eager execution

- With eager execution enabled, computation is not automatically offloaded to GPUs. Instead, you must explicitly specify when GPUs should be used. To utilize the GPU, place the code above within a `with tf.device("/gpu:0"):` block.
- `tfe.num_gpus()` returns the number of available GPUs.
- Methods on the `Tensor` object can be used to explicitly copy the `Tensor` to a different device. Operations are typically executed on the device on which the inputs are placed.
- For example:

```
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
```

- TensorFlow eager execution provides an autograd-style API for automatic differentiation. Specifically, the functions:
  - `tfe.gradients_function(f)`: Returns a Python function that computes the derivatives of the Python function `f` with respect to its arguments. `f` must return a scalar value. When the returned function is invoked, it returns a list of Tensor objects (one element for each argument of `f`).
  - `tfe.value_and_gradients_function(f)`: Similar to `tfe.gradients_function`, except that when the returned function is invoked, it returns the value of `f` in addition to the list of derivatives of `f` with respect to its arguments.
- `tfe.gradients_function(f)` introduced earlier computes the derivatives of f with respect to its arguments. However, it requires all parameters of interest to be arguments of f, which becomes cumbersome when f depends on a large number of trainable parameters. `tfe.implicit_gradients` is an alternative function with some useful properties: It computes the derivatives of f with respect to all the `tfe.Variables` used by f. When the returned function is invoked, it returns a list of `(gradient value, Variable object)` tuples.
- Instead of using one of these explicit gradient computation functions, its better to define a model as a python class and use the `tfe.implicit_gradients` function to perform optimization using SGD or using one of the other fancier optimizers.
- See the MNISTModel example here.
- TensorFlow Variables (`tfe.Variable`) provides a way to represent shared, persistent state of your model. The `tfe.Saver` class (which is a thin wrapper over the `tf.train.Saver` class) provides a means to save and restore variables to and from *checkpoints*.

- For example:

```
# Create variables.
x = tfe.Variable(10., name='x')
y = tfe.Variable(5., name='y')

# Create a Saver.
saver = tfe.Saver([x, y])

# Assign new values to the variables and save.
x.assign(2.)
saver.save('/tmp/ckpt')

# Change the variable after saving.
x.assign(11.)
assert 16. == (x + y).numpy()  # 11 + 5

# Restore the values in the checkpoint.
saver.restore('/tmp/ckpt')

assert 7. == (x + y).numpy()  # 2 + 5
```

- You may often want to organize your models using classes, like the `MNISTModel` class described above. We recommend inheriting from the `tfe.Network` class as it provides conveniences like keeping track of all model variables and methods to save and restore from checkpoints.
- Sub-classes of `tfe.Network` may register `Layer`s (like classes in `tf.layers`, or Keras layers) using a call to `self.track_layer()` and define the computation in an implementation of `call()`.
- The `tfe.Network` class is itself a sub-class of `tf.layers.Layer`. This allows instances of `tfe.Network` to be embedded in other networks.
- `tfe.Saver` in combination with `tfe.restore_variables_on_create` provides a convenient way to save and load checkpoints without changing the program once the checkpoint has been created. For example, we can set an objective for the output of our network, choose an optimizer, and a location for the checkpoint:

```
objective = tf.constant([[2., 3., 4., 5.]])
optimizer = tf.train.AdamOptimizer(0.01)
checkpoint_directory = '/tmp/tfe_example'
checkpoint_prefix = os.path.join(checkpoint_directory, 'ckpt')
net = ThreeLayerNet()
```

- Note that variables have not been created yet. We want them to be restored from a checkpoint, if one exists, so we create them inside a `tfe.restore_variables_on_create` context manager. Then our training loop is the same whether starting training or resuming from a previous checkpoint:

```
with tfe.restore_variables_on_create(
```

```
    tf.train.latest_checkpoint(checkpoint_directory)):
  global_step = tf.train.get_or_create_global_step()
  for _ in range(100):
    loss_fn = lambda: tf.norm(net(inp) - objective)
    optimizer.minimize(loss_fn, global_step=global_step)
    if tf.equal(global_step % 20, 0):
      print("Step %d, output %s" % (global_step.numpy(),
                                    net(inp).numpy()))
      all_variables = (
          net.variables
          + tfe.get_optimizer_variables(optimizer)
          + [global_step])
      # Save the checkpoint.
      tfe.Saver(all_variables).save(checkpoint_prefix, global_step=global_st
ep)
```

- `tf.summary` operations are not compatible with eager execution, but an equivalent alternative exists in `tf.contrib.summary` that is compatible with both eager execution and graph construction.
- During model construction simply insert summary ops like `tf.contrib.summary.scalar`. These operations do nothing by default, unless a summary writer is currently active and a writing policy is set.
- Example:

```
tf.train.get_or_create_global_step()  # Ensuring the global step variable ex
ists
writer = tf.contrib.summary.create_summary_file_writer(logdir)

for _ in range(iterations):
  with writer.as_default():
    with tf.contrib.summary.record_summaries_every_n_global_steps(100):
      # your model code goes here
      tf.contrib.summary.scalar('loss', loss)
      # ...
```

- Similarly to summaries, the metrics in `tf.metrics` are currently not compatible with eager execution. We instead provide object-oriented metrics in the `tfe.metrics` package.
- When eager execution is enabled, the discussion on iterator creation using `make_one_shot_iterator()` and `get_next()` in the Programmer's Guide is *not* applicable. Instead, a more Pythonic `Iterator` class is available.
- For example:

```
# Create a source Dataset from in-memory numpy arrays.
# For reading from files on disk, you may want to use other Dataset classes
# like the TextLineDataset or the TFRecordDataset.
dataset = tf.data.Dataset.from_tensor_slices([1, 2, 3, 4, 5, 6])
```

```
5
6  # Apply transformations, shuffling, batching etc.
7  dataset = dataset.map(tf.square).shuffle(2).batch(2)
8
9  # Use tfe.Iterator to iterate over the dataset.
10 for x in tfe.Iterator(dataset):
11   print(x)
```

- Some differences worth noting between session-based tensorflow and eager tensorflow:
  - There is no notion of a `tf.placeholder` or a `tf.Session` when eager execution is enabled.
  - Many properties on the `tf.Tensor` object, like `tf.Tensor.name`, `tf.Tensor.op`, `tf.Tensor.inputs` are not meaningful when eager execution is enabled and their use will raise an `AttributeError`.
  - Some API calls (such as the functional-style `tf.layers.dense`, `tf.layers.conv2d`) are not compatible with eager execution. Use of such methods should raise an error indicating the alternative (e.g., the `tf.layers.Dense` and `tf.layers.Conv2D` classes).