer for correctness, readability and adherence to style guidelines. Anecdotally, this test- and review-oriented development practice has caught many surprising and subtle bugs, and allowed a library as complex as AutoGraph to remain relatively easy to maintain and extend. Further, we built many useful utilities for manipulating Python source code that simplified development (described in Appendix C).

Alternative Approaches for Implementing Staged Programming An alternative approach to SCT would have been to build a new Python interpreter with non-standard execution semantics for Python programs that could map to TensorFlow graphs, and indeed, an early proposal for AutoGraph was to do exactly this. However, a non-standard Python interpreter would require reimplementing all aspects of the Python language, including those parts that require no modifications in machine learning code.

We could also parse Python to our own intermediate representation, a strategy taken recently by the Myia system (van Merrienboer et al., 2018). This intermediate representation could then be either back-converted to Python or executed in a dedicated VM. Indeed, this strategy is similar to our ability to work with Lantern; AutoGraph modifies the original Python source code such that it generates S-Expressions as an IR, which are then consumed by Lantern.

Our choice to emit Python code after conversion has several advantages. Unsupported code idioms are allowed to pass through conversion if they do not affect the program semantics. This simplifies the support for legacy TensorFlow code. Further, the generated code can be inspected, and even modified by the user.

Comparing Torch Script and AutoGraph Similar to ONNX, PyTorch's Torch Script framework (PyTorch Contributors, 2018) allows users to save models for later evaluation, while providing an even higher-level interface for programming: nearly native Python with two new decorators. These decorators, `torch.jit.trace` and `torch.jit.script`, produce Torch Script code (a subset of Python used as an IR for the eventual computation graph) from idiomatic Python, though they accomplish this via different methods.

The `torch.jit.trace` decorator works as the name suggests: it extracts computation graphs through tracing. This produces fully shape-specialized Torch Script code, which allows for highly optimized models (and an easy target for potential compilers). However, tracing in Torch Script has the same drawback as found in ONNX: as stated clearly by the Torch Script developers, "Tracing only correctly records

functions and modules which are not data dependent (e.g., have conditionals on data in tensors)...”

Torch Script's `torch.jit.script` decorator, on the other hand, will directly translate the decorated Python function to Torch Script code, which does allow for data-dependent control flow. While this seems similar to AutoGraph's source code transformation model (detailed in Section 6), there are a number of important differences between the two methodologies. Torch Script is inherently bound to the PyTorch runtime, which prevents the use of Torch Script with any other specialized or accelerated ML back-ends. Furthermore, `torch.jit.script` does all of its work at compile time, and thus the only view of staging available currently is the ability to do shape propagation on a dynamically-shaped graph (resulting from `torch.jit.script`). This drawback comes as a result of the decision to target a relatively basic IR (Torch Script), rather than Python code. One powerful consequence of this decision, however, is the ability to cleanly implement autobatching on Torch Script, which is otherwise difficult in systems targeting a broader IR.

Limitations   The Python language is large, and AutoGraph does not stage all of it. We focus on the subset that enables machine learning programming, but we are still missing many useful constructs, such as associative data structures and **`try`**/**`except`** blocks. In some cases, there is no corresponding construct in the TensorFlow or Lantern IR, but as we build support for more IRs, we anticipate being able to successfully convert more of the Python language. Although only a subset of the Python language is converted to TensorFlow constructs, AutoGraph does allow nearly every Python construct, and will simply call it unconverted. This allows AutoGraph to be compatible with the vast majority of existing graph code. Appendix E exhaustively documents Python language support in AutoGraph.

In addition, the data-dependent staging decisions made by AutoGraph are obscured from the user, much like Python operator overloading obscures computation made in the overloaded operators. For instance, if the user accidentally passes a Python boolean instead of a TensorFlow boolean to a conditional, it will not be staged into a graph, with potential performance implications. Currently, the user has few tools to catch and debug this behavior. We already provide better error messages than a system like this naively would (see Appendix B), but further work is required.

Additional challenges arise from the mismatch between Python and the IRs typing system. For example, TensorFlow does not support nullable types, so we impose additional constraints on the Python semantics by requiring that all code paths initialize a variable when control flow is staged in TensorFlow. Similarly, because Python types like lists are

generic, element access lacks type information and we may require additional user annotations when the IR is strongly typed, which is usually the case. More advanced type inference mechanics that could obviate these annotation is a subject for future work.

We make a best effort to guarantee that the conversion to IR is either semantics-preserving, or it explicitly fails. However, a more rigorous treatment of the correctness of our system is needed. We plan to treat this both formally and empirically, using a random code generation fuzzing system. In the meantime, we provide as evidence of correctness an expansive test suite for AutoGraph, containing hundreds of tests. Furthermore, due to AutoGraph being included in `tf.function`, the default way to accelerate code in TensorFlow 2.0, AutoGraph is also subject to all tests covering the TensorFlow codebase. While this notion of test-based correctness does not provide a formal guarantee of correctness, we note that this is consistent with other formal analyses of Python semantics (Politz et al., 2013).

Lastly, AutoGraph relies on Python introspection and reflection APIs, such as `inspect` and `imp`. While these are available in the vast majority if use cases, there are instances when AutoGraph cannot be used, for example when source code information is not available.

11   CONCLUSIONS AND FUTURE WORK

We have described AutoGraph, a staged programming system for automatically rewriting idiomatic Python code into an equivalent lower-level IR, including TensorFlow graphs and other, more experimental, back-ends. AutoGraph achieves a balance in the design space between *imperative* and *graph-based* code. These two programming models – fully-imperative with high runtime overhead, and fully-staged with high developer mental overhead – are not binary choices. Using SCT, we can eliminate the distinction between the two. We believe that this approach is applicable broadly, and are working to target a wider suite of IRs in new applications.

The entirety of AutoGraph is open sourced via the TensorFlow project on GitHub at `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/autograph`.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 472:473, 2016.

Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

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. *CoRR*, abs/1512.01274, 2015.

Decker, J. M., Moldovan, D., Wei, G., Bhardwaj, V., Essertel, G., Wang, F., Wiltschko, A. B., and Rompf, T. The 800 pound python in the machine learning room. 2019. URL https://www.cs.purdue.edu/homes/rompf/papers/decker-preprint201811.pdf.

DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., and Vitek, J. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pp. 105–116. ACM, 2013.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 1126–1135, 2017. URL http://proceedings.mlr.press/v70/finn17a.html.

Hale, J. Deep learning framework power scores 2018. https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a, 2018. Accessed: 2018-09-25.

Haoyi, L., Holmgren, J., and Berti, A. Macropy. https://github.com/lihaoyi/macropy. Accessed: 2018-09-25.

Hy Contributers. Hylang. https://github.com/hylang/hy, 2018. Accessed: 2018-09-25.

Jeong, E., Cho, S., Yu, G.-I., Jeong, J. S., Shin, D.-J., and Chun, B.-G. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 453–468, Boston, MA, 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/jeong.

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.

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., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

ONNX Contributors. Open neural network exchange. https://github.com/onnx/onnx, 2018. Accessed: 2018-09-24.

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. 2017.

Politz, J. G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., and Krishnamurthi, S. Python: the full monty. In *OOPSLA*, pp. 217–232. ACM, 2013.

PyTorch Contributors. Torch script. https://pytorch.org/docs/master/jit.html, 2018. Accessed: 2018-09-24.

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.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1631–1642. Association for Computational Linguistics, 2013. URL http://www.aclweb.org/anthology/D13-1170.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.

Tokui, S., Oono, K., Hido, S., and Clayton, J. 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)*, 2015a. URL http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

Tokui, S., Oono, K., Hido, S., and Clayton, J. Chainer: a next-generation open source framework for deep learning. In *NIPS 2015 LearningSys Workshop*, volume 5, 2015b.

Torch Autograd Contributors. torch-autograd. https://github.com/twitter/torch-autograd, 2018. Accessed: 2018-09-25.

van Merrienboer, B., Breuleux, O., Bergeron, A., and Lamblin, P. Automatic differentiation in ml: Where we are and where we should be going. In *Advances in neural information processing systems*, 2018.

van Merrinboer, B., Wiltschko, A. B., and Moldovan, D. Tangent: automatic differentiation using source code transformation in python. 2017. URL https://arxiv.org/pdf/1711.02712.pdf.

Wang, F. and Rompf, T. A language and compiler view on differentiable programming, 2018. URL https://openreview.net/forum?id=SJxJtYkPG.

Wang, F., Wu, X., Essertel, G. M., Decker, J. M., and Rompf, T. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018.

Yang, J., Hance, T., Austin, T. H., Solar-Lezama, A., Flanagan, C., and Chong, S. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, pp. 631–647, New York, NY, USA, 2016. ACM.

A DYNAMIC RNN IMPLEMENTATION

Below is the hand-written graph implementation of the `tf.dynamic_rnn` cell.

```python
def dynamic_rnn(rnn_cell, input_data,
  initial_state, sequence_len=None):
  input_data = tf.transpose(input_data,
    (1, 0, 2))
  outputs = tf.TensorArray(
    tf.float32, size=0, dynamic_size=True)
  if sequence_length is None:
    max_len = input_data.shape[0]
  else:
    max_len = tf.reduce_max(sequence_len)
```

```python
def while_body(i, state, outputs):
  prev_state = state
  output, state = rnn_cell(
    input_data[i], state)
  state = tf.where(
    i < sequence_len,
    state,
    prev_state)
  outputs = outputs.write(i, output)
  return i + 1, state, outputs
def while_cond(i, state, outputs):
  return i < max_len
_, state, outputs = tf.while_loop(
    while_cond,
    while_body,
    loop_vars=(tf.constant(0),
               initial_state,
               outputs))
outputs = outputs.stack()
outputs = tf.transpose(outputs, (1, 0, 2))
return outputs, state
```

B ERROR HANDLING

In AutoGraph, there are three distinct steps of execution in addition to the usual syntax verification performed by the Python runtime:

- Conversion
- Staging (e.g., TensorFlow graph construction)
- Runtime (e.g., TensorFlow graph execution)

The latter two steps can be associated with the two stages in the multi-stage programming model that platforms like TensorFlow and PyTorch's JIT model implement. Each of these steps has distinct requirements for error handling, but principally make use of these two technologies:

- *Source map construction.* Each node in the AST, even after several passes of SCT, is associated to an original line of the user's Python code.
- *Error rewriting.* Several frames in the stack trace of TensorFlow code, especially AutoGraph-generated TensorFlow code, point to lines of code written by the AutoGraph compiler system rather than the user. We are able to reassociate temporary files (used when generating code in AutoGraph) to the user's original source files.

Conversion Errors   Conversion errors may occur due to code that is otherwise legal Python, but is unsupported by AutoGraph. These errors usually originate inside AutoGraph internal code.

For usability, such errors must indicate the location in the converted code of the idiom that caused the error. In addition, the error message must provide sufficient information to allow the developer to remedy the error. Lastly, the error stack trace should avoid references to internal code, as they

are typically uninformative to the user.

Currently, we facilitate this requirement by generating a stack-trace-like message that indicates the location of the error. In the future, we plan to further improve the conciseness of error messages of this type.

Staging Errors Staging errors can occur in successfully converted code and are typically raised because of disallowed or invalid argument types, shapes, hyperparameter values or other conditions that are only detectable at runtime. To address this, we plan to generate a stack-trace-like message with frames from the original code from which the intermediate code was generated. This is facilitated by the AST source map that we maintain between each node in the generated AST and the user's original source code.

Another challenge is that error messages may refer to generated symbols or to contexts specific to generated code. Addressing this shortcoming is a subject of future work.

Runtime Errors The name of this class of errors refers to the staged IR runtime.

For example, integer division by zero errors in TensorFlow:

```python
def f(n):
  return tf.constant(10, dtype=tf.int32) / n
```

The IR execution environment typically includes facilities to trace the source of the error to user code. However, in the case of AutoGraph, that will be generated code. To remedy this, we plan to intercept these errors and attach information that helps the user further trace the source of the error to original, pre-conversion code. We plan to enhance the user experience with the addition of `tf.function` in the TensorFlow 2.0 API.

C USEFUL UTILITIES

In order to build the system as described, we created a large library of source code transformation tools that we anticipate will be useful to the broader Python community.

Easy Code Quoting and Unquoting A few of the utility functions are listed below:

- `parser.parse_entity(fn_or_class)` takes a Python class or function and returns the corresponding AST node, wrapped in a containing `Module` node.
- `parser.parse_str(code_string)` is dentical to `parse_entity`, except takes a string of Python code as input. The string may contain any valid Python code.
- `pretty_printer.fmt(ast_node)` returns a pretty-printable string representing the AST.
- `compiler.ast_to_source(ast_node)` unparses an AST into the equivalent Python code,

returned as a string.
- `compiler.ast_to_object(ast_node)` compiles an AST into an equivalent Python entity, returned as a module.

For example:

```python
node = parse_str('a = b')
print(fmt(node))

# Output:
Module:
| body=[
| | Assign:
| | | targets=[
| | | | Name:
| | | | | id="a"
| | | | | ctx=Store()
| | | | | annotation=None
| | | ]
| | | value=Name:
| | | | id="b"
| | | | ctx=Load()
| | | | annotation=None
| ]
```

These utilities make it easy to make small modifications to the AST.

```python
node = parse_str('a = b')
node.body[0].value.id = 'c'
print(ast_to_source(node))

# Output:
a = c
```

Templated Code Rewriting Example:

```python
code_quote = '''
def fn(args):
  body
'''
new_body = textwrap.dedent('''
  a = x
  b = y
  return a + b
''')
node = templates.replace(
  code_quote,
  fn='my_function',
  args=('x', 'y'),
  body=parser.parse_str(new_body).body
)
print(compiler.ast_to_source(node))

# Output:
def my_function(x, y):
  a = x
  b = y
  return a + b
```

The function inserts string symbols or AST nodes into the quoted code template, and performs additional integrity

checks. This allows for the easy construction of complicated code blocks, especially with respect to building the AST manually.

D EXPANDED EXAMPLES

We expand on the toy examples in the main paper to illustrate AutoGraph's utility when implementing more realistic algorithms and models. These were implemented using TensorFlow's benchmark utilities[2] so that they can more easily be run. This also allows us to compare the performance of AutoGraph generated code to other reference implementations both from AutoGraph's authors and distributed as part of TensorFlow. We report some preliminary findings for each example.

All example code mentioned in this section, as well as the full runnable code for examples found through the paper, can be found at `https://github.com/tensorflow/autograph/examples/sysml2019`.

D.1 Beam Search

Beam search is an algorithm often used in machine translation. The algorithm builds candidate sequences by taking the most-likely steps at each transition, possibly discarding less-likely sequences. This is an interesting use-case for AutoGraph because beam search consists of complex computation and decisions at each step, with the number of steps capped at a the maximum sequence size. The simplest implementation of beam search is a loop that breaks if all candidate sequences have terminated. More robust implementations will separately keep track of living and terminal candidate sequences, and break if no living candidate has the potential to outscore a terminal candidate. Breaking out of the loop is essential to the performance of beam search since it often can generate sequences that are far shorter than the maximum allowable size.

We implemented beam search using TensorFlow Eager. Using AutoGraph, the benchmark runs between 2 and 3.2 times faster than the same code run using TensorFlow Eager. The improvement varies as we change the maximum sequence length and vocabulary size. Longer sequences and smaller vocabularies typically show more improvement when using AutoGraph. Longer sequences result in more iterations of the loop, so embedding these loops in the TensorFlow graph with AutoGraph shows more relative improvement. A larger vocabulary results in more expensive vector and matrix operations, taking longer overall.

D.2 L-BFGS

The L-BFGS (Limited-Memory BroydenFletcherGoldfarb-Shannon) algorithm is often used for parameter estimation in Machine Learning. Our implementation is based on the TensorFlow Eager implementation written by Yaroslav Bulatov[3]. In our benchmark, AutoGraph is almost 2 times faster than Eager with a batch size of 10 in approximately the same amount of code.

D.3 Model-Agnostic Meta-Learning (MAML)

Model-Agnostic Meta-Learning (MAML, Finn et al. (2017)) is an algorithm for meta-learning, especially effective for few-shot learning. Our benchmark is based on the sinusoidal example from Finn et al. (2017).[4]

We implemented our MAML benchmark using code that is compatible with both TensorFlow Eager AutoGraph. When training a single meta-parameter, the AutoGraph converted code ran 1.9 times faster than the identical code run in Eager mode. AutoGraph converted code was 2.7 times faster when training 10 meta-parameters.

D.4 seq2seq

The seq2seq (Sequence-to-Sequence) model[5] is a general purpose encoder and decoder that can be used for tasks like machine translation. We implemented this model and a benchmark that measures the performance of the model on random input sequences.

We implemented this benchmark in TensorFlow Eager and converted that Eager code using AutoGraph. AutoGraph converted code was 1.18 to 3.05 times faster than the Eager equivalent. The performance improvement varies with vocabulary size: AutoGraph performs better on larger vocabularies. Varying sequence length from 64 to 128 had minimal effect on the performance improvement. We also implemented optional "teacher forcing", which almost doubles the improvement gained from AutoGraph. This is because teacher-forcing reduces the amount of time spent performing computations, so the overhead of Eager mode is a larger percentage of the overall time. AutoGraph is designed to reduce such overhead, in this case by embedding data-dependent control flow in the graph executed by TensorFlow.

E SUPPORTED FEATURES

Tables 4, 5 and 6 show the features of Python and TensorFlow that are presently supported by AutoGraph.

---

[2]`https://www.tensorflow.org/community/benchmarks`

[3]`https://github.com/yaroslavvb/stuff/tree/master/eager_lbfgs`

[4]`https://github.com/cbfinn/maml`

[5]`https://google.github.io/seq2seq/`

		Conversion Triggers	Python Semantics	TensorFlow Semantics
Control Flow	if	cond is `Tensor`-like *or* a nest collection[a] of `Tensor`-like	extraneous side effects possible when object mutation is used[b]	`tf.cond`. Object mutation inside control flow limited to visible operations[c] all code paths must produce consistent value
	for	iterated is `Tensor`-like *or* iterated is nest collection of `Tensor`-like *or* iterated is `tf.Dataset` *or* iterated is `tf.QueueBase`	extraneous side effects possible when object mutation is used	`tf.while_loop` for `Tensor` and `tf.QueueBase` `Dataset.reduce` for `tf.Dataset`. All code paths must produce consistent value
	while	condition *closure* is collection of any `Tensor`-like[d]	preserved	`tf.while_loop`, all code paths must produce consistent value
	break / continue / return	lowered to conditional checks	preserved	`tf.cond`, all return values must have a consistent value.
	try / except / finally / raise	not converted[e]	preserved	n/a
	with	not converted	preserved	n/a
	yield	not allowed[f]	n/a	n/a
Operators	unary	argument is `Tensor`-like	preserved	corresponding TF op
	binary, arithmetic	not converted[g]	preserved	corresponding TF op
	binary, equality	either argument is `Tensor`-like	preserved	corresponding TF op
	binary, boolean	either argument is `Tensor`-like	preserved	lazy boolean[h] using `tf.cond`
	ternary conditional	either argument is `Tensor`-like	preserved	`tf.cond`

Table 4. AutoGraph Supported Features

[a] A "nest collection" is a collection that is recognized by `tf.nest`.

[b] Example: conditionally setting an attribute / item may be changed to always set that attribute / item. This is something we plan to remedy in TF 2 release.

[c] For example: attribute and item mutations done inside the control flow body preserve semantics. Mutations done in functions calls do not necessarily preserve semantics.

[d] When staging while loops using `tf.while_loop`, the condition of the loop is only evaluated by `tf.while_loop` itself. However, we need to determine whether the loop is will be staged or not *before* `tf.while_loop` is called. For this reason, we do not evaluate the loop condition beforehand to avoid causing any Python side effects triggered by the evaluation of the loop condition twice. In the future, we plan to evaluate the condition function twice and clearly document this semantic.

[e] There is no currently no support for catching exceptions in TensorFlow.

[f] Plans to support `yield` (without conversion) soon.

[g] Note however that `Tensor` objects typically overload all arithmetic operators and expressions will be staged into TF ops.

[h] For example, `x` `and` `y` is converted to `tf.cond(x,` `lambda:` `y,` `lambda:` `x)` to be consistent with Pythons lazy boolean evaluation semantics.

		Conversion Triggers	Python Semantics	TensorFlow Semantics
Functions	user-defined	(converted in recursive mode *and* not part of a whitelisted module[a]) *or* function is passed directly to AutoGraph[b]	preserved	inlined
	lambda	converted in recursive mode	preserved	inlined
	constructors	not converted	preserved	n/a
	instance methods	converted	output is unbound function[c]	inlined
	class methods	converted	output is unbound function	inlined
	built-in	converted: print, len, range, int, float	preserved	corresponding TF op[d]
	native	not converted	preserved	n/a
Collections	list literals	**experimental**; list is empty	preserved	low level Tensor list
	list append	target is low-level Tensor list *or target is tf.TensorArray*	preserved	low-level Tensor list push back, tf.TensorArray.write respectively
	list pop	target is low-level Tensor list	preserved	low-level Tensor list pop back
	other (dict, set, etc.)	not converted[e]	preserved	n/a
	get item / set item	target is Tensor *or target is* tf.TensorArray *or* target is low-level Tensor list	preserved	Tensor.__getitem__ / Tensor.__setitem__, tf.TensorArray.read / tf.TensorArray.write, low level Tensor list get item / set item respectively
Comprehensions		not converted	preserved	n/a

Table 5. AutoGraph Supported Features (continued)

[a]Currently, the whitelist includes the TF module.
[b]That is, user functions directly passed to to_graph or tf.function are always converted.
[c]That is, it is a function that takes self as first argument.
[d]Not all Python built-ins have a corresponding TF op.
[e]Plans to add support as corresponding TF ops are added.

		Conversion Triggers	Python Semantics	TensorFlow Semantics
Variables	undefined		reified with special value[a]	not allowed
	global	not allowed[b]	n/a	n/a
	nonlocal	not allowed[c]	n/a	n/a
Literals		not converted	preserved	n/a[d]
Classes	class types	**experimental**; class is passed directly to AutoGraph[e]	preserved	new class with all qualifying[f] methods converted
	objects	object is callable	preserved	object's `__call__` is converted
	get / set attribute	not converted	preserved	n/a
Decorators	user	converted in recursive mode	preserved	n/a
	built-in	not converted; some not allowed[g]	preserved	n/a
Generators		not allowed[h]	preserved	n/a
Power Features	exec	not supported	n/a	n/a
	pdb	not converted	partially[i]	n/a
	inspect	not converted	partially[j]	n/a

Table 6. AutoGraph Supported Features (continued)

[a] Long term plans to fully obey Python semantics and raise a runtime exception when undefined variables are accessed.
[b] Plans to support it soon.
[c] Plans to support it soon.
[d] Note that many TF ops autobox certain values to `Tensor`.
[e] That is, user classes directly passed to `to_graph` or `tf.function` are always converted.
[f] See User functions. For example, a subclass of a **Keras** Model class will only convert the methods defined in the subclass, not the methods inherited from the Model class.
[g] Examples: `functools.lru_cache` is not supported. `functools.wraps` is supported, but not converted.
[h] Plans to allow the use of generators, without conversion.
[i] The `pdb` calls will be inserted in the generated code, and will take effect at staging, when the graph is constructed.
[j] Some `inspect` APIs, like `getsource` work correctly, but we have not extensively tested them.

17