Ray Documentation

Release 0.6.3

The Ray Team

Jul 27, 2019

Installation

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

Ray is easy to install: `pip install ray`

CHAPTER 1

Example Use

Basic Python	Distributed with Ray

```
# Execute f serially.


def f():
    time.sleep(1)
    return 1



results = [f() for i in range(4)]
```
	```
	# Execute f in parallel.

	@ray.remote
	def f():
	    time.sleep(1)
	    return 1


	ray.init()
	results = ray.get([f.remote() for i in range(4)])
	```

To launch a Ray cluster, either privately, on AWS, or on GCP, follow these instructions.

View the codebase on GitHub.

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

- Tune: Scalable Hyperparameter Search
- RLlib: Scalable Reinforcement Learning
- Distributed Training

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.

1.1.1 Latest stable version

You can install the latest stable version of Ray as follows.

```
pip install -U ray  # also recommended: ray[debug]
```

1.1.2 Trying snapshots from master

Here are links to the latest wheels (which are built off of master). To install these wheels, run the following command:

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

Linux	MacOS
Linux Python 3.7	MacOS Python 3.7
Linux Python 3.6	MacOS Python 3.6
Linux Python 3.5	MacOS Python 3.5
Linux Python 3.4	MacOS Python 3.4
Linux Python 2.7	MacOS Python 2.7

1.1.3 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==0.29.0
```

For MacOS, run the following commands:

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

pip install cython==0.29.0
```

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.

```
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 -m pytest -v test/mini_test.py
```

Cleaning the source tree

The source tree can be cleaned by running

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

in the `ray/` directory. Warning: this command will delete all untracked files and directories and will reset the repository to its checked out state. For a shallower working directory cleanup, you may want to try:

```
rm -rf ./build
```

under `ray/`. Incremental builds should work as follows:

```
pushd ./build && make && popd
```

under `ray/`.

1.2 Deploying on Kubernetes

Warning: These instructions have not been tested extensively. If you have a suggestion for how to improve them, please open a pull request or email ray-dev@googlegroups.com.

You can run Ray on top of Kubernetes. This document assumes that you have access to a Kubernetes cluster and have `kubectl` installed locally.

Start by cloning the Ray repository.

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

1.2.1 Work Interactively on the Cluster

To work interactively, first start Ray on Kubernetes.

```
kubectl create -f ray/kubernetes/head.yaml
kubectl create -f ray/kubernetes/worker.yaml
```

This will start one head pod and 3 worker pods. You can check that the pods are running by running `kubectl get pods`.

You should see something like the following (you will have to wait a couple minutes for the pods to enter the "Running" state).

```
$ kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
ray-head-5455bb66c9-6bxvz         1/1       Running   0          10s
ray-worker-5c49b7cc57-c6xs8       1/1       Running   0          5s
ray-worker-5c49b7cc57-d9m86       1/1       Running   0          5s
ray-worker-5c49b7cc57-kzk4s       1/1       Running   0          5s
```

To run tasks interactively on the cluster, connect to one of the pods, e.g.,

```
kubectl exec -it ray-head-5455bb66c9-6bxvz -- bash
```

Start an IPython interpreter, e.g., `ipython`

```
from collections import Counter
import time
import ray

# Note that if you run this script on a non-head node, then you must replace
# "localhost" with socket.gethostbyname("ray-head").
ray.init(redis_address="localhost:6379")

@ray.remote
def f(x):
    time.sleep(0.01)
    return x + (ray.services.get_node_ip_address(), )

# Check that objects can be transferred from each node to each other node.
%time Counter(ray.get([f.remote(f.remote(())) for _ in range(1000)]))
```

1.2.2 Submitting a Script to the Cluster

To submit a self-contained Ray application to your Kubernetes cluster, do the following.

```
kubectl create -f ray/kubernetes/submit.yaml
```

One of the pods will download and run this example script.

The script prints its output. To view the output, first find the pod name by running `kubectl get all`. You'll see output like the following.

```
$ kubectl get all
NAME                                  READY     STATUS    RESTARTS   AGE
pod/ray-head-5486648dc9-c6hz2         1/1       Running   0          11s
```

(continues on next page)

(continued from previous page)

```
pod/ray-worker-5c49b7cc57-2jz4l   1/1     Running   0          11s
pod/ray-worker-5c49b7cc57-8nwjk   1/1     Running   0          11s
pod/ray-worker-5c49b7cc57-xlksn   1/1     Running   0          11s

NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                                     ␣
↪                           AGE
service/ray-head        ClusterIP   10.110.54.241   <none>        6379/TCP,6380/TCP,6381/
↪TCP,12345/TCP,12346/TCP   11s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ray-head       1/1     1            1           11s
deployment.apps/ray-worker     3/3     3            3           11s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/ray-head-5486648dc9       1         1         1       11s
replicaset.apps/ray-worker-5c49b7cc57     3         3         3       11s
```

Find the name of the `ray-head` pod and run the equivalent of

```
kubectl logs ray-head-5486648dc9-c6hz2
```

1.2.3 Cleaning Up

To remove the services you have created, run the following.

```
kubectl delete service/ray-head \
               deployment.apps/ray-head \
               deployment.apps/ray-worker
```

1.2.4 Customization

You will probably need to do some amount of customization.

1. The example above uses the Docker image `rayproject/examples`, which is built using these Dockerfiles. You will most likely need to use your own Docker image.
2. You will need to modify the `command` and `args` fields to potentially install and run the script of your choice.
3. You will need to customize the resource requests.

1.2.5 TODO

The following are also important but haven't been documented yet. Contributions are welcome!

1. Request CPU/GPU/memory resources.
2. Increase shared memory.
3. How to make Kubernetes clean itself up once the script finishes.
4. Follow Kubernetes best practices.

1.3 Installation on Docker

You can install Ray from source 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.

1.3.1 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

Note: The Ray autoscaler can automatically install Docker on all of the nodes of your cluster.

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:

```
sudo apt-get install -y docker.io
sudo service docker start
```

Add the `ubuntu` user to the `docker` group to allow running Docker commands without sudo:

```
sudo usermod -a -G docker ubuntu
```

Initiate a new login to gain group permissions (alternatively, log out and log back in again):

```
exec sudo su -l ubuntu
```

Confirm that docker is running:

```
docker images
```

Should produce an empty table similar to the following:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
```

1.3.2 Clone the Ray repository

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

1.3.3 Build Docker images

Run the script to create Docker images.

```
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:

```
docker images
```

Output should look something like the following:

```
REPOSITORY                          TAG                 IMAGE ID            CREATED         ␣
↪            SIZE
ray-project/examples                latest              7584bde65894        4 days␣
↪ago              3.257 GB
ray-project/deploy                  latest              970966166c71        4 days␣
↪ago              2.899 GB
ray-project/base-deps               latest              f45d66963151        4 days␣
↪ago              2.649 GB
ubuntu                              xenial              f49eec89601e        3 weeks␣
↪ago            129.5 MB
```

1.3.4 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#
```

1.3.5 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 -m pytest -v test/mini_test.py  # This tests some basic functionality.
```

You are now ready to continue with the tutorial.

1.3.6 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
```

Batch L-BFGS

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

Learning to play Pong

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

1.4 Installation Troubleshooting

1.4.1 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.

1.4.2 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.

1.5 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.

1.5.1 Overview

Ray is a 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 **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.

1.5.2 Starting Ray

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

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

This starts Ray.

1.5.3 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.

```
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.

```
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.

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

1.5.4 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.

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

A remote function looks like this.

```
@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.

```
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.

```
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.

```
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.

```
@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.

```
@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.

```
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)]

# Perform a tree reduce.
while len(data) > 1:
    data.append(aggregate_data.remote(data.pop(0), data.pop(0)))

# Fetch the result.
ray.get(data)
```

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.

```
@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))
```

(continues on next page)

(continued from previous page)

```
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.

1.6 The Ray API

`ray.`**`init`**(*redis_address=None, num_cpus=None, num_gpus=None, resources=None, object_store_memory=None, redis_max_memory=None, node_ip_address=None, object_id_seed=None, num_workers=None, local_mode=False, driver_mode=None, redirect_worker_output=False, redirect_output=True, ignore_reinit_error=False, num_redis_shards=None, redis_max_clients=None, redis_password=None, plasma_directory=None, huge_pages=False, include_webui=True, driver_id=None, configure_logging=True, logging_level=20, logging_format='%(asctime)s\t%(levelname)s %(filename)s:%(lineno)s – %(message)s', plasma_store_socket_name=None, raylet_socket_name=None, temp_dir=None, _internal_config=None, use_raylet=None*)

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:

```
ray.init()
```

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

```
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. By default, this is capped at 20GB but can be set higher.
- **`redis_max_memory`** – The max amount of memory (in bytes) to allow each redis shard to use. Once the limit is exceeded, redis will start LRU eviction of entries. This only applies to the sharded redis tables (task, object, and profile tables). By default, this is capped at 10GB but can be set higher.
- **`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.
- **local_mode** (*bool*) – True if the code should be executed serially without Ray. This is useful for debugging.
- **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.
- **redis_password** (*str*) – Prevents external clients without the password from connecting to Redis if provided.
- **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.
- **driver_id** – The ID of driver.
- **configure_logging** – True if allow the logging cofiguration here. Otherwise, the users may want to configure it by their own.
- **logging_level** – Logging level, default will be logging.INFO.
- **logging_format** – Logging format, default contains a timestamp, filename, line number, and message. See ray_constants.py.
- **plasma_store_socket_name** (*str*) – If provided, it will specify the socket name used by the plasma store.
- **raylet_socket_name** (*str*) – If provided, it will specify the socket path used by the raylet process.
- **temp_dir** (*str*) – If provided, it will specify the root temporary directory for the Ray process.
- **_internal_config** (*str*) – JSON configuration for overriding RayConfig defaults. For testing purposes ONLY.

Returns Address information about the started processes.

Raises `Exception` – An exception is raised if an inappropriate combination of arguments is passed in.

`ray.`**`is_initialized`**`()`

Check if ray.init has been called yet.

Returns True if ray.init has already been called and false otherwise.

`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:

```
@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.
- **max_reconstructions**: Only for *actors*. This specifies the maximum number of times that the actor should be reconstructed when it dies unexpectedly. The minimum valid value is 0 (default), which indicates that the actor doesn't need to be reconstructed. And the maximum valid value is ray.ray_constants.INFINITE_RECONSTRUCTIONS.

This can be done as follows:

```
@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.

> **Warning:** The **timeout** argument used to be in **milliseconds** (up through `ray==0.6.1`) and now it is in **seconds**.

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`** (*`float`*) – The maximum amount of time in seconds 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.

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.

Raises `Exception` – An exception is raised if pickle=False and the class cannot be efficiently serialized by Ray. This can also raise an exception if use_dict is true and cls is not pickleable.

`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).

```
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.

```
@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.

1.6.1 The Ray Command Line API

ray start

```
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-password <redis_password>
: If provided, secure Redis ports with this password

--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

--node-manager-port <node_manager_port>
: the port to use for starting the node manager

--object-store-memory <object_store_memory>
: The amount of memory (in bytes) to start the object store with. By default, this is capped at 20GB but can be set higher.

--redis-max-memory `<redis_max_memory>`
The max amount of memory (in bytes) to allow redis to use. Once the limit is exceeded, redis will start LRU eviction of entries. This only applies to the sharded redis tables (task, object, and profile tables). By default this is capped at 10GB but can be set higher.

--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 concurent 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

--no-redirect-worker-output
do not redirect worker stdout and stderr to files

--no-redirect-output
do not redirect non-worker stdout and stderr to files

--plasma-store-socket-name `<plasma_store_socket_name>`
manually specify the socket name of the plasma store

--raylet-socket-name `<raylet_socket_name>`
manually specify the socket path of the raylet process

--temp-dir `<temp_dir>`
manually specify the root temporary dir of the Ray process

--include-java
Enable Java worker support.

--java-worker-options `<java_worker_options>`
Overwrite the options to start Java workers.

--internal-config `<internal_config>`
Do NOT use this. This is for debugging/development purposes ONLY.

ray stop

```
ray stop [OPTIONS]
```

ray up

Create or update a Ray cluster.

```
ray up [OPTIONS] CLUSTER_CONFIG_FILE
```

Options

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

--restart-only
Whether to skip running setup commands and only restart Ray. This cannot be used with 'no-restart'.

--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.

-n, --cluster-name `<cluster_name>`
Override the configured cluster name.

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

Arguments

CLUSTER_CONFIG_FILE
Required argument

ray down

Tear down the Ray cluster.

```
ray down [OPTIONS] CLUSTER_CONFIG_FILE
```

Options

--workers-only
Only destroy the workers.

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

-n, --cluster-name `<cluster_name>`
Override the configured cluster name.

Arguments

CLUSTER_CONFIG_FILE
Required argument

ray exec

```
ray exec [OPTIONS] CLUSTER_CONFIG_FILE CMD
```

Options

--stop
Stop the cluster after the command finishes running.

--start
Start the cluster if needed.

--screen
Run the command in a screen.

--tmux
Run the command in tmux.

-n, --cluster-name `<cluster_name>`
Override the configured cluster name.

--port-forward `<port_forward>`
Port to forward.

Arguments

CLUSTER_CONFIG_FILE
Required argument

CMD
Required argument

ray attach

```
ray attach [OPTIONS] CLUSTER_CONFIG_FILE
```

Options

--start
Start the cluster if needed.

--tmux
Run the command in tmux.

-n, --cluster-name `<cluster_name>`
Override the configured cluster name.

-N, --new
Force creation of a new screen.

Arguments

CLUSTER_CONFIG_FILE
Required argument

ray get_head_ip

```
ray get_head_ip [OPTIONS] CLUSTER_CONFIG_FILE
```

Options

-n, --cluster-name <cluster_name>
Override the configured cluster name.

Arguments

CLUSTER_CONFIG_FILE
Required argument

1.7 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()
```

1.7.1 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()`.

```
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.

1.7.2 Using an actor

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

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

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

1. A task is created.
2. The task is assigned directly to the local scheduler responsible for the actor by the driver's local scheduler.
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]
```

1.7.3 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.

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

1.7.4 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.

```
import tensorflow as tf

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

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

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

    return x, y_, train_step, accuracy
```

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

```
import os

# Define an actor that runs on GPUs. If there are no GPUs, then simply use
# ray.remote without any arguments and no parentheses.
@ray.remote(num_gpus=1)
class NeuralNetOnGPU(object):
    def __init__(self):
        # Set an environment variable to tell TensorFlow which GPUs to use. Note
        # that this must be done before the call to tf.Session.
        os.environ["CUDA_VISIBLE_DEVICES"] = ",".join([str(i) for i in ray.get_gpu_
↪ids()])
        with tf.Graph().as_default():
```

(continues on next page)

(continued from previous page)

```
            with tf.device("/gpu:0"):
                self.x, self.y_, self.train_step, self.accuracy = construct_network()
                # Allow this to run on CPUs if there aren't any GPUs.
                config = tf.ConfigProto(allow_soft_placement=True)
                self.sess = tf.Session(config=config)
                # Initialize the network.
                init = tf.global_variables_initializer()
                self.sess.run(init)
```

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

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

We can put this all together as follows.

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

ray.init(num_gpus=8)

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

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

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

    return x, y_, train_step, accuracy

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

(continues on next page)

(continued from previous page)

```
                # Initialize the network.
                init = tf.global_variables_initializer()
                self.sess.run(init)

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

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


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

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

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

1.7.5 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.

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

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

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

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

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

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

```
counter = Counter.remote()
```

(continues on next page)

(continued from previous page)

```
# 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()))
```

1.7.6 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.

1.8 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.

1.8.1 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.

```
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.

```
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.

1.8.2 Using Remote Functions with GPUs

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

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

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.

1.8.3 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.

```
@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.

```
@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()
```

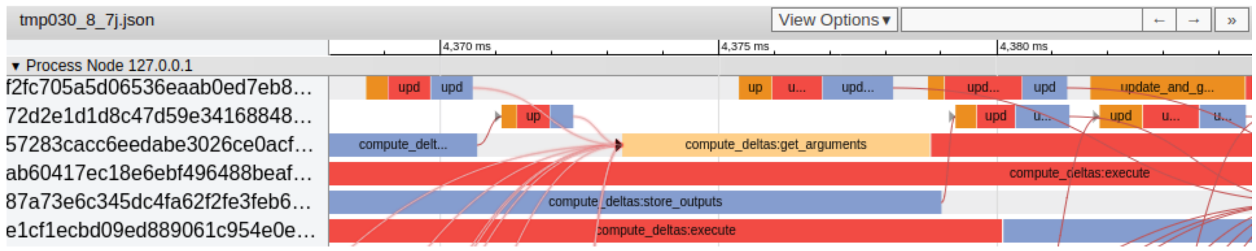
1.8.4 Troubleshooting

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.

1.9 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:

```
To view fullscreen, open chrome://tracing in Google Chrome and load `/tmp/tmpzoppr7k0.json`
```


tmp030_8_7j.json
View Options ▾
← → »
4,370 ms
4,375 ms
4,380 ms
▾ Process Node 127.0.0.1
f2fc705a5d06536eaab0ed7eb8…
72d2e1d1d8c47d59e34168848…
57283cacc6eedabe3026ce0acf…
ab60417ec18e6ebf496488beaf…
87a73e6c345dc4fa62f2fe3feb6…
e1cf1ecbd09ed889061c954e0e…
upd
up
u...
upd...
update_and_g...
compute_delt...
compute_deltas:get_arguments
compute_deltas:execute
compute_deltas:store_outputs

1.9.1 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.

1.9.2 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=89354a314e5a81bf56e023ad18bda3a3d272ee216f342938
=========================================================================
```

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
```

1.9.3 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 appropiate 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.

1.9.4 Troubleshooting

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

1.10 Async API (Experimental)

Since Python 3.5, it is possible to write concurrent code using the `async/await` syntax.

This document describes Ray's support for asyncio, which enables integration with popular async frameworks (e.g., aiohttp, aioredis, etc.) for high performance web and prediction serving.

1.10.1 Starting Ray

You must initialize Ray first.

Please refer to Starting Ray for instructions.

1.10.2 Converting Ray objects into asyncio futures

Ray object IDs can be converted into asyncio futures with `ray.experimental.async_api`.

```python
import asyncio
import time
import ray
from ray.experimental import async_api

@ray.remote
def f():
    time.sleep(1)
    return {'key1': ['value']}

ray.init()
future = async_api.as_future(f.remote())
asyncio.get_event_loop().run_until_complete(future)  # {'key1': ['value']}
```

`ray.experimental.async_api.`**`as_future`**`(`*object_id*`)`
Turn an object_id into a Future object.

Parameters **`object_id`** – A Ray object_id.

Returns A future object that waits the object_id.

Return type PlasmaObjectFuture

1.10.3 Example Usage

Basic Python

```python
# Execute f serially.


def f():
  time.sleep(1)
  return 1



results = [f() for i in range(4)]
```

Distributed with Ray

```python
# Execute f in parallel.


@ray.remote
def f():
    time.sleep(1)
    return 1

ray.init()
results = ray.get([f.remote() for i in␣
↪range(4)])
```

Async Python

```python
# Execute f asynchronously.




async def f():
    await asyncio.sleep(1)
    return 1


loop = asyncio.get_event_loop()
tasks = [f() for i in range(4)]

results = loop.run_until_complete(
    asyncio.gather(tasks))
```

Async Ray

```python
# Execute f asynchronously with Ray/
↪asyncio.

from ray.experimental import async_api

@ray.remote
def f():
    time.sleep(1)
    return 1

ray.init()
loop = asyncio.get_event_loop()
tasks = [async_api.as_future(f.remote())
         for i in range(4)]
results = loop.run_until_complete(
    asyncio.gather(tasks))
```

1.10.4 Known Issues

Async API support is experimental, and we are working to improve its performance. Please let us know any issues you encounter.

1.11 Cluster Setup and Auto-Scaling

This document provides instructions for launching a Ray cluster either privately, on AWS, or on GCP.

The `ray up` command starts or updates a Ray cluster from your personal computer. Once the cluster is up, you can then SSH into it to run Ray programs.

1.11.1 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")`.

```
# 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 up ray/python/ray/autoscaler/aws/example-full.yaml

# Reconfigure autoscaling behavior without interrupting running jobs
$ ray up ray/python/ray/autoscaler/aws/example-full.yaml \
    --max-workers=N --no-restart

# Teardown the cluster
$ ray down ray/python/ray/autoscaler/aws/example-full.yaml
```

1.11.2 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")`.

```
# 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 up ray/python/ray/autoscaler/gcp/example-full.yaml

# Reconfigure autoscaling behavior without interrupting running jobs
$ ray up ray/python/ray/autoscaler/gcp/example-full.yaml \
    --max-workers=N --no-restart

# Teardown the cluster
$ ray down ray/python/ray/autoscaler/gcp/example-full.yaml
```

1.11.3 Quick start (Private Cluster)

This is used when you have a list of machine IP addresses to connect in a Ray cluster. You can get started by filling out the fields in the provided ray/python/ray/autoscaler/local/example-full.yaml. Be sure to specify the proper `head_ip`, list of `worker_ips`, and the `ssh_user` field.

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")`.

```
# 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 up ray/python/ray/autoscaler/local/example-full.yaml

# Reconfigure autoscaling behavior without interrupting running jobs
$ ray up ray/python/ray/autoscaler/local/example-full.yaml \
    --max-workers=N --no-restart

# Teardown the cluster
$ ray down ray/python/ray/autoscaler/local/example-full.yaml
```

1.11.4 Running commands on new and existing clusters

You can use `ray exec` to conveniently run commands on clusters. Note that scripts you run should connect to Ray via `ray.init(redis_address="localhost:6379")`.

```
# Run a command on the cluster
$ ray exec cluster.yaml 'echo "hello world"'

# Run a command on the cluster, starting it if needed
$ ray exec cluster.yaml 'echo "hello world"' --start

# Run a command on the cluster, stopping the cluster after it finishes
$ ray exec cluster.yaml 'echo "hello world"' --stop

# Run a command on a new cluster called 'experiment-1', stopping it after
$ ray exec cluster.yaml 'echo "hello world"' \
    --start --stop --cluster-name experiment-1

# Run a command in a detached tmux session
$ ray exec cluster.yaml 'echo "hello world"' --tmux

# Run a command in a screen (experimental)
$ ray exec cluster.yaml 'echo "hello world"' --screen
```

You can also use `ray submit` to execute Python scripts on clusters. This will `rsync` the designated file onto the cluster and execute it with the given arguments.

```
# Run a Python script in a detached tmux session
$ ray submit cluster.yaml --tmux --start --stop tune_experiment.py
```

1.11.5 Attaching to the cluster

You can use `ray attach` to attach to an interactive console on the cluster.

```
# Open a screen on the cluster
$ ray attach cluster.yaml

# Open a screen on a new cluster called 'session-1'
$ ray attach cluster.yaml --start --cluster-name=session-1

# Attach to tmux session on cluster (creates a new one if none available)
$ ray attach cluster.yaml --tmux
```

1.11.6 Port-forwarding applications

To run connect to applications running on the cluster (e.g. Jupyter notebook) using a web browser, you can use the port-forward option for `ray exec`. The local port opened is the same as the remote port:

```
$ ray exec cluster.yaml --port-forward=8899 'source ~/anaconda3/bin/activate␣
↪tensorflow_p36 && jupyter notebook --port=8899'
```

1.11.7 Manually synchronizing files

To download or upload files to the cluster head node, use `ray rsync_down` or `ray rsync_up`:

```
$ ray rsync_down cluster.yaml '/path/on/cluster' '/local/path'
$ ray rsync_up cluster.yaml '/local/path' '/path/on/cluster'
```

1.11.8 Updating your cluster

When you run `ray up` 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 up` 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.

1.11.9 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.

1.11.10 Autoscaling

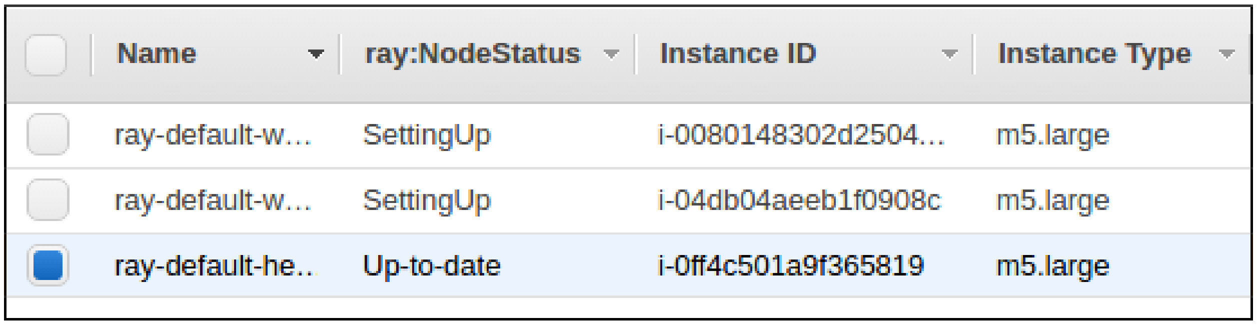
Ray clusters come with a load-based auto-scaler. When cluster resource usage exceeds a configurable threshold (80% by default), new nodes will be launched up the specified `max_workers` limit. When nodes are idle for more than a timeout, they will be removed, down to the `min_workers` limit. The head 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).

1.11.11 Monitoring cluster status

You can monitor cluster usage and auto-scaling status by tailing the autoscaling logs in `/tmp/ray/session_*/logs/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:



	Name	ray:NodeStatus	Instance ID	Instance Type
☐	ray-default-w…	SettingUp	i-0080148302d2504…	m5.large
☐	ray-default-w…	SettingUp	i-04db04aeeb1f0908c	m5.large
■	ray-default-he..	Up-to-date	i-0ff4c501a9f365819	m5.large

1.11.12 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.

1.11.13 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):

```
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 up` 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 up`

1.11.14 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. It will also automatically install Docker if Docker is not installed. 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: 1  # must have at least 1 GPU worker (issue #2106)
max_workers: 10
head_node:
    InstanceType: m4.large
worker_nodes:
    InstanceMarketOptions:
        MarketType: spot
    InstanceType: p2.xlarge
```

1.11.15 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:

```
provider:
    type: external
    module: mypackage.myclass
```

The module needs to be in the format *package.provider_class* or *package.sub_package.provider_class*.

1.11.16 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!

1.11.17 Questions or Issues?

You can post questions or issues or feedback through the following channels:

1. ray-dev@googlegroups.com: For discussions about development or any general questions and feedback.
2. StackOverflow: For questions about how to use Ray.
3. GitHub Issues: For bug reports and feature requests.

1.12 Manual Cluster Setup

Note: If you're using AWS or GCP you should use the automated setup commands.

The instructions in this document work well for small clusters. For larger clusters, consider using the pssh package: `sudo apt-get install pssh` or the setup commands for private clusters.

1.12.1 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.

```
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`).

```
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).

To run some commands, start up Python on one of the nodes in the cluster, and do the following.

```
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.

```
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.

1.13 Tune: Scalable Hyperparameter Search

Tune is a scalable framework for hyperparameter search with a focus on deep learning and deep reinforcement learning.

You can find the code for Tune here on GitHub. To get started with Tune, try going through our tutorial of using Tune with Keras.

(Experimental): You can try out the above tutorial on a free hosted server via Binder.

1.13.1 Features

- Supports any deep learning framework, including PyTorch, TensorFlow, and Keras.
- Choose among scalable hyperparameter and model search techniques such as:
  - Population Based Training (PBT)
  - Median Stopping Rule
  - HyperBand
- Mix and match different hyperparameter optimization approaches - such as using HyperOpt with HyperBand.
- Visualize results with TensorBoard, parallel coordinates (Plot.ly), and rllab's VisKit.
- Scale to running on a large distributed cluster without changing your code.
- Parallelize training for models with GPU requirements or algorithms that may themselves be parallel and distributed, using Tune's resource-aware scheduling,

Take a look at the User Guide for a comprehensive overview on how to use Tune's features.

1.13.2 Getting Started

Installation

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

```
pip install ray  # also recommended: ray[debug]
```

Quick Start

This example runs a small grid search over a neural network training function using Tune, reporting status on the command line until the stopping condition of `mean_accuracy >= 99` is reached. Tune works with any deep learning framework.

Tune uses Ray as a backend, so we will first import and initialize Ray.

```
import ray
import ray.tune as tune

ray.init()
```

For the function you wish to tune, pass in a `reporter` object:

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

    for idx, (data, target) in enumerate(dataset):
        accuracy = model.fit(data, target)
        reporter(mean_accuracy=accuracy) # report metrics
```

Finally, configure your search and execute it on your Ray cluster:

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

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.

1.13.3 Citing Tune

If Tune helps you in your academic research, you are encouraged to cite our paper. Here is an example bibtex:

```
@article{liaw2018tune,
    title={Tune: A Research Platform for Distributed Model Selection and Training},
    author={Liaw, Richard and Liang, Eric and Nishihara, Robert
            and Moritz, Philipp and Gonzalez, Joseph E and Stoica, Ion},
```

(continues on next page)

(continued from previous page)

```
    journal={arXiv preprint arXiv:1807.05118},
    year={2018}
}
```

1.14 Tune User Guide

1.14.1 Tune Overview

Tune schedules a number of *trials* in a cluster. Each trial runs a user-defined Python function or class and is parameterized either by a *config* variation from Tune's Variant Generator or a user-specified **search algorithm**. The trials are scheduled and managed by a **trial scheduler**.

More information about Tune's search algorithms can be found here. More information about Tune's trial schedulers can be found here.

Start by installing, importing, and initializing Ray.

```
import ray
import ray.tune as tune

ray.init()
```

1.14.2 Experiment Configuration

This section will cover the main steps needed to modify your code to run Tune: using the Training API and executing your Tune experiment.

You can checkout out our examples page for more code examples.

Training API

Training can be done with either the **function-based API** or **Trainable API**.

Python functions will need to have the following signature:

```
def trainable(config, reporter):
    """
    Args:
        config (dict): Parameters provided from the search algorithm
            or variant generation.
        reporter (Reporter): Handle to report intermediate metrics to Tune.
    """

    while True:
        # ...
        reporter(**kwargs)
```

The reporter will allow you to report metrics used for scheduling, search, or early stopping.

Tune will run this function on a separate thread in a Ray actor process. Note that this API is not checkpointable, since the thread will never return control back to its caller. The reporter documentation can be found here.

Note: If you have a lambda function that you want to train, you will need to first register the function: `tune.register_trainable("lambda_id", lambda x: ...)`. You can then use `lambda_id` in place of `my_trainable`.

Python classes passed into Tune will need to subclass `ray.tune.Trainable`. The Trainable interface can be found here.

Both the Trainable and function-based API will have autofilled metrics in addition to the metrics reported.

See the experiment specification section on how to specify and execute your training.

Specifying Experiments

There are two ways to specify the configuration for an experiment - one via Python and one via JSON.

Using Python: specify a configuration is to create an Experiment object.

class `ray.tune.`**`Experiment`**(*name*, *run*, *stop=None*, *config=None*, *resources_per_trial=None*, *num_samples=1*, *local_dir=None*, *upload_dir=None*, *trial_name_creator=None*, *custom_loggers=None*, *sync_function=None*, *checkpoint_freq=0*, *checkpoint_at_end=False*, *export_formats=None*, *max_failures=3*, *restore=None*, *repeat=None*, *trial_resources=None*)

Tracks experiment specifications.

Parameters

- **`name`** (*`str`*) – Name of experiment.
- **`run`** (*`function|class|str`*) – The algorithm or model to train. This may refer to the name of a built-on algorithm (e.g. RLLib's DQN or PPO), a user-defined trainable function or class, or the string identifier of a trainable function or class registered in the tune registry.
- **`stop`** (*`dict`*) – The stopping criteria. The keys may be any field in the return result of 'train()', whichever is reached first. Defaults to empty dict.
- **`config`** (*`dict`*) – Algorithm-specific configuration for Tune variant generation (e.g. env, hyperparams). Defaults to empty dict. Custom search algorithms may ignore this.
- **`resources_per_trial`** (*`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 in `Trainable.default_resource_request()`.
- **`num_samples`** (*`int`*) – Number of times to sample from the hyperparameter space. Defaults to 1. If *grid_search* is provided as an argument, the grid will be repeated *num_samples* of times.
- **`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://bucket`).
- **`trial_name_creator`** (*`func`*) – Optional function for generating the trial string representation.
- **`custom_loggers`** (*`list`*) – List of custom logger creators to be used with each Trial. See *ray/tune/logger.py*.

- **sync_function** (*func|str*) – Function for syncing the local_dir to upload_dir. If string, then it must be a string template for syncer to run. If not provided, the sync command defaults to standard S3 or gsutil sync comamnds.
- **checkpoint_freq** (*int*) – How many training iterations between checkpoints. A value of 0 (default) disables checkpointing.
- **checkpoint_at_end** (*bool*) – Whether to checkpoint at the end of the experiment regardless of the checkpoint_freq. Default is False.
- **export_formats** (*list*) – List of formats that exported at the end of the experiment. Default is None.
- **max_failures** (*int*) – Try to recover a trial from its last checkpoint at least this many times. Only applies if checkpointing is enabled. Setting to -1 will lead to infinite recovery retries. Defaults to 3.
- **restore** (*str*) – Path to checkpoint. Only makes sense to set if running 1 trial. Defaults to None.
- **repeat** – Deprecated and will be removed in future versions of Ray. Use *num_samples* instead.
- **trial_resources** – Deprecated and will be removed in future versions of Ray. Use *resources_per_trial* instead.

Examples

```
>>> experiment_spec = Experiment(
>>>     "my_experiment_name",
>>>     my_func,
>>>     stop={"mean_accuracy": 100},
>>>     config={
>>>         "alpha": tune.grid_search([0.2, 0.4, 0.6]),
>>>         "beta": tune.grid_search([1, 2]),
>>>     },
>>>     resources_per_trial={
>>>         "cpu": 1,
>>>         "gpu": 0
>>>     },
>>>     num_samples=10,
>>>     local_dir="~/ray_results",
>>>     upload_dir="s3://your_bucket/path",
>>>     checkpoint_freq=10,
>>>     max_failures=2)
```

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

Using JSON/Dict: This uses the same fields as the `ray.tune.Experiment`, except the experiment name is the key of the top level dictionary. Tune will convert the dict into an `ray.tune.Experiment` object.

```
experiment_spec = {
    "my_experiment_name": {
        "run": my_func,
        "stop": { "mean_accuracy": 100 },
        "config": {
            "alpha": tune.grid_search([0.2, 0.4, 0.6]),
            "beta": tune.grid_search([1, 2]),
```

(continues on next page)

(continued from previous page)

```
        },
        "resources_per_trial": { "cpu": 1, "gpu": 0 },
        "num_samples": 10,
        "local_dir": "~/ray_results",
        "upload_dir": "s3://your_bucket/path",
        "checkpoint_freq": 10,
        "max_failures": 2
    }
}
```

Tune provides a `run_experiments` function that generates and runs the trials.

`ray.tune.`**`run_experiments`**(*experiments*, *search_alg=None*, *scheduler=None*, *with_server=False*, *server_port=4321*, *verbose=2*, *resume=False*, *queue_trials=False*, *trial_executor=None*, *raise_on_failed_trial=True*)

Runs and blocks until all trials finish.

Parameters

- **`experiments`** (*`Experiment | list | dict`*) – Experiments to run. Will be passed to *search_alg* via *add_configurations*.
- **`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`** (*`int`*) – 0, 1, or 2. Verbosity mode. 0 = silent, 1 = only status updates, 2 = status and trial results.
- **`resume`** (*`bool | "prompt"`*) – If checkpoint exists, the experiment will resume from there. If resume is "prompt", Tune will prompt if checkpoint detected.
- **`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.
- **`trial_executor`** (*`TrialExecutor`*) – Manage the execution of trials.
- **`raise_on_failed_trial`** (*`bool`*) – Raise TuneError if there exists failed trial (of ERROR state) when the experiments complete.

Examples

```
>>> experiment_spec = Experiment("experiment", my_func)
>>> run_experiments(experiments=experiment_spec)
```

```
>>> experiment_spec = {"experiment": {"run": my_func}}
>>> run_experiments(experiments=experiment_spec)
>>> run_experiments(
>>>     experiments=experiment_spec,
>>>     scheduler=MedianStoppingRule(...))
```

```
>>> run_experiments(
>>>     experiments=experiment_spec,
>>>     search_alg=SearchAlgorithm(),
>>>     scheduler=MedianStoppingRule(...))
```

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

This function will report status on the command line until all Trials stop:

```
== 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
```

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

Custom Trial Names

To specify custom trial names, you can pass use the `trial_name_creator` argument in the Experiment object. This takes a function with the following signature, and be sure to wrap it with *tune.function*:

```
def trial_name_string(trial):
    """
    Args:
        trial (Trial): A generated trial object.

    Returns:
        trial_name (str): String representation of Trial.
    """
    return str(trial)

exp = Experiment(
    name="hyperband_test",
    run=MyTrainableClass,
    num_samples=1,
    trial_name_creator=tune.function(trial_name_string)
)
```

An example can be found in logging_example.py.

1.14.3 Training Features

Tune Search Space (Default)

You can use `tune.grid_search` to specify an axis of a grid search. By default, Tune also supports sampling parameters from user-specified lambda functions, which can be used independently or in combination with grid search.

Note: If you specify an explicit Search Algorithm such as any SuggestionAlgorithm, you may not be able to specify lambdas or grid search with this interface, as the search algorithm may require a different search space declaration.

The following shows grid search over two nested parameters combined with random sampling from two lambda functions, generating 9 different trials. 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.

```
run_experiments({
    "my_experiment_name": {
        "run": my_trainable,
        "config": {
            "alpha": tune.sample_from(lambda spec: np.random.uniform(100)),
            "beta": tune.sample_from(lambda spec: spec.config.alpha * np.random.
↪normal()),
            "nn_layers": [
                tune.grid_search([16, 64, 256]),
                tune.grid_search([16, 64, 256]),
            ],
        }
    }
})
```

Note: Use `tune.sample_from(...)` to sample from a function during trial variant generation. If you need to pass a literal function in your config, use `tune.function(...)` to escape it.

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

Sampling Multiple Times

By default, each random variable and grid search point is sampled once. To take multiple random samples, add `num_samples: N` to the experiment config. If *grid_search* is provided as an argument, the grid will be repeated *num_samples* of times.

```
run_experiments({
    "my_experiment_name": {
        "run": my_trainable,
        "config": {
            "alpha": tune.sample_from(lambda spec: np.random.uniform(100)),
            "beta": tune.sample_from(lambda spec: spec.config.alpha * np.random.
↪normal()),
            "nn_layers": [
                tune.grid_search([16, 64, 256]),
                tune.grid_search([16, 64, 256]),
            ],
        },
        "num_samples": 10
```

(continues on next page)

(continued from previous page)

```
    }
})
```

E.g. in the above, `"num_samples": 10` repeats the 3x3 grid search 10 times, for a total of 90 trials, each with randomly sampled values of `alpha` and `beta`.

Using GPUs (Resource Allocation)

Tune will allocate the specified GPU and CPU `resources_per_trial` to each individual trial (defaulting to 1 CPU per trial). Under the hood, Tune runs each trial as a Ray actor, using Ray's resource handling to allocate resources and place actors. A trial will not be scheduled unless at least that amount of resources is available in the cluster, preventing the cluster from being overloaded.

Fractional values are also supported, (i.e., `"gpu": 0.2`). You can find an example of this in the Keras MNIST example.

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`.

```
run_experiments({
    "my_experiment_name": {
        "run": my_trainable,
        "resources_per_trial": {
            "cpu": 1,
            "gpu": 1,
            "extra_gpu": 4
        }
    }
})
```

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 Trial Schedulers such as HyperBand and PBT.

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

```
class MyClass(Trainable):
    def _setup(self, config):
        self.saver = tf.train.Saver()
        self.sess = ...
        self.iteration = 0

    def _train(self):
        self.sess.run(...)
        self.iteration += 1
```

(continues on next page)

(continued from previous page)

```
    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.:

```
run_experiments({
    "my_experiment_name": {
        "run": my_trainable
        "checkpoint_freq": 10,
        "max_failures": 5,
    },
})
```

The checkpoint_freq may not coincide with the exact end of an experiment. If you want a checkpoint to be created at the end of a trial, you can additionally set the checkpoint_at_end to True. An example is shown below:

```
run_experiments({
    "my_experiment_name": {
        "run": my_trainable
        "checkpoint_freq": 10,
        "checkpoint_at_end": True,
        "max_failures": 5,
    },
})
```

Recovering From Failures (Experimental)

Tune automatically persists the progress of your experiments, so if an experiment crashes or is otherwise cancelled, it can be resumed with `resume=True`. The default setting of `resume=False` creates a new experiment, and `resume="prompt"` will cause Tune to prompt you for whether you want to resume. You can always force a new experiment to be created by changing the experiment name.

Note that trials will be restored to their last checkpoint. If trial checkpointing is not enabled, unfinished trials will be restarted from scratch.

E.g.:

```
run_experiments({
    "my_experiment_name": {
        "run": my_trainable
        "checkpoint_freq": 10,
        "local_dir": "~/path/to/results"
    },
}, resume=True)
```

Upon a second run, this will restore the entire experiment state from `~/path/to/results/my_experiment_name`. Importantly, any changes to the experiment specification upon resume will be ignored.

This feature is still experimental, so any provided Trial Scheduler or Search Algorithm will not be preserved. Only `FIFOScheduler` and `BasicVariantGenerator` will be supported.

1.14.4 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. 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`.

```
import ray
from ray.tune import run_experiments
from ray.tune.util import pin_in_object_store, get_pinned_object

import numpy as np

ray.init()

# X_id can be referenced in closures
X_id = pin_in_object_store(np.random.random(size=100000000))

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

run_experiments({
    "my_experiment_name": {
        "run": f
    }
})
```

1.14.5 Auto-Filled Results

During training, Tune will automatically fill certain fields if not already provided. All of these can be used as stopping conditions or in the Scheduler/Search Algorithm specification.

```
# (Optional/Auto-filled) training is terminated. Filled only if not provided.
DONE = "done"

# (Auto-filled) The hostname of the machine hosting the training process.
HOSTNAME = "hostname"

# (Auto-filled) The node ip of the machine hosting the training process.
NODE_IP = "node_ip"

# (Auto-filled) The pid of the training process.
PID = "pid"

# Number of episodes in this iteration.
EPISODES_THIS_ITER = "episodes_this_iter"

# (Optional/Auto-filled) Accumulated number of episodes for this experiment.
EPISODES_TOTAL = "episodes_total"

# Number of timesteps in this iteration.
```

(continues on next page)

(continued from previous page)

```
TIMESTEPS_THIS_ITER = "timesteps_this_iter"

# (Auto-filled) Accumulated number of timesteps for this entire experiment.
TIMESTEPS_TOTAL = "timesteps_total"

# (Auto-filled) Time in seconds this iteration took to run.
# This may be overriden to override the system-computed time difference.
TIME_THIS_ITER_S = "time_this_iter_s"

# (Auto-filled) Accumulated time in seconds for this entire experiment.
TIME_TOTAL_S = "time_total_s"

# (Auto-filled) The index of this training iteration.
TRAINING_ITERATION = "training_iteration"
```

The following fields will automatically show up on the console output, if provided:

1. `episode_reward_mean`
2. `mean_loss`
3. `mean_accuracy`
4. `timesteps_this_iter` (aggregated into `timesteps_total`).

```
Example_0:  TERMINATED [pid=68248], 179 s, 2 iter, 60000 ts, 94 rew
```

1.14.6 Logging and Visualizing Results

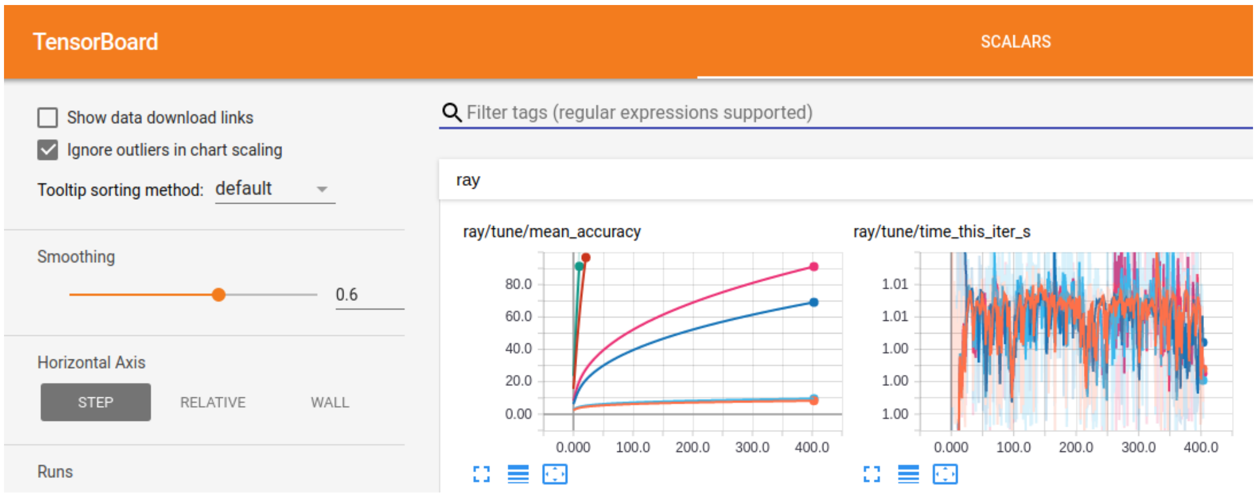
All results reported by the trainable will be logged locally to a unique directory per experiment, e.g. `~/ray_results/my_experiment` in the above example. On a cluster, incremental results will be synced to local disk on the head node. The log records are compatible with a number of visualization tools:

To visualize learning in tensorboard, install TensorFlow:

```
$ pip install tensorflow
```

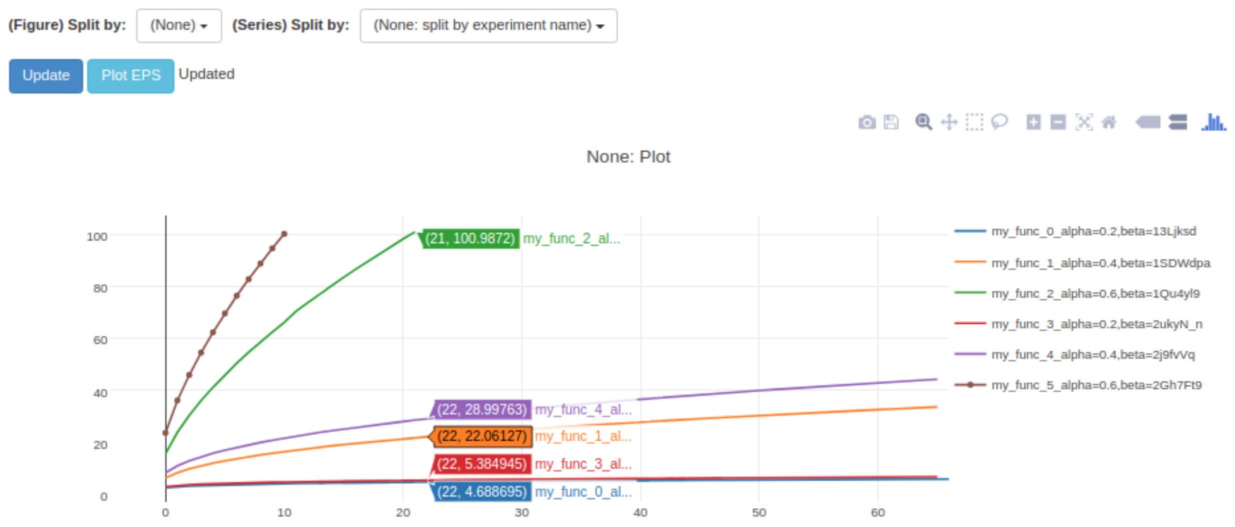
Then, after you run a experiment, you can visualize your experiment with TensorBoard by specifying the output directory of your results. Note that if you running Ray on a remote cluster, you can forward the tensorboard port to your local machine through SSH using `ssh -L 6006:localhost:6006 <address>`:

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


TensorBoard
SCALARS
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Filter tags (regular expressions supported)
ray
ray/tune/mean_accuracy
ray/tune/time_this_iter_s

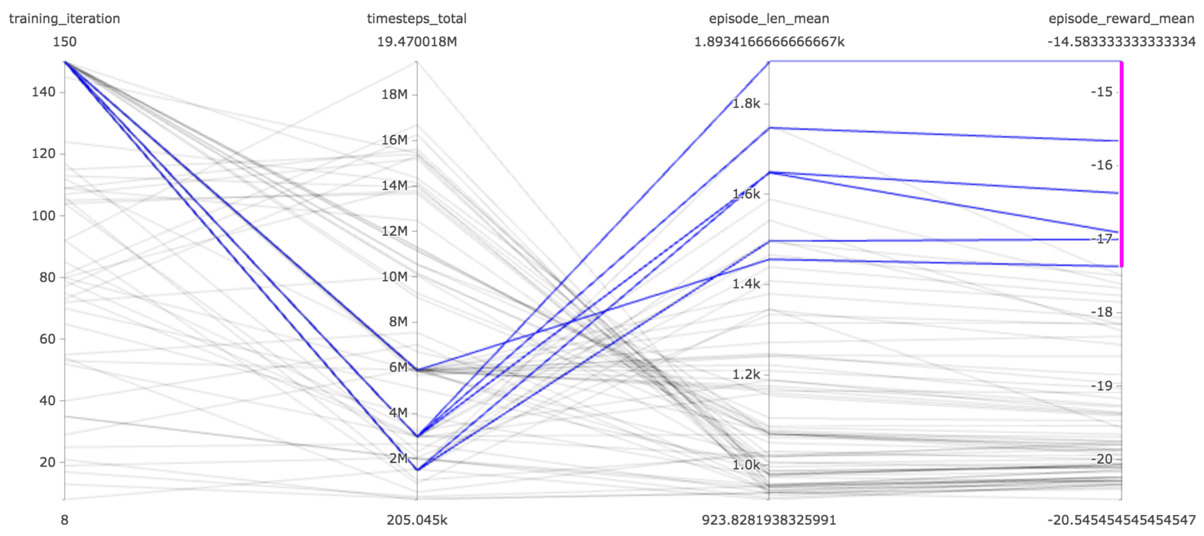
To use rllab's VisKit (you may have to install some dependencies), run:

```
$ git clone https://github.com/rll/rllab.git
$ python rllab/rllab/viskit/frontend.py ~/ray_results/my_experiment
```


(Figure) Split by: (None)
(Series) Split by: (None: split by experiment name)
Update
Plot EPS
Updated
None: Plot
(21, 100.9872) my_func_2_al...
(22, 28.99763) my_func_4_al...
(22, 22.06127) my_func_1_al...
(22, 5.384945) my_func_3_al...
(22, 4.688695) my_func_0_al...
my_func_0_alpha=0.2,beta=13Ljksd
my_func_1_alpha=0.4,beta=1SDWdpa
my_func_2_alpha=0.6,beta=1Qu4yl9
my_func_3_alpha=0.2,beta=2ukyN_n
my_func_4_alpha=0.4,beta=2j9fvVq
my_func_5_alpha=0.6,beta=2Gh7Ft9

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
```



Custom Loggers

You can pass in your own logging mechanisms to output logs in custom formats via the Experiment object as follows:

```
exp = Experiment(
    name="experiment_name",
    run=MyTrainableClass,
    custom_loggers=[CustomLogger1, CustomLogger2]
)
```

These loggers will be called along with the default Tune loggers. All loggers must inherit the Logger interface.

You can also check out logger.py for implementation details.

An example can be found in logging_example.py.

Custom Sync/Upload Commands

If an upload directory is provided, Tune will automatically sync results to the given directory with standard S3/gsutil commands. You can customize the upload command by providing either a function or a string.

If a string is provided, then it must include replacement fields `{local_dir}` and `{remote_dir}`, like `"aws s3 sync {local_dir} {remote_dir}"`.

Alternatively, a function can be provided with the following signature (and must be wrapped with `tune.function`):

```
def custom_sync_func(local_dir, remote_dir):
    sync_cmd = "aws s3 sync {local_dir} {remote_dir}".format(
        local_dir=local_dir,
        remote_dir=remote_dir)
    sync_process = subprocess.Popen(sync_cmd, shell=True)
    sync_process.wait()

exp = Experiment(
    name="experiment_name",
    run=MyTrainableClass,
    sync_function=tune.function(custom_sync_func)
)
```

1.14.7 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.

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.

1.14.8 Further Questions or Issues?

You can post questions or issues or feedback through the following channels:

1. ray-dev@googlegroups.com: For discussions about development or any general questions and feedback.
2. StackOverflow: For questions about how to use Ray.
3. GitHub Issues: For bug reports and feature requests.

1.15 Tune Trial Schedulers

By default, Tune schedules trials in serial order with the `FIFOScheduler` class. However, you can also specify a custom scheduling algorithm that can early stop trials or perturb parameters.

```
tune.run_experiments({...}, scheduler=AsyncHyperBandScheduler())
```

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. Currently, all schedulers take in a `reward_attr`, which is assumed to be maximized.

Current Available Trial Schedulers:

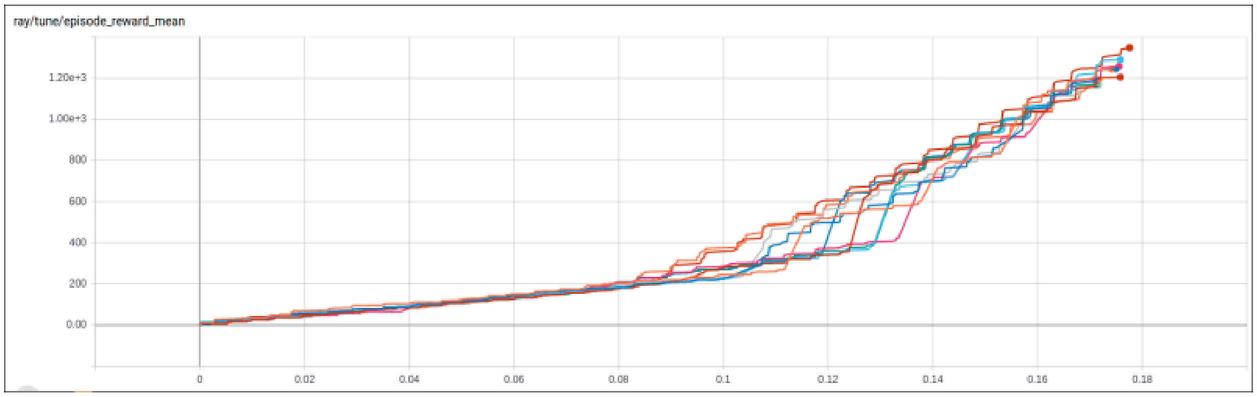
1.15.1 Population Based Training (PBT)

Tune includes a distributed implementation of Population Based Training (PBT). This can be enabled by setting the `scheduler` parameter of `run_experiments`, e.g.

```
pbt_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),
            ...
        })
run_experiments({...}, scheduler=pbt_scheduler)
```

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.schedulers.**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 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 training result 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 training result 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

```
>>> 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={
>>>         # Perturb factor1 by scaling it by 0.8 or 1.2. Resampling
>>>         # resets it to a value sampled from the lambda function.
>>>         "factor_1": lambda: random.uniform(0.0, 20.0),
>>>         # Perturb factor2 by changing it to an adjacent value, e.g.
>>>         # 10 -> 1 or 10 -> 100. Resampling will choose at random.
>>>         "factor_2": [1, 10, 100, 1000, 10000],
>>>     })
>>> run_experiments({...}, scheduler=pbt)
```

1.15.2 Asynchronous HyperBand

The asynchronous version of HyperBand scheduler can be used by setting the `scheduler` parameter of `run_experiments`, e.g.

```
async_hb_scheduler = AsyncHyperBandScheduler(
    time_attr='training_iteration',
    reward_attr='episode_reward_mean',
    max_t=100,
    grace_period=10,
    reduction_factor=3,
    brackets=3)
run_experiments({...}, scheduler=async_hb_scheduler)
```

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.**

class ray.tune.schedulers.**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=S1Y7OOlRZ

Parameters

- **time_attr** (*str*) – A training result 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 training result 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.

1.15.3 HyperBand

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

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

```
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.

```
== 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}
  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=69,width=45:  PENDING
 - my_class_15_height=41,width=11:  PENDING
 - my_class_16_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.schedulers.**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 Tune, all you need to do is specify the max length of time a trial can run *max_t*, the time units *time_attr*, and the name of the reported objective value *reward_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 *episode_mean_reward* attr, construct:

`HyperBand('time_total_s', 'episode_reward_mean', max_t=600)`

Note that Tune's stopping criteria will be applied in conjunction with HyperBand's early stopping mechanisms.

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

Parameters

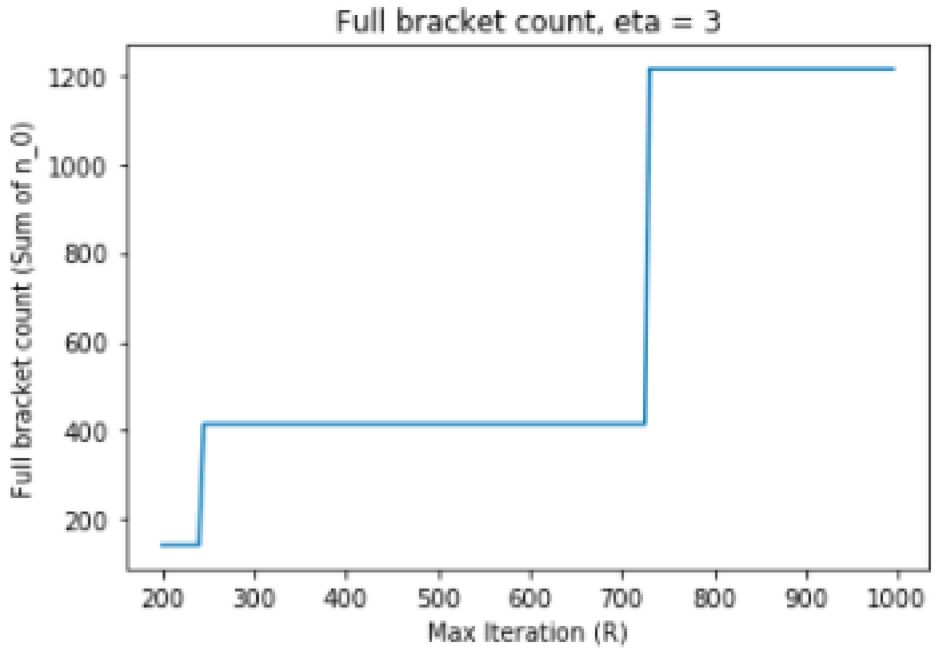
- **time_attr** (*str*) – The training result attr to use for comparing time. Note that you can pass in something non-temporal such as *training_iteration* as a measure of progress, the only requirement is that the attribute should increase monotonically.

- **reward_attr** (*str*) – The training result objective value attribute. As with *time_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 max_t time units (determined by time_attr) have passed. The scheduler will terminate trials after this time has passed. Note that this is different from the semantics of *max_t* as mentioned in the original HyperBand paper.

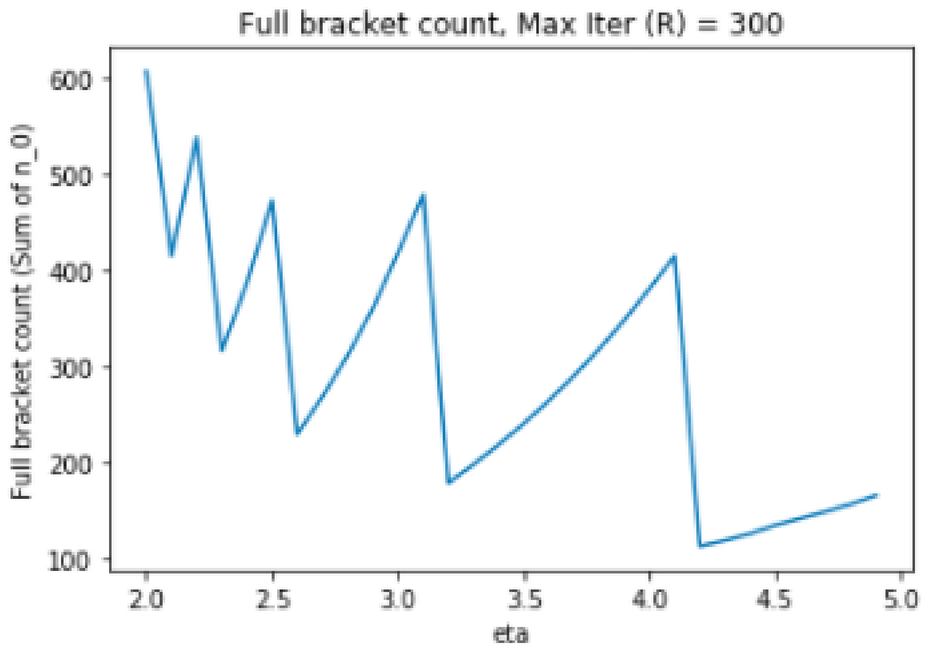
HyperBand Implementation Details

Implementation details may deviate slightly from theory but are focused on increasing usability. Note: `R`, `s_max`, and `eta` are parameters of HyperBand given by the paper. See this post for context.

1. Both `s_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 >= 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.

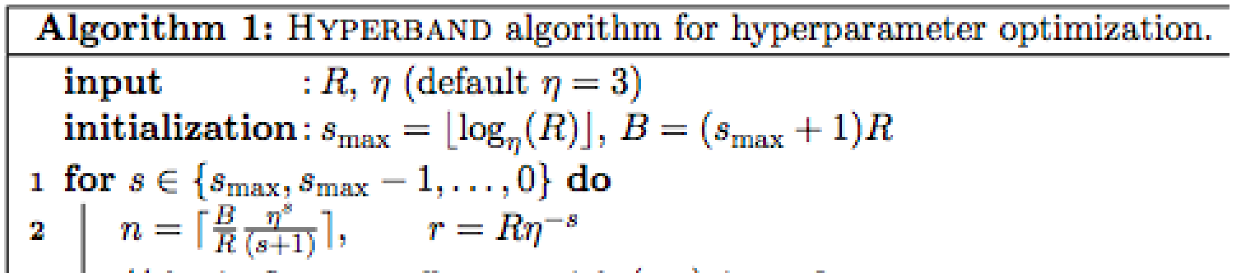


On the other hand, holding `R` constant at `R = 300` and varying `eta` also leads to HyperBand configurations that are not very intuitive:



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):



```
Algorithm 1: HYPERBAND algorithm for hyperparameter optimization.
    input          : $R$, $\eta$ (default $\eta = 3$)
    initialization : $s_{\max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{\max} + 1)R$
1   for $s \in \{s_{\max}, s_{\max} - 1, \dots, 0\}$ do
2   |   $n = \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil$,        $r = R\eta^{-s}$
```

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.

1.15.4 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:

```
run_experiments({...}, scheduler=MedianStoppingRule())
```

class `ray.tune.schedulers.`**`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 training result 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 training result 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.

1.16 Tune Search Algorithms

Tune provides various hyperparameter search algorithms to efficiently optimize your model. Tune allows you to use different search algorithms in combination with different trial schedulers. Tune will by default implicitly use the Variant Generation algorithm to create trials.

You can utilize these search algorithms as follows:

```
run_experiments(experiments, search_alg=SearchAlgorithm(...))
```

Currently, Tune offers the following search algorithms:

- Grid Search and Random Search
- HyperOpt
- SigOpt

1.16.1 Variant Generation (Grid Search/Random Search)

By default, Tune uses the default search space and variant generation process to create and queue trials. This supports random search and grid search as specified by the `config` parameter of the Experiment.

class `ray.tune.suggest.`**`BasicVariantGenerator`**

Bases: `ray.tune.suggest.search.SearchAlgorithm`

Uses Tune's variant generation for resolving variables.

See also: *ray.tune.suggest.variant_generator*.

Example

```
>>> searcher = BasicVariantGenerator()
>>> searcher.add_configurations({"experiment": { ... }})
>>> list_of_trials = searcher.next_trials()
>>> searcher.is_finished == True
```

Note that other search algorithms will not necessarily extend this class and may require a different search space declaration than the default Tune format.

1.16.2 BayesOpt Search

The `BayesOptSearch` is a SearchAlgorithm that is backed by the bayesian-optimization package to perform sequential model-based hyperparameter optimization. Note that this class does not extend `ray.tune.suggest.BasicVariantGenerator`, so you will not be able to use Tune's default variant generation/search space declaration when using BayesOptSearch.

In order to use this search algorithm, you will need to install Bayesian Optimization via the following command:

```
$ pip install bayesian-optimization
```

This algorithm requires setting a search space and defining a utility function. You can use BayesOptSearch like follows:

```
run_experiments(experiment_config, search_alg=BayesOptSearch(bayesopt_space, utility_
↪kwargs=utility_params, ... ))
```

An example of this can be found in bayesopt_example.py.

class `ray.tune.suggest.`**`BayesOptSearch`**(*space*, *max_concurrent=10*, *reward_attr='episode_reward_mean'*, *utility_kwargs=None*, *random_state=1*, *verbose=0*, ***kwargs*)

Bases: `ray.tune.suggest.suggestion.SuggestionAlgorithm`

A wrapper around BayesOpt to provide trial suggestions.

Requires BayesOpt to be installed. You can install BayesOpt with the command: *pip install bayesian-optimization*.

Parameters

- **space** (*dict*) – Continuous search space. Parameters will be sampled from this space which will be used to run trials.
- **max_concurrent** (*int*) – Number of maximum concurrent trials. Defaults to 10.
- **reward_attr** (*str*) – The training result objective value attribute. This refers to an increasing value.
- **utility_kwargs** (*dict*) – Parameters to define the utility function. Must provide values for the keys *kind*, *kappa*, and *xi*.
- **random_state** (*int*) – Used to initialize BayesOpt.
- **verbose** (*int*) – Sets verbosity level for BayesOpt packages.

Example

```
>>> space = {
>>>     'width': (0, 20),
>>>     'height': (-100, 100),
>>> }
>>> config = {
>>>     "my_exp": {
>>>         "run": "exp",
>>>         "num_samples": 10 if args.smoke_test else 1000,
>>>         "stop": {
>>>             "training_iteration": 100
>>>         },
>>>     }
>>> }
>>> algo = BayesOptSearch(
>>>     space, max_concurrent=4, reward_attr="neg_mean_loss")
```

1.16.3 HyperOpt Search (Tree-structured Parzen Estimators)

The `HyperOptSearch` is a SearchAlgorithm that is backed by HyperOpt to perform sequential model-based hyperparameter optimization. Note that this class does not extend `ray.tune.suggest.BasicVariantGenerator`, so you will not be able to use Tune's default variant generation/search space declaration when using HyperOptSearch.

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
```

This algorithm requires using the HyperOpt search space specification. You can use HyperOptSearch like follows:

```
run_experiments(experiment_config, search_alg=HyperOptSearch(hyperopt_space, ... ))
```

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

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

Bases: `ray.tune.suggest.suggestion.SuggestionAlgorithm`

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

- **`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 training result objective value attribute. This refers to an increasing value.

Example

```
>>> space = {
>>>     'width': hp.uniform('width', 0, 20),
>>>     'height': hp.uniform('height', -100, 100),
>>>     'activation': hp.choice("activation", ["relu", "tanh"])
>>> }
>>> config = {
>>>     "my_exp": {
>>>         "run": "exp",
>>>         "num_samples": 10 if args.smoke_test else 1000,
>>>         "stop": {
>>>             "training_iteration": 100
>>>         },
>>>     }
>>> }
>>> algo = HyperOptSearch(
>>>     space, max_concurrent=4, reward_attr="neg_mean_loss")
```

1.16.4 SigOpt Search

The `SigOptSearch` is a SearchAlgorithm that is backed by SigOpt to perform sequential model-based hyperparameter optimization. Note that this class does not extend `ray.tune.suggest.BasicVariantGenerator`, so you will not be able to use Tune's default variant generation/search space declaration when using SigOptSearch.

In order to use this search algorithm, you will need to install SigOpt via the following command:

```
$ pip install sigopt
```

This algorithm requires the user to have a SigOpt API key to make requests to the API. Store the API token as an environment variable named `SIGOPT_KEY` like follows:

```
$ export SIGOPT_KEY= ...
```

This algorithm requires using the SigOpt experiment and space specification. You can use SigOptSearch like follows:

```
run_experiments(experiment_config, search_alg=SigOptSearch(sigopt_space, ... ))
```

An example of this can be found in sigopt_example.py.

class `ray.tune.suggest.`**`SigOptSearch`**(*space*, *name='Default Tune Experiment'*, *max_concurrent=1*, *reward_attr='episode_reward_mean'*, ***kwargs*)

Bases: `ray.tune.suggest.suggestion.SuggestionAlgorithm`

A wrapper around SigOpt to provide trial suggestions.

Requires SigOpt to be installed. Requires user to store their SigOpt API key locally as an environment variable at *SIGOPT_KEY*.

Parameters

- **space** (*list of dict*) – SigOpt configuration. Parameters will be sampled from this configuration and will be used to override parameters generated in the variant generation process.
- **name** (*str*) – Name of experiment. Required by SigOpt.

- **max_concurrent** (*int*) – Number of maximum concurrent trials supported based on the user's SigOpt plan. Defaults to 1.
- **reward_attr** (*str*) – The training result objective value attribute. This refers to an increasing value.

Example

```
>>> space = [
>>>     {
>>>         'name': 'width',
>>>         'type': 'int',
>>>         'bounds': {
>>>             'min': 0,
>>>             'max': 20
>>>         },
>>>     },
>>>     {
>>>         'name': 'height',
>>>         'type': 'int',
>>>         'bounds': {
>>>             'min': -100,
>>>             'max': 100
>>>         },
>>>     },
>>> ]
>>> config = {
>>>     "my_exp": {
>>>         "run": "exp",
>>>         "num_samples": 10 if args.smoke_test else 1000,
>>>         "stop": {
>>>             "training_iteration": 100
>>>         },
>>>     }
>>> }
>>> algo = SigOptSearch(
>>>     parameters, name="SigOpt Example Experiment",
>>>     max_concurrent=1, reward_attr="neg_mean_loss")
```

1.16.5 Contributing a New Algorithm

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

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*.

add_configurations(*experiments*)

Tracks given experiment specifications.

Parameters **experiments** (*Experiment | list | dict*) – Experiments to run.

next_trials()

Provides Trial objects to be queued into the TrialRunner.

Returns Returns a list of trials.

Return type trials (list)

on_trial_result(*trial_id*, *result*)

Called on each intermediate result returned by a trial.

This will only be called when the trial is in the RUNNING state.

Parameters **trial_id** – Identifier for the trial.

on_trial_complete(*trial_id*, *result=None*, *error=False*, *early_terminated=False*)

Notification for the completion of trial.

Parameters

- **trial_id** – Identifier for the trial.
- **result** (*dict*) – Defaults to None. A dict will be provided with this notification when the trial is in the RUNNING state AND either completes naturally or by manual termination.
- **error** (*bool*) – Defaults to False. True if the trial is in the RUNNING state and errors.
- **early_terminated** (*bool*) – Defaults to False. True if the trial is stopped while in PAUSED or PENDING state.

is_finished()

Returns True if no trials left to be queued into TrialRunner.

Can return True before all trials have finished executing.

Model-Based Suggestion Algorithms

Often times, hyperparameter search algorithms are model-based and may be quite simple to implement. For this, one can extend the following abstract class and implement `on_trial_result`, `on_trial_complete`, and `_suggest`. The abstract class will take care of Tune-specific boilerplate such as creating Trials and queuing trials:

class ray.tune.suggest.**SuggestionAlgorithm**

Bases: `ray.tune.suggest.search.SearchAlgorithm`

Abstract class for suggestion-based algorithms.

Custom search algorithms can extend this class easily by overriding the *_suggest* method provide generated parameters for the trials.

To track suggestions and their corresponding evaluations, the method *_suggest* will be passed a trial_id, which will be used in subsequent notifications.

Example

```
>>> suggester = SuggestionAlgorithm()
>>> suggester.add_configurations({ ... })
>>> new_parameters = suggester._suggest()
>>> suggester.on_trial_complete(trial_id, result)
>>> better_parameters = suggester._suggest()
```

_suggest (*trial_id*)
Queries the algorithm to retrieve the next set of parameters.

Parameters **trial_id** – Trial ID used for subsequent notifications.

Returns

Configuration for a trial, if possible. Else, returns None, which will temporarily stop the TrialRunner from querying.

Return type dict|None

Example

```
>>> suggester = SuggestionAlgorithm(max_concurrent=1)
>>> suggester.add_configurations({ ... })
>>> parameters_1 = suggester._suggest()
>>> parameters_2 = suggester._suggest()
>>> parameters_2 is None
>>> suggester.on_trial_complete(trial_id, result)
>>> parameters_2 = suggester._suggest()
>>> parameters_2 is not None
```

1.17 Tune Package Reference

1.17.1 ray.tune

ray.tune.**grid_search** (*values*)
Convenience method for specifying grid search over a value.

Parameters **values** – An iterable whose parameters will be gridded.

ray.tune.**register_env** (*name*, *env_creator*)
Register a custom environment for use with RLlib.

Parameters

- **name** (*str*) – Name to register.
- **env_creator** (*obj*) – Function that creates an env.

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.**run_experiments** (*experiments*, *search_alg=None*, *scheduler=None*, *with_server=False*, *server_port=4321*, *verbose=2*, *resume=False*, *queue_trials=False*, *trial_executor=None*, *raise_on_failed_trial=True*)
Runs and blocks until all trials finish.

Parameters

- **experiments** (*Experiment | list | dict*) – Experiments to run. Will be passed to *search_alg* via *add_configurations*.
- **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** (*int*) – 0, 1, or 2. Verbosity mode. 0 = silent, 1 = only status updates, 2 = status and trial results.
- **resume** (*bool | "prompt"*) – If checkpoint exists, the experiment will resume from there. If resume is "prompt", Tune will prompt if checkpoint detected.
- **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.
- **trial_executor** (*TrialExecutor*) – Manage the execution of trials.
- **raise_on_failed_trial** (*bool*) – Raise TuneError if there exists failed trial (of ERROR state) when the experiments complete.

Examples

```
>>> experiment_spec = Experiment("experiment", my_func)
>>> run_experiments(experiments=experiment_spec)
```

```
>>> experiment_spec = {"experiment": {"run": my_func}}
>>> run_experiments(experiments=experiment_spec)
```

```
>>> run_experiments(
>>>     experiments=experiment_spec,
>>>     scheduler=MedianStoppingRule(...))
```

```
>>> run_experiments(
>>>     experiments=experiment_spec,
>>>     search_alg=SearchAlgorithm(),
>>>     scheduler=MedianStoppingRule(...))
```

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

class ray.tune.**Experiment**(*name*, *run*, *stop=None*, *config=None*, *resources_per_trial=None*, *num_samples=1*, *local_dir=None*, *upload_dir=None*, *trial_name_creator=None*, *custom_loggers=None*, *sync_function=None*, *checkpoint_freq=0*, *checkpoint_at_end=False*, *export_formats=None*, *max_failures=3*, *restore=None*, *repeat=None*, *trial_resources=None*)

Tracks experiment specifications.

Parameters

- **name** (*str*) – Name of experiment.
- **run** (*function | class | str*) – The algorithm or model to train. This may refer to the name of a built-on algorithm (e.g. RLLib's DQN or PPO), a user-defined trainable function or class, or the string identifier of a trainable function or class registered in the tune registry.
- **stop** (*dict*) – The stopping criteria. The keys may be any field in the return result of 'train()', whichever is reached first. Defaults to empty dict.
- **config** (*dict*) – Algorithm-specific configuration for Tune variant generation (e.g. env, hyperparams). Defaults to empty dict. Custom search algorithms may ignore this.
- **resources_per_trial** (*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 in `Trainable.default_resource_request()`.
- **num_samples** (*int*) – Number of times to sample from the hyperparameter space. Defaults to 1. If *grid_search* is provided as an argument, the grid will be repeated *num_samples* of times.
- **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://bucket`).
- **trial_name_creator** (*func*) – Optional function for generating the trial string representation.
- **custom_loggers** (*list*) – List of custom logger creators to be used with each Trial. See *ray/tune/logger.py*.
- **sync_function** (*func | str*) – Function for syncing the local_dir to upload_dir. If string, then it must be a string template for syncer to run. If not provided, the sync command defaults to standard S3 or gsutil sync comamnds.
- **checkpoint_freq** (*int*) – How many training iterations between checkpoints. A value of 0 (default) disables checkpointing.
- **checkpoint_at_end** (*bool*) – Whether to checkpoint at the end of the experiment regardless of the checkpoint_freq. Default is False.
- **export_formats** (*list*) – List of formats that exported at the end of the experiment. Default is None.
- **max_failures** (*int*) – Try to recover a trial from its last checkpoint at least this many times. Only applies if checkpointing is enabled. Setting to -1 will lead to infinite recovery retries. Defaults to 3.
- **restore** (*str*) – Path to checkpoint. Only makes sense to set if running 1 trial. Defaults to None.
- **repeat** – Deprecated and will be removed in future versions of Ray. Use *num_samples* instead.
- **trial_resources** – Deprecated and will be removed in future versions of Ray. Use *resources_per_trial* instead.

Examples

```
>>> experiment_spec = Experiment(
>>>     "my_experiment_name",
>>>     my_func,
>>>     stop={"mean_accuracy": 100},
>>>     config={
>>>         "alpha": tune.grid_search([0.2, 0.4, 0.6]),
>>>         "beta": tune.grid_search([1, 2]),
>>>     },
>>>     resources_per_trial={
>>>         "cpu": 1,
>>>         "gpu": 0
>>>     },
>>>     num_samples=10,
>>>     local_dir="~/ray_results",
>>>     upload_dir="s3://your_bucket/path",
>>>     checkpoint_freq=10,
>>>     max_failures=2)
```

classmethod from_json(*name*, *spec*)

Generates an Experiment object from JSON.

Parameters

- **name** (*str*) – Name of Experiment.
- **spec** (*dict*) – JSON configuration of experiment.

class ray.tune.**function**(*func*)

Wraps *func* to make sure it is not expanded during resolution.

The use of function arguments in tune configs must be disambiguated by either wrapped the function in tune.eval() or tune.function().

Parameters **func** – A function literal.

class ray.tune.**sample_from**(*func*)

Specify that tune should sample configuration values from this function.

The use of function arguments in tune configs must be disambiguated by either wrapped the function in tune.eval() or tune.function().

Parameters **func** – An callable function to draw a sample from.

class ray.tune.**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.

Note that, if you don't require checkpoint/restore functionality, then instead of implementing this class you can also get away with supplying just a `my_train(config, reporter)` function to the config. The function will be automatically converted to this interface (sans checkpoint functionality).

classmethod default_resource_request(*config*)

Returns the resource requirement for the given configuration.

This can be overriden by sub-classes to set the correct trial resource allocation, so the user does not need to.

classmethod resource_help(*config*)
Returns a help string for configuring this trainable's resources.

train()
Runs one logical iteration of training.

Subclasses should override `_train()` instead to return results. This class automatically fills the following fields in the result:

done (bool): training is terminated. Filled only if not provided.

time_this_iter_s (float): Time in seconds this iteration took to run. This may be overriden in order to override the system-computed time difference.

time_total_s (float): Accumulated time in seconds for this entire experiment.

experiment_id (str): Unique string identifier for this experiment. This id is preserved across checkpoint / restore calls.

training_iteration (int): The index of this training iteration, e.g. call to train().

pid (str): The pid of the training process.

date (str): A formatted date of when the result was processed.

timestamp (str): A UNIX timestamp of when the result was processed.

hostname (str): Hostname of the machine hosting the training process.

node_ip (str): Node ip of the machine hosting the training process.

Returns A dict that describes training progress.

save(*checkpoint_dir=None*)
Saves the current model state to a checkpoint.

Subclasses should override `_save()` instead to save state. This method dumps additional metadata alongside the saved path.

Parameters **checkpoint_dir** (`str`) – Optional dir to place the checkpoint.

Returns Checkpoint path that may be passed to restore().

save_to_object()
Saves the current model state to a Python object. It also saves to disk but does not return the checkpoint path.

Returns Object holding checkpoint data.

restore(*checkpoint_path*)
Restores training state from a given model checkpoint.

These checkpoints are returned from calls to save().

Subclasses should override `_restore()` instead to restore state. This method restores additional metadata saved with the checkpoint.

restore_from_object(*obj*)
Restores training state from a checkpoint object.

These checkpoints are returned from calls to save_to_object().

export_model (*export_formats*, *export_dir=None*)
Exports model based on export_formats.

Subclasses should override _export_model() to actually export model to local directory.

Parameters

- **export_formats** (`list`) – List of formats that should be exported.
- **export_dir** (`str`) – Optional dir to place the exported model. Defaults to self.logdir.

Returns A dict that maps ExportFormats to successfully exported models.

reset_config (*new_config*)
Resets configuration without restarting the trial.

Parameters **new_config** (`dir`) – Updated hyperparameter configuration for the trainable.

Returns True if configuration reset successfully else False.

stop ()
Releases all resources used by this trainable.

_train ()
Subclasses should override this to implement train().

Returns A dict that describes training progress.

_save (*checkpoint_dir*)
Subclasses should override this to implement save().

Parameters **checkpoint_dir** (`str`) – The directory where the checkpoint file must be stored.

Returns

If string, the return value is expected to be the checkpoint path that will be passed to *_restore()*. If dict, the return value will be automatically serialized by Tune and passed to *_restore()*.

Return type checkpoint (str | dict)

Examples

```
>>> print(trainable1._save("/tmp/checkpoint_1"))
"/tmp/checkpoint_1/my_checkpoint_file"
>>> print(trainable2._save("/tmp/checkpoint_2"))
{"some": "data"}
```

_restore (*checkpoint*)
Subclasses should override this to implement restore().

Parameters **checkpoint** (`str | dict`) – Value as returned by *_save*. If a string, then it is the checkpoint path.

_setup (*config*)
Subclasses should override this for custom initialization.

Parameters **config** (`dict`) – Hyperparameters and other configs given. Copy of *self.config*.

_log_result (*result*)
Subclasses can optionally override this to customize logging.

Parameters **result** (`dict`) – Training result returned by _train().

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

_export_model(*export_formats*, *export_dir*)
Subclasses should override this to export model.

Parameters

- **export_formats** (*list*) – List of formats that should be exported.
- **export_dir** (*str*) – Directory to place exported models.

Returns A dict that maps ExportFormats to successfully exported models.

class ray.tune.function_runner.**StatusReporter**
Object passed into your function that you can report status through.

Example

```
>>> def trainable_function(config, reporter):
>>>     assert isinstance(reporter, StatusReporter)
>>>     reporter(timesteps_total=1)
```

__call__(***kwargs*)
Report updated training status.

Pass in *done=True* when the training job is completed.

Parameters **kwargs** – Latest training result status.

Example

```
>>> reporter(mean_accuracy=1, training_iteration=4)
>>> reporter(mean_accuracy=1, training_iteration=4, done=True)
```

1.17.2 ray.tune.schedulers

class ray.tune.schedulers.**TrialScheduler**
Bases: object

Interface for implementing a Trial Scheduler class.

CONTINUE = 'CONTINUE'
Status for continuing trial execution

PAUSE = 'PAUSE'
Status for pausing trial execution

STOP = 'STOP'
Status for stopping trial execution

on_trial_add(*trial_runner*, *trial*)
Called when a new trial is added to the trial runner.

on_trial_error(*trial_runner*, *trial*)
Notification for the error of trial.

This will only be called when the trial is in the RUNNING state.

on_trial_result(*trial_runner*, *trial*, *result*)
Called on each intermediate result returned by a trial.

At this point, the trial scheduler can make a decision by returning one of CONTINUE, PAUSE, and STOP. This will only be called when the trial is in the RUNNING state.

on_trial_complete(*trial_runner*, *trial*, *result*)
Notification for the completion of trial.

This will only be called when the trial is in the RUNNING state and either completes naturally or by manual termination.

on_trial_remove(*trial_runner*, *trial*)
Called to remove trial.

This is called when the trial is in PAUSED or PENDING state. Otherwise, call *on_trial_complete*.

choose_trial_to_run(*trial_runner*)
Called to choose a new trial to run.

This should return one of the trials in trial_runner that is in the PENDING or PAUSED state. This function must be idempotent.

If no trial is ready, return None.

debug_string()
Returns a human readable message for printing to the console.

class ray.tune.schedulers.**HyperBandScheduler**(*time_attr='training_iteration'*, *reward_attr='episode_reward_mean'*, *max_t=81*)
Bases: `ray.tune.schedulers.trial_scheduler.FIFOScheduler`

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 Tune, all you need to do is specify the max length of time a trial can run *max_t*, the time units *time_attr*, and the name of the reported objective value *reward_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 *episode_mean_reward* attr, construct:

`HyperBand('time_total_s', 'episode_reward_mean', max_t=600)`

Note that Tune's stopping criteria will be applied in conjunction with HyperBand's early stopping mechanisms.

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

Parameters

- **time_attr** (*str*) – The training result 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 training result objective value attribute. As with *time_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 max_t time units (determined by time_attr) have passed. The scheduler will terminate trials after this time has passed. Note that this is different from the semantics of *max_t* as mentioned in the original HyperBand paper.

on_trial_add(*trial_runner*, *trial*)
Adds new trial.

On a new trial add, if current bracket is not filled, add to current bracket. Else, if current band is not filled, create new bracket, add to current bracket. Else, create new iteration, create new bracket, add to bracket.

on_trial_result(*trial_runner*, *trial*, *result*)
If bracket is finished, all trials will be stopped.

If a given trial finishes and bracket iteration is not done, the trial will be paused and resources will be given up.

This scheduler will not start trials but will stop trials. The current running trial will not be handled, as the trialrunner will be given control to handle it.

on_trial_remove(*trial_runner*, *trial*)
Notification when trial terminates.

Trial info is removed from bracket. Triggers halving if bracket is not finished.

on_trial_complete(*trial_runner*, *trial*, *result*)
Cleans up trial info from bracket if trial completed early.

on_trial_error(*trial_runner*, *trial*)
Cleans up trial info from bracket if trial errored early.

choose_trial_to_run(*trial_runner*)
Fair scheduling within iteration by completion percentage.

List of trials not used since all trials are tracked as state of scheduler. If iteration is occupied (ie, no trials to run), then look into next iteration.

debug_string()
This provides a progress notification for the algorithm.

For each bracket, the algorithm will output a string as follows:

> Bracket(Max Size (n)=5, Milestone (r)=33, completed=14.6%): {PENDING: 2, RUNNING: 3, TERMINATED: 2}

"Max Size" indicates the max number of pending/running experiments set according to the Hyperband algorithm.

"Milestone" indicates the iterations a trial will run for before the next halving will occur.

"Completed" indicates an approximate progress metric. Some brackets, like ones that are unfilled, will not reach 100%.

class ray.tune.schedulers.**AsyncHyperBandScheduler**(*time_attr='training_iteration'*, *reward_attr='episode_reward_mean'*, *max_t=100*, *grace_period=10*, *reduction_factor=3*, *brackets=3*)
Bases: `ray.tune.schedulers.trial_scheduler.FIFOScheduler`

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=S1Y7OOlRZ

Parameters

- **time_attr** (*str*) – A training result 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 training result 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.

on_trial_add(*trial_runner*, *trial*)
Called when a new trial is added to the trial runner.

on_trial_result(*trial_runner*, *trial*, *result*)
Called on each intermediate result returned by a trial.

At this point, the trial scheduler can make a decision by returning one of CONTINUE, PAUSE, and STOP. This will only be called when the trial is in the RUNNING state.

on_trial_complete(*trial_runner*, *trial*, *result*)
Notification for the completion of trial.

This will only be called when the trial is in the RUNNING state and either completes naturally or by manual termination.

on_trial_remove(*trial_runner*, *trial*)
Called to remove trial.

This is called when the trial is in PAUSED or PENDING state. Otherwise, call *on_trial_complete*.

debug_string()
Returns a human readable message for printing to the console.

class ray.tune.schedulers.**MedianStoppingRule**(*time_attr='time_total_s'*, *reward_attr='episode_reward_mean'*, *grace_period=60.0*, *min_samples_required=3*, *hard_stop=True*, *verbose=True*)

Bases: `ray.tune.schedulers.trial_scheduler.FIFOScheduler`

Implements the median stopping rule as described in the Vizier paper:

https://research.google.com/pubs/pub46180.html

Parameters

- **time_attr** (*str*) – The training result 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 training result 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.

on_trial_result (*trial_runner, trial, result*)
Callback for early stopping.

This stopping rule stops a running trial if the trial's best objective value by step *t* is strictly worse than the median of the running averages of all completed trials' objectives reported up to step *t*.

on_trial_complete (*trial_runner, trial, result*)
Notification for the completion of trial.

This will only be called when the trial is in the RUNNING state and either completes naturally or by manual termination.

on_trial_remove (*trial_runner, trial*)
Marks trial as completed if it is paused and has previously ran.

debug_string ()
Returns a human readable message for printing to the console.

class `ray.tune.schedulers.`**FIFOScheduler**
Bases: `ray.tune.schedulers.trial_scheduler.TrialScheduler`

Simple scheduler that just runs trials in submission order.

on_trial_add (*trial_runner, trial*)
Called when a new trial is added to the trial runner.

on_trial_error (*trial_runner, trial*)
Notification for the error of trial.

This will only be called when the trial is in the RUNNING state.

on_trial_result (*trial_runner, trial, result*)
Called on each intermediate result returned by a trial.

At this point, the trial scheduler can make a decision by returning one of CONTINUE, PAUSE, and STOP. This will only be called when the trial is in the RUNNING state.

on_trial_complete (*trial_runner, trial, result*)
Notification for the completion of trial.

This will only be called when the trial is in the RUNNING state and either completes naturally or by manual termination.

on_trial_remove (*trial_runner, trial*)
Called to remove trial.

This is called when the trial is in PAUSED or PENDING state. Otherwise, call *on_trial_complete*.

choose_trial_to_run (*trial_runner*)
Called to choose a new trial to run.

This should return one of the trials in trial_runner that is in the PENDING or PAUSED state. This function must be idempotent.

If no trial is ready, return None.

debug_string()

Returns a human readable message for printing to the console.

class ray.tune.schedulers.**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*)

Bases: `ray.tune.schedulers.trial_scheduler.FIFOScheduler`

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 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 training result 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 training result 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

```
>>> pbt = PopulationBasedTraining(
>>>     time_attr="training_iteration",
>>>     reward_attr="episode_reward_mean",
>>>     perturbation_interval=10,  # every 10 `time_attr` units
>>>                                # (training_iterations in this case)
```

(continues on next page)

(continued from previous page)

```
>>>     hyperparam_mutations={
>>>         # Perturb factor1 by scaling it by 0.8 or 1.2. Resampling
>>>         # resets it to a value sampled from the lambda function.
>>>         "factor_1": lambda: random.uniform(0.0, 20.0),
>>>         # Perturb factor2 by changing it to an adjacent value, e.g.
>>>         # 10 -> 1 or 10 -> 100. Resampling will choose at random.
>>>         "factor_2": [1, 10, 100, 1000, 10000],
>>>     })
>>> run_experiments({...}, scheduler=pbt)
```

`on_trial_add`(*trial_runner*, *trial*)
Called when a new trial is added to the trial runner.

`on_trial_result`(*trial_runner*, *trial*, *result*)
Called on each intermediate result returned by a trial.

At this point, the trial scheduler can make a decision by returning one of CONTINUE, PAUSE, and STOP. This will only be called when the trial is in the RUNNING state.

`choose_trial_to_run`(*trial_runner*)
Ensures all trials get fair share of time (as defined by time_attr).

This enables the PBT scheduler to support a greater number of concurrent trials than can fit in the cluster at any given time.

`debug_string`()
Returns a human readable message for printing to the console.

1.17.3 ray.tune.suggest

class `ray.tune.suggest.`**`SearchAlgorithm`**
Bases: `object`

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*.

`add_configurations`(*experiments*)
Tracks given experiment specifications.

Parameters **`experiments`** (*`Experiment | list | dict`*) – Experiments to run.

`next_trials`()
Provides Trial objects to be queued into the TrialRunner.

Returns Returns a list of trials.

Return type trials (list)

`on_trial_result`(*trial_id*, *result*)
Called on each intermediate result returned by a trial.

This will only be called when the trial is in the RUNNING state.

Parameters **`trial_id`** – Identifier for the trial.

on_trial_complete(*trial_id*, *result=None*, *error=False*, *early_terminated=False*)
Notification for the completion of trial.

Parameters

- **trial_id** – Identifier for the trial.
- **result** (*dict*) – Defaults to None. A dict will be provided with this notification when the trial is in the RUNNING state AND either completes naturally or by manual termination.
- **error** (*bool*) – Defaults to False. True if the trial is in the RUNNING state and errors.
- **early_terminated** (*bool*) – Defaults to False. True if the trial is stopped while in PAUSED or PENDING state.

is_finished()
Returns True if no trials left to be queued into TrialRunner.

Can return True before all trials have finished executing.

class ray.tune.suggest.**BasicVariantGenerator**
Bases: ray.tune.suggest.search.SearchAlgorithm

Uses Tune's variant generation for resolving variables.

See also: *ray.tune.suggest.variant_generator*.

Example

```
>>> searcher = BasicVariantGenerator()
>>> searcher.add_configurations({"experiment": { ... }})
>>> list_of_trials = searcher.next_trials()
>>> searcher.is_finished == True
```

add_configurations(*experiments*)
Chains generator given experiment specifications.

Parameters experiments (*Experiment | list | dict*) – Experiments to run.

next_trials()
Provides Trial objects to be queued into the TrialRunner.

Returns Returns a list of trials.

Return type trials (list)

is_finished()
Returns True if no trials left to be queued into TrialRunner.

Can return True before all trials have finished executing.

class ray.tune.suggest.**BayesOptSearch**(*space*, *max_concurrent=10*, *reward_attr='episode_reward_mean'*, *utility_kwargs=None*, *random_state=1*, *verbose=0*, ***kwargs*)
Bases: ray.tune.suggest.suggestion.SuggestionAlgorithm

A wrapper around BayesOpt to provide trial suggestions.

Requires BayesOpt to be installed. You can install BayesOpt with the command: *pip install bayesian-optimization*.

Parameters

- **space** (*dict*) – Continuous search space. Parameters will be sampled from this space which will be used to run trials.
- **max_concurrent** (*int*) – Number of maximum concurrent trials. Defaults to 10.
- **reward_attr** (*str*) – The training result objective value attribute. This refers to an increasing value.
- **utility_kwargs** (*dict*) – Parameters to define the utility function. Must provide values for the keys *kind*, *kappa*, and *xi*.
- **random_state** (*int*) – Used to initialize BayesOpt.
- **verbose** (*int*) – Sets verbosity level for BayesOpt packages.

Example

```
>>> space = {
>>>     'width': (0, 20),
>>>     'height': (-100, 100),
>>> }
>>> config = {
>>>     "my_exp": {
>>>         "run": "exp",
>>>         "num_samples": 10 if args.smoke_test else 1000,
>>>         "stop": {
>>>             "training_iteration": 100
>>>         },
>>>     }
>>> }
>>> algo = BayesOptSearch(
>>>     space, max_concurrent=4, reward_attr="neg_mean_loss")
```

on_trial_result (*trial_id*, *result*)
Called on each intermediate result returned by a trial.

This will only be called when the trial is in the RUNNING state.

Parameters trial_id – Identifier for the trial.

on_trial_complete (*trial_id*, *result=None*, *error=False*, *early_terminated=False*)
Passes the result to BayesOpt unless early terminated or errored

class ray.tune.suggest.**HyperOptSearch** (*space*, *max_concurrent=10*, *reward_attr='episode_reward_mean'*, ***kwargs*)
Bases: `ray.tune.suggest.suggestion.SuggestionAlgorithm`

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

- **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 training result objective value attribute. This refers to an increasing value.

Example

```
>>> space = {
>>>     'width': hp.uniform('width', 0, 20),
>>>     'height': hp.uniform('height', -100, 100),
>>>     'activation': hp.choice("activation", ["relu", "tanh"])
>>> }
>>> config = {
>>>     "my_exp": {
>>>         "run": "exp",
>>>         "num_samples": 10 if args.smoke_test else 1000,
>>>         "stop": {
>>>             "training_iteration": 100
>>>         },
>>>     }
>>> }
>>> algo = HyperOptSearch(
>>>     space, max_concurrent=4, reward_attr="neg_mean_loss")
```

on_trial_result(*trial_id*, *result*)
Called on each intermediate result returned by a trial.

This will only be called when the trial is in the RUNNING state.

Parameters **trial_id** – Identifier for the trial.

on_trial_complete(*trial_id*, *result=None*, *error=False*, *early_terminated=False*)
Passes the result to HyperOpt unless early terminated or errored.

The result is internally negated when interacting with HyperOpt so that HyperOpt can "maximize" this value, as it minimizes on default.

class ray.tune.suggest.**SigOptSearch**(*space*, *name='Default Tune Experiment'*, *max_concurrent=1*, *reward_attr='episode_reward_mean'*, ***kwargs*)
Bases: `ray.tune.suggest.suggestion.SuggestionAlgorithm`

A wrapper around SigOpt to provide trial suggestions.

Requires SigOpt to be installed. Requires user to store their SigOpt API key locally as an environment variable at *SIGOPT_KEY*.

Parameters

- **space** (*list of dict*) – SigOpt configuration. Parameters will be sampled from this configuration and will be used to override parameters generated in the variant generation process.
- **name** (*str*) – Name of experiment. Required by SigOpt.
- **max_concurrent** (*int*) – Number of maximum concurrent trials supported based on the user's SigOpt plan. Defaults to 1.
- **reward_attr** (*str*) – The training result objective value attribute. This refers to an increasing value.

Example

```
>>> space = [
>>>     {
>>>         'name': 'width',
>>>         'type': 'int',
>>>         'bounds': {
>>>             'min': 0,
>>>             'max': 20
>>>         },
>>>     },
>>>     {
>>>         'name': 'height',
>>>         'type': 'int',
>>>         'bounds': {
>>>             'min': -100,
>>>             'max': 100
>>>         },
>>>     },
>>> ]
>>> config = {
>>>     "my_exp": {
>>>         "run": "exp",
>>>         "num_samples": 10 if args.smoke_test else 1000,
>>>         "stop": {
>>>             "training_iteration": 100
>>>         },
>>>     }
>>> }
>>> algo = SigOptSearch(
>>>     parameters, name="SigOpt Example Experiment",
>>>     max_concurrent=1, reward_attr="neg_mean_loss")
```

`on_trial_result`(*trial_id*, *result*)
Called on each intermediate result returned by a trial.

This will only be called when the trial is in the RUNNING state.

Parameters **`trial_id`** – Identifier for the trial.

`on_trial_complete`(*trial_id*, *result=None*, *error=False*, *early_terminated=False*)
Passes the result to SigOpt unless early terminated or errored.

If a trial fails, it will be reported as a failed Observation, telling the optimizer that the Suggestion led to a metric failure, which updates the feasible region and improves parameter recommendation.

Creates SigOpt Observation object for trial.

class `ray.tune.suggest.`**`SuggestionAlgorithm`**
Bases: `ray.tune.suggest.search.SearchAlgorithm`

Abstract class for suggestion-based algorithms.

Custom search algorithms can extend this class easily by overriding the *_suggest* method provide generated parameters for the trials.

To track suggestions and their corresponding evaluations, the method *_suggest* will be passed a trial_id, which will be used in subsequent notifications.

Example

```
>>> suggester = SuggestionAlgorithm()
>>> suggester.add_configurations({ ... })
>>> new_parameters = suggester._suggest()
>>> suggester.on_trial_complete(trial_id, result)
>>> better_parameters = suggester._suggest()
```

add_configurations(*experiments*)
Chains generator given experiment specifications.

Parameters **experiments** (*Experiment | list | dict*) – Experiments to run.

next_trials()
Provides a batch of Trial objects to be queued into the TrialRunner.

A batch ends when self._trial_generator returns None.

Returns Returns a list of trials.

Return type trials (list)

_generate_trials(*experiment_spec*, *output_path=”*)
Generates trials with configurations from *_suggest*.

Creates a trial_id that is passed into *_suggest*.

Yields Trial objects constructed according to *spec*

is_finished()
Returns True if no trials left to be queued into TrialRunner.

Can return True before all trials have finished executing.

_suggest(*trial_id*)
Queries the algorithm to retrieve the next set of parameters.

Parameters **trial_id** – Trial ID used for subsequent notifications.

Returns

Configuration for a trial, if possible. Else, returns None, which will temporarily stop the TrialRunner from querying.

Return type dict|None

Example

```
>>> suggester = SuggestionAlgorithm(max_concurrent=1)
>>> suggester.add_configurations({ ... })
>>> parameters_1 = suggester._suggest()
>>> parameters_2 = suggester._suggest()
>>> parameters_2 is None
>>> suggester.on_trial_complete(trial_id, result)
>>> parameters_2 = suggester._suggest()
>>> parameters_2 is not None
```

1.17.4 ray.tune.logger

class `ray.tune.logger.`**`Logger`**(*config*, *logdir*, *upload_uri=None*)

Logging interface for ray.tune.

By default, the UnifiedLogger implementation is used which logs results in multiple formats (TensorBoard, rllab/viskit, plain json, custom loggers) at once.

Parameters

- **`config`** – Configuration passed to all logger creators.
- **`logdir`** – Directory for all logger creators to log to.
- **`upload_uri`** (`str`) – Optional URI where the logdir is sync'ed to.

`on_result`(*result*)

Given a result, appends it to the existing log.

`close`()

Releases all resources used by this logger.

`flush`()

Flushes all disk writes to storage.

1.18 Tune Examples

In our repository, we provide a variety of examples for the various use cases and features of Tune.

If any example is broken, or if you'd like to add an example to this page, feel free to raise an issue on our Github repository.

1.18.1 General Examples

- **async_hyperband_example:** Example of using a Trainable class with AsyncHyperBandScheduler.
- **hyperband_example:** Example of using a Trainable class with HyperBandScheduler. Also uses the Experiment class API for specifying the experiment configuration.
- **hyperopt_example:** Optimizes a basic function using the function-based API and the HyperOptSearch (SearchAlgorithm wrapper for HyperOpt TPE). Also uses the AsyncHyperBandScheduler.
- **pbt_example:** Example of using a Trainable class with PopulationBasedTraining scheduler.
- **pbt_ppo_example:** Example of optimizing a distributed RLlib algorithm (PPO) with the PopulationBased-Training scheduler.
- **logging_example:** Example of custom loggers and custom trial directory naming.

1.18.2 Keras Examples

- **tune_mnist_keras:** Converts the Keras MNIST example to use Tune with the function-based API and a Keras callback. Also shows how to easily convert something relying on argparse to use Tune.

1.18.3 PyTorch Examples

- **mnist_pytorch:** Converts the PyTorch MNIST example to use Tune with the function-based API. Also shows how to easily convert something relying on argparse to use Tune.
- **mnist_pytorch_trainable:** Converts the PyTorch MNIST example to use Tune with Trainable API. Also uses the HyperBandScheduler and checkpoints the model at the end.

1.18.4 TensorFlow Examples

- **tune_mnist_ray:** A basic example of tuning a TensorFlow model on MNIST using the Trainable class.
- **tune_mnist_ray_hyperband:** A basic example of tuning a TensorFlow model on MNIST using the Trainable class and the HyperBand scheduler.
- **tune_mnist_async_hyperband:** Example of tuning a TensorFlow model on MNIST using AsyncHyperBand.

1.18.5 Contributed Examples

- **pbt_tune_cifar10_with_keras:** A contributed example of tuning a Keras model on CIFAR10 with the PopulationBasedTraining scheduler.
- **genetic_example:** Optimizing the michalewicz function using the contributed GeneticSearch search algorithm with AsyncHyperBandScheduler.

1.19 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. To get started, take a look over the custom env example and the API documentation.

1.19.1 Installation

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

```
pip install tensorflow  # or tensorflow-gpu
pip install ray[rllib]  # also recommended: ray[debug]
```

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
```

1.19.2 Training APIs

- Command-line
- Configuration
- Python API
- Debugging
- REST API

1.19.3 Environments

- RLlib Environments Overview
- OpenAI Gym
- Vectorized
- Multi-Agent and Hierarchical
- Interfacing with External Agents
- Batch Asynchronous

1.19.4 Algorithms

- High-throughput architectures
  - Distributed Prioritized Experience Replay (Ape-X)
  - Importance Weighted Actor-Learner Architecture (IMPALA)
  - Asynchronous Proximal Policy Optimization (APPO)
- Gradient-based
  - Advantage Actor-Critic (A2C, A3C)
  - Deep Deterministic Policy Gradients (DDPG, TD3)
  - Deep Q Networks (DQN, Rainbow, Parametric DQN)
  - Policy Gradients
  - Proximal Policy Optimization (PPO)
- Derivative-free
  - Augmented Random Search (ARS)
  - Evolution Strategies
- Multi-agent specific
  - QMIX Monotonic Value Factorisation (QMIX, VDN, IQN)
- Offline
  - Advantage Re-Weighted Imitation Learning (MARWIL)

1.19.5 Models and Preprocessors

- RLlib Models and Preprocessors Overview
- Built-in Models and Preprocessors
- Custom Models (TensorFlow)
- Custom Models (PyTorch)
- Custom Preprocessors
- Customizing Policy Graphs
- Variable-length / Parametric Action Spaces
- Model-Based Rollouts

1.19.6 Offline Datasets

- Working with Offline Datasets
- Input API
- Output API

1.19.7 Development

- Development Install
- API Stability
- Features
- Benchmarks
- Contributing Algorithms

1.19.8 Concepts

- Policy Graphs
- Policy Evaluation
- Policy Optimization

1.19.9 Package Reference

- ray.rllib.agents
- ray.rllib.env
- ray.rllib.evaluation
- ray.rllib.models
- ray.rllib.optimizers
- ray.rllib.utils

1.19.10 Examples

You can find an index of RLlib code examples on this page. This includes tuned hyperparameters, demo scripts on how to use specific features of RLlib, and several community examples of applications built on RLlib.

1.19.11 Troubleshooting

If you encounter errors like *blas_thread_init: pthread_create: Resource temporarily unavailable* when using many workers, try setting `OMP_NUM_THREADS=1`. Similarly, check configured system limits with *ulimit -a* for other resource limit errors.

For debugging unexpected hangs or performance problems, you can run `ray stack` to dump the stack traces of all Ray workers on the current node. This requires py-spy to be installed.

1.20 RLlib Training APIs

1.20.1 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:

```
rllib train --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

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

The `rllib train` command (same as the `train.py` script in the repo) has a number of options you can show by running:

```
rllib train --help
-or-
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`, `A2C`, `A3C`, `IMPALA`, `ES`, `DDPG`, `DQN`, `APEX`, and `APEX_DDPG`).

Evaluating Trained Agents

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

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

```
rllib rollout \
    ~/ray_results/default/DQN_CartPole-v0_0upjmdgr0/checkpoint_1/checkpoint-1 \
    --run DQN --env CartPole-v0 --steps 10000
```

The rollout.py helper script reconstructs a DQN agent from the checkpoint located at ~/ray_results/default/DQN_CartPole-v0_0upjmdgr0/checkpoint_1/checkpoint-1 and renders its behavior in the environment specified by --env.

1.20.2 Configuration

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 A2C by specifying 8 workers through the config flag.

```
rllib train --env=PongDeterministic-v4 --run=A2C --config '{"num_workers": 8}'
```

Specifying Resources

You can control the degree of parallelism used by setting the num_workers hyperparameter for most agents. The number of GPUs the driver should use can be set via the num_gpus option. Similarly, the resource allocation to workers can be controlled via num_cpus_per_worker, num_gpus_per_worker, and custom_resources_per_worker. The number of GPUs can be a fractional quantity to allocate only a fraction of a GPU. For example, with DQN you can pack five agents onto one GPU by setting num_gpus: 0.2.

Common Parameters

The following is a list of the common agent hyperparameters:

```
COMMON_CONFIG = {
    # === Debugging ===
    # Whether to write episode stats and videos to the agent log dir
    "monitor": False,
    # Set the ray.rllib.* log level for the agent process and its evaluators
    "log_level": "INFO",
    # Callbacks that will be run during various phases of training. These all
    # take a single "info" dict as an argument. For episode callbacks, custom
    # metrics can be attached to the episode by updating the episode object's
    # custom metrics dict (see examples/custom_metrics_and_callbacks.py).
    "callbacks": {
        "on_episode_start": None,   # arg: {"env": .., "episode": ...}
        "on_episode_step": None,    # arg: {"env": .., "episode": ...}
        "on_episode_end": None,     # arg: {"env": .., "episode": ...}
        "on_sample_end": None,      # arg: {"samples": .., "evaluator": ...}
        "on_train_result": None,    # arg: {"agent": ..., "result": ...}
    },

    # === Policy ===
    # Arguments to pass to model. See models/catalog.py for a full list of the
    # available model options.
    "model": MODEL_DEFAULTS,
    # Arguments to pass to the policy optimizer. These vary by optimizer.
    "optimizer": {},
```

(continues on next page)

(continued from previous page)

```
    # === Environment ===
    # Discount factor of the MDP
    "gamma": 0.99,
    # Number of steps after which the episode is forced to terminate
    "horizon": None,
    # Arguments to pass to the env creator
    "env_config": {},
    # Environment name can also be passed via config
    "env": None,
    # Whether to clip rewards prior to experience postprocessing. Setting to
    # None means clip for Atari only.
    "clip_rewards": None,
    # Whether to np.clip() actions to the action space low/high range spec.
    "clip_actions": True,
    # Whether to use rllib or deepmind preprocessors by default
    "preprocessor_pref": "deepmind",

    # === Resources ===
    # Number of actors used for parallelism
    "num_workers": 2,
    # Number of GPUs to allocate to the driver. Note that not all algorithms
    # can take advantage of driver GPUs. This can be fraction (e.g., 0.3 GPUs).
    "num_gpus": 0,
    # Number of CPUs to allocate per worker.
    "num_cpus_per_worker": 1,
    # Number of GPUs to allocate per worker. This can be fractional.
    "num_gpus_per_worker": 0,
    # Any custom resources to allocate per worker.
    "custom_resources_per_worker": {},
    # Number of CPUs to allocate for the driver. Note: this only takes effect
    # when running in Tune.
    "num_cpus_for_driver": 1,

    # === Execution ===
    # Number of environments to evaluate vectorwise per worker.
    "num_envs_per_worker": 1,
    # Default sample batch size
    "sample_batch_size": 200,
    # Training batch size, if applicable. Should be >= sample_batch_size.
    # Samples batches will be concatenated together to this size for training.
    "train_batch_size": 200,
    # Whether to rollout "complete_episodes" or "truncate_episodes"
    "batch_mode": "truncate_episodes",
    # Whether to use a background thread for sampling (slightly off-policy)
    "sample_async": False,
    # Element-wise observation filter, either "NoFilter" or "MeanStdFilter"
    "observation_filter": "NoFilter",
    # Whether to synchronize the statistics of remote filters.
    "synchronize_filters": True,
    # Configure TF for single-process operation by default
    "tf_session_args": {
        # note: overriden by `local_evaluator_tf_session_args`
        "intra_op_parallelism_threads": 2,
        "inter_op_parallelism_threads": 2,
        "gpu_options": {
            "allow_growth": True,
        },
```

(continues on next page)

(continued from previous page)

```
        "log_device_placement": False,
        "device_count": {
            "CPU": 1
        },
        "allow_soft_placement": True,  # required by PPO multi-gpu
    },
    # Override the following tf session args on the local evaluator
    "local_evaluator_tf_session_args": {
        # Allow a higher level of parallelism by default, but not unlimited
        # since that can cause crashes with many concurrent drivers.
        "intra_op_parallelism_threads": 8,
        "inter_op_parallelism_threads": 8,
    },
    # Whether to LZ4 compress individual observations
    "compress_observations": False,
    # Drop metric batches from unresponsive workers after this many seconds
    "collect_metrics_timeout": 180,

    # === Offline Datasets ===
    # __sphinx_doc_input_begin__
    # Specify how to generate experiences:
    #  - "sampler": generate experiences via online simulation (default)
    #  - a local directory or file glob expression (e.g., "/tmp/*.json")
    #  - a list of individual file paths/URIs (e.g., ["/tmp/1.json",
    #    "s3://bucket/2.json"])
    #  - a dict with string keys and sampling probabilities as values (e.g.,
    #    {"sampler": 0.4, "/tmp/*.json": 0.4, "s3://bucket/expert.json": 0.2}).
    #  - a function that returns a rllib.offline.InputReader
    "input": "sampler",
    # Specify how to evaluate the current policy. This only makes sense to set
    # when the input is not already generating simulation data:
    #  - None: don't evaluate the policy. The episode reward and other
    #    metrics will be NaN if using offline data.
    #  - "simulation": run the environment in the background, but use
    #    this data for evaluation only and not for learning.
    "input_evaluation": None,
    # Whether to run postprocess_trajectory() on the trajectory fragments from
    # offline inputs. Note that postprocessing will be done using the *current*
    # policy, not the *behaviour* policy, which is typically undesirable for
    # on-policy algorithms.
    "postprocess_inputs": False,
    # __sphinx_doc_input_end__
    # __sphinx_doc_output_begin__
    # Specify where experiences should be saved:
    #  - None: don't save any experiences
    #  - "logdir" to save to the agent log dir
    #  - a path/URI to save to a custom output directory (e.g., "s3://bucket/")
    #  - a function that returns a rllib.offline.OutputWriter
    "output": None,
    # What sample batch columns to LZ4 compress in the output data.
    "output_compress_columns": ["obs", "new_obs"],
    # Max output file size before rolling over to a new file.
    "output_max_file_size": 64 * 1024 * 1024,
    # __sphinx_doc_output_end__

    # === Multiagent ===
    "multiagent": {
```

(continues on next page)

(continued from previous page)

```
        # Map from policy ids to tuples of (policy_graph_cls, obs_space,
        # act_space, config). See policy_evaluator.py for more info.
        "policy_graphs": {},
        # Function mapping agent ids to policy ids.
        "policy_mapping_fn": None,
        # Optional whitelist of policies to train, or None for all policies.
        "policies_to_train": None,
    },
}
```

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 `rllib train` command as follows:

```
rllib train -f /path/to/tuned/example.yaml
```

1.20.3 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 (for a more complete example, see custom_env.py):

```
import ray
import ray.rllib.agents.ppo as ppo
from ray.tune.logger import pretty_print

ray.init()
config = ppo.DEFAULT_CONFIG.copy()
config["num_gpus"] = 0
config["num_workers"] = 1
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(pretty_print(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:

```
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_gpus": 0,
            "num_workers": 1,
            "lr": 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: ~/ray_results/my_experiment
PENDING trials:
 - PPO_CartPole-v0_2_lr=0.0001:     PENDING
RUNNING trials:
 - PPO_CartPole-v0_0_lr=0.01:       RUNNING [pid=21940], 16 s, 4013 ts, 22 rew
 - PPO_CartPole-v0_1_lr=0.001:      RUNNING [pid=21942], 27 s, 8111 ts, 54.7 rew
```

Custom Training Workflows

In the basic training example, Tune will call `train()` on your agent once per iteration and report the new training results. Sometimes, it is desirable to have full control over training, but still run inside Tune. Tune supports custom trainable functions that can be used to implement custom training workflows (example).

Accessing Policy 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.get_policy()` or `agent.local_evaluator`, but note that updates here may not be immediately reflected in remote replicas if you have configured `num_workers > 0`. For example, to access the weights of a local TF policy, you can run `agent.get_policy().get_weights()`. This is also equivalent to `agent.local_evaluator.policy_map["default"].get_weights()`:

```
# Get weights of the default local policy
agent.get_policy().get_weights()

# Same as above
agent.local_evaluator.policy_map["default"].get_weights()
```

(continues on next page)

(continued from previous page)

```
# Get list of weights of each evaluator, including remote replicas
agent.optimizer.foreach_evaluator(lambda ev: ev.get_policy().get_weights())

# Same as above
agent.optimizer.foreach_evaluator_with_index(lambda ev, i: ev.get_policy().get_
↪weights())
```

Global Coordination

Sometimes, it is necessary to coordinate between pieces of code that live in different processes managed by RLlib. For example, it can be useful to maintain a global average of a certain variable, or centrally control a hyperparameter used by policies. Ray provides a general way to achieve this through *named actors* (learn more about Ray actors here). As an example, consider maintaining a shared global counter that is incremented by environments and read periodically from your driver program:

```
from ray.experimental import named_actors

@ray.remote
class Counter:
   def __init__(self):
      self.count = 0
   def inc(self, n):
      self.count += n
   def get(self):
      return self.count

# on the driver
counter = Counter.remote()
named_actors.register_actor("global_counter", counter)
print(ray.get(counter.get.remote()))  # get the latest count

# in your envs
counter = named_actors.get_actor("global_counter")
counter.inc.remote(1)  # async call to increment the global count
```

Ray actors provide high levels of performance, so in more complex cases they can be used implement communication patterns such as parameter servers and allreduce.

Callbacks and Custom Metrics

You can provide callback functions to be called at points during policy evaluation. These functions have access to an info dict containing state for the current episode. Custom state can be stored for the episode in the `info["episode"].user_data` dict, and custom scalar metrics reported by saving values to the `info["episode"].custom_metrics` dict. These custom metrics will be aggregated and reported as part of training results. The following example (full code here) logs a custom metric from the environment:

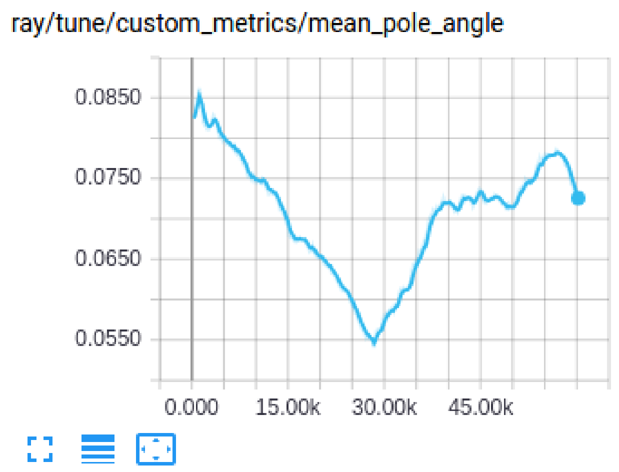
```
def on_episode_start(info):
    print(info.keys())  # -> "env", 'episode"
    episode = info["episode"]
    print("episode {} started".format(episode.episode_id))
    episode.user_data["pole_angles"] = []
```

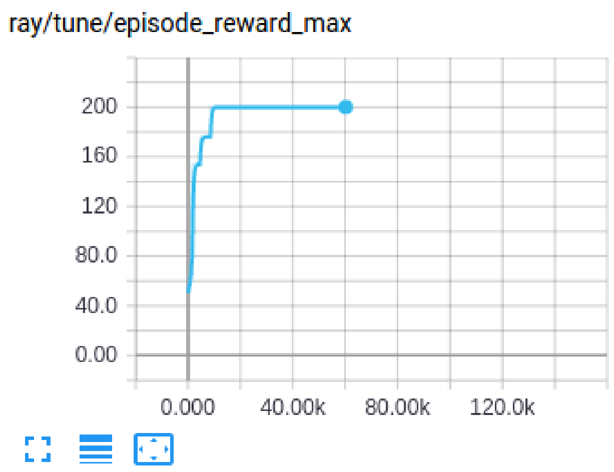
(continues on next page)

(continued from previous page)

```python
def on_episode_step(info):
    episode = info["episode"]
    pole_angle = abs(episode.last_observation_for()[2])
    episode.user_data["pole_angles"].append(pole_angle)

def on_episode_end(info):
    episode = info["episode"]
    pole_angle = np.mean(episode.user_data["pole_angles"])
    print("episode {} ended with length {} and pole angles {}".format(
        episode.episode_id, episode.length, pole_angle))
    episode.custom_metrics["pole_angle"] = pole_angle

def on_train_result(info):
    print("agent.train() result: {} -> {} episodes".format(
        info["agent"].__name__, info["result"]["episodes_this_iter"]))

ray.init()
trials = tune.run_experiments({
    "test": {
        "env": "CartPole-v0",
        "run": "PG",
        "config": {
            "callbacks": {
                "on_episode_start": tune.function(on_episode_start),
                "on_episode_step": tune.function(on_episode_step),
                "on_episode_end": tune.function(on_episode_end),
                "on_train_result": tune.function(on_train_result),
            },
        },
    }
})
```

Custom metrics can be accessed and visualized like any other training result:





Example: Curriculum Learning

Let's look at two ways to use the above APIs to implement curriculum learning. In curriculum learning, the agent task is adjusted over time to improve the learning process. Suppose that we have an environment class with a set_phase() method that we can call to adjust the task difficulty over time:

Approach 1: Use the Agent API and update the environment between calls to train(). This example shows the agent being run inside a Tune function:

```
import ray
from ray import tune
from ray.rllib.agents.ppo import PPOAgent

def train(config, reporter):
    agent = PPOAgent(config=config, env=YourEnv)
    while True:
        result = agent.train()
        reporter(**result)
        if result["episode_reward_mean"] > 200:
            phase = 2
        elif result["episode_reward_mean"] > 100:
            phase = 1
        else:
            phase = 0
        agent.optimizer.foreach_evaluator(lambda ev: ev.env.set_phase(phase))

ray.init()
tune.run_experiments({
    "curriculum": {
        "run": train,
        "config": {
            "num_gpus": 0,
            "num_workers": 2,
        },
        "resources_per_trial": {
            "cpu": 1,
            "gpu": lambda spec: spec.config.num_gpus,
            "extra_cpu": lambda spec: spec.config.num_workers,
        },
    },
})
```

Approach 2: Use the callbacks API to update the environment on new training results:

```
import ray
from ray import tune

def on_train_result(info):
    result = info["result"]
    if result["episode_reward_mean"] > 200:
        phase = 2
    elif result["episode_reward_mean"] > 100:
        phase = 1
    else:
        phase = 0
    agent = info["agent"]
    agent.optimizer.foreach_evaluator(lambda ev: ev.env.set_phase(phase))

ray.init()
tune.run_experiments({
    "curriculum": {
        "run": "PPO",
```

(continues on next page)

(continued from previous page)

```
        "env": YourEnv,
        "config": {
            "callbacks": {
                "on_train_result": tune.function(on_train_result),
            },
        },
    },
})
```

1.20.4 Debugging

Gym Monitor

The `"monitor": true` config can be used to save Gym episode videos to the result dir. For example:

```
rllib train --env=PongDeterministic-v4 \
    --run=A2C --config '{"num_workers": 2, "monitor": true}'

# videos will be saved in the ~/ray_results/<experiment> dir, for example
openaigym.video.0.31401.video000000.meta.json
openaigym.video.0.31401.video000000.mp4
openaigym.video.0.31403.video000000.meta.json
openaigym.video.0.31403.video000000.mp4
```

Log Verbosity

You can control the agent log level via the `"log_level"` flag. Valid values are "INFO" (default), "DEBUG", "WARN", and "ERROR". This can be used to increase or decrease the verbosity of internal logging. For example:

```
rllib train --env=PongDeterministic-v4 \
    --run=A2C --config '{"num_workers": 2, "log_level": "DEBUG"}'
```

Stack Traces

You can use the `ray stack` command to dump the stack traces of all the Python workers on a single node. This can be useful for debugging unexpected hangs or performance issues.

1.20.5 REST API

In some cases (i.e., when interacting with an externally hosted simulator or production 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 external agents interface.

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.

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**(*external_env*, *address*, *port*)
REST server than can be launched from a ExternalEnv.

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(ExternalEnv):
       def __init__(self):
           ExternalEnv.__init__(
               self, spaces.Discrete(2),
```

(continues on next page)

(continued from previous page)

```
                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)
```

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.

1.21 RLlib Environments

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

Compatibility matrix:

Algorithm	Discrete Actions	Continuous Actions	Multi-Agent	Recurrent Policies
A2C, A3C	**Yes** +parametric	**Yes**	**Yes**	**Yes**
PPO, APPO	**Yes** +parametric	**Yes**	**Yes**	**Yes**
PG	**Yes** +parametric	**Yes**	**Yes**	**Yes**
IMPALA	**Yes** +parametric	No	**Yes**	**Yes**
DQN, Rainbow	**Yes** +parametric	No	**Yes**	No
DDPG, TD3	No	**Yes**	**Yes**	No
APEX-DQN	**Yes** +parametric	No	**Yes**	No
APEX-DDPG	No	**Yes**	**Yes**	No
ES	**Yes**	**Yes**	No	No
ARS	**Yes**	**Yes**	No	No
QMIX	**Yes**	No	**Yes**	**Yes**
MARWIL	**Yes** +parametric	**Yes**	**Yes**	**Yes**

You can pass either a string name or a Python class to specify an environment. By default, strings will be interpreted as a gym environment name. Custom env classes passed directly to the agent must take a single `env_config` parameter in their constructor:

```
import gym, ray
from ray.rllib.agents import ppo

class MyEnv(gym.Env):
    def __init__(self, env_config):
        self.action_space = <gym.Space>
        self.observation_space = <gym.Space>
    def reset(self):
        return <obs>
    def step(self, action):
        return <obs>, <reward: float>, <done: bool>, <info: dict>

ray.init()
trainer = ppo.PPOAgent(env=MyEnv, config={
    "env_config": {},  # config to pass to env class
})

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

You can also register a custom env creator function with a string name. This function must take a single `env_config` parameter and return an env instance:

```
from ray.tune.registry import register_env

def env_creator(env_config):
    return MyEnv(...)  # return an env instance

register_env("my_env", env_creator)
trainer = ppo.PPOAgent(env="my_env")
```

For a full runnable code example using the custom environment API, see custom_env.py.

1.21.1 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:

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

1.21.2 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.

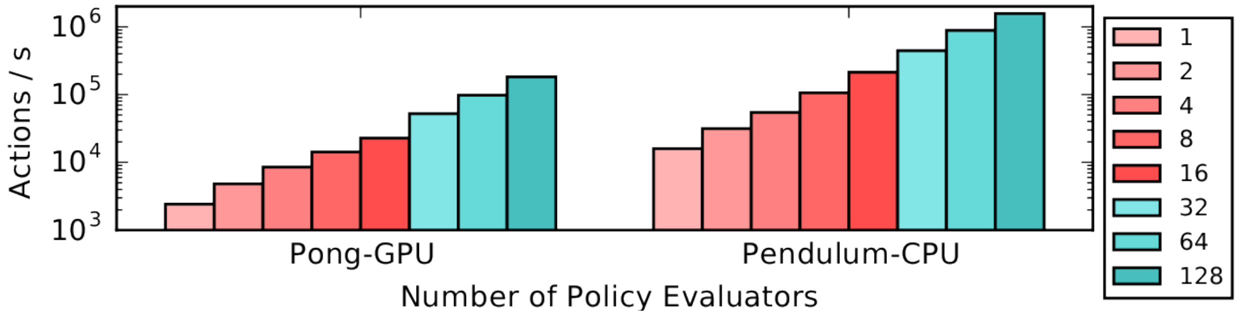
Performance

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

1. **Vectorization within a single process:** Though many envs can 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.



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`.

Expensive Environments

Some environments may be very resource-intensive to create. RLlib will create `num_workers + 1` copies of the environment since one copy is needed for the driver process. To avoid paying the extra overhead of the driver copy, which is needed to access the env's action and observation spaces, you can defer environment initialization until `reset()` is called.

1.21.3 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()`.

1.21.4 Multi-Agent and Hierarchical

Note: Learn more about multi-agent reinforcement learning in RLlib by reading the blog post.

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:

```
# 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:

```
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␣
↪policies
    },
})

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. Metrics are reported for each policy separately, for example:

```
Result for PPO_multi_cartpole_0:
  episode_len_mean: 34.025862068965516
  episode_reward_max: 159.0
  episode_reward_mean: 86.06896551724138
  info:
    policy_0:
      cur_lr: 4.999999873689376e-05
      entropy: 0.6833480000495911
      kl: 0.010264254175126553
      policy_loss: -11.95590591430664
      total_loss: 197.7039794921875
      vf_explained_var: 0.0010995268821716309
      vf_loss: 209.6578826904297
    policy_1:
      cur_lr: 4.999999873689376e-05
      entropy: 0.6827034950256348
      kl: 0.01119876280426979
      policy_loss: -8.787769317626953
      total_loss: 88.26161193847656
      vf_explained_var: 0.0005457401275634766
      vf_loss: 97.0471420288086
  policy_reward_mean:
    policy_0: 21.194444444444443
    policy_1: 21.798387096774192
```

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.

Hierarchical Environments

Hierarchical training can sometimes be implemented as a special case of multi-agent RL. For example, consider a three-level hierarchy of policies, where a top-level policy issues high level actions that are executed at finer timescales by a mid-level and low-level policy. The following timeline shows one step of the top-level policy, which corresponds to two mid-level actions and five low-level actions:

```
top_level ---------------------------------------------------------------> top_level -
↪-->
mid_level_0 -------------------------------> mid_level_0 ----------------> mid_level_
↪1 ->
low_level_0 -> low_level_0 -> low_level_0 -> low_level_1 -> low_level_1 -> low_level_
↪2 ->
```

This can be implemented as a multi-agent environment with three types of agents. Each higher-level action creates a new lower-level agent instance with a new id (e.g., low_level_0, low_level_1, low_level_2 in the above example). These lower-level agents pop in existence at the start of higher-level steps, and terminate when their higher-level action ends. Their experiences are aggregated by policy, so from RLlib's perspective it's just optimizing three different types of policies. The configuration might look something like this:

```
"multiagent": {
    "policy_graphs": {
        "top_level": (some_policy_graph, ...),
        "mid_level": (some_policy_graph, ...),
        "low_level": (some_policy_graph, ...),
    },
    "policy_mapping_fn":
        lambda agent_id:
            "low_level" if agent_id.startswith("low_level_") else
            "mid_level" if agent_id.startswith("mid_level_") else "top_level"
    "policies_to_train": ["top_level"],
},
```

In this setup, the appropriate rewards for training lower-level agents must be provided by the multi-agent env implementation. The environment class is also responsible for routing between the agents, e.g., conveying goals from higher-level agents to lower-level agents as part of the lower-level agent observation.

See this file for a runnable example: hierarchical_training.py.

Grouping Agents

It is common to have groups of agents in multi-agent RL. RLlib treats agent groups like a single agent with a Tuple action and observation space. The group agent can then be assigned to a single policy for centralized execution, or to specialized multi-agent policies such as Q-Mix that implement centralized training but decentralized execution. You can use the `MultiAgentEnv.with_agent_groups()` method to define these groups:

```
    @PublicAPI
    def with_agent_groups(self, groups, obs_space=None, act_space=None):
        """Convenience method for grouping together agents in this env.

        An agent group is a list of agent ids that are mapped to a single
        logical agent. All agents of the group must act at the same time in the
        environment. The grouped agent exposes Tuple action and observation
        spaces that are the concatenated action and obs spaces of the
        individual agents.

        The rewards of all the agents in a group are summed. The individual
        agent rewards are available under the "individual_rewards" key of the
        group info return.

        Agent grouping is required to leverage algorithms such as Q-Mix.

        This API is experimental.

        Arguments:
            groups (dict): Mapping from group id to a list of the agent ids
                of group members. If an agent id is not present in any group
                value, it will be left ungrouped.
            obs_space (Space): Optional observation space for the grouped
                env. Must be a tuple space.
            act_space (Space): Optional action space for the grouped env.
                Must be a tuple space.

        Examples:
            >>> env = YourMultiAgentEnv(...)
            >>> grouped_env = env.with_agent_groups(env, {
```

(continues on next page)

(continued from previous page)

```
            ...   "group1": ["agent1", "agent2", "agent3"],
            ...   "group2": ["agent4", "agent5"],
            ... })
        """

        from ray.rllib.env.group_agents_wrapper import _GroupAgentsWrapper
        return _GroupAgentsWrapper(self, groups, obs_space, act_space)
```

Variable-Sharing Between Policies

RLlib will create each policy's model in a separate `tf.variable_scope`. However, variables can still be shared between policies by explicitly entering a globally shared variable scope with `tf.VariableScope(reuse=tf.AUTO_REUSE)`:

```
with tf.variable_scope(
        tf.VariableScope(tf.AUTO_REUSE, "name_of_global_shared_scope"),
        reuse=tf.AUTO_REUSE,
        auxiliary_name_scope=False):
    <create the shared layers here>
```

There is a full example of this in the example training script.

Implementing a Centralized Critic

Implementing a centralized critic that takes as input the observations and actions of other concurrent agents requires the definition of custom policy graphs. It can be done as follows:

1. Querying the critic: this can be done in the `postprocess_trajectory` method of a custom policy graph, which has full access to the policies and observations of concurrent agents via the `other_agent_batches` and `episode` arguments. The batch of critic predictions can then be added to the postprocessed trajectory. Here's an example:

```
def postprocess_trajectory(self, sample_batch, other_agent_batches, episode):
    agents = ["agent_1", "agent_2", "agent_3"]  # simple example of 3 agents
    global_obs_batch = np.stack(
        [other_agent_batches[agent_id][1]["obs"] for agent_id in agents],
        axis=1)
    # add the global obs and global critic value
    sample_batch["global_obs"] = global_obs_batch
    sample_batch["central_vf"] = self.sess.run(
        self.critic_network, feed_dict={"obs": global_obs_batch})
    return sample_batch
```

2. Updating the critic: the centralized critic loss can be added to the loss of the custom policy graph, the same as with any other value function. For an example of defining loss inputs, see the PGPolicyGraph example.

1.21.5 Interfacing with External Agents

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 for an agent to query a service that serves policy decisions, and for that service to learn from experience over time. This case also naturally arises with **external simulators** that run independently outside the control of RLlib, but may still want to leverage RLlib for training.

RLlib provides the ExternalEnv class for this purpose. Unlike other envs, ExternalEnv has its own thread of control. At any point, agents on 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.

ExternalEnv 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`.

Logging off-policy actions

ExternalEnv 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).

Data ingest

The `log_action` API of ExternalEnv 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 ExternalEnv 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.

See also:

RLlib I/O provides higher-level interfaces for working with offline experience datasets.

1.21.6 Batch Asynchronous

The lowest-level "catch-all" environment supported by RLlib is BaseEnv. BaseEnv 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 BaseEnv by RLlib so that there is a common internal path for policy evaluation.

1.22 RLlib Algorithms

1.22.1 High-throughput architectures

Distributed Prioritized Experience Replay (Ape-X)

[paper] [implementation] Ape-X variations of DQN, DDPG, and QMIX (APEX_DQN, APEX_DDPG, APEX_QMIX) 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.

Tuned examples: PongNoFrameskip-v4, Pendulum-v0, MountainCarContinuous-v0, {BeamRider,Breakout,Qbert,SpaceInvaders}NoFrameskip-v4.

Atari results @10M steps: more details

Atari env	RLlib Ape-X 8-workers	Mnih et al Async DQN 16-workers
BeamRider	6134	~6000
Breakout	123	~50
Qbert	15302	~1200
SpaceInvaders	686	~600

Scalability:

Atari env	RLlib Ape-X 8-workers @1 hour	Mnih et al Async DQN 16-workers @1 hour
BeamRider	4873	~1000
Breakout	77	~10
Qbert	4083	~500
SpaceInvaders	646	~300

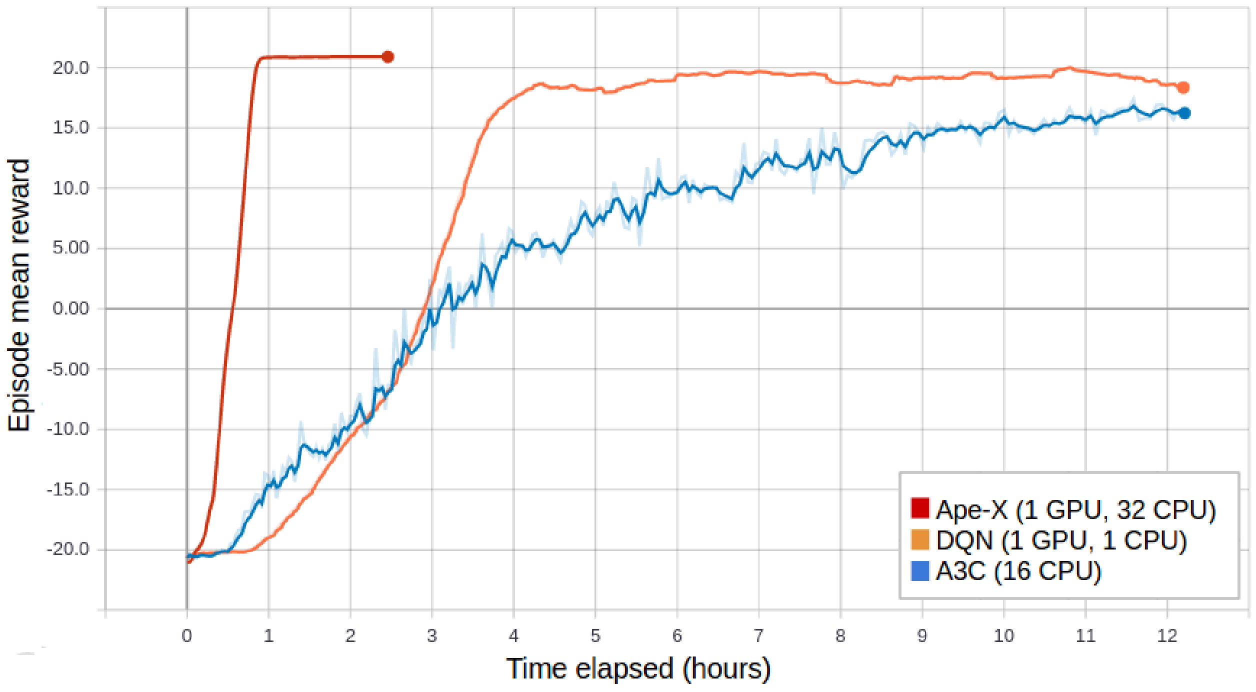


Fig. 1: Ape-X using 32 workers in RLlib vs vanilla DQN (orange) and A3C (blue) on PongNoFrameskip-v4.

Ape-X specific configs (see also common configs):

```
APEX_DEFAULT_CONFIG = merge_dicts(
    DQN_CONFIG,  # see also the options in dqn.py, which are also supported
    {
        "optimizer_class": "AsyncReplayOptimizer",
        "optimizer": merge_dicts(
            DQN_CONFIG["optimizer"], {
                "max_weight_sync_delay": 400,
                "num_replay_buffer_shards": 4,
                "debug": False
            }),
        "n_step": 3,
        "num_gpus": 1,
```

(continues on next page)

(continued from previous page)

```
        "num_workers": 32,
        "buffer_size": 2000000,
        "learning_starts": 50000,
        "train_batch_size": 512,
        "sample_batch_size": 50,
        "target_network_update_freq": 500000,
        "timesteps_per_iteration": 25000,
        "per_worker_exploration": True,
        "worker_side_prioritization": True,
        "min_iter_time_s": 30,
    },
)
```

Importance Weighted Actor-Learner Architecture (IMPALA)

[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. Multiple learner GPUs and experience replay are also supported.

Tuned examples: PongNoFrameskip-v4, vectorized configuration, multi-gpu configuration, {BeamRider,Breakout,Qbert,SpaceInvaders}NoFrameskip-v4

Atari results @10M steps: more details

Atari env	RLlib IMPALA 32-workers	Mnih et al A3C 16-workers
BeamRider	2071	~3000
Breakout	385	~150
Qbert	4068	~1000
SpaceInvaders	719	~600

Scalability:

Atari env	RLlib IMPALA 32-workers @1 hour	Mnih et al A3C 16-workers @1 hour
BeamRider	3181	~1000
Breakout	538	~10
Qbert	10850	~500
SpaceInvaders	843	~300

IMPALA-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # V-trace params (see vtrace.py).
    "vtrace": True,
    "vtrace_clip_rho_threshold": 1.0,
    "vtrace_clip_pg_rho_threshold": 1.0,

    # System params.
    #
    # == Overview of data flow in IMPALA ==
    # 1. Policy evaluation in parallel across `num_workers` actors produces
    #    batches of size `sample_batch_size * num_envs_per_worker`.
    # 2. If enabled, the replay buffer stores and produces batches of size
```

(continues on next page)

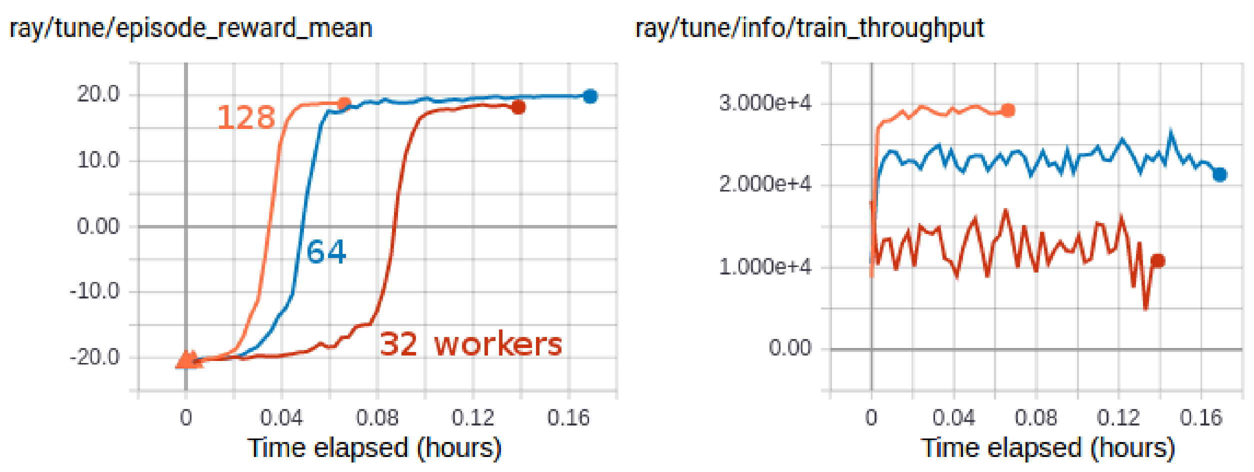


Fig. 2: Multi-GPU IMPALA scales up to solve PongNoFrameskip-v4 in ~3 minutes using a pair of V100 GPUs and 128 CPU workers. The maximum training throughput reached is ~30k transitions per second (~120k environment frames per second).

(continued from previous page)

```
    #    `sample_batch_size * num_envs_per_worker`.
    # 3. If enabled, the minibatch ring buffer stores and replays batches of
    #    size `train_batch_size` up to `num_sgd_iter` times per batch.
    # 4. The learner thread executes data parallel SGD across `num_gpus` GPUs
    #    on batches of size `train_batch_size`.
    #
    "sample_batch_size": 50,
    "train_batch_size": 500,
    "min_iter_time_s": 10,
    "num_workers": 2,
    # number of GPUs the learner should use.
    "num_gpus": 1,
    # set >1 to load data into GPUs in parallel. Increases GPU memory usage
    # proportionally with the number of buffers.
    "num_data_loader_buffers": 1,
    # how many train batches should be retained for minibatching. This conf
    # only has an effect if `num_sgd_iter > 1`.
    "minibatch_buffer_size": 1,
    # number of passes to make over each train batch
    "num_sgd_iter": 1,
    # set >0 to enable experience replay. Saved samples will be replayed with
    # a p:1 proportion to new data samples.
    "replay_proportion": 0.0,
    # number of sample batches to store for replay. The number of transitions
    # saved total will be (replay_buffer_num_slots * sample_batch_size).
    "replay_buffer_num_slots": 100,
    # level of queuing for sampling.
    "max_sample_requests_in_flight_per_worker": 2,
    # max number of workers to broadcast one set of weights to
    "broadcast_interval": 1,

    # Learning params.
    "grad_clip": 40.0,
    # either "adam" or "rmsprop"
    "opt_type": "adam",
```

(continues on next page)

(continued from previous page)

```
    "lr": 0.0005,
    "lr_schedule": None,
    # rmsprop considered
    "decay": 0.99,
    "momentum": 0.0,
    "epsilon": 0.1,
    # balancing the three losses
    "vf_loss_coeff": 0.5,
    "entropy_coeff": -0.01,
})
```

Asynchronous Proximal Policy Optimization (APPO)

[paper] [implementation] We include an asynchronous variant of Proximal Policy Optimization (PPO) based on the IMPALA architecture. This is similar to IMPALA but using a surrogate policy loss with clipping. Compared to synchronous PPO, APPO is more efficient in wall-clock time due to its use of asynchronous sampling. Using a clipped loss also allows for multiple SGD passes, and therefore the potential for better sample efficiency compared to IMPALA. V-trace can also be enabled to correct for off-policy samples.

This implementation is currently *experimental*. Consider also using PPO or IMPALA.

Tuned examples: PongNoFrameskip-v4

APPO-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_base_config(impala.DEFAULT_CONFIG, {
    # Whether to use V-trace weighted advantages. If false, PPO GAE advantages
    # will be used instead.
    "vtrace": False,

    # == These two options only apply if vtrace: False ==
    # If true, use the Generalized Advantage Estimator (GAE)
    # with a value function, see https://arxiv.org/pdf/1506.02438.pdf.
    "use_gae": True,
    # GAE(lambda) parameter
    "lambda": 1.0,

    # == PPO surrogate loss options ==
    "clip_param": 0.4,

    # == IMPALA optimizer params (see documentation in impala.py) ==
    "sample_batch_size": 50,
    "train_batch_size": 500,
    "min_iter_time_s": 10,
    "num_workers": 2,
    "num_gpus": 1,
    "num_data_loader_buffers": 1,
    "minibatch_buffer_size": 1,
    "num_sgd_iter": 1,
    "replay_proportion": 0.0,
    "replay_buffer_num_slots": 100,
    "max_sample_requests_in_flight_per_worker": 2,
    "broadcast_interval": 1,
    "grad_clip": 40.0,
    "opt_type": "adam",
    "lr": 0.0005,
```

(continues on next page)

(continued from previous page)

```
    "lr_schedule": None,
    "decay": 0.99,
    "momentum": 0.0,
    "epsilon": 0.1,
    "vf_loss_coeff": 0.5,
    "entropy_coeff": -0.01,
})
```

1.22.2 Gradient-based

Advantage Actor-Critic (A2C, A3C)

[paper] [implementation] RLlib implements A2C and A3C using SyncSamplesOptimizer and AsyncGradientsOptimizer respectively for policy optimization. These algorithms scale to up to 16-32 worker processes depending on the environment. Both a TensorFlow (LSTM), and PyTorch version are available.

Tuned examples: PongDeterministic-v4, PyTorch version, {BeamRider,Breakout,Qbert,SpaceInvaders}NoFrameskip-v4

Tip: Consider using *IMPALA* for faster training with similar timestep efficiency.

Atari results @10M steps: more details

Atari env	RLlib A2C 5-workers	Mnih et al A3C 16-workers
BeamRider	1401	~3000
Breakout	374	~150
Qbert	3620	~1000
SpaceInvaders	692	~600

A3C-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # Size of rollout batch
    "sample_batch_size": 10,
    # Use PyTorch as backend - no LSTM support
    "use_pytorch": False,
    # GAE(gamma) parameter
    "lambda": 1.0,
    # Max global norm for each gradient calculated by worker
    "grad_clip": 40.0,
    # Learning rate
    "lr": 0.0001,
    # Learning rate schedule
    "lr_schedule": None,
    # Value Function Loss coefficient
    "vf_loss_coeff": 0.5,
    # Entropy coefficient
    "entropy_coeff": -0.01,
    # Min time per iteration
    "min_iter_time_s": 5,
    # Workers sample async. Note that this increases the effective
    # sample_batch_size by up to 5x due to async buffering of batches.
```

(continues on next page)

(continued from previous page)

```
    "sample_async": True,
})
```

Deep Deterministic Policy Gradients (DDPG, TD3)

[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. The improvements from TD3 are available though not enabled by default.

Tuned examples: Pendulum-v0, TD3 configuration, MountainCarContinuous-v0, HalfCheetah-v2

DDPG-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # === Twin Delayed DDPG (TD3) and Soft Actor-Critic (SAC) tricks ===
    # TD3: https://spinningup.openai.com/en/latest/algorithms/td3.html
    # twin Q-net
    "twin_q": False,
    # delayed policy update
    "policy_delay": 1,
    # target policy smoothing
    # this also forces the use of gaussian instead of OU noise for exploration
    "smooth_target_policy": False,
    # gaussian stddev of act noise
    "act_noise": 0.1,
    # gaussian stddev of target noise
    "target_noise": 0.2,
    # target noise limit (bound)
    "noise_clip": 0.5,

    # === Evaluation ===
    # Evaluate with epsilon=0 every `evaluation_interval` training iterations.
    # The evaluation stats will be reported under the "evaluation" metric key.
    # Note that evaluation is currently not parallelized, and that for Ape-X
    # metrics are already only reported for the lowest epsilon workers.
    "evaluation_interval": None,
    # Number of episodes to run per evaluation period.
    "evaluation_num_episodes": 10,

    # === Model ===
    # Hidden layer sizes of the policy network
    "actor_hiddens": [64, 64],
    # Hidden layers activation of the policy network
    "actor_hidden_activation": "relu",
    # Hidden layer sizes of the critic network
    "critic_hiddens": [64, 64],
    # Hidden layers activation of the critic network
    "critic_hidden_activation": "relu",
    # N-step Q learning
    "n_step": 1,

    # === Exploration ===
    # Max num timesteps for annealing schedules. Exploration is annealed from
    # 1.0 to exploration_fraction over this number of timesteps scaled by
    # exploration_fraction
    "schedule_max_timesteps": 100000,
```

(continues on next page)

(continued from previous page)

```
    # Number of env steps to optimize for before returning
    "timesteps_per_iteration": 1000,
    # Fraction of entire training period over which the exploration rate is
    # annealed
    "exploration_fraction": 0.1,
    # Final value of random action probability
    "exploration_final_eps": 0.02,
    # OU-noise scale
    "noise_scale": 0.1,
    # theta
    "exploration_theta": 0.15,
    # sigma
    "exploration_sigma": 0.2,
    # Update the target network every `target_network_update_freq` steps.
    "target_network_update_freq": 0,
    # Update the target by \tau * policy + (1-\tau) * target_policy
    "tau": 0.002,

    # === Replay buffer ===
    # Size of the replay buffer. Note that if async_updates is set, then
    # each worker will have a replay buffer of this size.
    "buffer_size": 50000,
    # If True prioritized replay buffer will be used.
    "prioritized_replay": True,
    # Alpha parameter for prioritized replay buffer.
    "prioritized_replay_alpha": 0.6,
    # Beta parameter for sampling from prioritized replay buffer.
    "prioritized_replay_beta": 0.4,
    # Epsilon to add to the TD errors when updating priorities.
    "prioritized_replay_eps": 1e-6,
    # Whether to LZ4 compress observations
    "compress_observations": False,

    # === Optimization ===
    # Learning rate for adam optimizer.
    # Instead of using two optimizers, we use two different loss coefficients
    "lr": 1e-3,
    "actor_loss_coeff": 0.1,
    "critic_loss_coeff": 1.0,
    # If True, use huber loss instead of squared loss for critic network
    # Conventionally, no need to clip gradients if using a huber loss
    "use_huber": False,
    # Threshold of a huber loss
    "huber_threshold": 1.0,
    # Weights for L2 regularization
    "l2_reg": 1e-6,
    # If not None, clip gradients during optimization at this value
    "grad_norm_clipping": None,
    # How many steps of the model to sample before learning starts.
    "learning_starts": 1500,
    # Update the replay buffer with this many samples at once. Note that this
    # setting applies per-worker if num_workers > 1.
    "sample_batch_size": 1,
    # Size of a batched sampled from replay buffer for training. Note that
    # if async_updates is set, then each worker returns gradients for a
    # batch of this size.
    "train_batch_size": 256,
```

(continues on next page)

(continued from previous page)

```
    # === Parallelism ===
    # Number of workers for collecting samples with. This only makes sense
    # to increase if your environment is particularly slow to sample, or if
    # you"re using the Async or Ape-X optimizers.
    "num_workers": 0,
    # Optimizer class to use.
    "optimizer_class": "SyncReplayOptimizer",
    # Whether to use a distribution of epsilons across workers for exploration.
    "per_worker_exploration": False,
    # Whether to compute priorities on workers.
    "worker_side_prioritization": False,
    # Prevent iterations from going lower than this time span
    "min_iter_time_s": 1,
})
```

Deep Q Networks (DQN, Rainbow, Parametric DQN)

[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. All of the DQN improvements evaluated in Rainbow are available, though not all are enabled by default. See also how to use parametric-actions in DQN.

Tuned examples: PongDeterministic-v4, Rainbow configuration, {BeamRider,Breakout,Qbert,SpaceInvaders}NoFrameskip-v4, with Dueling and Double-Q, with Distributional DQN.

Tip: Consider using *Ape-X* for faster training with similar timestep efficiency.

Atari results @10M steps: more details

Atari env	RLlib DQN	RLlib Dueling DDQN	RLlib Dist. DQN	Hessel et al. DQN
BeamRider	2869	1910	4447	~2000
Breakout	287	312	410	~150
Qbert	3921	7968	15780	~4000
SpaceInvaders	650	1001	1025	~500

DQN-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # === Model ===
    # Number of atoms for representing the distribution of return. When
    # this is greater than 1, distributional Q-learning is used.
    # the discrete supports are bounded by v_min and v_max
    "num_atoms": 1,
    "v_min": -10.0,
    "v_max": 10.0,
    # Whether to use noisy network
    "noisy": False,
    # control the initial value of noisy nets
    "sigma0": 0.5,
    # Whether to use dueling dqn
    "dueling": True,
```

(continues on next page)

(continued from previous page)

```
    # Whether to use double dqn
    "double_q": True,
    # Hidden layer sizes of the state and action value networks
    "hiddens": [256],
    # N-step Q learning
    "n_step": 1,

    # === Evaluation ===
    # Evaluate with epsilon=0 every `evaluation_interval` training iterations.
    # The evaluation stats will be reported under the "evaluation" metric key.
    # Note that evaluation is currently not parallelized, and that for Ape-X
    # metrics are already only reported for the lowest epsilon workers.
    "evaluation_interval": None,
    # Number of episodes to run per evaluation period.
    "evaluation_num_episodes": 10,

    # === Exploration ===
    # Max num timesteps for annealing schedules. Exploration is annealed from
    # 1.0 to exploration_fraction over this number of timesteps scaled by
    # exploration_fraction
    "schedule_max_timesteps": 100000,
    # Number of env steps to optimize for before returning
    "timesteps_per_iteration": 1000,
    # Fraction of entire training period over which the exploration rate is
    # annealed
    "exploration_fraction": 0.1,
    # Final value of random action probability
    "exploration_final_eps": 0.02,
    # Update the target network every `target_network_update_freq` steps.
    "target_network_update_freq": 500,

    # === Replay buffer ===
    # Size of the replay buffer. Note that if async_updates is set, then
    # each worker will have a replay buffer of this size.
    "buffer_size": 50000,
    # If True prioritized replay buffer will be used.
    "prioritized_replay": True,
    # Alpha parameter for prioritized replay buffer.
    "prioritized_replay_alpha": 0.6,
    # Beta parameter for sampling from prioritized replay buffer.
    "prioritized_replay_beta": 0.4,
    # Fraction of entire training period over which the beta parameter is
    # annealed
    "beta_annealing_fraction": 0.2,
    # Final value of beta
    "final_prioritized_replay_beta": 0.4,
    # Epsilon to add to the TD errors when updating priorities.
    "prioritized_replay_eps": 1e-6,
    # Whether to LZ4 compress observations
    "compress_observations": True,

    # === Optimization ===
    # Learning rate for adam optimizer
    "lr": 5e-4,
    # Adam epsilon hyper parameter
    "adam_epsilon": 1e-8,
    # If not None, clip gradients during optimization at this value
```

(continues on next page)

(continued from previous page)

```
    "grad_norm_clipping": 40,
    # How many steps of the model to sample before learning starts.
    "learning_starts": 1000,
    # Update the replay buffer with this many samples at once. Note that
    # this setting applies per-worker if num_workers > 1.
    "sample_batch_size": 4,
    # Size of a batched sampled from replay buffer for training. Note that
    # if async_updates is set, then each worker returns gradients for a
    # batch of this size.
    "train_batch_size": 32,

    # === Parallelism ===
    # Number of workers for collecting samples with. This only makes sense
    # to increase if your environment is particularly slow to sample, or if
    # you"re using the Async or Ape-X optimizers.
    "num_workers": 0,
    # Optimizer class to use.
    "optimizer_class": "SyncReplayOptimizer",
    # Whether to use a distribution of epsilons across workers for exploration.
    "per_worker_exploration": False,
    # Whether to compute priorities on workers.
    "worker_side_prioritization": False,
    # Prevent iterations from going lower than this time span
    "min_iter_time_s": 1,
})
```

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

PG-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # No remote workers by default
    "num_workers": 0,
    # Learning rate
    "lr": 0.0004,
})
```

Proximal Policy Optimization (PPO)

[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, HalfCheetah-v2, {BeamRider,Breakout,Qbert,SpaceInvaders}NoFrameskip-v4

Atari results: more details

Atari env	RLlib PPO @10M	RLlib PPO @25M	Baselines PPO @10M
BeamRider	2807	4480	~1800
Breakout	104	201	~250
Qbert	11085	14247	~14000
SpaceInvaders	671	944	~800

Scalability: more details

MuJoCo env	RLlib PPO 16-workers @ 1h	Fan et al PPO 16-workers @ 1h
HalfCheetah	9664	~7700

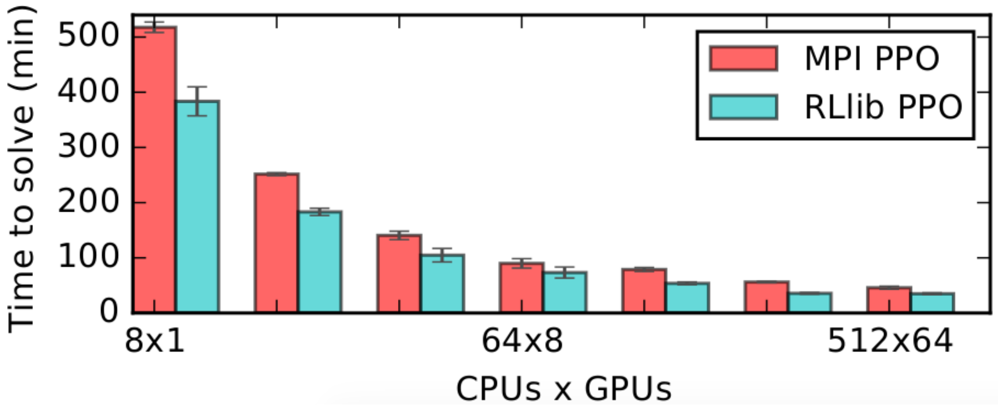


Fig. 3: RLlib's multi-GPU PPO scales to multiple GPUs and hundreds of CPUs on solving the Humanoid-v1 task. Here we compare against a reference MPI-based implementation.

PPO-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # If true, use the Generalized Advantage Estimator (GAE)
    # with a value function, see https://arxiv.org/pdf/1506.02438.pdf.
    "use_gae": True,
    # GAE(lambda) parameter
    "lambda": 1.0,
    # Initial coefficient for KL divergence
    "kl_coeff": 0.2,
    # Size of batches collected from each worker
    "sample_batch_size": 200,
    # Number of timesteps collected for each SGD round
    "train_batch_size": 4000,
    # Total SGD batch size across all devices for SGD
    "sgd_minibatch_size": 128,
    # Number of SGD iterations in each outer loop
    "num_sgd_iter": 30,
    # Stepsize of SGD
    "lr": 5e-5,
    # Learning rate schedule
    "lr_schedule": None,
    # Share layers for value function
    "vf_share_layers": False,
    # Coefficient of the value function loss
    "vf_loss_coeff": 1.0,
```

(continues on next page)

(continued from previous page)

```
    # Coefficient of the entropy regularizer
    "entropy_coeff": 0.0,
    # PPO clip parameter
    "clip_param": 0.3,
    # Clip param for the value function. Note that this is sensitive to the
    # scale of the rewards. If your expected V is large, increase this.
    "vf_clip_param": 10.0,
    # If specified, clip the global norm of gradients by this amount
    "grad_clip": None,
    # Target value for KL divergence
    "kl_target": 0.01,
    # Whether to rollout "complete_episodes" or "truncate_episodes"
    "batch_mode": "truncate_episodes",
    # Which observation filter to apply to the observation
    "observation_filter": "MeanStdFilter",
    # Uses the sync samples optimizer instead of the multi-gpu one. This does
    # not support minibatches.
    "simple_optimizer": False,
    # (Deprecated) Use the sampling behavior as of 0.6, which launches extra
    # sampling tasks for performance but can waste a large portion of samples.
    "straggler_mitigation": False,
})
```

1.22.3 Derivative-free

Augmented Random Search (ARS)

[paper] [implementation] ARS is a random search method for training linear policies for continuous control problems. Code here is adapted from https://github.com/modestyachts/ARS to integrate with RLlib APIs.

Tuned examples: CartPole-v0, Swimmer-v2

ARS-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    "noise_stdev": 0.02,  # std deviation of parameter noise
    "num_rollouts": 32,  # number of perturbs to try
    "rollouts_used": 32,  # number of perturbs to keep in gradient estimate
    "num_workers": 2,
    "sgd_stepsize": 0.01,  # sgd step-size
    "observation_filter": "MeanStdFilter",
    "noise_size": 250000000,
    "eval_prob": 0.03,  # probability of evaluating the parameter rewards
    "report_length": 10,  # how many of the last rewards we average over
    "offset": 0,
})
```

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

Scalability:

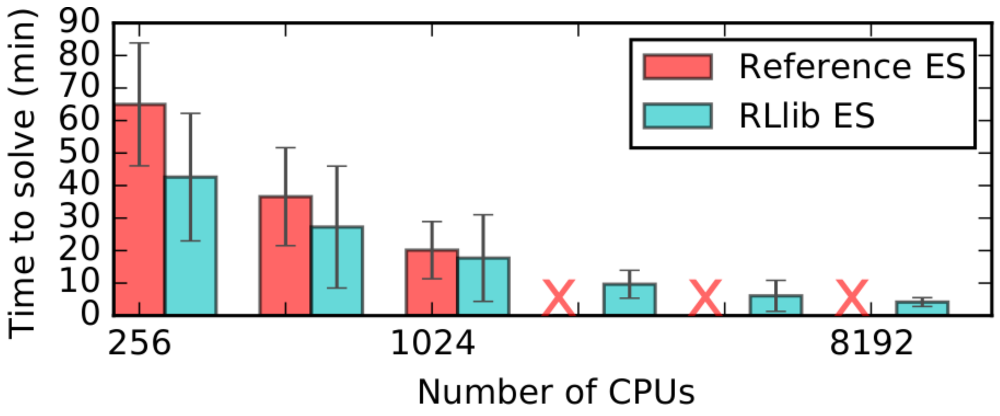


Fig. 4: RLlib's ES implementation scales further and is faster than a reference Redis implementation on solving the Humanoid-v1 task.

ES-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    "l2_coeff": 0.005,
    "noise_stdev": 0.02,
    "episodes_per_batch": 1000,
    "train_batch_size": 10000,
    "eval_prob": 0.003,
    "return_proc_mode": "centered_rank",
    "num_workers": 10,
    "stepsize": 0.01,
    "observation_filter": "MeanStdFilter",
    "noise_size": 250000000,
    "report_length": 10,
})
```

QMIX Monotonic Value Factorisation (QMIX, VDN, IQN)

[paper] [implementation] Q-Mix is a specialized multi-agent algorithm. Code here is adapted from https://github.com/oxwhirl/pymarl_alpha to integrate with RLlib multi-agent APIs. To use Q-Mix, you must specify an agent grouping in the environment (see the two-step game example). Currently, all agents in the group must be homogeneous. The algorithm can be scaled by increasing the number of workers or using Ape-X.

Q-Mix is implemented in PyTorch and is currently *experimental*.

Tuned examples: Two-step game

QMIX-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # === QMix ===
    # Mixing network. Either "qmix", "vdn", or None
    "mixer": "qmix",
    # Size of the mixing network embedding
    "mixing_embed_dim": 32,
    # Whether to use Double_Q learning
    "double_q": True,
    # Optimize over complete episodes by default.
    "batch_mode": "complete_episodes",
```

(continues on next page)

(continued from previous page)

```
    # === Evaluation ===
    # Evaluate with epsilon=0 every `evaluation_interval` training iterations.
    # The evaluation stats will be reported under the "evaluation" metric key.
    # Note that evaluation is currently not parallelized, and that for Ape-X
    # metrics are already only reported for the lowest epsilon workers.
    "evaluation_interval": None,
    # Number of episodes to run per evaluation period.
    "evaluation_num_episodes": 10,

    # === Exploration ===
    # Max num timesteps for annealing schedules. Exploration is annealed from
    # 1.0 to exploration_fraction over this number of timesteps scaled by
    # exploration_fraction
    "schedule_max_timesteps": 100000,
    # Number of env steps to optimize for before returning
    "timesteps_per_iteration": 1000,
    # Fraction of entire training period over which the exploration rate is
    # annealed
    "exploration_fraction": 0.1,
    # Final value of random action probability
    "exploration_final_eps": 0.02,
    # Update the target network every `target_network_update_freq` steps.
    "target_network_update_freq": 500,

    # === Replay buffer ===
    # Size of the replay buffer in steps.
    "buffer_size": 10000,

    # === Optimization ===
    # Learning rate for adam optimizer
    "lr": 0.0005,
    # RMSProp alpha
    "optim_alpha": 0.99,
    # RMSProp epsilon
    "optim_eps": 0.00001,
    # If not None, clip gradients during optimization at this value
    "grad_norm_clipping": 10,
    # How many steps of the model to sample before learning starts.
    "learning_starts": 1000,
    # Update the replay buffer with this many samples at once. Note that
    # this setting applies per-worker if num_workers > 1.
    "sample_batch_size": 4,
    # Size of a batched sampled from replay buffer for training. Note that
    # if async_updates is set, then each worker returns gradients for a
    # batch of this size.
    "train_batch_size": 32,

    # === Parallelism ===
    # Number of workers for collecting samples with. This only makes sense
    # to increase if your environment is particularly slow to sample, or if
    # you"re using the Async or Ape-X optimizers.
    "num_workers": 0,
    # Optimizer class to use.
    "optimizer_class": "SyncBatchReplayOptimizer",
    # Whether to use a distribution of epsilons across workers for exploration.
    "per_worker_exploration": False,
```

(continues on next page)

(continued from previous page)

```
    # Whether to compute priorities on workers.
    "worker_side_prioritization": False,
    # Prevent iterations from going lower than this time span
    "min_iter_time_s": 1,

    # === Model ===
    "model": {
        "lstm_cell_size": 64,
        "max_seq_len": 999999,
    },
})
```

Advantage Re-Weighted Imitation Learning (MARWIL)

[paper] [implementation] MARWIL is a hybrid imitation learning and policy gradient algorithm suitable for training on batched historical data. When the `beta` hyperparameter is set to zero, the MARWIL objective reduces to vanilla imitation learning. MARWIL requires the offline datasets API to be used.

Tuned examples: CartPole-v0

MARWIL-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # Scaling of advantages in exponential terms
    # When beta is 0, MARWIL is reduced to imitation learning
    "beta": 1.0,
    # Balancing value estimation loss and policy optimization loss
    "vf_coeff": 1.0,
    # Whether to calculate cumulative rewards
    "postprocess_inputs": True,
    # Whether to rollout "complete_episodes" or "truncate_episodes"
    "batch_mode": "complete_episodes",
    # Read data from historic data and evaluate by a sampler
    "input_evaluation": "simulation",
    # Learning rate for adam optimizer
    "lr": 1e-4,
    # Number of timesteps collected for each SGD round
    "train_batch_size": 2000,
    # Number of steps max to keep in the batch replay buffer
    "replay_buffer_size": 100000,
    # Number of steps to read before learning starts
    "learning_starts": 0,
    # === Parallelism ===
    "num_workers": 0,
})
```

1.23 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.

1.23.1 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 policy gradient algorithms (A3C, PPO, PG, IMPALA), 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 and Dict observations flattened (these are unflattened and accessible via the `input_dict` parameter in custom models). Note that for Atari, RLlib defaults to using the DeepMind preprocessors, which are also used by the OpenAI baselines library.

Built-in Model Parameters

The following is a list of the built-in model hyperparameters:

```
MODEL_DEFAULTS = {
    # === Built-in options ===
    # Filter config. List of [out_channels, kernel, stride] for each filter
    "conv_filters": None,
    # Nonlinearity for built-in convnet
    "conv_activation": "relu",
    # Nonlinearity for fully connected net (tanh, relu)
    "fcnet_activation": "tanh",
    # Number of hidden layers for fully connected net
    "fcnet_hiddens": [256, 256],
    # For control envs, documented in ray.rllib.models.Model
    "free_log_std": False,
    # (deprecated) Whether to use sigmoid to squash actions to space range
    "squash_to_range": False,

    # == LSTM ==
    # Whether to wrap the model with a LSTM
    "use_lstm": False,
    # Max seq len for training the LSTM, defaults to 20
    "max_seq_len": 20,
    # Size of the LSTM cell
    "lstm_cell_size": 256,
    # Whether to feed a_{t-1}, r_{t-1} to LSTM
    "lstm_use_prev_action_reward": False,

    # == Atari ==
    # Whether to enable framestack for Atari envs
    "framestack": True,
    # Final resized frame dimension
    "dim": 84,
```

(continues on next page)

(continued from previous page)

```
    # (deprecated) Converts ATARI frame to 1 Channel Grayscale image
    "grayscale": False,
    # (deprecated) Changes frame to range from [-1, 1] if true
    "zero_mean": True,

    # === Options for custom models ===
    # Name of a custom preprocessor to use
    "custom_preprocessor": None,
    # Name of a custom model to use
    "custom_model": None,
    # Extra options to pass to the custom classes
    "custom_options": {},
}
```

1.23.2 Custom Models (TensorFlow)

Custom TF models should subclass the common RLlib model class and override the `_build_layers_v2` method. This method takes in a dict of tensor inputs (the observation `obs`, `prev_action`, and `prev_reward`, `is_training`), and returns a feature layer and float vector of the specified output size. You can also override the `value_function` method to implement a custom value branch. A self-supervised loss can be defined via the `loss` method. The model can then be registered and used in place of a built-in model:

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

class MyModelClass(Model):
    def _build_layers_v2(self, input_dict, num_outputs, options):
        """Define the layers of a custom model.

        Arguments:
            input_dict (dict): Dictionary of input tensors, including "obs",
                "prev_action", "prev_reward", "is_training".
            num_outputs (int): Output tensor must be of size
                [BATCH_SIZE, num_outputs].
            options (dict): Model options.

        Returns:
            (outputs, feature_layer): Tensors of size [BATCH_SIZE, num_outputs]
                and [BATCH_SIZE, desired_feature_size].

        When using dict or tuple observation spaces, you can access
        the nested sub-observation batches here as well:

        Examples:
            >>> print(input_dict)
            {'prev_actions': <tf.Tensor shape=(?,) dtype=int64>,
             'prev_rewards': <tf.Tensor shape=(?,) dtype=float32>,
             'is_training': <tf.Tensor shape=(), dtype=bool>,
             'obs': OrderedDict([
                ('sensors', OrderedDict([
                    ('front_cam', [
                        <tf.Tensor shape=(?, 10, 10, 3) dtype=float32>,
                        <tf.Tensor shape=(?, 10, 10, 3) dtype=float32>]),
                    ('position', <tf.Tensor shape=(?, 3) dtype=float32>),
```

(continues on next page)

(continued from previous page)

```
                ('velocity', <tf.Tensor shape=(?, 3) dtype=float32>)]))])}
        """

        layer1 = slim.fully_connected(input_dict["obs"], 64, ...)
        layer2 = slim.fully_connected(layer1, 64, ...)
        ...
        return layerN, layerN_minus_1

    def value_function(self):
        """Builds the value function output.

        This method can be overridden to customize the implementation of the
        value function (e.g., not sharing hidden layers).

        Returns:
            Tensor of size [BATCH_SIZE] for the value function.
        """
        return tf.reshape(
            linear(self.last_layer, 1, "value", normc_initializer(1.0)), [-1])

    def loss(self):
        """Builds any built-in (self-supervised) loss for the model.

        For example, this can be used to incorporate auto-encoder style losses.
        Note that this loss has to be included in the policy graph loss to have
        an effect (done for built-in algorithms).

        Returns:
            Scalar tensor for the self-supervised loss.
        """
        return tf.constant(0.0)

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. You can also reference the unit tests for Tuple and Dict spaces, which show how to access nested observation fields.

Custom Recurrent Models

Instead of using the `use_lstm: True` option, it can be preferable use a custom recurrent model. This provides more control over postprocessing of the LSTM output and can also allow the use of multiple LSTM cells to process different portions of the input. The only difference from a normal custom model is that you have to define `self.state_init`, `self.state_in`, and `self.state_out`. You can refer to the existing lstm.py model as an example to implement your own model:

```
class MyCustomLSTM(Model):
    def _build_layers_v2(self, input_dict, num_outputs, options):
```

(continues on next page)

(continued from previous page)

```
        # Some initial layers to process inputs, shape [BATCH, OBS...].
        features = some_hidden_layers(input_dict["obs"])

        # Add back the nested time dimension for tf.dynamic_rnn, new shape
        # will be [BATCH, MAX_SEQ_LEN, OBS...].
        last_layer = add_time_dimension(features, self.seq_lens)

        # Setup the LSTM cell (see lstm.py for an example)
        lstm = rnn.BasicLSTMCell(256, state_is_tuple=True)
        self.state_init = ...
        self.state_in = ...
        lstm_out, lstm_state = tf.nn.dynamic_rnn(
            lstm,
            last_layer,
            initial_state=...,
            sequence_length=self.seq_lens,
            time_major=False,
            dtype=tf.float32)
        self.state_out = list(lstm_state)

        # Drop the time dimension again so back to shape [BATCH, OBS...].
        # Note that we retain the zero padding (see issue #2992).
        last_layer = tf.reshape(lstm_out, [-1, cell_size])
        logits = linear(last_layer, num_outputs, "action",
                        normc_initializer(0.01))
        return logits, last_layer
```

Batch Normalization

You can use `tf.layers.batch_normalization(x, training=input_dict["is_training"])` to add batch norm layers to your custom model: code example. RLlib will automatically run the update ops for the batch norm layers during optimization (see tf_policy_graph.py and multi_gpu_impl.py for the exact handling of these updates).

1.23.3 Custom Models (PyTorch)

Similarly, you can create and register custom PyTorch models for use with PyTorch-based algorithms (e.g., A2C, QMIX). See these examples of fully connected, convolutional, and recurrent torch models.

```
import ray
from ray.rllib.agents import a3c
from ray.rllib.models import ModelCatalog
from ray.rllib.models.pytorch.model import TorchModel

class CustomTorchModel(TorchModel):

    def __init__(self, obs_space, num_outputs, options):
        TorchModel.__init__(self, obs_space, num_outputs, options)
        ...  # setup hidden layers

    def _forward(self, input_dict, hidden_state):
        """Forward pass for the model.
```

(continues on next page)

(continued from previous page)

```
        Prefer implementing this instead of forward() directly for proper
        handling of Dict and Tuple observations.

        Arguments:
            input_dict (dict): Dictionary of tensor inputs, commonly
                including "obs", "prev_action", "prev_reward", each of shape
                [BATCH_SIZE, ...].
            hidden_state (list): List of hidden state tensors, each of shape
                [BATCH_SIZE, h_size].

        Returns:
            (outputs, feature_layer, values, state): Tensors of size
                [BATCH_SIZE, num_outputs], [BATCH_SIZE, desired_feature_size],
                [BATCH_SIZE], and [len(hidden_state), BATCH_SIZE, h_size].
        """
        obs = input_dict["obs"]
        ...
        return logits, features, value, hidden_state

ModelCatalog.register_custom_model("my_model", CustomTorchModel)

ray.init()
agent = a3c.A2CAgent(env="CartPole-v0", config={
    "use_pytorch": True,
    "model": {
        "custom_model": "my_model",
        "custom_options": {},  # extra options to pass to your model
    },
})
```

1.23.4 Custom Preprocessors

Custom preprocessors should subclass the RLlib preprocessor class and be registered in the model catalog. Note that you can alternatively use gym wrapper classes around your environment instead of preprocessors.

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

class MyPreprocessorClass(Preprocessor):
    def _init_shape(self, obs_space, options):
        return new_shape  # can vary depending on inputs

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

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
    },
})
```

1.23.5 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:

```
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:

```
from ray.rllib.agents.ddpg.ddpg import DDPGAgent
from custom_policy_graph import CustomDDPGPolicyGraph

DDPGAgent._policy_graph = CustomDDPGPolicyGraph
agent = DDPGAgent(...)
```

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.

1.23.6 Variable-length / Parametric Action Spaces

Custom models can be used to work with environments where (1) the set of valid actions varies per step, and/or (2) the number of valid actions is very large, as in OpenAI Five and Horizon. The general idea is that the meaning of actions can be completely conditioned on the observation, i.e., the `a` in `Q(s, a)` becomes just a token in `[0, MAX_AVAIL_ACTIONS)` that only has meaning in the context of `s`. This works with algorithms in the DQN and policy-gradient families and can be implemented as follows:

1. The environment should return a mask and/or list of valid action embeddings as part of the observation for each step. To enable batching, the number of actions can be allowed to vary from 1 to some max number:

```
class MyParamActionEnv(gym.Env):
    def __init__(self, max_avail_actions):
        self.action_space = Discrete(max_avail_actions)
        self.observation_space = Dict({
            "action_mask": Box(0, 1, shape=(max_avail_actions, )),
            "avail_actions": Box(-1, 1, shape=(max_avail_actions, action_embedding_
↪sz)),
            "real_obs": ...,
        })
```

2. A custom model can be defined that can interpret the `action_mask` and `avail_actions` portions of the observation. Here the model computes the action logits via the dot product of some network output and each action embedding. Invalid actions can be masked out of the softmax by scaling the probability to zero:

```
class MyParamActionModel(Model):
    def _build_layers_v2(self, input_dict, num_outputs, options):
        avail_actions = input_dict["obs"]["avail_actions"]
        action_mask = input_dict["obs"]["action_mask"]

        output = FullyConnectedNetwork(
            input_dict["obs"]["real_obs"], num_outputs=action_embedding_sz)

        # Expand the model output to [BATCH, 1, EMBED_SIZE]. Note that the
        # avail actions tensor is of shape [BATCH, MAX_ACTIONS, EMBED_SIZE].
        intent_vector = tf.expand_dims(output, 1)

        # Shape of logits is [BATCH, MAX_ACTIONS].
        action_logits = tf.reduce_sum(avail_actions * intent_vector, axis=2)

        # Mask out invalid actions (use tf.float32.min for stability)
        inf_mask = tf.maximum(tf.log(action_mask), tf.float32.min)
        masked_logits = inf_mask + action_logits

        return masked_logits, last_layer
```

Depending on your use case it may make sense to use just the masking, just action embeddings, or both. For a runnable example of this in code, check out parametric_action_cartpole.py. Note that since masking introduces `tf.float32.min` values into the model output, this technique might not work with all algorithm options. For example, algorithms might crash if they incorrectly process the `tf.float32.min` values. The cartpole example has working configurations for DQN (must set `hiddens=[]`), PPO (must disable running mean and set `vf_share_layers=True`), and several other algorithms.

1.23.7 Model-Based Rollouts

With a custom policy graph, you can also perform model-based rollouts and optionally incorporate the results of those rollouts as training data. For example, suppose you wanted to extend PGPolicyGraph for model-based rollouts. This involves overriding the `compute_actions` method of that policy graph:

```
class ModelBasedPolicyGraph(PGPolicyGraph):
     def compute_actions(self,
                         obs_batch,
                         state_batches,
                         prev_action_batch=None,
```

(continues on next page)

(continued from previous page)

```
                        prev_reward_batch=None,
                        episodes=None):
        # compute a batch of actions based on the current obs_batch
        # and state of each episode (i.e., for multiagent). You can do
        # whatever is needed here, e.g., MCTS rollouts.
        return action_batch
```

If you want take this rollouts data and append it to the sample batch, use the `add_extra_batch()` method of the episode objects passed in. For an example of this, see the `testReturningModelBasedRolloutsData` unit test.

1.24 RLlib Offline Datasets

1.24.1 Working with Offline Datasets

RLlib's offline dataset APIs enable working with experiences read from offline storage (e.g., disk, cloud storage, streaming systems, HDFS). For example, you might want to read experiences saved from previous training runs, or gathered from policies deployed in web applications. You can also log new agent experiences produced during online training for future use.

RLlib represents trajectory sequences (i.e., `(s, a, r, s', ...)` tuples) with SampleBatch objects. Using a batch format enables efficient encoding and compression of experiences. During online training, RLlib uses policy evaluation actors to generate batches of experiences in parallel using the current policy. RLlib also uses this same batch format for reading and writing experiences to offline storage.

Example: Training on previously saved experiences

Note: For custom models and enviroments, you'll need to use the Python API.

In this example, we will save batches of experiences generated during online training to disk, and then leverage this saved data to train a policy offline using DQN. First, we run a simple policy gradient algorithm for 100k steps with `"output": "/tmp/cartpole-out"` to tell RLlib to write simulation outputs to the `/tmp/cartpole-out` directory.

```
$ rllib train
    --run=PG \
    --env=CartPole-v0 \
    --config='{"output": "/tmp/cartpole-out", "output_max_file_size": 5000000}' \
    --stop='{"timesteps_total": 100000}'
```

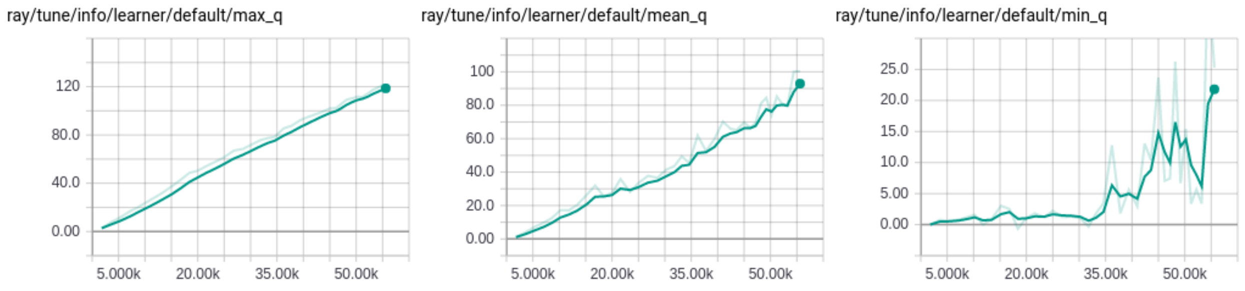
The experiences will be saved in compressed JSON batch format:

```
$ ls -l /tmp/cartpole-out
total 11636
-rw-rw-r-- 1 eric eric 5022257 output-2019-01-01_15-58-57_worker-0_0.json
-rw-rw-r-- 1 eric eric 5002416 output-2019-01-01_15-59-22_worker-0_1.json
-rw-rw-r-- 1 eric eric 1881666 output-2019-01-01_15-59-47_worker-0_2.json
```

Then, we can tell DQN to train using these previously generated experiences with `"input": "/tmp/cartpole-out"`. We disable exploration since it has no effect on the input:

```
$ rllib train \
    --run=DQN \
    --env=CartPole-v0 \
    --config='{
        "input": "/tmp/cartpole-out",
        "exploration_final_eps": 0,
        "exploration_fraction": 0}'
```

Since the input experiences are not from running simulations, RLlib cannot report the true policy performance during training. However, you can use `tensorboard --logdir=~/ray_results` to monitor training progress via other metrics such as estimated Q-value:



In offline input mode, no simulations are run, though you still need to specify the environment in order to define the action and observation spaces. If true simulation is also possible (i.e., your env supports `step()`), you can also set `"input_evaluation": "simulation"` to tell RLlib to run background simulations to estimate current policy performance. The output of these simulations will not be used for learning.

Example: Converting external experiences to batch format

When the env does not support simulation (e.g., it is a web application), it is necessary to generate the `*.json` experience batch files outside of RLlib. This can be done by using the JsonWriter class to write out batches. This runnable example shows how to generate and save experience batches for CartPole-v0 to disk:

```
import gym
import numpy as np

from ray.rllib.evaluation.sample_batch_builder import SampleBatchBuilder
from ray.rllib.offline.json_writer import JsonWriter

if __name__ == "__main__":
    batch_builder = SampleBatchBuilder()  # or MultiAgentSampleBatchBuilder
    writer = JsonWriter("/tmp/demo-out")

    # You normally wouldn't want to manually create sample batches if a
    # simulator is available, but let's do it anyways for example purposes:
    env = gym.make("CartPole-v0")

    for eps_id in range(100):
        obs = env.reset()
        prev_action = np.zeros_like(env.action_space.sample())
        prev_reward = 0
        done = False
        t = 0
        while not done:
            action = env.action_space.sample()
```

(continues on next page)

(continued from previous page)

```
            new_obs, rew, done, info = env.step(action)
            batch_builder.add_values(
                t=t,
                eps_id=eps_id,
                agent_index=0,
                obs=obs,
                actions=action,
                rewards=rew,
                prev_actions=prev_action,
                prev_rewards=prev_reward,
                dones=done,
                infos=info,
                new_obs=new_obs)
            obs = new_obs
            prev_action = action
            prev_reward = rew
            t += 1
        writer.write(batch_builder.build_and_reset())
```

On-policy algorithms and experience postprocessing

RLlib assumes that input batches are of postprocessed experiences. This isn't typically critical for off-policy algorithms (e.g., DQN's post-processing is only needed if `n_step > 1` or `worker_side_prioritization: True`). For off-policy algorithms, you can also safely set the `postprocess_inputs: True` config to auto-postprocess data.

However, for on-policy algorithms like PPO, you'll need to pass in the extra values added during policy evaluation and postprocessing to `batch_builder.add_values()`, e.g., `logits`, `vf_preds`, `value_target`, and `advantages` for PPO. This is needed since the calculation of these values depends on the parameters of the *behaviour* policy, which RLlib does not have access to in the offline setting (in online training, these values are automatically added during policy evaluation).

Note that for on-policy algorithms, you'll also have to throw away experiences generated by prior versions of the policy. This greatly reduces sample efficiency, which is typically undesirable for offline training, but can make sense for certain applications.

Mixing simulation and offline data

RLlib supports multiplexing inputs from multiple input sources, including simulation. For example, in the following example we read 40% of our experiences from `/tmp/cartpole-out`, 30% from `hdfs:/archive/cartpole`, and the last 30% is produced via policy evaluation. Input sources are multiplexed using np.random.choice:

```
$ rllib train \
    --run=DQN \
    --env=CartPole-v0 \
    --config='{
        "input": {
            "/tmp/cartpole-out": 0.4,
            "hdfs:/archive/cartpole": 0.3,
            "sampler": 0.3,
        },
        "exploration_final_eps": 0,
        "exploration_fraction": 0}'
```

Scaling I/O throughput

Similar to scaling online training, you can scale offline I/O throughput by increasing the number of RLlib workers via the `num_workers` config. Each worker accesses offline storage independently in parallel, for linear scaling of I/O throughput. Within each read worker, files are chosen in random order for reads, but file contents are read sequentially.

1.24.2 Input API

You can configure experience input for an agent using the following options:

```
    # Specify how to generate experiences:
    #  - "sampler": generate experiences via online simulation (default)
    #  - a local directory or file glob expression (e.g., "/tmp/*.json")
    #  - a list of individual file paths/URIs (e.g., ["/tmp/1.json",
    #    "s3://bucket/2.json"])
    #  - a dict with string keys and sampling probabilities as values (e.g.,
    #    {"sampler": 0.4, "/tmp/*.json": 0.4, "s3://bucket/expert.json": 0.2}).
    #  - a function that returns a rllib.offline.InputReader
    "input": "sampler",
    # Specify how to evaluate the current policy. This only makes sense to set
    # when the input is not already generating simulation data:
    #  - None: don't evaluate the policy. The episode reward and other
    #    metrics will be NaN if using offline data.
    #  - "simulation": run the environment in the background, but use
    #    this data for evaluation only and not for learning.
    "input_evaluation": None,
    # Whether to run postprocess_trajectory() on the trajectory fragments from
    # offline inputs. Note that postprocessing will be done using the *current*
    # policy, not the *behaviour* policy, which is typically undesirable for
    # on-policy algorithms.
    "postprocess_inputs": False,
```

The interface for a custom input reader is as follows:

class ray.rllib.offline.**InputReader**

Input object for loading experiences in policy evaluation.

next()

Return the next batch of experiences read.

Returns SampleBatch or MultiAgentBatch read.

1.24.3 Output API

You can configure experience output for an agent using the following options:

```
    # Specify where experiences should be saved:
    #  - None: don't save any experiences
    #  - "logdir" to save to the agent log dir
    #  - a path/URI to save to a custom output directory (e.g., "s3://bucket/")
    #  - a function that returns a rllib.offline.OutputWriter
    "output": None,
    # What sample batch columns to LZ4 compress in the output data.
    "output_compress_columns": ["obs", "new_obs"],
    # Max output file size before rolling over to a new file.
    "output_max_file_size": 64 * 1024 * 1024,
```

The interface for a custom output writer is as follows:

class `ray.rllib.offline.`**`OutputWriter`**

Writer object for saving experiences from policy evaluation.

`write`(*sample_batch*)

Save a batch of experiences.

Parameters **`sample_batch`** – SampleBatch or MultiAgentBatch to save.

1.25 RLlib Development

1.25.1 Development Install

You can develop RLlib locally without needing to compile Ray by using the setup-rllib-dev.py script. This sets up links between the `rllib` dir in your git repo and the one bundled with the `ray` package. When using this script, make sure that your git branch is in sync with the installed Ray binaries (i.e., you are up-to-date on master and have the latest wheel installed.)

1.25.2 API Stability

Objects and methods annotated with `@PublicAPI` or `@DeveloperAPI` have the following API compatibility guarantees:

`ray.rllib.utils.annotations.`**`PublicAPI`**(*obj*)

Annotation for documenting public APIs.

Public APIs are classes and methods exposed to end users of RLlib. You can expect these APIs to remain stable across RLlib releases.

Subclasses that inherit from a `@PublicAPI` base class can be assumed part of the RLlib public API as well (e.g., all agent classes are in public API because Agent is `@PublicAPI`).

In addition, you can assume all agent configurations are part of their public API as well.

`ray.rllib.utils.annotations.`**`DeveloperAPI`**(*obj*)

Annotation for documenting developer APIs.

Developer APIs are classes and methods explicitly exposed to developers for the purposes of building custom algorithms or advanced training strategies on top of RLlib internals. You can generally expect these APIs to be stable sans minor changes (but less stable than public APIs).

Subclasses that inherit from a `@DeveloperAPI` base class can be assumed part of the RLlib developer API as well (e.g., all policy optimizers are developer API because PolicyOptimizer is `@DeveloperAPI`).

1.25.3 Features

Feature development and upcoming priorities are tracked on the RLlib project board (note that this may not include all development efforts). For discussion of issues and new features, we use the Ray dev list and GitHub issues page.

1.25.4 Benchmarks

A number of training run results are available in the rl-experiments repo, and there is also a list of working hyper-parameter configurations in tuned_examples. Benchmark results are extremely valuable to the community, so if you happen to have results that may be of interest, consider making a pull request to either repo.

1.25.5 Contributing Algorithms

These are the guidelines for merging new algorithms into RLlib:

- **Contributed algorithms (rllib/contrib):**
  - must subclass Agent and implement the `_train()` method
  - must include a lightweight test (example) to ensure the algorithm runs
  - should include tuned hyperparameter examples and documentation
  - should offer functionality not present in existing algorithms
- **Fully integrated algorithms (rllib/agents) have the following additional requirements:**
  - must fully implement the Agent API
  - must offer substantial new functionality not possible to add to other algorithms
  - should support custom models and preprocessors
  - should use RLlib abstractions and support distributed execution

Both integrated and contributed algorithms ship with the `ray` PyPI package, and are tested as part of Ray's automated tests. The main difference between contributed and fully integrated algorithms is that the latter will be maintained by the Ray team to a much greater extent with respect to bugs and integration with RLlib features.

How to add an algorithm to `contrib`

It takes just two changes to add an algorithm to contrib. A minimal example can be found here. First, subclass Agent and implement the `_init` and `_train` methods:

```
class RandomAgent(Agent):
    """Agent that takes random actions and never learns."""

    _agent_name = "RandomAgent"
    _default_config = with_common_config({
        "rollouts_per_iteration": 10,
    })

    @override(Agent)
    def _init(self):
        self.env = self.env_creator(self.config["env_config"])

    @override(Agent)
    def _train(self):
        rewards = []
        steps = 0
        for _ in range(self.config["rollouts_per_iteration"]):
            obs = self.env.reset()
            done = False
            reward = 0.0
            while not done:
                action = self.env.action_space.sample()
                obs, r, done, info = self.env.step(action)
                reward += r
                steps += 1
            rewards.append(reward)
        return {
```

(continues on next page)

(continued from previous page)

```
            "episode_reward_mean": np.mean(rewards),
            "timesteps_this_iter": steps,
        }
```

Second, register the agent with a name in contrib/registry.py.

```
def _import_random_agent():
    from ray.rllib.contrib.random_agent.random_agent import RandomAgent
    return RandomAgent

def _import_random_agent_2():
    from ray.rllib.contrib.random_agent_2.random_agent_2 import RandomAgent2
    return RandomAgent2

CONTRIBUTED_ALGORITHMS = {
    "contrib/RandomAgent": _import_random_agent,
    "contrib/RandomAgent2": _import_random_agent_2,
    # ...
}
```

After registration, you can run and visualize agent progress using `rllib train`:

```
rllib train --run=contrib/RandomAgent --env=CartPole-v0
tensorboard --logdir=~/ray_results
```

1.26 RLlib Concepts

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

1.26.1 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.

1.26.2 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 `PolicyEvaluator.as_remote()`).

1.26.3 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.

1.27 RLlib Package Reference

1.27.1 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

Function that creates a new training env.

Type func

config

Algorithm-specific configuration data.

Type obj

logdir

Directory in which training outputs should be placed.

Type str

classmethod default_resource_request(*config*)

Returns the resource requirement for the given configuration.

This can be overriden by sub-classes to set the correct trial resource allocation, so the user does not need to.

train()

Overrides super.train to synchronize global vars.

compute_action(*observation*, *state=None*, *prev_action=None*, *prev_reward=None*, *info=None*, *policy_id='default'*)

Computes an action for the specified policy.

Note that you can also access the policy object through self.get_policy(policy_id) and call compute_actions() on it directly.

Parameters

- **observation** (*obj*) – observation from the environment.
- **state** (*list*) – RNN hidden state, if any. If state is not None, then all of compute_single_action(. . .) is returned (computed action, rnn state, logits dictionary). Otherwise compute_single_action(. . .)[0] is returned (computed action).
- **prev_action** (*obj*) – previous action value, if any

- **prev_reward** (*int*) – previous reward, if any
- **info** (*dict*) – info object, if any
- **policy_id** (*str*) – policy to query (only applies to multi-agent).

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

get_policy (*policy_id='default'*)
Return policy graph for the specified id, or None.

Parameters **policy_id** (*str*) – id of policy graph to return.

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.

make_local_evaluator (*env_creator*, *policy_graph*, *extra_config=None*)
Convenience method to return configured local evaluator.

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

export_policy_model (*export_dir*, *policy_id='default'*)
Export policy model with given policy_id to local directory.

Parameters

- **export_dir** (*string*) – Writable local directory.
- **policy_id** (*string*) – Optional policy id to export.

Example

```
>>> agent = MyAgent()
>>> for _ in range(10):
>>>     agent.train()
>>> agent.export_policy_model("/tmp/export_dir")
```

export_policy_checkpoint (*export_dir*, *filename_prefix='model'*, *policy_id='default'*)
Export tensorflow policy model checkpoint to local directory.

Parameters

- **export_dir** (*string*) – Writable local directory.
- **filename_prefix** (*string*) – file name prefix of checkpoint files.
- **policy_id** (*string*) – Optional policy id to export.

Example

```
>>> agent = MyAgent()
>>> for _ in range(10):
>>>     agent.train()
>>> agent.export_policy_checkpoint("/tmp/export_dir")
```

classmethod resource_help(*config*)
Returns a help string for configuring this trainable's resources.

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

class `ray.rllib.agents.a3c.`**A2CAgent**(*config=None*, *env=None*, *logger_creator=None*)
Synchronous variant of the A3CAgent.

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.

1.27.2 ray.rllib.env

class `ray.rllib.env.`**BaseEnv**
The lowest-level env interface used by RLlib for sampling.

BaseEnv 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 BaseEnv. RLlib handles these conversions internally in PolicyEvaluator, for example:

> gym.Env => rllib.VectorEnv => rllib.BaseEnv rllib.MultiAgentEnv => rllib.BaseEnv rllib.ExternalEnv => rllib.BaseEnv

action_space
Action space. This must be defined for single-agent envs. Multi-agent envs can set this to None.

Type gym.Space

observation_space
Observation space. This must be defined for single-agent envs. Multi-agent envs can set this to None.

Type gym.Space

Examples

```
>>> env = MyBaseEnv()
>>> 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],
        "car_1": [3.2, -4.2],
    }
}
>>> print(dones)
{
    "env_0": {
        "__all__": False,
        "car_0": False,
        "car_1": True,
    }
}
```

static to_base_env(*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 the underlying gym envs, if any.

Returns Underlying gym envs or [].

Return type envs (list)

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

```
>>> 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,
```

(continues on next page)

(continued from previous page)

```
    "car_1": -1,
    "traffic_light_1": 0,
}
>>> print(dones)
{
    "car_0": False,    # car_0 is still running
    "car_1": True,     # car_1 is done
    "__all__": False,  # the env is not done
}
>>> print(infos)
{
    "car_0": {},  # info for car_0
    "car_1": {},  # info for car_1
}
```

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__" (required) is used to indicate env termination.
- **infos (dict)** (*Optional info values for each agent id.*)

with_agent_groups(*groups*, *obs_space=None*, *act_space=None*)

Convenience method for grouping together agents in this env.

An agent group is a list of agent ids that are mapped to a single logical agent. All agents of the group must act at the same time in the environment. The grouped agent exposes Tuple action and observation spaces that are the concatenated action and obs spaces of the individual agents.

The rewards of all the agents in a group are summed. The individual agent rewards are available under the "individual_rewards" key of the group info return.

Agent grouping is required to leverage algorithms such as Q-Mix.

This API is experimental.

Parameters

- **groups** (*dict*) – Mapping from group id to a list of the agent ids of group members. If an agent id is not present in any group value, it will be left ungrouped.
- **obs_space** (*Space*) – Optional observation space for the grouped env. Must be a tuple space.
- **act_space** (*Space*) – Optional action space for the grouped env. Must be a tuple space.

Examples

```
>>> env = YourMultiAgentEnv(...)
>>> grouped_env = env.with_agent_groups(env, {
...   "group1": ["agent1", "agent2", "agent3"],
...   "group2": ["agent4", "agent5"],
... })
```

class ray.rllib.env.**ExternalEnv**(*action_space*, *observation_space*, *max_concurrent=100*)

An environment that interfaces with external agents.

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 ExternalEnv 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.

ExternalEnv supports both on-policy actions (through self.get_action()), and off-policy actions (through self.log_action()).

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

action_space

Action space.

Type gym.Space

observation_space

Observation space.

Type gym.Space

Examples

```
>>> register_env("my_env", lambda config: YourExternalEnv(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(episode_id)
2. **Call self.get_action(episode_id, obs)** -or- self.log_action(episode_id, obs, action)
3. Call self.log_returns(episode_id, reward)
4. Call self.end_episode(episode_id, obs)
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
Action space of individual envs.

Type gym.Space

observation_space
Observation space of individual envs.

Type gym.Space

`num_envs`
Number of envs in this vector env.

Type int

`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 the underlying env instances.

`ray.rllib.env.`**`ServingEnv`**
alias of `ray.rllib.env.external_env.ExternalEnv`

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

Type int

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

Type int

1.27.3 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

```
>>> 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

```
>>> 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

```
>>> 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

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

set_weights(*weights*)

Sets the model weights of this Evaluator.

This method must be implemented by subclasses.

Examples

```
>>> 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

```
>>> 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='deepmind'*, *sample_async=False*, *compress_observations=False*, *num_envs=1*, *observation_filter='NoFilter'*, *clip_rewards=None*, *clip_actions=True*, *env_config=None*, *model_config=None*, *policy_config=None*, *worker_index=0*, *monitor_path=None*, *log_dir=None*, *log_level=None*, *callbacks=None*, *input_creator=<function PolicyEvaluator.<lambda>>*, *input_evaluation_method=None*, *output_creator=<function PolicyEvaluator.<lambda>>*)

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={
...       # Use an ensemble of two policies for car agents
...       "car_policy1":
...         (PGPolicyGraph, Box(...), Discrete(...), {"gamma": 0.99}),
...       "car_policy2":
...         (PGPolicyGraph, Box(...), Discrete(...), {"gamma": 0.95}),
...       # Use a single shared policy for all traffic lights
...       "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")
>>> print(evaluator.sample())
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.

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

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

set_weights(*weights*)

Sets the model weights of this Evaluator.

This method must be implemented by subclasses.

Examples

```
>>> 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

```
>>> 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

```
>>> 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

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

get_policy (*policy_id='default'*)
Return policy graph for the specified id, or None.

Parameters **policy_id** (*str*) – id of policy graph to return.

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)

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
Observation space of the policy.

Type gym.Space

action_space
Action space of the policy.

Type gym.Space

compute_actions (*obs_batch*, *state_batches*, *prev_action_batch=None*, *prev_reward_batch=None*, *info_batch=None*, *episodes=None*, ***kwargs*)
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
- **prev_action_batch** (*np.ndarray*) – batch of previous action values
- **prev_reward_batch** (*np.ndarray*) – batch of previous rewards
- **info_batch** (*info*) – batch of info objects

- **`episodes`** (*`list`*) – MultiAgentEpisode for each obs in obs_batch. This provides access to all of the internal episode state, which may be useful for model-based or multiagent algorithms.
- **`kwargs`** – forward compatibility placeholder

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*, *prev_action=None*, *prev_reward=None*, *info=None*, *episode=None*, ***kwargs*)

Unbatched version of compute_actions.

Parameters

- **`obs`** (*`obj`*) – single observation
- **`state_batches`** (*`list`*) – list of RNN state inputs, if any
- **`prev_action`** (*`obj`*) – previous action value, if any
- **`prev_reward`** (*`int`*) – previous reward, if any
- **`info`** (*`dict`*) – info object, if any
- **`episode`** (`MultiAgentEpisode`) – this provides access to all of the internal episode state, which may be useful for model-based or multi-agent algorithms.
- **`kwargs`** – forward compatibility placeholder

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*, *episode=None*)

Implements algorithm-specific trajectory postprocessing.

This will be called on each trajectory fragment computed during policy evaluation. Each fragment is guaranteed to be only from one episode.

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.
- **`episode`** (`MultiAgentEpisode`) – this provides access to all of the internal episode state, which may be useful for model-based or multi-agent algorithms.

Returns postprocessed sample batch.

Return type *SampleBatch*

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)

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

```
>>> 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.

on_global_var_update(*global_vars*)
Called on an update to global vars.

Parameters **global_vars** (`dict`) – Global variables broadcast from the driver.

export_model(*export_dir*)
Export PolicyGraph to local directory for serving.

Parameters **export_dir** (`str`) – Local writable directory.

export_checkpoint(*export_dir*)
Export PolicyGraph checkpoint to local directory.

Argument: export_dir (str): Local writable directory.

class ray.rllib.evaluation.**TFPolicyGraph**(*observation_space*, *action_space*, *sess*, *obs_input*, *action_sampler*, *loss*, *loss_inputs*, *state_inputs=None*, *state_outputs=None*, *prev_action_input=None*, *prev_reward_input=None*, *seq_lens=None*, *max_seq_len=20*, *batch_divisibility_req=1*, *update_ops=None*)
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
observation space of the policy.

Type gym.Space

action_space
action space of the policy.

Type gym.Space

Examples

```
>>> policy = TFPolicyGraphSubclass(
    sess, obs_input, action_sampler, loss, loss_inputs)
```

```
>>> print(policy.compute_actions([1, 0, 2]))
(array([0, 1, 1]), [], {})
```

```
>>> print(policy.postprocess_trajectory(SampleBatch({...})))
SampleBatch({"action": ..., "advantages": ..., ...})
```

compute_actions(*obs_batch*, *state_batches=None*, *prev_action_batch=None*, *prev_reward_batch=None*, *info_batch=None*, *episodes=None*, ***kwargs*)
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
- **prev_action_batch** (*np.ndarray*) – batch of previous action values
- **prev_reward_batch** (*np.ndarray*) – batch of previous rewards
- **info_batch** (*info*) – batch of info objects
- **episodes** (*list*) – MultiAgentEpisode for each obs in obs_batch. This provides access to all of the internal episode state, which may be useful for model-based or multiagent algorithms.
- **kwargs** – forward compatibility placeholder

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)

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

Examples

```
>>> 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

export_model(*export_dir*)
Export tensorflow graph to export_dir for serving.

export_checkpoint(*export_dir*, *filename_prefix='model'*)
Export tensorflow checkpoint to export_dir.

copy(*existing_inputs*)
Creates a copy of self using existing input placeholders.

Optional, only required to work with the multi-GPU optimizer.

extra_compute_action_feed_dict()
Extra dict to pass to the compute actions session run.

extra_compute_action_fetches()
Extra values to fetch and return from compute_actions().

extra_compute_grad_feed_dict()
Extra dict to pass to the compute gradients session run.

extra_compute_grad_fetches()
Extra values to fetch and return from compute_gradients().

extra_apply_grad_feed_dict()
Extra dict to pass to the apply gradients session run.

extra_apply_grad_fetches()
Extra values to fetch and return from apply_gradients().

optimizer()
TF optimizer to use for policy optimization.

gradients(*optimizer*)
Override for custom gradient computation.

class ray.rllib.evaluation.**TorchPolicyGraph**(*observation_space*, *action_space*, *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
observation space of the policy.

Type gym.Space

action_space
action space of the policy.

Type gym.Space

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

Type Lock

compute_actions(*obs_batch*, *state_batches=None*, *prev_action_batch=None*, *prev_reward_batch=None*, *info_batch=None*, *episodes=None*, ***kwargs*)
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
- **prev_action_batch** (*np.ndarray*) – batch of previous action values
- **prev_reward_batch** (*np.ndarray*) – batch of previous rewards
- **info_batch** (*info*) – batch of info objects
- **episodes** (*list*) – MultiAgentEpisode for each obs in obs_batch. This provides access to all of the internal episode state, which may be useful for model-based or multiagent algorithms.
- **kwargs** – forward compatibility placeholder

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

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

`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.

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

```
>>> 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

```
>>> 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

```
>>> batch = SampleBatch({"a": [1], "b": [2], "c": [3]})
>>> print(batch.columns(["a", "b"]))
[[1], [2]]
```

shuffle()
Shuffles the rows of this batch in-place.

split_by_episode()
Splits this batch's data by *eps_id*.

Returns list of SampleBatch, one per distinct episode.

slice(*start*, *end*)
Returns a slice of the row data of this batch.

Parameters

- **start** (*int*) – Starting index.
- **end** (*int*) – Ending index.

Returns SampleBatch which has a slice of this batch's data.

class ray.rllib.evaluation.**MultiAgentBatch**(*policy_batches*, *count*)
A batch of experiences from multiple policies in the environment.

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

Type dict

count
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.

Type int

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*, *clip_rewards*)
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.

total()
Returns summed number of steps across all agent buffers.

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(*episode*)
Apply policy postprocessors to any unprocessed rows.

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

Parameters **episode** – current MultiAgentEpisode object or None

build_and_reset(*episode*)
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.

Parameters **episode** – current MultiAgentEpisode object or None

class ray.rllib.evaluation.**AsyncSampler**(*env*, *policies*, *policy_mapping_fn*, *preprocessors*, *obs_filters*, *clip_rewards*, *unroll_length*, *callbacks*, *horizon=None*, *pack=False*, *tf_sess=None*, *clip_actions=True*, *blackhole_outputs=False*)

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.

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 Estamation

Returns

Object with experience from rollout and processed rewards.

Return type *SampleBatch* (*SampleBatch*)

`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

`ray.rllib.evaluation.`**`collect_metrics`**(*local_evaluator, remote_evaluators=[], timeout_seconds=180*)

Gathers episode metrics from PolicyEvaluator instances.

`class` `ray.rllib.evaluation.`**`MultiAgentEpisode`**(*policies, policy_mapping_fn, batch_builder_factory, extra_batch_callback*)

Tracks the current state of a (possibly multi-agent) episode.

`new_batch_builder`

Create a new MultiAgentSampleBatchBuilder.

Type func

`add_extra_batch`

Return a built MultiAgentBatch to the sampler.

Type func

`batch_builder`

Batch builder for the current episode.

Type obj

`total_reward`

Summed reward across all agents in this episode.

Type float

`length`

Length of this episode.

Type int

`episode_id`
Unique id identifying this trajectory.

Type int

`agent_rewards`
Summed rewards broken down by agent.

Type dict

`custom_metrics`
Dict where the you can add custom metrics.

Type dict

`user_data`
Dict that you can use for temporary storage.

Type dict

Use case 1: Model-based rollouts in multi-agent: A custom compute_actions() function in a policy graph can inspect the current episode state and perform a number of rollouts based on the policies and state of other agents in the environment.

Use case 2: Returning extra rollouts data. The model rollouts can be returned back to the sampler by calling:

```
>>> batch = episode.new_batch_builder()
>>> for each transition:
       batch.add_values(...)  # see sampler for usage
>>> episode.extra_batches.add(batch.build_and_reset())
```

`policy_for`(*agent_id='single_agent'*)
Returns the policy graph for the specified agent.

If the agent is new, the policy mapping fn will be called to bind the agent to a policy for the duration of the episode.

`last_observation_for`(*agent_id='single_agent'*)
Returns the last observation for the specified agent.

`last_info_for`(*agent_id='single_agent'*)
Returns the last info for the specified agent.

`last_action_for`(*agent_id='single_agent'*)
Returns the last action for the specified agent, or zeros.

`prev_action_for`(*agent_id='single_agent'*)
Returns the previous action for the specified agent.

`prev_reward_for`(*agent_id='single_agent'*)
Returns the previous reward for the specified agent.

`rnn_state_for`(*agent_id='single_agent'*)
Returns the last RNN state for the specified agent.

`last_pi_info_for`(*agent_id='single_agent'*)
Returns the last info object for the specified agent.

1.27.4 ray.rllib.models

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 entroy 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 entroy 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*)
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 entroy 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

```
>>> 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, options)
>>> dist = dist_cls(model.outputs)
>>> action = dist.sample()
```

static get_action_dist(*action_space*, *config*, *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(*input_dict*, *obs_space*, *num_outputs*, *options*, *state_in=None*, *seq_lens=None*)
Returns a suitable model conforming to given input and output specs.

Parameters

- **input_dict** (*dict*) – Dict of input tensors to the model, including the observation under the "obs" key.
- **obs_space** (*Space*) – Observation space of the target gym env.
- **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 (*models.Model*)

static get_torch_model(*obs_space*, *num_outputs*, *options=None*, *default_model_cls=None*)
Returns a custom model for PyTorch algorithms.

Parameters

- **obs_space** (*Space*) – The input observation space.
- **num_outputs** (*int*) – The size of the output vector of the model.

- **options** (*dict*) – Optional args to pass to the model constructor.
- **default_model_cls** (*cls*) – Optional class to use if no custom model.

Returns Neural network model.

Return type model (*models.Model*)

static get_preprocessor(*env*, *options=None*)
Returns a suitable preprocessor for the given env.

This is a wrapper for get_preprocessor_for_space().

static get_preprocessor_for_space(*observation_space*, *options=None*)
Returns a suitable preprocessor for the given observation space.

Parameters

- **observation_space** (*Space*) – The input observation space.
- **options** (*dict*) – Options to pass to the preprocessor.

Returns Preprocessor for the observations.

Return type preprocessor (*Preprocessor*)

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**(*input_dict*, *obs_space*, *num_outputs*, *options*, *state_in=None*, *seq_lens=None*)
Defines an abstract network model for use with RLlib.

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).

input_dict
Dictionary of input tensors, including "obs", "prev_action", "prev_reward", "is_training".

Type dict

outputs
The output vector of this model, of shape [BATCH_SIZE, num_outputs].

Type Tensor

last_layer
The feature layer right before the model output, of shape [BATCH_SIZE, f].

Type Tensor

state_init
List of initial recurrent state tensors (if any).

Type list

state_in
List of input recurrent state tensors (if any).

Type list

state_out
List of output recurrent state tensors (if any).

Type list

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

Type Tensor

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).

value_function()
Builds the value function output.

This method can be overridden to customize the implementation of the value function (e.g., not sharing hidden layers).

Returns Tensor of size [BATCH_SIZE] for the value function.

loss()
Builds any built-in (self-supervised) loss for the model.

For example, this can be used to incorporate auto-encoder style losses. Note that this loss has to be included in the policy graph loss to have an effect (done for built-in algorithms).

Returns Scalar tensor for the self-supervised loss.

class ray.rllib.models.**Preprocessor**(*obs_space*, *options=None*)
Defines an abstract observation preprocessor function.

shape
Shape of the preprocessed output.

Type obj

transform(*observation*)
Returns the preprocessed observation.

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

class ray.rllib.models.**LSTM**(*input_dict*, *obs_space*, *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.

1.27.5 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`

The JSON configuration passed to this optimizer.

Type dict

`local_evaluator`

The embedded evaluator instance.

Type *PolicyEvaluator*

`remote_evaluators`

List of remote evaluator replicas, or [].

Type list

`num_steps_trained`

Number of timesteps trained on so far.

Type int

`num_steps_sampled`

Number of timesteps sampled so far.

Type int

`evaluator_resources`

Optional resource requests to set for evaluators created by this optimizer.

Type dict

`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.

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

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

stop()
Release any resources used by this optimizer.

collect_metrics(*timeout_seconds*, *min_history=100*, *selected_evaluators=None*)
Returns evaluator and optimizer stats.

Parameters

- **timeout_seconds** (`int`) – Max wait time for a evaluator before dropping its results. This usually indicates a hung evaluator.
- **min_history** (`int`) – Min history length to smooth results over.
- **selected_evaluators** (`list`) – Override the list of remote evaluators to collect metrics from.

Returns

A training result dict from evaluator metrics with *info* replaced with stats from self.

Return type res (dict)

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 dictionary 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 override worker *batch_steps*.
- **num_workers** (`int`) – Number of remote evaluators
- **num_envs_per_worker** (`int`) – (Optional) Sets the number environments per evaluator 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 has two modes of operation:

- normal replay: replays independent samples.
- **batch replay: simplified mode where entire sample batches are** replayed. This supports RNNs, but not prioritization.

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)

stop()

Release any resources used by this optimizer.

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)

stop()

Release any resources used by this optimizer.

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.

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

Variant of the sync replay optimizer that replays entire batches.

This enables RNN support. Does not currently support 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.

1.27.6 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 A copy of self.

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*, *update_remote=True*)

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.
- **`update_remote`** (*`bool`*) – Whether to push updates to remote 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**(*external_env*, *address*, *port*)

REST server than can be launched from a ExternalEnv.

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(ExternalEnv):
       def __init__(self):
           ExternalEnv.__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)
```

ray.rllib.utils.**merge_dicts**(*d1*, *d2*)

Returns a new dict that is d1 and d2 deep merged.

ray.rllib.utils.**deep_update**(*original*, *new_dict*, *new_keys_allowed*, *whitelist*)

Updates original dict with values from new_dict recursively. If new key is introduced in new_dict, then if new_keys_allowed is not True, an error will be thrown. Further, for sub-dicts, if the key is in the whitelist, then new subkeys can be introduced.

Parameters

- **original** (*dict*) – Dictionary with default values.
- **new_dict** (*dict*) – Dictionary with values to be updated
- **new_keys_allowed** (*bool*) – Whether new keys are allowed.
- **whitelist** (*list*) – List of keys that correspond to dict values where new subkeys can be introduced. This is only at the top level.

1.28 RLlib Examples

This page is an index of examples for the various use cases and features of RLlib.

If any example is broken, or if you'd like to add an example to this page, feel free to raise an issue on our Github repository.

1.28.1 Tuned Examples

- **Tuned examples:** Collection of tuned algorithm hyperparameters.
- **Atari benchmarks:** Collection of reasonably optimized Atari results.

1.28.2 Training Workflows

- **Custom training workflows:** Example of how to use Tune's support for custom training functions to implement custom training workflows.
- **Curriculum learning:** Example of how to adjust the configuration of an environment over time.
- **Custom metrics:** Example of how to output custom training metrics to TensorBoard.

1.28.3 Custom Envs and Models

- **Registering a custom env:** Example of defining and registering a gym env for use with RLlib.
- **Subprocess environment:** Example of how to ensure subprocesses spawned by envs are killed when RLlib exits.
- **Batch normalization:** Example of adding batch norm layers to a custom model.
- **Parametric actions:** Example of how to handle variable-length or parametric action spaces.

1.28.4 Serving and Offline

- **CartPole server:** Example of online serving of predictions for a simple CartPole policy.
- **Saving experiences:** Example of how to externally generate experience batches in RLlib-compatible format.

1.28.5 Multi-Agent and Hierarchical

- **Two-step game:** Example of the two-step game from the QMIX paper.
- **Weight sharing between policies:** Example of how to define weight-sharing layers between two different policies.
- **Multiple trainers:** Example of alternating training between two DQN and PPO trainers.
- **Hierarchical training:** Example of hierarchical training using the multi-agent API.

1.28.6 Community Examples

- **Traffic Flow:** Example of optimizing mixed-autonomy traffic simulations with RLlib / multi-agent.
- **Roboschool / SageMaker:** Example of training robotic control policies in SageMaker with RLlib.
- **StarCraft2:** Example of training in StarCraft2 maps with RLlib / multi-agent.

- **Sequential Social Dilemma Games:** Example of using the multi-agent API to model several social dilemma games.

1.29 Distributed SGD (Experimental)

Ray includes an implementation of synchronous distributed stochastic gradient descent (SGD), which is competitive in performance with implementations in Horovod and Distributed TensorFlow.

Ray SGD is built on top of the Ray task and actor abstractions to provide seamless integration into existing Ray applications.

1.29.1 Interface

To use Ray SGD, define a model class:

class `ray.experimental.sgd.`**Model**

Your class must implement this interface to be used with Ray SGD.

This supports any form of input pipeline: it is up to you to define it using TensorFlow. For an example implementation, see tfbench/test_model.py

Then, pass a model creator function to the `ray.experimental.sgd.DistributedSGD` class. To drive the distributed training, `sgd.step()` can be called repeatedly:

```
model_creator = lambda worker_idx, device_idx: YourModelClass()

sgd = DistributedSGD(
    model_creator,
    num_workers=2,
    devices_per_worker=4,
    gpu=True,
    strategy="ps")

for i in range(NUM_ITERS):
    sgd.step()
```

Under the hood, Ray SGD will create *replicas* of your model onto each hardware device (GPU) allocated to workers (controlled by `num_workers`). Multiple devices can be managed by each worker process (controlled by `devices_per_worker`). Each model instance will be in a separate TF variable scope. The `DistributedSGD` class coordinates the distributed computation and application of gradients to improve the model.

There are two distributed SGD strategies available for use:

- `strategy="simple"`: Gradients are averaged centrally on the driver before being applied to each model replica. This is a reference implementation for debugging purposes.
- `strategy="ps"`: Gradients are computed and averaged within each node. Gradients are then averaged across nodes through a number of parameter server actors. To pipeline the computation of gradients and transmission across the network, we use a custom TensorFlow op that can read and write to the Ray object store directly.

Note that when `num_workers=1`, only local allreduce will be used and the choice of distributed strategy is irrelevant.

The full documentation for `DistributedSGD` is as follows:

class ray.experimental.sgd.**DistributedSGD**(*model_creator*, *num_workers*, *devices_per_worker*, *gpu=True*, *strategy='ps'*, *grad_shard_bytes=10000000*, *all_reduce_alg='simple'*)

Experimental distributed SGD implementation in Ray.

This supports two modes: 'simple': centralized gradient aggregation 'ps': sharded parameter-server implementation

To use this class, you'll have to implement model.py:Model.

Parameters

- **model_creator** (*func*) – Function that returns a model given worker and device indexes as arguments. Each model replica will be created within its own variable scope.
- **num_workers** (*int*) – Number of Ray actors to use for SGD.
- **devices_per_worker** (*int*) – Number of GPU or CPU devices to use per worker. One model replica will be created per device.
- **gpu** (*bool*) – Whether to use GPU devices.
- **strategy** (*str*) – Strategy to use for distributed gradient aggregation. This only applies if num_workers > 1.
- **grad_shard_bytes** (*int*) – Fuse gradient tensors into chunks of at most this size (if applicable).
- **all_reduce_alg** (*str*) – TensorFlow strategy to use for gradient synchronization within the same worker (if applicable). See modified_allreduce.py for options.

Examples

```
>>> # Setup distributed SGD
>>> model_creator = (
...   lambda worker_idx, device_idx: YourModelClass(...))
>>> sgd = DistributedSGD(
...   model_creator, num_workers=2,
...   devices_per_worker=4, gpu=True, strategy="ps")
```

```
>>> # To train
>>> for i in range(100):
...   stats = sgd.step(fetch_stats=i % 10 == 0)
```

```
>>> # To access or update model state
>>> sgd.foreach_model(lambda model: ...)
```

```
>>> # To access or update worker state
>>> sgd.foreach_worker(lambda worker: ...)
```

1.29.2 Examples

For examples of end-to-end usage, check out the ImageNet synthetic data test and also the simple MNIST training example, which includes examples of how access the model weights and monitor accuracy as training progresses.

1.29.3 Performance

When using the new Ray backend (which will be enabled by default in Ray 0.6+), we expect performance competitive with other synchronous SGD implementations on 25Gbps Ethernet.

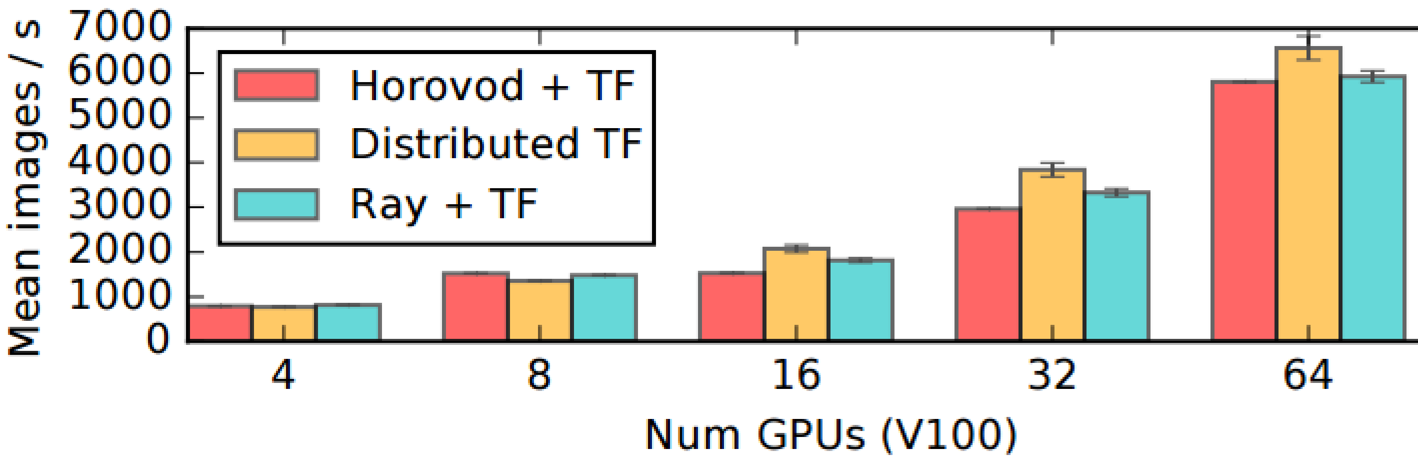


Fig. 5: Images per second reached when distributing the training of a ResNet-101 TensorFlow model (from the official TF benchmark). All experiments were run on p3.16xl instances connected by 25Gbps Ethernet, and workers allocated 4 GPUs per node as done in the Horovod benchmark.

1.30 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.

1.31 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.

1.31.1 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.

```
@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.

```
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.

```
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)
```

1.31.2 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).

```
export OPENBLAS_NUM_THREADS=1
```

1.32 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 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.

```
pip install gym[atari]
pip install tensorflow
```

Then you can run the example as follows.

```
rllib train --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.

```
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.

1.33 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.

```
pip install tensorflow
```

You can view the code for this example.

The examples can be run as follows.

```
# 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.

1.33.1 Asynchronous Parameter Server

The asynchronous parameter server itself is implemented as an actor, which exposes the methods `push` and `pull`.

```
@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.

```
@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.

```
ps = ParameterServer.remote(keys, initial_values)
worker_tasks = [worker_task.remote(ps) for _ in range(4)]
```

1.33.2 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.

```
@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`.

```
@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 **split the parameters across multiple actors**.

1.34 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.
```

(continues on next page)

(continued from previous page)

```
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.

```
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.

```
@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.

```
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.

```
while True:
    all_weights = ray.get([actor.compute_steps.remote(weight_id) for actor in train_
↪actors])
```

(continues on next page)

(continued from previous page)

```
    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)
```

1.35 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 RLlib.

To run the application, first install **ray** and then some dependencies:

```
pip install tensorflow
pip install six
pip install gym[atari]
pip install opencv-python
pip install scipy
```

You can run the code with

```
rllib train --env=Pong-ram-v4 --run=A3C --config='{"num_workers": N}'
```

1.35.1 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.

1.35.2 Worker Code Walkthrough

We use a Ray Actor to simulate the environment.

```
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."""
    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
```

1.35.3 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.

```
import numpy as np
import ray

def train(num_workers, env_name="PongDeterministic-v4"):
    # Setup a copy of the environment
    # Instantiate a copy of the policy - mainly used as a placeholder
    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
```

(continues on next page)

(continued from previous page)

```
    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]
        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
```

1.35.4 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`).

1.36 Batch L-BFGS

This document provides a walkthrough of the L-BFGS example. To run the application, first install these dependencies.

```
pip install tensorflow
pip install scipy
```

You can view the code for this example.

Then you can run the example as follows.

```
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.

1.36.1 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.

```
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.

```
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`.

```
theta_init = 1e-2 * np.random.normal(size=dim)
result = scipy.optimize.fmin_l_bfgs_b(full_loss, theta_init, fprime=full_grad)
```

1.36.2 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.

```
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, lets turn `loss` and `grad` into methods of an actor that will contain our network.

```
class Network(object):
    def __init__():
        # Initialize network.

    def loss(theta, xs, ys):
        # compute the loss
        return loss
```

(continues on next page)

(continued from previous page)

```
    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.

```
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␣
↪for use with fmin_l_bfgs_b.
```

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)
```

1.37 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.

```
rllib train --env=Humanoid-v1 --run=ES
```

To train a policy on a cluster (e.g., using 900 workers), run the following.

```
rllib train \
    --env=Humanoid-v1 \
    --run=ES \
    --redis-address=<redis-address> \
    --config='{"num_workers": 900, "episodes_per_batch": 10000, "train_batch_size":␣
↪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.

```
@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.

```
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.

```
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.

```
@ray.remote
def create_shared_noise():
    noise = np.random.randn(250000000)
    return noise
```

(continues on next page)

(continued from previous page)

```
noise_id = create_shared_noise.remote()
```

Recall that the `noise_id` argument is passed into the actor constructor.

1.38 Cython

1.38.1 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`:

```
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.

1.38.2 Notes

- You **must** include the following two lines at the top of any `*.pyx` file:

```
#!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:

```
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).

1.39 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

```
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
```

(continues on next page)

(continued from previous page)

```
   in 1475
   to 645
   a 583
   is 572
   as 352
   by 318
   for 306
...
```

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
         # in the argument 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.

1.40 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).

```
import tensorflow as tf
import numpy as np

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

loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
grads = optimizer.compute_gradients(loss)
train = optimizer.apply_gradients(grads)

init = tf.global_variables_initializer()
sess = tf.Session()
```

To extract the weights and set the weights, you can use the following helper method.

```
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.

```
# 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.

```
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.
```

1.40.1 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.

```
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()
```

(continues on next page)

(continued from previous page)

```
    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)]

# 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) / 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))
```

1.40.2 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.

```
x_values = [1] * 100
y_values = [2] * 100
numerical_grads = sess.run([grad[0] for grad in grads], feed_dict={x_data: x_values,␣
↪y_data: y_values})
```

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.

```
# 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:

```
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.
        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)
```

(continues on next page)

(continued from previous page)

```
        # 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. 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.
```

(continues on next page)

(continued from previous page)

```
    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
The tensorflow session used to run assignment.

Type tf.Session

variables
Extracted variables from the loss or additional variables that are passed in.

Type Dict[str, tf.Variable]

placeholders
Placeholders for weights.

Type Dict[str, tf.placeholders]

assignment_nodes
Nodes that assign weights.

Type Dict[str, tf.Tensor]

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.

1.40.3 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 definiton `Network` with a `TensorFlowVariables` instance:

```
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()
```

and run the following code:

```
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:

```
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.

1.41 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.

1.41.1 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, 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.

1.41.2 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.

1.41.3 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 another local 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.

    ```
    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.

  ```
  TODO: Fill this in.
  ```

  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.

1.41.4 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.

1.42 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.

1.42.1 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.

1.42.2 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.

  ```
  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).

  ```
  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.

1.42.3 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).

```
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.

1.43 Fault Tolerance

This document describes the handling of failures in Ray.

1.43.1 Machine and Process Failures

Each **raylet** (the scheduler process) sends heartbeats to a **monitor** process. If the monitor does not receive any heartbeats from a given raylet for some period of time (about ten seconds), then it will mark that process as dead.

1.43.2 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. This is the standard *lineage-based fault tolerance* strategy used by other systems like Spark.

1.43.3 Actors

When an actor dies (either because the actor process crashed or because the node that the actor was on died), by default any attempt to get an object from that actor that cannot be created will raise an exception. Subsequent releases will include an option for automatically restarting actors.

1.43.4 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 and 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.

1.44 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.

1.44.1 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"
# This typically corresponds to 20000 2MB pages (about 40GB), but this
# depends on the platform.
sudo bash -c "echo 20000 > /proc/sys/vm/nr_hugepages"
```

Note: Once you create the huge pages, they will take up memory which will never be freed unless you remove the huge pages. If you run into memory issues, that may be the issue.

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.

1.45 Resources (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.

1.45.1 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`.

```
# 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.

```
ray.init(num_cpus=8, num_gpus=1)
```

If the number of CPUs is unspecified, Ray will automatically determine the number by running `multiprocessing.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. Note that Ray supports **fractional** resource requirements.

```
@ray.remote(num_cpus=4, num_gpus=2)
def f():
    return 1

@ray.remote(num_gpus=0.5)
def h():
    return 1
```

The `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.

```
@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.

```
@ray.remote(num_gpus=1)
class Actor(object):
    pass
```

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**.

1.45.2 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.

```
ray start --head --resources='{"Resource1": 4, "Resource2": 16}'
```

It can be done through `ray.init` as follows.

```
ray.init(resources={'Resource1': 4, 'Resource2': 16})
```

To require custom resources in a task, specify the requirements in the remote decorator.

```
@ray.remote(resources={'Resource2': 1})
def f():
    return 1
```

1.45.3 Fractional Resources

Task and actor resource requirements can be fractional. This is particularly useful if you want multiple tasks or actors to share a single GPU.

1.46 Temporary Files

Ray will produce some temporary files during running. They are useful for logging, debugging & sharing object store with other programs.

1.46.1 Location of Temporary Files

First we introduce the concept of a session of Ray.

A session contains a set of processes. A session is created by executing `ray start` command or call `ray.init()` in a Python script and ended by executing `ray stop` or call `ray.shutdown()`.

For each session, Ray will create a *root temporary directory* to place all its temporary files. The path is `/tmp/ray/session_{datetime}_{pid}` by default. The pid belongs to the startup process (the process calling `ray.init()` or the Ray process executed by a shell in `ray start`). You can sort by their names to find the latest session.

You are allowed to change the *root temporary directory* in one of these ways:

- Pass `--temp-dir={your temp path}` to `ray start`
- Specify `temp_dir` when call `ray.init()`

You can also use `default_worker.py --temp-dir={your temp path}` to start a new worker with given *root temporary directory*.

The *root temporary directory* you specified will be given as it is, without pids or datetime attached.

1.46.2 Layout of Temporary Files

A typical layout of temporary files could look like this:

```
/tmp
└── ray
    └── session_{datetime}_{pid}
        ├── logs  # for logging
        │   ├── log_monitor.err
        │   ├── log_monitor.out
        │   ├── monitor.err
        │   ├── monitor.out
        │   ├── plasma_store_0.err  # array of plasma stores' outputs
        │   ├── plasma_store_0.out
        │   ├── raylet_0.err  # array of raylets' outputs. Control it with `--no-
↪redirect-worker-output` (in Ray's command line) or `redirect_worker_output` (in ray.
↪init())
        │   ├── raylet_0.out
        │   ├── redis-shard_0.err   # array of redis shards' outputs
        │   ├── redis-shard_0.out
        │   ├── redis.err  # redis
        │   ├── redis.out
        │   ├── webui.err  # ipython notebook web ui
        │   ├── webui.out
        │   ├── worker-{worker_id}.err  # redirected output of workers
        │   ├── worker-{worker_id}.out
        │   └── {other workers}
        ├── ray_ui.ipynb  # ipython notebook file
        └── sockets  # for sockets
            ├── plasma_store
            └── raylet  # this could be deleted by Ray's shutdown cleanup.
```

1.46.3 Plasma Object Store Socket

Plasma object store sockets can be used to share objects with other programs using Apache Arrow.

You are allowed to specify the plasma object store socket in one of these ways:

- Pass `--plasma-store-socket-name={your socket path}` to `ray start`
- Specify `plasma_store_socket_name` when call `ray.init()`

The path you specified will be given as it is without being affected any other paths.

1.46.4 Notes

Temporary file policies are defined in `python/ray/tempfile_services.py`.

Currently, we keep `/tmp/ray` as the default directory for temporary data files of RLlib as before. It is not very reasonable and could be changed later.

1.47 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.

1.47.1 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.

1.47.2 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/ray/session_*/logs/monitor*`. 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.

    ```
    sudo bash -c "echo $USER hard nofile 65536 >> /etc/security/limits.conf"
    ```

  - Logout and log back in.

1.47.3 Hanging

Tip: You can run `ray stack` to dump the stack traces of all Ray workers on the current node. This requires py-spy to be installed.

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.

1.47.4 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.

1.47.5 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.

```
@ray.remote
def f():
    return 1

@ray.remote
```

(continues on next page)

(continued from previous page)

```
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

  ```
  def h():
      return 1
  ```

  And you define remote function `f` as

  ```
  @ray.remote
  def f():
      return file.h()
  ```

  You can redefine `f` as follows.

  ```
  @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`.

1.48 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!

1.48.1 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:

```python
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:

```python
# 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:

```python
# 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()`:

```python
# A local function executed on the driver, not on Ray
def other_func():
    time.sleep(0.3)

def ex3():
    list3 = []
    for i in range(5):
        other_func()
        list3.append(func.remote())
    ray.get(list3)
```

1.48.2 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:

```
# 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()`:

```
@time_this  # 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, 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.

```
@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):
```

(continues on next page)

(continued from previous page)

```python
        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.

```python
# 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 Amdahls 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.

1.48.3 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:

```
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:

```
@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:

```
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:

```
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

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    29                                           @profile
    30                                           def ex1():
    31         1          3.0      3.0      0.0   list1 = []
    32         6         18.0      3.0      0.0   for i in range(5):
    33         5    2508805.0 501761.0    100.0     list1.append(ray.get(func.
↪remote()))
```

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

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    36                                           @profile
    37                                           def ex2():
    38         1          2.0      2.0      0.0   list2 = []
    39         6         13.0      2.2      0.0   for i in range(5):
    40         5        637.0    127.4      0.1     list2.append(func.remote())
    41         1     1002919.0 1002919.0     99.9     ray.get(list2)
```

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

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    44                                           @profile
    45                                           #@time_this
    46                                           def ex3():
    47         1          2.0      2.0      0.0   list3 = []
    48         6         13.0      2.2      0.0   for i in range(5):
    49         5    1501934.0 300386.8     74.9     other_func()
    50         5        917.0    183.4      0.0     list3.append(func.remote())
    51         1     501589.0 501589.0     25.0   ray.get(list3)
```

1.48.4 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:

```
import cProfile  # Added import statement

def ex1():
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))

def main():
    ray.init()
    cProfile.run('ex1()')  # Modified call to ex1
```

(continues on next page)

(continued from previous page)

```
    cProfile.run('ex2()')
    cProfile.run('ex3()')

if __name__ == "__main__":
    main()
```

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):
        self.sleepValue = 0.5

    # Equivalent to func(), but defined within an actor
    def actor_func(self):
        time.sleep(self.sleepValue)
```

Recalling the suboptimality of `ex1`, let's first see what happens if we attempt to perform all five `actor_func()` calls within a single actor:

```
def ex4():
    # This is suboptimal in Ray, and should only be used for the sake of this example
    actor_example = Sleeper.remote()

    five_results = []
    for i in range(5):
        five_results.append(actor_example.actor_func.remote())

    # Wait until the end to call ray.get()
    ray.get(five_results)
```

We enable cProfile on this example as follows:

```
def main():
    ray.init()
    cProfile.run('ex4()')

if __name__ == "__main__":
    main()
```

Running our new Actor example, 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 `actor_func()` to run in serial. We remember in `ex1` that this behavior was because we did not wait until after submitting all five remote function tasks to call `ray.get()`, but we can verify on cProfile's output line `worker.py:2535(get)` that `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 `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 `ex4`, we can take advantage that each call to `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 `actor_func()` at a time.

```
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
```

(continues on next page)

(continued from previous page)

```
    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)
...
```

1.48.5 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/ray/session_2018-11-01_14-31-43_
↪27211/logs.
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}.

======================================================================
View the web UI at http://localhost:8897/notebooks/ray_ui84907.ipynb?
↪token=025e8ab295270a57fac209204b37349fdf34e037671a13ff
======================================================================
```

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.

1.49 Security

This document describes best security practices for using Ray.

1.49.1 Intended Use and Threat Model

Ray instances should run on a secure network without public facing ports. The most common threat for Ray instances is unauthorized access to Redis, which can be exploited to gain shell access and run arbitray code. The best fix is to run Ray instances on a secure, trusted network.

Running Ray on a secured network is not always feasible, so Ray provides some basic security features:

1.49.2 Redis Port Authentication

To prevent exploits via unauthorized Redis access, Ray provides the option to password-protect Redis ports. While this is not a replacement for running Ray behind a firewall, this feature is useful for instances exposed to the internet where configuring a firewall is not possible. Because Redis is very fast at serving queries, the chosen password should be long.

Redis authentication is only supported on the raylet code path.

To add authentication via the Python API, start Ray using:

```
ray.init(redis_password="password")
```

To add authentication via the CLI, or connect to an existing Ray instance with password-protected Redis ports:

```
ray start [--head] --redis-password="password"
```

While Redis port authentication may protect against external attackers, Ray does not encrypt traffic between nodes so man-in-the-middle attacks are possible for clusters on untrusted networks.

1.49.3 Cloud Security

Launching Ray clusters on AWS or GCP using the `ray up` command automatically configures security groups that prevent external Redis access.

1.49.4 References

- The *Redis security documentation <https://redis.io/topics/security>*

1.50 Development Tips

1.50.1 Compilation

To speed 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
```

1.50.2 Debugging

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.

You can also get a core dump of the `raylet` process, which is especially useful when filing issues. The process to obtain a core dump is OS-specific, but usually involves running `ulimit -c unlimited` before starting Ray to allow core dump files to be written.

Inspecting Redis shards

To inspect Redis, you can use the `ray.experimental.state.GlobalState` Python API. The easiest way to do this is to start or connect to a Ray cluster with `ray.init()`, then query the API like so:

```
ray.init()
ray.worker.global_state.client_table()
# Returns current information about the nodes in the cluster, such as:
# [{'ClientID': '2a9d2b34ad24a37ed54e4fcd32bf19f915742f5b',
#   'IsInsertion': True,
#   'NodeManagerAddress': '1.2.3.4',
#   'NodeManagerPort': 43280,
#   'ObjectManagerPort': 38062,
#   'ObjectStoreSocketName': '/tmp/ray/session_2019-01-21_16-28-05_4216/sockets/
↪plasma_store',
#   'RayletSocketName': '/tmp/ray/session_2019-01-21_16-28-05_4216/sockets/raylet',
#   'Resources': {'CPU': 8.0, 'GPU': 1.0}}]
```

To inspect the primary Redis shard manually, you can also query 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.lrange('RedisShards', 0, -1)
```

Backend logging

The `raylet` process logs detailed information about events like task execution and object transfers between nodes. To set the logging level at runtime, you can set the `RAY_BACKEND_LOG_LEVEL` environment variable before starting Ray. For example, you can do:

```
export RAY_BACKEND_LOG_LEVEL=debug
ray start
```

This will print any `RAY_LOG(DEBUG)` lines in the source code to the `raylet.err` file, which you can find in the Temporary Files.

1.50.3 Testing 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 -m pytest -v test/runtest.py::test_keyword_args
```

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.

1.50.4 Linting

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`.

Autoformatting code. We use `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.

1.51 Profiling for Ray Developers

This document details, for Ray developers, how to use `pprof` to profile Ray binaries.

1.51.1 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
```

1.51.2 Launching the to-profile binary

If you want to launch Ray in profiling mode, define the following variables:

```
export RAYLET_PERFTOOLS_PATH=/usr/lib/x86_64-linux-gnu/libprofiler.so
export RAYLET_PERFTOOLS_LOGFILE=/tmp/pprof.out
```

The file `/tmp/pprof.out` will be empty until you let the binary run the target workload for a while and then `kill` it via `ray stop` or by letting the driver exit.

1.51.3 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.

```
# Use the appropriate path.
RAYLET=ray/python/ray/core/src/ray/raylet/raylet

google-pprof -svg $RAYLET /tmp/pprof.out > /tmp/pprof.svg
# Then open the .svg file with Chrome.

# If you realize the call graph is too large, use -focus=<some function> to zoom
# into subtrees.
google-pprof -focus=epoll_wait -svg $RAYLET /tmp/pprof.out > /tmp/pprof.svg
```

Here's a snapshot of an example svg output, taken from the official documentation:

1.51.4 References

- The pprof documentation.
- A Go version of pprof.
- The gperftools, including libprofiler, tcmalloc, and other goodies.

1.52 Contact

The following are good places to discuss Ray.

1. ray-dev@googlegroups.com: For discussions about development or any general questions.
2. StackOverflow: For questions about how to use Ray.
3. GitHub Issues: For bug reports and feature requests.

Python Module Index

Index