Actors

Remote functions in Ray should be thought of as functional and side-effect free. Restricting ourselves only to remote functions gives us distributed functional programming, which is great for many use cases, but in practice is a bit limited.

Ray extends the dataflow model with **actors**. An actor is essentially a stateful worker (or a service). When a new actor is instantiated, a new worker is created, and methods of the actor are scheduled on that specific worker and can access and mutate the state of that worker.

Suppose we've already started Ray.

```
import ray
ray.init()
```

Defining and creating an actor

Consider the following simple example. The `ray.remote` decorator indicates that instances of the `Counter` class will be actors.

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value
```

To actually create an actor, we can instantiate this class by calling `Counter.remote()`.

```
a1 = Counter.remote()
a2 = Counter.remote()
```

When an actor is instantiated, the following events happen.

1. A node in the cluster is chosen and a worker process is created on that node (by the local scheduler on that node) for the purpose of running methods called on the actor.
2. A `Counter` object is created on that worker and the `Counter` constructor is run.

Using an actor

We can schedule tasks on the actor by calling its methods.

```
a1.increment.remote()  # ray.get returns 1
a2.increment.remote()  # ray.get returns 1
```

When `a1.increment.remote()` is called, the following events happens.

1. A task is created.
2. The task is assigned directly to the local scheduler responsible for the actor by the driver's local scheduler. Thus, this scheduling procedure bypasses the global scheduler.
3. An object ID is returned.

We can then call `ray.get` on the object ID to retrieve the actual value.

Similarly, the call to `a2.increment.remote()` generates a task that is scheduled on the second `Counter` actor. Since these two tasks run on different actors, they can be executed in parallel (note that only actor methods will be scheduled on actor workers, regular remote functions will not be).

On the other hand, methods called on the same `Counter` actor are executed serially in the order that they are called. They can thus share state with one another, as shown below.

```
# Create ten Counter actors.
counters = [Counter.remote() for _ in range(10)]

# Increment each Counter once and get the results. These tasks all happen in
# parallel.
results = ray.get([c.increment.remote() for c in counters])
print(results)  # prints [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

# Increment the first Counter five times. These tasks are executed serially
# and share state.
results = ray.get([counters[0].increment.remote() for _ in range(5)])
print(results)  # prints [2, 3, 4, 5, 6]
```

A More Interesting Actor Example

A common pattern is to use actors to encapsulate the mutable state managed by an external library or service.

Gym provides an interface to a number of simulated environments for testing and training reinforcement learning agents. These simulators are stateful, and tasks that use these simulators must mutate their state. We can use actors to encapsulate the state of these simulators.

```
import gym

@ray.remote
class GymEnvironment(object):
    def __init__(self, name):
        self.env = gym.make(name)
        self.env.reset()

    def step(self, action):
        return self.env.step(action)

    def reset(self):
        self.env.reset()
```

We can then instantiate an actor and schedule a task on that actor as follows.

```
pong = GymEnvironment.remote("Pong-v0")
pong.step.remote(0)  # Take action 0 in the simulator.
```

Using GPUs on actors

A common use case is for an actor to contain a neural network. For example, suppose we have imported Tensorflow and have created a method for constructing a neural net.

```
import tensorflow as tf

def construct_network():
    x = tf.placeholder(tf.float32, [None, 784])
    y_ = tf.placeholder(tf.float32, [None, 10])

    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.nn.softmax(tf.matmul(x, W) + b)

    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return x, y_, train_step, accuracy
```

We can then define an actor for this network as follows.

```
import os

# Define an actor that runs on GPUs. If there are no GPUs, then simply use
# ray.remote without any arguments and no parentheses.
@ray.remote(num_gpus=1)
class NeuralNetOnGPU(object):
    def __init__(self):
        # Set an environment variable to tell TensorFlow which GPUs to use. Note
        # that this must be done before the call to tf.Session.
        os.environ["CUDA_VISIBLE_DEVICES"] = ",".join([str(i) for i in ray.get_gpu_ids()])
        with tf.Graph().as_default():
            with tf.device("/gpu:0"):
                self.x, self.y_, self.train_step, self.accuracy = construct_network()
                # Allow this to run on CPUs if there aren't any GPUs.
                config = tf.ConfigProto(allow_soft_placement=True)
                self.sess = tf.Session(config=config)
                # Initialize the network.
                init = tf.global_variables_initializer()
                self.sess.run(init)
```

To indicate that an actor requires one GPU, we pass in `num_gpus=1` to `ray.remote`. Note that in order for this to work, Ray must have been started with some GPUs, e.g., via `ray.init(num_gpus=2)`. Otherwise, when you try to instantiate the GPU version with `NeuralNetOnGPU.remote()`, an exception will be thrown saying that there aren't enough GPUs in the system.

When the actor is created, it will have access to a list of the IDs of the GPUs that it is allowed to use via `ray.get_gpu_ids()`. This is a list of integers, like `[]`, or `[1]`, or `[2, 5, 6]`. Since we passed in `ray.remote(num_gpus=1)`, this list will have length one.

We can put this all together as follows.

```
import os
import ray
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

ray.init(num_gpus=8)

def construct_network():
    x = tf.placeholder(tf.float32, [None, 784])
    y_ = tf.placeholder(tf.float32, [None, 10])

    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.nn.softmax(tf.matmul(x, W) + b)

    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return x, y_, train_step, accuracy

@ray.remote(num_gpus=1)
class NeuralNetOnGPU(object):
    def __init__(self, mnist_data):
        self.mnist = mnist_data
        # Set an environment variable to tell TensorFlow which GPUs to use. Note
        # that this must be done before the call to tf.Session.
        os.environ["CUDA_VISIBLE_DEVICES"] = ",".join([str(i) for i in ray.get_gpu_ids()])
        with tf.Graph().as_default():
            with tf.device("/gpu:0"):
                self.x, self.y_, self.train_step, self.accuracy = construct_network()
                # Allow this to run on CPUs if there aren't any GPUs.
                config = tf.ConfigProto(allow_soft_placement=True)
                self.sess = tf.Session(config=config)
                # Initialize the network.
                init = tf.global_variables_initializer()
                self.sess.run(init)

    def train(self, num_steps):
        for _ in range(num_steps):
            batch_xs, batch_ys = self.mnist.train.next_batch(100)
            self.sess.run(self.train_step, feed_dict={self.x: batch_xs, self.y_:
batch_ys})

    def get_accuracy(self):
        return self.sess.run(self.accuracy, feed_dict={self.x: self.mnist.test.images,
                                                       self.y_: self.mnist.test.labels})


# Load the MNIST dataset and tell Ray how to serialize the custom classes.
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

# Create the actor.
nn = NeuralNetOnGPU.remote(mnist)

# Run a few steps of training and print the accuracy.
nn.train.remote(100)
accuracy = ray.get(nn.get_accuracy.remote())
print("Accuracy is {}.".format(accuracy))
```

Passing Around Actor Handles (Experimental)

Actor handles can be passed into other tasks. To see an example of this, take a look at the asynchronous parameter server example. To illustrate this with a simple example, consider a simple actor definition. This functionality is currently **experimental** and subject to the limitations described below.

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.counter = 0

    def inc(self):
        self.counter += 1

    def get_counter(self):
        return self.counter
```

We can define remote functions (or actor methods) that use actor handles.

```
@ray.remote
def f(counter):
    while True:
        counter.inc.remote()
```

If we instantiate an actor, we can pass the handle around to various tasks.

```
counter = Counter.remote()

# Start some tasks that use the actor.
[f.remote(counter) for _ in range(4)]

# Print the counter value.
for _ in range(10):
    print(ray.get(counter.get_counter.remote()))
```

Current Actor Limitations

We are working to address the following issues.

1. **Actor lifetime management:** Currently, when the original actor handle for an actor goes out of scope, a task is scheduled on that actor that kills the actor process (this new task will run once all previous tasks have finished running). This could be an issue if the original actor

handle goes out of scope, but the actor is still being used by tasks that have been passed the actor handle.

2. **Returning actor handles:** Actor handles currently cannot be returned from a remote function or actor method. Similarly, `ray.put` cannot be called on an actor handle.
3. **Reconstruction of evicted actor objects:** If `ray.get` is called on an evicted object that was created by an actor method, Ray currently will not reconstruct the object. For more information, see the documentation on fault tolerance.
4. **Deterministic reconstruction of lost actors:** If an actor is lost due to node failure, the actor is reconstructed on a new node, following the order of initial execution. However, new tasks that are scheduled onto the actor in the meantime may execute in between re-executed tasks. This could be an issue if your application has strict requirements for state consistency.