Hi — my name is Akshay Agrawal. I was at Google Brain from 2017 to 2018, and I'm now a PhD student here at Stanford, in mathematical optimization. I'm going to talk about TensorFlow Eager. TensorFlow Eager is a part of TensorFlow that makes it possible to mix imperative and declarative programming.

This is joint work with many people.

Differentiable programming

From DNNs to differentiable programs

The emphasis in machine learning research has shifted from building static DNNs to differentiating through complicated programs. These complicted programs might have data-dependent control flow, use complicated data structures, and so on.

Embedded DSLs

Libraries for differentiable programming ~ domain-specific languages (DSLs)

Because of this shift, libraries for machine learning are becoming more and more complicated. To the point that an argument has been made they are beginning to resemble domain-specific languages.

Embedded DSLs

Libraries for differentiable programming ~ domain-specific languages (DSLs)

Modern 'DSLs' embedded in host languages

Python (Chainer, Torch, Theano, TensorFlow), Julia (Flux), Swift (Swift for TensorFlow)

These so-called “DSLs" are "embedded" in host languages. The language is usually, but not always, embedded in Python.

Embedded DSLs

Libraries for differentiable programming ~ domain-specific languages (DSLs)

Modern 'DSLs' embedded in host languages

Python (Chainer, Torch, Theano, TensorFlow), Julia (Flux), Swift (Swift for TensorFlow)

Most are either imperative or declarative, in the programming languages sense

Some exceptions: MXNet, JAX, PyTorch 1.0

Most of these libraries offer one of two different programming styles. They're either imperative, where operations are executed immediately, or declarative, where programmers first construct dataflow graphs before executing them. There are some exceptions to this dichotomy, though, like MXNet, JAX, and PyTorch 1.0, which let you hybridize the two programming models.

Imperative DSLs

+ Full extent of host language
+ Rapid development
+ Familiar programming model

The advantages of imperative programming are obvious: it lets programmers enjoy the full extent of the host language, which in turn offers a familiar programming model and enables rapid development.

Imperative DSLs

+ Full extent of host language
+ Rapid development
+ Familiar programming model
- Performance bottlenecked on interpreter

A big drawback is that when you’re using a language like Python, performance can become bottlenecked on the interpreter.

Declarative DSLs

+ Bypass host language interpretation by staging
+ Compiler optimizations
+ Deployment, distribution, code generation, and serialization

There’s a reason that TensorFlow was designed around dataflow graphs. Separating the specification of your program from its execution, via graphs, means that you’re no longer bottlenecked on the interpreter. Graphs also let you do whole-program analysis and optimizations. This means that your runtime can exploit inter-op parallelism, automatically reuse buffers, apply compiler optimizations, and so on. Graphs also simplify deployment, distribution, code generation, and serialization.

Declarative DSLs

+ Bypass host language interpretation by staging
+ Compiler optimizations
+ Deployment, distribution, code generation, and serialization
- Limited programming constructs
- Steep learning curve

The main problem with graph-based libraries like TensorFlow is that they have steep learning curves. And they also limit the programmers ability to express themselves, since they typically don’t let you use native python control flow, data structures, or python debuggers.

Ideally, we'd have an imperative DSL that let you drop into graph execution, when needed.

Bridging the gap

TensorFlow Eager: mix imperative and staged execution in Python via multi-stage programming

The solution that TensorFlow Eager proposes is to mix imperative programming and graphs, in a way that can be interpreted as multi-stage programming. Multi-stage programming is a well-studied technique in programming languages. It can be discussed in a very formal way, but we’re going to keep the discussion at a very high level in this talk.

Bridging the gap

TensorFlow Eager: mix imperative and staged execution in Python via multi-stage programming

- not unique: MXNet, JAX, PyTorch 1.0, Snek+AutoGraph ...

TensorFlow Eager is not unique in doing this. Libraries like MXNet, JAX, PyTorch 1.0, and Snek+Autograph are all doing similar things.

TensorFlow Eager

Executes operations imperatively (or 'eagerly')

TFE executes operations imperatively, or eagerly, by default.

TensorFlow Eager

Executes operations imperatively (or 'eagerly')

A decorator, `@tf.function`, stages Python functions into *graph functions*

for performance, hardware acceleration, distribution, and serialization

For users that want any of the benefits that graphs provide, we provide a decorator that stages Python functions into *graph functions,* just-in-time. A graph function is just a TensorFlow graph with named inputs and ouputs.

TensorFlow Eager

Executes operations imperatively (or 'eagerly')

A decorator, `@tf.function`, stages Python functions into *graph functions*

for performance, hardware acceleration, distribution, and serialization

Implemented as an opt-in extension to TensorFlow 1.x

compatible with essentially all TensorFlow APIs

always enabled in TensorFlow 2.0

TFE is implemented as an opt-in extension to TF, today. And it’s compatible with essentially all TensorFlow APIs.

TFE is *always* enabled in TF 2.0.

Imperative execution

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



def add(a):
  return a + a

add(tf.ones([2, 2]))  #  [[2., 2.], [2., 2.]]
```


Here's a simple code example that shows what imperative-mode tensorflow eager looks like.

The second line of code coverts TensorFlow into TensorFlow Eager.

tf.ones() returns actual numerical data, not just a symbolic tensor … the add function returns the result of adding the ones vector to itself. It's basically NumPy.

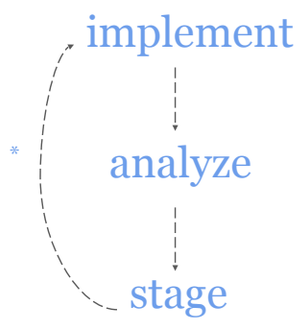
Graph functions

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

#  Calling `add` runs an op that computes an equivalent dataflow graph
@tf.function
def add(a):
  return a + a

add(tf.ones([2, 2]))  #  [[2., 2.], [2., 2.]]
add(tf.ones([2, 2], dtype=tf.int32))  #  functions are polymorphic
```


Adding a single line —- transforms the `add` function into a callable that executes an equivalent dataflow graph. From the programmer's perspective, executing a graph function is syntactically equivalent to executing the original python function. And, just like Python functions, graph functions are polymorphic in their inputs — even though TensorFlow graphs are definitely not polymoprhic.

Note that there are no TensorFlow Sesssions in sight. The tf.function decorator, and the callable returned from it, abstracts away both graph construction and the graph runtime.

Multi-stage workflow



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

At a high-level, this is the workflow we recommend to most users of TensorFlow Eager.

1. Implement in pure imperative mode.
2. Analyze the performance of your program, using whatever profilers or debuggers, you like. Find bottlenecks that consist of TensorFlow Eager code.
3. Use @tf.function to eliminate those bottlenecks.

Most people won't need to go onto steps 2 & 3, esp. if they use our high-level APIs ...

Imperative execution

I'll provide just a few details on how we've implemented imperative execution.

Operations and kernels

In TensorFlow ...

- primitive functions: *operations*
- implementations: *kernels*

TFE executes same operations, using same kernels

As a recap, in TensorFlow, primitive functions are called operations, and their implementations are called kernels.

Basically, TFE executes TensorFlow operations by instantiating their *kernels* right before they're needed and then executing them. This means that whether operations are executed as part of a graph function or eagerly, the same code is executed.

This is simple, conceptually. The details were complicated. Making Python TensorFlow compatible with eager execution required a lot of work, because our Python library code was designed for graph construction. But that's not the focus of this talk.

Graph functions

When transforming imperative code to graph functions, we try as much as possible to preserve the semantics of the original Python functions. Many of the details of graph functions I'll show you are motivated by this desire,

Graph functions

`@tf.function` : python function → graph function

JIT tracer, extracts TF graph

not a Python compiler

The @tf.function decorator is a just-in-time tracer. The first time the decorated function is run, it executes the Python function in a graph-construction context to generate an graph function. It is not a compiler for arbitrary Python functions.

Graph functions

`@tf.function` : python function → graph function

JIT tracer, extracts TF graph

not a Python compiler

`@tf.function(f)` is polymorphic

staging based on tensor types, run-time values of non-tensor args

see JAX, lightweight modular staging

The object returned by tf.function is polymorphic. It specializes its traces on the dtypes and shapes of tensor inputs, and the run-time values of non-tensor arguments. Essentially, when this object is called, it infers the signature of its inputs. If the signature doesn’t match something it’s seen before, then a new trace of the Python function is triggered. This means that multiple concrete graph functions might be generated for a single Python function. This kind of run-time specialization is similar to JAX & lightweight modular staging.

Graph functions

`@tf.function` : python function → graph function

JIT tracer, extracts TF graph

not a Python compiler

`@tf.function(f)` is polymorphic

staging based on tensor types, run-time values of non-tensor args

see JAX, lightweight modular staging

Supports lexical closure over tensors and `tf.Variable`s

And functions can lexically close over tensors and other TensorFlow Variables.

Automatic control dependencies

Functions preserve program-order semantics

```
a = tf.Variable(1.0)
b = tf.Variable(1.0)

@tf.function
def f(x, y):
  a.assign(y * b)
  b.assign_add(x * a)
  return a + b

f(1.0, 2.0)  # 10.0
```


When tensorflow variables are used in functions, we do a bit of extra work in order to ensure that ordering of reads and writes to variables in graph functions matches the Python ordering. When @tf.function generates graphs, it automatically inserts control dependencies that ensure all stateful ops are run, and that operations that modify the same resource run in program order. We try to insert the minimal amount of control dependencies required to do this, so that the runtime can still exploit parallelism in your graph.

Python control flow

`@tf.function` rewrites a subset of Python control flow into TF control flow

via AutoGraph — stick around for the next talk!

I lied, a little bit, when I said that tf.function was just a tracer. Even though @tf.function is not a compiler for arbitrary Python code, it can and does rewrite a subset of Python control flow into TF control flow. This is done via AutoGraph, which actually is the subject of the next talk. So make sure you stick around for that, it's a very cool technology.

Function runtime

Functions are graphs with named inputs and outputs

- instantiated in per-device *runtimes*
- Runtime automatically parallelizes and optimizes function

Functions are instantiated in per-device runtimes. The runtime automatically parallelizes and optimizes each function before running it. This is essentially the same execution model as classic, graph TensorFlow.

Function runtime

Functions are graphs with named inputs and outputs

- instantiated in per-device *runtimes*
- Runtime automatically parallelizes and optimizes function

Executed by operations, so ...

- functions are composable
- function execution is customizable

- Functions are executed by *operations,* just like any other primitive in tensorflow.
- This means that functions are trivially composable, since one function can include a function-call-operation that executes another function.
- // This fact can also, incidentally, help limit the size of graphs. For example, for distributed training, the coordinator's graph might just have N function call operations, one for each worker, instead of stamping out N copies of the //
- // same graph.
- The fact that functions are executed by operations also means that it's easy to customize the optimization and execution of functions by writing special function-call operations.

Multi-device functions

Special op executes a function across multiple devices

- reuses technology from TF Sessions (graph placer, partitioner, ...)
- variant used for load balancing TPU inference

- We use a special function call operation to execute a function across multiple devices.
- This function call essentially replicates a subset of TensorFlow Session inside an operation. When multi-device functions are instantiated in the runtime, we run the graph placer and partitioner on it.
- By extending this op, we created a special function call op whose device was configurable at run-time. This was useful for load balancing in production.

Automatic differentiation

- TensorFlow Eager would not be a library for differentiable programming if it didn’t have automatic differentiation.

Gradient tapes

Reverse-mode automatic differentiation, via tracing

TFE uses tracing-based reverse-mode automatic differentiation. In our programming model, programmers explicitly control the gradient trace, or tape, to limit overhead.

Gradient tapes

Reverse-mode automatic differentiation, via tracing

```
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
```


Here's a code example of using tapes to compute simple derivatives. The `watch` method marks a tensor as a quantity with respect to which we'll differentiate later. Notice how tapes can be nested, in order to compute higher-order derivatives.

Variables are watched automatically.

I should also mention that the tape is tightly integrated with functions. When a function is called under a tape, we modify the forward pass to output intermediate values as well and generate a function that computes the backwards pass.

programmers explicitly control the gradient trace (or *tape)*, to limit overhead

Jacobians

Automatic batching to compute Jacobians (`pfor`)

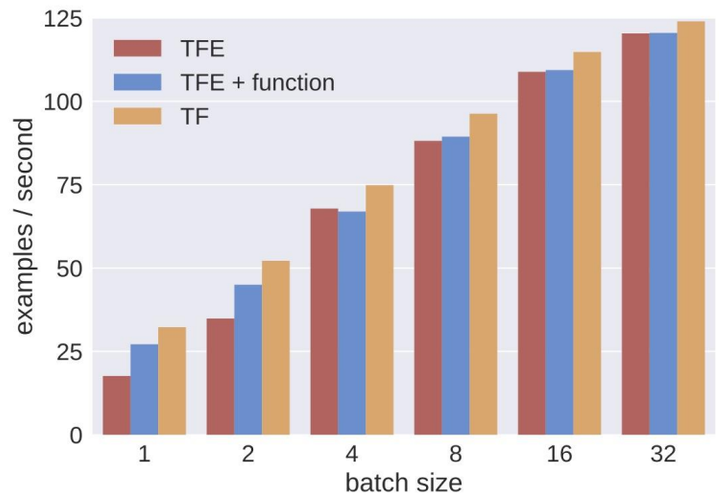
```
x = tf.constant([1., 2.])
y = tf.constant([3., 4.])

with tf.GradientTape() as t:
  t.watch(x)
  t.watch(y)
  z = x * x * y

t.jacobian(z, [x, y])  #  [tf.linalg.diag(2 * x * y), tf.linalg.diag(x * x)]
```


Here's something new, that's not actually in our paper. The GradientTape recently acquired the ability to compute Jacobians, and the computation is automatically batched using a parallel-for construct that was designed and implemented by Ashish.

Evaluation

I’ll show two experiments comparing the performance of TensorFlow Eager and classic graph-TensorFlow.

Training a ResNet-50*

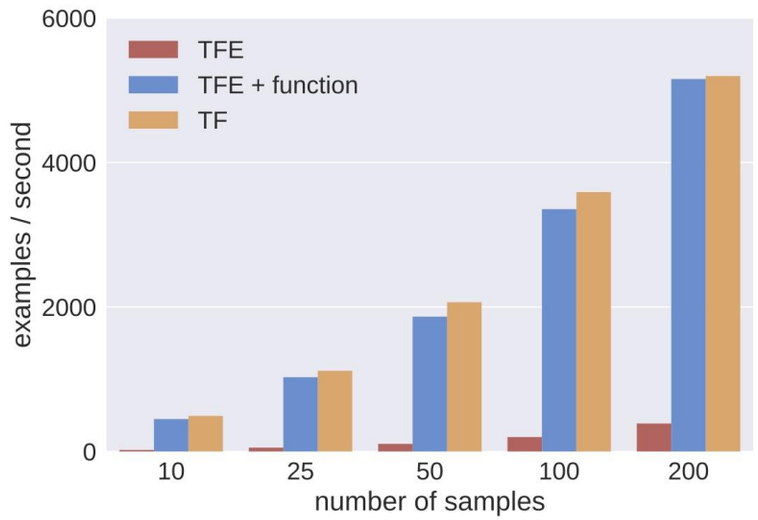


* as of Sept. 2018

This chart shows the examples per second achieved when training a ResNet-50 on ImageNet on a GPU. The red bar is purely imperative TFE; the blue bar is TFE, accelerated by the function decorator; and the orange bar is pure TF.

As the batch size increases, the difference between the three becomes small.

L2HMC*



* as of Sept. 2018

Where TFE takes a significant hit in performance compared to staged execution is when you're running programs with lots of small ops. This chart shows the examples/per second training learn-to-hamiltonian monte carlo, and you can see TFE is really struggling here. But strategically adding the function decorator makes TFE and TF essentially on par.

But, again, the data from Sept. 2018, so your mileage may vary …

Future work

Optimizations

Some performance optimizations are easier for graphs than for eager execution ...

Optimizations

Some performance optimizations are easier for graphs than for eager execution …

- Asynchronous execution
- Buffer reuse of temporaries
- SPMD across multiple devices

Stage all the code!

Represent more interesting data structures in graphs

AutoGraph will be instrumental for this task

We want to make more and more Python code possible to stage via tf.function. One big barrier to this is to represent more interesting data structures in graphs … again, we'll need help from our friends working on AutoGraph. Stay tuned!

See also …

Our paper for details on distribution, state management, …

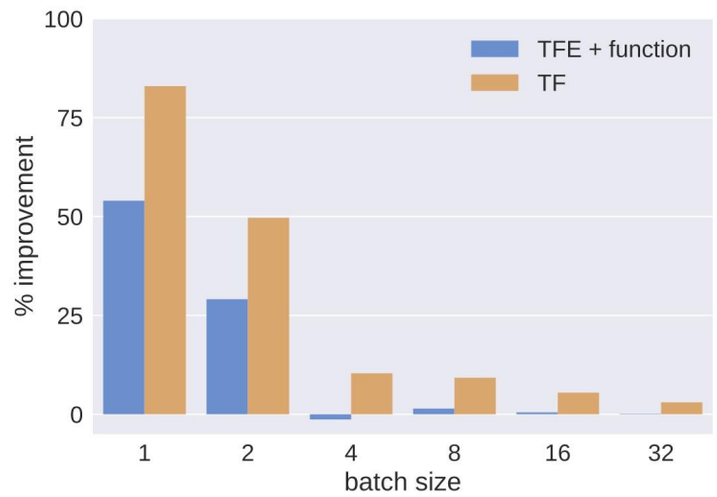
TensorFlow 2.0, in which TensorFlow Eager is always enabled

Contact

Alexandre Passos: apassos@google.com

Akshay Agrawal: akshayka@cs.stanford.edu

Training a ResNet-50*



* as of Sept. 2018

This slide is another way of looking at the Resnet-50 data. It shows the percent improvement in examples per second that TFE+function and plain TF have over TF. As the batch size increases, the improvement goes to zero ..

Training ResNet-50 on TPU

	1	2	4	8	16	32
TFE	1.06	1.99	4.3	8.4	16.6	30.3
tf.function	21.7	42.6	83.9	165.8	197.7	241.9

examples / second; not yet competitive with TF.

State

`@tf.function(f)` requires that `f` only creates `tf.Variable`s on its first call

```
class C: pass
obj = C(); obj.v = None

@tf.function
def g(x):
  if obj.v is None:
    obj.v = tf.Variable(1.0)
  return obj.v.assign_add(x)

g(1.0)  # 2.0
g(2.0)  # 4.0
```


Regarding the creation of state in functions, we had a long debate, and we decided that we should only allow tracing Python functions that create variables on their first invocation. The code on this slide is an example of a Python function that does just that.

This is common practice in machine learning code anyway.