Installation

1  Example Use 3

Python Module Index 147
Ray Documentation, Release 0.5.0

Ray is a flexible, high-performance distributed execution framework.

Ray is easy to install: `pip install ray`
Example Use

<table>
<thead>
<tr>
<th>Basic Python</th>
<th>Distributed with Ray</th>
</tr>
</thead>
<tbody>
<tr>
<td># Execute f serially.</td>
<td># Execute f in parallel.</td>
</tr>
</tbody>
</table>
| ```python
  def f():
      time.sleep(1)
      return 1
``` | ```python
  @ray.remote
  def f():
      time.sleep(1)
      return 1
``` |
| results = [f() for i in range(4)]                | ray.init()
| results = ray.get([f.remote() for i in range(4)]) |                                           |

View the codebase on GitHub.

Ray comes with libraries that accelerate deep learning and reinforcement learning development:

- Ray Tune: Hyperparameter Optimization Framework
- Ray RLlib: Scalable Reinforcement Learning

### 1.1 Installing Ray

Ray should work with Python 2 and Python 3. We have tested Ray on Ubuntu 14.04, Ubuntu 16.04, OS X 10.11 and 10.12.

You can install Ray as follows.

```
pip install ray
```
1.1.1 Trying the latest version of Ray

Here are links to the latest wheels (which are built off of master). These versions will have newer features but may be subject to more bugs. To install these wheels, run the following command:

```
pip install -U [link to wheel]
```

<table>
<thead>
<tr>
<th>Linux</th>
<th>MacOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Python 3.6</td>
<td>MacOS Python 3.6</td>
</tr>
<tr>
<td>Linux Python 3.5</td>
<td>MacOS Python 3.5</td>
</tr>
<tr>
<td>Linux Python 3.4</td>
<td>MacOS Python 3.4</td>
</tr>
<tr>
<td>Linux Python 2.7</td>
<td>MacOS Python 2.7</td>
</tr>
</tbody>
</table>

1.1.2 Building Ray from source

If you want to use the latest version of Ray, you can build it from source. Below, we have instructions for installing dependencies and building from source for both Linux and MacOS.

**Dependencies**

To build Ray, first install the following dependencies. We recommend using Anaconda.

For Ubuntu, run the following commands:

```
sudo apt-get update
sudo apt-get install -y cmake pkg-config build-essential autoconf curl libtool unzip
  --flex bison psmisc python # we install python here because python2 is required to
  --build the webui

# If you are not using Anaconda, you need the following.
sudo apt-get install python-dev # For Python 2.
sudo apt-get install python3-dev # For Python 3.

# If you are on Ubuntu 14.04, you need the following.
pip install cmake
```

```
pip install cython
```

For MacOS, run the following commands:

```
brew update
brew install cmake pkg-config automake autoconf libtool openssl bison wget
```

```
pip install cython
```

If you are using Anaconda, you may also need to run the following.

```
conda install libgcc
```

**Install Ray**

Ray can be built from the repository as follows.
```
git clone https://github.com/ray-project/ray.git
cd ray/python
pip install -e . --verbose  # Add --user if you see a permission denied error.
```

Alternatively, Ray can be built from the repository without cloning using pip.

```
# Alternatively, Ray can be built from the repository without cloning using pip.
pip install git+https://github.com/ray-project/ray.git#subdirectory=python
```

**Test if the installation succeeded**

To test if the installation was successful, try running some tests. This assumes that you’ve cloned the git repository.

```
python test/runtest.py
```

**Cleaning the source tree**

The source tree can be cleaned by running

```
git clean -f -f -x -d
```

in the `ray/` directory.

**1.2 Installation on Docker**

You can install Ray on any platform that runs Docker. We do not presently publish Docker images for Ray, but you can build them yourself using the Ray distribution.

Using Docker can streamline the build process and provide a reliable way to get up and running quickly.

**1.2.1 Install Docker**

**Mac, Linux, Windows platforms**

The Docker Platform release is available for Mac, Windows, and Linux platforms. Please download the appropriate version from the Docker website and follow the corresponding installation instructions. Linux user may find these alternate instructions helpful.

**Docker installation on EC2 with Ubuntu**

The instructions below show in detail how to prepare an Amazon EC2 instance running Ubuntu 16.04 for use with Docker.

Apply initialize the package repository and apply system updates:

```
sudo apt-get update
sudo apt-get -y dist-upgrade
```

Install Docker and start the service:
**1.2.2 Clone the Ray repository**

```bash
git clone https://github.com/ray-project/ray.git
```

**1.2.3 Build Docker images**

Run the script to create Docker images.

```bash
cd ray
./build-docker.sh
```

This script creates several Docker images:

- The `ray-project/deploy` image is a self-contained copy of code and binaries suitable for end users.
- The `ray-project/examples` adds additional libraries for running examples.
- The `ray-project/base-deps` image builds from Ubuntu Xenial and includes Anaconda and other basic dependencies and can serve as a starting point for developers.

Review images by listing them:

```bash
docker images
```

Output should look something like the following:

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>ray-project/examples</td>
<td>latest</td>
<td>7584bde65894</td>
<td>4 days</td>
</tr>
<tr>
<td>ray-project/deploy</td>
<td>latest</td>
<td>970966166c71</td>
<td>4 days</td>
</tr>
<tr>
<td>ray-project/base-deps</td>
<td>latest</td>
<td>f45d66963151</td>
<td>4 days</td>
</tr>
<tr>
<td>ubuntu</td>
<td>xenial</td>
<td>f49ec89601e</td>
<td>3 weeks</td>
</tr>
<tr>
<td></td>
<td>ago</td>
<td>129.5 MB</td>
<td></td>
</tr>
</tbody>
</table>
1.2.4 Launch Ray in Docker

Start out by launching the deployment container.

```
docker run --shm-size=<shm-size> -t -i ray-project/deploy
```

Replace `<shm-size>` with a limit appropriate for your system, for example 512M or 2G. The `-t` and `-i` options here are required to support interactive use of the container.

**Note:** Ray requires a large amount of shared memory because each object store keeps all of its objects in shared memory, so the amount of shared memory will limit the size of the object store.

You should now see a prompt that looks something like:

```
root@ebc78f68d100:/ray#
```

1.2.5 Test if the installation succeeded

To test if the installation was successful, try running some tests. Within the container shell enter the following commands:

```
python test/runtest.py  # This tests basic functionality.
```

You are now ready to continue with the tutorial.

1.2.6 Running examples in Docker

Ray includes a Docker image that includes dependencies necessary for running some of the examples. This can be an easy way to see Ray in action on a variety of workloads.

Launch the examples container.

```
docker run --shm-size=1024m -t -i ray-project/examples
```

**Hyperparameter optimization**

```
cd /ray/examples/hyperopt/
python /ray/examples/hyperopt/hyperopt_simple.py
```

**Batch L-BFGS**

```
python /ray/examples/lbfgs/driver.py
```

**Learning to play Pong**

```
python /ray/examples/rl_pong/driver.py
```
1.3 Installation Troubleshooting

1.3.1 Trouble installing Arrow

Some candidate possibilities.

You have a different version of Flatbuffers installed

Arrow pulls and builds its own copy of Flatbuffers, but if you already have Flatbuffers installed, Arrow may find the wrong version. If a directory like /usr/local/include/flatbuffers shows up in the output, this may be the problem. To solve it, get rid of the old version of flatbuffers.

There is some problem with Boost

If a message like Unable to find the requested Boost libraries appears when installing Arrow, there may be a problem with Boost. This can happen if you installed Boost using MacPorts. This is sometimes solved by using Brew instead.

1.3.2 Trouble installing or running Ray

One of the Ray libraries is compiled against the wrong version of Python

If there is a segfault or a sigabort immediately upon importing Ray, one of the components may have been compiled against the wrong Python libraries. CMake should normally find the right version of Python, but this process is not completely reliable. In this case, check the CMake output from installation and make sure that the version of the Python libraries that were found match the version of Python that you’re using.

Note that it’s common to have multiple versions of Python on your machine (for example both Python 2 and Python 3). Ray will be compiled against whichever version of Python is found when you run the python command from the command line, so make sure this is the version you wish to use.

1.4 Tutorial

To use Ray, you need to understand the following:

• How Ray executes tasks asynchronously to achieve parallelism.
• How Ray uses object IDs to represent immutable remote objects.

1.4.1 Overview

Ray is a Python-based distributed execution engine. The same code can be run on a single machine to achieve efficient multiprocessing, and it can be used on a cluster for large computations.

When using Ray, several processes are involved.

• Multiple worker processes execute tasks and store results in object stores. Each worker is a separate process.
• One object store per node stores immutable objects in shared memory and allows workers to efficiently share objects on the same node with minimal copying and deserialization.
• One local scheduler per node assigns tasks to workers on the same node.
• A **global scheduler** receives tasks from local schedulers and assigns them to other local schedulers.

• A **driver** is the Python process that the user controls. For example, if the user is running a script or using a Python shell, then the driver is the Python process that runs the script or the shell. A driver is similar to a worker in that it can submit tasks to its local scheduler and get objects from the object store, but it is different in that the local scheduler will not assign tasks to the driver to be executed.

• A **Redis server** maintains much of the system’s state. For example, it keeps track of which objects live on which machines and of the task specifications (but not data). It can also be queried directly for debugging purposes.

1.4.2 Starting Ray

To start Ray, start Python and run the following commands.

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

This starts Ray.

1.4.3 Immutable remote objects

In Ray, we can create and compute on objects. We refer to these objects as **remote objects**, and we use **object IDs** to refer to them. Remote objects are stored in **object stores**, and there is one object store per node in the cluster. In the cluster setting, we may not actually know which machine each object lives on.

An **object ID** is essentially a unique ID that can be used to refer to a remote object. If you’re familiar with Futures, our object IDs are conceptually similar.

We assume that remote objects are immutable. That is, their values cannot be changed after creation. This allows remote objects to be replicated in multiple object stores without needing to synchronize the copies.

**Put and Get**

The commands **ray.get** and **ray.put** can be used to convert between Python objects and object IDs, as shown in the example below.

```python
x = "example"
ray.put(x)  # ObjectID(b49a32d72057bdcfc4dda35584b3d838aad89f5d)
```

The command `ray.put(x)` would be run by a worker process or by the driver process (the driver process is the one running your script). It takes a Python object and copies it to the local object store (here **local** means **on the same node**). Once the object has been stored in the object store, its value cannot be changed.

In addition, `ray.put(x)` returns an object ID, which is essentially an ID that can be used to refer to the newly created remote object. If we save the object ID in a variable with `x_id = ray.put(x)`, then we can pass `x_id` into remote functions, and those remote functions will operate on the corresponding remote object.

The command `ray.get(x_id)` takes an object ID and creates a Python object from the corresponding remote object. For some objects like arrays, we can use shared memory and avoid copying the object. For other objects, this copies the object from the object store to the worker process’s heap. If the remote object corresponding to the object ID `x_id` does not live on the same node as the worker that calls `ray.get(x_id)`, then the remote object will first be transferred from an object store that has it to the object store that needs it.

```python
x_id = ray.put("example")
ray.get(x_id)  # "example"
```
If the remote object corresponding to the object ID `x_id` has not been created yet, the command `ray.get(x_id)` will wait until the remote object has been created.

A very common use case of `ray.get` is to get a list of object IDs. In this case, you can call `ray.get(object_ids)` where `object_ids` is a list of object IDs.

```python
result_ids = [ray.put(i) for i in range(10)]
ray.get(result_ids)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### 1.4.4 Asynchronous Computation in Ray

Ray enables arbitrary Python functions to be executed asynchronously. This is done by designating a Python function as a remote function.

For example, a normal Python function looks like this.

```python
def add1(a, b):
    return a + b
```

A remote function looks like this.

```python
@ray.remote
def add2(a, b):
    return a + b
```

Remote functions

Whereas calling `add1(1, 2)` returns 3 and causes the Python interpreter to block until the computation has finished, calling `add2.remote(1, 2)` immediately returns an object ID and creates a task. The task will be scheduled by the system and executed asynchronously (potentially on a different machine). When the task finishes executing, its return value will be stored in the object store.

```python
x_id = add2.remote(1, 2)
ray.get(x_id)  # 3
```

The following simple example demonstrates how asynchronous tasks can be used to parallelize computation.

```python
import time

def f1():
    time.sleep(1)

@ray.remote
def f2():
    time.sleep(1)

# The following takes ten seconds.
[f1() for _ in range(10)]

# The following takes one second (assuming the system has at least ten CPUs).
ray.get([(f2.remote() for _ in range(10))])
```

There is a sharp distinction between submitting a task and executing the task. When a remote function is called, the task of executing that function is submitted to a local scheduler, and object IDs for the outputs of the task are immediately returned. However, the task will not be executed until the system actually schedules the task on a worker.
Task execution is not done lazily. The system moves the input data to the task, and the task will execute as soon as its input dependencies are available and there are enough resources for the computation.

**When a task is submitted, each argument may be passed in by value or by object ID.** For example, these lines have the same behavior.

```python
add2.remote(1, 2)
add2.remote(1, ray.put(2))
add2.remote(ray.put(1), ray.put(2))
```

Remote functions never return actual values, they always return object IDs.

When the remote function is actually executed, it operates on Python objects. That is, if the remote function was called with any object IDs, the system will retrieve the corresponding objects from the object store.

Note that a remote function can return multiple object IDs.

```python
@ray.remote(num_return_vals=3)
def return_multiple():
    return 1, 2, 3
a_id, b_id, c_id = return_multiple.remote()
```

**Expressing dependencies between tasks**

Programmers can express dependencies between tasks by passing the object ID output of one task as an argument to another task. For example, we can launch three tasks as follows, each of which depends on the previous task.

```python
@ray.remote
def f(x):
    return x + 1
x = f.remote(0)
y = f.remote(x)
z = f.remote(y)
ray.get(z)  # 3
```

The second task above will not execute until the first has finished, and the third will not execute until the second has finished. In this example, there are no opportunities for parallelism.

The ability to compose tasks makes it easy to express interesting dependencies. Consider the following implementation of a tree reduce.

```python
import numpy as np
@ray.remote
def generate_data():
    return np.random.normal(size=1000)
@ray.remote
def aggregate_data(x, y):
    return x + y
# Generate some random data. This launches 100 tasks that will be scheduled on
# various nodes. The resulting data will be distributed around the cluster.
data = [generate_data.remote() for _ in range(100)]
```

(continues on next page)
Remote Functions Within Remote Functions

So far, we have been calling remote functions only from the driver. But worker processes can also call remote functions. To illustrate this, consider the following example.

```python
@ray.remote
def sub_experiment(i, j):
    # Run the jth sub-experiment for the ith experiment.
    return i + j

@ray.remote
def run_experiment(i):
    sub_results = []
    # Launch tasks to perform 10 sub-experiments in parallel.
    for j in range(10):
        sub_results.append(sub_experiment.remote(i, j))
    # Return the sum of the results of the sub-experiments.
    return sum(ray.get(sub_results))

results = [run_experiment.remote(i) for i in range(5)]
ray.get(results)  # [45, 55, 65, 75, 85]
```

When the remote function `run_experiment` is executed on a worker, it calls the remote function `sub_experiment` a number of times. This is an example of how multiple experiments, each of which takes advantage of parallelism internally, can all be run in parallel.

1.5 The Ray API

```python
ray.init()
```

Connect to an existing Ray cluster or start one and connect to it.

This method handles two cases. Either a Ray cluster already exists and we just attach this driver to it, or we start all of the processes associated with a Ray cluster and attach to the newly started cluster.

To start Ray and all of the relevant processes, use this as follows:

```python
ray.init()
```

To connect to an existing Ray cluster, use this as follows (substituting in the appropriate address):

```python
ray.init()
```
ray.init(redis_address="123.45.67.89:6379")

Parameters

- **redis_address** *(str)* – The address of the Redis server to connect to. If this address is not provided, then this command will start Redis, a global scheduler, a local scheduler, a plasma store, a plasma manager, and some workers. It will also kill these processes when Python exits.

- **num_cpus** *(int)* – Number of cpus the user wishes all local schedulers to be configured with.

- **num_gpus** *(int)* – Number of gpus the user wishes all local schedulers to be configured with.

- **resources** – A dictionary mapping the name of a resource to the quantity of that resource available.

- **object_store_memory** – The amount of memory (in bytes) to start the object store with.

- **node_ip_address** *(str)* – The IP address of the node that we are on.

- **object_id_seed** *(int)* – Used to seed the deterministic generation of object IDs. The same value can be used across multiple runs of the same job in order to generate the object IDs in a consistent manner. However, the same ID should not be used for different jobs.

- **num_workers** *(int)* – The number of workers to start. This is only provided if redis_address is not provided.

- **driver_mode** *(bool)* – The mode in which to start the driver. This should be one of ray.SCRIPT_MODE, ray.LOCAL_MODE, and ray.SILENT_MODE.

- **redirect_worker_output** – True if the stdout and stderr of worker processes should be redirected to files.

- **redirect_output** *(bool)* – True if stdout and stderr for non-worker processes should be redirected to files and false otherwise.

- **ignore_reinit_error** – True if we should suppress errors from calling ray.init() a second time.

- **num_redis_shards** – The number of Redis shards to start in addition to the primary Redis shard.

- **redis_max_clients** – If provided, attempt to configure Redis with this maxclients number.

- **plasma_directory** – A directory where the Plasma memory mapped files will be created.

- **huge_pages** – Boolean flag indicating whether to start the Object Store with hugetlbfs support. Requires plasma_directory.

- **include_webui** – Boolean flag indicating whether to start the web UI, which is a Jupyter notebook.

- **use_raylet** – True if the new raylet code path should be used.

Returns Address information about the started processes.

Raises Exception – An exception is raised if an inappropriate combination of arguments is passed in.
ray.remote(*args, **kwargs)

Define a remote function or an actor class.

This can be used with no arguments to define a remote function or actor as follows:

```python
@ray.remote
def f():
    return 1
@ray.remote
class Foo(object):
    def method(self):
        return 1
```

It can also be used with specific keyword arguments:

- **num_return_vals**: This is only for remote functions. It specifies the number of object IDs returned by the remote function invocation.
- **num_cpus**: The quantity of CPU cores to reserve for this task or for the lifetime of the actor.
- **num_gpus**: The quantity of GPUs to reserve for this task or for the lifetime of the actor.
- **resources**: The quantity of various custom resources to reserve for this task or for the lifetime of the actor. This is a dictionary mapping strings (resource names) to numbers.
- **max_calls**: Only for remote functions. This specifies the maximum number of times that a given worker can execute the given remote function before it must exit (this can be used to address memory leaks in third-party libraries or to reclaim resources that cannot easily be released, e.g., GPU memory that was acquired by TensorFlow). By default this is infinite.

This can be done as follows:

```python
@ray.remote(num_gpus=1, max_calls=1, num_return_vals=2)
def f():
    return 1, 2
@ray.remote(num_cpus=2, resources={"CustomResource": 1})
class Foo(object):
    def method(self):
        return 1
```

ray.get(object_ids, worker=<ray.worker.Worker object>)

Get a remote object or a list of remote objects from the object store.

This method blocks until the object corresponding to the object ID is available in the local object store. If this object is not in the local object store, it will be shipped from an object store that has it (once the object has been created). If object_ids is a list, then the objects corresponding to each object in the list will be returned.

**Parameters**

- **object_ids** – Object ID of the object to get or a list of object IDs to get.

**Returns**

A Python object or a list of Python objects.

**Raises**

Exception – An exception is raised if the task that created the object or that created one of the objects raised an exception.

ray.wait(object_ids, num_returns=1, timeout=None, worker=<ray.worker.Worker object>)

Return a list of IDs that are ready and a list of IDs that are not.

If timeout is set, the function returns either when the requested number of IDs are ready or when the timeout is reached, whichever occurs first. If it is not set, the function simply waits until that number of objects is ready and returns that exact number of object IDs.
This method returns two lists. The first list consists of object IDs that correspond to objects that are available in the object store. The second list corresponds to the rest of the object IDs (which may or may not be ready).

Ordering of the input list of object IDs is preserved. That is, if A precedes B in the input list, and both are in the ready list, then A will precede B in the ready list. This also holds true if A and B are both in the remaining list.

**Parameters**
- **object_ids (List[ObjectID])** – List of object IDs for objects that may or may not be ready. Note that these IDs must be unique.
- **num_returns (int)** – The number of object IDs that should be returned.
- **timeout (int)** – The maximum amount of time in milliseconds to wait before returning.

**Returns** A list of object IDs that are ready and a list of the remaining object IDs.

*ray.*_put_(*value*, worker=<ray.worker.Worker object>)*

Store an object in the object store.

**Parameters** *value* – The Python object to be stored.

**Returns** The object ID assigned to this value.

*ray.*_get_gpu_ids_()*

Get the IDs of the GPUs that are available to the worker.

If the CUDA_VISIBLE_DEVICES environment variable was set when the worker started up, then the IDs returned by this method will be a subset of the IDs in CUDA_VISIBLE_DEVICES. If not, the IDs will fall in the range [0, NUM_GPUS - 1], where NUM_GPUS is the number of GPUs that the node has.

**Returns** A list of GPU IDs.

*ray.*_get_resource_ids_()*

Get the IDs of the resources that are available to the worker.

This function is only supported in the raylet code path.

**Returns** A dictionary mapping the name of a resource to a list of pairs, where each pair consists of the ID of a resource and the fraction of that resource reserved for this worker.

*ray.*_get_webui_url_()*

Get the URL to access the web UI.

Note that the URL does not specify which node the web UI is on.

**Returns** The URL of the web UI as a string.

*ray.*_shutdown_(*worker=<ray.worker.Worker object>)*

Disconnect the worker, and terminate processes started by ray.init().

This will automatically run at the end when a Python process that uses Ray exits. It is ok to run this twice in a row. The primary use case for this function is to cleanup state between tests.

Note that this will clear any remote function definitions, actor definitions, and existing actors, so if you wish to use any previously defined remote functions or actors after calling ray.shutdown(), then you need to redefine them. If they were defined in an imported module, then you will need to reload the module.

*ray.*_register_custom_serializer_(cls, use_pickle=False, use_dict=False, serializer=None, deserializer=None, local=False, driver_id=None, class_id=None, worker=<ray.worker.Worker object>)*

Enable serialization and deserialization for a particular class.

This method runs the register_class function defined below on every worker, which will enable ray to properly serialize and deserialize objects of this class.
Parameters

- **cls** *(type)* – The class that ray should use this custom serializer for.
- **use_pickle** *(bool)* – If true, then objects of this class will be serialized using pickle.
- **use_dict** – If true, then objects of this class be serialized turning their __dict__ fields into a dictionary. Must be False if use_pickle is true.
- **serializer** – The custom serializer to use. This should be provided if and only if use_pickle and use_dict are False.
- **deserializer** – The custom deserializer to use. This should be provided if and only if use_pickle and use_dict are False.
- **local** – True if the serializers should only be registered on the current worker. This should usually be False.
- **driver_id** – ID of the driver that we want to register the class for.
- **class_id** – ID of the class that we are registering. If this is not specified, we will calculate a new one inside the function.

`ray.profile(event_type, extra_data=None, worker=None)`

Profile a span of time so that it appears in the timeline visualization.

Note that this only works in the raylet code path.

This function can be used as follows (both on the driver or within a task).

```python
with ray.profile("custom event", extra_data={'key': 'value'}):
    # Do some computation here.
```

Optionally, a dictionary can be passed as the “extra_data” argument, and it can have keys “name” and “cname” if you want to override the default timeline display text and box color. Other values will appear at the bottom of the chrome tracing GUI when you click on the box corresponding to this profile span.

Parameters

- **event_type** – A string describing the type of the event.
- **extra_data** – This must be a dictionary mapping strings to strings. This data will be added to the json objects that are used to populate the timeline, so if you want to set a particular color, you can simply set the “cname” attribute to an appropriate color. Similarly, if you set the “name” attribute, then that will set the text displayed on the box in the timeline.

Returns An object that can profile a span of time via a “with” statement.

`ray.method(*args, **kwargs)`

Annotate an actor method.

```python
@ray.remote
class Foo(object):
    @ray.method(num_return_vals=2)
    def bar(self):
        return 1, 2

f = Foo.remote()
_, _ = f.bar.remote()
```
Parameters **num_return_vals** – The number of object IDs that should be returned by invocations of this actor method.

### 1.5.1 The Ray Command Line API

**ray start**

```bash
ray start [OPTIONS]
```

**Options**

|--node-ip-address <node_ip_address>  
the IP address of this node

|--redis-address <redis_address>  
the address to use for connecting to Redis

|--redis-port <redis_port>  
the port to use for starting Redis

|--num-redis-shards <num_redis_shards>  
the number of additional Redis shards to use in addition to the primary Redis shard

|--redis-max-clients <redis_max_clients>  
If provided, attempt to configure Redis with this maximum number of clients.

|--redis-shard-ports <redis_shard_ports>  
the port to use for the Redis shards other than the primary Redis shard

|--object-manager-port <object_manager_port>  
the port to use for starting the object manager

|--object-store-memory <object_store_memory>  
the maximum amount of memory (in bytes) to allow the object store to use

|--num-workers <num_workers>  
The initial number of workers to start on this node, note that the local scheduler may start additional workers. If you wish to control the total number of concurrent tasks, then use --resources instead and specify the CPU field.

|--num-cpus <num_cpus>  
the number of CPUs on this node

|--num-gpus <num_gpus>  
the number of GPUs on this node

|--resources <resources>  
a JSON serialized dictionary mapping resource name to resource quantity

|--head  
provide this argument for the head node

|--no-ui  
provide this argument if the UI should not be started

|--block  
provide this argument to block forever in this command

|--plasma-directory <plasma_directory>  
object store directory for memory mapped files
--huge-pages
    enable support for huge pages in the object store

--autoscaling-config <autoscaling_config>
    the file that contains the autoscaling config

--use-raylet
    use the raylet code path

ray stop

    ray stop [OPTIONS]

ray create_or_update

    ray create_or_update [OPTIONS] CLUSTER_CONFIG_FILE

Options

--no-restart
    Whether to skip restarting Ray services during the update. This avoids interrupting running jobs.

--min-workers <min_workers>
    Override the configured min worker node count for the cluster.

--max-workers <max_workers>
    Override the configured max worker node count for the cluster.

-y, --yes
    Don’t ask for confirmation.

Arguments

CLUSTER_CONFIG_FILE
    Required argument

ray teardown

    ray teardown [OPTIONS] CLUSTER_CONFIG_FILE

Options

-y, --yes
    Don’t ask for confirmation.

Arguments

CLUSTER_CONFIG_FILE
    Required argument
Ray Documentation, Release 0.5.0

ray get_head_ip

Arguments

CLUSTER_CONFIG_FILE
  Required argument

1.6 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.

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

1.6.1 Defining and creating an actor

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

```python
@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()`.

```python
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.
1.6.2 Using an actor

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

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

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]
```

1.6.3 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.

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

### 1.6.4 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.

```python
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.

```python
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.

```python
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})
```

(continues on next page)
# 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))

1.6.5 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.

```python
@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.

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

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

```python
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()))
```

1.6.6 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.

### 1.7 Using Ray with GPUs

GPUs are critical for many machine learning applications. Ray enables remote functions and actors to specify their GPU requirements in the `ray.remote` decorator.

#### 1.7.1 Starting Ray with GPUs

In order for remote functions and actors to use GPUs, Ray must know how many GPUs are available. If you are starting Ray on a single machine, you can specify the number of GPUs as follows.

```python
ray.init(num_gpus=4)
```

If you don’t pass in the `num_gpus` argument, Ray will assume that there are 0 GPUs on the machine.

If you are starting Ray with the `ray start` command, you can indicate the number of GPUs on the machine with the `--num-gpus` argument.

```bash
ray start --head --num-gpus=4
```

**Note:** There is nothing preventing you from passing in a larger value of `num_gpus` than the true number of GPUs on the machine. In this case, Ray will act as if the machine has the number of GPUs you specified for the purposes of scheduling tasks that require GPUs. Trouble will only occur if those tasks attempt to actually use GPUs that don’t exist.

#### 1.7.2 Using Remote Functions with GPUs

If a remote function requires GPUs, indicate the number of required GPUs in the remote decorator.

```python
@ray.remote(num_gpus=1)
def gpu_method():
    return "This function is allowed to use GPUs {}."
```

Inside of the remote function, a call to `ray.get_gpu_ids()` will return a list of integers indicating which GPUs the remote function is allowed to use.

**Note:** The function `gpu_method` defined above doesn’t actually use any GPUs. Ray will schedule it on a machine which has at least one GPU, and will reserve one GPU for it while it is being executed, however it is up to the function to actually make use of the GPU. This is typically done through an external library like TensorFlow. Here is an example that actually uses GPUs. Note that for this example to work, you will need to install the GPU version of TensorFlow.
import os
import tensorflow as tf

@ray.remote(num_gpus=1)
def gpu_method():
os.environ['CUDA_VISIBLE_DEVICES'] = ','.join(map(str, ray.get_gpu_ids()))
    # Create a TensorFlow session. TensorFlow will restrict itself to use the
    # GPUs specified by the CUDA_VISIBLE_DEVICES environment variable.
    tf.Session()

Note: It is certainly possible for the person implementing `gpu_method` to ignore `ray.get_gpu_ids` and to use all of the GPUs on the machine. Ray does not prevent this from happening, and this can lead to too many workers using the same GPU at the same time. For example, if the `CUDA_VISIBLE_DEVICES` environment variable is not set, then TensorFlow will attempt to use all of the GPUs on the machine.

1.7.3 Using Actors with GPUs

When defining an actor that uses GPUs, indicate the number of GPUs an actor instance requires in the `ray.remote` decorator.

```python
@ray.remote(num_gpus=1)
class GPUActor(object):
    def __init__(self):
        return "This actor is allowed to use GPUs {}.".format(ray.get_gpu_ids())
```

When the actor is created, GPUs will be reserved for that actor for the lifetime of the actor.

Note that Ray must have been started with at least as many GPUs as the number of GPUs you pass into the `ray.remote` decorator. Otherwise, if you pass in a number greater than what was passed into `ray.init`, an exception will be thrown when instantiating the actor.

The following is an example of how to use GPUs in an actor through TensorFlow.

```python
@ray.remote(num_gpus=1)
class GPUActor(object):
    def __init__(self):
        self.gpu_ids = ray.get_gpu_ids()
        os.environ['CUDA_VISIBLE_DEVICES'] = ','.join(map(str, self.gpu_ids))
        # The call to tf.Session() will restrict TensorFlow to use the GPUs
        # specified in the CUDA_VISIBLE_DEVICES environment variable.
        self.sess = tf.Session()
```

1.7.4 Troubleshooting

Note: Currently, when a worker executes a task that uses a GPU, the task may allocate memory on the GPU and may not release it when the task finishes executing. This can lead to problems. See this issue.

1.8 Web UI

The Ray web UI includes tools for debugging Ray jobs. The following image shows an example of using the task timeline for performance debugging:
1.8.1 Dependencies

To use the UI, you will need to install the following.

```
pip install jupyter ipywidgets bokeh
```

If you see an error like

```
Widget Javascript not detected. It may not be installed properly.
```

Then you may need to run the following.

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

**Note:** If you are building Ray from source, then you will also need a `python2` executable.

1.8.2 Running the Web UI

Currently, the web UI is launched automatically when `ray.init` is called. The command will print a URL of the form:

```
View the web UI at http://localhost:8889/notebooks/ray_ui92131.ipynb?
˓→token=93954a314e5a81bf56e023ad18bda3a3d272ee216f342938
```

If you are running Ray on your local machine, then you can head directly to that URL with your browser to see the Jupyter notebook. Otherwise, if you are using Ray remotely, such as on EC2, you will need to ensure that port is open on that machine. Typically, when you `ssh` into the machine, you can also port forward with the `-L` option as such:

```
ssh -L <local_port>:localhost:<remote_port> <user>@<ip-address>
```

So for the above URL, you would use the port 8889. The Jupyter notebook attempts to run on port 8888, but if that fails it tries successive ports until it finds an open port.

You can also open the port on the machine as well, which is not recommended for security as the machine would be open to the Internet. In this case, you would need to replace localhost by the public IP the remote machine is using.

Once you have navigated to the URL, start the UI by clicking on the following.

```
Kernel -> Restart and Run all
```
1.8.3 Features

The UI supports a search for additional details on Task IDs and Object IDs, a task timeline, a distribution of task completion times, and time series for CPU utilization and cluster usage.

Task and Object IDs

These widgets show additional details about an object or task given the ID. If you have the object in Python, the ID can be found by simply calling `.hex` on an Object ID as below:

```python
# This will return a hex string of the ID.
objectid = ray.put(1)
literal_id = objectid.hex()
```

and pasting in the returned string with no quotes. Otherwise, they can be found in the task timeline in the output area below the timeline when you select a task.

For Task IDs, they can be found by searching for an object ID the task created, or via the task timeline in the output area.

The additional details for tasks here can also be found in the task timeline; the search just provides an easier method to find a specific task when you have millions.

Task Timeline

There are three components to this widget: the controls for the widget at the top, the timeline itself, and the details area at the bottom. In the controls, you first select whether you want to select a subset of tasks via the time they were completed or by the number of tasks. You can control the percentages either via a double sided slider, or by setting specific values in the text boxes. If you choose to select by the number of tasks, then entering a negative number N in the text field denotes the last N tasks run, while a positive value N denotes the first N tasks run. If there are ten tasks and you enter -1 into the field, then the slider will show 90% to 100%, where 1 would show 0% to 10%. Finally, you can choose if you want edges for task submission (if a task invokes another task) or object dependencies (if the result from a task is passed to another task) to be added, and if you want the different phases of a task broken up into separate tasks in the timeline.

For the timeline, each node has its own dropdown with a timeline, and each row in the dropdown is a worker. Moving and zooming are handled by selecting the appropriate icons on the floating taskbar. The first is selection, the second panning, the third zooming, and the fourth timing. To shown edges, you can enable Flow Events in View Options.

If you have selection enabled in the floating taskbar and select a task, then the details area at the bottom will fill up with information such as task ID, function ID, and the duration in seconds of each phase of the task.

Time Distributions and Time Series

The completion time distribution, CPU utilization, and cluster usage all have the same task selection controls as the task timeline.

The task completion time distribution tracks the histogram of completion tasks for all tasks selected.

CPU utilization gives you a count of how many CPU cores are being used at a given time. As typically each core has a worker assigned to it, this is equivalent to utilization of the workers running in Ray.

Cluster Usage gives you a heat-map with time on the x-axis, node IP addresses on the y-axis, and coloring based on how many tasks were running on that node at that given time.
1.8.4 Troubleshooting

The Ray timeline visualization may not work in Firefox or Safari.

1.9 Ray Tune: Hyperparameter Optimization Framework

Ray Tune is a scalable hyperparameter optimization framework for reinforcement learning and deep learning. Go from running one experiment on a single machine to running on a large cluster with efficient search algorithms without changing your code.

1.9.1 Getting Started

Installation

You'll need to first install ray to import Ray Tune.

Quick Start

```python
import ray
import ray.tune as tune

ray.init()
tune.register_trainable("train_func", train_func)

all_trials = tune.run_experiments(
    {
        "my_experiment": {
            "run": "train_func",
            "stop": {
                "mean_accuracy": 99,
                "config": {
                    "lr": tune.grid_search([0.2, 0.4, 0.6]),
                    "momentum": tune.grid_search([0.1, 0.2]),
                }
            }
        }
    }
)
```

For the function you wish to tune, add a two-line modification (note that we use PyTorch as an example but Ray Tune works with any deep learning framework):

```python
def train_func(config, reporter):  # add a reporter arg
    model = NeuralNet()
    optimizer = torch.optim.SGD(
        model.parameters(), lr=config["lr"], momentum=config["momentum"])
    dataset = ( ... )

    for idx, (data, target) in enumerate(dataset):
        # ...
        output = model(data)
        loss = F.MSELoss(output, target)
        loss.backward()
        optimizer.step()
        accuracy = eval_accuracy(...)
        reporter(timesteps_total=idx, mean_accuracy=accuracy)  # report metrics
```
This PyTorch script runs a small grid search over the `train_func` function using Ray Tune, reporting status on the command line until the stopping condition of `mean_accuracy >= 99` is reached (for metrics like `loss` that decrease over time, specify `neg_mean_loss` as a condition instead):

```
== Status ==
Using FIFO scheduling algorithm.
Resources used: 4/8 CPUs, 0/0 GPUs
Result logdir: ~/ray_results/my_experiment
- train_func_0_lr=0.2,momentum=1: RUNNING [pid=6778], 209 s, 20604 ts, 7.29 acc
- train_func_1_lr=0.4,momentum=1: RUNNING [pid=6780], 208 s, 20522 ts, 53.1 acc
- train_func_2_lr=0.6,momentum=1: TERMINATED [pid=6789], 21 s, 2190 ts, 100 acc
- train_func_3_lr=0.2,momentum=2: RUNNING [pid=6791], 208 s, 41004 ts, 8.37 acc
- train_func_4_lr=0.4,momentum=2: RUNNING [pid=6800], 209 s, 41204 ts, 70.1 acc
- train_func_5_lr=0.6,momentum=2: TERMINATED [pid=6809], 10 s, 2164 ts, 100 acc
```

In order to report incremental progress, `train_func` periodically calls the `reporter` function passed in by Ray Tune to return the current timestep and other metrics as defined in `ray.tune.result.TrainingResult`. Incremental results will be synced to local disk on the head node of the cluster.

`tune.run_experiments` returns a list of Trial objects which you can inspect results of via `trial.last_result`.

Learn more about specifying experiments.

1.9.2 Features

Ray Tune has the following features:

- Scalable implementations of search execution techniques such as Population Based Training (PBT), Median Stopping Rule, and HyperBand.
- The ability to combine search execution and search algorithms, such as Model-Based Optimization (HyperOpt) with HyperBand.
- Integration with visualization tools such as TensorBoard, rllab's VisKit, and a parallel coordinates visualization.
- Flexible trial variant generation, including grid search, random search, and conditional parameter distributions.
- Resource-aware scheduling, including support for concurrent runs of algorithms that may themselves be parallel and distributed.

1.9.3 Concepts

Ray Tune schedules a number of trials in a cluster. Each trial runs a user-defined Python function or class and is parameterized by a `config` variation passed to the user code.

In order to run any given function, you need to run `register_trainable` to a name. This makes all Ray workers aware of the function.

```python
ray.tune.register_trainable(name, trainable)
```

Register a trainable function or class.

Parameters

- `name (str)` – Name to register.
- `trainable (obj)` – Function or `tune.Trainable` class. Functions must take (config, status_reporter) as arguments and will be automatically converted into a class during registration.
Ray Tune provides a `run_experiments` function that generates and runs the trials described by the experiment specification. The trials are scheduled and managed by a `trial scheduler` that implements the search algorithm (default is FIFO).

```python
ray.tune.run_experiments(experiments=None, search_alg=None, scheduler=None, with_server=False, server_port=4321, verbose=True, queue_trials=False)
```

Tunes experiments.

**Parameters**

- **experiments** (Experiment / list / dict) – Experiments to run.
- **search_alg** (SearchAlgorithm) – Search Algorithm. Defaults to BasicVariantGenerator.
- **scheduler** (TrialScheduler) – Scheduler for executing the experiment. Choose among FIFO (default), MedianStopping, AsyncHyperBand, and HyperBand.
- **with_server** (bool) – Starts a background Tune server. Needed for using the Client API.
- **server_port** (int) – Port number for launching TuneServer.
- **verbose** (bool) – How much output should be printed for each trial.
- **queue_trials** (bool) – Whether to queue trials when the cluster does not currently have enough resources to launch one. This should be set to True when running on an autoscaling cluster to enable automatic scale-up.

**Returns** List of Trial objects, holding data for each executed trial.

Ray Tune can be used anywhere Ray can, e.g. on your laptop with `ray.init()` embedded in a Python script, or in an auto-scaling cluster for massive parallelism.

You can find the code for Ray Tune here on GitHub.

### 1.9.4 Trial Schedulers

By default, Ray Tune schedules trials in serial order with the `FIFOScheduler` class. However, you can also specify a custom scheduling algorithm that can early stop trials, perturb parameters, or incorporate suggestions from an external service. Currently implemented trial schedulers include Population Based Training (PBT), Median Stopping Rule, and HyperBand.

```python
run_experiments(..., scheduler=AsyncHyperBandScheduler())
```

### 1.9.5 Search Algorithms

Tune allows you to use different search algorithms in combination with different scheduling algorithms. Currently, Tune offers the following search algorithms:

- Grid search / Random Search
- Tree-structured Parzen Estimators (HyperOpt)

If you are interested in implementing or contributing a new Search Algorithm, the API is straightforward:

```python
class ray.tune.suggest.SearchAlgorithm
    Interface of an event handler API for hyperparameter search.
```
Unlike TrialSchedulers, SearchAlgorithms will not have the ability to modify the execution (i.e., stop and pause trials).

Trials added manually (i.e., via the Client API) will also notify this class upon new events, so custom search algorithms should maintain a list of trials ID generated from this class.

See also: `ray.tune.suggest.BasicVariantGenerator`.

### HyperOpt Integration

The `HyperOptSearch` is a SearchAlgorithm that is backed by HyperOpt to perform sequential model-based hyperparameter optimization. In order to use this search algorithm, you will need to install HyperOpt via the following command:

```
$ pip install --upgrade git+git://github.com/hyperopt/hyperopt.git
```

An example of this can be found in `hyperopt_example.py`.

**Note:** The HyperOptScheduler takes an increasing metric in the reward attribute. If trying to minimize a loss, be sure to specify `mean_loss` in the function/class reporting and `reward_attr=neg_mean_loss` in the HyperOptScheduler initializer.

```python
class ray.tune.suggest.HyperOptSearch(experiments, space, max_concurrent=10, reward_attr='episode_reward_mean', **kwargs)
```

A wrapper around HyperOpt to provide trial suggestions.

Requires HyperOpt to be installed from source. Uses the Tree-structured Parzen Estimators algorithm, although can be trivially extended to support any algorithm HyperOpt uses. Externally added trials will not be tracked by HyperOpt.

**Parameters**

- `experiments` *(Experiment | list | dict)* -- Experiments to run. Will be used by SuggestionAlgorithm parent class to initialize Trials.
- `space` *(dict)* -- HyperOpt configuration. Parameters will be sampled from this configuration and will be used to override parameters generated in the variant generation process.
- `max_concurrent` *(int)* -- Number of maximum concurrent trials. Defaults to 10.
- `reward_attr` *(str)* -- The TrainingResult objective value attribute. This refers to an increasing value, which is internally negated when interacting with HyperOpt so that HyperOpt can "maximize" this value.

### 1.9.6 Handling Large Datasets

You often will want to compute a large object (e.g., training data, model weights) on the driver and use that object within each trial. Ray Tune provides a `pin_in_object_store` utility function that can be used to broadcast such large objects. Objects pinned in this way will never be evicted from the Ray object store while the driver process is running, and can be efficiently retrieved from any task via `get_pinned_object`.

```python
import ray
from ray.tune import register_trainable, run_experiments
from ray.tune.util import pin_in_object_store, get_pinned_object
import numpy as np
```

(continues on next page)
ray.init()

# X_id can be referenced in closures
X_id = pin_in_object_store(np.random.random(size=10000000))

def f(config, reporter):
    X = get_pinned_object(X_id)
    # use X

register_trainable("f", f)
run_experiments(...)

1.9.7 Visualizing Results

Ray Tune logs trial results to a unique directory per experiment, e.g. ~/ray_results/my_experiment in the above example. The log records are compatible with a number of visualization tools:

To visualize learning in tensorboard, install TensorFlow:

```bash
$ pip install tensorflow
```

Then, after you run a experiment, you can visualize your experiment with TensorBoard by specifying the output directory of your results. Note that if you running Ray on a remote cluster, you can forward the tensorboard port to your local machine through SSH using ssh -L 6006:localhost:6006 <address>:

```bash
$ tensorboard --logdir=~/ray_results/my_experiment
```

To use rllab’s VisKit (you may have to install some dependencies), run:

```bash
$ git clone https://github.com/rll/rllab.git
$ python rllab/rllab/viskit/frontend.py ~/ray_results/my_experiment
```
Finally, to view the results with a parallel coordinates visualization, open ParallelCoordinatesVisualization.ipynb as follows and run its cells:

```
$ cd $RAY_HOME/python/ray/tune
$ jupyter-notebook ParallelCoordinatesVisualization.ipynb
```

1.9.8 Trial Checkpointing

To enable checkpointing, you must implement a Trainable class (Trainable functions are not checkpointable, since they never return control back to their caller). The easiest way to do this is to subclass the pre-defined `Trainable` class and implement its `_train`, `_save`, and `_restore` abstract methods (example): Implementing this interface is required to support resource multiplexing in schedulers such as HyperBand and PBT.

For TensorFlow model training, this would look something like this (full tensorflow example):

```python
class MyClass(Trainable):
    def _setup(self):
        self.saver = tf.train.Saver()
```

(continues on next page)
```python
def _train(self):
    self.sess.run(...)
    self.iteration += 1

def _save(self, checkpoint_dir):
    return self.saver.save(
        self.sess, checkpoint_dir + "/save",
        global_step=self.iteration)

def _restore(self, path):
    return self.saver.restore(self.sess, path)
```

Additionally, checkpointing can be used to provide fault-tolerance for experiments. This can be enabled by setting `checkpoint_freq: N` and `max_failures: M` to checkpoint trials every `N` iterations and recover from up to `M` crashes per trial, e.g.:

```python
run_experiments({
    "my_experiment": {
        ...
        "checkpoint_freq": 10,
        "max_failures": 5,
    },
})
```

The class interface that must be implemented to enable checkpointing is as follows:

```python
class ray.tune.trainable.Trainable(config=None, logger_creator=None)
Abstract class for trainable models, functions, etc.
A call to `train()` on a trainable will execute one logical iteration of training. As a rule of thumb, the execution time of one train call should be large enough to avoid overheads (i.e. more than a few seconds), but short enough to report progress periodically (i.e. at most a few minutes).
Calling `save()` should save the training state of a trainable to disk, and `restore(path)` should restore a trainable to the given state.
Generally you only need to implement `_train`, `_save`, and `_restore` here when subclassing `Trainable`.
```
```
_restore (checkpoint_path)
Subclasses should override this to implement restore().

_setup()
Subclasses should override this for custom initialization.

_stop()
Subclasses should override this for any cleanup on stop.

1.9.9 Client API

You can modify an ongoing experiment by adding or deleting trials using the Tune Client API. To do this, verify that you have the requests library installed:

```
$ pip install requests
```

To use the Client API, you can start your experiment with with_server=True:

```
run_experiments({...}, with_server=True, server_port=4321)
```

Then, on the client side, you can use the following class. The server address defaults to localhost:4321. If on a cluster, you may want to forward this port (e.g. `ssh -L <local_port>:localhost:<remote_port> <address>`) so that you can use the Client on your local machine.

```python
class ray.tune.web_server.TuneClient (tune_address)
    Client to interact with ongoing Tune experiment.
    Requires server to have started running.
    get_all_trials()
        Returns a list of all trials (trial_id, config, status).
    get_trial (trial_id)
        Returns the last result for queried trial.
    add_trial (name, trial_spec)
        Adds a trial of name with configurations.
    stop_trial (trial_id)
        Requests to stop trial.
```

For an example notebook for using the Client API, see the Client API Example.

1.9.10 Examples

You can find a list of examples using Ray Tune and its various features here, including examples using Keras, TensorFlow, and Population-Based Training.

1.10 Experiment Configuration

1.10.1 Experiment Setup

There are two ways to setup an experiment - one via Python and one via JSON.

The first is to create an Experiment object. You can then pass in either a single experiment or a list of experiments to run_experiments, as follows:
# Single experiment

```python
run_experiments(Experiment(...))
```

# Multiple experiments

```python
run_experiments([Experiment(...), Experiment(...), ...])
```

class ray.tune.Experiment (name, run, stop=None, config=None, trial_resources=None, repeat=1, local_dir=None, upload_dir=None, checkpoint_freq=0, max_failures=3)

Tracks experiment specifications.

**Parameters**

- **name** (*str*) – Name of experiment.
- **run** (*str*) – The algorithm or model to train. This may refer to the name of a built-in algorithm (e.g. RLlib’s DQN or PPO), or a user-defined trainable function or class registered in the tune registry.
- **stop** (*dict*) – The stopping criteria. The keys may be any field in TrainingResult, whichever is reached first. Defaults to empty dict.
- **config** (*dict*) – Algorithm-specific configuration (e.g. env, hyperparams). Defaults to empty dict.
- **trial_resources** (*dict*) – Machine resources to allocate per trial, e.g. `{"cpu": 64, "gpu": 8}`. Note that GPUs will not be assigned unless you specify them here. Defaults to 1 CPU and 0 GPUs.
- **repeat** (*int*) – Number of times to repeat each trial. Defaults to 1.
- **local_dir** (*str*) – Local dir to save training results to. Defaults to `~/ray_results`.
- **upload_dir** (*str*) – Optional URI to sync training results to (e.g. `s3://your_bucket/path`).
- **checkpoint_freq** (*int*) – How many training iterations between checkpoints. A value of 0 (default) disables checkpointing.
- **max_failures** (*int*) – Try to recover a trial from its last checkpoint at least this many times. Only applies if checkpointing is enabled. Defaults to 3.

An example of this can be found in `hyperband_example.py`.

Alternatively, you can pass in a Python dict. This uses the same fields as the `ray.tune.Experiment`, except the experiment name is the key of the top level dictionary.

```python
run_experiments({
    "my_experiment_name": {
        "run": "my_func",
        "trial_resources": { "cpu": 1, "gpu": 0 },
        "stop": { "mean_accuracy": 100 },
        "config": {
            "alpha": grid_search([0.2, 0.4, 0.6]),
            "beta": grid_search([1, 2]),
        },
        "upload_dir": "s3://your_bucket/path",
        "local_dir": "~/ray_results",
        "max_failures": 2
    }
})
```

An example of this can be found in `async_hyperband_example.py`. 

---

Chapter 1. Example Use
1.10.2 Trial Variant Generation

In the above example, we specified a grid search over two parameters using the `grid_search` helper function. Ray Tune also supports sampling parameters from user-specified lambda functions, which can be used in combination with grid search.

The following shows grid search over two nested parameters combined with random sampling from two lambda functions. Note that the value of `beta` depends on the value of `alpha`, which is represented by referencing `spec.config.alpha` in the lambda function. This lets you specify conditional parameter distributions.

```
"config": {
    "alpha": lambda spec: np.random.uniform(100),
    "beta": lambda spec: spec.config.alpha * np.random.normal(),
    "nn_layers": [grid_search([16, 64, 256]),
                  grid_search([16, 64, 256]),
                ],
},
"repeat": 10,
```

By default, each random variable and grid search point is sampled once. To take multiple random samples or repeat grid search runs, add `repeat: N` to the experiment config. E.g. in the above, "repeat": 10 repeats the 3x3 grid search 10 times, for a total of 90 trials, each with randomly sampled values of `alpha` and `beta`.

For more information on variant generation, see `basic_variant.py`.

1.10.3 Resource Allocation

Ray Tune runs each trial as a Ray actor, allocating the specified GPU and CPU `trial_resources` to each actor (defaulting to 1 CPU per trial). A trial will not be scheduled unless at least that amount of resources is available in the cluster, preventing the cluster from being overloaded.

If GPU resources are not requested, the `CUDA_VISIBLE_DEVICES` environment variable will be set as empty, disallowing GPU access. Otherwise, it will be set to the GPUs in the list (this is managed by Ray).

If your trainable function / class creates further Ray actors or tasks that also consume CPU / GPU resources, you will also want to set `extra_cpu` or `extra_gpu` to reserve extra resource slots for the actors you will create. For example, if a trainable class requires 1 GPU itself, but will launch 4 actors each using another GPU, then it should set "gpu": 1, "extra_gpu": 4.

1.11 HyperBand and Early Stopping

Ray Tune includes distributed implementations of early stopping algorithms such as Median Stopping Rule, HyperBand, and an asynchronous version of HyperBand. These algorithms are very resource efficient and can outperform Bayesian Optimization methods in many cases.

1.11.1 Asynchronous HyperBand

The asynchronous version of HyperBand scheduler can be plugged in on top of an existing grid or random search. This can be done by setting the `scheduler` parameter of `run_experiments`, e.g.

```
run_experiments(..., scheduler=AsyncHyperBandScheduler())
```
Compared to the original version of HyperBand, this implementation provides better parallelism and avoids straggler issues during eliminations. An example of this can be found in `async_hyperband_example.py`. We recommend using this over the standard HyperBand scheduler.

```python
class ray.tune.async_hyperband.AsyncHyperBandScheduler (time_attr='training_iteration',
reward_attr='episode_reward_mean',
max_t=100,
grace_period=10, reduction_factor=3, brackets=3)
```

Implements the Async Successive Halving.

This should provide similar theoretical performance as HyperBand but avoid straggler issues that HyperBand faces. One implementation detail is when using multiple brackets, trial allocation to bracket is done randomly with over a softmax probability.

See [https://openreview.net/forum?id=S1Y7OOIrZ](https://openreview.net/forum?id=S1Y7OOIrZ)

**Parameters**

- `time_attr` (**str**) – The TrainingResult attr to use for comparing time. Note that you can pass in something non-temporal such as `training_iteration` as a measure of progress, the only requirement is that the attribute should increase monotonically.
- `reward_attr` (**str**) – The TrainingResult objective value attribute. As with `time_attr`, this may refer to any objective value. Stopping procedures will use this attribute.
- `max_t` (**float**) – max time units per trial. Trials will be stopped after max_t time units (determined by time_attr) have passed.
- `grace_period` (**float**) – Only stop trials at least this old in time. The units are the same as the attribute named by `time_attr`.
- `reduction_factor` (**float**) – Used to set halving rate and amount. This is simply a unit-less scalar.
- `brackets` (**int**) – Number of brackets. Each bracket has a different halving rate, specified by the reduction factor.

### 1.11.2 HyperBand

**Note:** Note that the HyperBand scheduler requires your trainable to support checkpointing, which is described in Ray Tune documentation. Checkpointing enables the scheduler to multiplex many concurrent trials onto a limited size cluster.

Ray Tune also implements the standard version of HyperBand. You can use it as such:

```python
run_experiments(..., scheduler=HyperBandScheduler())
```

An example of this can be found in `hyperband_example.py`. The progress of one such HyperBand run is shown below.

```plaintext
== Status ==
Using HyperBand: num_stopped=0 total_brackets=5
Round #0:
  Bracket (n=5, r=100, completed=80%): {'PAUSED': 4, 'PENDING': 1}
  Bracket (n=8, r=33, completed=23%): {'PAUSED': 4, 'PENDING': 4}
  Bracket (n=15, r=11, completed=4%): {'RUNNING': 2, 'PAUSED': 2, 'PENDING': 11}
```

(continues on next page)
Bracket (n=34, r=3, completed=0%): {'RUNNING': 2, 'PENDING': 32}
Bracket (n=81, r=1, completed=0%): {'PENDING': 38}

Resources used: 4/4 CPUs, 0/0 GPUs

Result logdir: ~/.ray_results/hyperband_test

PAUSED trials:
  - my_class_0_height=99, width=43: PAUSED [pid=11664], 0 s, 100 ts, 97.1 rew
  - my_class_11_height=85, width=81: PAUSED [pid=11771], 0 s, 33 ts, 32.8 rew
  - my_class_12_height=0, width=52: PAUSED [pid=11785], 0 s, 33 ts, 0 rew
  - my_class_19_height=44, width=88: PAUSED [pid=11811], 0 s, 11 ts, 5.47 rew
  - my_class_27_height=96, width=84: PAUSED [pid=11840], 0 s, 11 ts, 12.5 rew
  ... 5 more not shown

PENDING trials:
  - my_class_10_height=12, width=25: PENDING
  - my_class_13_height=90, width=45: PENDING
  - my_class_14_height=49, width=11: PENDING
  - my_class_15_height=57, width=69: PENDING
  ... 81 more not shown

RUNNING trials:
  - my_class_23_height=75, width=51: RUNNING [pid=11843], 0 s, 1 ts, 1.47 rew
  - my_class_26_height=16, width=48: RUNNING
  - my_class_31_height=40, width=10: RUNNING
  - my_class_53_height=28, width=96: RUNNING

```
class ray.tune.hyperband.HyperBandScheduler(time_attr='training_iteration', reward_attr='episode_reward_mean', max_t=81)
```

Implements the HyperBand early stopping algorithm.

HyperBandScheduler early stops trials using the HyperBand optimization algorithm. It divides trials into brackets of varying sizes, and periodically early stops low-performing trials within each bracket.

To use this implementation of HyperBand with Ray Tune, all you need to do is specify the max length of time a trial can run \( \text{max}_t \), the time units \( \text{time}_{\text{attr}} \), and the name of the reported objective value \( \text{reward}_{\text{attr}} \). We automatically determine reasonable values for the other HyperBand parameters based on the given values.

For example, to limit trials to 10 minutes and early stop based on the \( \text{episode}_{\text{mean}}_{\text{reward}} \) attr, construct:

HyperBand('time_total_s', 'episode_reward_mean', 600)

See also: https://people.eecs.berkeley.edu/~kjamieson/hyperband.html

Parameters

- **time_attr (str)** – The TrainingResult attr to use for comparing time. Note that you can pass in something non-temporal such as \( \text{training}_{\text{iteration}} \) as a measure of progress, the only requirement is that the attribute should increase monotonically.

- **reward_attr (str)** – The TrainingResult objective value attribute. As with \( \text{time}_{\text{attr}} \), this may refer to any objective value. Stopping procedures will use this attribute.

- **max_t (int)** – max time units per trial. Trials will be stopped after \( \text{max}_t \) time units (determined by \( \text{time}_{\text{attr}} \)) have passed. The scheduler will terminate trials after this time has passed. Note that this is different from the semantics of \( \text{max}_t \) as mentioned in the original HyperBand paper.
HyperBand Implementation Details

Implementation details may deviate slightly from theory but are focused on increasing usability. Note: $R$, $s_{\text{max}}$, and $\eta$ are parameters of HyperBand given by the paper. See this post for context.

1. Both $s_{\text{max}}$ (representing the number of brackets - 1) and $\eta$, representing the downsampling rate, are fixed. In many practical settings, $R$, which represents some resource unit and often the number of training iterations, can be set reasonably large, like $R \geq 200$. For simplicity, assume $\eta = 3$. Varying $R$ between $R = 200$ and $R = 1000$ creates a huge range of the number of trials needed to fill up all brackets.

![Graph showing full bracket count vs. max iteration](image)

On the other hand, holding $R$ constant at $R = 300$ and varying $\eta$ also leads to HyperBand configurations that are not very intuitive:
The implementation takes the same configuration as the example given in the paper and exposes `max_t`, which is not a parameter in the paper.

2. The example in the post to calculate `n_0` is actually a little different than the algorithm given in the paper. In this implementation, we implement `n_0` according to the paper (which is `n` in the below example):

3. There are also implementation specific details like how trials are placed into brackets which are not covered in the paper. This implementation places trials within brackets according to smaller bracket first - meaning that with low number of trials, there will be less early stopping.

1.11.3 Median Stopping Rule

The Median Stopping Rule implements the simple strategy of stopping a trial if its performance falls below the median of other trials at similar points in time. You can set the `scheduler` parameter as such:

```python
run_experiments({...}, scheduler=MedianStoppingRule())
```
class ray.tune.median_stopping_rule.MedianStoppingRule(time_attr='time_total_s',
reward_attr='episode_reward_mean',
grace_period=60.0,
min_samples_required=3,
hard_stop=True, verbose=True)

Implements the median stopping rule as described in the Vizier paper:
https://research.google.com/pubs/pub46180.html

Parameters

- **time_attr**(str) – The TrainingResult attr to use for comparing time. Note that you can pass in something non-temporal such as training_iteration as a measure of progress, the only requirement is that the attribute should increase monotonically.

- **reward_attr**(str) – The TrainingResult objective value attribute. As with time_attr, this may refer to any objective value that is supposed to increase with time.

- **grace_period**(float) – Only stop trials at least this old in time. The units are the same as the attribute named by time_attr.

- **min_samples_required**(int) – Min samples to compute median over.

- **hard_stop**(bool) – If False, pauses trials instead of stopping them. When all other trials are complete, paused trials will be resumed and allowed to run FIFO.

- **verbose**(bool) – If True, will output the median and best result each time a trial reports. Defaults to True.

### 1.12 Population Based Training

Ray Tune includes a distributed implementation of Population Based Training (PBT).

#### 1.12.1 PBT Scheduler

Ray Tune’s PBT scheduler can be plugged in on top of an existing grid or random search experiment. This can be enabled by setting the scheduler parameter of run_experiments, e.g.

```python
run_experiments(
    {...},
    scheduler=PopulationBasedTraining(
        time_attr='time_total_s',
        reward_attr='mean_accuracy',
        perturbation_interval=600.0,
        hyperparam_mutations={
            "lr": [1e-3, 5e-4, 1e-4, 5e-5, 1e-5],
            "alpha": lambda: random.uniform(0.0, 1.0),
            ...
        })
)
```

When the PBT scheduler is enabled, each trial variant is treated as a member of the population. Periodically, top-performing trials are checkpointed (this requires your Trainable to support checkpointing). Low-performing trials clone the checkpoints of top performers and perturb the configurations in the hope of discovering an even better variation.
You can run this toy PBT example to get an idea of how how PBT operates. When training in PBT mode, a single trial may see many different hyperparameters over its lifetime, which is recorded in its result.json file. The following figure generated by the example shows PBT discovering new hyperparams over the course of a single experiment:

```
class ray.tune.pbt.PopulationBasedTraining(
    time_attr='time_total_s',
    reward_attr='episode_reward_mean',
    perturbation_interval=60.0,
    hyperparam_mutations={},
    resample_probability=0.25,
    custom_explore_fn=None)
```

Implements the Population Based Training (PBT) algorithm.

https://deepmind.com/blog/population-based-training-neural-networks

PBT trains a group of models (or agents) in parallel. Periodically, poorly performing models clone the state of the top performers, and a random mutation is applied to their hyperparameters in the hopes of outperforming the current top models.

Unlike other hyperparameter search algorithms, PBT mutates hyperparameters during training time. This enables very fast hyperparameter discovery and also automatically discovers good annealing schedules.

This Ray Tune PBT implementation considers all trials added as part of the PBT population. If the number of trials exceeds the cluster capacity, they will be time-multiplexed as to balance training progress across the population.

**Parameters**

- **time_attr** *(str)* – The TrainingResult attr to use for comparing time. Note that you can pass in something non-temporal such as training_iteration as a measure of progress, the only requirement is that the attribute should increase monotonically.

- **reward_attr** *(str)* – The TrainingResult objective value attribute. As with time_attr, this may refer to any objective value. Stopping procedures will use this attribute.

- **perturbation_interval** *(float)* – Models will be considered for perturbation at this interval of time_attr. Note that perturbation incurs checkpoint overhead, so you shouldn’t set this to be too frequent.

- **hyperparam_mutations** *(dict)* – Hyperparams to mutate. The format is as follows: for each key, either a list or function can be provided. A list specifies an allowed set of categorical values. A function specifies the distribution of a continuous parameter. You must specify at least one of hyperparam_mutations or custom_explore_fn.

- **resample_probability** *(float)* – The probability of resampling from the original distribution when applying hyperparam_mutations. If not resampled, the value will be perturbed by a factor of 1.2 or 0.8 if continuous, or changed to an adjacent value if discrete.
• **custom_explore_fn** (*func*) – You can also specify a custom exploration function. This function is invoked as \(f(config)\) after built-in perturbations from `hyperparam_mutations` are applied, and should return `config` updated as needed. You must specify at least one of `hyperparam_mutations` or `custom_explore_fn`.

**Example**

```python
>>> pbt = PopulationBasedTraining(
>>>     time_attr="training_iteration",
>>>     reward_attr="episode_reward_mean",
>>>     perturbation_interval=10,  # every 10 'time_attr' units
>>>     # (training_iterations in this case)
>>>     hyperparam_mutations={
>>>         "factor_1": lambda: random.uniform(0.0, 20.0),
>>>         "factor_2": [1, 10, 100, 1000, 10000],
>>>     })
>>> run_experiments({...}, scheduler=pbt)
```

## 1.13 RLlib: Scalable Reinforcement Learning

RLlib is an open-source library for reinforcement learning that offers both a collection of reference algorithms and scalable primitives for composing new ones.

Learn more about RLlib’s design by reading the ICML paper.

### 1.13.1 Installation

RLlib has extra dependencies on top of `ray`. First, you’ll need to install either PyTorch or TensorFlow. Then, install the Ray RLlib module:

```
pip install tensorflow  # or tensorflow-gpu
pip install ray[rllib]
```

You might also want to clone the Ray repo for convenient access to RLlib helper scripts:

```
git clone https://github.com/ray-project/ray
cd ray/python/ray/rllib
```

### 1.13.2 Training APIs

- Command-line
- Python API
- REST API
1.13.3 Environments

- RLlib Environments Overview
- OpenAI Gym
- Vectorized
- Multi-Agent
- Serving (Agent-oriented)
- Offline Data Ingest
- Batch Asynchronous

1.13.4 Algorithms

- Ape-X Distributed Prioritized Experience Replay
- Asynchronous Advantage Actor-Critic
- Deep Deterministic Policy Gradients
- Deep Q Networks
- Evolution Strategies
- Importance Weighted Actor-Learner Architecture
- Policy Gradients
- Proximal Policy Optimization

1.13.5 Models and Preprocessors

- RLlib Models and Preprocessors Overview
- Built-in Models and Preprocessors
- Custom Models
- Custom Preprocessors
- Customizing Policy Graphs

1.13.6 RLlib Concepts

- Policy Graphs
- Policy Evaluation
- Policy Optimization

1.13.7 Package Reference

- ray.rllib.agents
- ray.rllib.env
- ray.rllib.evaluation
1.14 RLlib Training APIs

1.14.1 Getting Started

At a high level, RLlib provides an Agent class which holds a policy for environment interaction. Through the agent interface, the policy can be trained, checkpointed, or an action computed.

You can train a simple DQN agent with the following command

```python
python ray/python/ray/rllib/train.py --run DQN --env CartPole-v0
```

By default, the results will be logged to a subdirectory of ~/ray_results. This subdirectory will contain a file `params.json` which contains the hyperparameters, a file `result.json` which contains a training summary for each episode and a TensorBoard file that can be used to visualize training process with TensorBoard by running

```bash
tensorboard --logdir=~/ray_results
```

The `train.py` script has a number of options you can show by running

```python
python ray/python/ray/rllib/train.py --help
```

The most important options are for choosing the environment with `--env` (any OpenAI gym environment including ones registered by the user can be used) and for choosing the algorithm with `--run` (available options are PPO, PG, A3C, IMPALA, ES, DDPG, DQN, APEX, and APEX_DDPG).

Specifying Parameters

Each algorithm has specific hyperparameters that can be set with `--config`, in addition to a number of common hyperparameters. See the algorithms documentation for more information.

In an example below, we train A3C by specifying 8 workers through the config flag. function that creates the env to refer to it by name. The contents of the env_config agent config field will be passed to that function to allow the environment to be configured. The return type should be an OpenAI gym.Env. For example:

```python
python ray/python/ray/rllib/train.py --env=PongDeterministic-v4
 --run=A3C --config '{"num_workers": 8}"
```

Evaluating Trained Agents

In order to save checkpoints from which to evaluate agents, set `--checkpoint-freq` (number of training iterations between checkpoints) when running `train.py`.

An example of evaluating a previously trained DQN agent is as follows:

```python
python ray/python/ray/rllib/rollout.py
 ~/ray_results/default/DQN_CartPole-v0_0upjmdgr0/checkpoint-1
 --run DQN --env CartPole-v0
```
The `rollout.py` helper script reconstructs a DQN agent from the checkpoint located at `~/ray_results/default/DQN_CartPole-v0_0upjmdgr0/checkpoint-1` and renders its behavior in the environment specified by `--env`.

### Tuned Examples

Some good hyperparameters and settings are available in the repository (some of them are tuned to run on GPUs). If you find better settings or tune an algorithm on a different domain, consider submitting a Pull Request!

You can run these with the `train.py` script as follows:

```bash
call python ray/python/ray/rllib/train.py -f /path/to/tuned/example.yaml
```

### 1.14.2 Python API

The Python API provides the needed flexibility for applying RLlib to new problems. You will need to use this API if you wish to use custom environments, preprocessors, or models with RLlib.

Here is an example of the basic usage:

```python
import ray
import ray.rllib.agents.ppo as ppo
ray.init()
config = ppo.DEFAULT_CONFIG.copy()
agent = ppo.PPOAgent(config=config, env="CartPole-v0")
# Can optionally call agent.restore(path) to load a checkpoint.
for i in range(1000):
    # Perform one iteration of training the policy with PPO
    result = agent.train()
    print("result: {}".format(result))
    if i % 100 == 0:
        checkpoint = agent.save()
        print("checkpoint saved at", checkpoint)
```

**Note:** It’s recommended that you run RLlib agents with Tune, for easy experiment management and visualization of results. Just set "run": AGENT_NAME, "env": ENV_NAME in the experiment config.

All RLlib agents are compatible with the Tune API. This enables them to be easily used in experiments with Tune. For example, the following code performs a simple hyperparam sweep of PPO:

```python
import ray
import ray.tune as tune
ray.init()
tune.run_experiments({
    "my_experiment": {
        "run": "PPO",
        "env": "CartPole-v0",
        "stop": {"episode_reward_mean": 200},
```
"config": {
  "num_workers": 1,
  "sgd_stepsize": tune.grid_search([0.01, 0.001, 0.0001]),
},
}
}
]

Tune will schedule the trials to run in parallel on your Ray cluster:

```
== Status ==
Using FIFO scheduling algorithm.
Resources requested: 4/4 CPUs, 0/0 GPUs
Result logdir: /home/eric/ray_results/my_experiment
Pending trials:
  - PPO_CartPole-v0_2_sgd_stepsize=0.0001: PENDING
Running trials:
  - PPO_CartPole-v0_0_sgd_stepsize=0.01: RUNNING [pid=21940], 16 s, 4013 ts, 22 rew
  - PPO_CartPole-v0_1_sgd_stepsize=0.001: RUNNING [pid=21942], 27 s, 8111 ts, 54.7 rew
```

### Accessing Global State

It is common to need to access an agent’s internal state, e.g., to set or get internal weights. In RLlib an agent’s state is replicated across multiple policy evaluators (Ray actors) in the cluster. However, you can easily get and update this state between calls to `train()` via `agent.optimizer.foreach_evaluator()` or `agent.optimizer.foreach_evaluator_with_index()`. These functions take a lambda function that is applied with the evaluator as an arg. You can also return values from these functions and those will be returned as a list.

You can also access just the “master” copy of the agent state through `agent.optimizer.local_evaluator`, but note that updates here may not be reflected in remote replicas if you have configured `num_workers > 0`.

### 1.14.3 REST API

In some cases (i.e., when interacting with an external environment) it makes more sense to interact with RLlib as if were an independently running service, rather than RLlib hosting the simulations itself. This is possible via RLlib’s serving env interface.

```python
class ray.rllib.utils.policy_client.PolicyClient(address)
    REST client to interact with a RLlib policy server.

    start_episode (episode_id=None, training_enabled=True)
        Record the start of an episode.

        Parameters
        • `episode_id` (str) – Unique string id for the episode or None for it to be auto-assigned.
        • `training_enabled` (bool) – Whether to use experiences for this episode to improve the policy.

        Returns
        Unique string id for the episode.

        Return type
        episode_id (str)

    get_action (episode_id, observation)
        Record an observation and get the on-policy action.
```


documentation/rllib/examples/ray经纪人/48: 桶将调度这些任务在您的Ray集群上并行运行：
Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().
- **observation** (*obj*) – Current environment observation.

Returns Action from the env action space.

Return type action (*obj*)

**log_action** (*episode_id, observation, action*)

Record an observation and (off-policy) action taken.

Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().
- **observation** (*obj*) – Current environment observation.
- **action** (*obj*) – Action for the observation.

**log_returns** (*episode_id, reward, info=None*)

Record returns from the environment.

The reward will be attributed to the previous action taken by the episode. Rewards accumulate until the next action. If no reward is logged before the next action, a reward of 0.0 is assumed.

Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().
- **reward** (*float*) – Reward from the environment.

**end_episode** (*episode_id, observation*)

Record the end of an episode.

Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().
- **observation** (*obj*) – Current environment observation.

---

**class** `ray.rllib.utils.policy_server.PolicyServer(serving_env, address, port)`

REST server than can be launched from a ServingEnv.

This launches a multi-threaded server that listens on the specified host and port to serve policy requests and forward experiences to RLlib.

### Examples

```python
>>> class CartpoleServing(ServingEnv):
    def __init__(self):
        ServingEnv.__init__(
            self, spaces.Discrete(2),
            spaces.Box(
                low=-10,
                high=10,
                shape=(4,),
                dtype=np.float32))
    def run(self):
        server = PolicyServer(self, "localhost", 8900)
        server.serve_forever()
>>> register_env("srv", lambda _: CartpoleServing())
(continues on next page)```
\begin{verbatim}
>>> pg = PGAgent(env="srv", config={"num_workers": 0})
>>> while True:
>>>     pg.train()

>>> client = PolicyClient("localhost:8900")
>>> eps_id = client.start_episode()

>>> action = client.get_action(eps_id, obs)

...  

>>> client.log_returns(eps_id, reward)

... 

>>> client.log_returns(eps_id, reward)
\end{verbatim}

For a full client / server example that you can run, see the example client script and also the corresponding server script, here configured to serve a policy for the toy CartPole-v0 environment.

## 1.15 RLlib Environments

RLlib works with several different types of environments, including OpenAI Gym, user-defined, multi-agent, and also batched environments.

In the high-level agent APIs, environments are identified with string names. By default, the string will be interpreted as a gym environment name, however you can also register custom environments by name:

```python
import ray
from ray.tune.registry import register_env
from ray.rllib import ppo

def env_creator(env_config):
    import gym
    return gym.make("CartPole-v0")  # or return your own custom env

register_env("my_env", env_creator)
ray.init()
trainer = ppo.PPOAgent(env="my_env", config={
    "env_config": {},  # config to pass to env creator
})

while True:
    print(trainer.train())
```

### 1.15.1 Configuring Environments

In the above example, note that the `env_creator` function takes in an `env_config` object. This is a dict containing options passed in through your agent. You can also access `env_config.worker_index` and `env_config.vector_index` to get the worker id and env id within the worker (if `num_envs_per_worker > 0`). This can be useful if you want to train over an ensemble of different environments, for example:

```python
class MultiEnv(gym.Env):
    def __init__(self, env_config):
        # pick actual env based on worker and env indexes
```

(continues on next page)
```python
self.env = gym.make(
    choose_env_for(env_config.worker_index, env_config.vector_index))
self.action_space = self.env.action_space
self.observation_space = self.env.observation_space

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

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

register_env("multienv", lambda config: MultiEnv(config))
```

### 1.15.2 OpenAI Gym

RLlib uses Gym as its environment interface for single-agent training. For more information on how to implement a custom Gym environment, see the `gym.Env` class definition. You may also find the SimpleCorridor and Carla simulator example env implementations useful as a reference.

#### Performance

There are two ways to scale experience collection with Gym environments:

1. **Vectorization within a single process:** Though many envs can very achieve high frame rates per core, their throughput is limited in practice by policy evaluation between steps. For example, even small TensorFlow models incur a couple milliseconds of latency to evaluate. This can be worked around by creating multiple envs per process and batching policy evaluations across these envs.

   You can configure `{"num_envs_per_worker": M}` to have RLlib create M concurrent environments per worker. RLlib auto-vectorizes Gym environments via `VectorEnv.wrap()`.

2. **Distribute across multiple processes:** You can also have RLlib create multiple processes (Ray actors) for experience collection. In most algorithms this can be controlled by setting the `{"num_workers": N}` config.

![Performance Graph](image)

You can also combine vectorization and distributed execution, as shown in the above figure. Here we plot just the throughput of RLlib policy evaluation from 1 to 128 CPUs. PongNoFrameskip-v4 on GPU scales from 2.4k to 200k actions/s, and Pendulum-v0 on CPU from 15k to 1.5M actions/s. One machine was used for 1-16 workers, and a Ray cluster of four machines for 32-128 workers. Each worker was configured with `num_envs_per_worker=64`.

### 1.15.3 Vectorized

RLlib will auto-vectorize Gym envs for batch evaluation if the `num_envs_per_worker` config is set, or you can define a custom environment class that subclasses `VectorEnv` to implement `vector_step()` and `vector_reset()`.

---

1.15. RLlib Environments
1.15.4 Multi-Agent

A multi-agent environment is one which has multiple acting entities per step, e.g., in a traffic simulation, there may be multiple “car” and “traffic light” agents in the environment. The model for multi-agent in RLlib as follows: (1) as a user you define the number of policies available up front, and (2) a function that maps agent ids to policy ids. This is summarized by the below figure:

The environment itself must subclass the `MultiAgentEnv` interface, which can returns observations and rewards from multiple ready agents per step:

```python
# Example: using a multi-agent env
env = MultiAgentTrafficEnv(num_cars=20, num_traffic_lights=5)

# Observations are a dict mapping agent names to their obs. Not all agents
# may be present in the dict in each time step.
print(env.reset())
{
    "car_1": [[...]],
    "car_2": [[...]],
    "traffic_light_1": [[...]],
}

# Actions should be provided for each agent that returned an observation.
new_obs, rewards, dones, infos = env.step(actions={"car_1": ..., "car_2": ...})

# Similarly, new_obs, rewards, dones, etc. also become dicts
print(rewards)
{"car_1": 3, "car_2": -1, "traffic_light_1": 0}

# Individual agents can early exit; env is done when "__all__" = True
print(dones)
{"car_2": True, "__all__": False}
```

If all the agents will be using the same algorithm class to train, then you can setup multi-agent training as follows:

```python
trainer = pg.PGAgent(env="my_multiagent_env", config={
    "multiagent": {
        "policy_graphs": {
            "car1": (PGPolicyGraph, car_obs_space, car_act_space, {"gamma": 0.85}),
            "car2": (PGPolicyGraph, car_obs_space, car_act_space, {"gamma": 0.99}),
            "traffic_light": (PGPolicyGraph, tl_obs_space, tl_act_space, {}),
        },
        "policy_mapping_fn": lambda agent_id:
            "traffic_light" # Traffic lights are always controlled by this policy
            if agent_id.startswith("traffic_light_")
            else random.choice(["car1", "car2"]), # Randomly choose from car_
    },
},

while True:
    print(trainer.train())
```

RLlib will create three distinct policies and route agent decisions to its bound policy. When an agent first appears in the env, `policy_mapping_fn` will be called to determine which policy it is bound to. RLlib reports separate
training statistics for each policy in the return from `train()`, along with the combined reward.

Here is a simple example training script in which you can vary the number of agents and policies in the environment. For how to use multiple training methods at once (here DQN and PPO), see the two-trainer example.

To scale to hundreds of agents, MultiAgentEnv batches policy evaluations across multiple agents internally. It can also be auto-vectorized by setting `num_envs_per_worker > 1`.

### 1.15.5 Serving

In many situations, it does not make sense for an environment to be “stepped” by RLlib. For example, if a policy is to be used in a web serving system, then it is more natural to instead query a service that serves policy decisions, and for that service to learn from experience over time.

RLlib provides the `ServingEnv` class for this purpose. Unlike other envs, ServingEnv runs as its own thread of control. At any point, that thread can query the current policy for decisions via `self.get_action()` and reports rewards via `self.log_returns()`. This can be done for multiple concurrent episodes as well.

For example, ServingEnv can be used to implement a simple REST policy server that learns over time using RLlib. In this example RLlib runs with `num_workers=0` to avoid port allocation issues, but in principle this could be scaled by increasing `num_workers`.

#### Offline Data

ServingEnv also provides a `self.log_action()` call to support off-policy actions. This allows the client to make independent decisions, e.g., to compare two different policies, and for RLlib to still learn from those off-policy actions. Note that this requires the algorithm used to support learning from off-policy decisions (e.g., DQN).

The `log_action` API of ServingEnv can be used to ingest data from offline logs. The pattern would be as follows: First, some policy is followed to produce experience data which is stored in some offline storage system. Then, RLlib creates a number of workers that use a ServingEnv to read the logs in parallel and ingest the experiences. After a round of training completes, the new policy can be deployed to collect more experiences.

Note that envs can read from different partitions of the logs based on the `worker_index` attribute of the `env context` passed into the environment constructor.

### 1.15.6 Batch Asynchronous

The lowest-level “catch-all” environment supported by RLlib is `AsyncVectorEnv`. AsyncVectorEnv models multiple agents executing asynchronously in multiple environments. A call to `poll()` returns observations from ready agents keyed by their environment and agent ids, and actions for those agents can be sent back via `send_actions()`. This interface can be subclassed directly to support batched simulators such as ELF.

Under the hood, all other envs are converted to AsyncVectorEnv by RLlib so that there is a common internal path for policy evaluation.

### 1.16 RLlib Algorithms

#### 1.16.1 Ape-X Distributed Prioritized Experience Replay

[A paper] [implementation] Ape-X variations of DQN and DDPG (APEX_DQN, APEX_DDPG in RLlib) use a single GPU learner and many CPU workers for experience collection. Experience collection can scale to hundreds of CPU workers due to the distributed prioritization of experience prior to storage in replay buffers.
1.16.2 Asynchronous Advantage Actor-Critic

[paper] [implementation] RLlib’s A3C uses the AsyncGradientsOptimizer to apply gradients computed remotely on policy evaluation actors. It scales to up to 16-32 worker processes, depending on the environment. Both a TensorFlow (LSTM), and PyTorch version are available. Note that if you have a GPU, IMPALA probably will perform better than A3C.

Tuned examples: PongDeterministic-v4, PyTorch version

1.16.3 Deep Deterministic Policy Gradients

[paper] [implementation] DDPG is implemented similarly to DQN (below). The algorithm can be scaled by increasing the number of workers, switching to AsyncGradientsOptimizer, or using Ape-X.

Tuned examples: Pendulum-v0, MountainCarContinuous-v0, HalfCheetah-v2

1.16.4 Deep Q Networks

[paper] [implementation] RLlib DQN is implemented using the SyncReplayOptimizer. The algorithm can be scaled by increasing the number of workers, using the AsyncGradientsOptimizer for async DQN, or using Ape-X. Memory usage is reduced by compressing samples in the replay buffer with LZ4.

Tuned examples: PongDeterministic-v4
1.16.5 Evolution Strategies

[paper] [implementation] Code here is adapted from https://github.com/openai/evolution-strategies-starter to execute in the distributed setting with Ray.

Tuned examples: Humanoid-v1

![Graph showing comparison between Reference ES and RLlib ES](https://example.com/chart.png)

Fig. 2: RLlib’s ES implementation scales further and is faster than a reference Redis implementation.

1.16.6 Importance Weighted Actor-Learner Architecture

[paper] [implementation] In IMPALA, a central learner runs SGD in a tight loop while asynchronously pulling sample batches from many actor processes. RLlib’s IMPALA implementation uses DeepMind’s reference V-trace code. Note that we do not provide a deep residual network out of the box, but one can be plugged in as a custom model.

Tuned examples: PongNoFrameskip-v4, vectorized configuration

![Graphs showing ray/tune/episode_reward_mean and ray/tune/timesteps_total](https://example.com/impala_graphs.png)

Fig. 3: RLlib’s IMPALA implementation scales from 16 to 128 workers on PongNoFrameskip-v4. With vectorization, similar learning performance to 128 workers can be achieved with only 32 workers. This is about an order of magnitude faster than A3C (not shown here), with similar sample efficiency.
1.16.7 Policy Gradients

[paper] [implementation] We include a vanilla policy gradients implementation as an example algorithm. This is usually outperformed by PPO.

Tuned examples: CartPole-v0

1.16.8 Proximal Policy Optimization

[paper] [implementation] PPO’s clipped objective supports multiple SGD passes over the same batch of experiences. RLlib’s multi-GPU optimizer pins that data in GPU memory to avoid unnecessary transfers from host memory, substantially improving performance over a naive implementation. RLlib’s PPO scales out using multiple workers for experience collection, and also with multiple GPUs for SGD.

Tuned examples: Humanoid-v1, Hopper-v1, Pendulum-v0, PongDeterministic-v4, Walker2d-v1

![Fig. 4: RLlib’s multi-GPU PPO scales to multiple GPUs and hundreds of CPUs. Here we compare against a reference MPI-based implementation.](image)

1.17 RLlib Models and Preprocessors

The following diagram provides a conceptual overview of data flow between different components in RLlib. We start with an Environment, which given an action produces an observation. The observation is preprocessed by a Preprocessor and Filter (e.g. for running mean normalization) before being sent to a neural network Model. The model output is in turn interpreted by an ActionDistribution to determine the next action.

The components highlighted in green can be replaced with custom user-defined implementations, as described in the next sections. The purple components are RLlib internal, which means they can only be modified by changing the algorithm source code.

1.17.1 Built-in Models and Preprocessors

RLlib picks default models based on a simple heuristic: a vision network for image observations, and a fully connected network for everything else. These models can be configured via the model config key, documented in the model catalog. Note that you’ll probably have to configure conv_filters if your environment observations have custom sizes, e.g., "model": {"dim": 42, "conv_filters": [[16, [4, 4], 2], [32, [4, 4], 2], [512, [11, 11], 1]} for 42x42 observations.
In addition, if you set "model": {"use_lstm": true}, then the model output will be further processed by a LSTM cell. More generally, RLlib supports the use of recurrent models for its algorithms (A3C, PG out of the box), and RNN support is built into its policy evaluation utilities.

For preprocessors, RLlib tries to pick one of its built-in preprocessor based on the environment’s observation space. Discrete observations are one-hot encoded, Atari observations downscaled, and Tuple observations flattened (there isn’t native tuple support yet, but you can reshape the flattened observation in a custom model). Note that for Atari, DQN defaults to using the DeepMind preprocessors, which are also used by the OpenAI baselines library.

### 1.17.2 Custom Models

Custom models should subclass the common RLlib model class and override the `_build_layers` method. This method takes in a tensor input (observation), and returns a feature layer and float vector of the specified output size. The model can then be registered and used in place of a built-in model:

```python
import ray
import ray.rllib.agents.ppo as ppo
from ray.rllib.models import ModelCatalog, Model

class MyModelClass(Model):
    def _build_layers(self, inputs, num_outputs, options):
        layer1 = slim.fully_connected(inputs, 64, ...)
        layer2 = slim.fully_connected(inputs, 64, ...)
        ...
        return layerN, layerN_minus_1

ModelCatalog.register_custom_model("my_model", MyModelClass)

ray.init()
agent = ppo.PPOAgent(env="CartPole-v0", config={
    "model": {
        "custom_model": "my_model",
        "custom_options": {}, # extra options to pass to your model
    },
})
```

For a full example of a custom model in code, see the Carla RLlib model and associated training scripts. The CarlaModel class defined there operates over a composite (Tuple) observation space including both images and scalar measurements.

### 1.17.3 Custom Preprocessors

Similarly, custom preprocessors should subclass the RLlib preprocessor class and be registered in the model catalog:

```python
import ray
import ray.rllib.agents.ppo as ppo
from ray.rllib.models.preprocessors import Preprocessor

class MyPreprocessorClass(Preprocessor):
    def _init(self):
        self.shape = ...  # perhaps varies depending on self._options

    def transform(self, observation):
        return ...  # return the preprocessed observation
```

(continues on next page)
ModelCatalog.register_custom_preprocessor("my_prep", MyPreprocessorClass)

ray.init()
agent = ppo.PPOAgent(env="CartPole-v0", config={
  "model": {
    "custom_preprocessor": "my_prep",
    "custom_options": {},  # extra options to pass to your preprocessor
  },
})

1.17.4 Customizing Policy Graphs

For deeper customization of algorithms, you can modify the policy graphs of the agent classes. Here's an example of extending the DDPG policy graph to specify custom sub-network modules:

```python
from ray.rllib.models import ModelCatalog
from ray.rllib.agents.ddpg.ddpg_policy_graph import DDPGPolicyGraph as BaseDDPGPolicyGraph
class CustomPNetwork(object):
    def __init__(self, dim_actions, hiddens, activation):
        action_out = ...
        # Use sigmoid layer to bound values within (0, 1)
        # shape of action_scores is [batch_size, dim_actions]
        self.action_scores = layers.fully_connected(
            action_out, num_outputs=dim_actions, activation_fn=tf.nn.sigmoid)

class CustomQNetwork(object):
    def __init__(self, action_inputs, hiddens, activation):
        q_out = ...
        self.value = layers.fully_connected(
            q_out, num_outputs=1, activation_fn=None)

class CustomDDPGPolicyGraph(BaseDDPGPolicyGraph):
    def _build_p_network(self, obs):
        return CustomPNetwork(
            self.dim_actions,
            self.config["actor_hiddens"],
            self.config["actor_hidden_activation"]).
            action_scores

    def _build_q_network(self, obs, actions):
        return CustomQNetwork(
            actions,
            self.config["critic_hiddens"],
            self.config["critic_hidden_activation"]).
            value
```

Then, you can create an agent with your custom policy graph by:

```python
from ray.rllib.agents.ddpg.ddpg import DDPGAgent
from custom_policy_graph import CustomDDPGPolicyGraph
DDPGAgent._policy_graph = CustomDDPGPolicyGraph
agent = DDPGAgent(...)  
```

That’s it. In this example we overrode existing methods of the existing DDPG policy graph, i.e., `_build_q_network`,
_build_p_network, _build_action_network, _build_actor_critic_loss, but you can also replace the entire graph class entirely.

1.18 RLlib Concepts

Note: To learn more about these concepts, see also the ICML paper.

1.18.1 Policy Graphs

Policy graph classes encapsulate the core numerical components of RL algorithms. This typically includes the policy model that determines actions to take, a trajectory postprocessor for experiences, and a loss function to improve the policy given postprocessed experiences. For a simple example, see the policy gradients graph definition.

Most interaction with deep learning frameworks is isolated to the PolicyGraph interface, allowing RLlib to support multiple frameworks. To simplify the definition of policy graphs, RLlib includes Tensorflow and PyTorch-specific templates.

1.18.2 Policy Evaluation

Given an environment and policy graph, policy evaluation produces batches of experiences. This is your classic “environment interaction loop”. Efficient policy evaluation can be burdensome to get right, especially when leveraging vectorization, RNNs, or when operating in a multi-agent environment. RLlib provides a PolicyEvaluator class that manages all of this, and this class is used in most RLlib algorithms.

You can also use policy evaluation standalone to produce batches of experiences. This can be done by calling ev.sample() on an evaluator instance, or ev.sample.remote() in parallel on evaluator instances created as Ray actors (see PolicyEvalutor.as_remote()).

1.18.3 Policy Optimization

Similar to how a gradient-descent optimizer can be used to improve a model, RLlib’s policy optimizers implement different strategies for improving a policy graph.

For example, in A3C you’d want to compute gradients asynchronously on different workers, and apply them to a central policy graph replica. This strategy is implemented by the AsyncGradientsOptimizer. Another alternative is to gather experiences synchronously in parallel and optimize the model centrally, as in SyncSamplesOptimizer. Policy optimizers abstract these strategies away into reusable modules.

1.19 RLlib Package Reference

1.19.1 ray.rllib.agents

class ray.rllib.agents.Agent(config=None, env=None, logger_creator=None)

All RLlib agents extend this base class.

Agent objects retain internal model state between calls to train(), so you should create a new agent instance for each training session.
env_creator
   func – Function that creates a new training env.

cfg
   obj – Algorithm-specific configuration data.

logdir
   str – Directory in which training outputs should be placed.

make_local_evaluator (env_creator, policy_graph)
   Convenience method to return configured local evaluator.

make_remote_evaluators (env_creator, policy_graph, count, remote_args)
   Convenience method to return a number of remote evaluators.

class method resource_help (config)
   Returns a help string for configuring this trainable’s resources.

iteration
   Current training iter, auto-incremented with each train() call.

compute_action (observation, state=None, policy_id='default')
   Computes an action for the specified policy.

   Parameters
      • observation (obj) – observation from the environment.
      • state (list) – RNN hidden state, if any.
      • policy_id (str) – policy to query (only applies to multi-agent).

get_weights (policies=None)
   Return a dictionary of policy ids to weights.

   Parameters policies (list) – Optional list of policies to return weights for, or None for all policies.

set_weights (weights)
   Set policy weights by policy id.

   Parameters weights (dict) – Map of policy ids to weights to set.

ray.rllib.agents.with_common_config (extra_config)
   Returns the given config dict merged with common agent confs.

class ray.rllib.agents.a3c.A3CAgent (config=None, env=None, logger_creator=None)
   A3C implementations in TensorFlow and PyTorch.

class ray.rllib.agents.ddpg.ApexDDPGAgent (config=None, env=None, logger_creator=None)
   DDPG variant that uses the Ape-X distributed policy optimizer.

   By default, this is configured for a large single node (32 cores). For running in a large cluster, increase the num_workers config var.

class ray.rllib.agents.ddpg.DDPGAgent (config=None, env=None, logger_creator=None)
   DDPG implementation in TensorFlow.

class ray.rllib.agents.dqn.ApexAgent (config=None, env=None, logger_creator=None)
   DQN variant that uses the Ape-X distributed policy optimizer.

   By default, this is configured for a large single node (32 cores). For running in a large cluster, increase the num_workers config var.
class ray.rllib.agents.dqn.DQNAgent (config=None, env=None, logger_creator=None)
DQN implementation in TensorFlow.

class ray.rllib.agents.es.ESAgent (config=None, env=None, logger_creator=None)
Large-scale implementation of Evolution Strategies in Ray.

class ray.rllib.agents.pg.PGAgent (config=None, env=None, logger_creator=None)
Simple policy gradient agent.

This is an example agent to show how to implement algorithms in RLlib. In most cases, you will probably want to use the PPO agent instead.

class ray.rllib.agents.impala.ImpalaAgent (config=None, env=None, logger_creator=None)
IMPALA implementation using DeepMind’s V-trace.

class ray.rllib.agents.ppo.PPOAgent (config=None, env=None, logger_creator=None)
Multi-GPU optimized implementation of PPO in TensorFlow.

1.19.2 ray.rllib.env

class ray.rllib.env.AsyncVectorEnv
The lowest-level env interface used by RLlib for sampling.

AsyncVectorEnv models multiple agents executing asynchronously in multiple environments. A call to poll() returns observations from ready agents keyed by their environment and agent ids, and actions for those agents can be sent back via send_actions().

All other env types can be adapted to AsyncVectorEnv. RLlib handles these conversions internally in PolicyEvaluator, for example:

gym.Env => rllib.VectorEnv => rllib.AsyncVectorEnv
rllib.MultiAgentEnv => rllib.AsyncVectorEnv
rllib.ServingEnv => rllib.AsyncVectorEnv

Examples

```python
>>> env = MyAsyncVectorEnv()
>>> obs, rewards, dones, infos, off_policy_actions = env.poll()
>>> print(obs)
{
    "env_0": {
        "car_0": [2.4, 1.6],
        "car_1": [3.4, -3.2],
    }
}
>>> env.send_actions(
    actions={
        "env_0": {
            "car_0": 0,
            "car_1": 1,
        }
    })
>>> obs, rewards, dones, infos, off_policy_actions = env.poll()
>>> print(obs)
{
    "env_0": {
        "car_0": [4.1, 1.7],
    }
}
```

(continues on next page)
"car_1": [3.2, -4.2],
}
}

>>> print(dones)
{
  "env_0": {
    "__all__": False,
    "car_0": False,
    "car_1": True,
  }
}

static wrap_async (env, make_env=None, num_envs=1)

Wraps any env type as needed to expose the async interface.

poll ()

Returns observations from ready agents.

The returns are two-level dicts mapping from env_id to a dict of agent_id to values. The number of agents
and envs can vary over time.

Returns

• obs (dict) (New observations for each ready agent.)

• rewards (dict) (Reward values for each ready agent. If the) – episode is just started, the
value will be None.

• dones (dict) (Done values for each ready agent. The special key) – “__all__” is used to
indicate env termination.

• infos (dict) (Info values for each ready agent.)

• off_policy_actions (dict) (Agents may take off-policy actions. When) – that happens,
there will be an entry in this dict that contains the taken action. There is no need to
send_actions() for agents that have already chosen off-policy actions.

send_actions (action_dict)

Called to send actions back to running agents in this env.

Actions should be sent for each ready agent that returned observations in the previous poll() call.

Parameters action_dict (dict) – Actions values keyed by env_id and agent_id.

try_reset (env_id)

Attempt to reset the env with the given id.

If the environment does not support synchronous reset, None can be returned here.

Returns Resetted observation or None if not supported.

Return type obs (dict|None)

get_unwrapped ()

Return a reference to some underlying gym env, if any.

Returns Underlying gym env or None.

Return type env (gym.Env|None)

class ray.rllib.env.MultiAgentEnv

An environment that hosts multiple independent agents.
Agents are identified by (string) agent ids. Note that these “agents” here are not to be confused with RLlib agents.

Examples

```python
>>> env = MyMultiAgentEnv()
>>> obs = env.reset()
>>> print(obs)
{
    "car_0": [2.4, 1.6],
    "car_1": [3.4, -3.2],
    "traffic_light_1": [0, 3, 5, 1],
}
>>> obs, rewards, dones, infos = env.step(
    action_dict={
        "car_0": 1, "car_1": 0, "traffic_light_1": 2,
    })
>>> print(rewards)
{
    "car_0": 3,
    "car_1": -1,
    "traffic_light_1": 0,
}
>>> print(dones)
{
    "car_0": False,
    "car_1": True,
    "__all__": False,
}
reset ()
Resets the env and returns observations from ready agents.

Returns New observations for each ready agent.

Return type obs (dict)

step (action_dict)
Returns observations from ready agents.

The returns are dicts mapping from agent_id strings to values. The number of agents in the env can vary over time.

Returns

- obs (dict) (New observations for each ready agent.)
- rewards (dict) (Reward values for each ready agent. If the) – episode is just started, the value will be None.
- dones (dict) (Done values for each ready agent. The special key) – “__all__” is used to indicate env termination.
- infos (dict) (Info values for each ready agent.)

class ray.rllib.env.ServingEnv (action_space, observation_space, max_concurrent=100)
An environment that provides policy serving.

Unlike simulator envs, control is inverted. The environment queries the policy to obtain actions and logs observations and rewards for training. This is in contrast to gym.Env, where the algorithm drives the simulation
through `env.step()` calls.

You can use `ServingEnv` as the backend for policy serving (by serving HTTP requests in the run loop), for ingesting offline logs data (by reading offline transitions in the run loop), or other custom use cases not easily expressed through `gym.Env`.

`ServingEnv` supports both on-policy serving (through `self.get_action()`), and off-policy serving (through `self.log_action()`).

This env is thread-safe, but individual episodes must be executed serially.

**Examples**

```python
>>> register_env("my_env", lambda config: YourServingEnv(config))
>>> agent = DQNAgent(env="my_env")
>>> while True:
    print(agent.train())
```

**run()**

Override this to implement the run loop.

Your loop should continuously:

1. Call `self.start_episode()`
2. Call `self.get_action()` or `self.log_action()`
3. Call `self.log_returns()`
4. Call `self.end_episode()`
5. Wait if nothing to do.

Multiple episodes may be started at the same time.

**start_episode** *(episode_id=None, training_enabled=True)*

Record the start of an episode.

**Parameters**

- `episode_id` *(str)* – Unique string id for the episode or None for it to be auto-assigned.
- `training_enabled` *(bool)* – Whether to use experiences for this episode to improve the policy.

**Returns** Unique string id for the episode.

**Return type** episode_id (str)

**get_action** *(episode_id, observation)*

Record an observation and get the on-policy action.

**Parameters**

- `episode_id` *(str)* – Episode id returned from `start_episode()`.
- `observation` *(obj)* – Current environment observation.

**Returns** Action from the env action space.

**Return type** action (obj)

**log_action** *(episode_id, observation, action)*

Record an observation and (off-policy) action taken.
Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().
- **observation** (*obj*) – Current environment observation.
- **action** (*obj*) – Action for the observation.

**log_returns** (*episode_id, reward, info=None*)

Record returns from the environment.

The reward will be attributed to the previous action taken by the episode. Rewards accumulate until the next action. If no reward is logged before the next action, a reward of 0.0 is assumed.

Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().
- **reward** (*float*) – Reward from the environment.
- **info** (*dict*) – Optional info dict.

**end_episode** (*episode_id, observation*)

Record the end of an episode.

Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().
- **observation** (*obj*) – Current environment observation.

**class** `ray.rllib.env.VectorEnv`

An environment that supports batch evaluation.

Subclasses must define the following attributes:

- **action_space**
  `gym.Space` – Action space of individual envs.

- **observation_space**
  `gym.Space` – Observation space of individual envs.

- **num_envs**
  `int` – Number of envs in this vector env.

- **vector_reset** ()
  Resets all environments.

  Returns Vector of observations from each environment.

  Return type obs (list)

- **reset_at** (*index*)
  Resets a single environment.

  Returns Observations from the resetted environment.

  Return type obs (obj)

- **vector_step** (*actions*)
  Vectorized step.

  Parameters **actions** (*list*) – Actions for each env.

  Returns New observations for each env. rewards (list): Reward values for each env. dones (list): Done values for each env. infos (list): Info values for each env.

  Return type obs (list)
get_unwrapped()

Returns a single instance of the underlying env.

class ray.rllib.env.EnvContext(env_config, worker_index, vector_index=0)

Wraps env configurations to include extra rllib metadata.

These attributes can be used to parameterize environments per process. For example, one might use `worker_index` to control which data file an environment reads in on initialization.

RLlib auto-sets these attributes when constructing registered envs.

worker_index

int – When there are multiple workers created, this uniquely identifies the worker the env is created in.

vector_index

int – When there are multiple envs per worker, this uniquely identifies the env index within the worker.

1.19.3 ray.rllib.evaluation

class ray.rllib.evaluation.EvaluatorInterface

This is the interface between policy optimizers and policy evaluation.

See also: PolicyEvaluator

sample()

Returns a batch of experience sampled from this evaluator.

This method must be implemented by subclasses.

Returns A columnar batch of experiences (e.g., tensors), or a multi-agent batch.

Return type SampleBatch|MultiAgentBatch

Examples

```python
>>> print(ev.sample())
SampleBatch({"obs": [1, 2, 3], "action": [0, 1, 0], ...})
```

compute_gradients(samples)

Returns a gradient computed w.r.t the specified samples.

This method must be implemented by subclasses.

Returns A list of gradients that can be applied on a compatible evaluator. In the multi-agent case, returns a dict of gradients keyed by policy graph ids. An info dictionary of extra metadata is also returned.

Return type (grads, info)

Examples

```python
>>> batch = ev.sample()
>>> grads, info = ev2.compute_gradients(samples)
```

apply_gradients(grads)

Applies the given gradients to this evaluator’s weights.

This method must be implemented by subclasses.
Examples

```python
>>> samples = ev1.sample()
>>> grads, info = ev2.compute_gradients(samples)
>>> ev1.apply_gradients(grads)
```

**get_weights()**
Returns the model weights of this Evaluator.

This method must be implemented by subclasses.

- **Returns**: weights that can be set on a compatible evaluator. info: dictionary of extra metadata.
- **Return type**: object

**Examples**

```python
>>> weights = ev1.get_weights()
```

**set_weights(weights)**
Sets the model weights of this Evaluator.

This method must be implemented by subclasses.

**Examples**

```python
>>> weights = ev1.get_weights()
>>> ev2.set_weights(weights)
```

**compute_apply(samples)**
Fused compute gradients and apply gradients call.

- **Returns**: dictionary of extra metadata from compute_gradients().
- **Return type**: info

**Examples**

```python
>>> batch = ev.sample()
>>> ev.compute_apply(samples)
```

**get_host()**
Returns the hostname of the process running this evaluator.

**apply(func, *args)**
Apply the given function to this evaluator instance.
class ray.rllib.evaluation.PolicyEvaluator:
    (env_creator, policy_graph, policy_mapping_fn=None, policies_to_train=None, tf_session_creator=None, batch_steps=100, batch_mode='truncate_episodes', episode_horizon=None, preprocessor'_pref='rllib', sample_async=False, compress_observations=False, num_envs=1, observation_filter='NoFilter', env_config=None, model_config=None, policy_config=None, worker_index=0)

Common PolicyEvaluator implementation that wraps a PolicyGraph.

This class wraps a policy graph instance and an environment class to collect experiences from the environment. You can create many replicas of this class as Ray actors to scale RL training.

This class supports vectorized and multi-agent policy evaluation (e.g., VectorEnv, MultiAgentEnv, etc.)

Examples

>>> # Create a policy evaluator and using it to collect experiences.
>>> evaluator = PolicyEvaluator(
...    env_creator=lambda _: gym.make("CartPole-v0"),
...    policy_graph=PGPolicyGraph)
>>> print(evaluator.sample())
SampleBatch({
    "obs": [[...]], "actions": [[...]], "rewards": [[...]],
    "dones": [[...]], "new_obs": [[...]]})

>>> # Creating policy evaluators using optimizer_cls.make().
>>> optimizer = SyncSamplesOptimizer.make(
...    evaluator_cls=PolicyEvaluator,
...    evaluator_args={
...        "env_creator": lambda _: gym.make("CartPole-v0"),
...        "policy_graph": PGPolicyGraph,
...    },
...    num_workers=10)
>>> for _ in range(10): optimizer.step()

>>> # Creating a multi-agent policy evaluator
>>> evaluator = PolicyEvaluator(
...    env_creator=lambda _: MultiAgentTrafficGrid(num_cars=25),
...    policy_graphs={
...        "car_policy1": (PGPolicyGraph, Box(...), Discrete(...), {"gamma": 0.99}),
...        "car_policy2": (PGPolicyGraph, Box(...), Discrete(...), {"gamma": 0.95}),
...        "traffic_light_policy": (PGPolicyGraph, Box(...), Discrete(...), {}),
...    },
...    policy_mapping_fn=lambda agent_id:
...        random.choice(["car_policy1", "car_policy2"])
...        if agent_id.startswith("car_") else "traffic_light_policy"
(continues on next page)
```python
>>> print(evaluator.sample().keys())
MultiAgentBatch({
    "car_policy1": SampleBatch(...),
    "car_policy2": SampleBatch(...),
    "traffic_light_policy": SampleBatch(...)})
```

**sample()**

Evaluate the current policies and return a batch of experiences.

**Returns** SampleBatch|MultiAgentBatch from evaluating the current policies.

**sample_with_count()**

Same as sample() but returns the count as a separate future.

**for_policy(func, policy_id='default')**

Apply the given function to the specified policy graph.

**foreach_policy(func)**

Apply the given function to each (policy, policy_id) tuple.

**foreach_trainable_policy(func)**

Apply the given function to each (policy, policy_id) tuple.

This only applies func to policies in `self.policies_to_train`.

**sync_filters(new_filters)**

Changes self’s filter to given and rebases any accumulated delta.

**Parameters**

- **new_filters (dict)** – Filters with new state to update local copy.

**get_filters(flush_after=False)**

Returns a snapshot of filters.

**Parameters**

- **flush_after (bool)** – Clears the filter buffer state.

**Returns** Dict for serializable filters

**Return type**

return_filters (dict)

**get_weights(policies=None)**

Returns the model weights of this Evaluator.

This method must be implemented by subclasses.

**Returns** weights that can be set on a compatible evaluator. info: dictionary of extra metadata.

**Return type**

object

**Examples**

```python
>>> weights = ev1.get_weights()
```

**set_weights(weights)**

Sets the model weights of this Evaluator.

This method must be implemented by subclasses.
Examples

```python
>>> weights = ev1.get_weights()
>>> ev2.set_weights(weights)
```

**compute_gradients** *(samples)*

Returns a gradient computed w.r.t the specified samples.

This method must be implemented by subclasses.

- **Returns**: A list of gradients that can be applied on a compatible evaluator. In the multi-agent case, returns a dict of gradients keyed by policy graph ids. An info dictionary of extra metadata is also returned.

- **Return type**: (grads, info)

Examples

```python
>>> batch = ev.sample()
>>> grads, info = ev2.compute_gradients(samples)
```

**apply_gradients** *(grads)*

Applies the given gradients to this evaluator’s weights.

This method must be implemented by subclasses.

Examples

```python
>>> samples = ev1.sample()
>>> grads, info = ev2.compute_gradients(samples)
>>> ev1.apply_gradients(grads)
```

**compute_apply** *(samples)*

Fused compute gradients and apply gradients call.

- **Returns**: dictionary of extra metadata from compute_gradients().

- **Return type**: info

Examples

```python
>>> batch = ev.sample()
>>> ev.compute_apply(samples)
```

**class** `ray.rllib.evaluation.PolicyGraph` *(observation_space, action_space, config)*

An agent policy and loss, i.e., a TFPolicyGraph or other subclass.

This object defines how to act in the environment, and also losses used to improve the policy based on its experiences. Note that both policy and loss are defined together for convenience, though the policy itself is logically separate.

All policies can directly extend PolicyGraph, however TensorFlow users may find TFPolicyGraph simpler to implement. TFPolicyGraph also enables RLlib to apply TensorFlow-specific optimizations such as fusing multiple policy graphs and multi-GPU support.
**observation_space**
gym.Space – Observation space of the policy.

**action_space**
gym.Space – Action space of the policy.

**compute_actions** *(obs_batch, state_batches, is_training=False)*
Compute actions for the current policy.

**Parameters**
- **obs_batch** *(np.ndarray)* – batch of observations
- **state_batches** *(list)* – list of RNN state input batches, if any
- **is_training** *(bool)* – whether we are training the policy

**Returns**
- batch of output actions, with shape like [BATCH_SIZE, ACTION_SHAPE].
- state_outs (list): list of RNN state output batches, if any, with shape like [STATE_SIZE, BATCH_SIZE].
- info (dict): dictionary of extra feature batches, if any, with shape like 
  {“f1”: [BATCH_SIZE, ...], “f2”: [BATCH_SIZE, ...]}.

**Return type** actions (np.ndarray)

**compute_single_action** *(obs, state, is_training=False)*
Unbatched version of compute_actions.

**Parameters**
- **obs** *(obj)* – single observation
- **state_batches** *(list)* – list of RNN state inputs, if any
- **is_training** *(bool)* – whether we are training the policy

**Returns**
- single action
- state_outs (list): list of RNN state outputs, if any
- info (dict): dictionary of extra features, if any

**Return type** actions (obj)

**postprocess_trajectory** *(sample_batch, other_agent_batches=None)*
Implements algorithm-specific trajectory postprocessing.

**Parameters**
- **sample_batch** *(SampleBatch)* – batch of experiences for the policy, which will contain at most one episode trajectory.
- **other_agent_batches** *(dict)* – In a multi-agent env, this contains a mapping of agent ids to (policy_graph, agent_batch) tuples containing the policy graph and experiences of the other agent.

**Returns**
- postprocessed sample batch.

**Return type** SampleBatch

**compute_gradients** *(postprocessed_batch)*
Computes gradients against a batch of experiences.

**Returns**
- List of gradient output values

**Return type** grads (list)
**apply_gradients** *(gradients)*  
Applies previously computed gradients.

**Returns** Extra policy-specific values  
**Return type** info (dict)

**compute_apply** *(samples)*  
Fused compute gradients and apply gradients call.

**Returns** dictionary of extra metadata from compute_gradients(). apply_info: dictionary of extra metadata from apply_gradients().  
**Return type** grad_info

**Examples**

```python  
>>> batch = ev.sample()  
>>> ev.compute_apply(samples)  
```

**get_weights** ()  
Returns model weights.

**Returns** Serializable copy or view of model weights  
**Return type** weights (obj)

**set_weights** *(weights)*  
Sets model weights.

**Parameters** weights *(obj)* – Serializable copy or view of model weights

**get_initial_state** ()  
Returns initial RNN state for the current policy.

**get_state** ()  
Saves all local state.

**Returns** Serialized local state.  
**Return type** state (obj)

**set_state** *(state)*  
Restores all local state.

**Parameters** state *(obj)* – Serialized local state.

class **ray.rllib.evaluation.TFPolicyGraph** *(observation_space, action_space, sess, obs_input, action_sampler, loss, loss_inputs, state_inputs=None, state_outputs=None, seq_lens=None, max_seq_len=20)*  
An agent policy and loss implemented in TensorFlow.

Extending this class enables RLlib to perform TensorFlow specific optimizations on the policy graph, e.g., parallelization across gpus or fusing multiple graphs together in the multi-agent setting.

Input tensors are typically shaped like [BATCH_SIZE, ...].

**observation_space**  
gym.Space – observation space of the policy.

**action_space**  
gym.Space – action space of the policy.
Examples

```python
>>> policy = TFPolicyGraphSubclass(
    sess, obs_input, action_sampler, loss, loss_inputs, is_training)
>>> print(policy.compute_actions([1, 0, 2]))
(array([0, 1, 1]), [], {})
>>> print(policy.postprocess_trajectory(SampleBatch({...})))
SampleBatch({"action": ..., "advantages": ..., ...})
```

**compute_actions** *(obs_batch=None, state_batches=None, is_training=False)*
Compute actions for the current policy.

Parameters

- **obs_batch** *(np.ndarray)* – batch of observations
- **state_batches** *(list)* – list of RNN state input batches, if any
- **is_training** *(bool)* – whether we are training the policy

Returns

- **batch of output actions, with shape like** *[BATCH_SIZE, ACTION SHAPE]*.
- **state_outs** *(list)*: list of RNN state output batches, if any, with shape *[STATE_SIZE, BATCH_SIZE]*.
- **info** *(dict)*: dictionary of extra feature batches, if any, with shape *["f1": [BATCH_SIZE, ...], "f2": [BATCH_SIZE, ...]]*.

Return type

*actions* *(np.ndarray)*

**compute_gradients** *(postprocessed_batch)*
Computes gradients against a batch of experiences.

Returns

- **List of gradient output values info** *(dict)*: Extra policy-specific values

Return type

*grads* *(list)*

**apply_gradients** *(gradients)*
Applies previously computed gradients.

Returns

- **Extra policy-specific values info** *(dict)*

Return type

*info* *(dict)*

**compute_apply** *(postprocessed_batch)*
Fused compute gradients and apply gradients call.

Returns

- **dictionary of extra metadata from compute_gradients(). apply_info** : dictionary of extra metadata from apply_gradients().

Return type

*grad_info*
get_weights()
    Returns model weights.
    Returns  Serializable copy or view of model weights
    Return type  weights (obj)

set_weights(weights)
    Sets model weights.
    Parameters weights (obj) – Serializable copy or view of model weights

class ray.rllib.evaluation.TorchPolicyGraph(observationspace, actionspace, model, loss, loss_inputs)
    Template for a PyTorch policy and loss to use with RLlib.
    This is similar to TFPolicyGraph, but for PyTorch.

    observation_space
        gym.Space – observation space of the policy.

    action_space
        gym.Space – action space of the policy.

    lock
        Lock – Lock that must be held around PyTorch ops on this graph. This is necessary when using the async
        sampler.

    extra_action_out(model_out)
        Returns dict of extra info to include in experience batch.
        Parameters model_out (list) – Outputs of the policy model module.

    optimizer()
        Custom PyTorch optimizer to use.

    compute_actions(obs_batch, state_batches=None, is_training=False)
        Compute actions for the current policy.
        Parameters
        • obs_batch (np.ndarray) – batch of observations
        • state_batches (list) – list of RNN state input batches, if any
        • is_training (bool) – whether we are training the policy
        Returns
        batch of output actions, with shape like [BATCH_SIZE, ACTION_SHAPE].
        state_outs (list): list of RNN state output batches, if any, with
            shape like [STATE_SIZE, BATCH_SIZE].
        info (dict): dictionary of extra feature batches, if any, with
            shape like {“f1”: [BATCH_SIZE, ...], “f2”: [BATCH_SIZE, ...]}.
        Return type  actions (np.ndarray)

    compute_gradients(postprocessed_batch)
        Computes gradients against a batch of experiences.
        Returns  List of gradient output values info (dict): Extra policy-specific values
        Return type  grads (list)
apply_gradients (gradients)
Applies previously computed gradients.

Returns Extra policy-specific values

Return type info (dict)

get_weights ()
Returns model weights.

Returns Serializable copy or view of model weights

Return type weights (obj)

set_weights (weights)
Sets model weights.

Parameters weights (obj) – Serializable copy or view of model weights

class ray.rllib.evaluation.SampleBatch (*args, **kwargs)
Wrapper around a dictionary with string keys and array-like values.

For example, {"obs": [1, 2, 3], "reward": [0, -1, 1]} is a batch of three samples, each with an “obs” and “reward” attribute.

concat (other)
Returns a new SampleBatch with each data column concatenated.

Examples

```python
>>> b1 = SampleBatch({"a": [1, 2]})
>>> b2 = SampleBatch({"a": [3, 4, 5]})
>>> print(b1.concat(b2))
{"a": [1, 2, 3, 4, 5]}
```

rows ()
Returns an iterator over data rows, i.e. dicts with column values.

Examples

```python
>>> batch = SampleBatch({"a": [1, 2, 3], "b": [4, 5, 6]})
>>> for row in batch.rows():
...     print(row)
  {"a": 1, "b": 4}
  {"a": 2, "b": 5}
  {"a": 3, "b": 6}
```

columns (keys)
Returns a list of just the specified columns.

Examples

```python
>>> batch = SampleBatch({"a": [1], "b": [2], "c": [3]})
>>> print(batch.columns(["a", "b"]))
[[1], [2]]
```
class ray.rllib.evaluation.MultiAgentBatch (policy_batches, count)
A batch of experiences from multiple policies in the environment.

policy_batches
dict – Mapping from policy id to a normal SampleBatch of experiences. Note that these batches may be of
different length.

count
int – The number of timesteps in the environment this batch contains. This will be less than the number of
transitions this batch contains across all policies in total.

class ray.rllib.evaluation.SampleBatchBuilder
Util to build a SampleBatch incrementally.

For efficiency, SampleBatches hold values in column form (as arrays). However, it is useful to add data one row
(dict) at a time.

add_values (**values)
Add the given dictionary (row) of values to this batch.

add_batch (batch)
Add the given batch of values to this batch.

build_and_reset ()
Returns a sample batch including all previously added values.

class ray.rllib.evaluation.MultiAgentSampleBatchBuilder (policy_map)
Util to build SampleBatches for each policy in a multi-agent env.

Input data is per-agent, while output data is per-policy. There is an M:N mapping between agents and policies.
We retain one local batch builder per agent. When an agent is done, then its local batch is appended into the
corresponding policy batch for the agent’s policy.

has_pending_data ()
Returns whether there is pending unprocessed data.

add_values (agent_id, policy_id, **values)
Add the given dictionary (row) of values to this batch.

Parameters

- agent_id (obj) – Unique id for the agent we are adding values for.
- policy_id (obj) – Unique id for policy controlling the agent.
- values (dict) – Row of values to add for this agent.

postprocess_batch_so_far ()
Apply policy postprocessors to any unprocessed rows.

This pushes the postprocessed per-agent batches onto the per-policy builders, clearing per-agent state.

build_and_reset ()
Returns the accumulated sample batches for each policy.

Any unprocessed rows will be first postprocessed with a policy postprocessor. The internal state of this
builder will be reset.

class ray.rllib.evaluation.SyncSampler (env, policies, policy_mapping_fn, obs_filters,
num_local_steps, horizon=None, pack=False, tf_sess=None)

This class interacts with the environment and tells it what to do.

Note that batch_size is only a unit of measure here. Batches can accumulate and the gradient can be calculated
on up to 5 batches.
This class provides data on invocation, rather than on a separate thread.

```python
class ray.rllib.evaluation.AsyncSampler(env, policies, policy_mapping_fn, obs_filters, num_local_steps, horizon=None, pack=False, tf_sess=None)
```

This class interacts with the environment and tells it what to do.

Note that batch_size is only a unit of measure here. Batches can accumulate and the gradient can be calculated on up to 5 batches.

```python
run()
```

Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

```python
ray.rllib.evaluation.compute_advantages(rollout, last_r, gamma=0.9, lambda_=1.0, use_gae=True)
```

Given a rollout, compute its value targets and the advantage.

**Parameters**

- `rollout (SampleBatch)` – SampleBatch of a single trajectory
- `last_r (float)` – Value estimation for last observation
- `gamma (float)` – Discount factor.
- `lambda_ (float)` – Parameter for GAE
- `use_gae (bool)` – Using Generalized Advantage Estimation

**Returns**

Object with experience from rollout and processed rewards.

**Return type** `SampleBatch (SampleBatch)`

```python
ray.rllib.evaluation.compute_targets(rollout, action_space, last_r=0.0, gamma=0.9, lambda_=1.0)
```

Given a rollout, compute targets.

Used for categorical crossentropy loss on the policy. Also assumes there is a value function. Uses GAE to calculate advantages.

**Parameters**

- `rollout (SampleBatch)` – SampleBatch of a single trajectory
- `action_space (gym.Space)` – Dimensions of the advantage targets.
- `last_r (float)` – Value estimation for last observation
- `gamma (float)` – Discount factor.
- `lambda_ (float)` – Parameter for GAE

```python
ray.rllib.evaluation.collect_metrics(local_evaluator, remote_evaluators=[])  
```

Gathers episode metrics from PolicyEvaluator instances.

### 1.19.4 ray.rllib.models

```python
class ray.rllib.models.ActionDistribution(inputs)
```

The policy action distribution of an agent.
Parameters `inputs` *(Tensor)* – The input vector to compute samples from.

- `logp(x)`
  The log-likelihood of the action distribution.

- `kl(other)`
  The KL-divergence between two action distributions.

- `entropy()`
  The entropy of the action distribution.

- `sample()`
  Draw a sample from the action distribution.

**class** `ray.rllib.models.Categorical(inputs)`
Categorical distribution for discrete action spaces.

- `logp(x)`
  The log-likelihood of the action distribution.

- `entropy()`
  The entropy of the action distribution.

- `kl(other)`
  The KL-divergence between two action distributions.

- `sample()`
  Draw a sample from the action distribution.

**class** `ray.rllib.models.DiagGaussian(inputs, low=None, high=None)`
Action distribution where each vector element is a gaussian.

- The first half of the input vector defines the gaussian means, and the second half the gaussian standard deviations.

- `logp(x)`
  The log-likelihood of the action distribution.

- `kl(other)`
  The KL-divergence between two action distributions.

- `entropy()`
  The entropy of the action distribution.

- `sample()`
  Draw a sample from the action distribution.

**class** `ray.rllib.models.Deterministic(inputs)`
Action distribution that returns the input values directly.

- This is similar to DiagGaussian with standard deviation zero.

- `sample()`
  Draw a sample from the action distribution.

**class** `ray.rllib.models.ModelCatalog`
Registry of models, preprocessors, and action distributions for envs.

**Examples**

```python
>>> prep = ModelCatalog.get_preprocessor(env)
>>> observation = prep.transform(raw_observation)
```
dist_cls, dist_dim = ModelCatalog.get_action_dist(env.action_space)
model = ModelCatalog.get_model(inputs, dist_dim)
dist = dist_cls(model.outputs)
action = dist.sample()

static get_action_dist (action_space, config=None, dist_type=None)
Returns action distribution class and size for the given action space.
Parameters
- action_space (Space) – Action space of the target gym env.
- config (dict) – Optional model config.
- dist_type (str) – Optional identifier of the action distribution.
Returns Python class of the distribution. dist_dim (int): The size of the input vector to the
distribution.
Return type dist_class (ActionDistribution)

static get_action_placeholder (action_space)
Returns an action placeholder that is consistent with the action space
Parameters action_space (Space) – Action space of the target gym env.
Returns A placeholder for the actions
Return type action_placeholder (Tensor)

static get_model (inputs, num_outputs, options=None, state_in=None, seq_lens=None)
Returns a suitable model conforming to given input and output specs.
Parameters
- inputs (Tensor) – The input tensor to the model.
- num_outputs (int) – The size of the output vector of the model.
- options (dict) – Optional args to pass to the model constructor.
- state_in (list) – Optional RNN state in tensors.
- seq_in (Tensor) – Optional RNN sequence length tensor.
Returns Neural network model.
Return type model (Model)

static get_torch_model (input_shape, num_outputs, options={})
Returns a PyTorch suitable model. This is currently only supported in A3C.
Parameters
- input_shape (tuple) – The input shape to the model.
- num_outputs (int) – The size of the output vector of the model.
- options (dict) – Optional args to pass to the model constructor.
Returns Neural network model.
Return type model (Model)

static get_preprocessor (env, options={})
Returns a suitable processor for the given environment.
Parameters
Ray Documentation, Release 0.5.0

- **env** (*gym.Env*) – The gym environment to preprocess.
- **options** (*dict*) – Options to pass to the preprocessor.

**Returns** Preprocessor for the env observations.

**Return type** preprocessor (*Preprocessor*)

**static get_preprocessor_as_wrapper** (*env, options={}*)

Returns a preprocessor as a gym observation wrapper.

**Parameters**
- **env** (*gym.Env*) – The gym environment to wrap.
- **options** (*dict*) – Options to pass to the preprocessor.

**Returns** Preprocessor in wrapper form.

**Return type** wrapper (*gym.ObservationWrapper*)

**static register_custom_preprocessor** (*preprocessor_name, preprocessor_class*)

Register a custom preprocessor class by name.

The preprocessor can be later used by specifying `{“custom_preprocessor”: preprocesor_name}` in the model config.

**Parameters**
- **preprocessor_name** (*str*) – Name to register the preprocessor under.
- **preprocessor_class** (*type*) – Python class of the preprocessor.

**static register_custom_model** (*model_name, model_class*)

Register a custom model class by name.

The model can be later used by specifying `{“custom_model”: model_name}` in the model config.

**Parameters**
- **model_name** (*str*) – Name to register the model under.
- **model_class** (*type*) – Python class of the model.

**class ray.rllib.models.Model** (*inputs, num_outputs, options, state_in=None, seq_lens=None*)

Defines an abstract network model for use with RLib.

Models convert input tensors to a number of output features. These features can then be interpreted by ActionDistribution classes to determine e.g. agent action values.

The last layer of the network can also be retrieved if the algorithm needs to further post-processing (e.g. Actor and Critic networks in A3C).

**inputs**
- **Tensor** – The input placeholder for this model, of shape [BATCH_SIZE, ...].

**outputs**
- **Tensor** – The output vector of this model, of shape [BATCH_SIZE, num_outputs].

**last_layer**
- **Tensor** – The network layer right before the model output, of shape [BATCH_SIZE, N].

**state_init**
- **list** – List of initial recurrent state tensors (if any).

**state_in**
- **list** – List of input recurrent state tensors (if any).
state_out
   list – List of output recurrent state tensors (if any).

seq_lens
   Tensor – The tensor input for RNN sequence lengths. This defaults to a Tensor of [1] * len(batch) in the
   non-RNN case.

If options[“free_log_std”] is True, the last half of the output layer will be free variables that are not dependent
on inputs. This is often used if the output of the network is used to parametrize a probability distribution. In this
case, the first half of the parameters can be interpreted as a location parameter (like a mean) and the second half
can be interpreted as a scale parameter (like a standard deviation).

class ray.rllib.models.Preprocessor(obs_space, options)
   Defines an abstract observation preprocessor function.

   shape
       obj – Shape of the preprocessed output.

   transform(observation)
       Returns the preprocessed observation.

class ray.rllib.models.FullyConnectedNetwork(inputs, num_outputs, options, state_in=None, seq_lens=None)
   Generic fully connected network.

class ray.rllib.models.LSTM(inputs, num_outputs, options, state_in=None, seq_lens=None)
   Adds a LSTM cell on top of some other model output.

   Uses a linear layer at the end for output.

   Important: we assume inputs is a padded batch of sequences denoted by self.seq_lens. See
   add_time_dimension() for more information.

1.19.5 ray.rllib.optimizers

class ray.rllib.optimizers.PolicyOptimizer(local_evaluator, remote_evaluators=None, config=None)
   Policy optimizers encapsulate distributed RL optimization strategies.

   Policy optimizers serve as the “control plane” of algorithms.

   For example, AsyncOptimizer is used for A3C, and LocalMultiGPUOptimizer is used for PPO. These optimizers
   are all pluggable, and it is possible to mix and match as needed.

   In order for an algorithm to use an RLlib optimizer, it must implement the PolicyEvaluator interface and pass
   a PolicyEvaluator class or set of PolicyEvaluators to its PolicyOptimizer of choice. The PolicyOptimizer uses
   these Evaluators to sample from the environment and compute model gradient updates.

   config
       dict – The JSON configuration passed to this optimizer.

   local_evaluator
       PolicyEvaluator – The embedded evaluator instance.

   remote_evaluators
       list – List of remote evaluator replicas, or [].

   num_steps_trained
       int – Number of timesteps trained on so far.

   num_steps_sampled
       int – Number of timesteps sampled so far.
evaluator_resources
   dict – Optional resource requests to set for evaluators created by this optimizer.

step()
   Takes a logical optimization step.
   This should run for long enough to minimize call overheads (i.e., at least a couple seconds), but short
   enough to return control periodically to callers (i.e., at most a few tens of seconds).
   Returns Optional fetches from compute grads calls.
   Return type fetches (dict|None)

stats()
   Returns a dictionary of internal performance statistics.

collect_metrics()
   Returns evaluator and optimizer stats.
   Returns
   TrainingResult from evaluator metrics with info replaced with stats from self.
   Return type res (TrainingResult)

save()
   Returns a serializable object representing the optimizer state.

restore(data)
   Restores optimizer state from the given data object.

foreach_evaluator(func)
   Apply the given function to each evaluator instance.

foreach_evaluator_with_index(func)
   Apply the given function to each evaluator instance.
   The index will be passed as the second arg to the given function.

classmethod make(env_creator, policy_graph, optimizer_batch_size=None, num_workers=0, num_envs_per_worker=None, optimizer_config=None, remote_num_cpus=None, remote_num_gpus=None, **eval_kwargs)
   Creates an Optimizer with local and remote evaluators.

   Parameters
   • env_creator(func) – Function that returns a gym.Env given an EnvContext wrapped
     configuration.
   • policy_graph(class|dict) – Either a class implementing PolicyGraph, or a diction-
     ary of policy id strings to (PolicyGraph, obs_space, action_space, config) tuples. See
     PolicyEvaluator documentation.
   • optimizer_batch_size(int) – Batch size summed across all workers. Will over-
     ride worker batch_steps.
   • num_workers(int) – Number of remote evaluators
   • num_envs_per_worker(int) – (Optional) Sets the number environments per evalu-
     ator for vectorization. If set, overrides num_envs in kwargs for PolicyEvaluator.__init__.
   • optimizer_config(dict) – Config passed to the optimizer.
   • remote_num_cpus(int) – CPU specification for remote evaluator.
   • remote_num_gpus(int) – GPU specification for remote evaluator.
• **eval_kwargs** – PolicyEvaluator Class non-positional args.

Returns

(Optimizer) Instance of cls with evaluators configured accordingly.

class ray.rllib.optimizers.AsyncReplayOptimizer(local_evaluator, remote_evaluators=None, config=None)

Main event loop of the Ape-X optimizer (async sampling with replay).

This class coordinates the data transfers between the learner thread, remote evaluators (Ape-X actors), and replay buffer actors.

This optimizer requires that policy evaluators return an additional “td_error” array in the info return of compute_gradients(). This error term will be used for sample prioritization.

step()

Takes a logical optimization step.

This should run for long enough to minimize call overheads (i.e., at least a couple seconds), but short enough to return control periodically to callers (i.e., at most a few tens of seconds).

Returns Optional fetches from compute grads calls.

Return type fetches (dict|None)

stats()

Returns a dictionary of internal performance statistics.

class ray.rllib.optimizers.AsyncSamplesOptimizer(local_evaluator, remote_evaluators=None, config=None)

Main event loop of the IMPALA architecture.

This class coordinates the data transfers between the learner thread and remote evaluators (IMPALA actors).

step()

Takes a logical optimization step.

This should run for long enough to minimize call overheads (i.e., at least a couple seconds), but short enough to return control periodically to callers (i.e., at most a few tens of seconds).

Returns Optional fetches from compute grads calls.

Return type fetches (dict|None)

stats()

Returns a dictionary of internal performance statistics.

class ray.rllib.optimizers.AsyncGradientsOptimizer(local_evaluator, remote_evaluators=None, config=None)

An asynchronous RL optimizer, e.g. for implementing A3C.

This optimizer asynchronously pulls and applies gradients from remote evaluators, sending updated weights back as needed. This pipelines the gradient computations on the remote workers.

step()

Takes a logical optimization step.

This should run for long enough to minimize call overheads (i.e., at least a couple seconds), but short enough to return control periodically to callers (i.e., at most a few tens of seconds).

Returns Optional fetches from compute grads calls.
**Return type** fetches (dict|None)

**stats()**

Returns a dictionary of internal performance statistics.

class ray.rllib.optimizers.SyncSamplesOptimizer (local_evaluator, remote_evaluators=None, config=None)

A simple synchronous RL optimizer.

In each step, this optimizer pulls samples from a number of remote evaluators, concatenates them, and then updates a local model. The updated model weights are then broadcast to all remote evaluators.

**step()**

Takes a logical optimization step.

This should run for long enough to minimize call overheads (i.e., at least a couple seconds), but short enough to return control periodically to callers (i.e., at most a few tens of seconds).

**Returns** Optional fetches from compute grads calls.

**Return type** fetches (dict|None)

**stats()**

Returns a dictionary of internal performance statistics.

class ray.rllib.optimizers.SyncReplayOptimizer (local_evaluator, remote_evaluators=None, config=None)

Variant of the local sync optimizer that supports replay (for DQN).

This optimizer requires that policy evaluators return an additional “td_error” array in the info return of compute_gradients(). This error term will be used for sample prioritization.

**step()**

Takes a logical optimization step.

This should run for long enough to minimize call overheads (i.e., at least a couple seconds), but short enough to return control periodically to callers (i.e., at most a few tens of seconds).

**Returns** Optional fetches from compute grads calls.

**Return type** fetches (dict|None)

**stats()**

Returns a dictionary of internal performance statistics.

class ray.rllib.optimizers.LocalMultiGPUOptimizer (local_evaluator, remote_evaluators=None, config=None)

A synchronous optimizer that uses multiple local GPUs.

Samples are pulled synchronously from multiple remote evaluators, concatenated, and then split across the memory of multiple local GPUs. A number of SGD passes are then taken over the in-memory data. For more details, see multi_gpu_impl.LocalSyncParallelOptimizer.

This optimizer is Tensorflow-specific and require the underlying PolicyGraph to be a TFPolicyGraph instance that support .copy().

Note that all replicas of the TFPolicyGraph will merge their extra_compute_grad and apply_grad feed_dicts and fetches. This may result in unexpected behavior.

**step()**

Takes a logical optimization step.
This should run for long enough to minimize call overheads (i.e., at least a couple seconds), but short enough to return control periodically to callers (i.e., at most a few tens of seconds).

**Returns** Optional fetches from compute grads calls.

**Return type** fetches (dict|None)

**stats()**
Returns a dictionary of internal performance statistics.

### 1.19.6 ray.rllib.utils

class ray.rllib.utils.Filter
Processes input, possibly statefully.

**apply_changes**(other, *args, **kwargs)**
Updates self with “new state” from other filter.

**copy()**
Creates a new object with same state as self.

**Returns** Copy of self

**Return type** copy (Filter)

**sync**(other)
Copies all state from other filter to self.

**clear_buffer()**
Creates copy of current state and clears accumulated state

class ray.rllib.utils.FilterManager
Manages filters and coordination across remote evaluators that expose get_filters and sync_filters.

**static synchronize**(local_filters, remotes)
Aggregates all filters from remote evaluators. Local copy is updated and then broadcasted to all remote evaluators.

**Parameters**
- **local_filters**(dict) – Filters to be synchronized.
- **remotes**(list) – Remote evaluators with filters.

class ray.rllib.utils.PolicyClient(address)
REST client to interact with a RLlib policy server.

**start_episode**(episode_id=None, training_enabled=True)
Record the start of an episode.

**Parameters**
- **episode_id**(str) – Unique string id for the episode or None for it to be auto-assigned.
- **training_enabled**(bool) – Whether to use experiences for this episode to improve the policy.

**Returns** Unique string id for the episode.

**Return type** episode_id (str)

**get_action**(episode_id, observation)
Record an observation and get the on-policy action.
Parameters

- **episode_id** *(str)* – Episode id returned from start_episode().
- **observation** *(obj)* – Current environment observation.

Returns  Action from the env action space.

Return type  action *(obj)*

log_action *(episode_id, observation, action)*

Record an observation and (off-policy) action taken.

Parameters

- **episode_id** *(str)* – Episode id returned from start_episode().
- **observation** *(obj)* – Current environment observation.
- **action** *(obj)* – Action for the observation.

log_returns *(episode_id, reward, info=None)*

Record returns from the environment.

The reward will be attributed to the previous action taken by the episode. Rewards accumulate until the next action. If no reward is logged before the next action, a reward of 0.0 is assumed.

Parameters

- **episode_id** *(str)* – Episode id returned from start_episode().
- **reward** *(float)* – Reward from the environment.

end_episode *(episode_id, observation)*

Record the end of an episode.

Parameters

- **episode_id** *(str)* – Episode id returned from start_episode().
- **observation** *(obj)* – Current environment observation.

class  ray.rllib.utils.PolicyServer *(serving_env, address, port)*

REST server than can be launched from a ServingEnv.

This launches a multi-threaded server that listens on the specified host and port to serve policy requests and forward experiences to RLlib.

Examples

```
>>> class CartpoleServing(ServingEnv):
    def __init__(self):
        ServingEnv.__init__(
            self, spaces.Discrete(2),
            spaces.Box(  
                low=-10,
                high=10,
                shape=(4,),
                dtype=np.float32))
    def run(self):
        server = PolicyServer(self, "localhost", 8900)
        server.serve_forever()
>>> register_env("srv", lambda _: CartpoleServing())
```
>>> pg = PGAgent(env="srv", config={"num_workers": 0})
>>> while True:
    pg.train()

>>> client = PolicyClient("localhost:8900")
>>> eps_id = client.start_episode()

>>> action = client.get_action(eps_id, obs)

>>> ... 

>>> client.log_returns(eps_id, reward)

>>> ...

>>> client.log_returns(eps_id, reward)

1.20 Pandas on Ray

Pandas on Ray has moved to Modin!

Pandas on Ray has moved into the Modin project with the intention of unifying the DataFrame APIs.

1.21 Learning to Play Pong

In this example, we’ll train a very simple neural network to play Pong using the OpenAI Gym. This application is adapted, with minimal modifications, from Andrej Karpathy’s code (see the accompanying blog post).

You can view the code for this example.

To run the application, first install some dependencies.

pip install gym[atari]

Then you can run the example as follows.

python ray/examples/rl_pong/driver.py --batch-size=10

To run the example on a cluster, simply pass in the flag --redis-address=<redis-address>.

At the moment, on a large machine with 64 physical cores, computing an update with a batch of size 1 takes about 1 second, a batch of size 10 takes about 2.5 seconds. A batch of size 60 takes about 3 seconds. On a cluster with 11 nodes, each with 18 physical cores, a batch of size 300 takes about 10 seconds. If the numbers you see differ from these by much, take a look at the Troubleshooting section at the bottom of this page and consider submitting an issue.

Note that these times depend on how long the rollouts take, which in turn depends on how well the policy is doing. For example, a really bad policy will lose very quickly. As the policy learns, we should expect these numbers to increase.

1.21.1 The distributed version

At the core of Andrej’s code, a neural network is used to define a “policy” for playing Pong (that is, a function that chooses an action given a state). In the loop, the network repeatedly plays games of Pong and records a gradient from each game. Every ten games, the gradients are combined together and used to update the network.

This example is easy to parallelize because the network can play ten games in parallel and no information needs to be shared between the games.
We define an actor for the Pong environment, which includes a method for performing a rollout and computing a gradient update. Below is pseudocode for the actor.

```python
@ray.remote
class PongEnv(object):
    def __init__(self):
        # Tell numpy to only use one core. If we don't do this, each actor may try
to use all of the cores and the resulting contention may result in no
# speedup over the serial version. Note that if numpy is using OpenBLAS,
# then you need to set OPENBLAS_NUM_THREADS=1, and you probably need to do
# it from the command line (so it happens before numpy is imported).
        os.environ["MKL_NUM_THREADS"] = "1"
        self.env = gym.make("Pong-v0")

    def compute_gradient(self, model):
        # Reset the game.
        observation = self.env.reset()
        while not done:
            # Choose an action using policy_forward.
            # Take the action and observe the new state of the world.
            # Compute a gradient using policy_backward. Return the gradient and reward.
            return [gradient, reward_sum]
```

We then create a number of actors, so that we can perform rollouts in parallel.

```python
actors = [PongEnv() for _ in range(batch_size)]
```

Calling this remote function inside of a for loop, we launch multiple tasks to perform rollouts and compute gradients in parallel.

```python
model_id = ray.put(model)
actions = []
# Launch tasks to compute gradients from multiple rollouts in parallel.
for i in range(batch_size):
    action_id = actors[i].compute_gradient.remote(model_id)
    actions.append(action_id)
```

### 1.21.2 Troubleshooting

If you are not seeing any speedup from Ray (and assuming you’re using a multicore machine), the problem may be that numpy is trying to use multiple threads. When many processes are each trying to use multiple threads, the result is often no speedup. When running this example, try opening up `top` and seeing if some python processes are using more than 100% CPU. If yes, then this is likely the problem.

The example tries to set `MKL_NUM_THREADS=1` in the actor. However, that only works if the numpy on your machine is actually using MKL. If it’s using OpenBLAS, then you’ll need to set `OPENBLAS_NUM_THREADS=1`. In fact, you may have to do this before running the script (it may need to happen before numpy is imported).

```bash
export OPENBLAS_NUM_THREADS=1
```

### 1.22 Policy Gradient Methods

This code shows how to do reinforcement learning with policy gradient methods. View the code for this example.
Note: For an overview of Ray’s reinforcement learning library, see Ray RLlib.

To run this example, you will need to install TensorFlow with GPU support (at least version 1.0.0) and a few other dependencies.

```bash
pip install gym[atari]
pip install tensorflow
```

Then you can run the example as follows.

```bash
python/ray/rllib/train.py --env=Pong-ram-v4 --run=PPO
```

This will train an agent on the Pong-ram-v4 Atari environment. You can also try passing in the Pong-v0 environment or the CartPole-v0 environment. If you wish to use a different environment, you will need to change a few lines in example.py.

Current and historical training progress can be monitored by pointing TensorBoard to the log output directory as follows.

```bash
tensorboard --logdir=~/ray_results
```

Many of the TensorBoard metrics are also printed to the console, but you might find it easier to visualize and compare between runs using the TensorBoard UI.

## 1.23 Parameter Server

This document walks through how to implement simple synchronous and asynchronous parameter servers using actors.

To run the application, first install some dependencies.

```bash
pip install tensorflow
```

You can view the code for this example.

The examples can be run as follows.

```bash
# Run the asynchronous parameter server.
python ray/examples/parameter_server/async_parameter_server.py --num-workers=4

# Run the synchronous parameter server.
python ray/examples/parameter_server/sync_parameter_server.py --num-workers=4
```

Note that this examples uses distributed actor handles, which are still considered experimental.

### 1.23.1 Asynchronous Parameter Server

The asynchronous parameter server itself is implemented as an actor, which exposes the methods `push` and `pull`.

```python
@ray.remote
class ParameterServer(object):
    def __init__(self, keys, values):
        values = [value.copy() for value in values]
        self.weights = dict(zip(keys, values))
```
def push(self, keys, values):
    for key, value in zip(keys, values):
        self.weights[key] += value

def pull(self, keys):
    return [self.weights[key] for key in keys]

We then define a worker task, which take a parameter server as an argument and submits tasks to it. The structure of the code looks as follows.

```python
@ray.remote
def worker_task(ps):
    while True:
        # Get the latest weights from the parameter server.
        weights = ray.get(ps.pull.remote(keys))

        # Compute an update.
        ...

        # Push the update to the parameter server.
        ps.push.remote(keys, update)
```

Then we can create a parameter server and initiate training as follows.

```python
ps = ParameterServer.remote(keys, initial_values)
worker_tasks = [worker_task.remote(ps) for _ in range(4)]
```

### 1.23.2 Synchronous Parameter Server

The parameter server is implemented as an actor, which exposes the methods `apply_gradients` and `get_weights`. A constant linear scaling rule is applied by scaling the learning rate by the number of workers.

```python
@ray.remote
class ParameterServer(object):
    def __init__(self, learning_rate):
        self.net = model.SimpleCNN(learning_rate=learning_rate)

    def apply_gradients(self, *gradients):
        self.net.apply_gradients(np.mean(gradients, axis=0))
        return self.net.variables.get_flat()

    def get_weights(self):
        return self.net.variables.get_flat()
```

Workers are actors which expose the method `compute_gradients`.

```python
@ray.remote
class Worker(object):
    def __init__(self, worker_index, batch_size=50):
        self.worker_index = worker_index
        self.batch_size = batch_size
        self.mnist = input_data.read_data_sets("MNIST_data", one_hot=True, seed=worker_index)
        self.net = model.SimpleCNN()
```
def compute_gradients(self, weights):
    self.net.variables.set_flat(weights)
    xs, ys = self.mnist.train.next_batch(self.batch_size)
    return self.net.compute_gradients(xs, ys)

Training alternates between computing the gradients given the current weights from the parameter server and updating the parameter server's weights with the resulting gradients.

while True:
    gradients = [worker.compute_gradients.remote(current_weights)
                 for worker in workers]
    current_weights = ps.apply_gradients.remote(*gradients)

Both of these examples implement the parameter server using a single actor, however they can be easily extended to shard the parameters across multiple actors.

1.24 ResNet

This code adapts the TensorFlow ResNet example to do data parallel training across multiple GPUs using Ray. View the code for this example.

To run the example, you will need to install TensorFlow (at least version 1.0.0). Then you can run the example as follows.

First download the CIFAR-10 or CIFAR-100 dataset.

# Get the CIFAR-10 dataset.
curl -o cifar-10-binary.tar.gz https://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz
tar -xvf cifar-10-binary.tar.gz

# Get the CIFAR-100 dataset.
curl -o cifar-100-binary.tar.gz https://www.cs.toronto.edu/~kriz/cifar-100-binary.tar.gz
tar -xvf cifar-100-binary.tar.gz

Then run the training script that matches the dataset you downloaded.

# Train Resnet on CIFAR-10.
python ray/examples/resnet/resnet_main.py \
   --eval_dir=/tmp/resnet-model/eval \
   --train_data_path=cifar-10-batches-bin/data_batch* \
   --eval_data_path=cifar-10-batches-bin/test_batch.bin \
   --dataset=cifar10 \
   --num_gpus=1

# Train Resnet on CIFAR-100.
python ray/examples/resnet/resnet_main.py \
   --eval_dir=/tmp/resnet-model/eval \
   --train_data_path=cifar-100-binary/train.bin \
   --eval_data_path=cifar-100-binary/test.bin \
   --dataset=cifar100 \
   --num_gpus=1

To run the training script on a cluster with multiple machines, you will need to also pass in the flag --redis-address=<redis_address>, where <redis-address> is the address of the Redis server on the
head node.

The script will print out the IP address that the log files are stored on. In the single-node case, you can ignore this and run tensorboard on the current machine.

```bash
python -m tensorflow.tensorboard --logdir=/tmp/resnet-model
```

If you are running Ray on multiple nodes, you will need to go to the node at the IP address printed, and run the command.

The core of the script is the actor definition.

```python
@ray.remote(num_gpus=1)
class ResNetTrainActor(object):
    def __init__(self, data, dataset, num_gpus):
        # data is the preprocessed images and labels extracted from the dataset.
        # Thus, every actor has its own copy of the data.
        # Set the CUDA_VISIBLE_DEVICES environment variable in order to restrict
        # which GPUs TensorFlow uses. Note that this only works if it is 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'):
                # We omit the code here that actually constructs the residual network
                # and initializes it. Uses the definition in the Tensorflow Resnet
                # Example.
                def compute_steps(self, weights):
                    # This method sets the weights in the network, runs some training steps,
                    # and returns the new weights. self.model.variables is a TensorFlowVariables
                    # class that we pass the train operation into.
                    self.model.variables.set_weights(weights)
                    for i in range(self.steps):
                        self.model.variables.sess.run(self.model.train_op)
                    return self.model.variables.get_weights()

The main script first creates one actor for each GPU, or a single actor if num_gpus is zero.

```python
train_actors = [ResNetTrainActor.remote(train_data, dataset, num_gpus) for _ in range(num_gpus)]
```

Then the main loop passes the same weights to every model, performs updates on each model, averages the updates, and puts the new weights in the object store.

```python
while True:
    all_weights = ray.get([actor.compute_steps.remote(weight_id) for actor in train_actors])
    mean_weights = {k: sum([weights[k] for weights in all_weights]) / num_gpus for k in all_weights[0]}
    weight_id = ray.put(mean_weights)
```

1.25 Asynchronous Advantage Actor Critic (A3C)

This document walks through A3C, a state-of-the-art reinforcement learning algorithm. In this example, we adapt the OpenAI Universe Starter Agent implementation of A3C to use Ray.
**View the code for this example.**

**Note:** For an overview of Ray’s reinforcement learning library, see Ray RLlib.

To run the application, first install `ray` and then some dependencies:

```bash
pip install tensorflow
pip install six
pip install gym[atari]
pip install opencv-python
pip install scipy
```

You can run the code with

```bash
python/ray/rllib/train.py --env=Pong-ram-v4 --run=A3C --config='{"num_workers": N}'
```

## 1.25.1 Reinforcement Learning

Reinforcement Learning is an area of machine learning concerned with **learning how an agent should act in an environment** so as to maximize some form of cumulative reward. Typically, an agent will observe the current state of the environment and take an action based on its observation. The action will change the state of the environment and will provide some numerical reward (or penalty) to the agent. The agent will then take in another observation and the process will repeat. **The mapping from state to action is a policy**, and in reinforcement learning, this policy is often represented with a deep neural network.

The **environment** is often a simulator (for example, a physics engine), and reinforcement learning algorithms often involve trying out many different sequences of actions within these simulators. These **rollouts** can often be done in parallel.

Policies are often initialized randomly and incrementally improved via simulation within the environment. To improve a policy, gradient-based updates may be computed based on the sequences of states and actions that have been observed. The gradient calculation is often delayed until a termination condition is reached (that is, the simulation has finished) so that delayed rewards have been properly accounted for. However, in the Actor Critic model, we can begin the gradient calculation at any point in the simulation rollout by predicting future rewards with a Value Function approximator.

In our A3C implementation, each worker, implemented as a Ray actor, continuously simulates the environment. The driver will create a task that runs some steps of the simulator using the latest model, computes a gradient update, and returns the update to the driver. Whenever a task finishes, the driver will use the gradient update to update the model and will launch a new task with the latest model.

There are two main parts to the implementation - the driver and the worker.

### 1.25.2 Worker Code Walkthrough

We use a Ray Actor to simulate the environment.

```python
import numpy as np
import ray

@ray.remote
class Runner(object):
    """Actor object to start running simulation on workers. Gradient computation is also executed on this object."""
```

(continues on next page)
def __init__(self, env_name, actor_id):
    # starts simulation environment, policy, and thread.
    # Thread will continuously interact with the simulation environment
    self.env = env = create_env(env_name)
    self.id = actor_id
    self.policy = LSTMPolicy()
    self.runner = RunnerThread(env, self.policy, 20)
    self.start()

def start(self):
    # starts the simulation thread
    self.runner.start_runner()

def pull_batch_from_queue(self):
    # Implementation details removed - gets partial rollout from queue
    return rollout

def compute_gradient(self, params):
    self.policy.set_weights(params)
    rollout = self.pull_batch_from_queue()
    batch = process_rollout(rollout, gamma=0.99, lambda_=1.0)
    gradient = self.policy.compute_gradients(batch)
    info = {
        "id": self.id,
        "size": len(batch.a)
    }
    return gradient, info

1.25.3 Driver Code Walkthrough

The driver manages the coordination among workers and handles updating the global model parameters. The main training script looks like the following.

```python
import numpy as np
import ray

def train(num_workers, env_name="PongDeterministic-v4"):
    # Setup a copy of the environment
    env = create_env(env_name, None, None)
    policy = LSTMPolicy(env.observation_space.shape, env.action_space.n, 0)
    obs = 0

    # Start simulations on actors
    agents = [Runner(env_name, i) for i in range(num_workers)]

    # Start gradient calculation tasks on each actor
    parameters = policy.get_weights()
    gradient_list = [agent.compute_gradient.remote(parameters) for agent in agents]

    while True:
        # Replace with your termination condition
        # wait for some gradient to be computed - unblock as soon as the earliest arrives
        done_id, gradient_list = ray.wait(gradient_list)

        # get the results of the task from the object store
        gradient, info = ray.get(done_id)[0]
```

(continues on next page)
obs += info["size"]
    # apply update, get the weights from the model, start a new task on the same actor object
    policy.apply_gradients(gradient)
    parameters = policy.get_weights()
    gradient_list.extend([agents[info["id"]].compute_gradient(parameters)])
    return policy

1.25.4 Benchmarks and Visualization

For the PongDeterministic-v4 and an Amazon EC2 m4.16xlarge instance, we are able to train the agent with 16 workers in around 15 minutes. With 8 workers, we can train the agent in around 25 minutes.

You can visualize performance by running tensorboard --logdir [directory] in a separate screen, where [directory] is defaulted to ~/ray_results/. If you are running multiple experiments, be sure to vary the directory to which Tensorflow saves its progress (found in a3c.py).

1.26 Batch L-BFGS

This document provides a walkthrough of the L-BFGS example. To run the application, first install these dependencies.

```bash
pip install tensorflow
pip install scipy
```

You can view the code for this example.

Then you can run the example as follows.

```bash
python ray/examples/lbfgs/driver.py
```

Optimization is at the heart of many machine learning algorithms. Much of machine learning involves specifying a loss function and finding the parameters that minimize the loss. If we can compute the gradient of the loss function, then we can apply a variety of gradient-based optimization algorithms. L-BFGS is one such algorithm. It is a quasi-Newton method that uses gradient information to approximate the inverse Hessian of the loss function in a computationally efficient manner.

1.26.1 The serial version

First we load the data in batches. Here, each element in batches is a tuple whose first component is a batch of 100 images and whose second component is a batch of the 100 corresponding labels. For simplicity, we use TensorFlow’s built-in methods for loading the data.

```python
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
batch_size = 100
num_batches = mnist.train.num_examples // batch_size
batches = [mnist.train.next_batch(batch_size) for _ in range(num_batches)]
```

Now, suppose we have defined a function which takes a set of model parameters $\theta$ and a batch of data (both images and labels) and computes the loss for that choice of model parameters on that batch of data. Similarly, suppose
we’ve also defined a function that takes the same arguments and computes the gradient of the loss for that choice of model parameters.

```python
def loss(theta, xs, ys):
    # compute the loss on a batch of data
    return loss
def grad(theta, xs, ys):
    # compute the gradient on a batch of data
    return grad
def full_loss(theta):
    # compute the loss on the full data set
    return sum([loss(theta, xs, ys) for (xs, ys) in batches])
def full_grad(theta):
    # compute the gradient on the full data set
    return sum([grad(theta, xs, ys) for (xs, ys) in batches])
```

Since we are working with a small dataset, we don’t actually need to separate these methods into the part that operates on a batch and the part that operates on the full dataset, but doing so will make the distributed version clearer.

Now, if we wish to optimize the loss function using L-BFGS, we simply plug these functions, along with an initial choice of model parameters, into `scipy.optimize.fmin_l_bfgs_b`.

```python
theta_init = 1e-2 * np.random.normal(size=dim)
result = scipy.optimize.fmin_l_bfgs_b(full_loss, theta_init, fprime=full_grad)
```

### 1.26.2 The distributed version

In this example, the computation of the gradient itself can be done in parallel on a number of workers or machines.

First, let’s turn the data into a collection of remote objects.

```python
batch_ids = [(ray.put(xs), ray.put(ys)) for (xs, ys) in batches]
```

We can load the data on the driver and distribute it this way because MNIST easily fits on a single machine. However, for larger data sets, we will need to use remote functions to distribute the loading of the data.

Now, let’s turn `loss` and `grad` into methods of an actor that will contain our network.

```python
class Network(object):
    def __init__(self):
        # Initialize network.
    def loss(theta, xs, ys):
        # compute the loss
        return loss
    def grad(theta, xs, ys):
        # compute the gradient
        return grad
```

Now, it is easy to speed up the computation of the full loss and the full gradient.

```python
def full_loss(theta):
    theta_id = ray.put(theta)
```
loss_ids = [actor.loss(theta_id) for actor in actors]
return sum(ray.get(loss_ids))

def full_grad(theta):
    theta_id = ray.put(theta)
    grad_ids = [actor.grad(theta_id) for actor in actors]
    return sum(ray.get(grad_ids)).astype("float64") # This conversion is necessary.

Note that we turn theta into a remote object with the line theta_id = ray.put(theta) before passing it
into the remote functions. If we had written

[actor.loss(theta_id) for actor in actors]

instead of

theta_id = ray.put(theta)
[actor.loss(theta_id) for actor in actors]

then each task that got sent to the scheduler (one for every element of batch_ids) would have had a copy of theta
serialized inside of it. Since theta here consists of the parameters of a potentially large model, this is inefficient.
Large objects should be passed by object ID to remote functions and not by value.

We use remote actors and remote objects internally in the implementation of full_loss and full_grad, but the
user-facing behavior of these methods is identical to the behavior in the serial version.

We can now optimize the objective with the same function call as before.

theta_init = 1e-2 * np.random.normal(size=dim)
result = scipy.optimize.fmin_l_bfgs_b(full_loss, theta_init, fprime=full_grad)

1.27 Evolution Strategies

This document provides a walkthrough of the evolution strategies example. To run the application, first install some
dependencies.

```
pip install tensorflow
pip install gym
```

You can view the code for this example.

The script can be run as follows. Note that the configuration is tuned to work on the Humanoid-v1 gym environment.

```
python ray/rllib/train.py --env=Humanoid-v1 --run=ES
```

To train a policy on a cluster (e.g., using 900 workers), run the following.

```
python ray/python/ray/rllib/train.py \
    --env=Humanoid-v1 \n    --run=ES \n    --redis-address=<redis-address> \n    --config=""num_workers": 900, "episodes_per_batch": 10000, "timesteps_per_batch" =": 100000"
```
At the heart of this example, we define a `Worker` class. These workers have a method `do_rollouts`, which will be used to perform simulate randomly perturbed policies in a given environment.

```python
@ray.remote
class Worker(object):
    def __init__(self, config, policy_params, env_name, noise):
        self.env = # Initialize environment.
        self.policy = # Construct policy.
        # Details omitted.

    def do_rollouts(self, params):
        perturbation = # Generate a random perturbation to the policy.

        self.policy.set_weights(params + perturbation)
        # Do rollout with the perturbed policy.

        self.policy.set_weights(params - perturbation)
        # Do rollout with the perturbed policy.

        # Return the rewards.
```

In the main loop, we create a number of actors with this class.

```python
workers = [Worker.remote(config, policy_params, env_name, noise_id)
            for _ in range(num_workers)]
```

We then enter an infinite loop in which we use the actors to perform rollouts and use the rewards from the rollouts to update the policy.

```python
while True:
    # Get the current policy weights.
    theta = policy.get_weights()
    # Put the current policy weights in the object store.
    theta_id = ray.put(theta)
    # Use the actors to do rollouts, note that we pass in the ID of the policy
    # weights.
    rollout_ids = [worker.do_rollouts.remote(theta_id), for worker in workers]
    # Get the results of the rollouts.
    results = ray.get(rollout_ids)
    # Update the policy.
    optimizer.update(...)
```

In addition, note that we create a large object representing a shared block of random noise. We then put the block in the object store so that each `Worker` actor can use it without creating its own copy.

```python
@ray.remote
def create_shared_noise():
    noise = np.random.randn(250000000)
    return noise

noise_id = create_shared_noise.remote()
```

Recall that the `noise_id` argument is passed into the actor constructor.
1.28 Cython

1.28.1 Getting Started

This document provides examples of using Cython-generated code in ray. To get started, run the following from directory $RAY_HOME/examples/cython:

```bash
pip install scipy # For BLAS example
pip install -e .
python cython_main.py --help
```

You can import the cython_examples module from a Python script or interpreter.

1.28.2 Notes

- You **must** include the following two lines at the top of any *.pyx file:

  ```python
  #!python
  # cython: embedsignature=True, binding=True
  ```

- You cannot decorate Cython functions within a *.pyx file (there are ways around this, but creates a leaky abstraction between Cython and Python that would be very challenging to support generally). Instead, prefer the following in your Python code:

  ```python
  some_cython_func = ray.remote(some_cython_module.some_cython_func)
  ```

- You cannot transfer memory buffers to a remote function (see example8, which currently fails); your remote function must return a value.

- Have a look at cython_main.py, cython_simple.pyx, and setup.py for examples of how to call, define, and build Cython code, respectively. The Cython documentation is also very helpful.

- Several limitations come from Cython’s own unsupported Python features.

- We currently do not support compiling and distributing Cython code to ray clusters. In other words, Cython developers are responsible for compiling and distributing any Cython code to their cluster (much as would be the case for users who need Python packages like scipy).

- For most simple use cases, developers need not worry about Python 2 or 3, but users who do need to care can have a look at the language_level Cython compiler directive (see here).

1.29 Streaming MapReduce

This document walks through how to implement a simple streaming application using Ray’s actor capabilities. It implements a streaming MapReduce which computes word counts on wikipedia articles.

You can view the code for this example.

To run the example, you need to install the dependencies

```bash
pip install wikipedia
```

and then execute the script as follows:
python ray/examples/streaming/streaming.py

For each round of articles read, the script will output the top 10 words in these articles together with their word count:

```
article index = 0
  the 2866
  of 1688
  and 1448
  in 1101
  to 593
  a 553
  is 509
  as 325
  are 284
  by 261
article index = 1
  the 3597
  of 1971
  and 1735
  in 1429
  to 670
  a 623
  is 578
  as 401
  by 293
  for 285
article index = 2
  the 3910
  of 2123
  and 1890
  in 1468
  to 658
  a 653
  is 488
  as 364
  by 362
  for 297
article index = 3
  the 2962
  of 1667
  and 1472
  in 1220
  a 546
  to 538
  is 516
  as 307
  by 253
  for 243
article index = 4
  the 3523
  of 1866
  and 1690
  in 1475
  to 645
  a 583
  is 572
  as 352
```
Note that this examples uses distributed actor handles, which are still considered experimental.

There is a `Mapper` actor, which has a method `get_range` used to retrieve word counts for words in a certain range:

```python
@ray.remote
class Mapper(object):
    def __init__(self, title_stream):
        # Constructor, the title stream parameter is a stream of wikipedia article titles that will be read by this mapper
    def get_range(self, article_index, keys):
        # Return counts of all the words with first letter between keys[0] and keys[1] in the articles that haven't been read yet with index up to article_index

The `Reducer` actor holds a list of mappers, calls `get_range` on them and accumulates the results.

```python
@ray.remote
class Reducer(object):
    def __init__(self, keys, *mappers):
        # Constructor for a reducer that gets input from the list of mappers and accumulates word counts for words with first letter between keys[0] and keys[1]
    def next_reduce_result(self, article_index):
        # Get articles up to article_index that haven't been read yet, accumulate the word counts and return them
```

On the driver, we then create a number of mappers and reducers and run the streaming MapReduce:

```python
streams = # Create list of num_mappers streams
keys = # Partition the keys among the reducers.

# Create a number of mappers.
mappers = [Mapper.remote(stream) for stream in streams]

# Create a number of reduces, each responsible for a different range of keys.
# This gives each Reducer actor a handle to each Mapper actor.
reducers = [Reducer.remote(key, *mappers) for key in keys]

article_index = 0
while True:
    counts = ray.get([reducer.next_reduce_result.remote(article_index) for reducer in reducers])
    article_index += 1
```

The actual example reads a list of articles and creates a stream object which produces an infinite stream of articles from the list. This is a toy example meant to illustrate the idea. In practice we would produce a stream of non-repeating items for each mapper.

1.29. Streaming MapReduce
1.30 Using Ray with TensorFlow

This document describes best practices for using Ray with TensorFlow.

To see more involved examples using TensorFlow, take a look at A3C, ResNet, Policy Gradients, and LBFGS.

If you are training a deep network in the distributed setting, you may need to ship your deep network between processes (or machines). For example, you may update your model on one machine and then use that model to compute a gradient on another machine. However, shipping the model is not always straightforward.

For example, a straightforward attempt to pickle a TensorFlow graph gives mixed results. Some examples fail, and some succeed (but produce very large strings). The results are similar with other pickling libraries as well.

Furthermore, creating a TensorFlow graph can take tens of seconds, and so serializing a graph and recreating it in another process will be inefficient. The better solution is to create the same TensorFlow graph on each worker once at the beginning and then to ship only the weights between the workers.

Suppose we have a simple network definition (this one is modified from the TensorFlow documentation).

```python
import tensorflow as tf
import numpy as np

# Define the network
x_data = tf.placeholder(tf.float32, shape=[100])
y_data = tf.placeholder(tf.float32, shape=[100])
w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = w * x_data + b

# Define the loss function and optimizer
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
gradients = optimizer.compute_gradients(loss)
train = optimizer.apply_gradients(gradients)
init = tf.global_variables_initializer()
sess = tf.Session()

# Initialize the weights
sess.run(init)
# Get the weights
weights = variables.get_weights()  # Returns a dictionary of numpy arrays
# Set the weights
variables.set_weights(weights)
```

To extract the weights and set the weights, you can use the following helper method.

```python
import ray
variables = ray.experimental.TensorFlowVariables(loss, sess)
```

The TensorFlowVariables object provides methods for getting and setting the weights as well as collecting all of the variables in the model.

Now we can use these methods to extract the weights, and place them back in the network as follows.

```python
# First initialize the weights.
sess.run(init)
# Get the weights
weights = variables.get_weights()  # Returns a dictionary of numpy arrays
# Set the weights
variables.set_weights(weights)
```

**Note:** If we were to set the weights using the assign method like below, each call to assign would add a node to the graph, and the graph would grow unmanageably large over time.

```python
w.assign(np.zeros(1))  # This adds a node to the graph every time you call it.
b.assign(np.zeros(1))  # This adds a node to the graph every time you call it.
```
1.30.1 Complete Example for Weight Averaging

Putting this all together, we would first embed the graph in an actor. Within the actor, we would use the `get_weights` and `set_weights` methods of the `TensorFlowVariables` class. We would then use those methods to ship the weights (as a dictionary of variable names mapping to numpy arrays) between the processes without shipping the actual TensorFlow graphs, which are much more complex Python objects.

```python
import tensorflow as tf
import numpy as np
import ray

ray.init()

BATCH_SIZE = 100
NUM_BATCHES = 1
NUM_ITERS = 201

class Network(object):
    def __init__(self, x, y):
        # Seed TensorFlow to make the script deterministic.
        tf.set_random_seed(0)

        # Define the inputs.
        self.x_data = tf.constant(x, dtype=tf.float32)
        self.y_data = tf.constant(y, dtype=tf.float32)

        # Define the weights and computation.
        w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
        b = tf.Variable(tf.zeros([1]))
        y = w * self.x_data + b

        # Define the loss.
        self.loss = tf.reduce_mean(tf.square(y - self.y_data))

        optimizer = tf.train.GradientDescentOptimizer(0.5)
        self.grads = optimizer.compute_gradients(self.loss)
        self.train = optimizer.apply_gradients(self.grads)

        # Define the weight initializer and session.
        init = tf.global_variables_initializer()
        self.sess = tf.Session()

    # Additional code for setting and getting the weights
    self.variables = ray.experimental.TensorFlowVariables(self.loss, self.sess)

    # Return all of the data needed to use the network.
    self.sess.run(init)

    # Define a remote function that trains the network for one step and returns the # new weights.
    def step(self, weights):
        # Set the weights in the network.
        self.variables.set_weights(weights)

        # Do one step of training.
        self.sess.run(self.train)

        # Return the new weights.
        return self.variables.get_weights()

    def get_weights(self):
        return self.variables.get_weights()

# Define a remote function for generating fake data.
@ray.remote(num_return_vals=2)
def generate_fake_x_y_data(num_data, seed=0):
    # Seed numpy to make the script deterministic.
    (continues on next page)
```
np.random.seed(seed)
x = np.random.rand(num_data)
y = x * 0.1 + 0.3
return x, y

# Generate some training data.
batch_ids = [generate_fake_x_y_data.remote(BATCH_SIZE, seed=i) for i in range(NUM_BATCHES)]
x_ids = [x_id for x_id, y_id in batch_ids]
y_ids = [y_id for x_id, y_id in batch_ids]

# Generate some test data.
x_test, y_test = ray.get(generate_fake_x_y_data.remote(BATCH_SIZE, seed=NUM_BATCHES))

# Create actors to store the networks.
remote_network = ray.remote(Network)
actor_list = [remote_network.remote(x_ids[i], y_ids[i]) for i in range(NUM_BATCHES)]

# Get initial weights of some actor.
weights = ray.get(actor_list[0].get_weights.remote())

# Do some steps of training.
for iteration in range(NUM_ITERS):
    # Put the weights in the object store. This is optional. We could instead pass
    # the variable weights directly into step.remote, in which case it would be
    # placed in the object store under the hood. However, in that case multiple
    # copies of the weights would be put in the object store, so this approach is
    # more efficient.
    weights_id = ray.put(weights)
    # Call the remote function multiple times in parallel.
    new_weights_ids = [actor.step.remote(weights_id) for actor in actor_list]
    # Get all of the weights.
    new_weights_list = ray.get(new_weights_ids)
    # Add up all the different weights. Each element of new_weights_list is a dict
    # of weights, and we want to add up these dicts component wise using the keys
    # of the first dict.
    weights = {variable: sum(weight_dict[variable] for weight_dict in new_weights_list[0]) / NUM_BATCHES for variable in new_weights_list[0]}
    # Print the current weights. They should converge to roughly to the values 0.1
    # and 0.3 used in generate_fake_x_y_data.
    if iteration % 20 == 0:
        print("Iteration {}: weights are {}.format(iteration, weights))

1.30.2 How to Train in Parallel using Ray and Gradients

In some cases, you may want to do data-parallel training on your network. We use the network above to illustrate how
to do this in Ray. The only differences are in the remote function step and the driver code.

In the function step, we run the grad operation rather than the train operation to get the gradients. Since Tensorflow
pairs the gradients with the variables in a tuple, we extract the gradients to avoid needless computation.

Extracting numerical gradients

Code like the following can be used in a remote function to compute numerical gradients.
Using the returned gradients to train the network

By pairing the symbolic gradients with the numerical gradients in a feed_dict, we can update the network.

```python
# We can feed the gradient values in using the associated symbolic gradient operation defined in tensorflow.
feed_dict = {grad[0]: numerical_grad for (grad, numerical_grad) in zip(grads, numerical_grads)}
sess.run(train, feed_dict=feed_dict)
```

You can then run `variables.get_weights()` to see the updated weights of the network.

For reference, the full code is below:

```python
import tensorflow as tf
import numpy as np
import ray

ray.init()  # Batch Size = 100
BATCH_SIZE = 100
NUM_BATCHES = 1
NUM_ITERS = 201

class Network(object):
    def __init__(self, x, y):
        # Seed TensorFlow to make the script deterministic.
        tf.set_random_seed(0)
        # Define the inputs.
        x_data = tf.constant(x, dtype=tf.float32)
        y_data = tf.constant(y, dtype=tf.float32)
        # Define the weights and computation.
        w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
        b = tf.Variable(tf.zeros([1]))
        y = w * x_data + b
        # Define the loss.
        self.loss = tf.reduce_mean(tf.square(y - y_data))
        optimizer = tf.train.GradientDescentOptimizer(0.5)
        self.grads = optimizer.compute_gradients(self.loss)
        self.train = optimizer.apply_gradients(self.grads)
        # Define the weight initializer and session.
        init = tf.global_variables_initializer()
        self.sess = tf.Session()
        # Additional code for setting and getting the weights
        self.variables = ray.experimental.TensorFlowVariables(self.loss, self.sess)
        # Return all of the data needed to use the network.
        self.sess.run(init)

    # Define a remote function that trains the network for one step and returns the new weights.
    def step(self, weights):

    (continues on next page)
```
# Set the weights in the network.
self.variables.set_weights(weights)

# Do one step of training. We only need the actual gradients so we filter over the list.
actual_grads = self.sess.run([grad[0] for grad in self.grads])
return actual_grads

def get_weights(self):
    return self.variables.get_weights()

# Define a remote function for generating fake data.
@ray.remote(num_return_vals=2)
def generate_fake_x_y_data(num_data, seed=0):
    # Seed numpy to make the script deterministic.
    np.random.seed(seed)
    x = np.random.rand(num_data)
    y = x * 0.1 + 0.3
    return x, y

# Generate some training data.
batch_ids = [generate_fake_x_y_data.remote(BATCH_SIZE, seed=i) for i in range(NUM_BATCHES)]
x_ids = [x_id for x_id, y_id in batch_ids]
y_ids = [y_id for x_id, y_id in batch_ids]

# Generate some test data.
x_test, y_test = ray.get(generate_fake_x_y_data.remote(BATCH_SIZE, seed=NUM_BATCHES))

# Create actors to store the networks.
remote_network = ray.remote(Network)
actor_list = [remote_network.remote(x_ids[i], y_ids[i]) for i in range(NUM_BATCHES)]
local_network = Network(x_test, y_test)

# Get initial weights of local network.
weights = local_network.get_weights()

# Do some steps of training.
for iteration in range(NUM_ITERS):
    # Put the weights in the object store. This is optional. We could instead pass
    # the variable weights directly into step.remote, in which case it would be
    # placed in the object store under the hood. However, in that case multiple
    # copies of the weights would be put in the object store, so this approach is
    # more efficient.
    weights_id = ray.put(weights)
    # Call the remote function multiple times in parallel.
    gradients_ids = [actor.step.remote(weights_id) for actor in actor_list]
    # Get all of the weights.
    gradients_list = ray.get(gradients_ids)

    # Take the mean of the different gradients. Each element of gradients_list is a
    # list # of gradients, and we want to take the mean of each one.
    mean_grads = [sum([gradients[i] for gradients in gradients_list]) / len(gradients_list) for i in range(len(gradients_list[0]))]

    feed_dict = {grad[0]: mean_grad for (grad, mean_grad) in zip(local_network.grads, mean_grads)}
    local_network.sess.run(local_network.train, feed_dict=feed_dict)
weights = local_network.get_weights()

# Print the current weights. They should converge to roughly to the values 0.1
# and 0.3 used in generate_fake_x_y_data.
if iteration % 20 == 0:
    print("Iteration {}: weights are {}".format(iteration, weights))

class ray.experimental.TensorFlowVariables(output, sess=None, input_variables=None)
A class used to set and get weights for Tensorflow networks.

    sess
tf.Session – The tensorflow session used to run assignment.

    variables
    Dict[str, tf.Variable] – Extracted variables from the loss or additional variables that are passed in.

    placeholders

    assignment_nodes

    set_session(sess)
Sets the current session used by the class.

        Parameters sess(tf.Session) – Session to set the attribute with.

    get_flat_size()
Returns the total length of all of the flattened variables.

        Returns The length of all flattened variables concatenated.

    get_flat()
Gets the weights and returns them as a flat array.

        Returns 1D Array containing the flattened weights.

    set_flat(new_weights)
Sets the weights to new_weights, converting from a flat array.

        Note: You can only set all weights in the network using this function, i.e., the length of the array must
match get_flat_size.

    Parameters new_weights(np.ndarray) – Flat array containing weights.

    get_weights()
Returns a dictionary containing the weights of the network.

        Returns Dictionary mapping variable names to their weights.

    set_weights(new_weights)
Sets the weights to new_weights.

        Note: Can set subsets of variables as well, by only passing in the variables you want to be set.

        Parameters new_weights(Dict) – Dictionary mapping variable names to their weights.
1.30.3 Troubleshooting

Note that `TensorFlowVariables` uses variable names to determine what variables to set when calling `set_weights`. One common issue arises when two networks are defined in the same TensorFlow graph. In this case, TensorFlow appends an underscore and integer to the names of variables to disambiguate them. This will cause `TensorFlowVariables` to fail. For example, if we have a class definition `Network` with a `TensorFlowVariables` instance:

```python
import ray
import tensorflow as tf

class Network(object):
    def __init__(self):
        a = tf.Variable(1)
        b = tf.Variable(1)
        c = tf.add(a, b)
        sess = tf.Session()
        init = tf.global_variables_initializer()
        sess.run(init)
        self.variables = ray.experimental.TensorFlowVariables(c, sess)

    def set_weights(self, weights):
        self.variables.set_weights(weights)

    def get_weights(self):
        return self.variables.get_weights()

a = Network()
b = Network()
b.set_weights(a.get_weights())
```

and run the following code:

```python
a = Network()
b = Network()
b.set_weights(a.get_weights())
```

the code would fail. If we instead defined each network in its own TensorFlow graph, then it would work:

```python
with tf.Graph().as_default():
a = Network()
with tf.Graph().as_default():
b = Network()
b.set_weights(a.get_weights())
```

This issue does not occur between actors that contain a network, as each actor is in its own process, and thus is in its own graph. This also does not occur when using `set_flat`.

Another issue to keep in mind is that `TensorFlowVariables` needs to add new operations to the graph. If you close the graph and make it immutable, e.g. creating a `MonitoredTrainingSession` the initialization will fail. To resolve this, simply create the instance before you close the graph.

1.31 An Overview of the Internals

In this document, we trace through in more detail what happens at the system level when certain API calls are made.
1.31.1 Connecting to Ray

There are two ways that a Ray script can be initiated. It can either be run in a standalone fashion or it can be connect to an existing Ray cluster.

Running Ray standalone

Ray can be used standalone by calling `ray.init()` within a script. When the call to `ray.init()` happens, all of the relevant processes are started. These include a local scheduler, a global scheduler, an object store and manager, a Redis server, and a number of worker processes.

When the script exits, these processes will be killed.

Note: This approach is limited to a single machine.

Connecting to an existing Ray cluster

To connect to an existing Ray cluster, simply pass the argument address of the Redis server as the `redis_address=` keyword argument into `ray.init`. In this case, no new processes will be started when `ray.init` is called, and similarly the processes will continue running when the script exits. In this case, all processes except workers that correspond to actors are shared between different driver processes.

1.31.2 Defining a remote function

A central component of this system is the centralized control plane. This is implemented using one or more Redis servers. Redis is an in-memory key-value store.

We use the centralized control plane in two ways. First, as persistent store of the system’s control state. Second, as a message bus for communication between processes (using Redis’s publish-subscribe functionality).

Now, consider a remote function definition as below.

```python
@ray.remote
def f(x):
    return x + 1
```

When the remote function is defined as above, the function is immediately pickled, assigned a unique ID, and stored in a Redis server. You can view the remote functions in the centralized control plane as below.

TODO: Fill this in.

Each worker process has a separate thread running in the background that listens for the addition of remote functions to the centralized control state. When a new remote function is added, the thread fetches the pickled remote function, unpickles it, and can then execute that function.

Notes and limitations

- Because we export remote functions as soon as they are defined, that means that remote functions can’t close over variables that are defined after the remote function is defined. For example, the following code gives an error.
@ray.remote
def f(x):
    return helper(x)
def helper(x):
    return x + 1

If you call `f.remote(0)`, it will give an error of the form:

```
Traceback (most recent call last):
  File "<ipython-input-3-12a5beeb2306>", line 3, in f
NameError: name 'helper' is not defined
```

On the other hand, if `helper` is defined before `f`, then it will work.

### 1.31.3 Calling a remote function

When a driver or worker invokes a remote function, a number of things happen.

- First, a task object is created. The task object includes the following.
  - The ID of the function being called.
  - The IDs or values of the arguments to the function. Python primitives like integers or short strings will be pickled and included as part of the task object. Larger or more complex objects will be put into the object store with an internal call to `ray.put`, and the resulting IDs are included in the task object. Object IDs that are passed directly as arguments are also included in the task object.
  - The ID of the task. This is generated uniquely from the above content.
  - The IDs for the return values of the task. These are generated uniquely from the above content.

- The task object is then sent to the local scheduler on the same node as the driver or worker.

- The local scheduler makes a decision to either schedule the task locally or to pass the task on to a global scheduler.
  - If all of the task’s object dependencies are present in the local object store and there are enough CPU and GPU resources available to execute the task, then the local scheduler will assign the task to one of its available workers.
  - If those conditions are not met, the task will be passed on to a global scheduler. This is done by adding the task to the task table, which is part of the centralized control state. The task table can be inspected as follows.

```python
todo: Fill this in.
```

A global scheduler will be notified of the update and will assign the task to a local scheduler by updating the task’s state in the task table. The local scheduler will be notified and pull the task object.

- Once a task has been scheduled to a local scheduler, whether by itself or by a global scheduler, the local scheduler queues the task for execution. A task is assigned to a worker when enough resources become available and the object dependencies are available locally, in first-in, first-out order.

- When the task has been assigned to a worker, the worker executes the task and puts the task’s return values into the object store. The object store will then update the object table, which is part of the centralized control state, to reflect the fact that it contains the newly created objects. The object table can be viewed as follows.
When the task’s return values are placed into the object store, they are first serialized into a contiguous blob of bytes using the Apache Arrow data layout, which is helpful for efficiently sharing data between processes using shared memory.

Notes and limitations

• When an object store on a particular node fills up, it will begin evicting objects in a least-recently-used manner. If an object that is needed later is evicted, then the call to `ray.get` for that object will initiate the reconstruction of the object. The local scheduler will attempt to reconstruct the object by replaying its task lineage.

Todo: Limitations on reconstruction.

1.31.4 Getting an object ID

Several things happen when a driver or worker calls `ray.get` on an object ID.

```
ray.get(x_id)
```

• The driver or worker goes to the object store on the same node and requests the relevant object. Each object store consists of two components, a shared-memory key-value store of immutable objects, and a manager to coordinate the transfer of objects between nodes.

  – If the object is not present in the object store, the manager checks the object table to see which other object stores, if any, have the object. It then requests the object directly from one of those object stores, via its manager. If the object doesn’t exist anywhere, then the centralized control state will notify the requesting manager when the object is created. If the object doesn’t exist anywhere because it has been evicted from all object stores, the worker will also request reconstruction of the object from the local scheduler. These checks repeat periodically until the object is available in the local object store, whether through reconstruction or through object transfer.

• Once the object is available in the local object store, the driver or worker will map the relevant region of memory into its own address space (to avoid copying the object), and will deserialize the bytes into a Python object. Note that any numpy arrays that are part of the object will not be copied.

1.32 Serialization in the Object Store

This document describes what Python objects Ray can and cannot serialize into the object store. Once an object is placed in the object store, it is immutable.

There are a number of situations in which Ray will place objects in the object store.

1. The return values of a remote function.
2. The value `x` in a call to `ray.put(x)`.
3. Arguments to remote functions (except for simple arguments like ints or floats).

A Python object may have an arbitrary number of pointers with arbitrarily deep nesting. To place an object in the object store or send it between processes, it must first be converted to a contiguous string of bytes. This process is known as serialization. The process of converting the string of bytes back into a Python object is known as deserialization. Serialization and deserialization are often bottlenecks in distributed computing.

Pickle is one example of a library for serialization and deserialization in Python.
Pickle (and the variant we use, cloudpickle) is general-purpose. It can serialize a large variety of Python objects. However, for numerical workloads, pickling and unpickling can be inefficient. For example, if multiple processes want to access a Python list of numpy arrays, each process must unpickle the list and create its own new copies of the arrays. This can lead to high memory overheads, even when all processes are read-only and could easily share memory.

In Ray, we optimize for numpy arrays by using the Apache Arrow data format. When we deserialize a list of numpy arrays from the object store, we still create a Python list of numpy array objects. However, rather than copy each numpy array, each numpy array object holds a pointer to the relevant array held in shared memory. There are some advantages to this form of serialization.

- Deserialization can be very fast.
- Memory is shared between processes so worker processes can all read the same data without having to copy it.

### 1.32.1 What Objects Does Ray Handle

Ray does not currently support serialization of arbitrary Python objects. The set of Python objects that Ray can serialize using Arrow includes the following.

1. Primitive types: ints, floats, longs, bools, strings, unicode, and numpy arrays.
2. Any list, dictionary, or tuple whose elements can be serialized by Ray.

For a more general object, Ray will first attempt to serialize the object by unpacking the object as a dictionary of its fields. This behavior is not correct in all cases. If Ray cannot serialize the object as a dictionary of its fields, Ray will fall back to using pickle. However, using pickle will likely be inefficient.

### 1.32.2 Notes and limitations

- We currently handle certain patterns incorrectly, according to Python semantics. For example, a list that contains two copies of the same list will be serialized as if the two lists were distinct.

```python
l1 = [0]
l2 = [l1, l1]
l3 = ray.get(ray.put(l2))

l2[0] is l2[1]  # True.
l3[0] is l3[1]  # False.
```

- For reasons similar to the above example, we also do not currently handle objects that recursively contain themselves (this may be common in graph-like data structures).

```python
l = []
l.append(l)

# Try to put this list that recursively contains itself in the object store.
ray.put(l)
```

This will throw an exception with a message like the following.

```
This object exceeds the maximum recursion depth. It may contain itself recursively.
```

- Whenever possible, use numpy arrays for maximum performance.
1.32.3 Last Resort Workaround

If you find cases where Ray serialization doesn’t work or does something unexpected, please let us know so we can fix it. In the meantime, you may have to resort to writing custom serialization and deserialization code (e.g., calling pickle by hand).

```python
import pickle

@ray.remote
def f(complicated_object):
    # Deserialize the object manually.
    obj = pickle.loads(complicated_object)
    return "Successfully passed {} into f.".format(obj)

# Define a complicated object.
l = []
l.append(l)

# Manually serialize the object and pass it in as a string.
ray.get(f.remote(pickle.dumps(l)))  # prints 'Successfully passed [...] into f.'
```

Note: If you have trouble with pickle, you may have better luck with cloudpickle.

1.33 Fault Tolerance

This document describes the handling of failures in Ray.

1.33.1 Machine and Process Failures

Currently, each local scheduler and each plasma manager send heartbeats to a monitor process. If the monitor does not receive any heartbeats from a given process for some duration of time (about ten seconds), then it will mark that process as dead. The monitor process will then clean up the associated state in the Redis servers. If a manager is marked as dead, the object table will be updated to remove all occurrences of that manager so that other managers don’t try to fetch objects from the dead manager. If a local scheduler is marked as dead, all of the tasks that are marked as executing on that local scheduler in the task table will be marked as lost and all actors associated with that local scheduler will be recreated by other local schedulers.

1.33.2 Lost Objects

If an object is needed but is lost or was never created, then the task that created the object will be re-executed to create the object. If necessary, tasks needed to create the input arguments to the task being re-executed will also be re-executed.

1.33.3 Actors

When a local scheduler is marked as dead, all actors associated with that local scheduler that were still alive will be recreated by other local schedulers. By default, all of the actor methods will be re-executed in the same order that they were initially executed. If actor checkpointing is enabled, then the actor state will be loaded from the most recent checkpoint and the actor methods that occurred after the checkpoint will be re-executed. Note that actor checkpointing is currently an experimental feature.
1.33.4 Current Limitations

At the moment, Ray does not handle all failure scenarios. We are working on addressing these known problems.

Process Failures

1. Ray does not recover from the failure of any of the following processes: a Redis server, the global scheduler, the monitor process.
2. If a driver fails, that driver will not be restarted and the job will not complete.

Lost Objects

1. If an object is constructed by a call to `ray.put` on the driver, is then evicted, and is later needed, Ray will not reconstruct this object.
2. If an object is constructed by an actor method, is then evicted, and is later needed, Ray will not reconstruct this object.

Actor Reconstruction

1. Actor reconstruction follows the order of initial execution, but new tasks may get interleaved with the re-executed tasks.

1.34 The Plasma Object Store

Plasma is a high-performance shared memory object store originally developed in Ray and now being developed in Apache Arrow. See the relevant documentation.

1.34.1 Using Plasma with Huge Pages

On Linux, it is possible to increase the write throughput of the Plasma object store by using huge pages. You first need to create a file system and activate huge pages as follows.

```
sudo mkdir -p /mnt/hugepages
gid=`id -g`
uid=`id -u`
sudo mount -t hugetlbfs -o uid=$uid -o gid=$gid none /mnt/hugepages
sudo bash -c "echo $gid > /proc/sys/vm/hugetlb_shm_group"
sudo bash -c "echo 20000 > /proc/sys/vm/nr_hugepages"
```

You need root access to create the file system, but not for running the object store.

You can then start Ray with huge pages on a single machine as follows.

```
ray.init(huge_pages=True, plasma_directory="/mnt/hugepages")
```

In the cluster case, you can do it by passing `--huge-pages` and `--plasma-directory=/mnt/hugepages` into `ray start` on any machines where huge pages should be enabled.

See the relevant Arrow documentation for huge pages.
1.35 Resource (CPUs, GPUs)

This document describes how resources are managed in Ray. Each node in a Ray cluster knows its own resource capacities, and each task specifies its resource requirements.

1.35.1 CPUs and GPUs

The Ray backend includes built-in support for CPUs and GPUs.

**Specifying a node’s resource requirements**

To specify a node’s resource requirements from the command line, pass the `--num-cpus` and `--num-cpus` flags into `ray start`.

```python
# To start a head node.
ray start --head --num-cpus=8 --num-gpus=1

# To start a non-head node.
ray start --redis-address=<redis-address> --num-cpus=4 --num-gpus=2
```

To specify a node’s resource requirements when the Ray processes are all started through `ray.init`, do the following.

```python
ray.init(num_cpus=8, num_gpus=1)
```

If the number of CPUs is unspecified, Ray will automatically determine the number by running `psutil.cpu_count()`. If the number of GPUs is unspecified, Ray will attempt to automatically detect the number of GPUs.

**Specifying a task’s CPU and GPU requirements**

To specify a task’s CPU and GPU requirements, pass the `num_cpus` and `num_gpus` arguments into the remote decorator.

```python
@ray.remote(num_cpus=4, num_gpus=2)
def f():
    return 1
```

When `f` tasks will be scheduled on machines that have at least 4 CPUs and 2 GPUs, and when one of the `f` tasks executes, 4 CPUs and 2 GPUs will be reserved for that task. The IDs of the GPUs that are reserved for the task can be accessed with `ray.get_gpu_ids()`. Ray will automatically set the environment variable `CUDA_VISIBLE_DEVICES` for that process. These resources will be released when the task finishes executing.

However, if the task gets blocked in a call to `ray.get`. For example, consider the following remote function.

```python
@ray.remote(num_cpus=1, num_gpus=1)
def g():
    return ray.get(f.remote())
```

When a `g` task is executing, it will release its CPU resources when it gets blocked in the call to `ray.get`. It will reacquire the CPU resources when `ray.get` returns. It will retain its GPU resources throughout the lifetime of the task because the task will most likely continue to use GPU memory.

To specify that an actor requires GPUs, do the following.
When an `Actor` instance is created, it will be placed on a node that has at least 1 GPU, and the GPU will be reserved for the actor for the duration of the actor’s lifetime (even if the actor is not executing tasks). The GPU resources will be released when the actor terminates. Note that currently only GPU resources are used for actor placement.

### 1.3.5.2 Custom Resources

While Ray has built-in support for CPUs and GPUs, nodes can be started with arbitrary custom resources. All custom resources behave like GPUs.

A node can be started with some custom resources as follows.

```bash
ray start --head --resources='{"Resource1": 4, "Resource2": 16}'
```

It can be done through `ray.init` as follows.

```python
ray.init(resources={'Resource1': 4, 'Resource2': 16})
```

To require custom resources in a task, specify the requirements in the remote decorator.

```python
@ray.remote(resources={'Resource2': 1})
def f():
    return 1
```

### 1.3.6 Redis Memory Management (EXPERIMENTAL)

Ray stores metadata associated with tasks and objects in one or more Redis servers, as described in An Overview of the Internals. Applications that are long-running or have high task/object generation rate could risk high memory pressure, potentially leading to out-of-memory (OOM) errors.

Here, we describe an experimental feature that transparently flushes metadata entries out of Redis memory.

#### 1.3.6.1 Requirements

As of early July 2018, the automatic memory management feature requires building Ray from source. We are planning on eliminating this step in the near future by releasing official wheels.

**Building Ray**

First, follow instructions to build Ray from source to install prerequisites. After the prerequisites are installed, instead of doing the regular `pip install` as referenced in that document, pass an additional special flag, `RAY_USE_NEW_GCS=on`:

```bash
git clone https://github.com/ray-project/ray.git
cd ray/python
RAY_USE_NEW_GCS=on pip install -e . --verbose  # Add --user if you see a permission denied error.
```
Running Ray applications

At run time the environment variables `RAY_USE_NEW_GCS=on` and `RAY_USE_XRAY=1` are required.

```bash
export RAY_USE_NEW_GCS=on
export RAY_USE_XRAY=1
python my_ray_script.py  # Or launch python/ipython.
```

### 1.36.2 Activate memory flushing

After building Ray using the method above, simply add these two lines after `ray.init()` to activate automatic memory flushing:

```python
ray.init(...)
policy = ray.experimental.SimpleGcsFlushPolicy()
ray.experimental.set_flushing_policy(policy)

# My awesome Ray application logic follows.
```

### 1.36.3 Parameters of the flushing policy

There are three user-configurable parameters of the `SimpleGcsFlushPolicy`:

- **flush_when_at_least_bytes**: Wait until this many bytes of memory usage accumulated in the redis server before flushing kicks in.
- **flush_period_secs**: Issue a flush to the Redis server every this many seconds.
- **flush_num_entries_each_time**: A hint to the system on the number of entries to flush on each request.

The default values should serve to be non-invasive for lightweight Ray applications. `flush_when_at_least_bytes` is set to `(1<<31)` or 2GB, `flush_period_secs` to 10, and `flush_num_entries_each_time` to 10000:

```python
# Default parameters.
ray.experimental.SimpleGcsFlushPolicy(
    flush_when_at_least_bytes=(1 << 31),
    flush_period_secs=10,
    flush_num_entries_each_time=10000)
```

In particular, these default values imply that

1. the Redis server would accumulate memory usage up to 2GB without any entries being flushed, then the flushing would kick in; and
2. generally, “older” metadata entries would be flushed first, and the Redis server would always keep the most recent window of metadata of 2GB in size.

**For advanced users.** Advanced users can tune the above parameters to their applications’ needs; note that the desired flush rate is equal to `(flush period) * (num entries each flush).`
1.37 Cloud Setup and Auto-Scaling

The `ray create_or_update` command starts an AWS or GCP Ray cluster from your personal computer. Once the cluster is up, you can then SSH into it to run Ray programs.

### 1.37.1 Quick start (AWS)

First, install boto (`pip install boto3`) and configure your AWS credentials in `~/.aws/credentials`, as described in the boto docs.

Then you’re ready to go. The provided `ray/python/ray/autoscaler/aws/example-full.yaml` cluster config file will create a small cluster with a m5.large head node (on-demand) configured to autoscale up to two m5.large spot workers.

Try it out by running these commands from your personal computer. Once the cluster is started, you can then SSH into the head node, `source activate tensorflow_p36`, and then run Ray programs with `ray.init(redis_address="localhost:6379")`.

```bash
# Create or update the cluster. When the command finishes, it will print
# out the command that can be used to SSH into the cluster head node.
$ ray create_or_update ray/python/ray/autoscaler/aws/example-full.yaml

# Reconfigure autoscaling behavior without interrupting running jobs
$ ray create_or_update ray/python/ray/autoscaler/aws/example-full.yaml
  --max-workers=N --no-restart

# Teardown the cluster
$ ray teardown ray/python/ray/autoscaler/aws/example-full.yaml
```

### 1.37.2 Quick start (GCP)

First, install the Google API client (`pip install google-api-python-client`), set up your GCP credentials, and create a new GCP project.

Then you’re ready to go. The provided `ray/python/ray/autoscaler/gcp/example-full.yaml` cluster config file will create a small cluster with a n1-standard-2 head node (on-demand) configured to autoscale up to two n1-standard-2 preemptible workers. Note that you’ll need to fill in your project id in those templates.

Try it out by running these commands from your personal computer. Once the cluster is started, you can then SSH into the head node and then run Ray programs with `ray.init(redis_address="localhost:6379")`.

```bash
# Create or update the cluster. When the command finishes, it will print
# out the command that can be used to SSH into the cluster head node.
$ ray create_or_update ray/python/ray/autoscaler/gcp/example-full.yaml

# Reconfigure autoscaling behavior without interrupting running jobs
$ ray create_or_update ray/python/ray/autoscaler/gcp/example-full.yaml
  --max-workers=N --no-restart

# Teardown the cluster
$ ray teardown ray/python/ray/autoscaler/gcp/example-full.yaml
```
1.37.3 Port-forwarding applications

To run connect to applications running on the cluster (e.g. Jupyter notebook) using a web browser, you can forward the port to your local machine using SSH:

```
$ ssh -L 8899:localhost:8899 -i <key> <user>@<addr> 'source ~/anaconda3/bin/activate tensorflow_p36 && jupyter notebook --port=8899'
```

1.37.4 Updating your cluster

When you run `ray create_or_update` with an existing cluster, the command checks if the local configuration differs from the applied configuration of the cluster. This includes any changes to synced files specified in the `file_mounts` section of the config. If so, the new files and config will be uploaded to the cluster. Following that, Ray services will be restarted.

You can also run `ray create_or_update` to restart a cluster if it seems to be in a bad state (this will restart all Ray services even if there are no config changes).

If you don’t want the update to restart services (e.g. because the changes don’t require a restart), pass `--no-restart` to the update call.

1.37.5 Security

By default, the nodes will be launched into their own security group, with traffic allowed only between nodes in the same group. A new SSH key will also be created and saved to your local machine for access to the cluster.

1.37.6 Autoscaling

Ray clusters come with a load-based auto-scaler. When cluster resource usage exceeds a configurable threshold (80% by default), new nodes will be launched up the specified `max_workers` limit. When nodes are idle for more than a timeout, they will be removed, down to the `min_workers` limit. The head node is never removed.

The default idle timeout is 5 minutes. This is to prevent excessive node churn which could impact performance and increase costs (in AWS / GCP there is a minimum billing charge of 1 minute per instance, after which usage is billed by the second).

1.37.7 Monitoring cluster status

You can monitor cluster usage and auto-scaling status by tailing the autoscaling logs in `/tmp/raylogs/monitor-*`.

The Ray autoscaler also reports per-node status in the form of instance tags. In your cloud provider console, you can click on a Node, go the the “Tags” pane, and add the `ray-node-status` tag as a column. This lets you see per-node statuses at a glance:
1.37.8 Customizing cluster setup

You are encouraged to copy the example YAML file and modify it to your needs. This may include adding additional setup commands to install libraries or sync local data files.

**Note:** After you have customized the nodes, it is also a good idea to create a new machine image and use that in the config file. This reduces worker setup time, improving the efficiency of auto-scaling.

The setup commands you use should ideally be *idempotent*, that is, can be run more than once. This allows Ray to update nodes after they have been created. You can usually make commands idempotent with small modifications, e.g. `git clone foo` can be rewritten as `test -e foo || git clone foo` which checks if the repo is already cloned first.

Most of the example YAML file is optional. Here is a reference minimal YAML file, and you can find the defaults for optional fields in this YAML file.

1.37.9 Syncing git branches

A common use case is syncing a particular local git branch to all workers of the cluster. However, if you just put a `git checkout <branch>` in the setup commands, the autoscaler won’t know when to rerun the command to pull in updates. There is a nice workaround for this by including the git SHA in the input (the hash of the file will change if the branch is updated):

```yaml
file_mounts: {
    "/tmp/current_branch_sha": "/path/to/local/repo/.git/refs/heads/<YOUR_BRANCH_NAME>
}

setup_commands:
- test -e <REPO_NAME> || git clone https://github.com/<REPO_ORG>/<REPO_NAME>.git
- cd <REPO_NAME> && git fetch && git checkout `cat /tmp/current_branch_sha`
```

This tells `ray create_or_update` to sync the current git branch SHA from your personal computer to a temporary file on the cluster (assuming you’ve pushed the branch head already). Then, the setup commands read that file to figure out which SHA they should checkout on the nodes. Note that each command runs in its own session. The final workflow to update the cluster then becomes just this:

1. Make local changes to a git branch
2. Commit the changes with `git commit` and `git push`
3. Update files on your Ray cluster with `ray create_or_update`
1.37.10 Common cluster configurations

The `example-full.yaml` configuration is enough to get started with Ray, but for more compute intensive workloads you will want to change the instance types to e.g. use GPU or larger compute instance by editing the yaml file. Here are a few common configurations:

**GPU single node**: use Ray on a single large GPU instance.

```yaml
max_workers: 0
head_node:
  InstanceType: p2.8xlarge
```

**Docker**: Specify docker image. This executes all commands on all nodes in the docker container, and opens all the necessary ports to support the Ray cluster. This currently does not have GPU support.

```yaml
docker:
  image: tensorflow/tensorflow:1.5.0-py3
  container_name: ray_docker
```

**Mixed GPU and CPU nodes**: for RL applications that require proportionally more CPU than GPU resources, you can use additional CPU workers with a GPU head node.

```yaml
max_workers: 10
head_node:
  InstanceType: p2.8xlarge
worker_nodes:
  InstanceType: m4.16xlarge
```

**Autoscaling CPU cluster**: use a small head node and have Ray auto-scale workers as needed. This can be a cost-efficient configuration for clusters with bursty workloads. You can also request spot workers for additional cost savings.

```yaml
min_workers: 0
max_workers: 10
head_node:
  InstanceType: m4.large
worker_nodes:
  InstanceMarketOptions:
    MarketType: spot
  InstanceType: m4.16xlarge
```

**Autoscaling GPU cluster**: similar to the autoscaling CPU cluster, but with GPU worker nodes instead.

```yaml
min_workers: 0
max_workers: 10
head_node:
  InstanceType: m4.large
worker_nodes:
  InstanceMarketOptions:
    MarketType: spot
  InstanceType: p2.xlarge
```

1.37.11 External Node Provider

Ray also supports external node providers (check `node_provider.py` implementation). You can specify the external node provider using the yaml config:
The module needs to be in the format `package.provider_class` or `package.sub_package.provider_class`.

### 1.37.12 Additional Cloud providers

To use Ray autoscaling on other Cloud providers or cluster management systems, you can implement the `NodeProvider` interface (~100 LOC) and register it in `node_provider.py`. Contributions are welcome!

### 1.38 Using Ray on a Cluster

#### Note: If you’re using AWS you can use the automated setup commands.

The instructions in this document work well for small clusters. For larger clusters, follow the instructions for managing a cluster with parallel ssh.

#### 1.38.1 Deploying Ray on a Cluster

This section assumes that you have a cluster running and that the nodes in the cluster can communicate with each other. It also assumes that Ray is installed on each machine. To install Ray, follow the installation instructions.

**Starting Ray on each machine**

On the head node (just choose some node to be the head node), run the following. If the `--redis-port` argument is omitted, Ray will choose a port at random.

```bash
ray start --head --redis-port=6379
```

The command will print out the address of the Redis server that was started (and some other address information).

Then on all of the other nodes, run the following. Make sure to replace `<redis-address>` with the value printed by the command on the head node (it should look something like `123.45.67.89:6379`).

```bash
ray start --redis-address=<redis-address>
```

If you wish to specify that a machine has 10 CPUs and 1 GPU, you can do this with the flags `--num-cpus=10` and `--num-gpus=1`. If these flags are not used, then Ray will detect the number of CPUs automatically and will assume there are 0 GPUs.

Now we’ve started all of the Ray processes on each node Ray. This includes:

- Some worker processes on each machine.
- An object store on each machine.
- A local scheduler on each machine.
- Multiple Redis servers (on the head node).
- One global scheduler (on the head node).
To run some commands, start up Python on one of the nodes in the cluster, and do the following.

```python
import ray
ray.init(redis_address="<redis-address>")
```

Now you can define remote functions and execute tasks. For example, to verify that the correct number of nodes have joined the cluster, you can run the following.

```python
import time

@ray.remote
def f():
    time.sleep(0.01)
    return ray.services.get_node_ip_address()

# Get a list of the IP addresses of the nodes that have joined the cluster.
set(ray.get([f.remote() for _ in range(1000)]))
```

**Stopping Ray**

When you want to stop the Ray processes, run `ray stop` on each node.

### 1.39 Using Ray on a Large Cluster

**Note:** If you’re using AWS you can use the automated setup commands.

Deploying Ray on a cluster requires a bit of manual work. The instructions here illustrate how to use parallel ssh commands to simplify the process of running commands and scripts on many machines simultaneously.

#### 1.39.1 Booting up a cluster on EC2

- Create an EC2 instance running Ray following the installation instructions.
  - Add any packages that you may need for running your application.
  - Install the pssh package: `sudo apt-get install pssh`.
- Create an AMI with Ray installed and with whatever code and libraries you want on the cluster.
- Use the EC2 console to launch additional instances using the AMI you created.
- Configure the instance security groups so that they machines can all communicate with one another.

#### 1.39.2 Deploying Ray on a Cluster

This section assumes that you have a cluster of machines running and that these nodes have network connectivity to one another. It also assumes that Ray is installed on each machine.

Additional assumptions:

- All of the following commands are run from a machine designated as the head node.
- The head node will run Redis and the global scheduler.
• The head node has ssh access to all other nodes.
• All nodes are accessible via ssh keys
• Ray is checked out on each node at the location $HOME/ray.

Note: The commands below will probably need to be customized for your specific setup.

Connect to the head node

In order to initiate ssh commands from the cluster head node we suggest enabling ssh agent forwarding. This will allow the session that you initiate with the head node to connect to other nodes in the cluster to run scripts on them. You can enable ssh forwarding by running the following command before connecting to the head node (replacing <ssh-key> with the path to the private key that you would use when logging in to the nodes in the cluster).

```bash
ssh-add <ssh-key>
```

Now log in to the head node with the following command, where <head-node-public-ip> is the public IP address of the head node (just choose one of the nodes to be the head node).

```bash
ssh -A ubuntu@<head-node-public-ip>
```

Build a list of node IP addresses

On the head node, populate a file workers.txt with one IP address on each line. Do not include the head node IP address in this file. These IP addresses should typically be private network IP addresses, but any IP addresses which the head node can use to ssh to worker nodes will work here. This should look something like the following.

```
172.31.27.16
172.31.29.173
172.31.24.132
172.31.29.224
```

Confirm that you can ssh to all nodes

```bash
for host in $(cat workers.txt); do
  ssh -o "StrictHostKeyChecking no" $host uptime
done
```

You may need to verify the host keys during this process. If so, run this step again to verify that it worked. If you see a permission denied error, you most likely forgot to run ssh-add <ssh-key> before connecting to the head node.

Starting Ray

Start Ray on the head node

On the head node, run the following:

```bash
ray start --head --redis-port=6379
```

Start Ray on the worker nodes

Create a file start_worker.sh that contains something like the following:
This script, when run on the worker nodes, will start up Ray. You will need to replace `<head-node-ip>` with the IP address that worker nodes will use to connect to the head node (most likely a **private IP address**). In this example we also export the path to the Python installation since our remote commands will not be executing in a login shell.

**Warning:** You will probably need to manually export the correct path to Python (you will need to change the first line of `start_worker.sh` to find the version of Python that Ray was built against). This is necessary because the `PATH` environment variable used by `parallel-ssh` can differ from the `PATH` environment variable that gets set when you `ssh` to the machine.

**Warning:** If the `parallel-ssh` command below appears to hang or otherwise fails, `head-node-ip` may need to be a private IP address instead of a public IP address (e.g., if you are using EC2). It’s also possible that you forgot to run `ssh-add <ssh-key>` or that you forgot the `-A` flag when connecting to the head node.

Now use `parallel-ssh` to start up Ray on each worker node.

```
parallel-ssh -h workers.txt -P -I < start_worker.sh
```

Note that on some distributions the `parallel-ssh` command may be called `pssh`.

### Verification

Now you have started all of the Ray processes on each node. These include:

- Some worker processes on each machine.
- An object store on each machine.
- A local scheduler on each machine.
- Multiple Redis servers (on the head node).
- One global scheduler (on the head node).

To confirm that the Ray cluster setup is working, start up Python on one of the nodes in the cluster and enter the following commands to connect to the Ray cluster.

```
import ray
ray.init(redis_address="<redis-address>")
```

Here `<redis-address>` should have the form `<head-node-ip>:6379`.

Now you can define remote functions and execute tasks. For example, to verify that the correct number of nodes have joined the cluster, you can run the following.

```
import time

@ray.remote
def f():
    time.sleep(0.01)
    return ray.services.get_node_ip_address()

set(ray.get([f.remote() for _ in range(1000)]))
```

---

1.39. Using Ray on a Large Cluster
Ray Documentation, Release 0.5.0

Stopping Ray

Stop Ray on worker nodes

Create a file `stop_worker.sh` that contains something like the following:

```bash
# Make sure the SSH session has the correct version of Python on its path.
# You will probably have to change the line below.
export PATH=/home/ubuntu/anaconda3/bin/:$PATH
ray stop
```

This script, when run on the worker nodes, will stop Ray. Note, you will need to replace `/home/ubuntu/anaconda3/bin/` with the correct path to your Python installation.

Now use `parallel-ssh` to stop Ray on each worker node.

```
parallel-ssh -h workers.txt -P -I < stop_worker.sh
```

Stop Ray on the head node

```
ray stop
```

Upgrading Ray

Ray remains under active development so you may at times want to upgrade the cluster to take advantage of improvements and fixes.

Create an upgrade script

On the head node, create a file called `upgrade.sh` that contains the commands necessary to upgrade Ray. It should look something like the following:

```bash
# Make sure the SSH session has the correct version of Python on its path.
# You will probably have to change the line below.
export PATH=/home/ubuntu/anaconda3/bin/:$PATH
# Do pushd/popd to make sure we end up in the same directory.
pushd .
# Upgrade Ray.
cd ray
git checkout master
git pull
cd python
pip install -e . --verbose
popd
```

This script executes a series of git commands to update the Ray source code, then builds and installs Ray.

Stop Ray on the cluster

Follow the instructions for Stopping Ray.

Run the upgrade script on the cluster

First run the upgrade script on the head node. This will upgrade the head node and help confirm that the upgrade script is working properly.

```
bash upgrade.sh
```

Next run the upgrade script on the worker nodes.
parallel-ssh -h workers.txt -P -t 0 -I < upgrade.sh

Note here that we use the \(-t 0\) option to set the timeout to infinite. You may also want to use the \(-p\) flag, which controls the degree of parallelism used by parallel ssh.

It is probably a good idea to ssh to one of the other nodes and verify that the upgrade script ran as expected.

### 1.39.3 Sync Application Files to other nodes

If you are running an application that reads input files or uses python libraries then you may find it useful to copy a directory on the head node to the worker nodes.

You can do this using the `parallel-rsync` command:

```
parallel-rsync -h workers.txt -r <workload-dir> /home/ubuntu/<workload-dir>
```

where \(<workload-dir>\) is the directory you want to synchronize. Note that the destination argument for this command must represent an absolute path on the worker node.

### 1.39.4 Troubleshooting

#### Problems with parallel-ssh

If any of the above commands fail, verify that the head node has SSH access to the other nodes by running

```bash
for host in $(cat workers.txt); do
  ssh $host uptime
done
```

If you get a permission denied error, then make sure you have SSH'ed to the head node with agent forwarding enabled. This is done as follows.

```
ssh-add <ssh-key>
ssh -A ubuntu@<head-node-public-ip>
```

### 1.40 Using Ray and Docker on a Cluster (EXPERIMENTAL)

Packaging and deploying an application using Docker can provide certain advantages. It can make managing dependencies easier, help ensure that each cluster node receives a uniform configuration, and facilitate swapping hardware resources between applications.

#### 1.40.1 Create your Docker image

First build a Ray Docker image by following the instructions for Installation on Docker. This will allow you to create the `ray-project/deploy` image that serves as a basis for using Ray on a cluster with Docker.

Docker images encapsulate the system state that will be used to run nodes in the cluster. We recommend building on top of the Ray-provided Docker images to add your application code and dependencies.

You can do this in one of two ways: by building from a customized Dockerfile or by saving an image after entering commands manually into a running container. We describe both approaches below.
Creating a customized Dockerfile

We recommend that you read the official Docker documentation for Building your own image ahead of starting this section. Your customized Dockerfile is a script of commands needed to set up your application, possibly packaged in a folder with related resources.

A simple template Dockerfile for a Ray application looks like this:

```
# Application Dockerfile template
FROM ray-project/deploy
RUN git clone <my-project-url>
RUN <my-project-installation-script>
```

This file instructs Docker to load the image tagged `ray-project/deploy`, check out the git repository at `<my-project-url>`, and then run the script `<my-project-installation-script>`.

Build the image by running something like:

```
docker build -t <my-app> .
```

Replace `<app-tag>` with a tag of your choice.

Creating a Docker image manually

Launch the `ray-project/deploy` image interactively

```
docker run -t -i ray-project/deploy
```

Next, run whatever commands are needed to install your application. When you are finished type `exit` to stop the container.

Run

```
docker ps -a
```

to identify the id of the container you just exited.

Next, commit the container

```
docker commit -t <app-tag> <container-id>
```

Replace `<app-tag>` with a name for your container and replace `<container-id>` id with the hash id of the container used in configuration.

1.40.2 Publishing your Docker image to a repository

When using Amazon EC2 it can be practical to publish images using the Repositories feature of Elastic Container Service. Follow the steps below and see documentation for creating a repository for additional context.

First ensure that the AWS command-line interface is installed.

```
sudo apt-get install -y awscli
```

Next create a repository in Amazon’s Elastic Container Registry. This results in a shared resource for storing Docker images that will be accessible from all nodes.
aws ecr create-repository --repository-name <repository-name> --region=<region>

Replace <repository-name> with a string describing the application. Replace <region> with the AWS region string, e.g., us-west-2. This should produce output like the following:

```json
{
    "repository": {
        "repositoryUri": "123456789012.dkr.ecr.us-west-2.amazonaws.com/my-app",
        "createdAt": 1487227244.0,
        "repositoryArn": "arn:aws:ecr:us-west-2:123456789012:repository/my-app",
        "registryId": "123456789012",
        "repositoryName": "my-app"
    }
}
```

Take note of the repositoryUri string, in this example 123456789012.dkr.ecr.us-west-2.amazonaws.com/my-app.

Tag the Docker image with the repository URI.

```bash
docker tag <app-tag> <repository-uri>
```

Replace the <app-tag> with the container name used previously and replace <repository-uri> with URI returned by the command used to create the repository.

Log into the repository:

```bash
eval $(aws ecr get-login --region <region>)
```

Replace <region> with your selected AWS region.

Push the image to the repository:

```bash
docker push <repository-uri>
```

Replace <repository-uri> with the URI of your repository. Now other hosts will be able to access your application Docker image.

### 1.40.3 Starting a cluster

We assume a cluster configuration like that described in instructions for using Ray on a large cluster. In particular, we assume that there is a head node that has ssh access to all of the worker nodes, and that there is a file workers.txt listing the IP addresses of all worker nodes.

**Install the Docker image on all nodes**

Create a script called setup-docker.sh on the head node.

```bash
# setup-docker.sh
sudo apt-get install -y docker.io
sudo service docker start
sudo usermod -a -G docker ubuntu
exec sudo su -l ubuntu
eval $(aws ecr get-login --region <region>)
docker pull <repository-uri>
```

1.40. Using Ray and Docker on a Cluster (EXPERIMENTAL)
Replace `<repository-uri>` with the URI of the repository created in the previous section. Replace `<region>` with the AWS region in which you created that repository. This script will install Docker, authenticate the session with the container registry, and download the container image from that registry.

Run `setup-docker.sh` on the head node (if you used the head node to build the Docker image then you can skip this step):

```
bash setup-docker.sh
```

Run `setup-docker.sh` on the worker nodes:

```
parallel-ssh -h workers.txt -P -t 0 -I < setup-docker.sh
```

### Launch Ray cluster using Docker

To start Ray on the head node run the following command:

```
eval $(aws ecr get-login --region <region>)
docker run -d -- shm-size=<shm-size> --net=host <repository-uri> 
  ray start --head 
  --object-manager-port=8076 
  --redis-port=6379 
  --num-workers=<num-workers>
```

Replace `<repository-uri>` with the URI of the repository. Replace `<region>` with the region of the repository. Replace `<num-workers>` with the number of workers, e.g., typically a number similar to the number of cores in the system. Replace `<shm-size>` with the the amount of shared memory to make available within the Docker container, e.g., 8G.

To start Ray on the worker nodes create a script `start-worker-docker.sh` with content like the following:

```
eval $(aws ecr get-login --region <region>)
docker run -d -- shm-size=<shm-size> --net=host <repository-uri> 
  ray start 
  --object-manager-port=8076 
  --redis-address=<redis-address> 
  --num-workers=<num-workers>
```

Replace `<redis-address>` with the string `<head-node-private-ip>:6379` where `<head-node-private-ip>` is the private network IP address of the head node.

Execute the script on the worker nodes:

```
parallel-ssh -h workers.txt -P -t 0 -I < setup-worker-docker.sh
```

### 1.40.4 Running jobs on a cluster

On the head node, identify the id of the container that you launched as the Ray head.

```
docker ps
```

the container id appears in the first column of the output.

Now launch an interactive shell within the container:
docker exec -t -i <container-id> bash

Replace <container-id> with the container id found in the previous step.

Next, launch your application program. The Python program should contain an initialization command that takes the Redis address as a parameter:

```python
ray.init(redis_address="<redis-address>")
```

### 1.40.5 Shutting down a cluster

Kill all running Docker images on the worker nodes:

```
parallel-ssh -h workers.txt -P 'docker kill $(docker ps -q)'
```

Kill all running Docker images on the head node:

```
docker kill $(docker ps -q)
```

### 1.41 Troubleshooting

This document discusses some common problems that people run into when using Ray as well as some known problems. If you encounter other problems, please let us know.

#### 1.41.1 No Speedup

You just ran an application using Ray, but it wasn’t as fast as you expected it to be. Or worse, perhaps it was slower than the serial version of the application! The most common reasons are the following.

- **Number of cores**: How many cores is Ray using? When you start Ray, it will determine the number of CPUs on each machine with `psutil.cpu_count()`. Ray usually will not schedule more tasks in parallel than the number of CPUs. So if the number of CPUs is 4, the most you should expect is a 4x speedup.

- **Physical versus logical CPUs**: Do the machines you’re running on have fewer physical cores than logical cores? You can check the number of logical cores with `psutil.cpu_count()` and the number of physical cores with `psutil.cpu_count(logical=False)`. This is common on a lot of machines and especially on EC2. For many workloads (especially numerical workloads), you often cannot expect a greater speedup than the number of physical CPUs.

- **Small tasks**: Are your tasks very small? Ray introduces some overhead for each task (the amount of overhead depends on the arguments that are passed in). You will be unlikely to see speedups if your tasks take less than ten milliseconds. For many workloads, you can easily increase the sizes of your tasks by batching them together.

- **Variable durations**: Do your tasks have variable duration? If you run 10 tasks with variable duration in parallel, you shouldn’t expect an N-fold speedup (because you’ll end up waiting for the slowest task). In this case, consider using `ray.wait` to begin processing tasks that finish first.

- **Multi-threaded libraries**: Are all of your tasks attempting to use all of the cores on the machine? If so, they are likely to experience contention and prevent your application from achieving a speedup. You can diagnose this by opening `top` while your application is running. If one process is using most of the CPUs, and the others are using a small amount, this may be the problem. This is very common with some versions of `numpy`, and in that case can usually be setting an environment variable like `MKL_NUM_THREADS` (or the equivalent depending on your installation) to 1.
If you are still experiencing a slowdown, but none of the above problems apply, we’d really like to know! Please create a GitHub issue and consider submitting a minimal code example that demonstrates the problem.

### 1.41.2 Crashes

If Ray crashed, you may wonder what happened. Currently, this can occur for some of the following reasons.

- **Stressful workloads:** Workloads that create many many tasks in a short amount of time can sometimes interfere with the heartbeat mechanism that we use to check that processes are still alive. On the head node in the cluster, you can check the files `/tmp/raylogs/monitor-******.out` and `/tmp/raylogs/monitor-******.err`. They will indicate which processes Ray has marked as dead (due to a lack of heartbeats). However, it is currently possible for a process to get marked as dead without actually having died.

- **Starting many actors:** Workloads that start a large number of actors all at once may exhibit problems when the processes (or libraries that they use) contend for resources. Similarly, a script that starts many actors over the lifetime of the application will eventually cause the system to run out of file descriptors. This is addressable, but currently we do not garbage collect actor processes until the script finishes.

- **Running out of file descriptors:** As a workaround, you may be able to increase the maximum number of file descriptors with a command like `ulimit -n 65536`. If that fails, double check that the hard limit is sufficiently large by running `ulimit -Hn`. If it is too small, you can increase the hard limit as follows (these instructions work on EC2).
  
  - Increase the hard ulimit for open file descriptors system-wide by running the following.

    ```bash
    sudo bash -c "echo $USER hard nofile 65536 >> /etc/security/limits.conf"
    ```

  - Logout and log back in.

### 1.41.3 Hanging

If a workload is hanging and not progressing, the problem may be one of the following.

- **Reconstructing an object created with put:** When an object that is needed has been evicted or lost, Ray will attempt to rerun the task that created the object. However, there are some cases that currently are not handled. For example, if the object was created by a call to `ray.put` on the driver process, then the argument that was passed into `ray.put` is no longer available and so the call to `ray.put` cannot be rerun (without rerunning the driver).

- **Reconstructing an object created by actor task:** Ray currently does not reconstruct objects created by actor methods.

### 1.41.4 Serialization Problems

Ray’s serialization is currently imperfect. If you encounter an object that Ray does not serialize/deserialize correctly, please let us know. For example, you may want to bring it up on this thread.

- Objects with multiple references to the same object.

- Subtypes of lists, dictionaries, or tuples.

### 1.41.5 Outdated Function Definitions

Due to subtleties of Python, if you redefine a remote function, you may not always get the expected behavior. In this case, it may be that Ray is not running the newest version of the function.
Suppose you define a remote function \( f \) and then redefine it. Ray should use the newest version.

```python
@ray.remote
def f():
    return 1

@ray.remote
def f():
    return 2

ray.get(f.remote())  # This should be 2.
```

However, the following are cases where modifying the remote function will not update Ray to the new version (at least without stopping and restarting Ray).

- **The function is imported from an external file:** In this case, \( f \) is defined in some external file `file.py`. If you import `file`, change the definition of \( f \) in `file.py`, then re-import `file`, the function \( f \) will not be updated.

  This is because the second import gets ignored as a no-op, so \( f \) is still defined by the first import.

  A solution to this problem is to use `reload(file)` instead of a second import `file`. Reloading causes the new definition of \( f \) to be re-executed, and exports it to the other machines. Note that in Python 3, you need to do `from importlib import reload`.

- **The function relies on a helper function from an external file:** In this case, \( f \) can be defined within your Ray application, but relies on a helper function \( h \) defined in some external file `file.py`. If the definition of \( h \) gets changed in `file.py`, redefining \( f \) will not update Ray to use the new version of \( h \).

  This is because when \( f \) first gets defined, its definition is shipped to all of the workers, and is unpickled. During unpickling, `file.py` gets imported in the workers. Then when \( f \) gets redefined, its definition is again shipped and unpickled in all of the workers. But since `file.py` has been imported in the workers already, it is treated as a second import and is ignored as a no-op.

  Unfortunately, reloading on the driver does not update \( h \), as the reload needs to happen on the worker.

  A solution to this problem is to redefine \( f \) to reload `file.py` before it calls \( h \). For example, if inside `file.py` you have

  ```python
def h():
    return 1
```

  And you define remote function \( f \) as

  ```python
@ray.remote
def f():
    return file.h()
```

  You can redefine \( f \) as follows.

  ```python
@ray.remote
def f():
    reload(file)
    return file.h()
```

  This forces the reload to happen on the workers as needed. Note that in Python 3, you need to do `from importlib import reload`. 

---

**1.41. Troubleshooting**
1.42 Profiling for Ray Users

This document is intended for users of Ray who want to know how to evaluate the performance of their code while running on Ray. Profiling the performance of your code can be very helpful to determine performance bottlenecks or to find out where your code may not be parallelized properly. If you are interested in pinpointing why your Ray application may not be achieving the expected speedup, read on!

1.42.1 A Basic Example to Profile

Let’s try to profile a simple example, and compare how different ways to write a simple loop can affect performance.

As a proxy for a computationally intensive and possibly slower function, let’s define our remote function to just sleep for 0.5 seconds:

```
import ray
import time

# Our time-consuming remote function
@ray.remote
def func():
    time.sleep(0.5)
```

In our example setup, we wish to call our remote function `func()` five times, and store the result of each call into a list. To compare the performance of different ways of looping our calls to our remote function, we can define each loop version as a separate function on the driver script.

For the first version `ex1`, each iteration of the loop calls the remote function, then calls `ray.get` in an attempt to store the current result into the list, as follows:

```
# This loop is suboptimal in Ray, and should only be used for the sake of this example
def ex1():
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))
```

For the second version `ex2`, each iteration of the loop calls the remote function, and stores it into the list without calling `ray.get` each time. `ray.get` is used after the loop has finished, in preparation for processing `func()`’s results:

```
# This loop is more proper in Ray
def ex2():
    list2 = []
    for i in range(5):
        list2.append(func.remote())
    ray.get(list2)
```

Finally, for an example that’s not so parallelizable, let’s create a third version `ex3` where the driver has to call a local function in between each call to the remote function `func()`:

```
# A local function executed on the driver, not on Ray
def other_func():
    time.sleep(0.3)

def ex3():
    list3 = []
```

(continues on next page)
for i in range(5):
    other_func()
    list3.append(func.remote())
ray.get(list3)

1.42.2 Timing Performance Using Python’s Timestamps

One way to sanity-check the performance of the three loops is simply to time how long it takes to complete each loop version. We can do this using python’s built-in time module.

The time module contains a useful time() function that returns the current timestamp in unix time whenever it’s called. We can create a generic function wrapper to call time() right before and right after each loop function to print out how long each loop takes overall:

```python
# This is a generic wrapper for any driver function you want to time
def time_this(f):
    def timed_wrapper(*args, **kw):
        start_time = time.time()
        result = f(*args, **kw)
        end_time = time.time()

        # Time taken = end_time - start_time
        print('| func:%r args:[%r, %r] took: %2.4f seconds |' % (
            f.__name__, args, kw, end_time - start_time))
        return result
    return timed_wrapper
```

To always print out how long the loop takes to run each time the loop function `ex1()` is called, we can evoke our `time_this` wrapper with a function decorator. This can similarly be done to functions `ex2()` and `ex3()`:

```python
@time_this  # Added decorator
def ex1():
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))
```

Then, running the three timed loops should yield output similar to this:

```
| func:'ex1' args:[] took: 2.5083 seconds |
| func:'ex2' args:[] took: 1.0032 seconds |
| func:'ex3' args:[] took: 2.0039 seconds |
```

Let’s interpret these results.

Here, `ex1()` took substantially more time than `ex2()`, where their only difference is that `ex1()` calls `ray.get` on the remote function before adding it to the list, while `ex2()` waits to fetch the entire list with `ray.get` at once.

1.42. Profiling for Ray Users 135
@ray.remote
def func():  # A single call takes 0.5 seconds
time.sleep(0.5)

def ex1():  # Took Ray 2.5 seconds
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))

def ex2():  # Took Ray 1 second
    list2 = []
    for i in range(5):
        list2.append(func.remote())
    ray.get(list2)

Notice how ex1() took 2.5 seconds, exactly five times 0.5 seconds, or the time it would take to wait for our remote function five times in a row.

By calling ray.get after each call to the remote function, ex1() removes all ability to parallelize work, by forcing the driver to wait for each func()'s result in succession. We are not taking advantage of Ray parallelization here!

Meanwhile, ex2() takes about 1 second, much faster than it would normally take to call func() five times iteratively. Ray is running each call to func() in parallel, saving us time.

ex1() is actually a common user mistake in Ray. ray.get is not necessary to do before adding the result of func() to the list. Instead, the driver should send out all parallelizable calls to the remote function to Ray before waiting to receive their results with ray.get. ex1()’s suboptimal behavior can be noticed just using this simple timing test.

Realistically, however, many applications are not as highly parallelizable as ex2(), and the application includes sections where the code must run in serial. ex3() is such an example, where the local function other_func() must run first before each call to func() can be submitted to Ray.

# A local function that must run in serial
def other_func():
    time.sleep(0.3)

def ex3():  # Took Ray 2 seconds, vs. ex1 taking 2.5 seconds
    list3 = []
    for i in range(5):
        other_func()
        list2.append(func.remote())
    ray.get(list3)

What results is that while ex3() still gained 0.5 seconds of speedup compared to the completely serialized ex1() version, this speedup is still nowhere near the ideal speedup of ex2().

The dramatic speedup of ex2() is possible because ex2() is theoretically completely parallelizable: if we were given 5 CPUs, all 5 calls to func() can be run in parallel. What is happening with ex3(), however, is that each parallelized call to func() is staggered by a wait of 0.3 seconds for the local other_func() to finish.

ex3() is thus a manifestation of Amdahl’s Law: the fastest theoretically possible execution time from parallelizing an application is limited to be no better than the time it takes to run all serial parts in serial.

Due to Amdahl’s Law, ex3() must take at least 1.5 seconds – the time it takes for 5 serial calls to other_func() to finish! After an additional 0.5 seconds to execute func and get the result, the computation is done.
1.42.3 Profiling Using An External Profiler (Line Profiler)

One way to profile the performance of our code using Ray is to use a third-party profiler such as `Line_profiler`. `Line_profiler` is a useful line-by-line profiler for pure Python applications that formats its output side-by-side with the profiled code itself.

Alternatively, another third-party profiler (not covered in this documentation) that you could use is `Pyflame`, which can generate profiling graphs.

First install `line_profiler` with pip:

```bash
pip install line_profiler
```

`line_profiler` requires each section of driver code that you want to profile as its own independent function. Conveniently, we have already done so by defining each loop version as its own function. To tell `line_profiler` which functions to profile, just add the `@profile` decorator to `ex1()`, `ex2()` and `ex3()`. Note that you do not need to import `line_profiler` into your Ray application:

```python
@profile  # Added decorator
def ex1():
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))

def main():
    ray.init()
    ex1()
    ex2()
    ex3()

if __name__ == '__main__':
    main()
```

Then, when we want to execute our Python script from the command line, instead of `python your_script_here.py`, we use the following shell command to run the script with `line_profiler` enabled:

```bash
kernprof -l your_script_here.py
```

This command runs your script and prints only your script’s output as usual. `Line_profiler` instead outputs its profiling results to a corresponding binary file called `your_script_here.py.lprof`.

To read `line_profiler`'s results to terminal, use this shell command:

```bash
python -m line_profiler your_script_here.py.lprof
```

In our loop example, this command outputs results for `ex1()` as follows. Note that execution time is given in units of 1e-06 seconds:

```
Timer unit: 1e-06 s
Total time: 2.50883 s
File: your_script_here.py
Function: ex1 at line 28

<table>
<thead>
<tr>
<th>Line</th>
<th>Hits</th>
<th>Time Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td>@profile</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td>def ex1():</td>
</tr>
</tbody>
</table>
```

(continues on next page)
Notice that each hit to `list1.append(ray.get(func.remote()))` at line 33 takes the full 0.5 seconds waiting for `func()` to finish. Meanwhile, in `ex2()` below, each call of `func.remote()` at line 40 only takes 0.127 ms, and the majority of the time (about 1 second) is spent on waiting for `ray.get()` at the end:

```
Total time: 1.00357 s
File: your_script_here.py
Function: ex2 at line 35

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>1</td>
<td>2.0</td>
<td>2.0</td>
<td>0.0</td>
<td>@profile def ex2():</td>
</tr>
<tr>
<td>37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>2.0</td>
<td>2.0</td>
<td>0.0</td>
<td>list2 = []</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>13.0</td>
<td>2.2</td>
<td>0.0</td>
<td>for i in range(5):</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>637.0</td>
<td>127.4</td>
<td>0.1</td>
<td>list2.append(func.remote())</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>1002919.0</td>
<td>1002919.0</td>
<td>99.9</td>
<td>ray.get(list2)</td>
</tr>
</tbody>
</table>
```

And finally, `line_profiler`'s output for `ex3()`. Each call to `func.remote()` at line 50 still take magnitudes faster than 0.5 seconds, showing that Ray is successfully parallelizing the remote calls. However, each call to the local function `other_func()` takes the full 0.3 seconds, totalling up to the guaranteed minimum application execution time of 1.5 seconds:

```
Total time: 2.00446 s
File: basic_kernprof.py
Function: ex3 at line 44

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>1</td>
<td>2.0</td>
<td>2.0</td>
<td>0.0</td>
<td>@profile @time_this def ex3():</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td>2.0</td>
<td>2.0</td>
<td>0.0</td>
<td>list3 = []</td>
</tr>
<tr>
<td>48</td>
<td>6</td>
<td>13.0</td>
<td>2.2</td>
<td>0.0</td>
<td>for i in range(5):</td>
</tr>
<tr>
<td>49</td>
<td>5</td>
<td>1501934.0</td>
<td>300386.8</td>
<td>74.9</td>
<td>other_func()</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>917.0</td>
<td>183.4</td>
<td>0.0</td>
<td>list3.append(func.remote())</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>501589.0</td>
<td>501589.0</td>
<td>25.0</td>
<td>ray.get(list3)</td>
</tr>
</tbody>
</table>
```

### 1.42.4 Profiling Using Python’s CProfile

A second way to profile the performance of your Ray application is to use Python’s native `cProfile` profiling module. Rather than tracking line-by-line of your application code, `cProfile` can give the total runtime of each loop function, as well as list the number of calls made and execution time of all function calls made within the profiled code.

Unlike `line_profiler` above, this detailed list of profiled function calls includes internal function calls and function calls made within Ray!

However, similar to `line_profiler`, `cProfile` can be enabled with minimal changes to your application code (given that each section of the code you want to profile is defined as its own function). To use `cProfile`, add an import statement, then replace calls to the loop functions as follows:
Now, when executing your Python script, a cProfile list of profiled function calls will be outputted to terminal for each call made to cProfile.run(). At the very top of cProfile’s output gives the total execution time for 'ex1()':

```
601 function calls (595 primitive calls) in 2.509 seconds
```

Following is a snippet of profiled function calls for 'ex1()'. Most of these calls are quick and take around 0.000 seconds, so the functions of interest are the ones with non-zero execution times:

```
ncalls tottime percall cumtime percall filename:lineno(function)
... 1  0.000   0.000  2.509  2.509 your_script_here.py:31(ex1)
 5  0.000   0.000   0.001   0.000 remote_function.py:103(remote)
 5  0.000   0.000   0.001   0.000 remote_function.py:107(_submit)
... 10  0.000   0.000   0.000   0.000 worker.py:2459(__init__)  
 5  0.000   0.000   2.508  0.502 worker.py:2535(get)
 5  0.000   0.000   0.000   0.000 worker.py:2695(get_global_worker)
10  0.000   0.000   2.507  0.251 worker.py:374(retrieve_and_deserialize)
 5  0.000   0.000   2.508  0.502 worker.py:424(get_object)
 5  0.000   0.000   0.000   0.000 worker.py:514(submit_task)
...  
```

The 5 separate calls to Ray’s get, taking the full 0.502 seconds each call, can be noticed at worker.py:2535(get). Meanwhile, the act of calling the remote function itself at remote_function.py:103(remote) only takes 0.001 seconds over 5 calls, and thus is not the source of the slow performance of ex1().

### Profiling Ray Actors with cProfile

Considering that the detailed output of cProfile can be quite different depending on what Ray functionalities we use, let us see what cProfile’s output might look like if our example involved Actors (for an introduction to Ray actors, see our Actor documentation here).

Now, instead of looping over five calls to a remote function like in `ex1`, let’s create a new example and loop over five calls to a remote function inside an actor. Our actor’s remote function again just sleeps for 0.5 seconds:

```
# Our actor
@ray.remote
class Sleeper(object):
    def __init__(self):
    (continues on next page)
```
self.sleepValue = 0.5

# Equivalent to func(), but defined within an actor
def actor_func(self):
    time.sleep(self.sleepValue)

Recalling the suboptimality of \texttt{ex1}, let’s first see what happens if we attempt to perform all five \texttt{actor_func()} calls within a single actor:

```python
def \texttt{ex4}():
    # This is suboptimal in Ray, and should only be used for the sake of this example
    actor_example = Sleeper.remote()

    five_results = []
    for \texttt{i} in range(5):
        five_results.append(actor_example.actor_func.remote())

    # Wait until the end to call \texttt{ray.get()}
    ray.get(five_results)
```

We enable \texttt{cProfile} on this example as follows:

```python
def \texttt{main}():
    ray.init()
    cProfile.run('\texttt{ex4()}')

if \_\_name\_ == "\_\_main\_":
    \texttt{main()}
```

Running our new Actor example, \texttt{cProfile}’s abbreviated output is as follows:

```
12519 function calls (11956 primitive calls) in 2.525 seconds

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
...  1   0.000   0.000    0.015   0.015 actor.py:546(remote)
  1   0.000   0.000    0.015   0.015 actor.py:560(_submit)
  1   0.000   0.000    0.000   0.000 actor.py:697(__init__)  
...  1   0.000   0.000    2.525    2.525 your_script_here.py:63(ex4)
...  9   0.000   0.000    0.000    0.000 worker.py:2459(__init__)  
  1   0.000   0.000    2.509    2.509 worker.py:2535(get)
  9   0.000   0.000    0.000    0.000 worker.py:2695(get_global_worker)
  4   0.000   0.000    2.508    0.627 worker.py:374(retrieve_and_deserialize)
  1   0.000   0.000    2.509    2.509 worker.py:424(get_object)
  8   0.000   0.000    0.001    0.000 worker.py:514(submit_task)
...```

It turns out that the entire example still took 2.5 seconds to execute, or the time for five calls to \texttt{actor_func()} to run in serial. We remember in \texttt{ex1} that this behavior was because we did not wait until after submitting all five remote function tasks to call \texttt{ray.get()}, but we can verify on \texttt{cProfile}’s output line \texttt{worker.py:2535(get)} that \texttt{ray.get()} was only called once at the end, for 2.509 seconds. What happened?

It turns out Ray cannot parallelize this example, because we have only initialized a single \texttt{Sleeper} actor. Because each actor is a single, stateful worker, our entire code is submitted and ran on a single worker the whole time.

To better parallelize the actors in \texttt{ex4}, we can take advantage that each call to \texttt{actor_func()} is independent, and
instead create five Sleeper actors. That way, we are creating five workers that can run in parallel, instead of creating a single worker that can only handle one call to \texttt{actor\_func()} at a time.

```python
def ex4():
    # Modified to create five separate Sleepers
    five_actors = [Sleeper.remote() for i in range(5)]

    # Each call to actor\_func now goes to a different Sleeper
    five_results = []
    for actor_example in five_actors:
        five_results.append(actor_example.actor\_func.remote())

    ray.get(five_results)
```

Our example in total now takes only 1.5 seconds to run:

```
1378 function calls (1363 primitive calls) in 1.567 seconds

ncalls tottime percall cumtime percall filename:lineno(function)
...
5  0.000  0.000  0.002  0.000 actor.py:546(remote)
5  0.000  0.000  0.002  0.000 actor.py:560(_submit)
5  0.000  0.000  0.000  0.000 actor.py:697(__init__)
...
1  0.000  0.000  1.566  1.566 your_script_here.py:71(ex4)
...
21  0.000  0.000  0.000  0.000 worker.py:2459(__init__)
1  0.000  0.000  1.564  1.564 worker.py:2535(get)
25  0.000  0.000  0.000  0.000 worker.py:2695(get\_global\_worker)
3  0.000  0.000  1.564  0.521 worker.py:374(retrieve\_and\_deserialize)
1  0.000  0.000  1.564  1.564 worker.py:424(get\_object)
20  0.001  0.000  0.001  0.000 worker.py:514(submit\_task)
...
```

### 1.42.5 Visualizing Tasks in the Ray Timeline

Profiling the performance of your Ray application doesn’t need to be an eye-straining endeavor of interpreting numbers among hundreds of lines of text. Ray comes with its own visual web UI to visualize the parallelization (or lack thereof) of user tasks submitted to Ray!

This method does have its own limitations, however. The Ray Timeline can only show timing info about Ray tasks, and not timing for normal Python functions. This can be an issue especially for debugging slow Python code that is running on the driver, and not running as a task on one of the workers. The other profiling techniques above are options that do cover profiling normal Python functions.

Currently, whenever initializing Ray, a URL is generated and printed in the terminal. This URL can be used to view Ray’s web UI as a Jupyter notebook:

```
~$: python your_script_here.py

Process STDOUT and STDERR is being redirected to /tmp/raylogs/.
Waiting for redis server at 127.0.0.1:61150 to respond...
Waiting for redis server at 127.0.0.1:21607 to respond...
Starting local scheduler with the following resources: ('CPU': 4, 'GPU': 0).
```

(continues on next page)
Ray’s web UI attempts to run on localhost at port 8888, and if it fails it tries successive ports until it finds an open port. In this above example, it has opened on port 8897.

Because this web UI is only available as long as your Ray application is currently running, you may need to add a user prompt to prevent your Ray application from exiting once it has finished executing, such as below. You can then browse the web UI for as long as you like:

```python
def main():
    ray.init()
    ex1()
    ex2()
    ex3()

    # Require user input confirmation before exiting
    hang = input('Examples finished executing. Press enter to exit:')

if __name__ == '__main__':
    main()
```

Now, when executing your python script, you can access the Ray timeline by copying the web UI URL into your web browser on the Ray machine. To load the web UI in the jupyter notebook, select Kernel -> Restart and Run All in the jupyter menu.

The Ray timeline can be viewed in the fourth cell of the UI notebook by using the task filter options, then clicking on the View task timeline button.

For example, here are the results of executing `ex1()`, `ex2()`, and `ex3()` visualized in the Ray timeline. Each red block is a call to one of our user-defined remote functions, namely `func()`, which sleeps for 0.5 seconds:

(highlighted color boxes for `ex1()`, `ex2()`, and `ex3()` added for the sake of this example)

Note how `ex1()` executes all five calls to `func()` in serial, while `ex2()` and `ex3()` are able to parallelize their remote function calls.

Because we have 4 CPUs available on our machine, we can only able to execute up to 4 remote functions in parallel. So, the fifth call to the remote function in `ex2()` must wait until the first batch of `func()` calls is finished.

In `ex3()`, because of the serial dependency on `other_func()`, we aren’t even able to use all 4 of our cores to parallelize calls to `func()`. The time gaps between the `func()` blocks are a result of staggering the calls to `func()` in between waiting 0.3 seconds for `other_func()`.

Also, notice that due to the aforementioned limitation of the Ray timeline, `other_func()`, as a driver function and not a Ray task, is never visualized on the Ray timeline.

For more on Ray’s Web UI, such as how to access the UI on a remote node over ssh, or for troubleshooting installation, please see our Web UI documentation section.

### 1.43 Development Tips

If you are doing development on the Ray codebase, the following tips may be helpful.

1. **Speeding up compilation:** Be sure to install Ray with
cd ray/python
pip install -e . --verbose

The `-e` means “editable”, so changes you make to files in the Ray directory will take effect without reinstalling the package. In contrast, if you do `python setup.py install`, files will be copied from the Ray directory to a directory of Python packages (often something like `/home/ubuntu/anaconda3/lib/python3.6/site-packages/ray`). This means that changes you make to files in the Ray directory will not have any effect.

If you run into Permission Denied errors when running `pip install`, you can try adding `--user`. You may also need to run something like `sudo chown -R $USER /home/ubuntu/anaconda3` (substituting in the appropriate path).

If you make changes to the C++ files, you will need to recompile them. However, you do not need to rerun `pip install -e .`. Instead, you can recompile much more quickly by doing

```
cd ray/build
make -j8
```

2. Starting processes in a debugger: When processes are crashing, it is often useful to start them in a debugger (gdb on Linux or lldb on MacOS). See the latest discussion about how to do this here.

3. Running tests locally: Suppose that one of the tests (e.g., `runtest.py`) is failing. You can run that test locally by running `python test/runtest.py`. However, doing so will run all of the tests which can take a while. To run a specific test that is failing, you can do

```
cd ray
python test/runtest.py APITest.testKeywordArgs
```

When running tests, usually only the first test failure matters. A single test failure often triggers the failure of subsequent tests in the same script.

4. Running linter locally: To run the Python linter on a specific file, run something like `flake8 ray/python/ray/worker.py`. You may need to first run `pip install flake8`.

5. Autoformatting code. We use yapf [https://github.com/google/yapf](https://github.com/google/yapf) for linting, and the config file is located at `.style.yapf`. We recommend running `scripts/yapf.sh` prior to pushing to format changed files. Note that some projects such as dataframes and rllib are currently excluded.

6. Inspecting Redis shards by hand: To inspect the primary Redis shard by hand, you can query it with commands like the following.

```
r_primary = ray.worker.global_worker.redis_client
r_primary.keys("*")
```

To inspect other Redis shards, you will need to create a new Redis client. For example (assuming the relevant IP address is `127.0.0.1` and the relevant port is `1234`), you can do this as follows.

```
import redis
r = redis.StrictRedis(host='127.0.0.1', port=1234)
```

You can find a list of the relevant IP addresses and ports by running

```
r_primary.lrangi('RedisShards', 0, -1)
```
1.44 Profiling for Ray Developers

This document details, for Ray developers, how to use `pprof` to profile Ray binaries.

1.44.1 Installation

These instructions are for Ubuntu only. Attempts to get `pprof` to correctly symbolize on Mac OS have failed.

```
sudo apt-get install google-perftools libgoogle-perftools-dev
```

1.44.2 Changes to compilation and linking

Let’s say we want to profile the `plasma_manager`. Change the link instruction in `src/plasma/CMakeLists.txt` from

```
target_link_libraries(plasma_manager common $(PLASMA_STATIC_LIB) ray_static $(ARROW_STATIC_LIB) -lpthread)
```

to additionally include `-lprofiler`:

```
target_link_libraries(plasma_manager common $(PLASMA_STATIC_LIB) ray_static $(ARROW_STATIC_LIB) -lpthread -lprofiler)
```

Additionally, add `-g -ggdb` to `CMAKE_C_FLAGS` and `CMAKE_CXX_FLAGS` to enable the debug symbols. (Keeping `-O3` seems okay.)

Recompile.

1.44.3 Launching the to-profile binary

In various places, instead of launching the target binary via `plasma_manager <args>`, it must be launched with

```
LD_PRELOAD=/usr/lib/libprofiler.so CPUPROFILE=/tmp/pprof.out plasma_manager <args>
```

In practice, this means modifying `python/ray/plasma/plasma.py` so that the manager is launched with a command that passes a `modified_env` into `Popen`.

```
modified_env = os.environ.copy()
modified_env["LD_PRELOAD"] = "/usr/lib/libprofiler.so"
modified_env["CPUPROFILE"] = "/tmp/pprof.out"

process = subprocess.Popen(command,
    stdout=stdout_file,
    stderr=stderr_file,
    env=modified_env)
```

The file `/tmp/pprof.out` will be empty until you let the binary run the target workload for a while and then kill it.
1.44.4 Visualizing the CPU profile

The output of pprof can be visualized in many ways. Here we output it as a zoomable .svg image displaying the call graph annotated with hot paths.

```bash
# Use the appropriate path.
PLASMA_MANAGER=ray/python/ray/core/src/plasma/plasma_manager

google-pprof -svg $PLASMA_MANAGER /tmp/pprof.out > /tmp/pprof.svg
# Then open the .svg file with Chrome.

google-pprof -focus=epoll_wait -svg $PLASMA_MANAGER /tmp/pprof.out > /tmp/pprof.svg
```

Here’s a snapshot of an example svg output, taken from the official documentation:

1.44.5 References

- The pprof documentation.
- A Go version of pprof.
- The gperftools, including libprofiler, tcmalloc, and other goodies.

1.45 Contact

The following are good places to discuss Ray.

1. **Our Mailing List**: For discussions about development, questions about usage, or any general questions.
2. **GitHub Issues**: For bug reports and feature requests.
r

ray.rllib.agents, 59
ray.rllib.env, 61
ray.rllib.evaluation, 66
ray.rllib.models, 77
ray.rllib.optimizers, 81
ray.rllib.utils, 85
Symbols

--autoscaling-config <autoscaling_config>
  ray-start command line option, 18
--block
  ray-start command line option, 17
--head
  ray-start command line option, 17
--huge-pages
  ray-start command line option, 18
--max-workers <max_workers>
  ray-create_or_update command line option, 18
--min-workers <min_workers>
  ray-create_or_update command line option, 18
--no-restart
  ray-create_or_update command line option, 18
--no-ui
  ray-start command line option, 17
--node-ip-address <node_ip_address>
  ray-start command line option, 17
--num-cpus <num_cpus>
  ray-start command line option, 17
--num-gpus <num_gpus>
  ray-start command line option, 17
--num-redis-shards <num_redis_shards>
  ray-start command line option, 17
--num-workers <num_workers>
  ray-start command line option, 17
--object-manager-port <object_manager_port>
  ray-start command line option, 17
--object-store-memory <object_store_memory>
  ray-start command line option, 17
--plasma-directory <plasma_directory>
  ray-start command line option, 17
--redis-address <redis_address>
  ray-start command line option, 17
--redis-max-clients <redis_max_clients>
  ray-start command line option, 17
--redis-port <redis_port>
  ray-start command line option, 17
--redis-shard-ports <redis_shard_ports>
  ray-start command line option, 18
--resources <resources>
  ray-start command line option, 17
--use-raylet
  ray-start command line option, 18
-y, --yes
  ray-create_or_update command line option, 18
  ray-teardown command line option, 18
_restore() (ray.tune.trainable.Trainable method), 34
_save() (ray.tune.trainable.Trainable method), 34
_setup() (ray.tune.trainable.Trainable method), 35
_stop() (ray.tune.trainable.Trainable method), 35
_train() (ray.tune.trainable.Trainable method), 34

A

A3CAgent (class in ray.rllib.agents.a3c), 60
action_space (ray.rllib.env.VectorEnv attribute), 65
action_space (ray.rllib.evaluation.PolicyGraph attribute), 71
action_space (ray.rllib.evaluation.TFPolicyGraph attribute), 72
action_space (ray.rllib.evaluation.TorchPolicyGraph attribute), 74
ActionDistribution (class in ray.rllib.models), 77
add_batch() (ray.rllib.evaluation.SampleBatchBuilder method), 76
add_trial() (ray.rllib.evaluation.TuneClient method), 35
add_values() (ray.rllib.evaluation.MultiAgentSampleBatchBuilder method), 76
add_values() (ray.rllib.evaluation.SampleBatchBuilder method), 76
Agent (class in ray.rllib.agents), 59
ApexAgent (class in ray.rllib.agents.dqn), 60
ApexDDPGAgent (class in ray.rllib.agents.ddpg), 60
apply() (ray.rllib.evaluation.EvaluatorInterface method), 67
apply_changes() (ray.rllib.utils.Filter method), 85
apply_gradients() (ray.rllib.evaluation.EvaluatorInterface method), 66
apply_gradients() (ray.rllib.evaluation.PolicyEvaluator method), 70
apply_gradients() (ray.rllib.evaluation.PolicyGraph method), 71
apply_gradients() (ray.rllib.evaluation.TFPolicyGraph method), 73
apply_gradients() (ray.rllib.evaluation.TorchPolicyGraph method), 74
assignment_nodes (ray.experimental.TensorFlowVariables attribute), 107
AsyncGradientsOptimizer (class in ray.rllib.optimizers), 83
AsyncHyperBandScheduler (class in ray.tune.async_hyperband), 38
AsyncReplayOptimizer (class in ray.rllib.optimizers), 83
AsyncSampler (class in ray.rllib.evaluation), 77
AsyncSamplesOptimizer (class in ray.rllib.optimizers), 83
AsyncVectorEnv (class in ray.rllib.env), 61
build_and_reset() (ray.rllib.evaluation.MultiAgentSampleBatchBuilder method), 76
build_and_reset() (ray.rllib.evaluation.SampleBatchBuilder method), 76
Categorical (class in ray.rllib.models), 78
clear_buffer() (ray.rllib.utils.Filter method), 85
CLUSTER_CONFIG_FILE
    ray-create_or_update command line option, 18
    ray-get_head_ip command line option, 19
    ray-teardown command line option, 18
collect_metrics() (in module ray.rllib.evaluation), 77
collect_metrics() (ray.rllib.optimizers.PolicyOptimizer method), 82
columns() (ray.rllib.evaluation.SampleBatch method), 75
compute_action() (ray.rllib.agents.Agent method), 60
compute_action() (ray.rllib.evaluation.PolicyGraph method), 71
compute_action() (ray.rllib.evaluation.TFPolicyGraph method), 73
compute_action() (ray.rllib.evaluation.TorchPolicyGraph method), 74
compute_advantages() (in module ray.rllib.evaluation), 77
compute_apply() (ray.rllib.evaluation.EvaluatorInterface method), 67
compute_apply() (ray.rllib.evaluation.PolicyEvaluator method), 70
compute_apply() (ray.rllib.evaluation.PolicyGraph method), 72
compute_apply() (ray.rllib.evaluation.TFPolicyGraph method), 73
compute_gradients() (ray.rllib.evaluation.EvaluatorInterface method), 66
compute_gradients() (ray.rllib.evaluation.PolicyEvaluator method), 70
compute_gradients() (ray.rllib.evaluation.PolicyGraph method), 71
compute_gradients() (ray.rllib.evaluation.TFPolicyGraph method), 73
compute_gradients() (ray.rllib.evaluation.TorchPolicyGraph method), 74
calculate_single_action() (ray.rllib.evaluation.PolicyGraph method), 71
calculate_targets() (in module ray.rllib.evaluation), 77
collect() (ray.rllib.evaluation.PolicyOptimizer method), 75
collect() (ray.rllib.models.ActionDistribution method), 78
collect() (ray.rllib.models.Categorical method), 78
collect() (ray.rllib.models.DiagGaussian method), 78
collect() (ray.rllib.utils.PolicyClient method), 86
collect() (ray.rllib.utils.PolicyClient method), 86
collect() (ray.rllib.utils.PolicyClient method), 86
collect() (ray.rllib.utils.PolicyClient method), 86
config (ray.rllib.agents.Agent attribute), 60
config (ray.rllib.agents.Agent attribute), 60
config (ray.rllib.agents.Agent attribute), 60
copy() (ray.rllib.utils.Filter method), 85
copy() (ray.rllib.utils.Filter method), 85
count (ray.rllib.evaluation.MultiAgentBatch attribute), 76
C
DDPGAgent (class in ray.rllib.agents.ddpg), 60
deterministic (class in ray.rllib.models), 78
DiagGaussian (class in ray.rllib.models), 78
DDQNAgent (class in ray.rllib.agents.dqn), 60
E
end_episode() (ray.rllib.env.ServingEnv method), 65
end_episode() (ray.rllib.utils.policy_client.PolicyClient method), 86
entropy() (ray.rllib.models.ActionDistribution method), 78
entropy() (ray.rllib.models.Categorical method), 78
entropy() (ray.rllib.models.DiagGaussian method), 78
evaluation_resources (ray.rllib.optimizers.PolicyOptimizer attribute), 81
EnvContext (class in ray.rllib.env), 66
ESAgent (class in ray.rllib.agents.es), 61
Evaluators (class in ray.rllib.evaluation), 66
Experiment (class in ray.rllib.evaluation), 66
extra_action_out() (ray.rllib.evaluation.TorchPolicyGraph method), 74
F
Filter (class in ray.rllib.utils), 85
FilterManager (class in ray.rllib.utils), 85
for_policy() (ray.rllib.evaluation.PolicyEvaluator method), 69
foreach_evaluator() (ray.rllib.optimizers.PolicyOptimizer method), 82
foreach_evaluator_with_index() (ray.rllib.optimizers.PolicyOptimizer method), 82
<table>
<thead>
<tr>
<th>Class/Module</th>
<th>Class/Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>MedianStoppingRule</td>
<td>ray.tune.median_stopping_rule</td>
</tr>
<tr>
<td>Method()</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>ModelCatalog</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>MultiAgentBatch</td>
<td>ray.rllib.evaluation</td>
</tr>
<tr>
<td>MultiAgentEnv</td>
<td>ray.rllib.env</td>
</tr>
<tr>
<td>MultiAgentSampleBatchBuilder</td>
<td>ray.rllib.evaluation</td>
</tr>
<tr>
<td>num_envs</td>
<td>ray.rllib.env.VectorEnv attribute</td>
</tr>
<tr>
<td>num_steps_sampled</td>
<td>ray.rllib.optimizers.PolicyOptimizer attribute</td>
</tr>
<tr>
<td>num_steps_trained</td>
<td>ray.rllib.optimizers.PolicyOptimizer attribute</td>
</tr>
<tr>
<td>observation_space</td>
<td>ray.rllib.env.VectorEnv attribute</td>
</tr>
<tr>
<td>observation_space</td>
<td>ray.rllib.evaluation.PolicyGraph attribute</td>
</tr>
<tr>
<td>observation_space</td>
<td>ray.rllib.evaluation.TFPolicyGraph attribute</td>
</tr>
<tr>
<td>observation_space</td>
<td>ray.rllib.evaluation.TorchPolicyGraph attribute</td>
</tr>
<tr>
<td>optimizer()</td>
<td>ray.rllib.evaluation.TorchPolicyGraph method</td>
</tr>
<tr>
<td>outputs</td>
<td>ray.rllib.models.Model attribute</td>
</tr>
<tr>
<td>PGAgent</td>
<td>ray.rllib.agents.pg</td>
</tr>
<tr>
<td>PolicyClient</td>
<td>ray.rllib.utils</td>
</tr>
<tr>
<td>PolicyClient</td>
<td>ray.rllib.utils.policy_client</td>
</tr>
<tr>
<td>PolicyEvaluator</td>
<td>ray.rllib.evaluation</td>
</tr>
<tr>
<td>PolicyGraph</td>
<td>ray.rllib.evaluation</td>
</tr>
<tr>
<td>PolicyOptimizer</td>
<td>ray.rllib.optimizers</td>
</tr>
<tr>
<td>PolicyServer</td>
<td>ray.rllib.utils</td>
</tr>
<tr>
<td>PolicyServer</td>
<td>ray.rllib.utils.policy_server</td>
</tr>
<tr>
<td>poll()</td>
<td>ray.rllib.env.AsyncVectorEnv method</td>
</tr>
<tr>
<td>PopulationBasedTraining</td>
<td>ray.tune.pbt</td>
</tr>
<tr>
<td>postprocess_batch_so_far()</td>
<td>ray.rllib.evaluation.MultiAgentSampleBatchBuilder method</td>
</tr>
<tr>
<td>postprocess_trajectory()</td>
<td>ray.rllib.evaluation.PolicyGraph method</td>
</tr>
<tr>
<td>PPOAgent</td>
<td>ray.rllib.agents.ppo</td>
</tr>
<tr>
<td>Preprocessor</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>profile()</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>put()</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>ray-create_or_update</td>
<td>command line option</td>
</tr>
<tr>
<td>–max-workers</td>
<td>ray.rllib.agents</td>
</tr>
<tr>
<td>–min-workers</td>
<td>ray.rllib.env</td>
</tr>
<tr>
<td>–no-restart</td>
<td>ray.rllib.evaluation</td>
</tr>
<tr>
<td>–y, –yes</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>ray-get_head_ip</td>
<td>command line option</td>
</tr>
<tr>
<td>ray-start</td>
<td>command line option</td>
</tr>
<tr>
<td>–autoscaling-config</td>
<td>ray.rllib.agents</td>
</tr>
<tr>
<td>–block</td>
<td>ray.rllib.env</td>
</tr>
<tr>
<td>–head</td>
<td>ray.rllib.evaluation</td>
</tr>
<tr>
<td>–huge-pages</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–no-ui</td>
<td>ray.rllib.optimizers</td>
</tr>
<tr>
<td>–node-ip-address</td>
<td>ray.rllib.utils</td>
</tr>
<tr>
<td>–num-cpus</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–num-gpus</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–num-redis-shards</td>
<td>ray.rllib.optimizers</td>
</tr>
<tr>
<td>–num-workers</td>
<td>ray.rllib.optimizers</td>
</tr>
<tr>
<td>–object-manager-port</td>
<td>ray.rllib.utils</td>
</tr>
<tr>
<td>–object-store-memory</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–plasma-directory</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–redis-address</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–redis-max-clients</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–redis-port</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–redis-shard-ports</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–resources</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>–use-raylet</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>ray-teardown</td>
<td>command line option</td>
</tr>
<tr>
<td>–y, –yes</td>
<td>ray.rllib.models</td>
</tr>
<tr>
<td>resource_help()</td>
<td>ray.rllib.agents</td>
</tr>
<tr>
<td>register_custom_model()</td>
<td>ray.rllib.models.ModelCatalog static method</td>
</tr>
<tr>
<td>register_custom_preprocessor()</td>
<td>ray.rllib.models.ModelCatalog static method</td>
</tr>
<tr>
<td>register_custom_serializer()</td>
<td>(in module ray)</td>
</tr>
<tr>
<td>register_trainable()</td>
<td>(in module ray)</td>
</tr>
<tr>
<td>remote()</td>
<td>ray.rllib.optimizers.PolicyOptimizer attribute</td>
</tr>
<tr>
<td>remote_evaluators</td>
<td>ray.rllib.optimizers.PolicyOptimizer method</td>
</tr>
<tr>
<td>reset()</td>
<td>ray.rllib.env.MultiAgentEnv method</td>
</tr>
<tr>
<td>reset_at()</td>
<td>ray.rllib.env.VectorEnv method</td>
</tr>
<tr>
<td>resource_help()</td>
<td>ray.rllib.agents.Agent class method</td>
</tr>
<tr>
<td>restore()</td>
<td>ray.rllib.optimizers.PolicyOptimizer method</td>
</tr>
<tr>
<td>rows()</td>
<td>ray.rllib.evaluation.SampleBatch method</td>
</tr>
</tbody>
</table>
run() (ray.rllib.env.ServingEnv method), 64
run() (ray.rllib.evaluation.AsyncSampler method), 77
run_experiments() (in module ray.tune), 30

S
sample() (ray.rllib.evaluation.EvaluatorInterface method), 66
sample() (ray.rllib.evaluation.PolicyEvaluator method), 69
sample() (ray.rllib.models.ActionDistribution method), 78
sample() (ray.rllib.models.Categorical method), 78
sample() (ray.rllib.models.DiagGaussian method), 78
sample_with_count() (ray.rllib.evaluation.PolicyEvaluator method), 69
SampleBatch (class in ray.rllib.evaluation), 75
SampleBatchBuilder (class in ray.rllib.evaluation), 76
save() (ray.rllib.optimizers.PolicyOptimizer method), 82
SearchAlgorithm (class in ray.tune.suggest), 30
send_actions() (ray.rllib.env.AsyncVectorEnv method), 62
seq_lens (ray.rllib.models.Model attribute), 81
ServingEnv (class in ray.rllib.env), 63
sess (ray.experimental.TensorFlowVariables attribute), 107
set_flat() (ray.experimental.TensorFlowVariables method), 107
set_session() (ray.experimental.TensorFlowVariables method), 107
set_state() (ray.rllib.evaluation.PolicyGraph method), 72
set_weights() (ray.experimental.TensorFlowVariables method), 107
set_weights() (ray.rllib.agents.Agent method), 60
set_weights() (ray.rllib.evaluation.EvaluatorInterface method), 67
set_weights() (ray.rllib.evaluation.PolicyEvaluator method), 69
set_weights() (ray.rllib.evaluation.PolicyGraph method), 72
set_weights() (ray.rllib.evaluation.TFPolicyGraph method), 74
set_weights() (ray.rllib.evaluation.TorchPolicyGraph method), 75
shape (ray.rllib.models.Preprocessor attribute), 81
shutdown() (in module ray), 15
start_episode() (ray.rllib.env.ServingEnv method), 64
start_episode() (ray.rllib.utils.policy_client.PolicyClient method), 48
start_episode() (ray.rllib.utils.PolicyClient method), 85
state_in (ray.rllib.models.Model attribute), 80
state_init (ray.rllib.models.Model attribute), 80
state_out (ray.rllib.models.Model attribute), 80
stats() (ray.rllib.optimizers.AsyncGradientsOptimizer method), 84
stats() (ray.rllib.optimizers.AsyncReplayOptimizer method), 83
stats() (ray.rllib.optimizers.AsyncSamplesOptimizer method), 83
stats() (ray.rllib.optimizers.LocalMultiGPUOptimizer method), 85
stats() (ray.rllib.optimizers.PolicyOptimizer method), 82
stats() (ray.rllib.optimizers.PolicyOptimizer method), 84
stats() (ray.rllib.optimizers.SyncReplayOptimizer method), 84
stats() (ray.rllib.optimizers.SyncSamplesOptimizer method), 84
step() (ray.rllib.env.MultiAgentEnv method), 63
step() (ray.rllib.optimizers.AsyncGradientsOptimizer method), 83
step() (ray.rllib.optimizers.AsyncReplayOptimizer method), 83
step() (ray.rllib.optimizers.AsyncSamplesOptimizer method), 83
step() (ray.rllib.optimizers.LocalMultiGPUOptimizer method), 84
step() (ray.rllib.optimizers.PolicyOptimizer method), 82
step() (ray.rllib.optimizers.SyncReplayOptimizer method), 84
step() (ray.rllib.optimizers.SyncSamplesOptimizer method), 84
step() (ray.rllib.optimizers.PolicyOptimizer method), 84
step() (ray.rllib.optimizers.PolicyOptimizer method), 84
step() (ray.rllib.optimizers.PolicyOptimizer method), 84
stop_trial() (ray.tune.web_server.TuneClient method), 35
sync() (ray.rllib.utils.Filter method), 85
sync_filters() (ray.rllib.evaluation.PolicyEvaluator method), 69
synchronize() (ray.rllib.utils.FilterManager static method), 85
SyncReplayOptimizer (class in ray.rllib.optimizers), 84
SyncSampler (class in ray.rllib.evaluation), 76
SyncSamplesOptimizer (class in ray.rllib.optimizers), 84
T
TensorFlowVariables (class in ray.experimental), 107
TFPolicyGraph (class in ray.rllib.evaluation), 72
TorchPolicyGraph (class in ray.rllib.evaluation), 74
Trainable (class in ray.tune.trainable), 34
transform() (ray.rllib.models.Preprocessor method), 81
try_reset() (ray.rllib.env.AsyncVectorEnv method), 62
TuneClient (class in ray.tune.web_server), 35
V
variables (ray.experimental.TensorFlowVariables attribute), 107
vector_index (ray.rllib.env.EnvContext attribute), 66
vector_reset() (ray.rllib.env.VectorEnv method), 65
vector_step() (ray.rllib.env.VectorEnv method), 65
VectorEnv (class in ray.rllib.env), 65
W

wait() (in module ray), 14
with_common_config() (in module ray.rllib.agents), 60
worker_index (ray.rllib.env.EnvContext attribute), 66
wrap_async() (ray.rllib.env.AsyncVectorEnv static method), 62