TensorFlow 2.0 Beta is available **Learn more** (/beta/)

Eager Execution

Run in
Google (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/er
Colab

TensorFlow's eager execution is an imperative programming environment that evaluates operations immediately, without building graphs: operations return concrete values instead of constructing a computational graph to run later. This makes it easy to get started with TensorFlow and debug models, and it reduces boilerplate as well. To follow along with this guide, run the code samples below in an interactive `python` interpreter.

Eager execution is a flexible machine learning platform for research and experimentation, providing:

- *An intuitive interface*—Structure your code naturally and use Python data structures. Quickly iterate on small models and small data.
- *Easier debugging*—Call ops directly to inspect running models and test changes. Use standard Python debugging tools for immediate error reporting.
- *Natural control flow*—Use Python control flow instead of graph control flow, simplifying the specification of dynamic models.

Eager execution supports most TensorFlow operations and GPU acceleration. For a collection of examples running in eager execution, see: tensorflow/contrib/eager/python/examples (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/eager/python/examples)
.

Note: Some models may experience increased overhead with eager execution enabled. Performance improvements are ongoing, but please file a bug (https://github.com/tensorflow/tensorflow/issues) if you find a problem and share your benchmarks.

Setup and basic usage

To start eager execution, add `tf.enable_eager_execution()` (https://www.tensorflow.org/api_docs/python/tf/enable_eager_execution) to the beginning of the program or console session. Do not add this operation to other modules that the program calls.

```
from __future__ import absolute_import, division, print_function, unicode_litera

import tensorflow as tf

tf.enable_eager_execution()
```

Now you can run TensorFlow operations and the results will return immediately:

```
tf.executing_eagerly()
```

```
True
```

```
x = [[2.]]
m = tf.matmul(x, x)
print("hello, {}".format(m))
```

```
hello, [[4.]]
```

Enabling eager execution changes how TensorFlow operations behave—now they immediately evaluate and return their values to Python. **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects reference concrete values instead of symbolic handles to nodes in a computational graph. Since there isn't a computational graph to build and run later in a session, it's easy to inspect results using **`print()`** or a debugger. Evaluating, printing, and checking tensor values does not break the flow for computing gradients.

Eager execution works nicely with NumPy (http://www.numpy.org/). NumPy operations accept **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor) arguments. TensorFlow math operations (https://www.tensorflow.org/api_guides/python/math_ops) convert Python objects and NumPy arrays to **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects. The **`tf.Tensor.numpy`** method returns the object's value as a NumPy **`ndarray`**.

```
a = tf.constant([[1, 2],
                 [3, 4]])
print(a)
```

```
tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)
```

```
# Broadcasting support
b = tf.add(a, 1)
print(b)
```

```
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)
```

```
# Operator overloading is supported
print(a * b)
```

```
tf.Tensor(
[[ 2  6]
 [12 20]], shape=(2, 2), dtype=int32)
```

```
# Use NumPy values
import numpy as np

c = np.multiply(a, b)
print(c)
```

```
[[ 2  6]
 [12 20]]
```

```
# Obtain numpy value from a tensor:
print(a.numpy())
# => [[1 2]
#     [3 4]]
```

```
[[1 2]
 [3 4]]
```

The **`tf.contrib.eager`** (https://www.tensorflow.org/api_docs/python/tf/contrib/eager) module contains symbols available to both eager and graph execution environments and is useful for writing code to work with graphs (#work_with_graphs):

```
tfe = tf.contrib.eager
```

```
WARNING: Logging before flag parsing goes to stderr.
W0619 22:58:07.194561 140281377249024 lazy_loader.py:50]
The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-su
  * https://github.com/tensorflow/addons
  * https://github.com/tensorflow/io (for I/O related ops)
If you depend on functionality not listed there, please file an issue.
```

Dynamic control flow

A major benefit of eager execution is that all the functionality of the host language is available while your model is executing. So, for example, it is easy to write fizzbuzz (https://en.wikipedia.org/wiki/Fizz_buzz):

```
def fizzbuzz(max_num):
  counter = tf.constant(0)
  max_num = tf.convert_to_tensor(max_num)
  for num in range(1, max_num.numpy()+1):
    num = tf.constant(num)
    if int(num % 3) == 0 and int(num % 5) == 0:
      print('FizzBuzz')
    elif int(num % 3) == 0:
      print('Fizz')
    elif int(num % 5) == 0:
      print('Buzz')
    else:
      print(num.numpy())
    counter += 1
```

```
fizzbuzz(15)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
```

This has conditionals that depend on tensor values and it prints these values at runtime.

Build a model

Many machine learning models are represented by composing layers. When using TensorFlow with eager execution you can either write your own layers or use a layer provided in the **`tf.keras.layers`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers) package.

While you can use any Python object to represent a layer, TensorFlow has **`tf.keras.layers.Layer`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer) as a convenient base class. Inherit from it to implement your own layer:

```
class MySimpleLayer(tf.keras.layers.Layer):
  def __init__(self, output_units):
    super(MySimpleLayer, self).__init__()
    self.output_units = output_units

  def build(self, input_shape):
    # The build method gets called the first time your layer is used.
    # Creating variables on build() allows you to make their shape depend
    # on the input shape and hence removes the need for the user to specify
    # full shapes. It is possible to create variables during __init__() if
    # you already know their full shapes.
    self.kernel = self.add_variable(
      "kernel", [input_shape[-1], self.output_units])

  def call(self, input):
    # Override call() instead of __call__ so we can perform some bookkeeping.
    return tf.matmul(input, self.kernel)
```

Use **`tf.keras.layers.Dense`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense) layer instead of `MySimpleLayer` above as it has a superset of its functionality (it can also add a bias).

When composing layers into models you can use **`tf.keras.Sequential`** (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) to represent models which are a linear stack of layers. It is easy to use for basic models:

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(10, input_shape=(784,)),  # must declare input shape
  tf.keras.layers.Dense(10)
])
```

Alternatively, organize models in classes by inheriting from **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model). This is a container for layers that is a layer itself, allowing **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) objects to contain other **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) objects.

```
class MNISTModel(tf.keras.Model):
  def __init__(self):
    super(MNISTModel, self).__init__()
    self.dense1 = tf.keras.layers.Dense(units=10)
    self.dense2 = tf.keras.layers.Dense(units=10)

  def call(self, input):
    """Run the model."""
    result = self.dense1(input)
    result = self.dense2(result)
    result = self.dense2(result)  # reuse variables from dense2 layer
    return result

model = MNISTModel()
```

It's not required to set an input shape for the **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) class since the parameters are set the first time input is passed to the layer.

`tf.keras.layers` (https://www.tensorflow.org/api_docs/python/tf/keras/layers) classes create and contain their own model variables that are tied to the lifetime of their layer objects. To share layer variables, share their objects.

Eager training

Computing gradients

Automatic differentiation (https://en.wikipedia.org/wiki/Automatic_differentiation) is useful for implementing machine learning algorithms such as backpropagation (https://en.wikipedia.org/wiki/Backpropagation) for training neural networks. During eager execution, use **`tf.GradientTape`** (https://www.tensorflow.org/api_docs/python/tf/GradientTape) to trace operations for computing gradients later.

`tf.GradientTape` (https://www.tensorflow.org/api_docs/python/tf/GradientTape) is an opt-in feature to provide maximal performance when not tracing. Since different operations can occur during each call, all forward-pass operations get recorded to a "tape". To compute the gradient, play the tape backwards and then discard. A particular **`tf.GradientTape`** (https://www.tensorflow.org/api_docs/python/tf/GradientTape) can only compute one gradient; subsequent calls throw a runtime error.

```
w = tf.Variable([[1.0]])
with tf.GradientTape() as tape:
  loss = w * w
```

```
grad = tape.gradient(loss, w)
print(grad)  # => tf.Tensor([[ 2.]], shape=(1, 1), dtype=float32)
```

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

Train a model

The following example creates a multi-layer model that classifies the standard MNIST handwritten digits. It demonstrates the optimizer and layer APIs to build trainable graphs in an eager execution environment.

```
# Fetch and format the mnist data
(mnist_images, mnist_labels), _ = tf.keras.datasets.mnist.load_data()

dataset = tf.data.Dataset.from_tensor_slices(
  (tf.cast(mnist_images[...,tf.newaxis]/255, tf.float32),
   tf.cast(mnist_labels,tf.int64)))
dataset = dataset.shuffle(1000).batch(32)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
11493376/11490434 [==============================] - 0s 0us/step
```

```
# Build the model
mnist_model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(16,[3,3], activation='relu'),
  tf.keras.layers.Conv2D(16,[3,3], activation='relu'),
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(10)
])
```

Even without training, call the model and inspect the output in eager execution:

```
for images,labels in dataset.take(1):
  print("Logits: ", mnist_model(images[0:1]).numpy())
```

```
Logits:  [[-0.01264522  0.02846     0.01766706 -0.02451055 -0.00637072  0.011844
  -0.01011865  0.01680402 -0.01979431  0.01359601]]
```

While keras models have a builtin training loop (using the `fit` method), sometimes you need more customization. Here's an example, of a training loop implemented with eager:

```
optimizer = tf.train.AdamOptimizer()

loss_history = []
```

```
for (batch, (images, labels)) in enumerate(dataset.take(400)):
  if batch % 10 == 0:
    print('.', end='')
  with tf.GradientTape() as tape:
    logits = mnist_model(images, training=True)
    loss_value = tf.losses.sparse_softmax_cross_entropy(labels, logits)

  loss_history.append(loss_value.numpy())
  grads = tape.gradient(loss_value, mnist_model.trainable_variables)
  optimizer.apply_gradients(zip(grads, mnist_model.trainable_variables),
                            global_step=tf.train.get_or_create_global_step())
```

```
........................................
```

```
import matplotlib.pyplot as plt

plt.plot(loss_history)
plt.xlabel('Batch #')
plt.ylabel('Loss [entropy]')
```

```
Text(0, 0.5, 'Loss [entropy]')
```

Variables and optimizers

`tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) objects store mutable **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor) values accessed during training to make automatic differentiation easier. The parameters of a model can be encapsulated in classes as variables.

Better encapsulate model parameters by using **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) with **`tf.GradientTape`** (https://www.tensorflow.org/api_docs/python/tf/GradientTape). For example, the automatic differentiation example above can be rewritten:

```
class Model(tf.keras.Model):
  def __init__(self):
    super(Model, self).__init__()
    self.W = tf.Variable(5., name='weight')
    self.B = tf.Variable(10., name='bias')
  def call(self, inputs):
    return inputs * self.W + self.B

# A toy dataset of points around 3 * x + 2
NUM_EXAMPLES = 2000
training_inputs = tf.random_normal([NUM_EXAMPLES])
noise = tf.random_normal([NUM_EXAMPLES])
training_outputs = training_inputs * 3 + 2 + noise

# The loss function to be optimized
def loss(model, inputs, targets):
  error = model(inputs) - targets
  return tf.reduce_mean(tf.square(error))

def grad(model, inputs, targets):
  with tf.GradientTape() as tape:
    loss_value = loss(model, inputs, targets)
  return tape.gradient(loss_value, [model.W, model.B])

# Define:
# 1. A model.
# 2. Derivatives of a loss function with respect to model parameters.
# 3. A strategy for updating the variables based on the derivatives.
model = Model()
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)

print("Initial loss: {:.3f}".format(loss(model, training_inputs, training_output

# Training loop
for i in range(300):
  grads = grad(model, training_inputs, training_outputs)
  optimizer.apply_gradients(zip(grads, [model.W, model.B]),
                            global_step=tf.train.get_or_create_global_step())
  if i % 20 == 0:
    print("Loss at step {:03d}: {:.3f}".format(i, loss(model, training_inputs, t

print("Final loss: {:.3f}".format(loss(model, training_inputs, training_outputs)
print("W = {}, B = {}".format(model.W.numpy(), model.B.numpy()))
```

```
Initial loss: 69.477
Loss at step 000: 66.765
Loss at step 020: 30.311
Loss at step 040: 14.065
Loss at step 060: 6.826
Loss at step 080: 3.599
Loss at step 100: 2.161
Loss at step 120: 1.521
Loss at step 140: 1.235
Loss at step 160: 1.108
Loss at step 180: 1.051
Loss at step 200: 1.026
Loss at step 220: 1.015
Loss at step 240: 1.010
```

Use objects for state during eager execution

With graph execution, program state (such as the variables) is stored in global collections and their lifetime is managed by the **`tf.Session`** (https://www.tensorflow.org/api_docs/python/tf/Session) object. In contrast, during eager execution the lifetime of state objects is determined by the lifetime of their corresponding Python object.

Variables are objects

During eager execution, variables persist until the last reference to the object is removed, and is then deleted.

```
if tf.test.is_gpu_available():
  with tf.device("gpu:0"):
    v = tf.Variable(tf.random_normal([1000, 1000]))
    v = None  # v no longer takes up GPU memory
```

Object-based saving

`tf.train.Checkpoint` (https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint) can save and restore **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable)s to and from checkpoints:

```
x = tf.Variable(10.)
checkpoint = tf.train.Checkpoint(x=x)
```

```
x.assign(2.)   # Assign a new value to the variables and save.
checkpoint_path = './ckpt/'
checkpoint.save('./ckpt/')
```

```
'./ckpt/-1'
```

```
x.assign(11.)  # Change the variable after saving.

# Restore values from the checkpoint
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_path))

print(x)  # => 2.0
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=2.0>
```

To save and load models, **`tf.train.Checkpoint`** (https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint) stores the internal state of objects, without requiring hidden variables. To record the state of a **`model`**, an **`optimizer`**, and a global step, pass them to a **`tf.train.Checkpoint`** (https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint):

```
import os
import tempfile

model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(16,[3,3], activation='relu'),
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(10)
])
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
checkpoint_dir = tempfile.mkdtemp()
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
root = tf.train.Checkpoint(optimizer=optimizer,
                           model=model,
                           optimizer_step=tf.train.get_or_create_global_step())

root.save(checkpoint_prefix)
root.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f90835d5400
```

Object-oriented metrics

`tfe.metrics` are stored as objects. Update a metric by passing the new data to the callable, and retrieve the result using the **`tfe.metrics.result`** method, for example:

```
m = tfe.metrics.Mean("loss")
m(0)
m(5)
m.result()  # => 2.5
m([8, 9])
m.result()  # => 5.5
```

```
<tf.Tensor: id=62570, shape=(), dtype=float64, numpy=5.5>
```

Summaries and TensorBoard

TensorBoard (https://www.tensorflow.org/guide/summaries_and_tensorboard) is a visualization tool for understanding, debugging and optimizing the model training process. It uses summary events that are written while executing the program.

`tf.contrib.summary` (https://www.tensorflow.org/api_docs/python/tf/contrib/summary) is compatible with both eager and graph execution environments. Summary operations, such as **`tf.contrib.summary.scalar`** (https://www.tensorflow.org/api_docs/python/tf/contrib/summary/scalar), are inserted during model construction. For example, to record summaries once every 100 global steps:

```
global_step = tf.train.get_or_create_global_step()

logdir = "./tb/"
writer = tf.contrib.summary.create_file_writer(logdir)
writer.set_as_default()

for _ in range(10):
  global_step.assign_add(1)
  # Must include a record_summaries method
  with tf.contrib.summary.record_summaries_every_n_global_steps(100):
    # your model code goes here
    tf.contrib.summary.scalar('global_step', global_step)
```

```
!ls tb/
```

```
events.out.tfevents.1560985098.kokoro-image-debugging.v2
```

Advanced automatic differentiation topics

Dynamic models

`tf.GradientTape` (https://www.tensorflow.org/api_docs/python/tf/GradientTape) can also be used in dynamic models. This example for a backtracking line search (https://wikipedia.org/wiki/Backtracking_line_search) algorithm looks like normal NumPy code, except there are gradients and is differentiable, despite the complex control flow:

```
def line_search_step(fn, init_x, rate=1.0):
  with tf.GradientTape() as tape:
    # Variables are automatically recorded, but manually watch a tensor
    tape.watch(init_x)
    value = fn(init_x)
  grad = tape.gradient(value, init_x)
  grad_norm = tf.reduce_sum(grad * grad)
  init_value = value
  while value > init_value - rate * grad_norm:
    x = init_x - rate * grad
    value = fn(x)
    rate /= 2.0
  return x, value
```

Additional functions to compute gradients

`tf.GradientTape` (https://www.tensorflow.org/api_docs/python/tf/GradientTape) is a powerful interface for computing gradients, but there is another Autograd (https://github.com/HIPS/autograd)-style API available for automatic differentiation. These functions are useful if writing math code with only tensors and gradient functions, and without `tf.variables`:

- `tfe.gradients_function` —Returns a function that computes the derivatives of its input function parameter with respect to its arguments. The input function parameter must return a scalar value. When the returned function is invoked, it returns a list of `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects: one element for each argument of the input function. Since anything of interest must be passed as a function parameter, this becomes unwieldy if there's a dependency on many trainable parameters.
- `tfe.value_and_gradients_function` —Similar to `tfe.gradients_function`, but when the returned function is invoked, it returns the value from the input function in addition to the list of derivatives of the input function with respect to its arguments.

In the following example, `tfe.gradients_function` takes the `square` function as an argument and returns a function that computes the partial derivatives of `square` with respect to its inputs. To calculate the derivative of `square` at `3`, `grad(3.0)` returns `6`.

```
def square(x):
  return tf.multiply(x, x)

grad = tfe.gradients_function(square)
```

```
square(3.).numpy()
```

```
9.0
```

```
grad(3.)[0].numpy()
```

```
6.0
```

```
# The second-order derivative of square:
gradgrad = tfe.gradients_function(lambda x: grad(x)[0])
gradgrad(3.)[0].numpy()
```

```
2.0
```

```
# The third-order derivative is None:
gradgradgrad = tfe.gradients_function(lambda x: gradgrad(x)[0])
gradgradgrad(3.)
```

```
[None]
```

```
# With flow control:
def abs(x):
  return x if x > 0. else -x

grad = tfe.gradients_function(abs)
```

```
grad(3.)[0].numpy()
```

```
1.0
```

```
grad(-3.)[0].numpy()
```

```
-1.0
```

Custom gradients

Custom gradients are an easy way to override gradients in eager and graph execution. Within the forward function, define the gradient with respect to the inputs, outputs, or intermediate results. For example, here's an easy way to clip the norm of the gradients in the backward pass:

```
@tf.custom_gradient
def clip_gradient_by_norm(x, norm):
  y = tf.identity(x)
  def grad_fn(dresult):
    return [tf.clip_by_norm(dresult, norm), None]
  return y, grad_fn
```

Custom gradients are commonly used to provide a numerically stable gradient for a sequence of operations:

```
def log1pexp(x):
  return tf.log(1 + tf.exp(x))
grad_log1pexp = tfe.gradients_function(log1pexp)
```

```
# The gradient computation works fine at x = 0.
grad_log1pexp(0.)[0].numpy()
```

```
0.5
```

```
# However, x = 100 fails because of numerical instability.
grad_log1pexp(100.)[0].numpy()
```

```
nan
```

Here, the **`log1pexp`** function can be analytically simplified with a custom gradient. The implementation below reuses the value for **`tf.exp(x)`** (https://www.tensorflow.org/api_docs/python/tf/math/exp) that is computed during the forward pass—making it more efficient by eliminating redundant calculations:

```
@tf.custom_gradient
def log1pexp(x):
  e = tf.exp(x)
  def grad(dy):
    return dy * (1 - 1 / (1 + e))
  return tf.log(1 + e), grad

grad_log1pexp = tfe.gradients_function(log1pexp)
```

```
# As before, the gradient computation works fine at x = 0.
grad_log1pexp(0.)[0].numpy()
```

```
0.5
```

```
# And the gradient computation also works at x = 100.
grad_log1pexp(100.)[0].numpy()
```

```
1.0
```

Performance

Computation is automatically offloaded to GPUs during eager execution. If you want control over where a computation runs you can enclose it in a **`tf.device('/gpu:0')`** block (or the CPU equivalent):

```
import time

def measure(x, steps):
  # TensorFlow initializes a GPU the first time it's used, exclude from timing.
  tf.matmul(x, x)
  start = time.time()
  for i in range(steps):
    x = tf.matmul(x, x)
  # tf.matmul can return before completing the matrix multiplication
  # (e.g., can return after enqueing the operation on a CUDA stream).
  # The x.numpy() call below will ensure that all enqueued operations
  # have completed (and will also copy the result to host memory,
  # so we're including a little more than just the matmul operation
  # time).
  _ = x.numpy()
  end = time.time()
  return end - start

shape = (1000, 1000)
steps = 200
print("Time to multiply a {} matrix by itself {} times:".format(shape, steps))

# Run on CPU:
with tf.device("/cpu:0"):
  print("CPU: {} secs".format(measure(tf.random_normal(shape), steps)))

# Run on GPU, if available:
if tfe.num_gpus() > 0:
  with tf.device("/gpu:0"):
    print("GPU: {} secs".format(measure(tf.random_normal(shape), steps)))
else:
  print("GPU: not found")
```

```
Time to multiply a (1000, 1000) matrix by itself 200 times:
CPU: 0.814579963684082 secs
```

```
GPU: 0.03975391387939453 secs
```

A **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor) object can be copied to a different device to execute its operations:

```
if tf.test.is_gpu_available():
  x = tf.random_normal([10, 10])

  x_gpu0 = x.gpu()
  x_cpu = x.cpu()

  _ = tf.matmul(x_cpu, x_cpu)    # Runs on CPU
  _ = tf.matmul(x_gpu0, x_gpu0)  # Runs on GPU:0

  if tfe.num_gpus() > 1:
    x_gpu1 = x.gpu(1)
    _ = tf.matmul(x_gpu1, x_gpu1)  # Runs on GPU:1
```

Benchmarks

For compute-heavy models, such as ResNet50 (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/eager/python/examples/resnet50) training on a GPU, eager execution performance is comparable to graph execution. But this gap grows larger for models with less computation and there is work to be done for optimizing hot code paths for models with lots of small operations.

Work with graphs

While eager execution makes development and debugging more interactive, TensorFlow graph execution has advantages for distributed training, performance optimizations, and production deployment. However, writing graph code can feel different than writing regular Python code and more difficult to debug.

For building and training graph-constructed models, the Python program first builds a graph representing the computation, then invokes **`Session.run`** (https://www.tensorflow.org/api_docs/python/tf/InteractiveSession#run) to send the graph for execution on the C++-based runtime. This provides:

- Automatic differentiation using static autodiff.
- Simple deployment to a platform independent server.
- Graph-based optimizations (common subexpression elimination, constant-folding, etc.).
- Compilation and kernel fusion.
- Automatic distribution and replication (placing nodes on the distributed system).

Deploying code written for eager execution is more difficult: either generate a graph from the model, or run the Python runtime and code directly on the server.

Write compatible code

The same code written for eager execution will also build a graph during graph execution. Do this by simply running the same code in a new Python session where eager execution is not enabled.

Most TensorFlow operations work during eager execution, but there are some things to keep in mind:

- Use `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) for input processing instead of queues. It's faster and easier.
- Use object-oriented layer APIs—like `tf.keras.layers` (https://www.tensorflow.org/api_docs/python/tf/keras/layers) and `tf.keras.Model` (https://www.tensorflow.org/api_docs/python/tf/keras/Model)—since they have explicit storage for variables.
- Most model code works the same during eager and graph execution, but there are exceptions. (For example, dynamic models using Python control flow to change the computation based on inputs.)
- Once eager execution is enabled with `tf.enable_eager_execution` (https://www.tensorflow.org/api_docs/python/tf/enable_eager_execution), it cannot be turned off. Start a new Python session to return to graph execution.

It's best to write code for both eager execution *and* graph execution. This gives you eager's interactive experimentation and debuggability with the distributed performance benefits of graph execution.

Write, debug, and iterate in eager execution, then import the model graph for production deployment. Use **`tf.train.Checkpoint`** (https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint) to save and restore model variables, this allows movement between eager and graph execution environments. See the examples in: tensorflow/contrib/eager/python/examples (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/eager/python/examples)
.

Use eager execution in a graph environment

Selectively enable eager execution in a TensorFlow graph environment using **`tfe.py_func`**. This is used when **`tf.enable_eager_execution()`** (https://www.tensorflow.org/api_docs/python/tf/enable_eager_execution) has *not* been called.

```
def my_py_func(x):
  x = tf.matmul(x, x)  # You can use tf ops
  print(x)  # but it's eager!
  return x

with tf.Session() as sess:
  x = tf.placeholder(dtype=tf.float32)
  # Call eager function in graph!
  pf = tfe.py_func(my_py_func, [x], tf.float32)

  sess.run(pf, feed_dict={x: [[2.0]]})  # [[4.0]]
```

```
tf.Tensor([[4.]], shape=(1, 1), dtype=float32)
```