

---

# Ray Documentation

*Release 0.8.5*

**The Ray Team**

**May 07, 2020**



# INSTALLATION

<b>1</b>	<b>Quick Start</b>	<b>3</b>
<b>2</b>	<b>Tune Quick Start</b>	<b>5</b>
<b>3</b>	<b>RLLib Quick Start</b>	<b>7</b>
<b>4</b>	<b>More Information</b>	<b>9</b>
4.1	Blog and Press . . . . .	9
4.2	Talks (Videos) . . . . .	10
4.3	Slides . . . . .	10
4.4	Academic Papers . . . . .	10
<b>5</b>	<b>Getting Involved</b>	<b>11</b>
5.1	Installing Ray . . . . .	11
5.2	Ray Core Walkthrough . . . . .	15
5.3	Using Ray . . . . .	21
5.4	Configuring Ray . . . . .	60
5.5	Ray Dashboard . . . . .	63
5.6	Deploying Ray . . . . .	71
5.7	Ray Tutorials and Examples . . . . .	96
5.8	Ray Package Reference . . . . .	124
5.9	Tune: Scalable Hyperparameter Tuning . . . . .	140
5.10	Tutorials, User Guides, Examples . . . . .	143
5.11	Tune Trial Schedulers . . . . .	180
5.12	Tune Search Algorithms . . . . .	188
5.13	Tune API Reference . . . . .	203
5.14	Contributing to Tune . . . . .	256
5.15	RLLib: Scalable Reinforcement Learning . . . . .	258
5.16	RLLib Table of Contents . . . . .	260
5.17	RLLib Training APIs . . . . .	264
5.18	RLLib Environments . . . . .	289
5.19	RLLib Models, Preprocessors, and Action Distributions . . . . .	298
5.20	RLLib Algorithms . . . . .	312
5.21	RLLib Offline Datasets . . . . .	338
5.22	RLLib Concepts and Custom Algorithms . . . . .	344
5.23	RLLib Examples . . . . .	355
5.24	RLLib Package Reference . . . . .	357
5.25	Contributing to RLLib . . . . .	404
5.26	RaySGD: Distributed Training Wrappers . . . . .	407
5.27	Distributed PyTorch . . . . .	408

5.28	Distributed TensorFlow . . . . .	423
5.29	RaySGD API Documentation . . . . .	426
5.30	RayServe: Scalable and Programmable Serving . . . . .	436
5.31	Distributed multiprocessing.Pool . . . . .	440
5.32	Distributed Scikit-learn / Joblib . . . . .	441
5.33	Parallel Iterators . . . . .	442
5.34	Pandas on Ray . . . . .	451
5.35	Development Tips . . . . .	451
5.36	Profiling for Ray Developers . . . . .	456
5.37	Fault Tolerance . . . . .	458
5.38	Getting Involved . . . . .	459
	<b>Python Module Index</b>	<b>463</b>
	<b>Index</b>	<b>465</b>



**Ray is a fast and simple framework for building and running distributed applications.**

Ray is packaged with the following libraries for accelerating machine learning workloads:

- [Tune](#): Scalable Hyperparameter Tuning
- [RLlib](#): Scalable Reinforcement Learning
- [RaySGD](#): Distributed Training Wrappers
- [\*RayServe: Scalable and Programmable Serving\*](#)

Star us on [on GitHub](#). You can also get started by visiting our [Tutorials](#). For the latest wheels (nightlies), see the [installation page](#).

---

**Important:** Join our [community slack](#) to discuss Ray!

---



---

# CHAPTER ONE

---

## QUICK START

First, install Ray with: pip install ray

```
# Execute Python functions in parallel.

import ray
ray.init()

@ray.remote
def f(x):
    return x * x

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

To use Ray's actor model:

```
import ray
ray.init()

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

    def increment(self):
        self.n += 1

    def read(self):
        return self.n

counters = [Counter.remote() for i in range(4)]
[c.increment.remote() for c in counters]
futures = [c.read.remote() for c in counters]
print(ray.get(futures))
```

Visit the [Walkthrough](#) page a more comprehensive overview of Ray features.

Ray programs can run on a single machine, and can also seamlessly scale to large clusters. To execute the above Ray script in the cloud, just download [this configuration file](#), and run:

```
ray submit [CLUSTER.YAML] example.py --start
```

Read more about [launching clusters](#).



---

## CHAPTER TWO

---

### TUNE QUICK START

Tune is a library for hyperparameter tuning at any scale. With Tune, you can launch a multi-node distributed hyperparameter sweep in less than 10 lines of code. Tune supports any deep learning framework, including PyTorch, TensorFlow, and Keras.

---

**Note:** To run this example, you will need to install the following:

```
$ pip install ray torch torchvision filelock
```

---

This example runs a small grid search to train a CNN using PyTorch and Tune.

```
import torch.optim as optim
from ray import tune
from ray.tune.examples.mnist_pytorch import get_data_loaders, ConvNet, train, test

def train_mnist(config):
    train_loader, test_loader = get_data_loaders()
    model = ConvNet()
    optimizer = optim.SGD(model.parameters(), lr=config["lr"])
    for i in range(10):
        train(model, optimizer, train_loader)
        acc = test(model, test_loader)
        tune.track.log(mean_accuracy=acc)

analysis = tune.run(
    train_mnist, config={"lr": tune.grid_search([0.001, 0.01, 0.1])})

print("Best config: ", analysis.get_best_config(metric="mean_accuracy"))

# Get a dataframe for analyzing trial results.
df = analysis.dataframe()
```

If TensorBoard is installed, automatically visualize all trial results:

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



## RLLIB QUICK START

RLLib is an open-source library for reinforcement learning built on top of Ray that offers both high scalability and a unified API for a variety of applications.

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

```
import gym
from gym.spaces import Discrete, Box
from ray import tune

class SimpleCorridor(gym.Env):
    def __init__(self, config):
        self.end_pos = config["corridor_length"]
        self.cur_pos = 0
        self.action_space = Discrete(2)
        self.observation_space = Box(0.0, self.end_pos, shape=(1,))

    def reset(self):
        self.cur_pos = 0
        return [self.cur_pos]

    def step(self, action):
        if action == 0 and self.cur_pos > 0:
            self.cur_pos -= 1
        elif action == 1:
            self.cur_pos += 1
        done = self.cur_pos >= self.end_pos
        return [self.cur_pos], 1 if done else 0, done, {}

tune.run(
    "PPO",
    config={
        "env": SimpleCorridor,
        "num_workers": 4,
        "env_config": {"corridor_length": 5}})
```



## MORE INFORMATION

Here are some talks, papers, and press coverage involving Ray and its libraries. Please raise an issue if any of the below links are broken!

### 4.1 Blog and Press

- [Modern Parallel and Distributed Python: A Quick Tutorial on Ray](#)
- [Why Every Python Developer Will Love Ray](#)
- [Ray: A Distributed System for AI \(BAIR\)](#)
- [10x Faster Parallel Python Without Python Multiprocessing](#)
- [Implementing A Parameter Server in 15 Lines of Python with Ray](#)
- [Ray Distributed AI Framework Curriculum](#)
- [RayOnSpark: Running Emerging AI Applications on Big Data Clusters with Ray and Analytics Zoo](#)
- [First user tips for Ray](#)
- [\[Tune\] Tune: a Python library for fast hyperparameter tuning at any scale](#)
- [\[Tune\] Cutting edge hyperparameter tuning with Ray Tune](#)
- [\[RLLib\] New Library Targets High Speed Reinforcement Learning](#)
- [\[RLLib\] Scaling Multi Agent Reinforcement Learning](#)
- [\[RLLib\] Functional RL with Keras and Tensorflow Eager](#)
- [\[Modin\] How to Speed up Pandas by 4x with one line of code](#)
- [\[Modin\] Quick Tip – Speed up Pandas using Modin](#)
- [Ray Blog](#)

## 4.2 Talks (Videos)

- Programming at any Scale with Ray | SF Python Meetup Sept 2019
- Ray for Reinforcement Learning | Data Council 2019
- Scaling Interactive Pandas Workflows with Modin
- Ray: A Distributed Execution Framework for AI | SciPy 2018
- Ray: A Cluster Computing Engine for Reinforcement Learning Applications | Spark Summit
- RLlib: Ray Reinforcement Learning Library | RISECamp 2018
- Enabling Composition in Distributed Reinforcement Learning | Spark Summit 2018
- Tune: Distributed Hyperparameter Search | RISECamp 2018

## 4.3 Slides

- Talk given at UC Berkeley DS100
- Talk given in October 2019
- [Tune] Talk given at RISECamp 2019

## 4.4 Academic Papers

- Ray paper
- Ray HotOS paper
- RLlib paper
- Tune paper

## GETTING INVOLVED

- [ray-dev@googlegroups.com](mailto:ray-dev@googlegroups.com): For discussions about development or any general questions.
- [StackOverflow](#): For questions about how to use Ray.
- [GitHub Issues](#): For reporting bugs and feature requests.
- [Pull Requests](#): For submitting code contributions.

### 5.1 Installing Ray

---

**Important:** Join our [community slack](#) to discuss Ray!

---

Ray currently supports MacOS and Linux. Windows support is planned for the future.

#### 5.1.1 Latest stable version

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

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

#### 5.1.2 Latest Snapshots (Nightlies)

Here are links to the latest wheels (which are built for each commit on the master branch). To install these wheels, run the following command:

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

Linux	MacOS
Linux Python 3.8	MacOS Python 3.8
Linux Python 3.7	MacOS Python 3.7
Linux Python 3.6	MacOS Python 3.6
Linux Python 3.5	MacOS Python 3.5

### 5.1.3 Installing from a specific commit

You can install the Ray wheels of any particular commit on master with the following template. You need to specify the commit hash, Ray version, Operating System, and Python version:

```
pip install https://ray-wheels.s3-us-west-2.amazonaws.com/master/{COMMIT_HASH}/ray-
˓→{RAY_VERSION}-{PYTHON_VERSION}-{PYTHON_VERSION}m-{OS_VERSION}_intel.whl
```

For example, here are the Ray 0.9.0.dev0 wheels for Python 3.5, MacOS for commit a0ba4499ac645c9d3e82e68f3a281e48ad57f873:

```
pip install https://ray-wheels.s3-us-west-2.amazonaws.com/master/
˓→a0ba4499ac645c9d3e82e68f3a281e48ad57f873/ray-0.9.0.dev0-cp35-cp35m-macosx_10_13_
˓→intel.whl
```

### 5.1.4 Building Ray from Source

Installing from pip should be sufficient for most Ray users.

However, should you need to build from source, follow instructions below for both Linux and MacOS.

#### Dependencies

To build Ray, first install the following dependencies.

For Ubuntu, run the following commands:

```
sudo apt-get update
sudo apt-get install -y build-essential curl unzip psmisc

pip install cython==0.29.0 pytest
```

For MacOS, run the following commands:

```
brew update
brew install wget

pip install cython==0.29.0 pytest
```

#### Install Ray

Ray can be built from the repository as follows.

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

# Install Bazel.
ray/ci/travis/install-bazel.sh

# Optionally build the dashboard (requires Node.js, see below for more information).
pushd ray/python/ray/dashboard/client
npm ci
npm run build
popd
```

(continues on next page)

(continued from previous page)

```
# Install Ray.
cd ray/python
pip install -e . --verbose # Add --user if you see a permission denied error.
```

### [Optional] Dashboard support

If you would like to use the dashboard, you will additionally need to install [Node.js](#) and build the dashboard before installing Ray. The relevant build steps are included in the installation instructions above.

The dashboard requires a few additional Python packages, which can be installed via pip.

```
pip install ray[dashboard]
```

The command `ray.init()` or `ray start --head` will print out the address of the dashboard. For example,

```
>>> import ray
>>> ray.init()
=====
View the dashboard at http://127.0.0.1:8265.
Note: If Ray is running on a remote node, you will need to set up an
SSH tunnel with local port forwarding in order to access the dashboard
in your browser, e.g. by running 'ssh -L 8265:127.0.0.1:8265
<username>@<host>'. Alternatively, you can set webui_host="0.0.0.0" in
the call to ray.init() to allow direct access from external machines.
=====
```

### 5.1.5 Installing Ray on Arch Linux

Note: Installing Ray on Arch Linux is not tested by the Project Ray developers.

Ray is available on Arch Linux via the Arch User Repository ([AUR](#)) as `python-ray`.

You can manually install the package by following the instructions on the [Arch Wiki](#) or use an [AUR helper](#) like `yay` (recommended for ease of install) as follows:

```
yay -S python-ray
```

To discuss any issues related to this package refer to the comments section on the AUR page of `python-ray` [here](#).

### 5.1.6 Installing Ray with Anaconda

If you use [Anaconda](#) and want to use Ray in a defined environment, e.g., `ray`, use these commands:

```
conda create --name ray
conda activate ray
conda install --name ray pip
pip install ray
```

Use `pip list` to confirm that `ray` is installed.

### 5.1.7 Docker Source 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	SIZE	TAG	IMAGE ID	CREATED
ray-project/examples	3.257 GB	latest	7584bde65894	4 days
ray-project/deploy	2.899 GB	latest	970966166c71	4 days
ray-project/base-deps	2.649 GB	latest	f45d66963151	4 days
ubuntu	129.5 MB	xenial	f49eec89601e	3 weeks

### 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#
```

## 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 python/ray/tests/test_mini.py
```

### 5.1.8 Troubleshooting 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.

## 5.2 Ray Core Walkthrough

This walkthrough will overview the core concepts of Ray:

1. Starting Ray
2. Using remote functions (tasks) [`ray.remote`]
3. Fetching results (object IDs) [`ray.put`, `ray.get`, `ray.wait`]
4. Using remote classes (actors) [`ray.remote`]

With Ray, your code will work on a single machine and can be easily scaled to large cluster.

### 5.2.1 Installation

To run this walkthrough, install Ray with `pip install -U ray`. For the latest wheels (for a snapshot of master), you can use these instructions at [Latest Snapshots \(Nightlies\)](#).

### 5.2.2 Starting Ray

You can start Ray on a single machine by adding this to your python script.

```
import ray

# Start Ray. If you're connecting to an existing cluster, you would use
# ray.init(address=<cluster-address>) instead.
ray.init()
```

(continues on next page)

(continued from previous page)

```
...
```

Ray will then be able to utilize all cores of your machine. Find out how to configure the number of cores Ray will use at [Configuring Ray](#).

To start a multi-node Ray cluster, see the [cluster setup page](#).

### 5.2.3 Remote functions (Tasks)

Ray enables arbitrary Python functions to be executed asynchronously. These asynchronous Ray functions are called “remote functions”. The standard way to turn a Python function into a remote function is to add the `@ray.remote` decorator. Here is an example.

```
# A regular Python function.
def regular_function():
    return 1

# A Ray remote function.
@ray.remote
def remote_function():
    return 1
```

This causes a few changes in behavior:

1. **Invocation:** The regular version is called with `regular_function()`, whereas the remote version is called with `remote_function.remote()`.
2. **Return values:** `regular_function` immediately executes and returns 1, whereas `remote_function` immediately returns an object ID (a future) and then creates a task that will be executed on a worker process. The result can be retrieved with `ray.get`.

```
assert regular_function() == 1

object_id = remote_function.remote()

# The value of the original `regular_function`
assert ray.get(object_id) == 1
```

3. **Parallelism:** Invocations of `regular_function` happen **serially**, for example

```
# These happen serially.
for _ in range(4):
    regular_function()
```

whereas invocations of `remote_function` happen in **parallel**, for example

```
# These happen in parallel.
for _ in range(4):
    remote_function.remote()
```

The invocations are executed in parallel because the call to `remote_function.remote()` doesn’t block. All computation is performed in the background, driven by Ray’s internal event loop.

See the [ray.remote package reference](#) page for specific documentation on how to use `ray.remote`.

**Object IDs** can also be passed into remote functions. When the function actually gets executed, **the argument will be a retrieved as a regular Python object**. For example, take this function:

```
@ray.remote
def remote_chain_function(value):
    return value + 1

y1_id = remote_function.remote()
assert ray.get(y1_id) == 1

chained_id = remote_chain_function.remote(y1_id)
assert ray.get(chained_id) == 2
```

Note the following behaviors:

- The second task will not be executed until the first task has finished executing because the second task depends on the output of the first task.
- If the two tasks are scheduled on different machines, the output of the first task (the value corresponding to `y1_id`) will be sent over the network to the machine where the second task is scheduled.

Oftentimes, you may want to specify a task's resource requirements (for example one task may require a GPU). The `ray.init()` command will automatically detect the available GPUs and CPUs on the machine. However, you can override this default behavior by passing in specific resources, e.g., `ray.init(num_cpus=8, num_gpus=4, resources={'Custom': 2})`.

To specify a task's CPU and GPU requirements, pass the `num_cpus` and `num_gpus` arguments into the `remote` decorator. The task will only run on a machine if there are enough CPU and GPU (and other custom) resources available to execute the task. Ray can also handle arbitrary custom resources.

**Note:**

- If you do not specify any resources in the `@ray.remote` decorator, the default is 1 CPU resource and no other resources.
- If specifying CPUs, Ray does not enforce isolation (i.e., your task is expected to honor its request).
- If specifying GPUs, Ray does provide isolation in forms of visible devices (setting the environment variable `CUDA_VISIBLE_DEVICES`), but it is the task's responsibility to actually use the GPUs (e.g., through a deep learning framework like TensorFlow or PyTorch).

```
@ray.remote(num_cpus=4, num_gpus=2)
def f():
    return 1
```

The resource requirements of a task have implications for the Ray's scheduling concurrency. In particular, the sum of the resource requirements of all of the concurrently executing tasks on a given node cannot exceed the node's total resources.

Below are more examples of resource specifications:

```
# Ray also supports fractional resource requirements
@ray.remote(num_gpus=0.5)
def h():
    return 1

# Ray support custom resources too.
```

(continues on next page)

(continued from previous page)

```
@ray.remote(resources={'Custom': 1})
def f():
    return 1
```

Further, remote functions 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()
```

## 5.2.4 Objects in Ray

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 [shared-memory 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.

Object IDs can be created in multiple ways.

1. They are returned by remote function calls.
2. They are returned by `ray.put`.

```
y = 1
object_id = ray.put(y)
```

```
ray.put(value, weakref=False)
Store an object in the object store.
```

The object may not be evicted while a reference to the returned ID exists.

### Parameters

- **value** – The Python object to be stored.
- **weakref** – If set, allows the object to be evicted while a reference to the returned ID exists. You might want to set this if putting a lot of objects that you might not need in the future. It allows Ray to more aggressively reclaim memory.

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

---

**Important:** 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.

---

## 5.2.5 Fetching Results

The command `ray.get(x_id, timeout=None)` takes an object ID and creates a Python object from the corresponding remote object. First, if the current node's object store does not contain the object, the object is downloaded. Then, if the object is a [numpy array](#) or a collection of numpy arrays, the `get` call is zero-copy and returns arrays backed by shared object store memory. Otherwise, we deserialize the object data into a Python object.

```
y = 1
obj_id = ray.put(y)
assert ray.get(obj_id) == 1
```

You can also set a timeout to return early from a `get` that's blocking for too long.

```
from ray.exceptions import RayTimeoutError

@ray.remote
def long_running_function():
    time.sleep(8)

obj_id = long_running_function.remote()
try:
    ray.get(obj_id, timeout=4)
except RayTimeoutError:
    print("`get` timed out.")
```

`ray.get(object_ids, timeout=None)`

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.

This method will issue a warning if it's running inside async context, you can use `await object_id` instead of `ray.get(object_id)`. For a list of object ids, you can use `await asyncio.gather(*object_ids)`.

### Parameters

- `object_ids` – Object ID of the object to get or a list of object IDs to get.
- `timeout (Optional [float])` – The maximum amount of time in seconds to wait before returning.

**Returns** A Python object or a list of Python objects.

### Raises

- `RayTimeoutError` – A `RayTimeoutError` is raised if a timeout is set and the `get` takes longer than `timeout` to return.
- `Exception` – An exception is raised if the task that created the object or that created one of the objects raised an exception.

After launching a number of tasks, you may want to know which ones have finished executing. This can be done with `ray.wait`. The function works as follows.

```
ready_ids, remaining_ids = ray.wait(object_ids, num_returns=1, timeout=None)
```

`ray.wait(object_ids, num_returns=1, timeout=None)`

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

If timeout is set, the function returns either when the requested number of IDs are ready or when the timeout is reached, whichever occurs first. If it is not set, the function simply waits until that number of objects is ready and returns that exact number of object IDs.

This method returns two lists. The first list consists of object IDs that correspond to objects that are available in the object store. The second list corresponds to the rest of the object IDs (which may or may not be ready).

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

This method will issue a warning if it's running inside an `async` context. Instead of `ray.wait(object_ids)`, you can use `await asyncio.wait(object_ids)`.

#### 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.

## 5.2.6 Object Eviction

When the object store gets full, objects will be evicted to make room for new objects. This happens in approximate LRU (least recently used) order. To avoid objects from being evicted, you can call `ray.get` and store their values instead. Numpy array objects cannot be evicted while they are mapped in any Python process. You can also configure `memory limits` to control object store usage by actors.

---

**Note:** Objects created with `ray.put` are pinned in memory while a Python reference to the object ID returned by the `put` exists. This only applies to the specific ID returned by `put`, not IDs in general or copies of that IDs.

---

## 5.2.7 Remote Classes (Actors)

Actors extend the Ray API from functions (tasks) to classes. The `ray.remote` decorator indicates that instances of the Counter class will be actors. An actor is essentially a stateful worker. Each actor runs in its own Python process.

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

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

To create a couple actors, we can instantiate this class as follows:

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

When an actor is instantiated, the following events happen.

1. A worker Python process is started on a node of the cluster.

2. A Counter object is instantiated on that worker.

You can specify resource requirements in Actors too (see the [Actors](#) section for more details.)

```
@ray.remote(num_cpus=2, num_gpus=0.5)
class Actor(object):
    pass
```

We can interact with the actor by calling its methods with the `.remote` operator. We can then call `ray.get` on the object ID to retrieve the actual value.

```
obj_id = a1.increment.remote()
ray.get(obj_id) == 1
```

Methods called on different actors can execute in parallel, and methods called on the same actor are executed serially in the order that they are called. Methods on the same actor will 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]
```

To learn more about Ray Actors, see the [Actors](#) section.

## 5.3 Using Ray

If you're brand new to Ray, we recommend starting with our [tutorials](#).

Below, you'll find information ranging from how to [start Ray](#) to advanced [usage](#). There are also detailed instructions on how to work with Ray concepts such as Actors and managing GPUs.

Finally, we've also included some content on using core Ray APIs with [Tensorflow](#) and [PyTorch](#).

### 5.3.1 Starting Ray

This page covers how to start Ray on your single machine or cluster of machines.

- *Installation*
- *Starting Ray on a single machine*
- *Using Ray on a cluster*
  - *Manual cluster setup*
- *Turning off parallelism*
- *What's next?*

## Installation

Install Ray with `pip install -U ray`. For the latest wheels (a snapshot of the master branch), you can use the instructions at [Latest Snapshots \(Nightlies\)](#).

### Starting Ray on a single machine

You can start Ray by calling `ray.init()` in your Python script. This will start the local services that Ray uses to schedule remote tasks and actors and then connect to them. Note that you must initialize Ray before any tasks or actors are called (i.e., `function.remote()` will not work until `ray.init()` is called).

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

To stop or restart Ray, use `ray.shutdown()`.

```
import ray
ray.init()
... # ray program
ray.shutdown()
```

To check if Ray is initialized, you can call `ray.is_initialized()`:

```
import ray
ray.init()
assert ray.is_initialized() == True

ray.shutdown()
assert ray.is_initialized() == False
```

See the [Configuration](#) documentation for the various ways to configure Ray.

## Using Ray on a cluster

There are two steps needed to use Ray in a distributed setting:

1. You must first start the Ray cluster.
2. You need to add the `address` parameter to `ray.init` (like `ray.init(address=...)`). This causes Ray to connect to the existing cluster instead of starting a new one on the local node.

If you have a Ray cluster specification ([Automatic Cluster Setup](#)), you can launch a multi-node cluster with Ray initialized on each node with `ray up`. **From your local machine/laptop:**

```
ray up cluster.yaml
```

You can monitor the Ray cluster status with `ray monitor cluster.yaml` and ssh into the head node with `ray attach cluster.yaml`.

Your Python script **only** needs to execute on one machine in the cluster (usually the head node). To connect your program to the Ray cluster, add the following to your Python script:

```
ray.init(address="auto")
```

---

**Note:** Without `ray.init(address...)`, your Ray program will only be parallelized across a single machine!

---

## Manual cluster setup

You can also use the manual cluster setup ([Manual Cluster Setup](#)) by running initialization commands on each node.

### On the head node:

```
# 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 <address> with the value printed by the command on the head node (it should look something like 123.45.67.89:6379).

```
$ ray start --address=<address>
```

## Turning off parallelism

**Caution:** This feature is maintained solely to help with debugging, so it's possible you may encounter some issues. If you do, please [file an issue](#).

By default, Ray will parallelize its workload. However, if you need to debug your Ray program, it may be easier to do everything on a single process. You can force all Ray functions to occur on a single process with `local_mode` by calling the following:

```
ray.init(local_mode=True)
```

Note that some behavior such as setting global process variables may not work as expected.

## What's next?

Check out our [Deployment](#) section for more information on deploying Ray in different settings, including Kubernetes, YARN, and SLURM.

### 5.3.2 Using 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.

#### Creating an actor

You can convert a standard Python class into a Ray actor class as follows:

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

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

Note that the above is equivalent to the following:

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

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

Counter = ray.remote(Counter)
```

When the above actor is instantiated, the following events happen.

1. A node in the cluster is chosen and a worker process is created 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.

## Actor Methods

Any method of the actor can return multiple object IDs with the `ray.method` decorator:

```
@ray.remote
class Foo(object):

    @ray.method(num_return_vals=2)
    def bar(self):
        return 1, 2

f = Foo.remote()

obj_id1, obj_id2 = f.bar.remote()
assert ray.get(obj_id1) == 1
assert ray.get(obj_id2) == 2
```

## Resources with Actors

You can specify that an actor requires CPUs or GPUs in the decorator. While Ray has built-in support for CPUs and GPUs, Ray can also handle custom resources.

When using GPUs, Ray will automatically set the environment variable `CUDA_VISIBLE_DEVICES` for the actor after instantiated. The actor 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]`.

```
@ray.remote(num_cpus=2, num_gpus=1)
class GPUActor(object):
    pass
```

When an `GPUActor` 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.

If you want to use custom resources, make sure your cluster is configured to have these resources (see [configuration instructions](#)):

---

**Important:**

- If you specify resource requirements in an actor class's remote decorator, then the actor will acquire those resources for its entire lifetime (if you do not specify CPU resources, the default is 1), even if it is not executing any methods. The actor will not acquire any additional resources when executing methods.
  - If you do not specify any resource requirements in the actor class's remote decorator, then by default, the actor will not acquire any resources for its lifetime, but every time it executes a method, it will need to acquire 1 CPU resource.
- 

```
@ray.remote(resources={'Resource2': 1})
class GPUActor(object):
    pass
```

If you need to instantiate many copies of the same actor with varying resource requirements, you can do so as follows.

```
@ray.remote(num_cpus=4)
class Counter(object):
    def __init__(self):
        self.value = 0

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

a1 = Counter.options(num_cpus=1, resources={"Custom1": 1}).remote()
a2 = Counter.options(num_cpus=2, resources={"Custom2": 1}).remote()
a3 = Counter.options(num_cpus=3, resources={"Custom3": 1}).remote()
```

Note that to create these actors successfully, Ray will need to be started with sufficient CPU resources and the relevant custom resources.

## Terminating Actors

Actor processes will be terminated automatically when the initial actor handle goes out of scope in Python. If we create an actor with `actor_handle = Counter.remote()`, then when `actor_handle` goes out of scope and is destructed, the actor process will be terminated. Note that this only applies to the original actor handle created for the actor and not to subsequent actor handles created by passing the actor handle to other tasks.

If necessary, you can manually terminate an actor by calling `ray.actor.exit_actor()` from within one of the actor methods. This will kill the actor process and release resources associated/assigned to the actor. This approach should generally not be necessary as actors are automatically garbage collected. The `ObjectID` resulting from the task can be waited on to wait for the actor to exit (calling `ray.get()` on it will raise a `RayActorError`). Note that this method of termination will wait until any previously submitted tasks finish executing. If you want to terminate an actor immediately, you can call `ray.kill(actor_handle)`. This will cause the actor to exit immediately and any pending tasks to fail. Any exit handlers installed in the actor using `atexit` will be called.

## Passing Around Actor Handles

Actor handles can be passed into other tasks. To illustrate this with a simple example, consider a simple actor definition.

```
@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.

```
import time

@ray.remote
def f(counter):
    for _ in range(1000):
        time.sleep(0.1)
        counter.inc.remote()
```

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

```
counter = Counter.remote()

# Start some tasks that use the actor.
[f.remote(counter) for _ in range(3)]

# Print the counter value.
for _ in range(10):
    time.sleep(1)
    print(ray.get(counter.get_counter.remote()))
```

## Actor Pool

The `ray.util` module contains a utility class, `ActorPool`. This class is similar to `multiprocessing.Pool` and lets you schedule Ray tasks over a fixed pool of actors.

```
from ray.util import ActorPool

a1, a2 = Actor.remote(), Actor.remote()
pool = ActorPool([a1, a2])
print(pool.map(lambda a, v: a.double.remote(v), [1, 2, 3, 4]))
# [2, 4, 6, 8]
```

See the [package reference](#) for more information.

### 5.3.3 AsyncIO / Concurrency for Actors

Since Python 3.5, it is possible to write concurrent code using the `async/await` syntax. Ray natively integrates with asyncio. You can use ray alongside with popular async frameworks like aiohttp, aioredis, etc.

You can try it about by running the following snippet in ipython or a shell that supports top level await:

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

@ray.remote
class AsyncActor:
    # multiple invocation of this method can be running in
    # the event loop at the same time
    async def run_concurrent(self):
        print("started")
        await asyncio.sleep(2) # concurrent workload here
        print("finished")

actor = AsyncActor.remote()

# regular ray.get
ray.get([actor.run_concurrent.remote() for _ in range(4)])

# async ray.get
await actor.run_concurrent.remote()
```

### ObjectIDs as asyncio.Futures

ObjectIDs can be translated to asyncio.Future. This feature make it possible to `await` on ray futures in existing concurrent applications.

Instead of:

```
@ray.remote
def some_task():
    return 1

ray.get(some_task.remote())
ray.wait([some_task.remote()])
```

you can do:

```
@ray.remote
def some_task():
    return 1

await some_task.remote()
await asyncio.wait([some_task.remote()])
```

Please refer to [asyncio doc](#) for more `asyncio` patterns including timeouts and `asyncio.gather`.

## Async Actor

Ray also supports concurrent multitasking by executing many actor tasks at once. To do so, you can define an actor with async methods:

```
import asyncio

@ray.remote
class AsyncActor:
    async def run_task(self):
        print("started")
        await asyncio.sleep(1) # Network, I/O task here
        print("ended")

actor = AsyncActor.remote()
# All 50 tasks should start at once. After 1 second they should all finish.
# They should finish at the same time
ray.get([actor.run_task.remote() for _ in range(50)])
```

Under the hood, Ray runs all of the methods inside a single python event loop. Please note that running blocking `ray.get` or `ray.wait` inside async actor method is not allowed, because `ray.get` will block the execution of the event loop.

You can limit the number of concurrent task running at once using the `max_concurrency` flag. By default, 1000 tasks can be running concurrently.

```
import asyncio

@ray.remote
class AsyncActor:
    async def run_task(self):
        print("started")
        await asyncio.sleep(1) # Network, I/O task here
        print("ended")

actor = AsyncActor.options(max_concurrency=10).remote()

# Only 10 tasks will be running concurrently. Once 10 finish, the next 10 should run.
ray.get([actor.run_task.remote() for _ in range(50)])
```

## Known Issues

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

### 5.3.4 GPU Support

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

## Starting Ray with GPUs

Ray will automatically detect the number of GPUs available on a machine. If you need to, you can override this by specifying `ray.init(num_gpus=N)` or `ray start --num-gpus=N`.

**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.

## Using Remote Functions with GPUs

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

```
import os

@ray.remote(num_gpus=1)
def use_gpu():
    print("ray.get_gpu_ids(): {}".format(ray.get_gpu_ids()))
    print("CUDA_VISIBLE_DEVICES: {}".format(os.environ["CUDA_VISIBLE_DEVICES"]))
```

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. Typically, it is not necessary to call `ray.get_gpu_ids()` because Ray will automatically set the `CUDA_VISIBLE_DEVICES` environment variable.

**Note:** The function `use_gpu` 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 tensorflow as tf

@ray.remote(num_gpus=1)
def use_gpu():
    # 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 `use_gpu` 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. However, Ray does automatically set the `CUDA_VISIBLE_DEVICES` environment variable, which will restrict the GPUs used by most deep learning frameworks.

## Fractional GPUs

If you want two tasks to share the same GPU, then the tasks can each request half (or some other fraction) of a GPU.

```
import ray
import time

ray.init(num_cpus=4, num_gpus=1)

@ray.remote(num_gpus=0.25)
def f():
    time.sleep(1)
```

(continues on next page)

(continued from previous page)

```
# The four tasks created here can execute concurrently.  
ray.get([f.remote() for _ in range(4)])
```

It is the developer's responsibility to make sure that the individual tasks don't use more than their share of the GPU memory. TensorFlow can be configured to limit its memory usage.

## 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. If sufficient GPU resources are not available, then the actor will not be created.

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):  
        # The call to tf.Session() will restrict TensorFlow to use the GPUs  
        # specified in the CUDA_VISIBLE_DEVICES environment variable.  
        self.sess = tf.Session()
```

## Workers not Releasing GPU Resources

**Note:** Currently, when a worker executes a task that uses a GPU (e.g., through TensorFlow), the task may allocate memory on the GPU and may not release it when the task finishes executing. This can lead to problems the next time a task tries to use the same GPU. You can address this by setting `max_calls=1` in the remote decorator so that the worker automatically exits after executing the task (thereby releasing the GPU resources).

```
import tensorflow as tf  
  
@ray.remote(num_gpus=1, max_calls=1)  
def leak_gpus():  
    # This task will allocate memory on the GPU and then never release it, so  
    # we include the max_calls argument to kill the worker and release the  
    # resources.  
    sess = tf.Session()
```

### 5.3.5 Serialization

Since Ray processes do not share memory space, data transferred between workers and nodes will need to **serialized** and **deserialized**. Ray uses the [Plasma object store](#) to efficiently transfer objects across different processes and different nodes. Numpy arrays in the object store are shared between workers on the same node (zero-copy deserialization).

#### Plasma Object Store

Plasma is an in-memory object store that is being developed as part of [Apache Arrow](#). Ray uses Plasma to efficiently transfer objects across different processes and different nodes. All objects in Plasma object store are **immutable** and held in shared memory. This is so that they can be accessed efficiently by many workers on the same node.

Each node has its own object store. When data is put into the object store, it does not get automatically broadcasted to other nodes. Data remains local to the writer until requested by another task or actor on another node.

#### Overview

Objects that are serialized for transfer among Ray processes go through three stages:

**1. Serialize directly:** Below is the set of Python objects that Ray can serialize using `memcpy`:

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.

**2. ``\_\_dict\_\_`` serialization:** If a direct usage is not possible, Ray will recursively extract the object's `__dict__` and serialize that directly. This behavior is not correct in all cases.

**3. Cloudpickle:** Ray falls back to `cloudpickle` as a final attempt for serialization. This may be slow.

#### Numpy Arrays

Ray optimizes for numpy arrays by using the [Apache Arrow](#) data format. The numpy array is stored as a read-only object, and all Ray workers on the same node can read the numpy array in the object store without copying (zero-copy reads). Each numpy array object in the worker process holds a pointer to the relevant array held in shared memory. Any writes to the read-only object will require the user to first copy it into the local process memory.

---

**Tip:** You can often avoid serialization issues by using only native types (e.g., numpy arrays or lists/dicts of numpy arrays and other primitive types), or by using Actors hold objects that cannot be serialized.

---

#### Serialization notes and limitations

- Ray currently handles 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.

```

11 = [0]
12 = [11, 11]
13 = ray.get(ray.put(12))

assert 12[0] is 12[1]
assert not 13[0] is 13[1]

```

- 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 or Python collections of numpy arrays for maximum performance.

## Last resort: Custom Serialization

If none of these options work, you can try registering a custom serializer.

```
ray.register_custom_serializer(cls, serializer, deserializer, use_pickle=False, use_dict=False,
                               class_id=None)
```

Registers custom functions for efficient object serialization.

The serializer and deserializer are used when transferring objects of *cls* across processes and nodes. This can be significantly faster than the Ray default fallbacks. Wraps *register\_custom\_serializer* underneath.

### Parameters

- **cls** (*type*) – The class that ray should use this custom serializer for.
- **serializer** – The custom serializer that takes in a *cls* instance and outputs a serialized representation. *use\_pickle* and *use\_dict* must be False if provided.
- **deserializer** – The custom deserializer that takes in a serialized representation of the *cls* and outputs a *cls* instance. *use\_pickle* and *use\_dict* must be False if provided.
- **use\_pickle** – Deprecated.
- **use\_dict** – Deprecated.
- **class\_id** (*str*) – Unique ID of the class. Autogenerated if None.

Below is an example of using *ray.register\_custom\_serializer*:

```
import ray

ray.init()

class Foo(object):
    def __init__(self, value):
        self.value = value

def custom_serializer(obj):
    return obj.value

def custom_deserializer(value):
    object = Foo()
    object.value = value
    return object
```

(continues on next page)

(continued from previous page)

```
ray.register_custom_serializer(
    Foo, serializer=custom_serializer, deserializer=custom_deserializer)

object_id = ray.put(Foo(100))
assert ray.get(object_id).value == 100
```

If you find cases where Ray serialization doesn't work or does something unexpected, please [let us know](#) so we can fix it.

## Advanced: Huge Pages

On Linux, it is possible to increase the write throughput of the Plasma object store by using huge pages. See the [Configuration page](#) for information on how to use huge pages in Ray.

### 5.3.6 Memory Management

This page describes how memory management works in Ray and how you can set memory quotas to ensure memory-intensive applications run predictably and reliably.

#### ObjectID Reference Counting

Ray implements distributed reference counting so that any `ObjectID` in scope in the cluster is pinned in the object store. This includes local python references, arguments to pending tasks, and IDs serialized inside of other objects.

#### Frequently Asked Questions (FAQ)

##### **My application failed with `ObjectStoreFullError`. What happened?**

Ensure that you're removing `ObjectID` references when they're no longer needed. See [Debugging using ‘ray memory’](#) for information on how to identify what objects are in scope in your application.

This exception is raised when the object store on a node was full of pinned objects when the application tried to create a new object (either by calling `ray.put()` or returning an object from a task). If you're sure that the configured object store size was large enough for your application to run, ensure that you're removing `ObjectID` references when they're no longer in use so their objects can be evicted from the object store.

##### **I'm running Ray inside IPython or a Jupyter Notebook and there are `ObjectID` references causing problems even though I'm not storing them anywhere.**

Try [Enabling LRU Fallback](#), which will cause unused objects referenced by IPython to be LRU evicted when the object store is full instead of erroring.

IPython stores the output of every cell in a local Python variable indefinitely. This causes Ray to pin the objects even though your application may not actually be using them.

##### **My application used to run on previous versions of Ray but now I'm getting `ObjectStoreFullError`.**

Either modify your application to remove `ObjectID` references when they're no longer needed or try [Enabling LRU Fallback](#) to revert to the old behavior.

In previous versions of Ray, there was no reference counting and instead objects in the object store were LRU evicted once the object store ran out of space. Some applications (e.g., applications that keep references to all objects ever created) may have worked with LRU eviction but do not with reference counting.

## Debugging using ‘ray memory’

The `ray memory` command can be used to help track down what `ObjectID` references are in scope and may be causing an `ObjectStoreFullError`.

Running `ray memory` from the command line while a Ray application is running will give you a dump of all of the `ObjectID` references that are currently held by the driver, actors, and tasks in the cluster.

Object ID	Reference Type	Object Size	Reference
<code>Object ID</code>			
<code>Creation Site</code>			
<code>; worker pid=18301</code>			
<code>45b95b1c8bd3a9c4fffffffff010000c801000000</code>	<code>LOCAL_REFERENCE</code>		<code>? (task</code>
<code>  ↳(deserialize task arg) __main__..f</code>			<code>)</code>
<code>; driver pid=18281</code>			
<code>f66d17bae2b0e765fffffffff010000c801000000</code>	<code>LOCAL_REFERENCE</code>		<code>? (task</code>
<code>  ↳call) test.py:&lt;module&gt;:12</code>			<code>)</code>
<code>45b95b1c8bd3a9c4fffffffff010000c801000000</code>	<code>USED_BY_PENDING_TASK</code>		<code>? (task</code>
<code>  ↳call) test.py:&lt;module&gt;:10</code>			<code>)</code>
<code>ef0a6c221819881cfffffff010000c801000000</code>	<code>LOCAL_REFERENCE</code>		<code>? (task</code>
<code>  ↳call) test.py:&lt;module&gt;:11</code>			<code>)</code>
<code>fffffffffffffffffffff0100008801000000</code>	<code>LOCAL_REFERENCE</code>	77	<code>(put</code>
<code>  ↳object) test.py:&lt;module&gt;:9</code>			<code>)</code>
<code>Object ID</code>			
<code>Creation Site</code>			

Each entry in this output corresponds to an `ObjectID` that’s currently pinning an object in the object store along with where the reference is (in the driver, in a worker, etc.), what type of reference it is (see below for details on the types of references), the size of the object in bytes, and where in the application the reference was created.

There are five types of references that can keep an object pinned:

### 1. Local `ObjectID` references

```
@ray.remote
def f(arg):
    return arg

a = ray.put(None)
b = f.remote(None)
```

In this example, we create references to two objects: one that is `ray.put()` in the object store and another that’s the return value from `f.remote()`.

Object ID	Reference Type	Object Size	Reference
<code>Object ID</code>			
<code>Creation Site</code>			
<code>; driver pid=18867</code>			
<code>fffffffffffff0100008801000000</code>	<code>LOCAL_REFERENCE</code>	77	<code>(put</code>
<code>  ↳object) ../test.py:&lt;module&gt;:9</code>			<code>)</code>
<code>45b95b1c8bd3a9c4fffffffff010000c801000000</code>	<code>LOCAL_REFERENCE</code>		<code>? (task</code>
<code>  ↳call) ../test.py:&lt;module&gt;:10</code>			<code>)</code>
<code>Object ID</code>			
<code>Creation Site</code>			

In the output from `ray memory`, we can see that each of these is marked as a `LOCAL_REFERENCE` in the driver process, but the annotation in the “Reference Creation Site” indicates that the first was created as a “put object” and the second from a “task call.”

## 2. Objects pinned in memory

```
import numpy as np

a = ray.put(np.zeros(1))
b = ray.get(a)
del a
```

In this example, we create a numpy array and then store it in the object store. Then, we fetch the same numpy array from the object store and delete its `ObjectID`. In this case, the object is still pinned in the object store because the deserialized copy (stored in `b`) points directly to the memory in the object store.

Object ID ↳ Creation Site	Reference Type	Object Size	Reference ↳
; driver pid=25090 ffffffffffffffffff0100008801000000	PINNED_IN_MEMORY	229	test.py:
↳ <module>:7			

The output from `ray memory` displays this as the object being `PINNED_IN_MEMORY`. If we `del b`, the reference can be freed.

## 3. Pending task references

```
@ray.remote
def f(arg):
    while True:
        pass

a = ray.put(None)
b = f.remote(a)
```

In this example, we first create an object via `ray.put()` and then submit a task that depends on the object.

Object ID ↳ Creation Site	Reference Type	Object Size	Reference ↳
; worker pid=18971 ffffffffffffffffff0100008801000000	PINNED_IN_MEMORY	77	↳
↳ (deserialize task arg) __main__..f			
; driver pid=18958 ffffffffffffffffff0100008801000000	USED_BY_PENDING_TASK	77	(put_
↳ object) ../test.py:<module>:9			
45b95b1c8bd3a9c4fffffff010000c801000000	LOCAL_REFERENCE	?	(task_
↳ call) ../test.py:<module>:10			

While the task is running, we see that `ray memory` shows both a `LOCAL_REFERENCE` and a

USED\_BY\_PENDING\_TASK reference for the object in the driver process. The worker process also holds a reference to the object because it is PINNED\_IN\_MEMORY, because the Python arg is directly referencing the memory in the plasma, so it can't be evicted.

#### 4. Serialized ObjectId references

```
@ray.remote
def f(arg):
    while True:
        pass

a = ray.put(None)
b = f.remote([a])
```

In this example, we again create an object via `ray.put()`, but then pass it to a task wrapped in another object (in this case, a list).

Object ID ↳ Creation Site	Reference Type	Object Size	Reference URL
; worker pid=19002 ffffffffffffffffff0100008801000000	LOCAL_REFERENCE	77	↳ (put)
↳ (deserialize task arg) __main__..f			
; driver pid=18989 ffffffffffffffffff0100008801000000	USED_BY_PENDING_TASK	77	(put)
↳ object) ../test.py:<module>:9			
45b95b1c8bd3a9c4fffffff010000c801000000	LOCAL_REFERENCE	?	(task)
↳ call) ../test.py:<module>:10			

Now, both the driver and the worker process running the task hold a LOCAL\_REFERENCE to the object in addition to it being USED\_BY\_PENDING\_TASK on the driver. If this was an actor task, the actor could even hold a LOCAL\_REFERENCE after the task completes by storing the ObjectID in a member variable.

#### 5. Captured ObjectId references

```
a = ray.put(None)
b = ray.put([a])
```

In this example, we first create an object via `ray.put()`, then capture its ObjectID inside of another `ray.put()` object, and delete the first ObjectID. In this case, both objects are still pinned.

Object ID ↳ Creation Site	Reference Type	Object Size	Reference URL
; driver pid=19047 ffffffffffffffffff0100008802000000	LOCAL_REFERENCE	1551	(put)
↳ object) ../test.py:<module>:10			
ffffffffffffffffff0100008801000000	CAPTURED_IN_OBJECT	77	(put)
↳ object) ../test.py:<module>:9			

In the output of `ray memory`, we see that the second object displays as a normal `LOCAL_REFERENCE`, but the first object is listed as `CAPTURED_IN_OBJECT`.

## Enabling LRU Fallback

By default, Ray will raise an exception if the object store is full of pinned objects when an application tries to create a new object. However, in some cases applications might keep references to objects much longer than they actually use them, so simply LRU evicting objects from the object store when it's full can prevent the application from failing.

Please note that relying on this is **not recommended** - instead, if possible you should try to remove references as they're no longer needed in your application to free space in the object store.

To enable LRU eviction when the object store is full, initialize ray with the `lru_evict` option set:

```
ray.init(lru_evict=True)
```

```
ray start --lru-evict
```

## Memory Quotas

You can set memory quotas to ensure your application runs predictably on any Ray cluster configuration. If you're not sure, you can start with a conservative default configuration like the following and see if any limits are hit.

For Ray initialization on a single node, consider setting the following fields:

```
ray.init(
    memory=2000 * 1024 * 1024,
    object_store_memory=200 * 1024 * 1024,
    driver_object_store_memory=100 * 1024 * 1024)
```

For Ray usage on a cluster, consider setting the following fields on both the command line and in your Python script:

---

**Tip:** `200 * 1024 * 1024` bytes is 200 MiB. Use double parentheses to evaluate math in Bash: `$((200 * 1024 * 1024))`.

---

```
# On the head node
ray start --head --redis-port=6379 \
    --object-store-memory=$((200 * 1024 * 1024)) \
    --memory=$((200 * 1024 * 1024)) \
    --num-cpus=1

# On the worker node
ray start --object-store-memory=$((200 * 1024 * 1024)) \
    --memory=$((200 * 1024 * 1024)) \
    --num-cpus=1 \
    --address=$RAY_HEAD_ADDRESS:6379
```

```
# In your Python script connecting to Ray:
ray.init(
    address="auto", # or "<hostname>:<port>" if not using the default port
    driver_object_store_memory=100 * 1024 * 1024
)
```

For any custom remote method or actor, you can set requirements as follows:

```
@ray.remote(  
    memory=2000 * 1024 * 1024,  
)
```

## Concept Overview

There are several ways that Ray applications use memory:

### Ray system memory: this is memory used internally by Ray

- **Redis**: memory used for storing task lineage and object metadata. When Redis becomes full, lineage will start to be LRU evicted, which makes the corresponding objects ineligible for reconstruction on failure.
- **Raylet**: memory used by the C++ raylet process running on each node. This cannot be controlled, but is usually quite small.

### Application memory: this is memory used by your application

- **Worker heap**: memory used by your application (e.g., in Python code or TensorFlow), best measured as the *resident set size (RSS)* of your application minus its *shared memory usage (SHR)* in commands such as `top`. The reason you need to subtract *SHR* is that object store shared memory is reported by the OS as shared with each worker. Not subtracting *SHR* will result in double counting memory usage.
- **Object store memory**: memory used when your application creates objects in the objects store via `ray.put` and when returning values from remote functions. Objects are LRU evicted when the store is full, prioritizing objects that are no longer in scope on the driver or any worker. There is an object store server running on each node.
- **Object store shared memory**: memory used when your application reads objects via `ray.get`. Note that if an object is already present on the node, this does not cause additional allocations. This allows large objects to be efficiently shared among many actors and tasks.

By default, Ray will cap the memory used by Redis at `min(30% of node memory, 10GiB)`, and object store at `min(10% of node memory, 20GiB)`, leaving half of the remaining memory on the node available for use by worker heap. You can also manually configure this by setting `redis_max_memory=<bytes>` and `object_store_memory=<bytes>` on Ray init.

It is important to note that these default Redis and object store limits do not address the following issues:

- Actor or task heap usage exceeding the remaining available memory on a node.
- Heavy use of the object store by certain actors or tasks causing objects required by other tasks to be prematurely evicted.

To avoid these potential sources of instability, you can set *memory quotas* to reserve memory for individual actors and tasks.

## Heap memory quota

When Ray starts, it queries the available memory on a node / container not reserved for Redis and the object store or being used by other applications. This is considered “available memory” that actors and tasks can request memory out of. You can also set `memory=<bytes>` on Ray init to tell Ray explicitly how much memory is available.

---

**Important:** Setting available memory for the node does NOT impose any limits on memory usage unless you specify memory resource requirements in decorators. By default, tasks and actors request no memory (and hence have no limit).

---

To tell the Ray scheduler a task or actor requires a certain amount of available memory to run, set the `memory` argument. The Ray scheduler will then reserve the specified amount of available memory during scheduling, similar to how it handles CPU and GPU resources:

```
# reserve 500MiB of available memory to place this task
@ray.remote(memory=500 * 1024 * 1024)
def some_function(x):
    pass

# reserve 2.5GiB of available memory to place this actor
@ray.remote(memory=2500 * 1024 * 1024)
class SomeActor(object):
    def __init__(self, a, b):
        pass
```

In the above example, the memory quota is specified statically by the decorator, but you can also set them dynamically at runtime using `.options()` as follows:

```
# override the memory quota to 100MiB when submitting the task
some_function.options(memory=100 * 1024 * 1024).remote(x=1)

# override the memory quota to 1GiB when creating the actor
SomeActor.options(memory=1000 * 1024 * 1024).remote(a=1, b=2)
```

**Enforcement:** If an actor exceeds its memory quota, calls to it will throw `RayOutOfMemoryError` and it may be killed. Memory quota is currently enforced on a best-effort basis for actors only (but quota is taken into account during scheduling in all cases).

## Object store memory quota

Use `@ray.remote(object_store_memory=<bytes>)` to cap the amount of memory an actor can use for `ray.put` and method call returns. This gives the actor its own LRU queue within the object store of the given size, both protecting its objects from eviction by other actors and preventing it from using more than the specified quota. This quota protects objects from unfair eviction when certain actors are producing objects at a much higher rate than others.

Ray takes this resource into account during scheduling, with the caveat that a node will always reserve ~30% of its object store for global shared use.

For the driver, you can set its object store memory quota with `driver_object_store_memory`. Setting object store quota is not supported for tasks.

## Object store shared memory

Object store memory is also used to map objects returned by `ray.get` calls in shared memory. While an object is mapped in this way (i.e., there is a Python reference to the object), it is pinned and cannot be evicted from the object store. However, ray does not provide quota management for this kind of shared memory usage.

### Questions or Issues?

If you have a question or issue that wasn't covered by this page, please get in touch via one of the following channels:

1. [ray-dev@googlegroups.com](mailto: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.

## 5.3.7 Debugging and Profiling

### Observing Ray Work

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. See the [Troubleshooting page](#) for more details.

### Visualizing Tasks in the Ray Timeline

The most important tool is the timeline visualization tool. To visualize tasks in the Ray timeline, you can dump the timeline as a JSON file by running `ray timeline` from the command line or by using the following command.

```
ray.timeline(filename="/tmp/timeline.json")
```

Then open `chrome://tracing` in the Chrome web browser, and load `timeline.json`.

### 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
    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())
```

(continues on next page)

(continued from previous page)

```
# 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
    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)
...
```

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.

## 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.

## 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. 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.

For many - but not all - libraries, 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. The most common exception is PyTorch, which will appear to be using all the cores despite needing `torch.set_num_threads(1)` to be called to avoid contention.

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.

## 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
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`.

### 5.3.8 Advanced Usage

This page will cover some more advanced examples of using Ray's flexible programming model.

#### Dynamic Remote Parameters

You can dynamically adjust resource requirements or return values of `ray.remote` during execution with `.` options.

For example, here we instantiate many copies of the same actor with varying resource requirements. Note that to create these actors successfully, Ray will need to be started with sufficient CPU resources and the relevant custom resources:

```
@ray.remote(num_cpus=4)
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
```

(continues on next page)

(continued from previous page)

```

    self.value += 1
    return self.value

a1 = Counter.options(num_cpus=1, resources={"Custom1": 1}).remote()
a2 = Counter.options(num_cpus=2, resources={"Custom2": 1}).remote()
a3 = Counter.options(num_cpus=3, resources={"Custom3": 1}).remote()

```

You can specify different resource requirements for tasks (but not for actor methods):

```

@ray.remote
def g():
    return ray.get_gpu_ids()

object_gpu_ids = g.remote()
assert ray.get(object_gpu_ids) == [0]

dynamic_object_gpu_ids = g.options(num_cpus=1, num_gpus=1).remote()
assert ray.get(dynamic_object_gpu_ids) == [0]

```

And vary the number of return values for tasks (and actor methods too):

```

@ray.remote
def f(n):
    return list(range(n))

id1, id2 = f.options(num_return_vals=2).remote(2)
assert ray.get(id1) == 0
assert ray.get(id2) == 1

```

## Dynamic Custom Resources

Ray enables explicit developer control with respect to the task and actor placement by using custom resources. Further, users are able to dynamically adjust custom resources programmatically with `ray.experimental.set_resource`. This allows the Ray application to implement virtually any scheduling policy, including task affinity, data locality, anti-affinity, load balancing, gang scheduling, and priority-based scheduling.

```

ray.init()
resource_name = "test_resource"
resource_capacity = 1.0

@ray.remote
def set_resource(resource_name, resource_capacity):
    ray.experimental.set_resource(resource_name, resource_capacity)

ray.get(set_resource.remote(resource_name, resource_capacity))

available_resources = ray.available_resources()
cluster_resources = ray.cluster_resources()

assert available_resources[resource_name] == resource_capacity
assert cluster_resources[resource_name] == resource_capacity

```

`ray.experimental.set_resource(resource_name, capacity, client_id=None)`  
Set a resource to a specified capacity.

This creates, updates or deletes a custom resource for a target clientId. If the resource already exists, it's capacity is updated to the new value. If the capacity is set to 0, the resource is deleted. If ClientID is not specified or set to None, the resource is created on the local client where the actor is running.

#### Parameters

- **resource\_name** (*str*) – Name of the resource to be created
- **capacity** (*int*) – Capacity of the new resource. Resource is deleted if capacity is 0.
- **client\_id** (*str*) – The ClientId of the node where the resource is to be set.

**Returns** None

**Raises** `ValueError` – This exception is raised when a non-negative capacity is specified.

### Nested Remote Functions

Remote functions can call other remote functions, resulting in nested tasks. For example, consider the following.

```
@ray.remote
def f():
    return 1

@ray.remote
def g():
    # Call f 4 times and return the resulting object IDs.
    return [f.remote() for _ in range(4)]

@ray.remote
def h():
    # Call f 4 times, block until those 4 tasks finish,
    # retrieve the results, and return the values.
    return ray.get([f.remote() for _ in range(4)])
```

Then calling `g` and `h` produces the following behavior.

```
>>> ray.get(g.remote())
[ObjectID(b1457ba0911ae84989aae86f89409e953dd9a80e),
 ObjectID(7c14a1d13a56d8dc01e800761a66f09201104275),
 ObjectID(99763728ffc1a2c0766a2000ebabded52514e9a6),
 ObjectID(9c2f372e1933b04b2936bb6f58161285829b9914)]

>>> ray.get(h.remote())
[1, 1, 1, 1]
```

**One limitation** is that the definition of `f` must come before the definitions of `g` and `h` because as soon as `g` is defined, it will be pickled and shipped to the workers, and so if `f` hasn't been defined yet, the definition will be incomplete.

## Circular Dependencies

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.

## Cython Code in Ray

To use Cython code in Ray, 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.

## 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](#)).

## Inspecting Cluster State

Applications written on top of Ray will often want to have some information or diagnostics about the cluster. Some common questions include:

1. How many nodes are in my autoscaling cluster?
2. What resources are currently available in my cluster, both used and total?
3. What are the objects currently in my cluster?

For this, you can use the global state API.

### Node Information

To get information about the current nodes in your cluster, you can use `ray.nodes()`:

`ray.nodes()`

Get a list of the nodes in the cluster.

**Returns** Information about the Ray clients in the cluster.

```
import ray

ray.init()

print(ray.nodes())

"""
[{'ClientID': 'a9e430719685f3862ed7ba411259d4138f8afble',
 'IsInsertion': True,
 'NodeManagerAddress': '192.168.19.108',
 'NodeManagerPort': 37428,
 'ObjectManagerPort': 43415,
 'ObjectStoreSocketName': '/tmp/ray/session_2019-07-28_17-03-53_955034_24883/sockets/
plasma_store',
 'RayletSocketName': '/tmp/ray/session_2019-07-28_17-03-53_955034_24883/sockets/
raylet',
 'Resources': {'CPU': 4.0},
 'alive': True}]
"""
```

The above information includes:

- *ClientID*: A unique identifier for the raylet.
- *alive*: Whether the node is still alive.
- *NodeManagerAddress*: PrivateIP of the node that the raylet is on.
- *Resources*: The total resource capacity on the node.

## Resource Information

To get information about the current total resource capacity of your cluster, you can use `ray.cluster_resources()`.

```
ray.cluster_resources()
```

Get the current total cluster resources.

Note that this information can grow stale as nodes are added to or removed from the cluster.

### Returns

A dictionary mapping resource name to the total quantity of that resource in the cluster.

To get information about the current available resource capacity of your cluster, you can use `ray.available_resources()`.

```
ray.available_resources()
```

Get the current available cluster resources.

This is different from `cluster_resources` in that this will return idle (available) resources rather than total resources.

Note that this information can grow stale as tasks start and finish.

### Returns

A dictionary mapping resource name to the total quantity of that resource in the cluster.

## Detached Actors

When original actor handles goes out of scope or the driver that originally created the actor exits, ray will clean up the actor by default. If you want to make sure the actor is kept alive, you can use `_remote(name="some_name", detached=True)` to keep the actor alive after the driver exits. The actor will have a globally unique name and can be accessed across different drivers.

For example, you can instantiate and register a persistent actor as follows:

```
counter = Counter.options(name="CounterActor", detached=True).remote()
```

The CounterActor will be kept alive even after the driver running above script exits. Therefore it is possible to run the following script in a different driver:

```
counter = ray.util.get_actor("CounterActor")
print(ray.get(counter.get_counter.remote()))
```

Note that just creating a named actor is allowed, this actor will be cleaned up after driver exits:

```
Counter.options(name="CounterActor").remote()
```

However, creating a detached actor without name is not allowed because there will be no way to retrieve the actor handle and the resource is leaked.

```
# Can't do this!
Counter.options(detached=True).remote()
```

### 5.3.9 Best Practices: Ray with Tensorflow

This document describes best practices for using the Ray core APIs with TensorFlow. Ray also provides higher-level utilities for working with Tensorflow, such as distributed training APIs ([training tensorflow example](#)), Tune for hyperparameter search ([Tune tensorflow example](#)), RLlib for reinforcement learning ([RLlib tensorflow example](#)).

Feel free to contribute if you think this document is missing anything.

#### Common Issues: Pickling

One common issue with TensorFlow2.0 is a pickling error like the following:

```
File "/home/***/venv/lib/python3.6/site-packages/ray/actor.py", line 322, in remote
    return self._remote(args=args, kwargs=kwargs)
File "/home/***/venv/lib/python3.6/site-packages/ray/actor.py", line 405, in _remote
    self._modified_class, self._actor_method_names)
File "/home/***/venv/lib/python3.6/site-packages/ray/function_manager.py", line 578, in
    ↪ in export_actor_class
    "class": pickle.dumps(Class),
File "/home/***/venv/lib/python3.6/site-packages/ray/cloudpickle/cloudpickle.py", line
    ↪ 1123, in dumps
    cp.dump(obj)
File "/home/***/lib/python3.6/site-packages/ray/cloudpickle/cloudpickle.py", line 482,
    ↪ in dump
    return Pickler.dump(self, obj)
File "/usr/lib/python3.6/pickle.py", line 409, in dump
    self.save(obj)
File "/usr/lib/python3.6/pickle.py", line 476, in save
    f(self, obj) # Call unbound method with explicit self
File "/usr/lib/python3.6/pickle.py", line 751, in save_tuple
    save(element)
File "/usr/lib/python3.6/pickle.py", line 808, in _batch_appends
    save(tmp[0])
File "/usr/lib/python3.6/pickle.py", line 496, in save
    rv = reduce(self.proto)
TypeError: can't pickle _LazyLoader objects
```

To resolve this, you should move all instances of `import tensorflow` into the Ray actor or function, as follows:

```
def create_model():
    import tensorflow as tf
    ...
```

This issue is caused by side-effects of importing TensorFlow and setting global state.

#### Use Actors for Parallel Models

If you are training a deep network in the distributed setting, you may need to ship your deep network between processes (or machines). However, shipping the model is not always straightforward.

---

**Tip:** Avoid sending the Tensorflow model directly. A straightforward attempt to pickle a TensorFlow graph gives mixed results. Furthermore, creating a TensorFlow graph can take tens of seconds, and so serializing a graph and recreating it in another process will be inefficient.

It is recommended to replicate 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).

```
def create_keras_model():
    from tensorflow import keras
    from tensorflow.keras import layers
    model = keras.Sequential()
    # Adds a densely-connected layer with 64 units to the model:
    model.add(layers.Dense(64, activation="relu", input_shape=(32, )))
    # Add another:
    model.add(layers.Dense(64, activation="relu"))
    # Add a softmax layer with 10 output units:
    model.add(layers.Dense(10, activation="softmax"))

    model.compile(
        optimizer=keras.optimizers.RMSprop(0.01),
        loss=keras.losses.categorical_crossentropy,
        metrics=[keras.metrics.categorical_accuracy])
    return model
```

It is strongly recommended you create actors to handle this. To do this, first initialize ray and define an Actor class:

```
import ray
import numpy as np

ray.init()

def random_one_hot_labels(shape):
    n, n_class = shape
    classes = np.random.randint(0, n_class, n)
    labels = np.zeros((n, n_class))
    labels[np.arange(n), classes] = 1
    return labels

# Use GPU wth
# @ray.remote(num_gpus=1)
@ray.remote
class Network(object):
    def __init__(self):
        self.model = create_keras_model()
        self.dataset = np.random.random((1000, 32))
        self.labels = random_one_hot_labels((1000, 10))

    def train(self):
        history = self.model.fit(self.dataset, self.labels, verbose=False)
        return history.history

    def get_weights(self):
        return self.model.get_weights()

    def set_weights(self, weights):
        # Note that for simplicity this does not handle the optimizer state.
        self.model.set_weights(weights)
```

Then, we can instantiate this actor and train it on the separate process:

```
NetworkActor = Network.remote()
result_object_id = NetworkActor.train.remote()
ray.get(result_object_id)
```

We can then use `set_weights` and `get_weights` to move the weights of the neural network around. This allows us to manipulate weights between different models running in parallel without shipping the actual TensorFlow graphs, which are much more complex Python objects.

```
NetworkActor2 = Network.remote()
NetworkActor2.train.remote()
weights = ray.get(
    [NetworkActor.get_weights.remote(),
     NetworkActor2.get_weights.remote()])

averaged_weights = [(layer1 + layer2) / 2
                     for layer1, layer2 in zip(weights[0], weights[1])]

weight_id = ray.put(averaged_weights)
[
    actor.set_weights.remote(weight_id)
    for actor in [NetworkActor, NetworkActor2]
]
ray.get([actor.train.remote() for actor in [NetworkActor, NetworkActor2]])
```

## Lower-level TF Utilities

Given a low-level TF definition:

```
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.experimental.tf_utils
variables = ray.experimental.tf_utils.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.

```
sess = tf.Session()
# 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.
```

**class** ray.experimental.tf\_utils.**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.

**Note:** This may not work with *tf.Keras*.

## Troubleshooting

Note that TensorFlowVariables uses variable names to determine what variables to set when calling `set_weights`. One common issue arises when two networks are defined in the same TensorFlow graph. In this case, TensorFlow appends an underscore and integer to the names of variables to disambiguate them. This will cause TensorFlowVariables to fail. For example, if we have a class definition Network with a TensorFlowVariables instance:

```
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.tf_utils.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():
```

(continues on next page)

(continued from previous page)

```
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.

### 5.3.10 Best Practices: Ray with PyTorch

This document describes best practices for using Ray with PyTorch. Feel free to contribute if you think this document is missing anything.

#### Downloading Data

It is very common for multiple Ray actors running PyTorch to have code that downloads the dataset for training and testing.

```
# This is running inside a Ray actor
# ...
torch.utils.data.DataLoader(
    datasets.MNIST(
        "./data", train=True, download=True,
        transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ]),
        128, shuffle=True, **kwargs)
# ...
```

This may cause different processes to simultaneously download the data and cause data corruption. One easy workaround for this is to use `Filelock`:

```
from filelock import FileLock

with FileLock("./data.lock"):
    torch.utils.data.DataLoader(
        datasets.MNIST(
            "./data", train=True, download=True,
            transform=transforms.Compose([
                transforms.ToTensor(),
                transforms.Normalize((0.1307,), (0.3081,))
            ]),
            128, shuffle=True, **kwargs)
```

## Use Actors for Parallel Models

One common use case for using Ray with PyTorch is to parallelize the training of multiple models.

---

**Tip:** Avoid sending the PyTorch model directly. Send `model.state_dict()`, as PyTorch tensors are natively supported by the Plasma Object Store.

---

Suppose we have a simple network definition (this one is modified from the PyTorch documentation).

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4 * 4 * 50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4 * 4 * 50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Along with these helper training functions:

```
from filelock import FileLock
from torchvision import datasets, transforms

def train(model, device, train_loader, optimizer):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # This break is for speeding up the tutorial.
        if batch_idx * len(data) > 1024:
            return
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
```

(continues on next page)

(continued from previous page)

```

with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)

        # sum up batch loss
        test_loss += F.nll_loss(
            output, target, reduction="sum").item()
        pred = output.argmax(
            dim=1,
            keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)
return {
    "loss": test_loss,
    "accuracy": 100. * correct / len(test_loader.dataset)
}

def dataset_creator(use_cuda):
    kwargs = {"num_workers": 1, "pin_memory": True} if use_cuda else {}
    with FileLock("./data.lock"):
        train_loader = torch.utils.data.DataLoader(
            datasets.MNIST(
                "./data",
                train=True,
                download=True,
                transform=transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize((0.1307, ), (0.3081, ))
                ]),
                128,
                shuffle=True,
                **kwargs)
        test_loader = torch.utils.data.DataLoader(
            datasets.MNIST(
                "./data",
                train=False,
                transform=transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize((0.1307, ), (0.3081, ))
                ]),
                128,
                shuffle=True,
                **kwargs)

    return train_loader, test_loader

```

Let's now define a class that captures the training process.

```

import torch.optim as optim

class Network(object):
    def __init__(self, lr=0.01, momentum=0.5):
        use_cuda = torch.cuda.is_available()

```

(continues on next page)

(continued from previous page)

```

self.device = device = torch.device("cuda" if use_cuda else "cpu")
self.train_loader, self.test_loader = dataset_creator(use_cuda)

self.model = Model().to(device)
self.optimizer = optim.SGD(
    self.model.parameters(), lr=lr, momentum=momentum)

def train(self):
    train(self.model, self.device, self.train_loader, self.optimizer)
    return test(self.model, self.device, self.test_loader)

def get_weights(self):
    return self.model.state_dict()

def set_weights(self, weights):
    self.model.load_state_dict(weights)

def save(self):
    torch.save(self.model.state_dict(), "mnist_cnn.pt")

net = Network()
net.train()

```

To train multiple models, you can convert the above class into a Ray Actor class.

```

import ray
ray.init()

RemoteNetwork = ray.remote(Network)
# Use the below instead of `ray.remote(network)` to leverage the GPU.
# RemoteNetwork = ray.remote(num_gpus=1)(Network)

```

Then, we can instantiate multiple copies of the Model, each running on different processes. If GPU is enabled, each copy runs on a different GPU. See the [GPU guide](#) for more information.

```

NetworkActor = RemoteNetwork.remote()
NetworkActor2 = RemoteNetwork.remote()

ray.get([NetworkActor.train.remote(), NetworkActor2.train.remote()])

```

We can then use `set_weights` and `get_weights` to move the weights of the neural network around. The below example averages the weights of the two networks and sends them back to update the original actors.

```

weights = ray.get(
    [NetworkActor.get_weights.remote(),
     NetworkActor2.get_weights.remote()])

from collections import OrderedDict
averaged_weights = OrderedDict([
    (k, (weights[0][k] + weights[1][k]) / 2) for k in weights[0]])

weight_id = ray.put(averaged_weights)
[
    actor.set_weights.remote(weight_id)
    for actor in [NetworkActor, NetworkActor2]
]

```

(continues on next page)

(continued from previous page)

```
]
ray.get([actor.train.remote() for actor in [NetworkActor, NetworkActor2]])
```

## 5.4 Configuring Ray

This page discusses the various way to configure Ray, both from the Python API and from the command line. Take a look at the [ray.init documentation](#) for a complete overview of the configurations.

---

**Important:** For the multi-node setting, you must first run `ray start` on the command line to start the Ray cluster services on the machine before `ray.init` in Python to connect to the cluster services. On a single machine, you can run `ray.init()` without `ray start`, which will both start the Ray cluster services and connect to them.

---

### 5.4.1 Cluster Resources

Ray by default detects available resources.

```
# This automatically detects available resources in the single machine.
ray.init()
```

If not running cluster mode, you can specify cluster resources overrides through `ray.init` as follows.

```
# If not connecting to an existing cluster, you can specify resources overrides:
ray.init(num_cpus=8, num_gpus=1)

# Specifying custom resources
ray.init(num_gpus=1, resources={'Resource1': 4, 'Resource2': 16})
```

When starting Ray from the command line, pass the `--num-cpus` and `--num-gpus` flags into `ray start`. You can also specify custom resources.

```
# To start a head node.
$ ray start --head --num-cpus=<NUM_CPUS> --num-gpus=<NUM_GPUS>

# To start a non-head node.
$ ray start --address=<address> --num-cpus=<NUM_CPUS> --num-gpus=<NUM_GPUS>

# Specifying custom resources
ray start [--head] --num-cpus=<NUM_CPUS> --resources='{"Resource1": 4, "Resource2": 16}'
```

If using the command line, connect to the Ray cluster as follow:

```
# Connect to ray. Notice if connected to existing cluster, you don't specify
# resources.
ray.init(address=<address>)
```

---

**Note:** Ray sets the environment variable `OMP_NUM_THREADS=1` by default. This is done to avoid performance degradation with many workers (issue #6998). You can override this by explicitly setting `OMP_NUM_THREADS`.

OMP\_NUM\_THREADS is commonly used in numpy, PyTorch, and Tensorflow to perform multi-threaded linear algebra. In multi-worker setting, we want one thread per worker instead of many threads per worker to avoid contention.

## 5.4.2 Logging and Debugging

Each Ray session will have a unique name. By default, the name is `session_{timestamp}_{pid}`. The format of timestamp is `%Y-%m-%d_%H-%M-%S_%f` (See [Python time format](#) for details); the pid belongs to the startup process (the process calling `ray.init()` or the Ray process executed by a shell in `ray start`).

For each session, Ray will place all its temporary files under the *session directory*. A *session directory* is a sub-directory of the *root temporary path* (`/tmp/ray` by default), so the default session directory is `/tmp/ray/{ray_session_name}`. You can sort by their names to find the latest session.

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 the given *root temporary directory*.

**Layout of logs:**

```
/tmp
└── ray
    └── session_{datetime}_{pid}
        ├── logs # for logging
        │   ├── log_monitor.err
        │   ├── log_monitor.out
        │   ├── monitor.err
        │   ├── monitor.out
        │   ├── plasma_store.err # outputs of the plasma store
        │   ├── plasma_store.out
        │   ├── raylet.err # outputs of the raylet process
        │   ├── raylet.out
        │   ├── redis-shard_0.err # outputs of redis shards
        │   ├── 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}
        └── sockets # for sockets
            ├── plasma_store
            └── raylet # this could be deleted by Ray's shutdown cleanup.
```

### 5.4.3 Redis Port Authentication

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 arbitrary code. The best fix is to run Ray instances on a secure, trusted network.

Running Ray on a secured network is not always feasible. 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.

---

**Note:** The Redis passwords provided below may not contain spaces.

---

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 to 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.

One of most common attack with Redis is port-scanning attack. Attacker scans open port with unprotected redis instance and execute arbitrary code. Ray enables a default password for redis. Even though this does not prevent brute force password cracking, the default password should alleviate most of the port-scanning attack. Furthermore, redis and other ray services are bind to localhost when the ray is started using `ray.init`.

See the [Redis security documentation](#) for more information.

### 5.4.4 Using the Object Store with Huge Pages

Plasma is a high-performance shared memory object store originally developed in Ray and now being developed in [Apache Arrow](#). See the relevant documentation.

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](#).

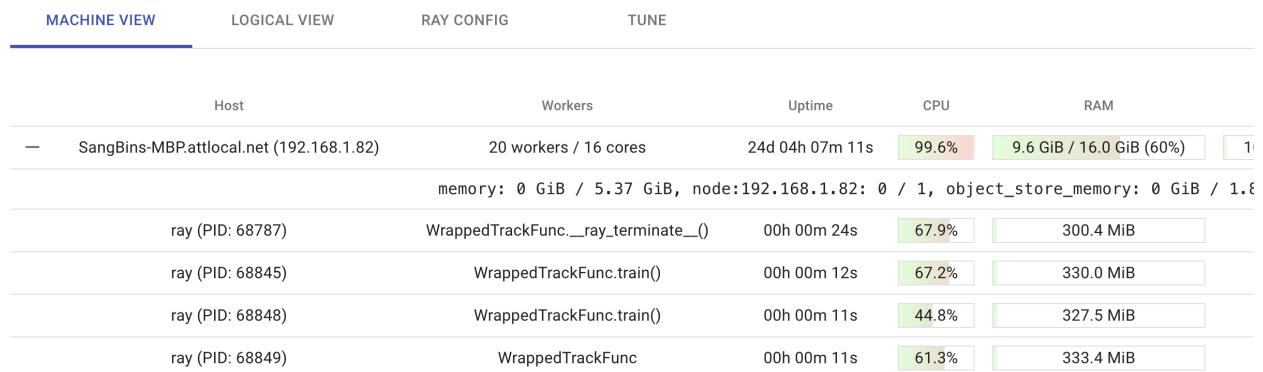
## 5.5 Ray Dashboard

Ray's built-in dashboard provides metrics, charts, and other features that help Ray users to understand Ray clusters and libraries.

Through the dashboard, you can

- View cluster metrics.
- Visualize the actor relationships and statistics.
- Kill actors and profile your Ray jobs.
- See Tune jobs and trial information.
- Detect cluster anomalies and debug them.

### Ray Dashboard



### 5.5.1 Getting Started

You can access the dashboard through its default URL, **localhost:8265**. (Note that the port number increases if the default port is not available).

The URL is printed when `ray.init()` is called.

```
INFO services.py:1093 -- View the Ray dashboard at localhost:8265
```

The dashboard is also available when using the autoscaler. Read about how to [use the dashboard with the autoscaler](#).

## 5.5.2 Views



### Machine View

The machine view shows you:

- System resource usage for each machine and worker such as RAM, CPU, disk, and network usage information.
- Logs and error messages for each machine and worker.
- Actors or tasks assigned to each worker process.

Host	Workers	Uptime	CPU	RAM
SangBins-MBP.attlocal.net (192.168.1.82)	16 workers / 16 cores	24d 04h 01m 02s	15.4%	8.7 GiB / 16.0 GiB (54%)
memory: 0 GiB / 5.47 GiB, node:192.168.1.82: 0 / 1, object_store_memory: 0				
ray (PID: 67684)	IDLE	00h 02m 22s	0.8%	208.4 MiB
ray (PID: 67685)	IDLE	00h 02m 22s	0.8%	208.4 MiB

### Logical View

The logical view shows you:

- Created and killed actors.
- Actor statistics such as actor status, number of executed tasks, pending tasks, and memory usage.
- Actor hierarchy.

```
Actor 45b95b1c0100 (Collapse) (Profile for 10s 30s 60s) Kill Actor
ActorTitle: Parent([], {}) State: 0 Pending: 0 Executed: 1 NumObjectIdsInScope: 0 NumLc
```

```
Actor 4d81fd5d0100 (Collapse) (Profile for 10s 30s 60s) Kill Actor
ActorTitle: Child([], {}) State: 0 Pending: 0 Executed: 1 NumObjectIdsInScope: 0 NumLc
```

```
Actor bbb6f3340100 (Profile for 10s 30s 60s) Kill Actor
ActorTitle: Grandchild([], {}) State: 0 Pending: 0 Executed: 1 NumObjectIdsInScope: 0
```

## Ray Config

The ray config tab shows you the current autoscaler configuration.

Ray cluster configuration:

Setting	Value
Autoscaling mode	default
Head node type	m5.4xlarge
Worker node type	m5.large
Min worker nodes	0
Initial worker nodes	0
Max worker nodes	1
Idle timeout	5 minutes

## Tune

The Tune tab shows you:

- Tune jobs and their statuses.
- Hyperparameters for each job.

Trial ID	Job ID	Start Time
29488348	train_mnist	1582569213.114
95845882	train_mnist	1582570253.683
df462194	train_mnist	1584679697.227
ee256284	train_mnist	1582572397.314

### 5.5.3 Advanced Usage

#### Killing Actors

You can kill actors when actors are hanging or not in progress.

Actor 45b95b1c0100 ([Collapse](#)) ([Profile for 10s 30s 60s](#)) [Kill Actor](#)  
 ActorTitle: Parent([], {}) State: 0 Pending: 0 Executed: 1

## Debugging a Blocked Actor

You can find hanging actors through the Logical View tab.

If creating an actor requires resources (e.g., CPUs, GPUs, or other custom resources) that are not currently available, the actor cannot be created until those resources are added to the cluster or become available. This can cause an application to hang. To alert you to this issue, infeasible tasks are shown in red in the dashboard, and pending tasks are shown in yellow.

Below is an example.

```
import ray

ray.init(num_gpus=2)

@ray.remote(num_gpus=1)
class Actor1:
    def __init__(self):
        pass

@ray.remote(num_gpus=4)
class Actor2:
    def __init__(self):
        pass

actor1_list = [Actor1.remote() for _ in range(4)]
actor2 = Actor2.remote()
```

⚠ Note: This tab is experimental.

Actor ef0a6c220100 (Profile for 10s 30s 60s) Kill Actor  
ActorTitle: Actor1([], {}) State: 0 Resources: 1 CPU, 1 GPU Pending: 0 Executed: 1 NumObjectIdsInScope: 0 NumLocalObjects: 0 UsedLocalObjectMemory: 0

Actor 45b95b1c0100 (Profile for 10s 30s 60s) Kill Actor  
ActorTitle: Actor1([], {}) State: 0 Resources: 1 CPU, 1 GPU Pending: 0 Executed: 1 NumObjectIdsInScope: 0 NumLocalObjects: 0 UsedLocalObjectMemory: 0

Actor2 is infeasible. (Infeasible actor means an actor cannot be created because Ray cluster cannot satisfy resources requirement).  
Required resources: 1 CPU, 4 GPU

Actor1 is pending until resources are available.  
Required resources: 1 CPU, 1 GPU

Actor1 is pending until resources are available.  
Required resources: 1 CPU, 1 GPU

This cluster has two GPUs, and so it only has room to create two copies of Actor1. As a result, the rest of Actor1 will be pending.

You can also see it is infeasible to create Actor2 because it requires 4 GPUs which is bigger than the total gpus available in this cluster (2 GPUs).

## Inspect Memory Usage

You can detect local memory anomalies through the Logical View tab. If NumObjectIdsInScope, NumLocalObjects, or UsedLocalObjectMemory keeps growing without bound, it can lead to out of memory errors or eviction of objectIDs that your program still wants to use.

## Profiling (Experimental)

Use profiling features when you want to find bottlenecks in your Ray applications.

Clicking one of the profiling buttons on the dashboard launches py-spy, which will profile your actor process for the given duration. Once the profiling has been done, you can click the “profiling result” button to visualize the profiling information as a flamegraph.

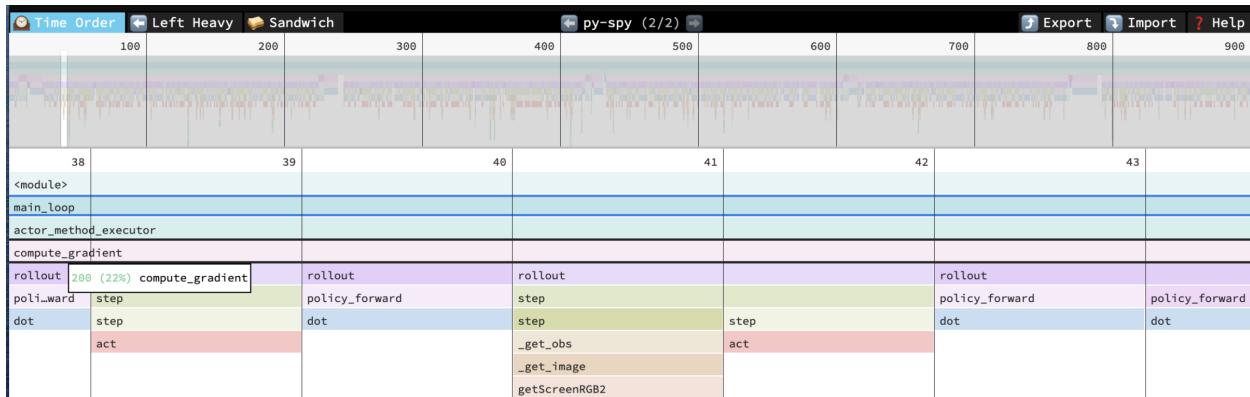
This visualization can help reveal computational bottlenecks.

---

**Note:** The profiling button currently only works when you use **passwordless sudo**. It is still experimental. Please report any issues you run into.

---

More information on how to interpret the flamegraph is available at <https://github.com/jlfwong/speedscope#usage>.



## 5.5.4 References

### Machine View

**Machine/Worker Hierarchy:** The dashboard visualizes hierarchical relationship of workers (processes) and machines (nodes). Each host consists of many workers, and you can see them by clicking the + button.

Host	Workers	Uptime	CPU
SangBins-MBP.attlocal.net (192.168.1.82)	16 workers / 16 cores	24d 04h 31m 15s	14.9%
Totals (1 host)	16 workers / 16 cores	N/A	14.9%

Last updated: 3/19

You can hide it again by clicking the - button.

Host	Workers
SangBins-MBP.attlocal.net (192.168.1.82)	16 workers / 16 cores
	memory: 0 GiB / 5.62
ray (PID: 69564)	Parent
ray (PID: 69565)	Child
ray (PID: 69566)	IDLE

### Resource Configuration

```
memory: 0 GiB / 5.62 GiB, node:192.168.1.82: 0 / 1, object_store_memory: 0 GiB / 1.9 GiB, CPU: 0 / 16
```

Resource configuration is represented as ([Resource]: [Used Resources] / [Configured Resources]). For example, when a Ray cluster is configured with 4 cores, `ray.init(num_cpus=4)`, you can see (CPU: 0 / 4).

```
memory: 0 GiB / 5.22 GiB, node:192.168.1.82: 0 / 1, object_store_memory: 0 GiB / 1.81 GiB, CPU: 0 / 4
```

When you spawn a new actor that uses 1 CPU, you can see this will be (CPU: 1/4).

Below is an example.

```
import ray

ray.init(num_cpus=4)

@ray.remote(num_cpus=1)
class A:
    pass

a = A.remote()
```

memory: 0 GiB / 5.22 GiB, node:192.168.1.82: 0 / 1, object\_store\_memory: 0 GiB / 1.81 GiB, CPU: 1 / 4

**Host:** If it is a node, it shows host information. If it is a worker, it shows a pid.

**Workers:** If it is a node, it shows a number of workers and virtual cores. Note that number of workers can exceed number of cores.

**Uptime:** Uptime of each worker and process.

**CPU:** CPU usage of each node and worker.

**RAM:** RAM usage of each node and worker.

**Disk:** Disk usage of each node and worker.

**Sent:** Network bytes sent for each node and worker.

**Received:** Network bytes received for each node and worker.

**Logs:** Logs messages at each node and worker. You can see log messages by clicking it.

**Errors:** Error messages at each node and worker. You can see error messages by clicking it.

## Logical View (Experimental)

**Actor Titles:** Name of an actor and its arguments.

**State:** State of an actor.

- 0: Alive
- 1: Reconstructing
- 2: Dead

**Pending:** A number of pending tasks for this actor.

**Executed:** A number of executed tasks for this actor.

**NumObjectIdsInScope:** Number of object IDs in scope for this actor. object IDs in scope will not be evicted unless object stores are full.

**NumLocalObjects:** Number of object IDs that are in this actor's local memory. Only big objects (>100KB) are residing in plasma object stores, and other small objects are staying in local memory.

**UsedLocalObjectMemory:** Used memory used by local objects.

**kill actor:** A button to kill an actor in a cluster. It is corresponding to `ray.kill`.

**profile for:** A button to run profiling. We currently support profiling for 10s, 30s and 60s. It requires passwordless sudo.

**Infeasible Actor Creation:** Actor creation is infeasible when an actor requires more resources than a Ray cluster can provide. This is depicted as a red colored actor.

**Pending Actor Creation:** Actor creation is pending when there are no available resources for this actor because they are already taken by other tasks and actors. This is depicted as a yellow colored actor.

**Actor Hierarchy:** The logical view renders actor information in a tree format.

To illustrate this, in the code block below, the Parent actor creates two Child actors and each Child actor creates one GrandChild actor. This relationship is visible in the dashboard *Logical View* tab.

```
import ray
ray.init()

@ray.remote
class Grandchild:
    def __init__(self):
        pass

@ray.remote
class Child:
    def __init__(self):
        self.grandchild_handle = Grandchild.remote()

@ray.remote
class Parent:
    def __init__(self):
        self.children_handles = [Child.remote() for _ in range(2)]

parent_handle = Parent.remote()
```

You can see that the dashboard shows the parent/child relationship as expected.

---

Actor 45b95b1c0100 ([Collapse](#)) (Profile for 10s 30s 60s) Kill Actor  
ActorTitle: Parent([], {}) State: 0 Pending: 0 Executed: 1 NumObjectIdsInScope: 0 NumL

---

Actor 4d81fd5d0100 ([Collapse](#)) (Profile for 10s 30s 60s) Kill Actor  
ActorTitle: Child([], {}) State: 0 Pending: 0 Executed: 1 NumObjectIdsInScope: 0 NumL

---

Actor bbb6f3340100 (Profile for 10s 30s 60s) Kill Actor  
ActorTitle: Grandchild([], {}) State: 0 Pending: 0 Executed: 1 NumObjectIdsInScope: 0

## Ray Config

If you are using the autoscaler, this Configuration defined at `cluster.yaml` is shown. See [Cluster.yaml reference](#) for more details.

### Tune (Experimental)

**Trial ID:** Trial IDs for hyperparameter tuning.

**Job ID:** Job IDs for hyperparameter tuning.

**STATUS:** Status of each trial.

**Start Time:** Start time of each trial.

**Hyperparameters:** There are many hyperparameter users specify. All of values will be visible at the dashboard.

## 5.6 Deploying Ray

How to setup your cluster and use Ray most effectively.

### 5.6.1 Automatic Cluster Setup

Ray comes with a built-in autoscaler that makes deploying a Ray cluster simple, just run `ray up` from your local machine to start or update a cluster in the cloud or on an on-premise cluster. Once the Ray cluster is running, you can manually SSH into it or use provided commands like `ray attach`, `ray rsync-up`, and `ray exec` to access it and run Ray programs.

#### Setup

This section provides instructions for configuring the autoscaler to launch a Ray cluster on AWS/Azure/GCP, an existing Kubernetes cluster, or on a private cluster of host machines.

Once you have finished configuring the autoscaler to create a cluster, see the Quickstart guide below for more details on how to get started running Ray programs on it.

#### AWS

First, install boto (`pip install boto3`) and configure your AWS credentials in `~/.aws/credentials`, as described in [the boto docs](#).

Once boto is configured to manage resources on your AWS account, you should be ready to run the autoscaler. The provided `ray/python/ray/autoscaler/aws/example-full.yaml` cluster config file will create a small cluster with an m5.large head node (on-demand) configured to autoscale up to two m5.large spot workers.

Test that it works by running the following commands from your local machine:

```
# 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

# Get a remote screen on the head node.
```

(continues on next page)

(continued from previous page)

```
$ ray attach ray/python/ray/autoscaler/aws/example-full.yaml
$ source activate tensorflow_p36
$ # Try running a Ray program with 'ray.init(address="auto")'.

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

---

**Tip:** For the AWS node configuration, you can set "ImageId: latest\_dlami" to automatically use the newest Deep Learning AMI for your region. For example, head\_node: {InstanceType: c5.xlarge, ImageId: latest\_dlami}.

---

**Note:** You may see a message like: bash: cannot set terminal process group (-1): Inappropriate ioctl for device bash: no job control in this shell This is a harmless error. If the cluster launcher fails, it is most likely due to some other factor.

---

## Azure

First, install the Azure CLI (pip install azure-cli azure-core) then login using (az login).

Set the subscription to use from the command line (az account set -s <subscription\_id>) or by modifying the provider section of the config provided e.g: ray/python/ray/autoscaler/azure/example-full.yaml

Once the Azure CLI is configured to manage resources on your Azure account, you should be ready to run the autoscaler. The provided ray/python/ray/autoscaler/azure/example-full.yaml cluster config file will create a small cluster with a Standard DS2v3 head node (on-demand) configured to autoscale up to two Standard DS2v3 spot workers. Note that you'll need to fill in your resource group and location in those templates.

Test that it works by running the following commands from your local machine:

```
# 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/azure/example-full.yaml

# Get a remote screen on the head node.
$ ray attach ray/python/ray/autoscaler/azure/example-full.yaml
# test ray setup
# enable conda environment
$ exec bash -l
$ conda activate py37_tensorflow
$ python -c 'import ray; ray.init()'
$ exit
# Tear down the cluster.
$ ray down ray/python/ray/autoscaler/azure/example-full.yaml
```

## Azure Portal

Alternatively, you can deploy a cluster using Azure portal directly. Please note that auto scaling is done using Azure VM Scale Sets and not through the Ray autoscaler. This will deploy [Azure Data Science VMs \(DSVM\)](#) for both the head node and the auto-scalable cluster managed by [Azure Virtual Machine Scale Sets](#). The head node conveniently exposes both SSH as well as JupyterLab.

Once the template is successfully deployed the deployment output page provides the ssh command to connect and the link to the JupyterHub on the head node (username/password as specified on the template input). Use the following code in a Jupyter notebook to connect to the Ray cluster.

```
import ray
ray.init(address='auto')
```

Note that on each node the `azure-init.sh` script is executed and performs the following actions:

1. Activates one of the conda environments available on DSVM
2. Installs Ray and any other user-specified dependencies
3. Sets up a systemd task (`/lib/systemd/system/ray.service`) to start Ray in head or worker mode

## GCP

First, install the Google API client (`pip install google-api-python-client`), set up your GCP credentials, and create a new GCP project.

Once the API client is configured to manage resources on your GCP account, you should be ready to run the autoscaler. 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.

Test that it works by running the following commands from your local machine:

```
# 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

# Get a remote screen on the head node.
$ ray attach ray/python/ray/autoscaler/gcp/example-full.yaml
$ source activate tensorflow_p36
$ # Try running a Ray program with 'ray.init(address="auto")'.

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

## Kubernetes

The autoscaler can also be used to start Ray clusters on an existing Kubernetes cluster. First, install the Kubernetes API client (`pip install kubernetes`), then make sure your Kubernetes credentials are set up properly to access the cluster (if a command like `kubectl get pods` succeeds, you should be good to go).

Once you have `kubectl` configured locally to access the remote cluster, you should be ready to run the autoscaler. The provided `ray/python/ray/autoscaler/kubernetes/example-full.yaml` cluster config file will create a small cluster of one pod for the head node configured to autoscale up to two worker node pods, with all pods requiring 1 CPU and 0.5GiB of memory.

Test that it works by running the following commands from your local machine:

```
# Create or update the cluster. When the command finishes, it will print
# out the command that can be used to get a remote shell into the head node.
$ ray up ray/python/ray/autoscaler/kubernetes/example-full.yaml

# List the pods running in the cluster. You shoud only see one head node
# until you start running an application, at which point worker nodes
# should be started. Don't forget to include the Ray namespace in your
# 'kubectl' commands ('ray' by default).
$ kubectl -n ray get pods

# Get a remote screen on the head node.
$ ray attach ray/python/ray/autoscaler/kubernetes/example-full.yaml
$ # Try running a Ray program with 'ray.init(address="auto")'.

# Tear down the cluster
$ ray down ray/python/ray/autoscaler/kubernetes/example-full.yaml
```

## Private Cluster

The autoscaler can also be used to run a Ray cluster on a private cluster of hosts, specified as a list of machine IP addresses to connect to. 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.

Test that it works by running the following commands from your local machine:

```
# Create or update the cluster. When the command finishes, it will print
# out the command that can be used to get a remote shell into the head node.
$ ray up ray/python/ray/autoscaler/local/example-full.yaml

# Get a remote screen on the head node.
$ ray attach ray/python/ray/autoscaler/local/example-full.yaml
$ # Try running a Ray program with 'ray.init(address="auto")'.

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

## 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`.

## 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!

## Quickstart

### Starting and updating a 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.

```
# Replace '<your_backend>' with one of: 'aws', 'gcp', 'kubernetes', or 'local'.
$ BACKEND=<your_backend>

# Create or update the cluster.
$ ray up ray/python/ray/autoscaler/$BACKEND/example-full.yaml

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

# Tear down the cluster.
$ ray down ray/python/ray/autoscaler/$BACKEND/example-full.yaml
```

### 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(address="auto")`.

```
# 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
```

(continues on next page)

(continued from previous page)

```
# 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
```

## Attaching to a running cluster

You can use `ray attach` to attach to an interactive screen session 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
```

## Port-forwarding applications

If you want to run applications on the cluster that are accessible from a web browser (e.g., Jupyter notebook), you can use the `--port-forward` option for `ray exec`. The local port opened is the same as the remote port.

Note: For Kubernetes clusters, the `port-forward` option cannot be used while executing a command. To port forward and run a command you need to call `ray exec` twice separately.

```
$ ray exec cluster.yaml --port-forward=8899 'source ~/anaconda3/bin/activate_
˓→tensorflow_p36 && jupyter notebook --port=8899'
```

## 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'
```

## Security

On cloud providers, nodes will be launched into their own security group by default, 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.

## Autoscaling

Ray clusters come with a load-based autoscaler. 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).

## Monitoring cluster status

The ray also comes with an online dashboard. The dashboard is accessible via HTTP on the head node (by default it listens on `localhost : 8265`). To access it locally, you'll need to forward the port to your local machine. You can also use the built-in `ray dashboard` to do this automatically.

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 to 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
<input type="checkbox"/>	ray-default-w...	SettingUp	i-0080148302d2504...	m5.large
<input type="checkbox"/>	ray-default-w...	SettingUp	i-04db04aeeb1f0908c	m5.large
<input checked="" type="checkbox"/>	ray-default-he..	Up-to-date	i-0ff4c501a9f365819	m5.large

## 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 (or docker container) 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](#).

## 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`

## Using Amazon EFS

To use Amazon EFS, install some utilities and mount the EFS in `setup_commands`. Note that these instructions only work if you are using the AWS Autoscaler.

---

**Note:** You need to replace the `{FileSystemId}` to your own EFS ID before using the config. You may also need to set correct `SecurityGroupIds` for the instances in the config file.

---

```
setup_commands:
  - sudo kill -9 `sudo lsof /var/lib/dpkg/lock-frontend | awk '{print $2}' | tail -n 1`;
    sudo pkill -9 apt-get;
    sudo pkill -9 dpkg;
    sudo dpkg --configure -a;
    sudo apt-get -y install binutils;
    cd $HOME;
    git clone https://github.com/aws/efs-utils;
    cd $HOME/efs-utils;
    ./build-deb.sh;
    sudo apt-get -y install ./build/amazon-efs-utils*deb;
    cd $HOME;
    mkdir efs;
    sudo mount -t efs {{FileSystemId}}:/ efs;
    sudo chmod 777 efs;
```

## 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.

```
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.

```
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.

```
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.

```
min_workers: 0
max_workers: 10
head_node:
  InstanceType: m4.large
worker_nodes:
```

(continues on next page)

(continued from previous page)

```
InstanceMarketOptions:  
    MarketType: spot  
    InstanceType: m4.16xlarge
```

**Autoscaling GPU cluster:** similar to the autoscaling CPU cluster, but with GPU worker nodes instead.

```
min_workers: 0 # NOTE: older Ray versions may need 1+ GPU workers (#2106)  
max_workers: 10  
head_node:  
    InstanceType: m4.large  
worker_nodes:  
    InstanceMarketOptions:  
        MarketType: spot  
        InstanceType: p2.xlarge
```

## Questions or Issues?

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

1. [ray-dev@googlegroups.com](mailto: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.

## 5.6.2 Manual Cluster Setup

---

**Note:** If you're using AWS, Azure 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.

### 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 `<address>` with the value printed by the command on the head node (it should look something like `123.45.67.89:6379`).

```
ray start --address=<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`. See the [Configuration](#) page for more information.

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 raylet 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(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.

### 5.6.3 Deploying on YARN

**Warning:** Running Ray on YARN is still a work in progress. If you have a suggestion for how to improve this documentation or want to request a missing feature, please feel free to create a pull request or get in touch using one of the channels in the [Questions or Issues?](#) section below.

This document assumes that you have access to a YARN cluster and will walk you through using [Skein](#) to deploy a YARN job that starts a Ray cluster and runs an example script on it.

Skein uses a declarative specification (either written as a yaml file or using the Python API) and allows users to launch jobs and scale applications without the need to write Java code.

You will first need to install Skein: `pip install skein`.

The Skein yaml file and example Ray program used here are provided in the [Ray repository](#) to get you started. Refer to the provided yaml files to be sure that you maintain important configuration options for Ray to function properly.

## Skein Configuration

A Ray job is configured to run as two *Skein services*:

1. The `ray-head` service that starts the Ray head node and then runs the application.
2. The `ray-worker` service that starts worker nodes that join the Ray cluster. You can change the number of instances in this configuration or at runtime using `skein container scale` to scale the cluster up/down.

The specification for each service consists of necessary files and commands that will be run to start the service.

```
services:  
    ray-head:  
        # There should only be one instance of the head node per cluster.  
        instances: 1  
        resources:  
            # The resources for the worker node.  
            vcores: 1  
            memory: 2048  
        files:  
            ...  
        script:  
            ...  
    ray-worker:  
        # Number of ray worker nodes to start initially.  
        # This can be scaled using 'skein container scale'.  
        instances: 3  
        resources:  
            # The resources for the worker node.  
            vcores: 1  
            memory: 2048  
        files:  
            ...  
        script:  
            ...
```

## Packaging Dependencies

Use the `files` option to specify files that will be copied into the YARN container for the application to use. See [the Skein file distribution page](#) for more information.

```
services:  
    ray-head:  
        # There should only be one instance of the head node per cluster.  
        instances: 1  
        resources:  
            # The resources for the head node.  
            vcores: 1  
            memory: 2048  
        files:  
            # ray/doc/yarn/example.py  
            example.py: example.py  
            #     # A packaged python environment using `conda-pack`. Note that Skein  
            #     # doesn't require any specific way of distributing files, but this  
            #     # is a good one for python projects. This is optional.  
            #     # See https://jchrist.github.io/skein/distributing-files.html  
            #     environment: environment.tar.gz
```

## Ray Setup in YARN

Below is a walkthrough of the bash commands used to start the `ray-head` and `ray-worker` services. Note that this configuration will launch a new Ray cluster for each application, not reuse the same cluster.

### Head node commands

Start by activating a pre-existing environment for dependency management.

```
source environment/bin/activate
```

Register the Ray head address needed by the workers in the Skein key-value store.

```
skein kv put --key=RAY_HEAD_ADDRESS --value=$(hostname -i) current
```

Start all the processes needed on the ray head node. By default, we set object store memory and heap memory to roughly 200 MB. This is conservative and should be set according to application needs.

```
ray start --head --redis-port=6379 --object-store-memory=200000000 --memory 200000000
↪--num-cpus=1
```

Execute the user script containing the Ray program.

```
python example.py
```

Clean up all started processes even if the application fails or is killed.

```
ray stop
skein application shutdown current
```

Putting things together, we have:

```
ray-head:
# There should only be one instance of the head node per cluster.
instances: 1
resources:
# The resources for the head node.
vcores: 1
memory: 2048
files:
# ray/doc/yarn/example.py
example.py: example.py
#     # A packaged python environment using `conda-pack`. Note that Skein
#     # doesn't require any specific way of distributing files, but this
#     # is a good one for python projects. This is optional.
#     # See https://jchrist.github.io/skein/distributing-files.html
#     # environment: environment.tar.gz
script: |
# Activate the packaged conda environment
# - source environment/bin/activate

# This stores the Ray head address in the Skein key-value store so that
↪the workers can retrieve it later.
skein kv put current --key=RAY_HEAD_ADDRESS --value=$(hostname -i)

# This command starts all the processes needed on the ray head node.
```

(continues on next page)

(continued from previous page)

```

# By default, we set object store memory and heap memory to roughly 200MB.
# This is conservative
# and should be set according to application needs.
#
ray start --head --redis-port=6379 --object-store-memory=200000000 --
memory 200000000 --num-cpus=1

# This executes the user script.
python example.py

# After the user script has executed, all started processes should also die.
ray stop
skein application shutdown current

```

## Worker node commands

Fetch the address of the head node from the Skein key-value store.

```
RAY_HEAD_ADDRESS=$(skein kv get current --key=RAY_HEAD_ADDRESS)
```

Start all of the processes needed on a ray worker node, blocking until killed by Skein/YARN via SIGTERM. After receiving SIGTERM, all started processes should also die (ray stop).

```
ray start --object-store-memory=200000000 --memory 200000000 --num-cpus=1 --address=
$RAY_HEAD_ADDRESS:6379 --block; ray stop
```

Putting things together, we have:

```

ray-worker:
# The number of instances to start initially. This can be scaled
# dynamically later.
instances: 4
resources:
# The resources for the worker node
vcores: 1
memory: 2048
# files:
#   environment: environment.tar.gz
depends:
# Don't start any worker nodes until the head node is started
- ray-head
script: |
# Activate the packaged conda environment
# - source environment/bin/activate

# This command gets any addresses it needs (e.g. the head node) from
# the skein key-value store.
RAY_HEAD_ADDRESS=$(skein kv get --key=RAY_HEAD_ADDRESS current)

# The below command starts all the processes needed on a ray worker node,
# blocking until killed with sigterm.
# After sigterm, all started processes should also die (ray stop).
ray start --object-store-memory=200000000 --memory 200000000 --num-cpus=1
--address=$RAY_HEAD_ADDRESS:6379 --block; ray stop

```

## Running a Job

Within your Ray script, use the following to connect to the started Ray cluster:

```
DRIVER_MEMORY = 100 * 1024 * 1024
ray.init(
    address="localhost:6379", driver_object_store_memory=DRIVER_MEMORY)
main()
```

You can use the following command to launch the application as specified by the Skein YAML file.

```
skein application submit [TEST.YAML]
```

Once it has been submitted, you can see the job running on the YARN dashboard.

**Logs for container\_1573672473382\_0014\_01\_000002**

```
19/11/13 20:08:11 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
19/11/13 20:08:12 INFO skein.Driver: Driver started, listening on 38063
19/11/13 20:08:12 INFO conf.Configuration: resource-types.xml not found
19/11/13 20:08:12 INFO conf.Configuration: Using default resource-types.xml".
19/11/13 20:08:12 INFO resource.ResourcesUtils: Adding resource type - name = memory-MB, units = Mi, type = COUNTABLE
19/11/13 20:08:12 INFO resource.ResourcesUtils: Adding resource type - name = vcores, units = 1, type = COUNTABLE
2019-11-13 20:08:14,113 INFO scripts.py:118 -- Using IP address 10.1.2.5 for this node.
2019-11-13 20:08:14,124 INFO resource.Spec:py216 -- Starting Ray with 0.19 GB memory available for workers and up to 0.19 GB for objects. You can adjust these settings with ray.init(memory=<bytes>, object_store_memory=<bytes>).
2019-11-13 20:08:14,124 INFO resource.Spec:py216 -- Starting Ray with 0.19 GB memory available for workers and up to 0.19 GB for objects. You can adjust these settings with ray.init(memory=<bytes>, object_store_memory=<bytes>).
2019-11-13 20:08:14,133 INFO raylet:py16 -- Starting Raylet on port 6379 (http://10.1.2.5:6379/dashboard).
2019-11-13 20:08:14,136 WARNING services.py:964 -- Failed to start the reporter. The reporter requires 'pip install putil'.
2019-11-13 20:08:14,136 WARNING services.py:964 -- Failed to start the reporter. The reporter requires 'pip install putil'.
Started Ray on this node. You can add additional nodes to the cluster by calling
ray start --redis-address=10.1.2.5:6379
from the node you wish to add. You can connect a driver to the cluster from Python by running
import ray
ray.init(redis_address="10.1.2.5:6379")
If you have trouble connecting from a different machine, check that your firewall is configured properly. If you wish to terminate the processes that have been started, run
ray stop
WARNING: Not monitoring node memory since 'putil' is not installed. Install this with 'pip install putil' (or ray[debug]) to enable debugging of memory-related crashes.
2019-11-13 20:08:15,123 WARNING worker.py:674 -- WARNING: Not updating worker name since 'setproctitle' is not installed. Install this with 'pip install setproctitle' (or ray[debug]) to enable monitoring of worker processes.
WARNING: Not monitoring node memory since 'putil' is not installed. Install this with 'pip install putil' (or ray[debug]) to enable debugging of memory-related crashes.
1 nodes have joined so far, waiting for 3 more.
1 nodes have joined so far, waiting for 3 more.
2 nodes have joined so far, waiting for 1 more.
3 nodes have joined so far, waiting for 1 more.
4 nodes have joined so far, waiting for 0 more.
Counter({'rayon yarn': 100})
Iteration 1
Counter({'rayon yarn': 100})
Iteration 2
Counter({'rayon yarn': 100})
Iteration 3
Counter({'rayon yarn': 100})
Iteration 4
Counter({'rayon yarn': 100})
Iteration 5
Counter({'rayon yarn': 100})
Iteration 6
Counter({'rayon yarn': 100})
Iteration 7
Counter({'rayon yarn': 100})
Iteration 8
Counter({'rayon yarn': 100})
Iteration 9
Counter({'rayon yarn': 100})
Counter({'rayon yarn': 100})
success!
```

## Cleaning Up

To clean up a running job, use the following (using the application ID):

```
skein application shutdown $appid
```

## Questions or Issues?

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

1. [ray-dev@googlegroups.com](mailto: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.

## 5.6.4 Deploying on Kubernetes

---

**Note:** The easiest way to run a Ray cluster is by using the built-in autoscaler, which has support for running on top of Kubernetes. Please see the [autoscaler documentation](#) for details.

---

**Warning:** Running Ray on Kubernetes is still a work in progress. If you have a suggestion for how to improve this documentation or want to request a missing feature, please get in touch using one of the channels in the [Questions or Issues?](#) section below.

This document assumes that you have access to a Kubernetes cluster and have `kubectl` installed locally and configured to access the cluster. It will first walk you through how to deploy a Ray cluster on your existing Kubernetes cluster, then explore a few different ways to run programs on the Ray cluster.

The configuration `yaml` files used here are provided in the [Ray repository](#) as examples to get you started. When deploying real applications, you will probably want to build and use your own container images, add more worker nodes to the cluster (or use the [Kubernetes Horizontal Pod Autoscaler](#)), and change the resource requests for the head and worker nodes. Refer to the provided `yaml` files to be sure that you maintain important configuration options for Ray to function properly.

### Creating a Ray Namespace

First, create a Kubernetes Namespace for Ray resources on your cluster. The following commands will create resources under this Namespace, so if you want to use a different one than `ray`, please be sure to also change the `namespace` fields in the provided `yaml` files and anytime you see a `-n` flag passed to `kubectl`.

```
$ kubectl create -f ray/doc/kubernetes/ray-namespace.yaml
```

### Starting a Ray Cluster

A Ray cluster consists of a single head node and a set of worker nodes (the provided `ray-cluster.yaml` file will start 3 worker nodes). In the example Kubernetes configuration, this is implemented as:

- A `ray-head` Kubernetes Service that enables the worker nodes to discover the location of the head node on start up.
- A `ray-head` Kubernetes Deployment that backs the `ray-head` Service with a single head node pod (replica).
- A `ray-worker` Kubernetes Deployment with multiple worker node pods (replicas) that connect to the `ray-head` pod using the `ray-head` Service.

Note that because the head and worker nodes are Deployments, Kubernetes will automatically restart pods that crash to maintain the correct number of replicas.

- If a worker node goes down, a replacement pod will be started and joined to the cluster.
- If the head node goes down, it will be restarted. This will start a new Ray cluster. Worker nodes that were connected to the old head node will crash and be restarted, connecting to the new head node when they come back up.

Try deploying a cluster with the provided Kubernetes config by running the following command:

```
$ kubectl apply -f ray/doc/kubernetes/ray-cluster.yaml
```

Verify that the pods are running by running `kubectl get pods -n ray`. You may have to wait up to a few minutes for the pods to enter the ‘Running’ state on the first run.

\$ kubectl -n ray 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

---

**Note:** You might see a nonzero number of RESTARTS for the worker pods. That can happen when the worker pods start up before the head pod and the workers aren’t able to connect. This shouldn’t affect the behavior of the cluster.

---

To change the number of worker nodes in the cluster, change the `replicas` field in the worker deployment configuration in that file and then re-apply the config as follows:

```
# Edit 'ray/doc/kubernetes/ray-cluster.yaml' and change the 'replicas'
# field under the ray-worker deployment to, e.g., 4.

# Re-apply the new configuration to the running deployment.
$ kubectl apply -f ray/doc/kubernetes/ray-cluster.yaml
service/ray-head unchanged
deployment.apps/ray-head unchanged
deployment.apps/ray-worker configured

# Verify that there are now the correct number of worker pods running.
$ kubectl -n ray get pods
NAME                  READY   STATUS    RESTARTS   AGE
ray-head-5455bb66c9-6bxvz   1/1    Running   0          30s
ray-worker-5c49b7cc57-c6xs8   1/1    Running   0          25s
ray-worker-5c49b7cc57-d9m86   1/1    Running   0          25s
ray-worker-5c49b7cc57-kzk4s   1/1    Running   0          25s
ray-worker-5c49b7cc57-zzfg2   1/1    Running   0          0s
```

To validate that the restart behavior is working properly, try killing pods and checking that they are restarted by Kubernetes:

```
# Delete a worker pod.
$ kubectl -n ray delete ray-worker-5c49b7cc57-c6xs8
pod "ray-worker-5c49b7cc57-c6xs8" deleted

# Check that a new worker pod was started (this may take a few seconds).
$ kubectl -n ray get pods
NAME                  READY   STATUS    RESTARTS   AGE
ray-head-5455bb66c9-6bxvz   1/1    Running   0          45s
ray-worker-5c49b7cc57-d9m86   1/1    Running   0          40s
ray-worker-5c49b7cc57-kzk4s   1/1    Running   0          40s
ray-worker-5c49b7cc57-ypq8x   1/1    Running   0          0s

# Delete the head pod.
$ kubectl -n ray delete ray-head-5455bb66c9-6bxvz
pod "ray-head-5455bb66c9-6bxvz" deleted

# Check that a new head pod was started and the worker pods were restarted.
$ kubectl -n ray get pods
NAME                  READY   STATUS    RESTARTS   AGE
```

(continues on next page)

(continued from previous page)

```

ray-head-5455bb66c9-gqzql    1/1    Running   0        0s
ray-worker-5c49b7cc57-d9m86   1/1    Running   1        50s
ray-worker-5c49b7cc57-kzk4s   1/1    Running   1        50s
ray-worker-5c49b7cc57-ypq8x   1/1    Running   1        10s

# You can even try deleting all of the pods in the Ray namespace and checking
# that Kubernetes brings the right number back up.
$ kubectl -n ray delete pods --all
$ kubectl -n ray get pods
NAME                  READY   STATUS    RESTARTS   AGE
ray-head-5455bb66c9-716xj   1/1    Running   0          10s
ray-worker-5c49b7cc57-57tpv  1/1    Running   0          10s
ray-worker-5c49b7cc57-6m4kp  1/1    Running   0          10s
ray-worker-5c49b7cc57-jx2w2  1/1    Running   0          10s

```

## Running Ray Programs

This section assumes that you have a running Ray cluster (if you don't, please refer to the section above to get started) and will walk you through three different options to run a Ray program on it:

1. Using `kubectl exec` to run a Python script.
2. Using `kubectl exec -it bash` to work interactively in a remote shell.
3. Submitting a `Kubernetes Job`.

### Running a program using ‘`kubectl exec`’

To run an example program that tests object transfers between nodes in the cluster, try the following commands (don't forget to replace the head pod name - you can find it by running `kubectl -n ray get pods`):

```

# Copy the test script onto the head node.
$ kubectl -n ray cp ray/doc/kubernetes/example.py ray-head-5455bb66c9-716xj:/example.
˓→py

# Run the example program on the head node.
$ kubectl -n ray exec ray-head-5455bb66c9-716xj -- python example.py
# You should see repeated output for 10 iterations and then 'Success!'

```

### Running a program in a remote shell

You can also run tasks interactively on the cluster by connecting a remote shell to one of the pods.

```

# Copy the test script onto the head node.
$ kubectl -n ray cp ray/doc/kubernetes/example.py ray-head-5455bb66c9-716xj:/example.
˓→py

# Get a remote shell to the head node.
$ kubectl -n ray exec -it ray-head-5455bb66c9-716xj -- bash

# Run the example program on the head node.
root@ray-head-6f566446c-5rdmb:/# python example.py
# You should see repeated output for 10 iterations and then 'Success!'

```

You can also start an IPython interpreter to work interactively:

```
# From your local machine.
$ kubectl -n ray exec -it ray-head-5455bb66c9-716xj -- ipython

# From a remote shell on the head node.
$ kubectl -n ray exec -it ray-head-5455bb66c9-716xj -- bash
root@ray-head-6f566446c-5rdmb:/# ipython
```

Once you have the IPython interpreter running, try running the following example program:

```
from collections import Counter
import socket
import time
import ray

ray.init(address="$RAY_HEAD_SERVICE_HOST:$RAY_HEAD_SERVICE_PORT_REDIS_PRIMARY")

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

# Check that objects can be transferred from each node to each other node.
%time Counter(ray.get([f.remote(f.remote()) for _ in range(100)]))
```

## Submitting a Job

You can also submit a Ray application to run on the cluster as a [Kubernetes Job](#). The Job will run a single pod running the Ray driver program to completion, then terminate the pod but allow you to access the logs.

To submit a Job that downloads and executes an [example program](#) that tests object transfers between nodes in the cluster, run the following command:

```
$ kubectl create -f ray/doc/kubernetes/ray-job.yaml
job.batch/ray-test-job-kw5gn created
```

To view the output of the Job, first find the name of the pod that ran it, then fetch its logs:

```
$ kubectl -n ray get pods
NAME                      READY   STATUS    RESTARTS   AGE
ray-head-5455bb66c9-716xj  1/1     Running   0          15s
ray-test-job-kw5gn-5g7tv   0/1     Completed  0          10s
ray-worker-5c49b7cc57-57tpv 1/1     Running   0          15s
ray-worker-5c49b7cc57-6m4kp 1/1     Running   0          15s
ray-worker-5c49b7cc57-jx2w2 1/1     Running   0          15s

# Fetch the logs. You should see repeated output for 10 iterations and then
# 'Success!'
$ kubectl -n ray logs ray-test-job-kw5gn-5g7tv
```

To clean up the resources created by the Job after checking its output, run the following:

```
# List Jobs run in the Ray namespace.
$ kubectl -n ray get jobs
NAME          COMPLETIONS   DURATION   AGE
```

(continues on next page)

(continued from previous page)

```

ray-test-job-kw5gn  1/1           10s          30s

# Delete the finished Job.
$ kubectl -n ray delete job ray-test-job-kw5gn

# Verify that the Job's pod was cleaned up.
$ kubectl -n ray get pods
NAME                READY   STATUS    RESTARTS   AGE
ray-head-5455bb66c9-716xj  1/1     Running   0          60s
ray-worker-5c49b7cc57-57tpv  1/1     Running   0          60s
ray-worker-5c49b7cc57-6m4kp  1/1     Running   0          60s
ray-worker-5c49b7cc57-jx2w2  1/1     Running   0          60s

```

## Cleaning Up

To delete a running Ray cluster, you can run the following command:

```
kubectl delete -f ray/doc/kubernetes/ray-cluster.yaml
```

## Questions or Issues?

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

1. [ray-dev@googlegroups.com](mailto: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.

## 5.6.5 Deploying on Slurm

Clusters managed by Slurm may require that Ray is initialized as a part of the submitted job. This can be done by using `srun` within the submitted script. For example:

```

#!/bin/bash

#SBATCH --job-name=test
#SBATCH --cpus-per-task=5
#SBATCH --mem-per-cpu=1GB
#SBATCH --nodes=3
#SBATCH --tasks-per-node 1

worker_num=2 # Must be one less than the total number of nodes

# module load Langs/Python/3.6.4 # This will vary depending on your environment
# source venv/bin/activate

nodes=$(scontrol show hostnames $SLURM_JOB_NODELIST) # Getting the node names
nodes_array=( $nodes )

node1=${nodes_array[0]}

ip_prefix=$(srun --nodes=1 --ntasks=1 -w $node1 hostname --ip-address) # Making
˓→address

```

(continues on next page)

(continued from previous page)

```

suffix=':6379'
ip_head=$ip_prefix$suffix
redis_password=$(uuidgen)

export ip_head # Exporting for latter access by trainer.py

srun --nodes=1 --ntasks=1 -w $node1 ray start --block --head --redis-port=6379 --
redis-password=$redis_password & # Starting the head
sleep 5
# Make sure the head successfully starts before any worker does, otherwise
# the worker will not be able to connect to redis. In case of longer delay,
# adjust the sleep time above to ensure proper order.

for (( i=1; i<=$worker_num; i++ ))
do
    node2=${nodes_array[$i]}
    srun --nodes=1 --ntasks=1 -w $node2 ray start --block --address=$ip_head --redis-
password=$redis_password & # Starting the workers
    # Flag --block will keep ray process alive on each compute node.
    sleep 5
done

python -u trainer.py $redis_password 15 # Pass the total number of allocated CPUs

```

```

# trainer.py
from collections import Counter
import os
import sys
import time
import ray

redis_password = sys.argv[1]
num_cpus = int(sys.argv[2])

ray.init(address=os.environ["ip_head"], redis_password=redis_password)

print("Nodes in the Ray cluster:")
print(ray.nodes())

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

# The following takes one second (assuming that ray was able to access all of the_
# allocated nodes).
for i in range(60):
    start = time.time()
    ip_addresses = ray.get([f.remote() for _ in range(num_cpus)])
    print(Counter(ip_addresses))
    end = time.time()
    print(end - start)

```

## 5.6.6 Ray Projects (Experimental)

Ray projects make it easy to package a Ray application so it can be rerun later in the same environment. They allow for the sharing and reliable reuse of existing code.

### Quick start (CLI)

```
# Creates a project in the current directory. It will create a
# project.yaml defining the code and environment and a cluster.yaml
# describing the cluster configuration. Both will be created in the
# ray-project subdirectory of the current directory.
$ ray project create <project-name>

# Create a new session from the given project. Launch a cluster and run
# the command, which must be specified in the project.yaml file. If no
# command is specified, the "default" command in ray-project/project.yaml
# will be used. Alternatively, use --shell to run a raw shell command.
$ ray session start <command-name> [arguments] [--shell]

# Open a console for the given session.
$ ray session attach

# Stop the given session and terminate all of its worker nodes.
$ ray session stop
```

### Examples

See the [readme](#) for instructions on how to run these examples:

- [Open Tacotron](#): A TensorFlow implementation of Google’s Tacotron speech synthesis with pre-trained model (unofficial)
- [PyTorch Transformers](#): A library of state-of-the-art pretrained models for Natural Language Processing (NLP)

### Tutorial

We will walk through how to use projects by executing the [streaming MapReduce example](#). Commands always apply to the project in the current directory. Let us switch into the project directory with

```
cd ray/doc/examples/streaming
```

A session represents a running instance of a project. Let’s start one with

```
ray session start
```

The `ray session start` command will bring up a new cluster and initialize the environment of the cluster according to the `environment` section of the `project.yaml`, installing all dependencies of the project.

Now we can execute a command in the session. To see a list of all available commands of the project, run

```
ray session commands
```

which produces the following output:

```
Active project: ray-example-streaming

Command "run":
usage: run [--num-mappers NUM_MAPPERS] [--num-reducers NUM_REDUCERS]

Start the streaming example.

optional arguments:
--num-mappers NUM_MAPPERS
                    Number of mapper actors used
--num-reducers NUM_REDUCERS
                    Number of reducer actors used
```

As you see, in this project there is only a single `run` command which has arguments `--num-mappers` and `--num-reducers`. We can execute the streaming wordcount with the default parameters by running

```
ray session execute run
```

You can interrupt the command with `<Control>-c` and attach to the running session by executing

```
ray session attach --tmux
```

Inside the session you can for example edit the streaming applications with

```
cd ray-example-streaming
emacs streaming.py
```

Try for example to add the following lines after the `for count in counts:` loop:

```
if "million" in wordcounts:
    print("Found the word!")
```

and re-run the application from outside the session with

```
ray session execute run
```

The session can be terminated from outside the session with

```
ray session stop
```

## Project file format (`project.yaml`)

A project file contains everything required to run a project. This includes a cluster configuration, the environment and dependencies for the application, and the specific inputs used to run the project.

Here is an example for a minimal project format:

```
name: test-project
description: "This is a simple test project"
repo: https://github.com/ray-project/ray

# Cluster to be instantiated by default when starting the project.
cluster:
    config: ray-project/cluster.yaml

# Commands/information to build the environment, once the cluster is
```

(continues on next page)

(continued from previous page)

```

# instantiated. This can include the versions of python libraries etc.
# It can be specified as a Python requirements.txt, a conda environment,
# a Dockerfile, or a shell script to run to set up the libraries.
environment:
  requirements: requirements.txt

# List of commands that can be executed once the cluster is instantiated
# and the environment is set up.
# A command can also specify a cluster that overwrites the default cluster.
commands:
  - name: default
    command: python default.py
    help: "The command that will be executed if no command name is specified"
  - name: test
    command: python test.py --param1={{param1}} --param2={{param2}}
    help: "A test command"
    params:
      - name: "param1"
        help: "The first parameter"
        # The following line indicates possible values this parameter can take.
        choices: ["1", "2"]
      - name: "param2"
        help: "The second parameter"

```

Project files have to adhere to the following schema:

<b>type</b>	<i>object</i>	
<b>properties</b>		
• <b>name</b>	The name of the project	
	<b>type</b>	<i>string</i>
• <b>description</b>	A short description of the project	
	<b>type</b>	<i>string</i>
• <b>repo</b>	The URL of the repo this project is part of	
	<b>type</b>	<i>string</i>
• <b>documentation</b>	Link to the documentation of this project	
	<b>type</b>	<i>string</i>
• <b>tags</b>	Relevant tags for this project	
	<b>type</b>	<i>array</i>
	<b>items</b>	
	•	<b>type</b> <i>string</i>
• <b>cluster</b>	<b>type</b>	<i>object</i>
<b>properties</b>		
• <b>config</b>	Path to a .yaml cluster configuration file (relative to the project root)	
	<b>type</b>	<i>string</i>
• <b>params</b>	<b>type</b>	<i>array</i>
	<b>items</b>	
	•	<b>type</b> <i>object</i>
	<b>properties</b>	
	• <b>name</b>	<b>type</b> <i>string</i>

continues on next page

Table 1 – continued from previous page

			• help	type	<i>string</i>
			• choices	type	<i>array</i>
			• default		
		• type	type	<i>string</i>	
				enum	int, float, str
			additionalProperties	<b>False</b>	
		additionalProperties	<b>False</b>		
•	environment	The environment that needs to be set up to run the project			
	type	<i>object</i>			
	properties				
	• dockerimage	URL to a docker image that can be pulled to run the project in			
	type	<i>string</i>			
	• dockerfile	Path to a Dockerfile to set up an image the project can run in (relative to the project root)			
	type	<i>string</i>			
	• requirements	Path to a Python requirements.txt file to set up project dependencies (relative to the project root)			
	type	<i>string</i>			
	• shell	A sequence of shell commands to run to set up the project environment			
	type	<i>array</i>			
	items				
		•	type	<i>string</i>	
	additionalProperties	<b>False</b>			
•	commands	type	<i>array</i>		
	items				
	•	Possible commands to run to start a session			
	type	<i>object</i>			
	properties				
	• name	Name of the command			
	type	<i>string</i>			
	• help	Help string for the command			
	type	<i>string</i>			
	• command	Shell command to run on the cluster			
	type	<i>string</i>			
	• params	type	<i>array</i>		
	items				
	•	Possible parameters in the command			
	type	<i>object</i>			
	properties				
	• name	Name of the parameter			
	type	<i>string</i>			
	• help	Help string for the parameter			
	type	<i>string</i>			
	• choices	Possible values the parameter can take			

continues on next page

Table 1 – continued from previous page

					type	array		
				• default				
				• type	Required type for the parameter			
					type	<i>string</i>		
					enum	int, float, str		
				additionalProperties	<b>False</b>			
		• config	Configuration options for the command					
			type	<i>object</i>				
			properties					
			• tmux	If true, the command will be run inside of tmux				
				type	<i>boolean</i>			
				additionalProperties	<b>False</b>			
		additionalProperties	<b>False</b>					
•	output_file	type	array					
	items							
		•	type	<i>string</i>				
		additionalProperties	<b>False</b>					

### Cluster file format (cluster.yaml)

This is the same as for the autoscaler, see [Cluster Launch](#) page.

## 5.7 Ray Tutorials and Examples

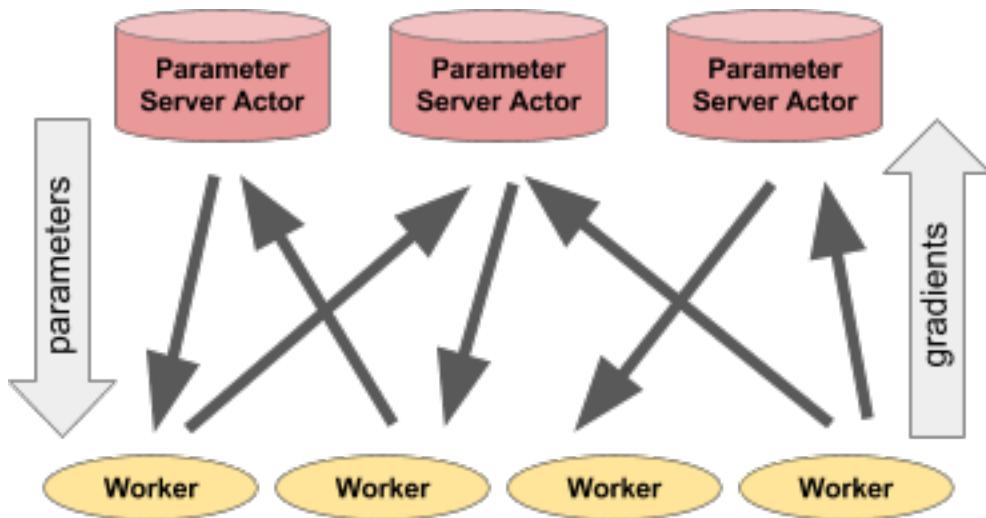
Get started with Ray, Tune, and RLLib with these notebooks that you can run online in CoLab or Binder:

- Ray Tutorial Notebooks

### 5.7.1 Parameter Server

The parameter server is a framework for distributed machine learning training.

In the parameter server framework, a centralized server (or group of server nodes) maintains global shared parameters of a machine-learning model (e.g., a neural network) while the data and computation of calculating updates (i.e., gradient descent updates) are distributed over worker nodes.



Parameter servers are a core part of many machine learning applications. This document walks through how to implement simple synchronous and asynchronous parameter servers using Ray actors.

To run the application, first install some dependencies.

```
pip install torch torchvision filelock
```

Let's first define some helper functions and import some dependencies.

```
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from filelock import FileLock
import numpy as np

import ray

def get_data_loader():
    """Safely downloads data. Returns training/validation set dataloader."""
    mnist_transforms = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.1307, ), (0.3081, ))])

    # We add FileLock here because multiple workers will want to
    # download data, and this may cause overwrites since
    # DataLoader is not threadsafe.
    with FileLock(os.path.expanduser("~/data.lock")):
        train_loader = torch.utils.data.DataLoader(
            datasets.MNIST(
                "~/data",
                train=True,
                download=True,
                transform=mnist_transforms),
            batch_size=128,
            shuffle=True)
        test_loader = torch.utils.data.DataLoader(
            datasets.MNIST("~/data", train=False, transform=mnist_transforms),
```

(continues on next page)

(continued from previous page)

```

        batch_size=128,
        shuffle=True)
    return train_loader, test_loader

def evaluate(model, test_loader):
    """Evaluates the accuracy of the model on a validation dataset."""
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(test_loader):
            # This is only set to finish evaluation faster.
            if batch_idx * len(data) > 1024:
                break
            outputs = model(data)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
    return 100. * correct / total

```

## Setup: Defining the Neural Network

We define a small neural network to use in training. We provide some helper functions for obtaining data, including getter/setter methods for gradients and weights.

```

class ConvNet(nn.Module):
    """Small ConvNet for MNIST."""

    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 3, kernel_size=3)
        self.fc = nn.Linear(192, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 3))
        x = x.view(-1, 192)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)

    def get_weights(self):
        return {k: v.cpu() for k, v in self.state_dict().items()}

    def set_weights(self, weights):
        self.load_state_dict(weights)

    def get_gradients(self):
        grads = []
        for p in self.parameters():
            grad = None if p.grad is None else p.grad.data.cpu().numpy()
            grads.append(grad)
        return grads

    def set_gradients(self, gradients):
        for g, p in zip(gradients, self.parameters()):

```

(continues on next page)

(continued from previous page)

```
if g is not None:
    p.grad = torch.from_numpy(g)
```

## Defining the Parameter Server

The parameter server will hold a copy of the model. During training, it will:

1. Receive gradients and apply them to its model.
2. Send the updated model back to the workers.

The `@ray.remote` decorator defines a remote process. It wraps the `ParameterServer` class and allows users to instantiate it as a remote actor.

```
@ray.remote
class ParameterServer(object):
    def __init__(self, lr):
        self.model = ConvNet()
        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=lr)

    def apply_gradients(self, *gradients):
        summed_gradients = [
            np.stack(gradient_zip).sum(axis=0)
            for gradient_zip in zip(*gradients)
        ]
        self.optimizer.zero_grad()
        self.model.set_gradients(summed_gradients)
        self.optimizer.step()
        return self.model.get_weights()

    def get_weights(self):
        return self.model.get_weights()
```

## Defining the Worker

The worker will also hold a copy of the model. During training, it will continuously evaluate data and send gradients to the parameter server. The worker will synchronize its model with the Parameter Server model weights.

```
@ray.remote
class DataWorker(object):
    def __init__(self):
        self.model = ConvNet()
        self.data_iterator = iter(get_data_loader()[0])

    def compute_gradients(self, weights):
        self.model.set_weights(weights)
        try:
            data, target = next(self.data_iterator)
        except StopIteration: # When the epoch ends, start a new epoch.
            self.data_iterator = iter(get_data_loader()[0])
            data, target = next(self.data_iterator)
        self.model.zero_grad()
        output = self.model(data)
        loss = F.nll_loss(output, target)
```

(continues on next page)

(continued from previous page)

```
loss.backward()
return self.model.get_gradients()
```

## Synchronous Parameter Server Training

We'll now create a synchronous parameter server training scheme. We'll first instantiate a process for the parameter server, along with multiple workers.

```
iterations = 200
num_workers = 2

ray.init(ignore_reinit_error=True)
ps = ParameterServer.remote(1e-2)
workers = [DataWorker.remote() for i in range(num_workers)]
```

We'll also instantiate a model on the driver process to evaluate the test accuracy during training.

```
model = ConvNet()
test_loader = get_data_loader()[1]
```

Training alternates between:

1. Computing the gradients given the current weights from the server
2. Updating the parameter server's weights with the gradients.

```
print("Running synchronous parameter server training.")
current_weights = ps.get_weights.remote()
for i in range(iterations):
    gradients = [
        worker.compute_gradients.remote(current_weights) for worker in workers
    ]
    # Calculate update after all gradients are available.
    current_weights = ps.apply_gradients.remote(*gradients)

    if i % 10 == 0:
        # Evaluate the current model.
        model.set_weights(ray.get(current_weights))
        accuracy = evaluate(model, test_loader)
        print("Iter {}: \taccuracy is {:.1f}".format(i, accuracy))

print("Final accuracy is {:.1f}.".format(accuracy))
# Clean up Ray resources and processes before the next example.
ray.shutdown()
```

## Asynchronous Parameter Server Training

We'll now create a synchronous parameter server training scheme. We'll first instantiate a process for the parameter server, along with multiple workers.

```
print("Running Asynchronous Parameter Server Training.")

ray.init(ignore_reinit_error=True)
ps = ParameterServer.remote(1e-2)
workers = [DataWorker.remote() for i in range(num_workers)]
```

Here, workers will asynchronously compute the gradients given its current weights and send these gradients to the parameter server as soon as they are ready. When the Parameter server finishes applying the new gradient, the server will send back a copy of the current weights to the worker. The worker will then update the weights and repeat.

```
current_weights = ps.get_weights.remote()

gradients = {}
for worker in workers:
    gradients[worker.compute_gradients.remote(current_weights)] = worker

for i in range(iterations * num_workers):
    ready_gradient_list, _ = ray.wait(list(gradients))
    ready_gradient_id = ready_gradient_list[0]
    worker = gradients.pop(ready_gradient_id)

    # Compute and apply gradients.
    current_weights = ps.apply_gradients.remote(*[ready_gradient_id])
    gradients[worker.compute_gradients.remote(current_weights)] = worker

    if i % 10 == 0:
        # Evaluate the current model after every 10 updates.
        model.set_weights(ray.get(current_weights))
        accuracy = evaluate(model, test_loader)
        print("Iter {}: \taccuracy is {:.1f}.".format(i, accuracy))

print("Final accuracy is {:.1f}.".format(accuracy))
```

## Final Thoughts

This approach is powerful because it enables you to implement a parameter server with a few lines of code as part of a Python application. As a result, this simplifies the deployment of applications that use parameter servers and to modify the behavior of the parameter server.

For example, sharding the parameter server, changing the update rule, switch between asynchronous and synchronous updates, ignoring straggler workers, or any number of other customizations, will only require a few extra lines of code.

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 5.7.2 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.](#)

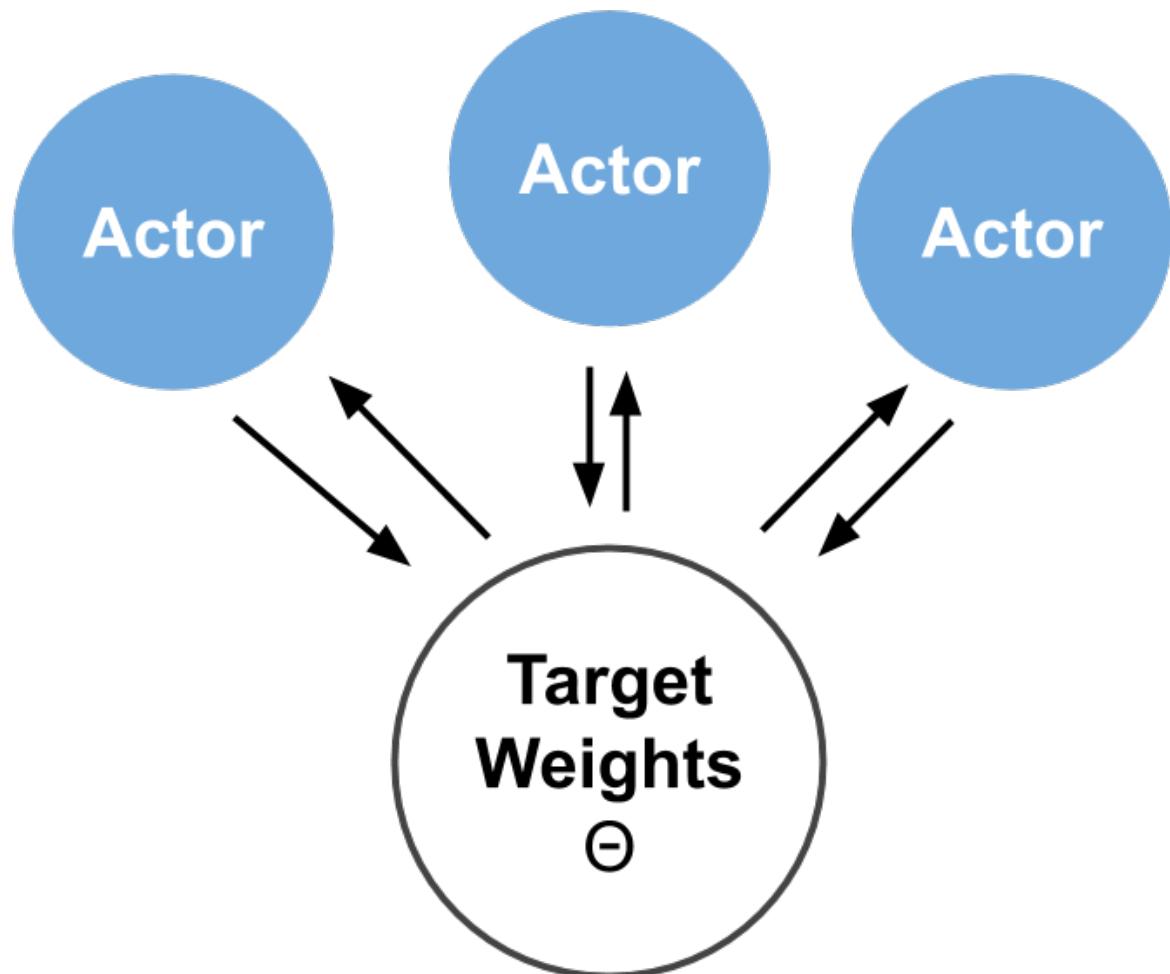
---

**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-headless
pip install scipy
```



You can run the code with

```
rllib train --env=Pong-ram-v4 --run=A3C --config='{"num_workers": N}'
```

## 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.

## 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)
```

(continues on next page)

(continued from previous page)

```

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

```

## 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.remote(env_name, i) for i in range(num_workers)]

    # Start gradient calculation tasks on each actor
    parameters = policy.get_weights()
    gradient_list = [agent.compute_gradient.remote(parameters) for agent in agents]

    while True: # Replace with your termination condition
        # wait for some gradient to be computed - unblock as soon as the earliest
        # arrives
        done_id, gradient_list = ray.wait(gradient_list)

        # get the results of the task from the object store
        gradient, info = ray.get(done_id)[0]
        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.remote(parameters)])
    return policy

```

## Benchmarks and Visualization

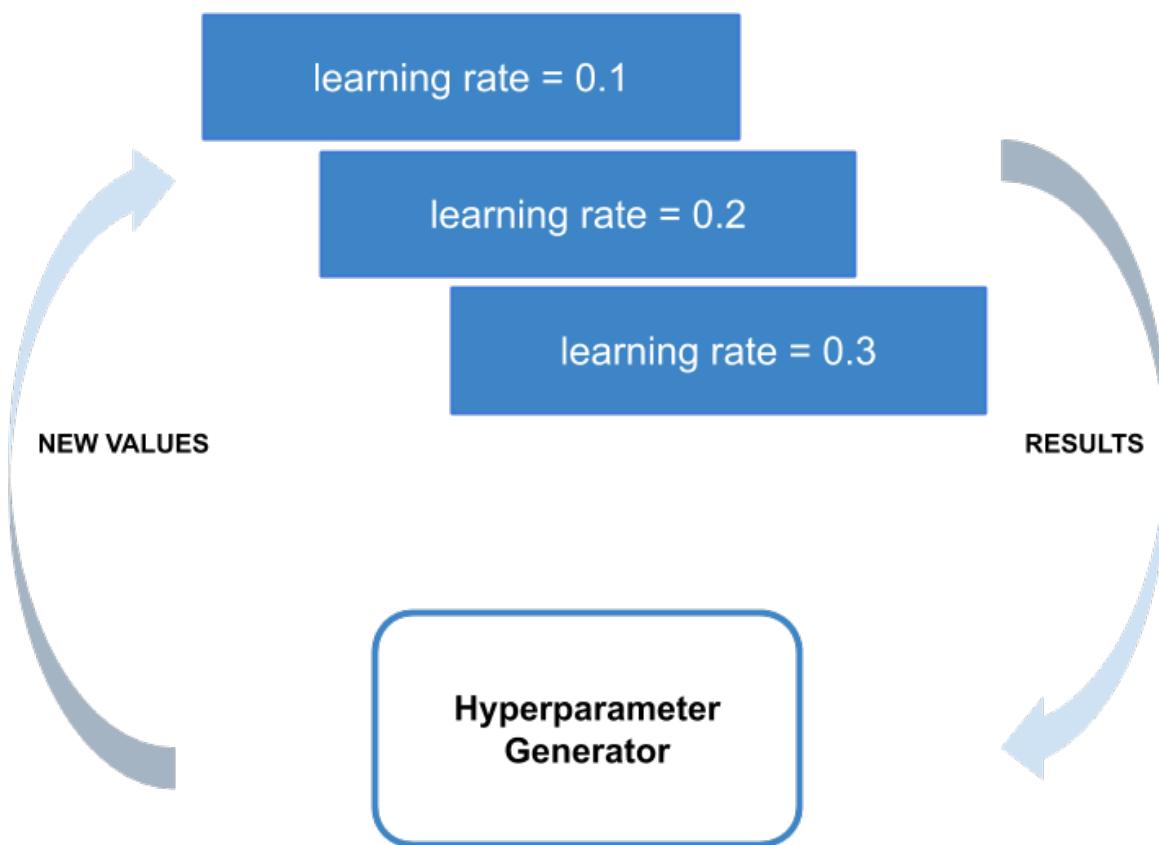
For the PongDeterministic-v4 and an Amazon EC2 m4.16xlarge instance, we are able to train the agent with 16 workers in around 15 minutes. With 8 workers, we can train the agent in around 25 minutes.

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`).

### 5.7.3 Simple Parallel Model Selection

In this example, we'll demonstrate how to quickly write a hyperparameter tuning script that evaluates a set of hyperparameters in parallel.

This script will demonstrate how to use two important parts of the Ray API: using `ray.remote` to define remote functions and `ray.wait` to wait for their results to be ready.




---

**Important:** For a production-grade implementation of distributed hyperparameter tuning, use [Tune](#), a scalable hyperparameter tuning library built using Ray's Actor API.

## Setup: Dependencies

First, import some dependencies and define functions to generate random hyperparameters and retrieve data.

```
import os
import numpy as np
from filelock import FileLock

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

import ray

ray.init()

# The number of sets of random hyperparameters to try.
num_evaluations = 10

# A function for generating random hyperparameters.
def generate_hyperparameters():
    return {
        "learning_rate": 10**np.random.uniform(-5, 1),
        "batch_size": np.random.randint(1, 100),
        "momentum": np.random.uniform(0, 1)
    }

def get_data_loaders(batch_size):
    mnist_transforms = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.1307, ), (0.3081, ))])

    # We add FileLock here because multiple workers will want to
    # download data, and this may cause overwrites since
    # DataLoader is not threadsafe.
    with FileLock(os.path.expanduser("~/data.lock")):
        train_loader = torch.utils.data.DataLoader(
            datasets.MNIST(
                "~/data",
                train=True,
                download=True,
                transform=mnist_transforms),
            batch_size=batch_size,
            shuffle=True)
        test_loader = torch.utils.data.DataLoader(
            datasets.MNIST("~/data", train=False, transform=mnist_transforms),
            batch_size=batch_size,
            shuffle=True)
    return train_loader, test_loader
```

## Setup: Defining the Neural Network

We define a small neural network to use in training. In addition, we created methods to train and test this neural network.

```
class ConvNet(nn.Module):
    """Simple two layer Convolutional Neural Network."""

    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 3, kernel_size=3)
        self.fc = nn.Linear(192, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 3))
        x = x.view(-1, 192)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)

    def train(model, optimizer, train_loader, device=torch.device("cpu")):
        """Optimize the model with one pass over the data.

        Cuts off at 1024 samples to simplify training.
        """
        model.train()
        for batch_idx, (data, target) in enumerate(train_loader):
            if batch_idx * len(data) > 1024:
                return
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
            optimizer.step()

    def test(model, test_loader, device=torch.device("cpu")):
        """Checks the validation accuracy of the model.

        Cuts off at 512 samples for simplicity.
        """
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for batch_idx, (data, target) in enumerate(test_loader):
                if batch_idx * len(data) > 512:
                    break
                data, target = data.to(device), target.to(device)
                outputs = model(data)
                _, predicted = torch.max(outputs.data, 1)
                total += target.size(0)
                correct += (predicted == target).sum().item()

        return correct / total
```

## Evaluating the Hyperparameters

For a given configuration, the neural network created previously will be trained and return the accuracy of the model. These trained networks will then be tested for accuracy to find the best set of hyperparameters.

The `@ray.remote` decorator defines a remote process.

```
@ray.remote
def evaluate_hyperparameters(config):
    model = ConvNet()
    train_loader, test_loader = get_data_loaders(config["batch_size"])
    optimizer = optim.SGD(
        model.parameters(),
        lr=config["learning_rate"],
        momentum=config["momentum"])
    train(model, optimizer, train_loader)
    return test(model, test_loader)
```

## Synchronous Evaluation of Randomly Generated Hyperparameters

We will create multiple sets of random hyperparameters for our neural network that will be evaluated in parallel.

```
# Keep track of the best hyperparameters and the best accuracy.
best_hyperparameters = None
best_accuracy = 0
# A list holding the object IDs for all of the experiments that we have
# launched but have not yet been processed.
remaining_ids = []
# A dictionary mapping an experiment's object ID to its hyperparameters.
# hyperparameters used for that experiment.
hyperparameters_mapping = {}
```

Launch asynchronous parallel tasks for evaluating different hyperparameters. `accuracy_id` is an `ObjectID` that acts as a handle to the remote task. It is used later to fetch the result of the task when the task finishes.

```
# Randomly generate sets of hyperparameters and launch a task to evaluate it.
for i in range(num_evaluations):
    hyperparameters = generate_hyperparameters()
    accuracy_id = evaluate_hyperparameters.remote(hyperparameters)
    remaining_ids.append(accuracy_id)
    hyperparameters_mapping[accuracy_id] = hyperparameters
```

Process each hyperparameter and corresponding accuracy in the order that they finish to store the hyperparameters with the best accuracy.

```
# Fetch and print the results of the tasks in the order that they complete.
while remaining_ids:
    # Use ray.wait to get the object ID of the first task that completes.
    done_ids, remaining_ids = ray.wait(remaining_ids)
    # There is only one return result by default.
    result_id = done_ids[0]

    hyperparameters = hyperparameters_mapping[result_id]
    accuracy = ray.get(result_id)
    print("""We achieve accuracy {:.3}% with
          learning_rate: {:.2}""")

```

(continues on next page)

(continued from previous page)

```

batch_size: {}
momentum: {:.2}
    """.format(100 * accuracy, hyperparameters["learning_rate"],
               hyperparameters["batch_size"], hyperparameters["momentum"]))
if accuracy > best_accuracy:
    best_hyperparameters = hyperparameters
    best_accuracy = accuracy

# Record the best performing set of hyperparameters.
print("""Best accuracy over {} trials was {:.3} with
      learning_rate: {:.2}
      batch_size: {}
      momentum: {:.2}
      """.format(num_evaluations, 100 * best_accuracy,
                 best_hyperparameters["learning_rate"],
                 best_hyperparameters["batch_size"],
                 best_hyperparameters["momentum"]))

```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 5.7.4 Learning to Play Pong

In this example, we'll train a **very simple** neural network to play Pong using the OpenAI Gym.

At a high level, we will use multiple Ray actors to obtain simulation rollouts and calculate gradient simultaneously. We will then centralize these gradients and update the neural network. The updated neural network will then be passed back to each Ray actor for more gradient calculation.

This application is adapted, with minimal modifications, from Andrej Karpathy's [source code](#) (see the accompanying [blog post](#)).

To run the application, first install some dependencies.

```
pip install gym[atari]
```

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.

```

import numpy as np
import os
import ray
import time

import gym

```

## Hyperparameters

Here we'll define a couple of the hyperparameters that are used.

```
H = 200 # The number of hidden layer neurons.  
gamma = 0.99 # The discount factor for reward.  
decay_rate = 0.99 # The decay factor for RMSProp leaky sum of grad^2.  
D = 80 * 80 # The input dimensionality: 80x80 grid.  
learning_rate = 1e-4 # Magnitude of the update.
```

## Helper Functions

We first define a few helper functions:

1. Preprocessing: The `preprocess` function will preprocess the original 210x160x3 uint8 frame into a one-dimensional 6400 float vector.
2. Reward Processing: The `process_rewards` function will calculate a discounted reward. This formula states that the “value” of a sampled action is the weighted sum of all rewards afterwards, but later rewards are exponentially less important.
3. Rollout: The `rollout` function plays an entire game of Pong (until either the computer or the RL agent loses).

```
def preprocess(img):  
    # Crop the image.  
    img = img[35:195]  
    # Downsample by factor of 2.  
    img = img[::2, ::2, 0]  
    # Erase background (background type 1).  
    img[img == 144] = 0  
    # Erase background (background type 2).  
    img[img == 109] = 0  
    # Set everything else (paddles, ball) to 1.  
    img[img != 0] = 1  
    return img.astype(np.float).ravel()  
  
def process_rewards(r):  
    """Compute discounted reward from a vector of rewards."""  
    discounted_r = np.zeros_like(r)  
    running_add = 0  
    for t in reversed(range(0, r.size)):  
        # Reset the sum, since this was a game boundary (pong specific!).  
        if r[t] != 0:  
            running_add = 0  
        running_add = running_add * gamma + r[t]  
        discounted_r[t] = running_add  
    return discounted_r  
  
def rollout(model, env):  
    """Evaluates env and model until the env returns "Done".  
  
    Returns:  
        xs: A list of observations  
        hs: A list of model hidden states per observation  
        dlogps: A list of gradients
```

(continues on next page)

(continued from previous page)

```

drs: A list of rewards.

"""

# Reset the game.
observation = env.reset()
# Note that prev_x is used in computing the difference frame.
prev_x = None
xs, hs, dlogps, drs = [], [], [], []
done = False
while not done:
    cur_x = preprocess(observation)
    x = cur_x - prev_x if prev_x is not None else np.zeros(D)
    prev_x = cur_x

    aprob, h = model.policy_forward(x)
    # Sample an action.
    action = 2 if np.random.uniform() < aprob else 3

    # The observation.
    xs.append(x)
    # The hidden state.
    hs.append(h)
    y = 1 if action == 2 else 0 # A "fake label".
    # The gradient that encourages the action that was taken to be
    # taken (see http://cs231n.github.io/neural-networks-2/#losses if
    # confused).
    dlogps.append(y - aprob)

    observation, reward, done, info = env.step(action)

    # Record reward (has to be done after we call step() to get reward
    # for previous action).
    drs.append(reward)
return xs, hs, dlogps, drs

```

## Neural Network

Here, a neural network is used to define a “policy” for playing Pong (that is, a function that chooses an action given a state).

To implement a neural network in NumPy, we need to provide helper functions for calculating updates and computing the output of the neural network given an input, which in our case is an observation.

```

class Model(object):
    """This class holds the neural network weights."""

    def __init__(self):
        self.weights = {}
        self.weights["W1"] = np.random.randn(H, D) / np.sqrt(D)
        self.weights["W2"] = np.random.randn(H) / np.sqrt(H)

    def policy_forward(self, x):
        h = np.dot(self.weights["W1"], x)
        h[h < 0] = 0 # ReLU nonlinearity.
        logp = np.dot(self.weights["W2"], h)

```

(continues on next page)

(continued from previous page)

```

# Softmax
p = 1.0 / (1.0 + np.exp(-logp))
# Return probability of taking action 2, and hidden state.
return p, h

def policy_backward(self, eph, epx, epdlogp):
    """Backward pass to calculate gradients.

    Arguments:
        eph: Array of intermediate hidden states.
        epx: Array of experiences (observations).
        epdlogp: Array of logps (output of last layer before softmax/
    """

    dW2 = np.dot(eph.T, epdlogp).ravel()
    dh = np.outer(epdlogp, self.weights["W2"])
    # Backprop relu.
    dh[eph <= 0] = 0
    dW1 = np.dot(dh.T, epx)
    return {"W1": dW1, "W2": dW2}

def update(self, grad_buffer, rmsprop_cache, lr, decay):
    """Applies the gradients to the model parameters with RMSProp."""
    for k, v in self.weights.items():
        g = grad_buffer[k]
        rmsprop_cache[k] = (decay * rmsprop_cache[k] + (1 - decay) * g**2)
        self.weights[k] += lr * g / (np.sqrt(rmsprop_cache[k]) + 1e-5)

def zero_grads(grad_buffer):
    """Reset the batch gradient buffer."""
    for k, v in grad_buffer.items():
        grad_buffer[k] = np.zeros_like(v)

```

## Parallelizing Gradients

We define an **actor**, which is responsible for taking a model and an env and performing a rollout + computing a gradient update.

```

ray.init()

@ray.remote
class RolloutWorker(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):

```

(continues on next page)

(continued from previous page)

```

# Compute a simulation episode.
xs, hs, dlogps, drs = rollout(model, self.env)
reward_sum = sum(drs)
# Vectorize the arrays.
epx = np.vstack(xs)
eph = np.vstack(hs)
epdlogp = np.vstack(dlogps)
epr = np.vstack(drs)

# Compute the discounted reward backward through time.
discounted_epr = process_rewards(epr)
# Standardize the rewards to be unit normal (helps control the gradient
# estimator variance).
discounted_epr -= np.mean(discounted_epr)
discounted_epr /= np.std(discounted_epr)
# Modulate the gradient with advantage (the policy gradient magic
# happens right here).
epdlogp *= discounted_epr
return model.policy_backward(eph, epx, epdlogp), reward_sum

```

## Running

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.

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.

```

iterations = 20
batch_size = 4
model = Model()
actors = [RolloutWorker.remote() for _ in range(batch_size)]

running_reward = None
# "Xavier" initialization.
# Update buffers that add up gradients over a batch.
grad_buffer = {k: np.zeros_like(v) for k, v in model.weights.items()}
# Update the rmsprop memory.
rmsprop_cache = {k: np.zeros_like(v) for k, v in model.weights.items()}

for i in range(1, 1 + iterations):
    model_id = ray.put(model)
    gradient_ids = []
    # Launch tasks to compute gradients from multiple rollouts in parallel.
    start_time = time.time()
    gradient_ids = [
        actor.compute_gradient.remote(model_id) for actor in actors
    ]
    for batch in range(batch_size):
        [grad_id], gradient_ids = ray.wait(gradient_ids)
        grad, reward_sum = ray.get(grad_id)
        # Accumulate the gradient over batch.
        for k in model.weights:
            grad_buffer[k] += grad[k]
        running_reward = (reward_sum if running_reward is None else

```

(continues on next page)

(continued from previous page)

```

        running_reward * 0.99 + reward_sum * 0.01)
end_time = time.time()
print("Batch {} computed {} rollouts in {} seconds, "
      "running mean is {}".format(i, batch_size, end_time - start_time,
                                    running_reward))
model.update(grad_buffer, rmsprop_cache, learning_rate, decay_rate)
zero_grads(grad_buffer)

```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 5.7.5 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/doc/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.

### 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

```

(continues on next page)

(continued from previous page)

```
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)
```

## 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, let's turn `loss` and `grad` into methods of an actor that will contain our network.

```
class Network(object):
    def __init__(self):
        # Initialize network.

    def loss(theta, xs, ys):
        # compute the loss
        return loss

    def grad(theta, xs, ys):
        # compute the gradient
        return grad
```

Now, it is easy to speed up the computation of the full loss and the full gradient.

```
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)
```

## 5.7.6 News Reader

This document shows how to implement a simple news reader using Ray. The reader consists of a simple Vue.js [frontend](#) and a backend consisting of a Flask server and a Ray actor. View the code for this example.

To run this example, you will need to install NPM and a few python dependencies.

```
pip install atoma
pip install flask
```

To use this example you need to

- In the `ray/doc/examples/newsreader` directory, start the server with `python server.py`.
- Clone the client code with `git clone https://github.com/ray-project/qreader`
- Start the client with `cd qreader; npm install; npm run dev`
- You can now add a channel by clicking “Add channel” and for example pasting `http://news.ycombinator.com/rss` into the field.
- Star some of the articles and dump the database by running `sqlite3 newsreader.db` in a terminal in the `ray/doc/examples/newsreader` directory and entering `SELECT * FROM news;`.

## 5.7.7 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/doc/examples/streaming/streaming.py
```

For each round of articles read, the script will output the top 10 words in these articles together with their word count:

```
article index = 0
the 2866
of 1688
and 1448
in 1101
to 593
a 553
is 509
as 325
are 284
by 261
article index = 1
the 3597
of 1971
and 1735
in 1429
to 670
a 623
is 578
as 401
by 293
for 285
article index = 2
the 3910
of 2123
and 1890
in 1468
to 658
a 653
is 488
as 364
by 362
for 297
article index = 3
the 2962
of 1667
and 1472
in 1220
a 546
to 538
is 516
as 307
by 253
for 243
article index = 4
the 3523
of 1866
and 1690
in 1475
to 645
a 583
is 572
as 352
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:

```
@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.

```
@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:

```
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.

## 5.7.8 Fault-Tolerant Fairseq Training

This document provides a walkthrough of adapting the [Fairseq library](#) to perform fault-tolerant distributed training on AWS. As an example, we use the WikiText-103 dataset to pretrain the RoBERTa model following [this tutorial](#). The pipeline and configurations in this document will work for other models supported by Fairseq, such as sequence-to-sequence machine translation models.

To run this example, you will need to install Ray on your local machine to use Ray Autoscaler.

You can view the [code for this example](#).

To use Ray Autoscaler on AWS, install boto (`pip install boto3`) and configure your AWS credentials in `~/.aws/credentials` as described on [Automatic Cluster Setup page](#). We provide an [example config file](#) (`lm-cluster.yaml`).

In the example config file, we use an `m5.xlarge` on-demand instance as the head node, and use `p3.2xlarge` GPU spot instances as the worker nodes. We set the minimal number of workers to 1 and maximum workers to 2 in the config, which can be modified according to your own demand.

We also mount [Amazon EFS](#) to store code, data and checkpoints.

---

**Note:** The `{{SecurityGroupId}}` and `{{FileSystemId}}` fields in the config file should be replaced by your own IDs.

---

In `setup_commands`, we use the PyTorch environment in the Deep Learning AMI, and install Ray and Fairseq:

```
setup_commands:
  - echo 'export PATH="$HOME/anaconda3/envs/pytorch_p36/bin:$PATH"' >> ~/.bashrc;
    source ~/.bashrc;
    pip install -U ray;
    pip install -U fairseq==0.8.0;
```

Run the following command on your local machine to start the Ray cluster:

```
ray up lm-cluster.yaml
```

`ray_train.sh` also assumes that all of the `lm/` files are in `$HOME/efs`. You can move these files manually, or use the following command to upload files from a local path:

```
ray rsync-up lm-cluster.yaml PATH/TO/LM '~/efs/lm'
```

### Preprocessing Data

Once the cluster is started, you can then SSH into the head node using `ray attach lm-cluster.yaml` and download or preprocess the data on EFS for training. We can run `preprocess.sh` ([code](#)) to do this, which adapts instructions from [the RoBERTa tutorial](#).

## Training

We provide `ray_train.py` (code) as an entrypoint to the Fairseq library. Since we are training the model on spot instances, we provide fault-tolerance in `ray_train.py` by checkpointing and restarting when a node fails. The code will also check whether there are new resources available after checkpointing. If so, the program will make use of them by restarting and resizing.

Two main components of `ray_train.py` are a `RayDistributedActor` class and a function `run_fault_tolerant_loop()`. The `RayDistributedActor` sets proper arguments for different ray actor processes, adds a checkpoint hook to enable the process to make use of new available GPUs, and calls the main of Fairseq:

```
import math
import copy
import socket
import time

import ray

import fairseq
from fairseq import options
from fairseq_cli.train import main
from contextlib import closing

_original_save_checkpoint = fairseq.checkpoint_utils.save_checkpoint

class RayDistributedActor:
    """Actor to perform distributed training."""

    def run(self, url, world_rank, args):
        """Runs the fairseq training.

        We set args for different ray actors for communication,
        add a checkpoint hook, and call the main function of fairseq.
        """

        # Set the init_method and rank of the process for distributed training.
        print("Ray worker at {} rank {}".format(
            url=url, rank=world_rank))
        self.url = url
        self.world_rank = world_rank
        args.distributed_rank = world_rank
        args.distributed_init_method = url

        # Add a checkpoint hook to make use of new resources.
        self.add_checkpoint_hook(args)

        # Call the original main function of fairseq.
        main(args, init_distributed=(args.distributed_world_size > 1))

    def add_checkpoint_hook(self, args):
        """Add a hook to the original save_checkpoint function.

        This checks if there are new computational resources available.
        If so, raise exception to restart the training process and
        make use of the new resources.
        """

```

(continues on next page)

(continued from previous page)

```

if args.cpu:
    original_n_cpus = args.distributed_world_size

def _new_save_checkpoint(*args, **kwargs):
    _original_save_checkpoint(*args, **kwargs)
    n_cpus = int(ray.cluster_resources()["CPU"])
    if n_cpus > original_n_cpus:
        raise Exception(
            "New CPUs find (original %d CPUs, now %d CPUs)" %
            (original_n_cpus, n_cpus))
    else:
        original_n_gpus = args.distributed_world_size

def _new_save_checkpoint(*args, **kwargs):
    _original_save_checkpoint(*args, **kwargs)
    n_gpus = int(ray.cluster_resources().get("GPU", 0))
    if n_gpus > original_n_gpus:
        raise Exception(
            "New GPUs find (original %d GPUs, now %d GPUs)" %
            (original_n_gpus, n_gpus))

fairseq.checkpoint_utils.save_checkpoint = _new_save_checkpoint

def get_node_ip(self):
    """Returns the IP address of the current node."""
    return ray.services.get_node_ip_address()

def find_free_port(self):
    """Finds a free port on the current node."""
    with closing(socket.socket(socket.AF_INET, socket.SOCK_STREAM)) as s:
        s.bind(("", 0))
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    return s.getsockname()[1]

```

The function `run_fault_tolerant_loop()` provides fault-tolerance by catching failure and restart the computation:

```

def run_fault_tolerant_loop():
    """Entrance function to the fairseq library, providing fault-tolerance."""

    # Parse the command line arguments.
    parser = options.get_training_parser()
    add_ray_args(parser)
    args = options.parse_args_and_arch(parser)
    original_args = copy.deepcopy(args)

    # Main loop for fault-tolerant training.
    retry = True
    while retry:
        args = copy.deepcopy(original_args)

        # Initialize Ray.
        ray.init(address=args.ray_address)

        set_num_resources(args)

```

(continues on next page)

(continued from previous page)

```

set_batch_size(args)

# Set up Ray distributed actors.
Actor = ray.remote(
    num_cpus=1, num_gpus=int(not args.cpu)) (RayDistributedActor)
workers = [Actor.remote() for i in range(args.distributed_world_size)]

# Get the IP address and a free port of actor 0, which is used for
# fairseq distributed training.
ip = ray.get(workers[0].get_node_ip.remote())
port = ray.get(workers[0].find_free_port.remote())
address = "tcp://{}:{port}".format(ip=ip, port=port)

# Start the remote processes, and check whether there are any process
# fails. If so, restart all the processes.
unfinished = [
    worker.run.remote(address, i, args)
    for i, worker in enumerate(workers)
]
try:
    while len(unfinished) > 0:
        finished, unfinished = ray.wait(unfinished)
        finished = ray.get(finished)
        retry = False
except Exception as inst:
    print("Ray restart because following error occurs:")
    print(inst)
    retry = True
ray.shutdown()

```

In `ray_train.py`, we also define a set of helper functions. `add_ray_args()` adds Ray and fault-tolerant training related arguments to the argument parser:

```

def add_ray_args(parser):
    """Add ray and fault-tolerance related parser arguments to the parser."""
    group = parser.add_argument_group("Ray related arguments")
    group.add_argument(
        "--ray-address",
        default="auto",
        type=str,
        help="address for ray initialization")
    group.add_argument(
        "--fix-batch-size",
        default=None,
        metavar="B1,B2,...,BN",
        type=lambda uf: options.eval_str_list(uf, type=int),
        help="fix the actual batch size (max_sentences * update_freq "
             "* n_GPUs) to be the fixed input values by adjusting update_freq "
             "according to actual n_GPUs; the batch size is fixed to Bi for "
             "epoch i; all epochs >N are fixed to BN")
    return group

```

`set_num_resources()` sets the distributed world size to be the number of resources. Also if we want to use GPUs but the current number of GPUs is 0, the function will wait until there is GPU available:

```
def set_num_resources(args):
```

(continues on next page)

(continued from previous page)

```
"""Get the number of resources and set the corresponding fields."""
if args.cpu:
    args.distributed_world_size = int(ray.cluster_resources()["CPU"])
else:
    n_gpus = int(ray.cluster_resources().get("GPU", 0))
    while n_gpus == 0:
        print("No GPUs available, wait 10 seconds")
        time.sleep(10)
        n_gpus = int(ray.cluster_resources().get("GPU", 0))
    args.distributed_world_size = n_gpus
```

`set_batch_size()` keeps the effective batch size to be relatively the same given different number of GPUs:

```
def set_batch_size(args):
    """Fixes the total batch_size to be agnostic to the GPU count."""
    if args.fix_batch_size is not None:
        args.update_freq = [
            math.ceil(batch_size /
                      (args.max_sentences * args.distributed_world_size))
            for batch_size in args.fix_batch_size
        ]
    print("Training on %d GPUs, max_sentences=%d, update_freq=%s" %
          (args.distributed_world_size, args.max_sentences,
           repr(args.update_freq)))
```

To start training, run following commands (`ray_train.sh`) on the head machine:

```
cd ~/efs/lm

TOTAL_UPDATES=125000      # Total number of training steps
WARMUP_UPDATES=10000       # Warmup the learning rate over this many updates
PEAK_LR=0.0005              # Peak learning rate, adjust as needed
TOKENS_PER_SAMPLE=512       # Max sequence length
MAX_POSITIONS=512          # Num. positional embeddings (usually same as above)
MAX_SENTENCES=8             # Number of sequences per batch on one GPU (batch size)
FIX_BATCH_SIZE=2048          # Number of batch size in total (max_sentences * update_
                            ↪freq * n_gpus)
SAVE_INTERVAL_UPDATES=1000   # save a checkpoint every N updates

LOG_DIR=$HOME/efs/lm/log/
DATA_DIR=$HOME/efs/lm/data-bin/wikitext-103/
mkdir -p $LOG_DIR

python $HOME/efs/lm/ray_train.py --fp16 $DATA_DIR \
    --task masked_lm --criterion masked_lm \
    --arch roberta_base --sample-break-mode complete --tokens-per-sample $TOKENS_PER_
    ↪SAMPLE \
    --optimizer adam --adam-betas '(0.9, 0.98)' --adam-eps 1e-6 --clip-norm 0.0 \
    --lr-scheduler polynomial_decay --lr $PEAK_LR --warmup-updates $WARMUP_UPDATES --
    ↪total-num-update $TOTAL_UPDATES \
    --dropout 0.1 --attention-dropout 0.1 --weight-decay 0.01 \
    --max-sentences $MAX_SENTENCES \
    --fix-batch-size $FIX_BATCH_SIZE \
    --max-update $TOTAL_UPDATES --log-format simple --log-interval 1 \
    --save-interval-updates $SAVE_INTERVAL_UPDATES \
    --save-dir $LOG_DIR --ddp-backend=no_c10d
```

SAVE\_INTERVAL\_UPDATES controls how often to save a checkpoint, which can be tuned based on the stability of chosen instances. FIX\_BATCH\_SIZE controls the total batch size to be a roughly fixed number.

## Helpful Ray Commands

To let Ray automatically stop the cluster after the training finished, you can download the `ray_train.sh` to `~/efs` of the remote machine, and run the following command on your local machine:

```
ray exec --stop lm-cluster.yaml 'bash $HOME/efs/lm/ray_train.sh'
```

or run the following command on the remote head node:

```
ray exec --stop ~/ray_bootstrap_config.yaml 'bash $HOME/efs/lm/ray_train.sh'
```

To test the fault-tolerance, you can run the following command on your local machine to randomly kill one node:

```
ray kill-random-node lm-cluster.yaml
```

## 5.7.9 Example Gallery

## 5.8 Ray Package Reference

```
ray.init(address=None,          redis_address=None,          redis_port=None,          num_cpus=None,
         num_gpus=None,        memory=None,          object_store_memory=None,      resources=None,
         driver_object_store_memory=None,    redis_max_memory=None,      log_to_driver=True,
         node_ip_address='127.0.0.1',    object_id_seed=None,      local_mode=False,       redirect_worker_output=None,    redirect_output=None,      ignore_reinit_error=False,
         num_redis_shards=None,    redis_max_clients=None,    redis_password='5241590000000000',   plasma_directory=None,    huge_pages=False,      include_java=False,     include_webui=None,
         webui_host='localhost',    job_id=None,          configure_logging=True,      logging_level=20,
         logging_format='%(asctime)s %(levelname)s %%(filename)s:%(lineno)s -- %(message)s',   plasma_store_socket_name=None,    raylet_socket_name=None,    temp_dir=None,
         load_code_from_local=False, use_pickle=True,      _internal_config=None,    lru_evict=False)
```

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(address="123.45.67.89:6379")
```

You can also define an environment variable called `RAY_ADDRESS` in the same format as the `address` parameter to connect to an existing cluster with `ray.init()`.

### Parameters

- **address** (`str`) – The address of the Ray cluster to connect to. If this address is not provided, then this command will start Redis, a raylet, a plasma store, a plasma manager, and some workers. It will also kill these processes when Python exits. If the driver is running

on a node in a Ray cluster, using *auto* as the value tells the driver to detect the the cluster, removing the need to specify a specific node address.

- **redis\_address** (*str*) – Deprecated; same as address.
- **redis\_port** (*int*) – The port that the primary Redis shard should listen to. If None, then a random port will be chosen.
- **num\_cpus** (*int*) – Number of CPUs the user wishes to assign to each raylet.
- **num\_gpus** (*int*) – Number of GPUs the user wishes to assign to each raylet.
- **resources** – A dictionary mapping the names of custom resources to the quantities for them available.
- **memory** – The amount of memory (in bytes) that is available for use by workers requesting memory resources. By default, this is automatically set based on available system memory.
- **object\_store\_memory** – The amount of memory (in bytes) to start the object store with. By default, this is automatically set based on available system memory, subject to a 20GB cap.
- **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 autoset based on available system memory, subject to a 10GB cap.
- **log\_to\_driver** (*bool*) – If true, the output from all of the worker processes on all nodes will be directed to the driver.
- **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 driver in order to generate the object IDs in a consistent manner. However, the same ID should not be used for different drivers.
- **local\_mode** (*bool*) – If true, the code will be executed serially. This is useful for debugging.
- **driver\_object\_store\_memory** (*int*) – Limit the amount of memory the driver can use in the object store for creating objects. By default, this is autoset based on available system memory, subject to a 20GB cap.
- **ignore\_reinit\_error** – If true, Ray suppresses errors from calling ray.init() a second time. Ray won't be restarted.
- **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 hugetlfs support. Requires plasma\_directory.
- **include\_java** – Boolean flag indicating whether or not to enable java workers.

- **include\_webui** – Boolean flag indicating whether or not to start the web UI for the Ray dashboard, which displays the status of the Ray cluster. If this argument is None, then the UI will be started if the relevant dependencies are present.
- **webui\_host** – The host to bind the web UI server to. Can either be localhost (127.0.0.1) or 0.0.0.0 (available from all interfaces). By default, this is set to localhost to prevent access from external machines.
- **job\_id** – The ID of this job.
- **configure\_logging** – True (default) if configuration of logging is allowed here. Otherwise, the user may want to configure it separately.
- **logging\_level** – Logging level, defaults to logging.INFO. Ignored unless “configure\_logging” is true.
- **logging\_format** – Logging format, defaults to string containing a timestamp, filename, line number, and message. See the source file ray\_constants.py for details. Ignored unless “configure\_logging” is true.
- **plasma\_store\_socket\_name (str)** – If provided, specifies the socket name used by the plasma store.
- **raylet\_socket\_name (str)** – If provided, specifies the socket path used by the raylet process.
- **temp\_dir (str)** – If provided, specifies the root temporary directory for the Ray process. Defaults to an OS-specific conventional location, e.g., “/tmp/ray”.
- **load\_code\_from\_local** – Whether code should be loaded from a local module or from the GCS.
- **use\_pickle** – Deprecated.
- **\_internal\_config (str)** – JSON configuration for overriding RayConfig defaults. For testing purposes ONLY.
- **lru\_evict (bool)** – If True, when an object store is full, it will evict objects in LRU order to make more space and when under memory pressure, ray.UnreconstructableError may be thrown. If False, then reference counting will be used to decide which objects are safe to evict and when under memory pressure, ray.ObjectStoreFullError may be thrown.

**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:
```

(continues on next page)

(continued from previous page)

```
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\_RECONSTRUCTION).
- **max\_retries**: Only for *remote functions*. This specifies the maximum number of times that the remote function should be rerun when the worker process executing it crashes unexpectedly. The minimum valid value is 0, the default is 4 (default), and the maximum valid value is ray(ray\_constants.INFINITE\_RECONSTRUCTION).

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:
    def method(self):
        return 1
```

Remote task and actor objects returned by @ray.remote can also be dynamically modified with the same arguments as above using .options() as follows:

```
@ray.remote(num_gpus=1, max_calls=1, num_return_vals=2)
def f():
    return 1, 2
g = f.options(num_gpus=2, max_calls=None)

@ray.remote(num_cpus=2, resources={"CustomResource": 1})
class Foo:
    def method(self):
        return 1
Bar = Foo.options(num_cpus=1, resources=None)
```

Running remote actors will be terminated when the actor handle to them in Python is deleted, which will cause them to complete any outstanding work and then shut down. If you want to kill them immediately, you can also call ray.kill(actor).

`ray.get (object_ids, timeout=None)`

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.

This method will issue a warning if it's running inside an async context, you can use `await object_id` instead of `ray.get(object_id)`. For a list of object IDs, you can use `await asyncio.gather(*object_ids)`.

**Parameters**

- **object\_ids** – Object ID of the object to get or a list of object IDs to get.
- **timeout** (*Optional [float]*) – The maximum amount of time in seconds to wait before returning.

**Returns** A Python object or a list of Python objects.

**Raises**

- **RayTimeoutError** – A RayTimeoutError is raised if a timeout is set and the get takes longer than timeout to return.
- **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)`

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

If timeout is set, the function returns either when the requested number of IDs are ready or when the timeout is reached, whichever occurs first. If it is not set, the function simply waits until that number of objects is ready and returns that exact number of object IDs.

This method returns two lists. The first list consists of object IDs that correspond to objects that are available in the object store. The second list corresponds to the rest of the object IDs (which may or may not be ready).

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

This method will issue a warning if it's running inside an async context. Instead of `ray.wait(object_ids)`, you can use `await asyncio.wait(object_ids)`.

**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, weakref=False)`

Store an object in the object store.

The object may not be evicted while a reference to the returned ID exists.

**Parameters**

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

- **weakref** – If set, allows the object to be evicted while a reference to the returned ID exists. You might want to set this if putting a lot of objects that you might not need in the future. It allows Ray to more aggressively reclaim memory.

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

`ray.kill(actor)`

Kill an actor forcefully.

This will interrupt any running tasks on the actor, causing them to fail immediately. Any atexit handlers installed in the actor will still be run.

If you want to kill the actor but let pending tasks finish, you can call `actor.__ray_terminate__.remote()` instead to queue a termination task.

In both cases, the worker is actually killed, but it will be restarted by Ray.

If this actor is reconstructable, an attempt will be made to reconstruct it.

**Parameters** `actor` (*ActorHandle*) – Handle to the actor to kill.

`ray.cancel(object_id, force=False)`

Cancels a locally-submitted task according to the following conditions.

If the specified task is pending execution, it will not be executed. If the task is currently executing, the behavior depends on the `force` flag. When `force=False`, a KeyboardInterrupt will be raised in Python and when `force=True`, the executing task will immediately exit. If the task is already finished, nothing will happen.

Only non-actor tasks can be canceled. Canceled tasks will not be retried (max\_retries will not be respected).

Calling `ray.get` on a canceled task will raise a `RayCancellationError`.

**Parameters**

- **object\_id** (*ObjectID*) – ObjectID returned by the task that should be canceled.
- **force** (*boolean*) – Whether to force-kill a running task by killing the worker that is running the task.

**Raises** `ValueError` – This is also raised for actor tasks, already completed tasks, and non-locally submitted tasks.

`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(existing_interpreter=False)
```

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.

**Parameters** `existing_interpreter` (`bool`) – True if this is called by the atexit hook and false otherwise. If we are exiting the interpreter, we will wait a little while to print any extra error messages.

```
ray.register_custom_serializer(cls, serializer, deserializer, use_pickle=False, use_dict=False,  
                               class_id=None)
```

Registers custom functions for efficient object serialization.

The serializer and deserializer are used when transferring objects of `cls` across processes and nodes. This can be significantly faster than the Ray default fallbacks. Wraps `register_custom_serializer` underneath.

#### Parameters

- `cls` (`type`) – The class that ray should use this custom serializer for.
- `serializer` – The custom serializer that takes in a `cls` instance and outputs a serialized representation. `use_pickle` and `use_dict` must be False if provided.
- `deserializer` – The custom deserializer that takes in a serialized representation of the `cls` and outputs a `cls` instance. `use_pickle` and `use_dict` must be False if provided.
- `use_pickle` – Deprecated.
- `use_dict` – Deprecated.
- `class_id` (`str`) – Unique ID of the class. Autogenerated if None.

```
ray.profile(event_type, extra_data=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:
    @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.

### 5.8.1 Inspect the Cluster State

`ray.nodes()`

Get a list of the nodes in the cluster.

**Returns** Information about the Ray clients in the cluster.

`ray.objects(object_id=None)`

Fetch and parse the object table info for one or more object IDs.

**Parameters** `object_id` – An object ID to fetch information about. If this is None, then the entire object table is fetched.

**Returns** Information from the object table.

`ray.timeline(filename=None)`

Return a list of profiling events that can viewed as a timeline.

To view this information as a timeline, simply dump it as a json file by passing in “filename” or using using json.dump, and then load go to chrome://tracing in the Chrome web browser and load the dumped file.

**Parameters** `filename` – If a filename is provided, the timeline is dumped to that file.

**Returns**

**If filename is not provided, this returns a list of profiling events.** Each profile event is a dictionary.

`ray.object_transfer_timeline(filename=None)`

Return a list of transfer events that can viewed as a timeline.

To view this information as a timeline, simply dump it as a json file by passing in “filename” or using using json.dump, and then load go to chrome://tracing in the Chrome web browser and load the dumped file. Make sure to enable “Flow events” in the “View Options” menu.

**Parameters** `filename` – If a filename is provided, the timeline is dumped to that file.

**Returns**

**If filename is not provided, this returns a list of profiling events.** Each profile event is a dictionary.

`ray.cluster_resources()`

Get the current total cluster resources.

Note that this information can grow stale as nodes are added to or removed from the cluster.

**Returns**

A dictionary mapping resource name to the total quantity of that resource in the cluster.

```
ray.available_resources()
```

Get the current available cluster resources.

This is different from *cluster\_resources* in that this will return idle (available) resources rather than total resources.

Note that this information can grow stale as tasks start and finish.

#### Returns

A dictionary mapping resource name to the total quantity of that resource in the cluster.

```
ray.errors(all_jobs=False)
```

Get error messages from the cluster.

**Parameters** `all_jobs` – False if we should only include error messages for this specific job, or True if we should include error messages for all jobs.

#### Returns

Error messages pushed from the cluster. This will be a single list if `all_jobs` is False, or a dictionary mapping from job ID to a list of error messages for that job if `all_jobs` is True.

## 5.8.2 Experimental APIs

```
ray.experimental.get(object_ids)
```

Get a single or a collection of remote objects from the object store.

This method is identical to *ray.get* except it adds support for tuples, ndarrays and dictionaries.

**Parameters** `object_ids` – Object ID of the object to get, a list, tuple, ndarray of object IDs to get or a dict of {key: object ID}.

**Returns** object}.

**Return type** A Python object, a list of Python objects or a dict of {key

```
ray.experimental.wait(object_ids, num_returns=1, timeout=None)
```

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

This method is identical to *ray.wait* except it adds support for tuples and ndarrays.

#### Parameters

- `object_ids` (*List[ObjectID]*, *Tuple[ObjectID]*, *np.array(ObjectID)*) – List like 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.experimental.set_resource(resource_name, capacity, client_id=None)
```

Set a resource to a specified capacity.

This creates, updates or deletes a custom resource for a target clientId. If the resource already exists, its capacity is updated to the new value. If the capacity is set to 0, the resource is deleted. If ClientID is not specified or set to None, the resource is created on the local client where the actor is running.

**Parameters**

- **resource\_name** (*str*) – Name of the resource to be created
- **capacity** (*int*) – Capacity of the new resource. Resource is deleted if capacity is 0.
- **client\_id** (*str*) – The ClientId of the node where the resource is to be set.

**Returns** None**Raises** `ValueError` – This exception is raised when a non-negative capacity is specified.

### 5.8.3 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>  
same as --address
- address** <address>  
the address to use for Ray
- 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
- memory** <memory>  
The amount of memory (in bytes) to make available to workers. By default, this is set to the available memory on the node.
- 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-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

**--include-webui** <include\_webui>  
provide this argument if the UI should be started

**--webui-host** <webui\_host>  
The host to bind the web UI server to. Can either be localhost (127.0.0.1) or 0.0.0.0 (available from all interfaces). By default, this is set to localhost to prevent access from external machines.

**--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.

**--load-code-from-local**  
Specify whether load code from local file or GCS serialization.

## ray stop

```
ray stop [OPTIONS]
```

### Options

#### **-f, --force**

If set, ray will send SIGKILL instead of SIGTERM.

#### **-v, --verbose**

If set, ray prints out more information about processes to kill.

## 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.

#### --keep-min-workers

Retain the minimal amount of workers specified in the config.

#### -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

#### --docker

Runs command in the docker container specified in cluster\_config.

#### --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.

#### -p, --port-forward <port\_forward>

Port to forward. Use this multiple times to forward multiple ports.

## Arguments

**CLUSTER\_CONFIG\_FILE**

Required argument

**CMD**

Required argument

## ray submit

Uploads and runs a script on the specified cluster.

The script is automatically synced to the following location:

```
os.path.join("~", os.path.basename(script))
```

**Example:**

```
>>> ray submit [CLUSTER.YAML] experiment.py -- --smoke-test
```

```
ray submit [OPTIONS] CLUSTER_CONFIG_FILE SCRIPT [SCRIPT_ARGS]...
```

## Options

**--docker**

Runs command in the docker container specified in cluster\_config.

**--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.

**-p, --port-forward <port\_forward>**

Port to forward. Use this multiple times to forward multiple ports.

**--args <args>**

(deprecated) Use ‘–arg1 –arg2’ for script args.

## Arguments

### **CLUSTER\_CONFIG\_FILE**

Required argument

### **SCRIPT**

Required argument

### **SCRIPT\_ARGS**

Optional argument(s)

## ray attach

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

## Options

### **--start**

Start the cluster if needed.

### **--screen**

Run the command in screen.

### **--tmux**

Run the command in tmux.

### **-n, --cluster-name <cluster\_name>**

Override the configured cluster name.

### **-N, --new**

Force creation of a new screen.

### **-p, --port-forward <port\_forward>**

Port to forward. Use this multiple times to forward multiple ports.

## 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

### ray stack

```
ray stack [OPTIONS]
```

### ray stat

```
ray stat [OPTIONS]
```

## Options

**--address** <address>  
Override the address to connect to.

### ray memory

```
ray memory [OPTIONS]
```

## Options

**--address** <address>  
Override the address to connect to.

### ray globalgc

```
ray globalgc [OPTIONS]
```

## Options

**--address** <address>  
Override the address to connect to.

## ray timeline

```
ray timeline [OPTIONS]
```

### Options

**--address** <address>

Override the redis address to connect to.

## 5.9 Tune: Scalable Hyperparameter Tuning



Tune is a Python library for experiment execution and hyperparameter tuning at any scale. Core features:

- Launch a multi-node *distributed hyperparameter sweep* in less than 10 lines of code.
- Supports any machine learning framework, *including PyTorch, XGBoost, MXNet, and Keras*.
- Natively integrates with optimization libraries such as `HyperOpt`, `Bayesian Optimization`, and `Facebook Ax`.
- Choose among scalable algorithms such as Population Based Training (PBT), Vizier's Median Stopping Rule, `HyperBand/ASHA`.
- Visualize results with `TensorBoard`.

**Want to get started?** Head over to the *60 second Tune tutorial*.

## 5.9.1 Quick Start

To run this example, install the following: `pip install 'ray[tune]' torch torchvision`.

This example runs a small grid search to train a convolutional neural network using PyTorch and Tune.

```
import torch.optim as optim
from ray import tune
from ray.tune.examples.mnist_pytorch import get_data_loaders, ConvNet, train, test

def train_mnist(config):
    train_loader, test_loader = get_data_loaders()
    model = ConvNet()
    optimizer = optim.SGD(model.parameters(), lr=config["lr"])
    for i in range(10):
        train(model, optimizer, train_loader)
        acc = test(model, test_loader)
        tune.track.log(mean_accuracy=acc)

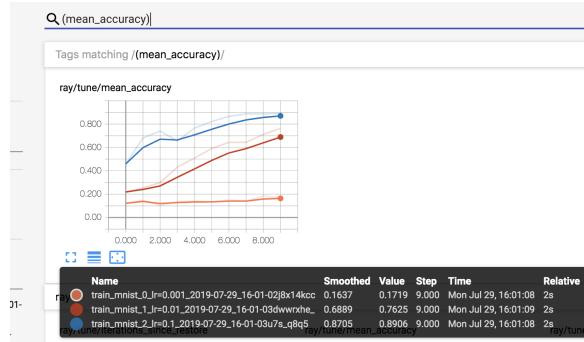
analysis = tune.run(
    train_mnist, config={"lr": tune.grid_search([0.001, 0.01, 0.1])})

print("Best config: ", analysis.get_best_config(metric="mean_accuracy"))

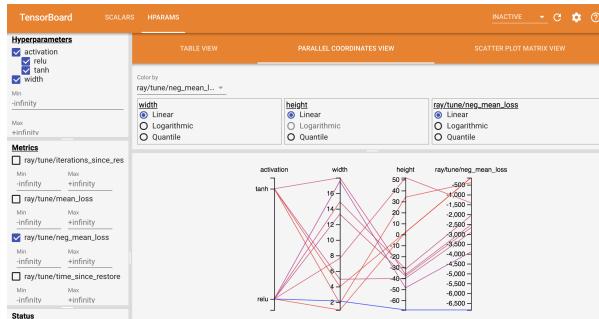
# Get a dataframe for analyzing trial results.
df = analysis.dataframe()
```

If TensorBoard is installed, automatically visualize all trial results:

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



If using TF2 and TensorBoard, Tune will also automatically generate TensorBoard HPrams output:



---

**Tip:** Join the Ray community slack to discuss Ray Tune (and other Ray libraries)!

---

## 5.9.2 Guides/Materials

Here are some reference materials for Tune:

- [Tune Tutorials, Guides, and Examples](#)
- [Code](#): GitHub repository for Tune

Below are some blog posts and talks about Tune:

- [blog] [Tune: a Python library for fast hyperparameter tuning at any scale](#)
- [blog] [Cutting edge hyperparameter tuning with Ray Tune](#)
- [blog] [Simple hyperparameter and architecture search in tensorflow with Ray Tune](#)
- [slides] [Talk given at RISECamp 2019](#)
- [video] [Talk given at RISECamp 2018](#)
- [video] [A Guide to Modern Hyperparameter Optimization \(PyData LA 2019\) \(slides\)](#)

## 5.9.3 Open Source Projects using Tune

Here are some of the popular open source repositories and research projects that leverage Tune. Feel free to submit a pull-request adding (or requesting a removal!) of a listed project.

- [Softlearning](#): Softlearning is a reinforcement learning framework for training maximum entropy policies in continuous domains. Includes the official implementation of the Soft Actor-Critic algorithm.
- [Flambe](#): An ML framework to accelerate research and its path to production. See [flambe.ai](#).
- [Population Based Augmentation](#): Population Based Augmentation (PBA) is a algorithm that quickly and efficiently learns data augmentation functions for neural network training. PBA matches state-of-the-art results on CIFAR with one thousand times less compute.
- [Fast AutoAugment by Kakao](#): Fast AutoAugment (Accepted at NeurIPS 2019) learns augmentation policies using a more efficient search strategy based on density matching.
- [Allentune](#): Hyperparameter Search for AllenNLP from AllenAI.
- [machinable](#): A modular configuration system for machine learning research. See [machinable.org](#).

## 5.9.4 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},
    journal={arXiv preprint arXiv:1807.05118},
    year={2018}
}
```

## 5.10 Tutorials, User Guides, Examples

In this section, you can find material on how to use Tune and its various features. If any of the materials is out of date or broken, or if you'd like to add an example to this page, feel free to raise an issue on our Github repository.

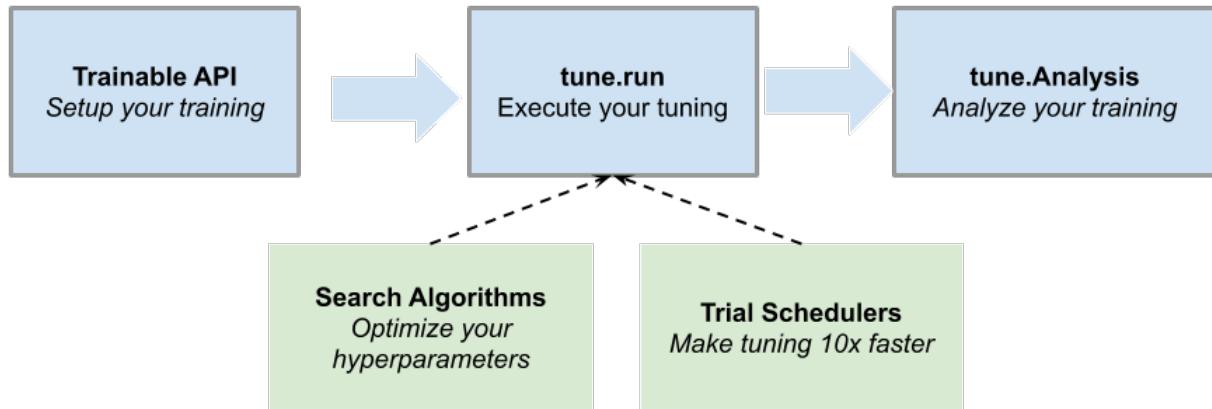
### 5.10.1 Tutorials

Take a look at any of the below tutorials to get started with Tune.

#### Tune in 60 Seconds

Let's quickly walk through the key concepts you need to know to use Tune. In this guide, we'll be covering the following:

- *Trainables*
- *tune.run*
- *Search Algorithms*
- *Trial Schedulers*
- *Analysis*



#### Trainables

Tune will optimize your training process using the *Trainable API*. To start, let's try to maximize this objective function:

```
def objective(x, a, b):
    return a * (x ** 0.5) + b
```

Here's an example of specifying the objective function using *the function-based Trainable API*:

```
def trainable(config):
    # config (dict): A dict of hyperparameters.

    for x in range(20):
```

(continues on next page)

(continued from previous page)

```
score = objective(x, config["a"], config["b"])

tune.track.log(score=score) # This sends the score to Tune.
```

Now, there's two Trainable APIs - one being the *function-based API* that we demonstrated above.

The other is a *class-based API* that enables *checkpointing and pausing*. Here's an example of specifying the objective function using the *class-based API*:

```
from ray import tune

class Trainable(tune.Trainable):
    def _setup(self, config):
        # config (dict): A dict of hyperparameters
        self.x = 0
        self.a = config["a"]
        self.b = config["b"]

    def _train(self): # This is called iteratively.
        score = objective(self.x, self.a, self.b)
        self.x += 1
        return {"score": score}
```

---

**Tip:** Do not use `tune.track.log` within a Trainable class.

---

See the documentation: *Training (tune.Trainable, tune.track)* and *examples*.

## tune.run

Use `tune.run` execute hyperparameter tuning using the core Ray APIs. This function manages your experiment and provides many features such as *logging*, *checkpointing*, and *early stopping*.

```
# Pass in a Trainable class or function to tune.run.
tune.run(trainable)
```

This function will report status on the command line until all trials stop (each trial is one instance of a *Trainable*):

```
== Status ==
Memory usage on this node: 11.4/16.0 GiB
Using FIFO scheduling algorithm.
Resources requested: 1/12 CPUs, 0/0 GPUs, 0.0/3.17 GiB heap, 0.0/1.07 GiB objects
Result logdir: /Users/foo/ray_results/myexp
Number of trials: 1 (1 RUNNING)
+-----+-----+-----+-----+
| Trial name      | status   | loc          |       a |       b | score |
+-----+-----+-----+-----+
| total time (s) | iter   |
+-----+-----+-----+-----+
| MyTrainable_a826033a | RUNNING | 10.234.98.164:31115 | 0.303706 | 0.0761 | 0.1289 |
| 7.54952 | 15 |
+-----+-----+-----+-----+
```

You can also easily run 10 trials. Tune automatically *determines how many trials will run in parallel*.

```
tune.run(trainable, num_samples=10)
```

Finally, you can randomly sample or grid search hyperparameters via Tune's *search space API*:

```
space = {"x": tune.uniform(0, 1)}
tune.run(my_trainable, config=space, num_samples=10)
```

See more documentation: [tune.run](#).

## Search Algorithms

To optimize the hyperparameters of your training process, you will want to use a *Search Algorithm* which will help suggest better hyperparameters.

```
# Be sure to first run `pip install hyperopt`

import hyperopt as hp
from ray.tune.suggest.hyperopt import HyperOptSearch

# Create a HyperOpt search space
space = {
    "a": hp.uniform("a", 0, 1),
    "b": hp.uniform("b", 0, 20)

    # Note: Arbitrary HyperOpt search spaces should be supported!
    # "foo": hp.lognormal("foo", 0, 1))
}

# Specify the search space and maximize score
hyperopt = HyperOptSearch(space, metric="score", mode="max")

# Execute 20 trials using HyperOpt and stop after 20 iterations
tune.run(
    trainable,
    search_alg=hyperopt,
    num_samples=20,
    stop={"training_iteration": 20}
)
```

Tune has SearchAlgorithms that integrate with many popular **optimization** libraries, such as *Nevergrad* and *Hyperopt*.

See the documentation: [Search Algorithms \(tune.suggest\)](#).

## Trial Schedulers

In addition, you can make your training process more efficient by using a *Trial Scheduler*.

Trial Schedulers can stop/pause/tweak the hyperparameters of running trials, making your hyperparameter tuning process much faster.

```
from ray.tune.schedulers import HyperBandScheduler

# Create HyperBand scheduler and maximize score
```

(continues on next page)

(continued from previous page)

```
hyperband = HyperBandScheduler(metric="score", mode="max")

# Execute 20 trials using HyperBand using a search space
configs = {"a": tune.uniform(0, 1), "b": tune.uniform(0, 1)}

tune.run(
    MyTrainableClass,
    config=configs,
    num_samples=20,
    scheduler=hyperband
)
```

*Population-based Training* and *HyperBand* are examples of popular optimization algorithms implemented as Trial Schedulers.

Unlike **Search Algorithms**, *Trial Scheduler* do not select which hyperparameter configurations to evaluate. However, you can use them together.

See the documentation: *Trial Schedulers (tune.schedulers)*.

## Analysis

`tune.run` returns an *Analysis* object which has methods you can use for analyzing your training.

```
analysis = tune.run(trainable, search_alg=algo, stop={"training_iteration": 20})

# Get the best hyperparameters
best_hyperparameters = analysis.get_best_config()
```

This object can also retrieve all training runs as dataframes, allowing you to do ad-hoc data analysis over your results.

```
# Get a dataframe for the max score seen for each trial
df = analysis.dataframe(metric="score", mode="max")
```

## What's Next?

Now that you have a working understanding of Tune, check out:

- *Tune Guides and Examples*: Examples and templates for using Tune with your preferred machine learning library.
- *A Basic Tune Tutorial*: A simple tutorial that walks you through the process of setting up a Tune experiment.
- *Tune User Guide*: A comprehensive overview of Tune's features.

## Further Questions or Issues?

Reach out to us if you have any questions or issues or feedback through the following channels:

1. [StackOverflow](#): For questions about how to use Ray.
2. [GitHub Issues](#): For bug reports and feature requests.

## A Basic Tune Tutorial

This tutorial will walk you through the following process to setup a Tune experiment using Pytorch. Specifically, we'll leverage ASHA and Bayesian Optimization (via HyperOpt) via the following steps:

1. Integrating Tune into your workflow
2. Specifying a TrialScheduler
3. Adding a SearchAlgorithm
4. Getting the best model and analyzing results

**Note:** To run this example, you will need to install the following:

```
$ pip install ray torch torchvision
```

We first run some imports:

```
import numpy as np
import torch
import torch.optim as optim
from torchvision import datasets

from ray import tune
from ray.tune import track
from ray.tune.schedulers import ASHAScheduler
from ray.tune.examples.mnist_pytorch import get_data_loaders, ConvNet, train, test
```

Below, we have some boiler plate code for a PyTorch training function.

```
def train_mnist(config):
    model = ConvNet()
    train_loader, test_loader = get_data_loaders()
    optimizer = optim.SGD(
        model.parameters(), lr=config["lr"], momentum=config["momentum"])
    for i in range(10):
        train(model, optimizer, train_loader)
        acc = test(model, test_loader)
        track.log(mean_accuracy=acc)
        if i % 5 == 0:
            # This saves the model to the trial directory
            torch.save(model, "./model.pth")
```

Notice that there's a couple helper functions in the above training script. You can take a look at these functions in the imported module `examples/mnist_pytorch`; there's no black magic happening. For example, `train` is simply a for loop over the data loader.

```
EPOCH_SIZE = 20

def train(model, optimizer, train_loader):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if batch_idx * len(data) > EPOCH_SIZE:
            return
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Let's run 1 trial, randomly sampling from a uniform distribution for learning rate and momentum.

```
search_space = {
    "lr": tune.sample_from(lambda spec: 10**(-10 * np.random.rand())),
    "momentum": tune.uniform(0.1, 0.9)
}

# Uncomment this to enable distributed execution
# `ray.init(address=...)`

analysis = tune.run(train_mnist, config=search_space)
```

We can then plot the performance of this trial.

```
dfs = analysis.trial_dataframes
[d.mean_accuracy.plot() for d in dfs.values()]
```

---

**Important:** Tune will automatically run parallel trials across all available cores/GPUs on your machine or cluster. To limit the number of cores that Tune uses, you can call `ray.init(num_cpus=<int>, num_gpus=<int>)` before `tune.run`.

---

## Early Stopping with ASHA

Let's integrate a Trial Scheduler to our search - ASHA, a scalable algorithm for principled early stopping.

How does it work? On a high level, it terminates trials that are less promising and allocates more time and resources to more promising trials. See [this blog post](#) for more details.

We can afford to **increase the search space by 5x**, by adjusting the parameter `num_samples`. See [Tune Trial Schedulers](#) for more details of available schedulers and library integrations.

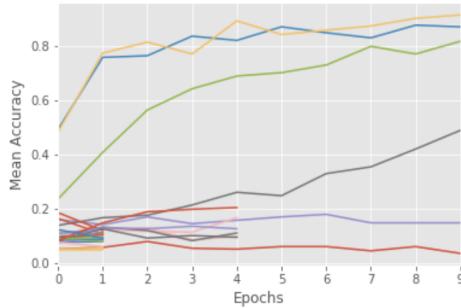
```
analysis = tune.run(
    train_mnist,
    num_samples=30,
    scheduler=ASHAScheduler(metric="mean_accuracy", mode="max"),
    config=search_space)

# Obtain a trial dataframe from all run trials of this `tune.run` call.
dfs = analysis.trial_dataframes
```

You can run the below in a Jupyter notebook to visualize trial progress.

```
# Plot by epoch
ax = None # This plots everything on the same plot
for d in dfs.values():
    ax = d.mean_accuracy.plot(ax=ax, legend=False)
```

```
[25]: # Plot by epoch
ax = None
for d in dfs.values():
    ax = d.mean_accuracy.plot(ax=ax, legend=False)
ax.set_xlabel("Epochs")
ax.set_ylabel("Mean Accuracy");
```



You can also use Tensorboard for visualizing results.

```
$ tensorboard --logdir {logdir}
```

## Search Algorithms in Tune

With Tune you can combine powerful hyperparameter search libraries such as [HyperOpt](#) and [Ax](#) with state-of-the-art algorithms such as HyperBand without modifying any model training code. Tune allows you to use different search algorithms in combination with different trial schedulers. See [Tune Search Algorithms](#) for more details of available algorithms and library integrations.

```
from hyperopt import hp
from ray.tune.suggest.hyperopt import HyperOptSearch

space = {
    "lr": hp.loguniform("lr", 1e-10, 0.1),
    "momentum": hp.uniform("momentum", 0.1, 0.9),
}

hyperopt_search = HyperOptSearch(space, metric="mean_accuracy", mode="max")

analysis = tune.run(train_mnist, num_samples=10, search_alg=hyperopt_search)
```

## Evaluate your model

You can evaluate best trained model using the Analysis object to retrieve the best model:

```
import os

df = analysis.dataframe()
logdir = analysis.get_best_logdir("mean_accuracy", mode="max")
model = torch.load(os.path.join(logdir, "model.pth"))
```

## Next Steps

Take a look at the [Tune User Guide](#) for a more comprehensive overview of Tune's features.

### 5.10.2 User Guides

These pages will demonstrate the various features and configurations of Tune.

#### Tune User Guide

**Warning:** Before you continue, be sure to have read [Tune in 60 Seconds](#).

This document provides an overview of the core concepts as well as some of the configurations for running Tune.

- [Parallelism / GPUs](#)
- [Search Space \(Grid/Random\)](#)
- [Reporting Metrics](#)
- [Checkpointing](#)
- [Handling Large Datasets](#)
- [Stopping Trials](#)
- [Logging/Tensorboard](#)
- [Console Output](#)
- [Uploading Results](#)
- [Debugging](#)
- [Stopping after the first failure](#)
- [Further Questions or Issues?](#)

## Parallelism / GPUs

**Tip:** To run everything sequentially, use [Ray Local Mode](#).

Parallelism is determined by `resources_per_trial` (defaulting to 1 CPU, 0 GPU per trial) and the resources available to Tune (`ray.cluster_resources()`).

Tune will allocate the specified GPU and CPU from `resources_per_trial` to each individual trial. A trial will not be scheduled unless at least that amount of resources is available, preventing the cluster from being overloaded.

By default, Tune automatically runs N concurrent trials, where N is the number of CPUs (cores) on your machine.

```
# If you have 4 CPUs on your machine, this will run 4 concurrent trials at a time.
tune.run(trainable, num_samples=10)
```

You can override this parallelism with `resources_per_trial`:

```
# If you have 4 CPUs on your machine, this will run 2 concurrent trials at a time.
tune.run(trainable, num_samples=10, resources_per_trial={"cpu": 2})

# If you have 4 CPUs on your machine, this will run 1 trial at a time.
tune.run(trainable, num_samples=10, resources_per_trial={"cpu": 4})

# Fractional values are also supported, (i.e., {"cpu": 0.5}).
tune.run(trainable, num_samples=10, resources_per_trial={"cpu": 0.5})
```

To leverage GPUs, you must set `gpu` in `resources_per_trial`. This will automatically set `CUDA_VISIBLE_DEVICES` for each trial.

```
# If you have 8 GPUs, this will run 8 trials at once.
tune.run(trainable, num_samples=10, resources_per_trial={"gpu": 1})

# If you have 4 CPUs on your machine and 1 GPU, this will run 1 trial at a time.
tune.run(trainable, num_samples=10, resources_per_trial={"cpu": 2, "gpu": 1})
```

You can find an example of this in the [Keras MNIST](#) example.

**Warning:** If ‘gpu’ is not set, `CUDA_VISIBLE_DEVICES` environment variable will be set as empty, disallowing GPU access.

To attach to a Ray cluster, simply run `ray.init` before `tune.run`:

```
# Connect to an existing distributed Ray cluster
ray.init(address=<ray_address>)
tune.run(trainable, num_samples=100, resources_per_trial={"cpu": 2, "gpu": 1})
```

## Search Space (Grid/Random)

**Warning:** If you use a Search Algorithm, you will need to use a different search space API.

You can specify a grid search or random search via the dict passed into `tune.run(config=)`.

```
parameters = {
    "qux": tune.sample_from(lambda spec: 2 + 2),
    "bar": tune.grid_search([True, False]),
    "foo": tune.grid_search([1, 2, 3]),
    "baz": "asd", # a constant value
}

tune.run(trainable, config=parameters)
```

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.

```
# num_samples=10 repeats the 3x3 grid search 10 times, for a total of 90 trials
tune.run(
    my_trainable,
    name="my_trainable",
    config={
        "alpha": tune.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
)
```

Read about this in the [Grid/Random Search API](#) page.

## Reporting Metrics

You can log arbitrary values and metrics in both training APIs:

```
def trainable(config):
    num_epochs = 100
    for i in range(num_epochs):
        accuracy = model.train()
        metric_1 = f(model)
        metric_2 = model.get_loss()
        tune.track.log(acc=accuracy, metric_foo=random_metric_1, bar=metric_2)

class Trainable(tune.Trainable):
    ...

    def _train(self): # this is called iteratively
        accuracy = self.model.train()
```

(continues on next page)

(continued from previous page)

```

metric_1 = f(self.model)
metric_2 = self.model.get_loss()
# don't call track.log here!
return dict(acc=accuracy, metric_foo=random_metric_1, bar=metric_2)

```

During training, Tune will automatically log the below metrics in addition to the user-provided values. All of these can be used as stopping conditions or passed as a parameter to Trial Schedulers/Search Algorithms.

```

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

# (Optional) Enum for user controlled checkpoint
SHOULD_CHECKPOINT = "should_checkpoint"

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

# (Auto-filled) The auto-assigned id of the trial.
TRIAL_ID = "trial_id"

# (Auto-filled) The auto-assigned id of the trial.
EXPERIMENT_TAG = "experiment_tag"

# (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"

# (Optional) Mean reward for current training iteration
EPISODE_REWARD_MEAN = "episode_reward_mean"

# (Optional) Mean loss for training iteration
MEAN_LOSS = "mean_loss"

# (Optional) Mean accuracy for training iteration
MEAN_ACCURACY = "mean_accuracy"

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

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

# Number of timesteps in this iteration.
TIMESTEPS_THIS_ITER = "timesteps_this_iter"

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

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

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

```

(continues on next page)

(continued from previous page)

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

## Checkpointing

When running a hyperparameter search, Tune can automatically and periodically save/checkpoint your model. Checkpointing is used for

- saving a model throughout training
- fault-tolerance when using pre-emptible machines.
- Pausing trials when using Trial Schedulers such as HyperBand and PBT.

To enable checkpointing, you must implement a [Trainable class](#) (the function-based API are not checkpointable, since they never return control back to their caller).

Checkpointing assumes that the model state will be saved to disk on whichever node the Trainable is running on. You can checkpoint with three different mechanisms: manually, periodically, and at termination.

**Manual Checkpointing:** A custom Trainable can manually trigger checkpointing by returning `should_checkpoint: True` (or `tune.result.SHOULD_CHECKPOINT: True`) in the result dictionary of `_train`. This can be especially helpful in spot instances:

```
def _train(self):
    # training code
    result = {"mean_accuracy": accuracy}
    if detect_instance_preemption():
        result.update(should_checkpoint=True)
    return result
```

**Periodic Checkpointing:** periodic checkpointing can be used to provide fault-tolerance for experiments. This can be enabled by setting `checkpoint_freq=<int>` and `max_failures=<int>` to checkpoint trials every  $N$  iterations and recover from up to  $M$  crashes per trial, e.g.:

```
tune.run(
    my_trainable,
    checkpoint_freq=10,
    max_failures=5,
)
```

**Checkpointing at Termination:** 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=True`:

```
tune.run(
    my_trainable,
    checkpoint_freq=10,
    checkpoint_at_end=True,
    max_failures=5,
)
```

The checkpoint will be saved at a path that looks like `local_dir/exp_name/trial_name/checkpoint_x/`, where the  $x$  is the number of iterations so far when the checkpoint is saved. To restore the checkpoint, you can use the `restore` argument and specify a checkpoint file. By doing this, you can change whatever experiments' configuration such as the experiment's name, the training iteration or so:

```
# Restored previous trial from the given checkpoint
tune.run(
    "PG",
    name="RestoredExp", # The name can be different.
    stop={"training_iteration": 10}, # train 5 more iterations than previous
    restore="~/ray_results/Original/PG_<xxx>/checkpoint_5/checkpoint-5",
    config={"env": "CartPole-v0"},
)
```

## 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 import tune
from ray.tune.utils 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

tune.run(f)
```

## Stopping Trials

You can control when trials are stopped early by passing the `stop` argument to `tune.run`. This argument takes either a dictionary or a function.

If a dictionary is passed in, the keys may be any field in the return result of `tune.track.log` in the Function API or `_train()` (including the results from `_train` and auto-filled metrics).

In the example below, each trial will be stopped either when it completes 10 iterations OR when it reaches a mean accuracy of 0.98. These metrics are assumed to be **increasing**.

```
# training_iteration is an auto-filled metric by Tune.
tune.run(
    my_trainable,
    stop={"training_iteration": 10, "mean_accuracy": 0.98}
)
```

For more flexibility, you can pass in a function instead. If a function is passed in, it must take `(trial_id, result)` as arguments and return a boolean (`True` if trial should be stopped and `False` otherwise).

```
def stopper(trial_id, result):
    return result["mean_accuracy"] / result["training_iteration"] > 5

tune.run(my_trainable, stop=stopper)
```

Finally, you can implement the `Stopper` abstract class for stopping entire experiments. For example, the following example stops all trials after the criteria is fulfilled by any individual trial, and prevents new ones from starting:

```
from ray.tune import Stopper

class CustomStopper(Stopper):
    def __init__(self):
        self.should_stop = False

    def __call__(self, trial_id, result):
        if not self.should_stop and result['foo'] > 10:
            self.should_stop = True
        return self.should_stop

    def stop_all(self):
        """Returns whether to stop trials and prevent new ones from starting."""
        return self.should_stop

stopper = CustomStopper()
tune.run(my_trainable, stop=stopper)
```

Note that in the above example the currently running trials will not stop immediately but will do so once their current iterations are complete. See the [Stopper \(tune.Stopper\)](#) documentation.

## Logging/Tensorboard

Tune will log the results of each trial to a subfolder under a specified local dir, which defaults to `~/ray_results`. Tune by default will log results for Tensorboard, CSV, and JSON formats.

```
# This logs to 2 different trial folders:
# ~/ray_results/trainable_name/trial_name_1 and ~/ray_results/trainable_name/trial_
# name_2
# trainable_name and trial_name are autogenerated.
tune.run(trainable, num_samples=2)
```

Learn about how to customize logging paths and outputs: [Loggers \(tune.logger\)](#).

Tune automatically outputs Tensorboard files during `tune.run`. To visualize learning in tensorboard, install tensorboardX:

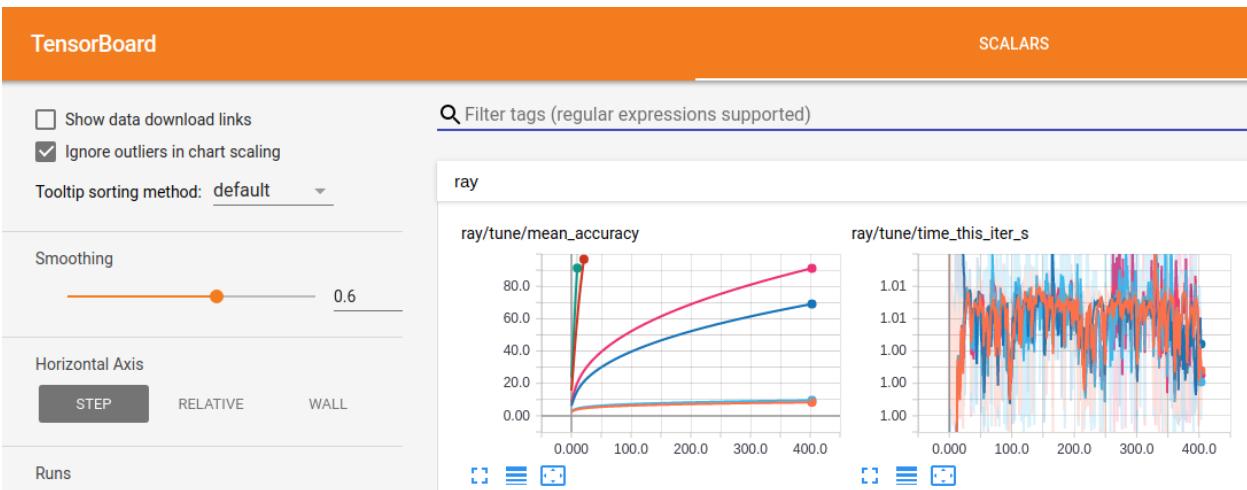
```
$ pip install tensorboardX
```

Then, after you run an experiment, you can visualize your experiment with TensorBoard by specifying the output directory of your results.

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

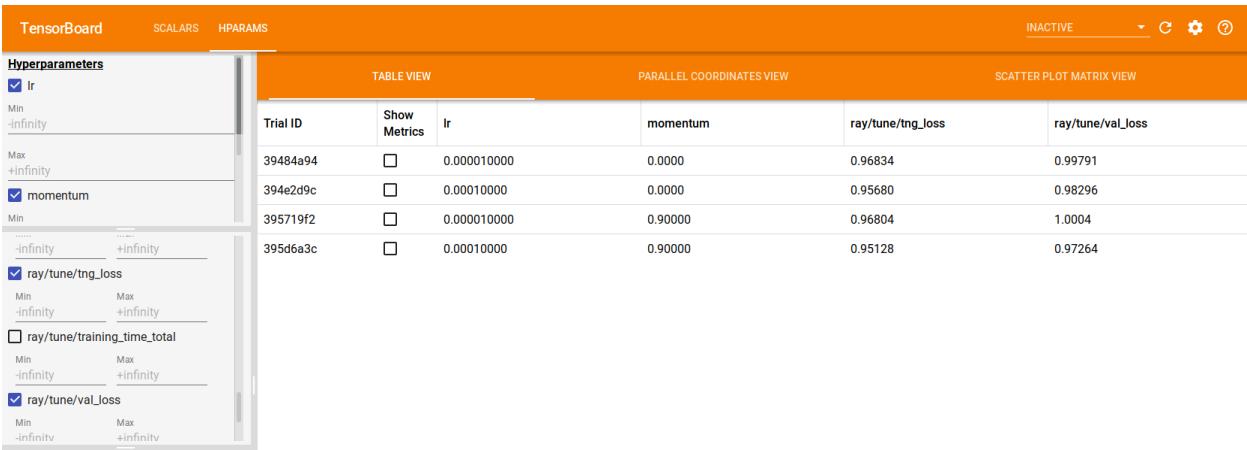
If you are running Ray on a remote multi-user cluster where you do not have sudo access, you can run the following commands to make sure tensorboard is able to write to the tmp directory:

```
$ export TMPDIR=/tmp/$USER; mkdir -p $TMPDIR; tensorboard --logdir=~/ray_results
```



If using TF2, Tune also automatically generates TensorBoard HPrams output, as shown below:

```
tune.run(
    ...,
    config={
        "lr": tune.grid_search([1e-5, 1e-4]),
        "momentum": tune.grid_search([0, 0.9])
    }
)
```



## Console Output

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).

Below is an example of the console output:

```
== Status ==
Memory usage on this node: 11.4/16.0 GiB
Using FIFO scheduling algorithm.
Resources requested: 4/12 CPUs, 0/0 GPUs, 0.0/3.17 GiB heap, 0.0/1.07 GiB objects
Result logdir: /Users/foo/ray_results/myexp
Number of trials: 4 (4 RUNNING)
+-----+-----+-----+-----+-----+
| Trial name          | status    | loc                 | param1 | param2 | acc |
| total time (s)     | iter     |                     |         |         |      |
+-----+-----+-----+-----+-----+
| MyTrainable_a826033a | RUNNING   | 10.234.98.164:31115 | 0.303706 | 0.0761 | 0.1289 |
| 7.54952             | 15       |                     |         |         |      |
| MyTrainable_a8263fc6 | RUNNING   | 10.234.98.164:31117 | 0.929276 | 0.158  | 0.4865 |
| 7.0501              | 14       |                     |         |         |      |
| MyTrainable_a8267914 | RUNNING   | 10.234.98.164:31111 | 0.068426 | 0.0319 | 0.9585 |
| 7.0477              | 14       |                     |         |         |      |
| MyTrainable_a826b7bc | RUNNING   | 10.234.98.164:31112 | 0.729127 | 0.0748 | 0.1797 |
| 7.05715             | 14       |                     |         |         |      |
+-----+-----+-----+-----+-----+
```

You can use a *Reporter* object to customize the console output.

## Uploading Results

If an upload directory is provided, Tune will automatically sync results from the `local_dir` to the given directory, natively supporting standard S3/gsutil URIs.

```
tune.run(
    MyTrainableClass,
    local_dir="~/ray_results",
    upload_dir="s3://my-log-dir"
)
```

You can customize this to specify arbitrary storages with the `sync_to_cloud` argument in `tune.run`. This argument supports either strings with the same replacement fields OR arbitrary functions.

```
tune.run(
    MyTrainableClass,
    upload_dir="s3://my-log-dir",
    sync_to_cloud=custom_sync_str_or_func,
)
```

If a string is provided, then it must include replacement fields `{source}` and `{target}`, like `s3 sync {source} {target}`. Alternatively, a function can be provided with the following signature:

```
def custom_sync_func(source, target):
    # do arbitrary things inside
    sync_cmd = "s3 {source} {target}".format(
        source=source,
        target=target)
    sync_process = subprocess.Popen(sync_cmd, shell=True)
    sync_process.wait()
```

## Debugging

By default, Tune will run hyperparameter evaluations on multiple processes. However, if you need to debug your training process, it may be easier to do everything on a single process. You can force all Ray functions to occur on a single process with `local_mode` by calling the following before `tune.run`.

```
ray.init(local_mode=True)
```

Local mode with multiple configuration evaluations will interleave computation, so it is most naturally used when running a single configuration evaluation.

## Stopping after the first failure

By default, `tune.run` will continue executing until all trials have terminated or errored. To stop the entire Tune run as soon as **any** trial errors:

```
tune.run(trainable, fail_fast=True)
```

This is useful when you are trying to setup a large hyperparameter experiment.

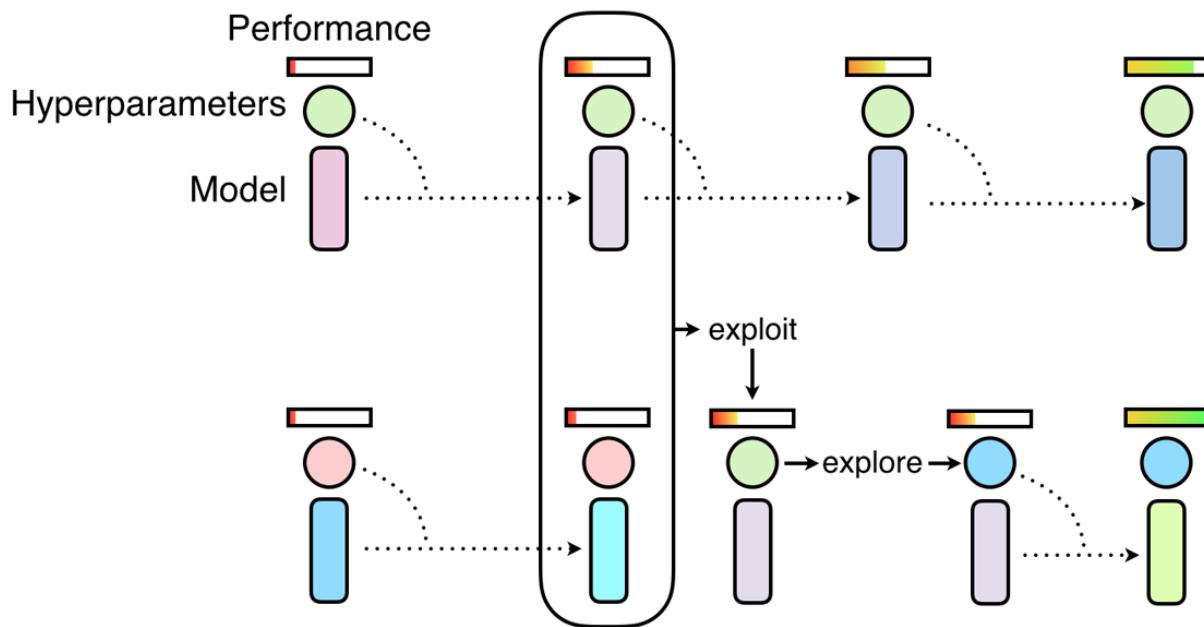
## Further Questions or Issues?

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

1. [StackOverflow](#): For questions about how to use Ray.
2. [GitHub Issues](#): For bug reports and feature requests.

## Guide to Population Based Training (PBT)

Tune includes a distributed implementation of Population Based Training (PBT) as a *scheduler*.



PBT starts by training many neural networks in parallel with random hyperparameters, using information from the rest of the population to refine these hyperparameters and allocate resources to promising models. Let's walk through how to use this algorithm.

- *Trainable API with Population Based Training*
- *DCGAN with Trainable and PBT*
  - *Visualization*

### Trainable API with Population Based Training

PBT takes its inspiration from genetic algorithms where each member of the population can exploit information from the remainder of the population. For example, a worker might copy the model parameters from a better performing worker. It can also explore new hyperparameters by changing the current values randomly.

As the training of the population of neural networks progresses, this process of exploiting and exploring is performed periodically, ensuring that all the workers in the population have a good base level of performance and also that new hyperparameters are consistently explored.

This means that PBT can quickly exploit good hyperparameters, can dedicate more training time to promising models and, crucially, can adapt the hyperparameter values throughout training, leading to automatic learning of the best configurations.

First, we define a Trainable that wraps a ConvNet model.

```
class PytorchTrainable(tune.Trainable):
    """Train a Pytorch ConvNet with Trainable and PopulationBasedTraining
```

(continues on next page)

(continued from previous page)

```

scheduler. The example reuse some of the functions in mnist_pytorch,
and is a good demo for how to add the tuning function without
changing the original training code.

"""

def _setup(self, config):
    self.train_loader, self.test_loader = get_data_loaders()
    self.model = ConvNet()
    self.optimizer = optim.SGD(
        self.model.parameters(),
        lr=config.get("lr", 0.01),
        momentum=config.get("momentum", 0.9))

def _train(self):
    train(self.model, self.optimizer, self.train_loader)
    acc = test(self.model, self.test_loader)
    return {"mean_accuracy": acc}

def _save(self, checkpoint_dir):
    checkpoint_path = os.path.join(checkpoint_dir, "model.pth")
    torch.save(self.model.state_dict(), checkpoint_path)
    return checkpoint_path

def _restore(self, checkpoint_path):
    self.model.load_state_dict(torch.load(checkpoint_path))

def _export_model(self, export_formats, export_dir):
    if export_formats == [ExportFormat.MODEL]:
        path = os.path.join(export_dir, "exported_convnet.pt")
        torch.save(self.model.state_dict(), path)
        return {export_formats[0]: path}
    else:
        raise ValueError("unexpected formats: " + str(export_formats))

def reset_config(self, new_config):
    for param_group in self.optimizer.param_groups:
        if "lr" in new_config:
            param_group["lr"] = new_config["lr"]
        if "momentum" in new_config:
            param_group["momentum"] = new_config["momentum"]

    self.config = new_config
    return True

```

The example reuses some of the functions in ray/tune/examples/mnist\_pytorch.py, and is also a good demo for how to decouple the tuning logic and original training code.

Here, we also override `reset_config`. This method is optional but can be implemented to speed up algorithms such as PBT, and to allow performance optimizations such as running experiments with `reuse_actors=True`.

Then, we define a PBT scheduler:

```

scheduler = PopulationBasedTraining(
    time_attr="training_iteration",
    metric="mean_accuracy",

```

(continues on next page)

(continued from previous page)

```

        mode="max",
        perturbation_interval=5,
        hyperparam_mutations={
            # distribution for resampling
            "lr": lambda: np.random.uniform(0.0001, 1),
            # allow perturbations within this set of categorical values
            "momentum": [0.8, 0.9, 0.99],
        })
    )
)

```

Some of the most important parameters are:

- `hyperparam_mutations` and `custom_explore_fn` are used to mutate the hyperparameters. `hyperparam_mutations` is a dictionary where each key/value pair specifies the candidates or function for a hyperparameter. `custom_explore_fn` is applied after built-in perturbations from `hyperparam_mutations` are applied, and should return config updated as needed.
- `resample_probability`: 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. Note that `resample_probability` by default is 0.25, thus hyperparameter with a distribution may go out of the specific range.

Now we can kick off the tuning process by invoking `tune.run`:

```

class CustomStopper(tune.Stopper):
    def __init__(self):
        self.should_stop = False

    def __call__(self, trial_id, result):
        max_iter = 5 if args.smoke_test else 100
        if not self.should_stop and result["mean_accuracy"] > 0.96:
            self.should_stop = True
        return self.should_stop or result["training_iteration"] >= max_iter

    def stop_all(self):
        return self.should_stop

stopper = CustomStopper()

analysis = tune.run(
    PytorchTrainble,
    name="pbt_test",
    scheduler=scheduler,
    reuse_actors=True,
    verbose=1,
    stop=stopper,
    export_formats=[ExportFormat.MODEL],
    checkpoint_score_attr="mean_accuracy",
    checkpoint_freq=5,
    keep_checkpoints_num=4,
    num_samples=4,
    config={
        "lr": tune.uniform(0.001, 1),
        "momentum": tune.uniform(0.001, 1),
    })
)

```

During the training, we can constantly check the status of the models from console log:

```

== Status ==
Memory usage on this node: 10.4/16.0 GiB
PopulationBasedTraining: 4 checkpoints, 1 perturbs
Resources requested: 4/12 CPUs, 0/0 GPUs, 0.0/3.42 GiB heap, 0.0/1.17 GiB objects
Number of trials: 4 ({'RUNNING': 4})
Result logdir: /Users/yuhao.yang/ray_results/pbt_test
+-----+-----+-----+-----+-----+
| Trial name           | status   | loc          |      lr | momentum |
|-----+-----+-----+-----+-----+
| iter | total time (s) | acc |
|-----+-----+-----+-----+-----+
| PytorchTrainble_3b42d914 | RUNNING | 30.57.180.224:49840 | 0.122032 | 0.302176 |
| 18 | 3.8689 | 0.8875 |
| PytorchTrainble_3b45091e | RUNNING | 30.57.180.224:49835 | 0.505325 | 0.628559 |
| 18 | 3.90404 | 0.134375 |
| PytorchTrainble_3b454c46 | RUNNING | 30.57.180.224:49843 | 0.490228 | 0.969013 |
| 17 | 3.72111 | 0.0875 |
| PytorchTrainble_3b458a9c | RUNNING | 30.57.180.224:49833 | 0.961861 | 0.169701 |
| 13 | 2.72594 | 0.1125 |
+-----+-----+-----+-----+-----+
|-----+-----+-----+-----+

```

In {LOG\_DIR}/{MY\_EXPERIMENT\_NAME}/, all mutations are logged in pbt\_global.txt and individual policy perturbations are recorded in pbt\_policy\_{i}.txt. Tune logs: [target trial tag, clone trial tag, target trial iteration, clone trial iteration, old config, new config] on each perturbation step.

Checking the accuracy:

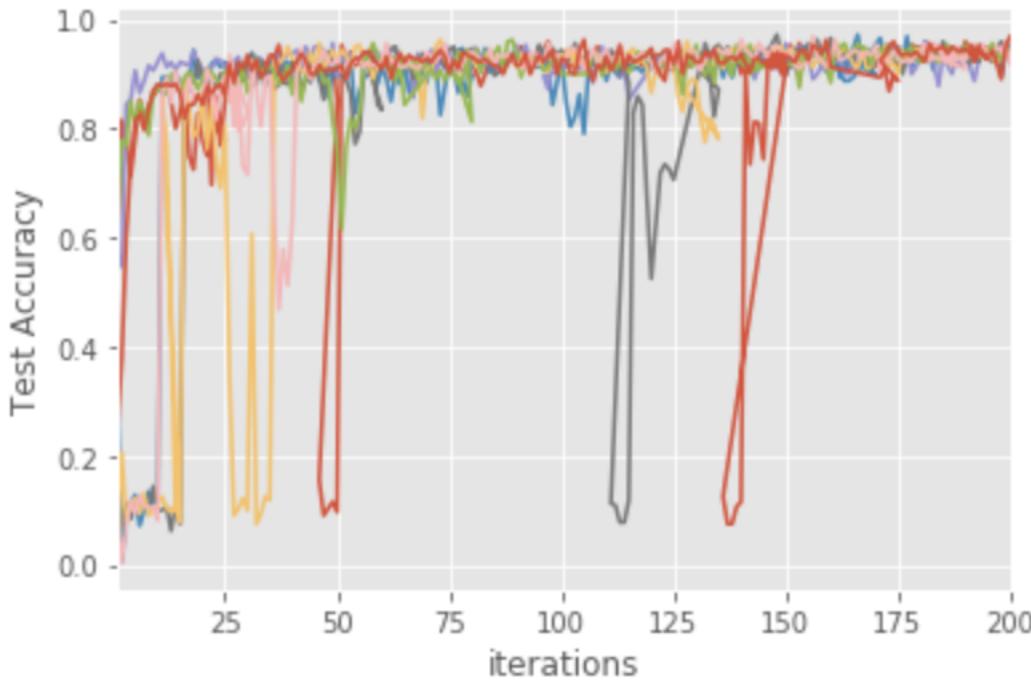
```

# Plot by wall-clock time
dfs = analysis.fetch_trial_dataframes()
# This plots everything on the same plot
ax = None
for d in dfs.values():
    ax = d.plot("training_iteration", "mean_accuracy", ax=ax, legend=False)

plt.xlabel("iterations")
plt.ylabel("Test Accuracy")

print('best config:', analysis.get_best_config("mean_accuracy"))

```



## DCGAN with Trainable and PBT

The Generative Adversarial Networks (GAN) (Goodfellow et al., 2014) framework learns generative models via a training paradigm consisting of two competing modules – a generator and a discriminator. GAN training can be remarkably brittle and unstable in the face of suboptimal hyperparameter selection with generators often collapsing to a single mode or diverging entirely.

As presented in [Population Based Training \(PBT\)](#), PBT can help with the DCGAN training. We will now walk through how to do this in Tune. Complete code example at [github](#)

We define the Generator and Discriminator with standard Pytorch API:

```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find("BatchNorm") != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Generator Code
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz, ngf * 4, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
```

(continues on next page)

(continued from previous page)

```

        nn.BatchNorm2d(ngf * 2),
        nn.ReLU(True),
        nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
        nn.BatchNorm2d(ngf),
        nn.ReLU(True),
        nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
        nn.Tanh()))

def forward(self, input):
    return self.main(input)

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False), nn.Sigmoid())

    def forward(self, input):
        return self.main(input)

```

To train the model with PBT, we need to define a metric for the scheduler to evaluate the model candidates. For a GAN network, inception score is arguably the most commonly used metric. We trained a mnist classification model (LeNet) and use it to inference the generated images and evaluate the image quality.

```

class Net(nn.Module):
    """
    LeNet for MNist classification, used for inception_score
    """

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

```

(continues on next page)

(continued from previous page)

```

def inception_score(imgs, batch_size=32, splits=1):
    N = len(imgs)
    dtype = torch.FloatTensor
    dataloader = torch.utils.data.DataLoader(imgs, batch_size=batch_size)
    cm = ray.get(mnist_model_ref)
    up = nn.Upsample(size=(28, 28), mode="bilinear").type(dtype)

    def get_pred(x):
        x = up(x)
        x = cm(x)
        return F.softmax(x).data.cpu().numpy()

    preds = np.zeros((N, 10))
    for i, batch in enumerate(dataloader, 0):
        batch = batch.type(dtype)
        batchv = Variable(batch)
        batch_size_i = batch.size()[0]
        preds[i * batch_size:i * batch_size + batch_size_i] = get_pred(batchv)

    # Now compute the mean kl-div
    split_scores = []
    for k in range(splits):
        part = preds[k * (N // splits):(k + 1) * (N // splits), :]
        py = np.mean(part, axis=0)
        scores = []
        for i in range(part.shape[0]):
            pyx = part[i, :]
            scores.append(entropy(pyx, py))
        split_scores.append(np.exp(np.mean(scores)))

    return np.mean(split_scores), np.std(split_scores)

```

The Trainable class includes a Generator and a Discriminator, each with an independent learning rate and optimizer.

```

class PytorchTrainable(tune.Trainable):
    def __init__(self, config):
        use_cuda = config.get("use_gpu") and torch.cuda.is_available()
        self.device = torch.device("cuda" if use_cuda else "cpu")
        self.netD = Discriminator().to(self.device)
        self.netD.apply(weights_init)
        self.netG = Generator().to(self.device)
        self.netG.apply(weights_init)
        self.criterion = nn.BCELoss()
        self.optimizerD = optim.Adam(
            self.netD.parameters(),
            lr=config.get("lr", 0.01),
            betas=(beta1, 0.999))
        self.optimizerG = optim.Adam(
            self.netG.parameters(),
            lr=config.get("lr", 0.01),
            betas=(beta1, 0.999))
        with FileLock(os.path.expanduser("~/data.lock")):
            self.dataloader = get_data_loader()

```

(continues on next page)

(continued from previous page)

```

def _train(self):
    lossG, lossD, is_score = train(
        self.netD, self.netG, self.optimizerG, self.optimizerD,
        self.criterion, self.dataloader, self._iteration, self.device)
    return {"lossG": lossG, "lossD": lossD, "is_score": is_score}

def _save(self, checkpoint_dir):
    path = os.path.join(checkpoint_dir, "checkpoint")
    torch.save({
        "netDmodel": self.netD.state_dict(),
        "netGmodel": self.netG.state_dict(),
        "optimD": self.optimizerD.state_dict(),
        "optimG": self.optimizerG.state_dict(),
    }, path)

    return checkpoint_dir

def _restore(self, checkpoint_dir):
    path = os.path.join(checkpoint_dir, "checkpoint")
    checkpoint = torch.load(path)
    self.netD.load_state_dict(checkpoint["netDmodel"])
    self.netG.load_state_dict(checkpoint["netGmodel"])
    self.optimizerD.load_state_dict(checkpoint["optimD"])
    self.optimizerG.load_state_dict(checkpoint["optimG"])

def reset_config(self, new_config):
    if "netD_lr" in new_config:
        for param_group in self.optimizerD.param_groups:
            param_group["lr"] = new_config["netD_lr"]
    if "netG_lr" in new_config:
        for param_group in self.optimizerG.param_groups:
            param_group["lr"] = new_config["netG_lr"]

    self.config = new_config
    return True

def _export_model(self, export_formats, export_dir):
    if export_formats == [ExportFormat.MODEL]:
        path = os.path.join(export_dir, "exported_models")
        torch.save({
            "netDmodel": self.netD.state_dict(),
            "netGmodel": self.netG.state_dict()
        }, path)
        return {ExportFormat.MODEL: path}
    else:
        raise ValueError("unexpected formats: " + str(export_formats))

```

We specify inception score as the metric and start the tuning:

```

scheduler = PopulationBasedTraining(
    time_attr="training_iteration",
    metric="is_score",
    mode="max",
    perturbation_interval=5,

```

(continues on next page)

(continued from previous page)

```

hyperparam_mutations={
    # distribution for resampling
    "netG_lr": lambda: np.random.uniform(1e-2, 1e-5),
    "netD_lr": lambda: np.random.uniform(1e-2, 1e-5),
}

tune_iter = 5 if args.smoke_test else 300
analysis = tune.run(
    PytorchTrainable,
    name="pbt_dcgan_mnist",
    scheduler=scheduler,
    reuse_actors=True,
    verbose=1,
    checkpoint_at_end=True,
    stop={
        "training_iteration": tune_iter,
    },
    num_samples=8,
    export_formats=[ExportFormat.MODEL],
    config={
        "netG_lr": tune.sample_from(
            lambda spec: random.choice([0.0001, 0.0002, 0.0005])),
        "netD_lr": tune.sample_from(
            lambda spec: random.choice([0.0001, 0.0002, 0.0005]))
    })

```

The trained Generator models can be loaded from log directory, and generate images from noise signals.

## Visualization

Below, we visualize the increasing inception score from the training logs.

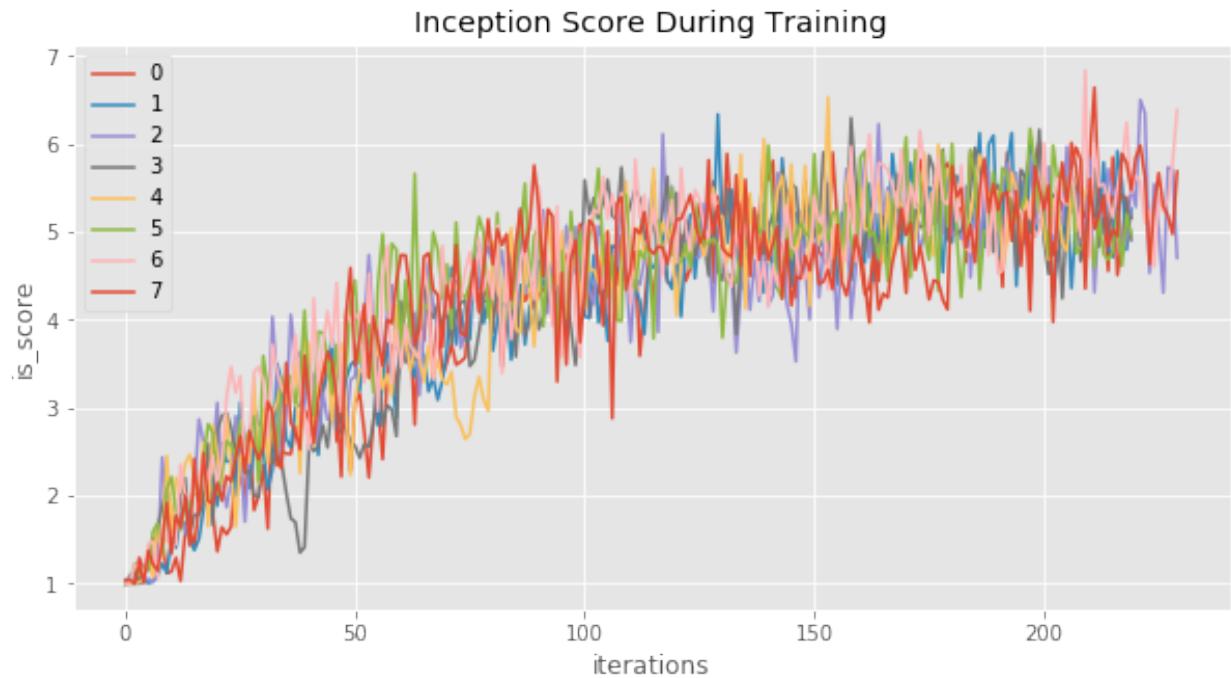
```

lossG = [df['is_score'].tolist() for df in list(analysis.trial_dataframes.values())]

plt.figure(figsize=(10,5))
plt.title("Inception Score During Training")
for i, lossG in enumerate(lossG):
    plt.plot(lossG, label=i)

plt.xlabel("iterations")
plt.ylabel("is_score")
plt.legend()
plt.show()

```

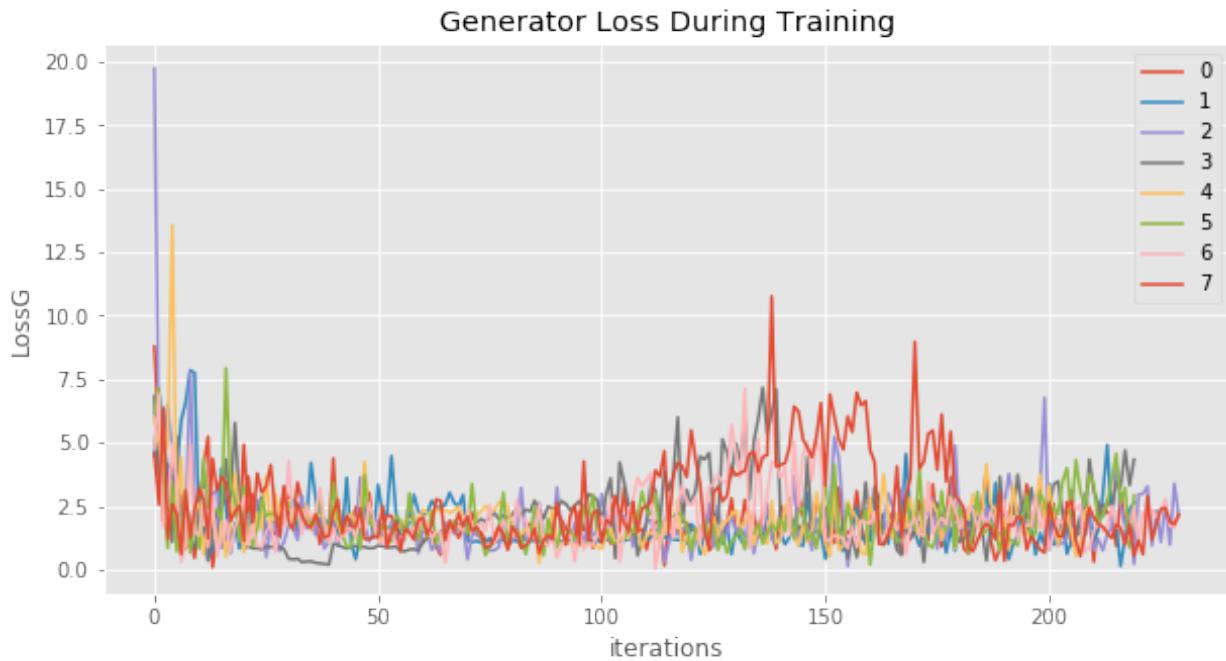


And the Generator loss:

```
lossG = [df['lossg'].tolist() for df in list(analysis.trial_dataframes.values())]

plt.figure(figsize=(10,5))
plt.title("Generator Loss During Training")
for i, lossg in enumerate(lossG):
    plt.plot(lossg,label=i)

plt.xlabel("iterations")
plt.ylabel("LossG")
plt.legend()
plt.show()
```



Training of the MNist Generator takes a couple of minutes. The example can be easily altered to generate images for other datasets, e.g. cifar10 or LSUN.

## Tune Distributed Experiments

Tune is commonly used for large-scale distributed hyperparameter optimization. This page will overview:

1. How to setup and launch a distributed experiment,
2. *Commonly used commands*, including fast file mounting, one-line cluster launching, and result uploading to cloud storage.

**Quick Summary:** To run a distributed experiment with Tune, you need to:

1. Make sure your script has `ray.init(address=...)` to connect to the existing Ray cluster.
2. If a ray cluster does not exist, start a Ray cluster.
3. Run the script on the head node (or use `ray submit`).

- *Running a distributed experiment*
- *Local Cluster Setup*
  - *Manual Local Cluster Setup*
- *Launching a cloud cluster*
- *Syncing*
- *Pre-emptible Instances (Cloud)*
  - *Example for using spot instances (AWS)*
- *Fault Tolerance*

- *Recovering From Failures*
- *Common Commands*
- *Troubleshooting*

## Running a distributed experiment

Running a distributed (multi-node) experiment requires Ray to be started already. You can do this on local machines or on the cloud.

Across your machines, Tune will automatically detect the number of GPUs and CPUs without you needing to manage CUDA\_VISIBLE\_DEVICES.

To execute a distributed experiment, call `ray.init(address=XXX)` before `tune.run`, where XXX is the Ray redis address, which defaults to `localhost:6379`. The Tune python script should be executed only on the head node of the Ray cluster.

One common approach to modifying an existing Tune experiment to go distributed is to set an `argparse` variable so that toggling between distributed and single-node is seamless.

```
import ray
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--address")
args = parser.parse_args()
ray.init(address=args.address)

tune.run(...)

# On the head node, connect to an existing ray cluster
$ python tune_script.py --ray-address=localhost:XXXX
```

If you used a cluster configuration (starting a cluster with `ray up` or `ray submit --start`), use:

```
ray submit tune-default.yaml tune_script.py -- --ray-address=localhost:6379
```

---

### Tip:

1. In the examples, the Ray redis address commonly used is `localhost:6379`.
  2. If the Ray cluster is already started, you should not need to run anything on the worker nodes.
- 

## Local Cluster Setup

If you already have a list of nodes, you can follow the local private cluster setup [instructions here](#). Below is an example cluster configuration as `tune-default.yaml`:

```
cluster_name: local-default
provider:
  type: local
  head_ip: YOUR_HEAD_NODE_HOSTNAME
```

(continues on next page)

(continued from previous page)

```
worker_ips: [WORKER_NODE_1_HOSTNAME, WORKER_NODE_2_HOSTNAME, ... ]
auth: {ssh_user: YOUR_USERNAME, ssh_private_key: ~/.ssh/id_rsa}
## Typically for local clusters, min_workers == max_workers.
min_workers: 3
max_workers: 3
setup_commands: # Set up each node.
    - pip install ray torch torchvision tabulate tensorboard
```

ray up starts Ray on the cluster of nodes.

```
ray up tune-default.yaml
```

ray submit uploads tune\_script.py to the cluster and runs python tune\_script.py [args].

```
ray submit tune-default.yaml tune_script.py -- --ray-address=localhost:6379
```

## Manual Local Cluster Setup

If you run into issues using the local cluster setup (or want to add nodes manually), you can use the manual cluster setup. [Full documentation here](#). At a glance,

### On the head node:

```
# 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 <address> with the value printed by the command on the head node (it should look something like 123.45.67.89:6379).

```
$ ray start --address=<address>
```

Then, you can run your Tune Python script on the head node like:

```
# On the head node, execute using existing ray cluster
$ python tune_script.py --ray-address=<address>
```

## Launching a cloud cluster

---

**Tip:** If you have already have a list of nodes, go to [Local Cluster Setup](#).

---

Ray currently supports AWS and GCP. Follow the instructions below to launch nodes on AWS (using the Deep Learning AMI). See the [cluster setup documentation](#). Save the below cluster configuration (tune-default.yaml):

```
cluster_name: tune-default
provider: {type: aws, region: us-west-2}
auth: {ssh_user: ubuntu}
min_workers: 3
max_workers: 3
# Deep Learning AMI (Ubuntu) Version 21.0
```

(continues on next page)

(continued from previous page)

```
head_node: {InstanceType: c5.xlarge, ImageId: ami-0b294f219d14e6a82}
worker_nodes: {InstanceType: c5.xlarge, ImageId: ami-0b294f219d14e6a82}
setup_commands: # Set up each node.
    - pip install ray torch torchvision tabulate tensorboard
```

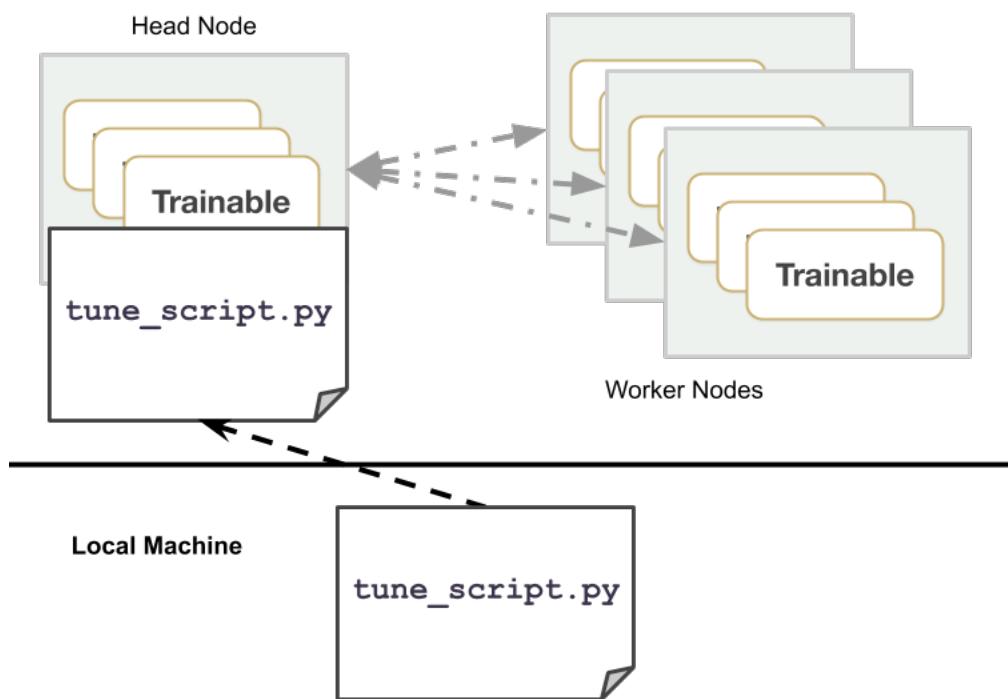
ray up starts Ray on the cluster of nodes.

```
ray up tune-default.yaml
```

ray submit --start starts a cluster as specified by the given cluster configuration YAML file, uploads tune\_script.py to the cluster, and runs python tune\_script.py [args].

**Note:** You may see a message like: bash: cannot set terminal process group (-1): Inappropriate ioctl for device bash: no job control in this shell This is a harmless error. If the cluster launcher fails, it is most likely due to some other factor.

```
ray submit tune-default.yaml tune_script.py --start -- --ray-address=localhost:6379
```



Analyze your results on TensorBoard by starting TensorBoard on the remote head machine.

```
# Go to http://localhost:6006 to access TensorBoard.
ray exec tune-default.yaml 'tensorboard --logdir=~/ray_results/ --port 6006' --port-
→forward 6006
```

Note that you can customize the directory of results by running: tune.run(local\_dir=...). You can then point TensorBoard to that directory to visualize results. You can also use awless for easy cluster management on AWS.

## Syncing

Tune automatically syncs the trial folder on remote nodes back to the head node. This requires the ray cluster to be started with the [autoscaler](#). By default, local syncing requires rsync to be installed. You can customize the sync command with the `sync_to_driver` argument in `tune.run` by providing either a function or a string.

If a string is provided, then it must include replacement fields `{source}` and `{target}`, like `rsync -savz -e "ssh -i ssh_key.pem" {source} {target}`. Alternatively, a function can be provided with the following signature:

```
def custom_sync_func(source, target):
    sync_cmd = "rsync {source} {target}".format(
        source=source,
        target=target)
    sync_process = subprocess.Popen(sync_cmd, shell=True)
    sync_process.wait()

tune.run(
    MyTrainableClass,
    name="experiment_name",
    sync_to_driver=custom_sync_func,
)
```

When syncing results back to the driver, the source would be a path similar to `ubuntu@192.0.0.1:/home/ubuntu/ray_results/trial1`, and the target would be a local path. This custom sync command is used to restart trials under failure. The `sync_to_driver` is invoked to push a checkpoint to new node for a paused/pre-empted trial to resume.

## Pre-emptible Instances (Cloud)

Running on spot instances (or pre-emptible instances) can reduce the cost of your experiment. You can enable spot instances in AWS via the following configuration modification:

```
# Provider-specific config for worker nodes, e.g. instance type.
worker_nodes:
    InstanceType: m5.large
    ImageId: ami-0b294f219d14e6a82 # Deep Learning AMI (Ubuntu) Version 21.0

    # Run workers on spot by default. Comment this out to use on-demand.
    InstanceMarketOptions:
        MarketType: spot
        SpotOptions:
            MaxPrice: 1.0 # Max Hourly Price
```

In GCP, you can use the following configuration modification:

```
worker_nodes:
    machineType: n1-standard-2
    disks:
        - boot: true
          autoDelete: true
          type: PERSISTENT
          initializeParams:
              diskSizeGb: 50
              # See https://cloud.google.com/compute/docs/images for more images
              sourceImage: projects/deeplearning-platform-release/global/images/family/tf-
```

↳ 1-13-cpu

(continues on next page)

(continued from previous page)

```
# Run workers on preemptible instances.
scheduling:
- preemptible: true
```

Spot instances may be removed suddenly while trials are still running. Often times this may be difficult to deal with when using other distributed hyperparameter optimization frameworks. Tune allows users to mitigate the effects of this by preserving the progress of your model training through checkpointing.

The easiest way to do this is to subclass the pre-defined Trainable class and implement `_save`, and `_restore` abstract methods, as seen in the example below:

```
class TrainMNIST(tune.Trainable):
    def _setup(self, config):
        use_cuda = config.get("use_gpu") and torch.cuda.is_available()
        self.device = torch.device("cuda" if use_cuda else "cpu")
        self.train_loader, self.test_loader = get_data_loaders()
        self.model = ConvNet().to(self.device)
        self.optimizer = optim.SGD(
            self.model.parameters(),
            lr=config.get("lr", 0.01),
            momentum=config.get("momentum", 0.9))

    def _train(self):
        train(
            self.model, self.optimizer, self.train_loader, device=self.device)
        acc = test(self.model, self.test_loader, self.device)
        return {"mean_accuracy": acc}

    def _save(self, checkpoint_dir):
        checkpoint_path = os.path.join(checkpoint_dir, "model.pth")
        torch.save(self.model.state_dict(), checkpoint_path)
        return checkpoint_path

    def _restore(self, checkpoint_path):
        self.model.load_state_dict(torch.load(checkpoint_path))
```

This can then be used similarly to the Function API as before:

```
search_space = {
    "lr": tune.sample_from(lambda spec: 10**(-10 * np.random.rand())),
    "momentum": tune.uniform(0.1, 0.9)
}

analysis = tune.run(
    TrainMNIST, config=search_space, stop={"training_iteration": 10})
```

## Example for using spot instances (AWS)

Here is an example for running Tune on spot instances. This assumes your AWS credentials have already been setup (`aws configure`):

1. Download a full example Tune experiment script here. This includes a Trainable with checkpointing: `mnist_pytorch_trainable.py`. To run this example, you will need to install the following:

```
$ pip install ray torch torchvision filelock
```

2. Download an example cluster yaml here: `tune-default.yaml`
3. Run `ray submit` as below to run Tune across them. Append `--start` if the cluster is not up yet. Append `--stop` to automatically shutdown your nodes after running.

```
ray submit tune-default.yaml mnist_pytorch_trainable.py --start -- --ray-
→address=localhost:6379
```

4. Optionally for testing on AWS or GCP, you can use the following to kill a random worker node after all the worker nodes are up

```
$ ray kill-random-node tune-default.yaml --hard
```

To summarize, here are the commands to run:

```
wget https://raw.githubusercontent.com/ray-project/ray/master/python/ray/tune/
→examples/mnist_pytorch_trainable.py
wget https://raw.githubusercontent.com/ray-project/ray/master/python/ray/tune/tune-
→default.yaml
ray submit tune-default.yaml mnist_pytorch_trainable.py --start -- --ray-
→address=localhost:6379

# wait a while until after all nodes have started
ray kill-random-node tune-default.yaml --hard
```

You should see Tune eventually continue the trials on a different worker node. See the [Fault Tolerance](#) section for more details.

You can also specify `tune.run(upload_dir=...)` to sync results with a cloud storage like S3, allowing you to persist results in case you want to start and stop your cluster automatically.

## Fault Tolerance

Tune will automatically restart trials in case of trial failures/error (if `max_failures != 0`), both in the single node and distributed setting.

Tune will restore trials from the latest checkpoint, where available. In the distributed setting, if using the autoscaler with `rsync` enabled, Tune will automatically sync the trial folder with the driver. For example, if a node is lost while a trial (specifically, the corresponding Trainable actor of the trial) is still executing on that node and a checkpoint of the trial exists, Tune will wait until available resources are available to begin executing the trial again.

If the trial/actor is placed on a different node, Tune will automatically push the previous checkpoint file to that node and restore the remote trial actor state, allowing the trial to resume from the latest checkpoint even after failure.

## Recovering From Failures

Tune automatically persists the progress of your entire experiment (`tune.run` session), so if an experiment crashes or is otherwise cancelled, it can be resumed by passing one of `True`, `False`, “`LOCAL`”, “`REMOTE`”, or “`PROMPT`” to `tune.run(resume=...)`. Note that this only works if trial checkpoints are detected, whether it be by manual or periodic checkpointing.

### Settings:

- The default setting of `resume=False` creates a new experiment.
- `resume="LOCAL"` and `resume=True` restore the experiment from `local_dir/[experiment_name]`.
- `resume="REMOTE"` syncs the upload dir down to the local dir and then restores the experiment from `local_dir/experiment_name`.
- `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.:

```
tune.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. For example, if the previous experiment has reached its termination, then resuming it with a new stop criterion will not run. The new experiment will terminate immediately after initialization. If you want to change the configuration, such as training more iterations, you can do so restore the checkpoint by setting `restore=<path-to-checkpoint>` - note that this only works for a single trial.

**Warning:** This feature is still experimental, so any provided Trial Scheduler or Search Algorithm will not be checkpointed and able to resume. Only `FIFOScheduler` and `BasicVariantGenerator` will be supported.

## Common Commands

Below are some commonly used commands for submitting experiments. Please see the [Autoscaler page](#) to see find more comprehensive documentation of commands.

```
# Upload `tune_experiment.py` from your local machine onto the cluster. Then,
# run `python tune_experiment.py --address=localhost:6379` on the remote machine.
$ ray submit CLUSTER.YAML tune_experiment.py -- --address=localhost:6379

# Start a cluster and run an experiment in a detached tmux session,
# and shut down the cluster as soon as the experiment completes.
# In `tune_experiment.py`, set `tune.run(upload_dir="s3://...")` to persist results
$ ray submit CLUSTER.YAML --tmux --start --stop tune_experiment.py -- --
  ↵address=localhost:6379
```

(continues on next page)

(continued from previous page)

```
# To start or update your cluster:  
$ ray up CLUSTER.YAML [-y]  
  
# Shut-down all instances of your cluster:  
$ ray down CLUSTER.YAML [-y]  
  
# Run Tensorboard and forward the port to your own machine.  
$ ray exec CLUSTER.YAML 'tensorboard --logdir ~/ray_results/ --port 6006' --port-  
  ↪forward 6006  
  
# Run Jupyter Lab and forward the port to your own machine.  
$ ray exec CLUSTER.YAML 'jupyter lab --port 6006' --port-forward 6006  
  
# Get a summary of all the experiments and trials that have executed so far.  
$ ray exec CLUSTER.YAML 'tune ls ~/ray_results'  
  
# Upload and sync file_mounts up to the cluster with this command.  
$ ray rsync-up CLUSTER.YAML  
  
# Download the results directory from your cluster head node to your local machine on  
  ↪`~/cluster_results`.  
$ ray rsync-down CLUSTER.YAML '~/ray_results' ~/cluster_results  
  
# Launching multiple clusters using the same configuration.  
$ ray up CLUSTER.YAML -n="cluster1"  
$ ray up CLUSTER.YAML -n="cluster2"  
$ ray up CLUSTER.YAML -n="cluster3"
```

## Troubleshooting

Sometimes, your program may freeze. Run this to restart the Ray cluster without running any of the installation commands.

```
$ ray up CLUSTER.YAML --restart-only
```

### 5.10.3 Colab Exercises

Learn how to use Tune in your browser with the following Colab-based exercises.

Tutorial source files [can be found here](#).

### 5.10.4 Tune Examples

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.

## 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. 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 PopulationBasedTraining scheduler.
- `logging_example`: Example of custom loggers and custom trial directory naming.

## Search Algorithm Examples

- `Ax example`: Optimize a Hartmann function with Ax with 4 parallel workers.
- `HyperOpt Example`: Optimizes a basic function using the function-based API and the HyperOptSearch (SearchAlgorithm wrapper for HyperOpt TPE).
- `Nevergrad example`: Optimize a simple toy function with the gradient-free optimization package Nevergrad with 4 parallel workers.
- `Bayesian Optimization example`: Optimize a simple toy function using Bayesian Optimization with 4 parallel workers.

## Tensorflow/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.
- `pbt_memnn_example`: Example of training a Memory NN on bAbI with Keras using PBT.
- `Tensorflow 2 Example`: Converts the Advanced TF2.0 MNIST example to use Tune with the Trainable. This uses `tf.function`. Original code from tensorflow: <https://www.tensorflow.org/tutorials/quickstart/advanced>

## 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.

## XGBoost Example

- `xgboost_example`: Trains a basic XGBoost model with Tune with the function-based API and an XGBoost callback.

## LightGBM Example

- [lightgbm\\_example](#): Trains a basic LightGBM model with Tune with the function-based API and a LightGBM callback.

## 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 algorithm with AsyncHyperBandScheduler.
- [tune\\_cifar10\\_gluon](#): MXNet Gluon example to use Tune with the function-based API on CIFAR-10 dataset.

## 5.11 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( ... , 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](#). All schedulers take in a `metric`, which is a value returned in the result dict of your Trainable and is maximized or minimized according to `mode`.

Current Available Trial Schedulers:

- [Population Based Training \(PBT\)](#)
- [Asynchronous HyperBand](#)
- [HyperBand](#)
  - [HyperBand Implementation Details](#)
- [HyperBand \(BOHB\)](#)
- [Median Stopping Rule](#)

### 5.11.1 Population Based Training (PBT)

Tune includes a distributed implementation of [Population Based Training \(PBT\)](#). This can be enabled by setting the `scheduler` parameter of `tune.run`, e.g.

```
pbt_scheduler = PopulationBasedTraining(  
    time_attr='time_total_s',  
    metric='mean_accuracy',  
    mode='max',  
    perturbation_interval=600.0,  
    hyperparam_mutations={  
        "lr": [1e-3, 5e-4, 1e-4, 5e-5, 1e-5],
```

(continues on next page)

(continued from previous page)

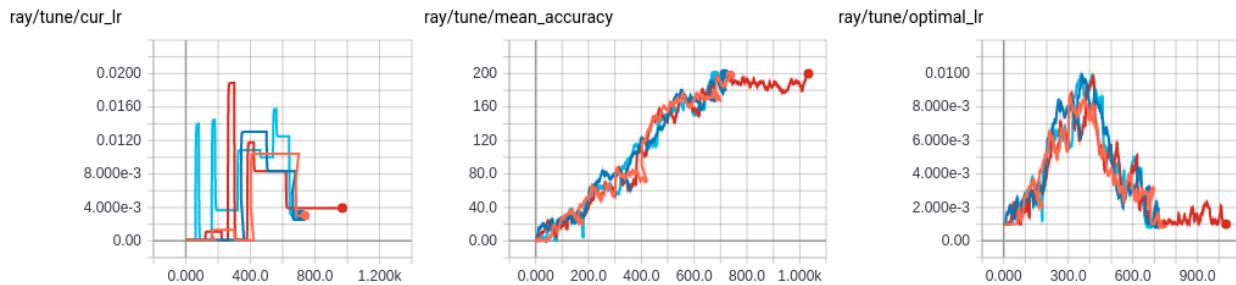
```

    "alpha": lambda: random.uniform(0.0, 1.0),
    ...
})
tune.run(..., 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 [save and restore](#)). 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 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 with optimizing a LR schedule over the course of a single experiment:



```

class ray.tune.schedulers.PopulationBasedTraining(time_attr='time_total_s',
                                                 reward_attr=None,
                                                 metric='episode_reward_mean',
                                                 mode='max',
                                                 perturbation_interval=60.0,
                                                 hyperparam_mutations={},
                                                 quantile_fraction=0.25,
                                                 sample_probability=0.25,
                                                 custom_explore_fn=None,
                                                 log_config=True)

```

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. To run multiple trials, use `tune.run(num_samples=<int>)`.

In `{LOG_DIR}/{MY_EXPERIMENT_NAME}/`, all mutations are logged in `pbt_global.txt` and individual policy perturbations are recorded in `pbt_policy_{i}.txt`. Tune logs: [target trial tag, clone trial tag, target trial iteration, clone trial iteration, old config, new config] on each perturbation step.

## 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.

- **metric** (*str*) – The training result objective value attribute. Stopping procedures will use this attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric 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*.
- **quantile\_fraction** (*float*) – Parameters are transferred from the top *quantile\_fraction* fraction of trials to the bottom *quantile\_fraction* fraction. Needs to be between 0 and 0.5. Setting it to 0 essentially implies doing no exploitation at all.
- **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*.
- **log\_config** (*bool*) – Whether to log the ray config of each model to local\_dir at each exploit. Allows config schedule to be reconstructed.

```
import random
from ray import tune
from ray.tune.schedulers import PopulationBasedTraining

pbt = PopulationBasedTraining(
    time_attr="training_iteration",
    metric="episode_reward_mean",
    mode="max",
    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],
    })
tune.run(..., num_samples=8, scheduler=pbt)
```

## 5.11.2 Asynchronous HyperBand

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

```
async_hb_scheduler = AsyncHyperBandScheduler(
    time_attr='training_iteration',
    metric='episode_reward_mean',
    mode='max',
    max_t=100,
    grace_period=10,
    reduction_factor=3,
    brackets=3)
tune.run( ... , 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=None,               met-
                                                 ric='episode_reward_mean',
                                                 mode='max',                  max_t=100,
                                                 grace_period=1,              reduc-
                                                 tion_factor=4, brackets=1)
```

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://arxiv.org/abs/1810.05934>

### 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.
- **metric** (*str*) – The training result objective value attribute. Stopping procedures will use this attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric 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.

### 5.11.3 HyperBand

**Note:** Note that the HyperBand scheduler requires your trainable to support *saving and restoring*. 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:

```
tune.run( ... , 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=None,
                                              metric='episode_reward_mean', mode='max',
                                              max_t=81, reduction_factor=3)
```

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`, the name of the reported objective value `metric`, and if `metric` is to be maximized or minimized (`mode`). 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>

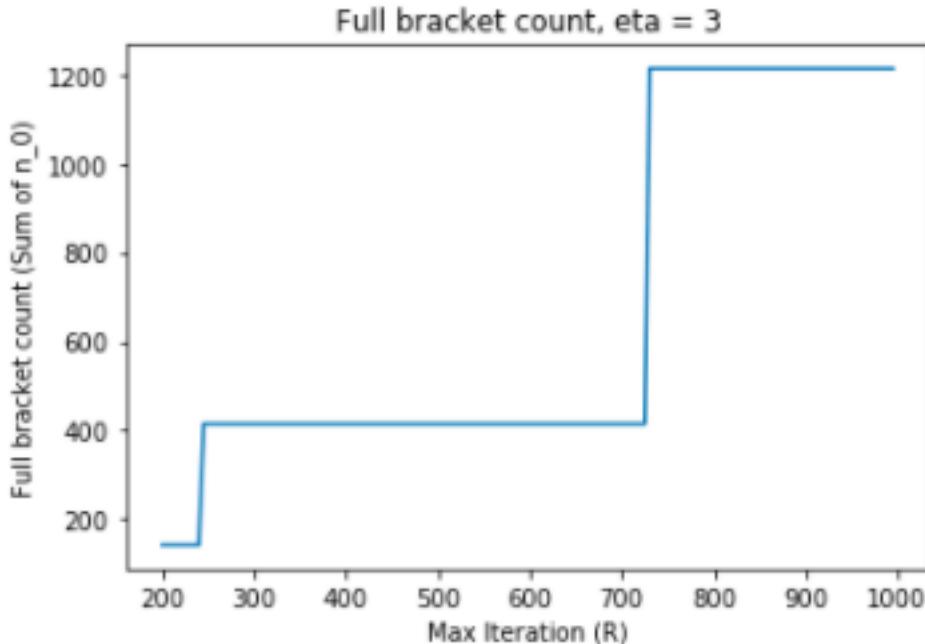
### Parameters

- **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.
- **metric** (*str*) – The training result objective value attribute. Stopping procedures will use this attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric 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.
- **reduction\_factor** (*float*) – Same as *eta*. Determines how sharp the difference is between bracket space-time allocation ratios.

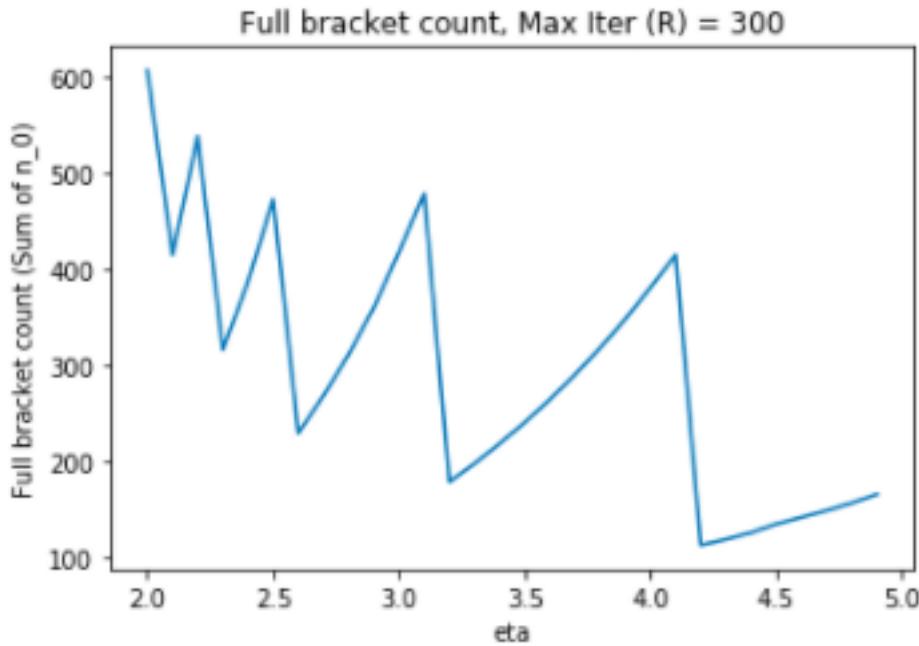
### HyperBand Implementation Details

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  $R \geq 200$ . For simplicity, assume  $\text{eta} = 3$ . 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.

#### 5.11.4 HyperBand (BOHB)

---

**Tip:** This implementation is still experimental. Please report issues on <https://github.com/ray-project/ray/issues/>. Thanks!

---

This class is a variant of HyperBand that enables the BOHB Algorithm. This implementation is true to the original HyperBand implementation and does not implement pipelining nor straggler mitigation.

This is to be used in conjunction with the Tune BOHB search algorithm. See [TuneBOHB](#) for package requirements, examples, and details.

An example of this in use can be found in `bohb_example.py`.

```
class ray.tune.schedulers.HyperBandForBOHB(time_attr='training_iteration',
                                             reward_attr=None,
                                             metric='episode_reward_mean',
                                             mode='max',
                                             max_t=81, reduction_factor=3)
```

Extends HyperBand early stopping algorithm for BOHB.

This implementation removes the `HyperBandScheduler` pipelining. This class introduces key changes:

1. Trials are now placed so that the bracket with the largest size is filled first.
  2. Trials will be paused even if the bracket is not filled. This allows BOHB to insert new trials into the training.
- See `ray.tune.schedulers.HyperBandScheduler` for parameter docstring.

### 5.11.5 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:

```
tune.run( ... , scheduler=MedianStoppingRule())
```

```
class ray.tune.schedulers.MedianStoppingRule(time_attr='time_total_s',
                                              reward_attr=None,
                                              metric='episode_reward_mean',
                                              mode='max',
                                              grace_period=60.0,
                                              min_samples_required=3,
                                              min_time_slice=0, hard_stop=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.
- **metric** (*str*) – The training result objective value attribute. Stopping procedures will use this attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **grace\_period** (*float*) – Only stop trials at least this old in time. The mean will only be computed from this time onwards. The units are the same as the attribute named by `time_attr`.
- **min\_samples\_required** (*int*) – Minimum number of trials to compute median over.
- **min\_time\_slice** (*float*) – Each trial runs at least this long before yielding (assuming it isn't stopped). Note: trials ONLY yield if there are not enough samples to evaluate performance for the current result AND there are other trials waiting to run. The units are the same as the attribute named by `time_attr`.
- **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.

## 5.12 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:

```
tune.run(my_function, search_alg=SearchAlgorithm(...))
```

Currently, Tune offers the following search algorithms (and library integrations):

- Grid Search and Random Search
- BayesOpt
- HyperOpt
- SigOpt
- Nevergrad
- Scikit-Optimize
- Ax
- BOHB

### 5.12.1 Variant Generation (Grid Search/Random Search)

By default, Tune uses a `BasicVariantGenerator` to sample trials. This supports random search and grid search as specified by the `config` parameter of `tune.run`.

`class ray.tune.suggest.BasicVariantGenerator(shuffle=False)`  
Bases: `ray.tune.suggest.search.SearchAlgorithm`

Uses Tune's variant generation for resolving variables.

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

**Parameters** `shuffle` (`bool`) – Shuffles the generated list of configurations.

User API:

```
from ray import tune
from ray.tune.suggest import BasicVariantGenerator

searcher = BasicVariantGenerator()
tune.run(my_trainable_func, algo=searcher)
```

Internal API:

```
from ray.tune.suggest import BasicVariantGenerator

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

Read about this in the [Grid/Random Search API](#).

Note that other search algorithms will require a different search space declaration than the default Tune format.

## 5.12.2 Repeated Evaluations

Use `ray.tune.suggest.Repeater` to average over multiple evaluations of the same hyperparameter configurations. This is useful in cases where the evaluated training procedure has high variance (i.e., in reinforcement learning).

By default, `Repeater` will take in a `repeat` parameter and a `search_alg`. The `search_alg` will suggest new configurations to try, and the `Repeater` will run `repeat` trials of the configuration. It will then average the `search_alg.metric` from the final results of each repeated trial.

See the API documentation ([Repeater](#)) for more details.

```
from ray.tune.suggest import Repeater

search_alg = BayesOpt(...)
re_search_alg = Repeater(search_alg, repeat=10)

# Repeat 2 samples 10 times each.
tune.run(trainable, num_samples=20, search_alg=re_search_alg)
```

---

**Note:** This does not apply for grid search and random search.

---

**Warning:** It is recommended to not use `Repeater` with a `TrialScheduler`. Early termination can negatively affect the average reported metric.

## 5.12.3 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:

```
tune.run(..., search_alg=BayesOptSearch(bayesopt_space, utility_kwarg=utility_
    ↴params, ...))
```

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

```
class ray.tune.suggest.bayesopt.BayesOptSearch(space, metric='episode_reward_mean',
                                                mode='max', utility_kwarg=None,
                                                random_state=1, verbose=0,
                                                max_concurrent=None,
                                                use_early_stopped_trials=None)
```

Bases: `ray.tune.suggest.Searcer`

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.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **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.
- **max\_concurrent** – Deprecated.
- **use\_early\_stopped\_trials** – Deprecated.

```
from ray import tune
from ray.tune.suggest.bayesopt import BayesOptSearch

space = {
    'width': (0, 20),
    'height': (-100, 100),
}
algo = BayesOptSearch(space, metric="mean_loss", mode="min")
tune.run(my_func, algo=algo)
```

### 5.12.4 HyperOpt Search (Tree-structured Parzen Estimators)

The HyperOptSearch is a SearchAlgorithm that is backed by [HyperOpt](#) to perform sequential model-based hyper-parameter 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:

```
tune.run(..., search_alg=HyperOptSearch(hyperopt_space, ...))
```

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

```
class ray.tune.suggest.hyperopt.HyperOptSearch(space, metric='episode_reward_mean',
                                                mode='max', points_to_evaluate=None,
                                                n_initial_points=20, random_state=None, gamma=0.25,
                                                max_concurrent=None, use_early_stopped_trials=None)
```

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

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. Trials of the current run can be saved using save method, trials of a previous run can be loaded using restore method, thus enabling a warm start feature.

### 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.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **points\_to\_evaluate** (*list*) – Initial parameter suggestions to be run first. This is for when you already have some good parameters you want hyperopt to run first to help the TPE algorithm make better suggestions for future parameters. Needs to be a list of dict of hyperopt-named variables. Choice variables should be indicated by their index in the list (see example)
- **n\_initial\_points** (*int*) – number of random evaluations of the objective function before starting to approximate it with tree parzen estimators. Defaults to 20.
- **random\_state\_seed** (*int, array\_like, None*) – seed for reproducible results. Defaults to None.
- **gamma** (*float in range (0, 1)*) – parameter governing the tree parzen estimators suggestion algorithm. Defaults to 0.25.
- **max\_concurrent** – Deprecated.
- **use\_early\_stopped\_trials** – Deprecated.

```
space = {
    'width': hp.uniform('width', 0, 20),
    'height': hp.uniform('height', -100, 100),
    'activation': hp.choice("activation", ["relu", "tanh"])
}
current_best_params = [{
    'width': 10,
    'height': 0,
    'activation': 0, # The index of "relu"
}]
algo = HyperOptSearch(
    space, metric="mean_loss", mode="min",
    points_to_evaluate=current_best_params)
```

## 5.12.5 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:

```
tune.run(..., search_alg=SigOptSearch(sigopt_space, ...))
```

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

```
class ray.tune.suggest.sigopt.SigOptSearch(space, name='Default Tune Experiment',
                                             max_concurrent=1, reward_attr=None, metric='episode_reward_mean', mode='max',
                                             **kwargs)
```

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

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`.

This module manages its own concurrency.

#### 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.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.

Example:

```
space = [
    {
        'name': 'width',
        'type': 'int',
        'bounds': {
            'min': 0,
            'max': 20
        },
    },
    {
        'name': 'height',
        'type': 'int',
        'bounds': {
            'min': -100,
            'max': 100
        },
    },
]
algo = SigOptSearch(
    space, name="SigOpt Example Experiment",
    max_concurrent=1, metric="mean_loss", mode="min")
```

## 5.12.6 Nevergrad Search

The `NevergradSearch` is a `SearchAlgorithm` that is backed by `Nevergrad` 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 `NevergradSearch`.

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

```
$ pip install nevergrad
```

Keep in mind that `nevergrad` is a Python 3.6+ library.

This algorithm requires using an optimizer provided by `nevergrad`, of which there are many options. A good rundown can be found on their README's [Optimization](#) section. You can use `NevergradSearch` like follows:

```
tune.run(..., search_alg=NevergradSearch(optimizer, parameter_names, ...))
```

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

```
class ray.tune.suggest.nevergrad.NevergradSearch(optimizer, parameter_names,
                                                metric='episode_reward_mean',
                                                mode='max', max_concurrent=None,
                                                **kwargs)
```

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

A wrapper around Nevergrad to provide trial suggestions.

Requires Nevergrad to be installed.

Nevergrad is an open source tool from Facebook for derivative free optimization of parameters and/or hyperparameters. It features a wide range of optimizers in a standard ask and tell interface. More information can be found at <https://github.com/facebookresearch/nevergrad>.

### Parameters

- **optimizer** (`nevergrad.optimization.Optimizer`) – Optimizer provided from Nevergrad.
- **parameter\_names** (`list`) – List of parameter names. Should match the dimension of the optimizer output. Alternatively, set to None if the optimizer is already instrumented with kwargs (see nevergrad v0.2.0+).
- **metric** (`str`) – The training result objective value attribute.
- **mode** (`str`) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **use\_early\_stopped\_trials** – Deprecated.
- **max\_concurrent** – Deprecated.

```
from nevergrad.optimization import optimizerlib

instrumentation = 1
optimizer = optimizerlib.OnePlusOne(instrumentation, budget=100)
algo = NevergradSearch(
    optimizer, ["lr"], metric="mean_loss", mode="min")
```

---

**Note:** In nevergrad v0.2.0+, optimizers can be instrumented. For instance, the following will specifies searching for "lr" from 1 to 2.

```
>>> from nevergrad.optimization import optimizerlib
>>> from nevergrad import instrumentation as inst
>>> lr = inst.var.Array(1).bounded(1, 2).asfloat()
>>> instrumentation = inst.Instrumentation(lr=lr)
>>> optimizer = optimizerlib.OnePlusOne(instrumentation, budget=100)
>>> algo = NevergradSearch(
    optimizer, None, metric="mean_loss", mode="min")
```

---

## 5.12.7 Scikit-Optimize Search

The `SkOptSearch` is a `SearchAlgorithm` that is backed by [Scikit-Optimize](#) to perform sequential model-based hyper-parameter 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 `SkOptSearch`.

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

```
$ pip install scikit-optimize
```

This algorithm requires using the [Scikit-Optimize ask and tell interface](#). This interface requires using the [Optimizer](#) provided by Scikit-Optimize. You can use `SkOptSearch` like follows:

```
optimizer = Optimizer(dimension, ...)
tune.run(..., search_alg=SkOptSearch(optimizer, parameter_names, ...))
```

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

```
class ray.tune.suggest.skopt.SkOptSearch(optimizer, parameter_names, metric='episode_reward_mean', mode='max', points_to_evaluate=None, evaluated_rewards=None, max_concurrent=None, use_early_stopped_trials=None)
```

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

A wrapper around skopt to provide trial suggestions.

Requires skopt to be installed.

### Parameters

- **optimizer** (`skopt.optimizer.Optimizer`) – Optimizer provided from skopt.
- **parameter\_names** (`list`) – List of parameter names. Should match the dimension of the optimizer output.
- **metric** (`str`) – The training result objective value attribute.
- **mode** (`str`) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **points\_to\_evaluate** (`list of lists`) – A list of points you'd like to run first before sampling from the optimiser, e.g. these could be parameter configurations you already know work well to help the optimiser select good values. Each point is a list of the parameters using the order definition given by `parameter_names`.
- **evaluated\_rewards** (`list`) – If you have previously evaluated the parameters passed in as `points_to_evaluate` you can avoid re-running those trials by passing in

the reward attributes as a list so the optimiser can be told the results without needing to re-compute the trial. Must be the same length as `points_to_evaluate`. (See `tune/examples/skopt_example.py`)

- `max_concurrent` – Deprecated.
- `use_early_stopped_trials` – Deprecated.

## Example

```
>>> from skopt import Optimizer
>>> optimizer = Optimizer([(0,20),(-100,100)])
>>> current_best_params = [[10, 0], [15, -20]]
>>> algo = SkOptSearch(optimizer,
>>>     ["width", "height"],
>>>     metric="mean_loss",
>>>     mode="min",
>>>     points_to_evaluate=current_best_params)
```

### 5.12.8 Dragonfly Search

The `DragonflySearch` is a `SearchAlgorithm` that is backed by `Dragonfly` to perform sequential Bayesian 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 `DragonflySearch`.

```
$ pip install dragonfly-opt
```

This algorithm requires using the `Dragonfly` ask and tell interface. This interface requires using `FunctionCallers` and optimizers provided by `Dragonfly`. You can use `DragonflySearch` like follows:

```
from dragonfly.opt.gp_bandit import EuclideanGPBandit
from dragonfly.exd.experiment_caller import EuclideanFunctionCaller
from dragonfly import load_config
domain_config = load_config({'domain': ...})
func_caller = EuclideanFunctionCaller(None, domain_config.domain.list_of_domains[0])
optimizer = EuclideanGPBandit(func_caller, ask_tell_mode=True)
algo = DragonflySearch(optimizer, ...)
```

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

```
class ray.tune.suggest.dragonfly.DragonflySearch(optimizer, met-
                                                 ric='episode_reward_mean',
                                                 mode='max',
                                                 points_to_evaluate=None, evaluated_rewards=None, **kwargs)
```

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

A wrapper around Dragonfly to provide trial suggestions.

Requires Dragonfly to be installed via `pip install dragonfly-opt`.

#### Parameters

- `optimizer` (`dragonfly.opt.BlackboxOptimiser`) – Optimizer provided from `dragonfly`. Choose an optimiser that extends `BlackboxOptimiser`.
- `metric` (`str`) – The training result objective value attribute.

- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **points\_to\_evaluate** (*list of lists*) – A list of points you'd like to run first before sampling from the optimiser, e.g. these could be parameter configurations you already know work well to help the optimiser select good values. Each point is a list of the parameters using the order definition given by parameter\_names.
- **evaluated\_rewards** (*list*) – If you have previously evaluated the parameters passed in as points\_to\_evaluate you can avoid re-running those trials by passing in the reward attributes as a list so the optimiser can be told the results without needing to re-compute the trial. Must be the same length as points\_to\_evaluate.

```
from ray import tune
from dragonfly.opt.gp_bandit import EuclideanGPBandit
from dragonfly.exd.experiment_caller import EuclideanFunctionCaller
from dragonfly import load_config

domain_vars = [
    {
        "name": "LiNO3_vol",
        "type": "float",
        "min": 0,
        "max": 7
    },
    {
        "name": "Li2SO4_vol",
        "type": "float",
        "min": 0,
        "max": 7
    },
    {
        "name": "NaClO4_vol",
        "type": "float",
        "min": 0,
        "max": 7
    }
]

domain_config = load_config({"domain": domain_vars})
func_caller = EuclideanFunctionCaller(None,
    domain_config.domain.list_of_domains[0])
optimizer = EuclideanGPBandit(func_caller, ask_tell_mode=True)

algo = DragonflySearch(optimizer, metric="objective", mode="max")

tune.run(my_func, algo=algo)
```

### 5.12.9 Ax Search

The AxSearch is a SearchAlgorithm that is backed by Ax to perform sequential model-based hyperparameter optimization. Ax is a platform for understanding, managing, deploying, and automating adaptive experiments. Ax provides an easy to use interface with BoTorch, a flexible, modern library for Bayesian optimization in PyTorch. 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 AxSearch.

In order to use this search algorithm, you will need to install PyTorch, Ax, and sqlalchemy. Instructions to install PyTorch locally can be found [here](#). You can install Ax and sqlalchemy via the following command:

```
$ pip install ax-platform sqlalchemy
```

This algorithm requires specifying a search space and objective. You can use `AxSearch` like follows:

```
client = AxClient(enforce_sequential_optimization=False)
client.create_experiment( ... )
tune.run(... , search_alg=AxSearch(client))
```

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

```
class ray.tune.suggest.ax.AxSearch(ax_client, mode='max', use_early_stopped_trials=None,
                                         max_concurrent=None)
Bases: ray.tune.suggest.suggestion.Searcher
```

A wrapper around Ax to provide trial suggestions.

Requires Ax to be installed. Ax is an open source tool from Facebook for configuring and optimizing experiments. More information can be found in <https://ax.dev/>.

### Parameters

- **parameters** (`list[dict]`) – Parameters in the experiment search space. Required elements in the dictionaries are: “name” (name of this parameter, string), “type” (type of the parameter: “range”, “fixed”, or “choice”, string), “bounds” for range parameters (list of two values, lower bound first), “values” for choice parameters (list of values), and “value” for fixed parameters (single value).
- **objective\_name** (`str`) – Name of the metric used as objective in this experiment. This metric must be present in `raw_data` argument to `log_data`. This metric must also be present in the dict reported/returned by the Trainable.
- **mode** (`str`) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute. Defaults to “max”.
- **parameter\_constraints** (`list[str]`) – Parameter constraints, such as “ $x3 \geq x4$ ” or “ $x3 + x4 \geq 2$ ”.
- **outcome\_constraints** (`list[str]`) – Outcome constraints of form “`metric_name >= bound`”, like “`m1 <= 3`.”
- **max\_concurrent** (`int`) – Deprecated.
- **use\_early\_stopped\_trials** – Deprecated.

```
from ax.service.ax_client import AxClient
from ray import tune
from ray.tune.suggest.ax import AxSearch

parameters = [
    {"name": "x1", "type": "range", "bounds": [0.0, 1.0]},
    {"name": "x2", "type": "range", "bounds": [0.0, 1.0]},
]

def easy_objective(config):
    for i in range(100):
        intermediate_result = config["x1"] + config["x2"] * i
        tune.track.log(score=intermediate_result)

client = AxClient(enforce_sequential_optimization=False)
client.create_experiment(parameters=parameters, objective_name="score")
algo = AxSearch(client)
tune.run(easy_objective, search_alg=algo)
```

## 5.12.10 BOHB

**Tip:** This implementation is still experimental. Please report issues on <https://github.com/ray-project/ray/issues/>. Thanks!

---

BOHB (Bayesian Optimization HyperBand) is a SearchAlgorithm that is backed by HpBandSter to perform sequential model-based hyperparameter optimization in conjunction with HyperBand. 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 BOHB.

Importantly, BOHB is intended to be paired with a specific scheduler class: `HyperBandForBOHB`.

This algorithm requires using the `ConfigSpace` search space specification. In order to use this search algorithm, you will need to install `HpBandSter` and `ConfigSpace`:

```
$ pip install hpbandster ConfigSpace
```

You can use TuneBOHB in conjunction with `HyperBandForBOHB` as follows:

```
# BOHB uses ConfigSpace for their hyperparameter search space
import ConfigSpace as CS

config_space = CS.ConfigurationSpace()
config_space.add_hyperparameter(
    CS.UniformFloatHyperparameter("height", lower=10, upper=100))
config_space.add_hyperparameter(
    CS.UniformFloatHyperparameter("width", lower=0, upper=100))

experiment_metrics = dict(metric="episode_reward_mean", mode="min")
bohb_hyperband = HyperBandForBOHB(
    time_attr="training_iteration", max_t=100, **experiment_metrics)
bohb_search = TuneBOHB(
    config_space, max_concurrent=4, **experiment_metrics)

tune.run(MyTrainableClass,
         name="bohb_test",
         scheduler=bohb_hyperband,
         search_alg=bohb_search,
         num_samples=5)
```

Take a look at [an example here](#). See the [BOHB paper](#) for more details.

```
class ray.tune.suggest.bohb.TuneBOHB(space, bohb_config=None, max_concurrent=10, metric='neg_mean_loss', mode='max')
Bases: ray.tune.suggest.suggestion.Searcher
```

BOHB suggestion component.

Requires `HpBandSter` and `ConfigSpace` to be installed. You can install `HpBandSter` and `ConfigSpace` with: `pip install hpbandster ConfigSpace`.

This should be used in conjunction with `HyperBandForBOHB`.

### Parameters

- **space** (`ConfigurationSpace`) – Continuous `ConfigSpace` search space. Parameters will be sampled from this space which will be used to run trials.
- **bohb\_config** (`dict`) – configuration for `HpBandSter` BOHB algorithm

- **max\_concurrent** (*int*) – Number of maximum concurrent trials. Defaults to 10.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.

Example:

```
import ConfigSpace as CS

config_space = CS.ConfigurationSpace()
config_space.add_hyperparameter(
    CS.UniformFloatHyperparameter('width', lower=0, upper=20))
config_space.add_hyperparameter(
    CS.UniformFloatHyperparameter('height', lower=-100, upper=100))
config_space.add_hyperparameter(
    CS.CategoricalHyperparameter(
        name='activation', choices=['relu', 'tanh']))

algo = TuneBOHB(
    config_space, max_concurrent=4, metric='mean_loss', mode='min')
bohb = HyperBandForBOHB(
    time_attr='training_iteration',
    metric='mean_loss',
    mode='min',
    max_t=100)
run(MyTrainableClass, scheduler=bohb, search_alg=algo)
```

### 5.12.11 ZOOpt Search

The ZOOptSearch is a SearchAlgorithm for derivative-free optimization. It is backed by the [ZOOpt](#) package. Currently, Asynchronous Sequential RAndomized COordinate Shrinking (ASRacos) algorithm is implemented in Tune. 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 ZOOptSearch.

In order to use this search algorithm, you will need to install the ZOOpt package (**>=0.4.0**) via the following command:

```
$ pip install -U zoopt
```

Keep in mind that zoopt only supports Python 3.

This algorithm allows users to mix continuous dimensions and discrete dimensions, for example:

```
dim_dict = {
    # for continuous dimensions: (continuous, search_range, precision)
    "height": (ValueType.CONTINUOUS, [-10, 10], 1e-2),
    # for discrete dimensions: (discrete, search_range, has_order)
    "width": (ValueType.DISCRETE, [-10, 10], False)
}

config = {
    "num_samples": 200 if args.smoke_test else 1000,
    "config": {
        "iterations": 10, # evaluation times
    },
    "stop": {
        "timesteps_total": 10 # custom stop rules
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

zoopt_search = ZOOptSearch(
    algo="Asracos", # only support ASRacos currently
    budget=config["num_samples"],
    dim_dict=dim_dict,
    max_concurrent=4,
    metric="mean_loss",
    mode="min")

run(my_objective,
    search_alg=zoopt_search,
    name="zoopt_search",
    **config)

```

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

```
class ray.tune.suggest.zoopt.ZOOptSearch(algo='asracos', budget=None, dim_dict=None,  

                                         metric='episode_reward_mean', mode='min',  

                                         **kwargs)
```

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

A wrapper around ZOOpt to provide trial suggestions.

Requires `zoopt` package ( $\geq 0.4.0$ ) to be installed. You can install it with the command: `pip install -U zoopt`.

#### Parameters

- **algo** (*str*) – To specify an algorithm in `zoopt` you want to use. Only support ASRacos currently.
- **budget** (*int*) – Number of samples.
- **dim\_dict** (*dict*) – Dimension dictionary. For continuous dimensions: (continuous, `search_range`, `precision`); For discrete dimensions: (discrete, `search_range`, `has_order`). More details can be found in `zoopt` package.
- **metric** (*str*) – The training result objective value attribute. Defaults to “`episode_reward_mean`”.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute. Defaults to “min”.

```

from ray.tune import run
from ray.tune.suggest.zoopt import ZOOptSearch
from zoopt import ValueType

dim_dict = {
    "height": (ValueType.CONTINUOUS, [-10, 10], 1e-2),
    "width": (ValueType.DISCRETE, [-10, 10], False)
}

config = {
    "num_samples": 200,
    "config": {
        "iterations": 10, # evaluation times
    },
}

```

(continues on next page)

(continued from previous page)

```

    "stop": {
        "timesteps_total": 10 # custom stop rules
    }
}

zoopt_search = ZOOptSearch(
    algo="Asracos", # only support Asracos currently
    budget=config["num_samples"],
    dim_dict=dim_dict,
    metric="mean_loss",
    mode="min")

run(my_objective,
    search_alg=zoopt_search,
    name="zoopt_search",
    **config)

```

### 5.12.12 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)**  
 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.

**is\_finished()**

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

Can return True before all trials have finished executing.

**set\_finished()**

Marks the search algorithm as finished.

## 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_complete`, and `suggest`.

```
class ray.tune.suggest.Searcher(metric='episode_reward_mean', mode='max', max_concurrent=None, use_early_stopped_trials=None)
```

Bases: object

Abstract class for wrapping suggesting algorithms.

Custom algorithms can extend this class easily by overriding the `suggest` method provide generated parameters for the trials.

Any subclass that implements `__init__` must also call the constructor of this class: `super(Subclass, self).__init__(...)`.

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

### Parameters

- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.

```
class ExampleSearch(Searcher):  
    def __init__(self, metric="mean_loss", mode="min", **kwargs):  
        super(ExampleSearch, self).__init__(  
            metric=metric, mode=mode, **kwargs)  
        self.optimizer = Optimizer()  
        self.configurations = {}  
  
    def suggest(self, trial_id):  
        configuration = self.optimizer.query()  
        self.configurations[trial_id] = configuration  
  
    def on_trial_complete(self, trial_id, result, **kwargs):  
        configuration = self.configurations[trial_id]  
        if result and self.metric in result:  
            self.optimizer.update(configuration, result[self.metric])  
  
tune.run(trainable_function, search_alg=ExampleSearch())
```

## 5.13 Tune API Reference

This section contains a reference for the Tune API. If there is anything missing, please open an issue on [Github](#).

### 5.13.1 Training (tune.run, tune.Experiment)

#### tune.run

```
ray.tune.run(run_or_experiment, name=None, stop=None, config=None, resources_per_trial=None,
             num_samples=1, local_dir=None, upload_dir=None, trial_name_creator=None,
             loggers=None, sync_to_cloud=None, sync_to_driver=None, checkpoint_freq=0,
             checkpoint_at_end=False, sync_on_checkpoint=True, keep_checkpoints_num=None,
             checkpoint_score_attr=None, global_checkpoint_period=10, export_formats=None,
             max_failures=0, fail_fast=False, restore=None, search_alg=None, scheduler=None,
             with_server=False, server_port=4321, verbose=2, progress_reporter=None, re-
             sume=False, queue_trials=False, reuse_actors=False, trial_executor=None,
             raise_on_failed_trial=True, return_trials=False, ray_auto_init=True)
```

Executes training.

#### Parameters

- **run\_or\_experiment** (function | class | str | *Experiment*) – If function|class|str, this is 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. If Experiment, then Tune will execute training based on Experiment.spec.
- **name** (str) – Name of experiment.
- **stop** (dict | callable | *Stopper*) – Stopping criteria. If dict, the keys may be any field in the return result of ‘train()’, whichever is reached first. If function, it must take (trial\_id, result) as arguments and return a boolean (True if trial should be stopped, False otherwise). This can also be a subclass of `ray.tune.Stopper`, which allows users to implement custom experiment-wide stopping (i.e., stopping an entire Tune run based on some time constraint).
- **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 and checkpoints to (e.g. s3://bucket or gs://bucket).
- **trial\_name\_creator** (func) – Optional function for generating the trial string representation.
- **loggers** (list) – List of logger creators to be used with each Trial. If None, defaults to `ray.tune.logger.DEFAULT_LOGGERS`. See `ray/tune/logger.py`.

- **sync\_to\_cloud** (*func/str*) – Function for syncing the local\_dir to and from upload\_dir. If string, then it must be a string template that includes *{source}* and *{target}* for the syncer to run. If not provided, the sync command defaults to standard S3 or gsutil sync commands.
- **sync\_to\_driver** (*func/str/bool*) – Function for syncing trial logdir from remote node to local. If string, then it must be a string template that includes *{source}* and *{target}* for the syncer to run. If True or not provided, it defaults to using rsync. If False, syncing to driver is disabled.
- **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.
- **sync\_on\_checkpoint** (*bool*) – Force sync-down of trial checkpoint to driver. If set to False, checkpoint syncing from worker to driver is asynchronous and best-effort. This does not affect persistent storage syncing. Defaults to True.
- **keep\_checkpoints\_num** (*int*) – Number of checkpoints to keep. A value of *None* keeps all checkpoints. Defaults to *None*. If set, need to provide *checkpoint\_score\_attr*.
- **checkpoint\_score\_attr** (*str*) – Specifies by which attribute to rank the best checkpoint. Default is increasing order. If attribute starts with *min-* it will rank attribute in decreasing order, i.e. *min-validation\_loss*.
- **global\_checkpoint\_period** (*int*) – Seconds between global checkpointing. This does not affect *checkpoint\_freq*, which specifies frequency for individual trials.
- **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 at least this many times. Ray will recover from the latest checkpoint if present. Setting to -1 will lead to infinite recovery retries. Setting to 0 will disable retries. Defaults to 3.
- **fail\_fast** (*bool*) – Whether to fail upon the first error.
- **restore** (*str*) – Path to checkpoint. Only makes sense to set if running 1 trial. Defaults to None.
- **search\_alg** (*Searcher*) – Search algorithm for optimization.
- **scheduler** (*TrialScheduler*) – Scheduler for executing the experiment. Choose among FIFO (default), MedianStopping, AsyncHyperBand, HyperBand and Population-BasedTraining. Refer to ray.tune.schedulers for more options.
- **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.
- **progress\_reporter** (*ProgressReporter*) – Progress reporter for reporting intermediate experiment progress. Defaults to CLIReporter if running in command-line, or JupyterNotebookReporter if running in a Jupyter notebook.
- **resume** (*str/bool*) – One of “LOCAL”, “REMOTE”, “PROMPT”, or bool. LOCAL/True restores the checkpoint from the local\_checkpoint\_dir. REMOTE restores the

checkpoint from remote\_checkpoint\_dir. PROMPT provides CLI feedback. False forces a new experiment. If resume is set but checkpoint does not exist, ValueError will be thrown.

- **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.
- **reuse\_actors** (*bool*) – Whether to reuse actors between different trials when possible. This can drastically speed up experiments that start and stop actors often (e.g., PBT in time-multiplexing mode). This requires trials to have the same resource requirements.
- **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.
- **ray\_auto\_init** (*bool*) – Automatically starts a local Ray cluster if using a RayTrialExecutor (which is the default) and if Ray is not initialized. Defaults to True.

**Returns** Object for experiment analysis.

**Return type** *ExperimentAnalysis*

**Raises** **TuneError** – Any trials failed and *raise\_on\_failed\_trial* is True.

Examples:

```
# Run 10 trials (each trial is one instance of a Trainable). Tune runs
# in parallel and automatically determines concurrency.
tune.run(trainable, num_samples=10)

# Run 1 trial, stop when trial has reached 10 iterations
tune.run(my_trainable, stop={"training_iteration": 10})

# Run 1 trial, search over hyperparameters, stop after 10 iterations.
space = {"lr": tune.uniform(0, 1), "momentum": tune.uniform(0, 1)}
tune.run(my_trainable, config=space, stop={"training_iteration": 10})
```

## tune.run\_experiments

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

Runs and blocks until all trials finish.

### 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.

## tune.Experiment

```
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, loggers=None, sync_to_driver=None, check-  
                     point_freq=0, checkpoint_at_end=False, sync_on_checkpoint=True,  
                     keep_checkpoints_num=None, checkpoint_score_attr=None, ex-  
                     port_formats=None, max_failures=0, restore=None)
```

Tracks experiment specifications.

Implicitly registers the Trainable if needed. The args here take the same meaning as the arguments defined *tune.py:run*.

```
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",  
    checkpoint_freq=10,  
    max_failures=2)
```

## Stopper (tune.Stopper)

```
class ray.tune.Stopper
```

Base class for implementing a Tune experiment stopper.

Allows users to implement experiment-level stopping via `stop_all`. By default, this class does not stop any trials. Subclasses need to implement `__call__` and `stop_all`.

```
import time  
from ray import tune  
from ray.tune import Stopper  
  
class TimeStopper(Stopper):  
    def __init__(self):  
        self._start = time.time()  
        self._deadline = 300
```

(continues on next page)

(continued from previous page)

```

def __call__(self, trial_id, result):
    return False

def stop_all(self):
    return time.time() - self._start > self.deadline

tune.run(Trainable, num_samples=200, stop=TimeStopper())

```

**`__call__(trial_id, result)`**

Returns true if the trial should be terminated given the result.

**`stop_all()`**

Returns true if the experiment should be terminated.

### 5.13.2 Training (`tune.Trainable`, `tune.track`)

Training can be done with either a **Class API** (`tune.Trainable`) or **function-based API** (`track.log`).

You can use the **function-based API** for fast prototyping. On the other hand, the `tune.Trainable` interface supports checkpoint/restore functionality and provides more control for advanced algorithms.

For the sake of example, let's maximize this objective function:

```

def objective(x, a, b):
    return a * (x ** 0.5) + b

```

#### Function-based API

```

def trainable(config):
    # config (dict): A dict of hyperparameters.

    for x in range(20):
        score = objective(x, config["a"], config["b"])

        tune.track.log(score=score) # This sends the score to Tune.

analysis = tune.run(
    trainable,
    config={
        "a": 2,
        "b": 4
    })

print("best config: ", analysis.get_best_config(metric="score", mode="max"))

```

---

**Tip:** Do not use `tune.track.log` within a `Trainable` class.

---

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.

---

**Note:** If you want to pass in a Python lambda, 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.

---

## Trainable Class API

**Caution:** Do not use `tune.track.log` within a Trainable class.

The Trainable **class API** will require users to subclass `ray.tune.Trainable`. Here's a naive example of this API:

```
from ray import tune

class Trainable(tune.Trainable):
    def _setup(self, config):
        # config (dict): A dict of hyperparameters
        self.x = 0
        self.a = config["a"]
        self.b = config["b"]

    def _train(self): # This is called iteratively.
        score = objective(self.x, self.a, self.b)
        self.x += 1
        return {"score": score}

analysis = tune.run(
    Trainable,
    stop={"training_iteration": 20},
    config={
        "a": 2,
        "b": 4
    })

print('best config: ', analysis.get_best_config(metric="score", mode="max"))
```

As a subclass of `tune.Trainable`, Tune will create a `Trainable` object on a separate process (using the [Ray Actor API](#)).

1. `_setup` function is invoked once training starts.
2. `_train` is invoked **multiple times**. Each time, the `Trainable` object executes one logical iteration of training in the tuning process, which may include one or more iterations of actual training.
3. `_stop` is invoked when training is finished.

---

**Tip:** As a rule of thumb, the execution time of `_train` 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).

---

In this example, we only implemented the `_setup` and `_train` methods for simplification. Next, we'll implement `_save` and `_restore` for checkpoint and fault tolerance.

## Save and Restore

Many Tune features rely on `_save`, and `_restore`, including the usage of certain Trial Schedulers, fault tolerance, and checkpointing.

```
class MyTrainableClass(Trainable):
    def _save(self, tmp_checkpoint_dir):
        checkpoint_path = os.path.join(tmp_checkpoint_dir, "model.pth")
        torch.save(self.model.state_dict(), checkpoint_path)
        return tmp_checkpoint_dir

    def _restore(self, tmp_checkpoint_dir):
        checkpoint_path = os.path.join(tmp_checkpoint_dir, "model.pth")
        self.model.load_state_dict(torch.load(checkpoint_path))
```

Checkpoints will be saved by training iteration to `local_dir/exp_name/trial_name/checkpoint_<iter>`. You can restore a single trial checkpoint by using `tune.run(restore=<checkpoint_dir>)`.

Tune also generates temporary checkpoints for pausing and switching between trials. For this purpose, it is important not to depend on absolute paths in the implementation of `save`.

Use `validate_save_restore` to catch `_save/_restore` errors before execution.

```
from ray.tune.utils import validate_save_restore

# both of these should return
validate_save_restore(MyTrainableClass)
validate_save_restore(MyTrainableClass, use_object_store=True)
```

## Advanced Resource Allocation

Trainables can themselves be distributed. If your trainable function / class creates further Ray actors or tasks that also consume CPU / GPU resources, you will want to set `extra_cpu` or `extra_gpu` inside `tune.run` to reserve extra resource slots. For example, if a trainable class requires 1 GPU itself, but also launches 4 actors, each using another GPU, then you should set `"gpu": 1, "extra_gpu": 4`.

```
tune.run(
    my_trainable,
    name="my_trainable",
    resources_per_trial={
        "cpu": 1,
        "gpu": 1,
        "extra_gpu": 4
    }
)
```

The Trainable also provides the `default_resource_requests` interface to automatically declare the `resources_per_trial` based on the given configuration.

## Advanced: Reusing Actors

Your Trainable can often take a long time to start. To avoid this, you can do `tune.run(reuse_actors=True)` to reuse the same Trainable Python process and object for multiple hyperparameters.

This requires you to implement `Trainable.reset_config`, which provides a new set of hyperparameters. It is up to the user to correctly update the hyperparameters of your trainable.

```
class PytorchTrainable(tune.Trainable):
    """Train a Pytorch ConvNet."""

    def __init__(self, config):
        self.train_loader, self.test_loader = get_data_loaders()
        self.model = ConvNet()
        self.optimizer = optim.SGD(
            self.model.parameters(),
            lr=config.get("lr", 0.01),
            momentum=config.get("momentum", 0.9))

    def reset_config(self, new_config):
        for param_group in self.optimizer.param_groups:
            if "lr" in new_config:
                param_group["lr"] = new_config["lr"]
            if "momentum" in new_config:
                param_group["momentum"] = new_config["momentum"]

        self.model = ConvNet()
        self.config = new_config
        return True
```

## tune.Trainable

```
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 `_setup`, `_train`, `_save`, and `_restore` when subclassing `Trainable`.

Other implementation methods that may be helpful to override are `_log_result`, `reset_config`, `_stop`, and `_export_model`.

When using Tune, Tune will convert this class into a Ray actor, which runs on a separate process. Tune will also change the current working directory of this process to `self.logdir`.

`_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.

**\_log\_result (result)**

Subclasses can optionally override this to customize logging.

The logging here is done on the worker process rather than the driver. You may want to turn off driver logging via the `loggers` parameter in `tune.run` when overriding this function.

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

**\_restore (checkpoint)**

Subclasses should override this to implement `restore()`.

**Warning:** In this method, do not rely on absolute paths. The absolute path of the `checkpoint_dir` used in `_save` may be changed.

If `_save` returned a prefixed string, the prefix of the checkpoint string returned by `_save` may be changed. This is because trial pausing depends on temporary directories.

The directory structure under the `checkpoint_dir` provided to `_save` is preserved.

See the example below.

```
class Example(Trainable):
    def _save(self, checkpoint_path):
        print(checkpoint_path)
        return os.path.join(checkpoint_path, "my/check/point")

    def _restore(self, checkpoint):
        print(checkpoint)

>>> trainer = Example()
>>> obj = trainer.save_to_object()  # This is used when PAUSED.
<logdir>/tmpc8k_c_6hsave_to_object/checkpoint_0/my/check/point
>>> trainer.restore_from_object(obj)  # Note the different prefix.
<logdir>/tmpb87b5axfrestore_from_object/checkpoint_0/my/check/point
```

**Parameters checkpoint (str/dict)** – If dict, the return value is as returned by `_save`. If a string, then it is a checkpoint path that may have a different prefix than that returned by `_save`. The directory structure underneath the `checkpoint_dir` `_save` is preserved.

**\_save (tmp\_checkpoint\_dir)**

Subclasses should override this to implement `save()`.

**Warning:** Do not rely on absolute paths in the implementation of `_save` and `_restore`.

Use `validate_save_restore` to catch `_save/_restore` errors before execution.

```
>>> from ray.tune.utils import validate_save_restore
>>> validate_save_restore(MyTrainableClass)
>>> validate_save_restore(MyTrainableClass, use_object_store=True)
```

**Parameters tmp\_checkpoint\_dir (str)** – The directory where the checkpoint file must be stored. In a Tune run, if the trial is paused, the provided path may be temporary and moved.

**Returns** A dict or string. If string, the return value is expected to be prefixed by `tmp_checkpoint_dir`. If dict, the return value will be automatically serialized by Tune and passed to `_restore()`.

## Examples

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

```
>>> trainable._save("/tmp/bad_example")
"/tmp/NEW_CHECKPOINT_PATH/my_checkpoint_file" # This will error.
```

### `_setup(config)`

Subclasses should override this for custom initialization.

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

### `_stop()`

Subclasses should override this for any cleanup on stop.

If any Ray actors are launched in the Trainable (i.e., with a RLlib trainer), be sure to kill the Ray actor process here.

You can kill a Ray actor by calling `actor._ray_terminate_.remote()` on the actor.

### `_train()`

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

The return value will be automatically passed to the loggers. Users can also return `tune.result.DONE` or `tune.result.SHOULD_CHECKPOINT` as a key to manually trigger termination or checkpointing of this trial. Note that manual checkpointing only works when subclassing Trainables.

**Returns** A dict that describes training progress.

### `classmethod default_resource_request(config)`

Provides a static resource requirement for the given configuration.

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

```
@classmethod
def default_resource_request(cls, config):
    return Resources(
        cpu=0,
        gpu=0,
        extra_cpu=config["workers"],
        extra_gpu=int(config["use_gpu"]) * config["workers"])
```

**Returns** A `Resources` object consumed by Tune for queueing.

**Return type** `Resources`

### `delete_checkpoint(checkpoint_path)`

Deletes local copy of checkpoint.

**Parameters** `checkpoint_path(str)` – Path to checkpoint.

**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** (*Union[list, str]*) – Format or list of (str) 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.

**get\_config()**

Returns configuration passed in by Tune.

**reset\_config**(*new\_config*)

Resets configuration without restarting the trial.

This method is optional, but can be implemented to speed up algorithms such as PBT, and to allow performance optimizations such as running experiments with `reuse_actors=True`. Note that `self.config` need to be updated to reflect the latest parameter information in Ray logs.

**Parameters** **new\_config**(*dir*) – Updated hyperparameter configuration for the trainable.

**Returns** True if reset was successful else False.

**classmethod resource\_help**(*config*)

Returns a help string for configuring this trainable's resources.

**Parameters** **config**(*dict*) – The Trainer's config dict.

**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()`.

**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 or prefix that may be passed to `restore()`.

**Return type** str

**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.

**stop**()

Releases all resources used by this trainable.

**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()`. This is incremented after `_train()` is called.

*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.

**property iteration**

Current training iteration.

This value is automatically incremented every time `train()` is called and is automatically inserted into the training result dict.

**property logdir**

Directory of the results and checkpoints for this Trainable.

Tune will automatically sync this folder with the driver if execution is distributed.

Note that the current working directory will also be changed to this.

**property training\_iteration**

Current training iteration (same as `self.iteration`).

This value is automatically incremented every time `train()` is called and is automatically inserted into the training result dict.

**property trial\_id**

Trial ID for the corresponding trial of this Trainable.

This is not set if not using Tune.

```
trial_id = self.trial_id
```

**property trial\_name**

Trial name for the corresponding trial of this Trainable.

This is not set if not using Tune.

```
name = self.trial_name
```

## tune.DurableTrainable

```
class ray.tune.DurableTrainable(remote_checkpoint_dir, *args, **kwargs)
```

Abstract class for a remote-storage backed fault-tolerant Trainable.

Supports checkpointing to and restoring from remote storage. To use this class, implement the same private methods as ray.tune.Trainable (`_save`, `_train`, `_restore`, `reset_config`, `_setup`, `_stop`).

**Warning:** This class is currently **experimental** and may be subject to change.

Run this with Tune as follows. Setting `sync_to_driver=False` disables syncing to the driver to avoid keeping redundant checkpoints around, as well as preventing the driver from syncing up the same checkpoint.

See `tune/trainable.py`.

**remote\_checkpoint\_dir**

Upload directory (S3 or GS path).

**Type** str

**storage\_client**

Tune-internal interface for interacting with external storage.

```
>>> tune.run(MyDurableTrainable, sync_to_driver=False)
```

## tune.track

```
ray.tune.track.shutdown()
```

Cleans up the trial and removes it from the global context.

```
ray.tune.track.log(**kwargs)
```

Logs all keyword arguments.

```
import time
from ray import tune
from ray.tune import track

def run_me(config):
    for iter in range(100):
        time.sleep(1)
        track.log(hello="world", ray="tune")

analysis = tune.run(run_me)
```

**Parameters** `**kwargs` – Any key value pair to be logged by Tune. Any of these metrics can be used for early stopping or optimization.

```
ray.tune.track.trial_dir()
```

Returns the directory where trial results are saved.

This includes json data containing the session's parameters and metrics.

```
ray.tune.track.trial_name()
```

Trial name for the corresponding trial of this Trainable.

This is not set if not using Tune.

```
ray.tune.track.trial_id()  
Trial id for the corresponding trial of this Trainable.  
This is not set if not using Tune.
```

## KerasCallback

### StatusReporter

```
class ray.tune.function_runner.StatusReporter(result_queue, continue_semaphore,  
                                              trial_name=None, trial_id=None,  
                                              logdir=None)
```

Object passed into your function that you can report status through.

#### Example

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

**Raises StopIteration** – A StopIteration exception is raised if the trial has been signaled to stop.

### 5.13.3 Console Output (Reporters)

By default, Tune reports experiment progress periodically to the command-line as follows.

```
== Status ==  
Memory usage on this node: 11.4/16.0 GiB  
Using FIFO scheduling algorithm.  
Resources requested: 4/12 CPUs, 0/0 GPUs, 0.0/3.17 GiB heap, 0.0/1.07 GiB objects  
Result logdir: /Users/foo/ray_results/myexp  
Number of trials: 4 (4 RUNNING)  
+-----+-----+-----+-----+-----+  
| Trial name | status | loc | param1 | param2 | param3 |  
| acc | loss | total time (s) | iter |  
+-----+-----+-----+-----+-----+  
| MyTrainable_a826033a | RUNNING | 10.234.98.164:31115 | 0.303706 | 0.0761 | 0.4328 |  
| 0.1289 | 1.8572 | 7.54952 | 15 |
```

(continues on next page)

(continued from previous page)

MyTrainable_a8263fc6   RUNNING   10.234.98.164:31117   0.929276   0.158   0.3417 ↵  0.4865   1.6307   7.0501   14
MyTrainable_a8267914   RUNNING   10.234.98.164:31111   0.068426   0.0319   0.1147 ↵  0.9585   1.9603   7.0477   14
MyTrainable_a826b7bc   RUNNING   10.234.98.164:31112   0.729127   0.0748   0.1784 ↵  0.1797   1.7161   7.05715   14
+-----+-----+-----+-----+-----+-----+
↪+-----+-----+-----+-----+-----+-----+

Note that columns will be hidden if they are completely empty. The output can be configured in various ways by instantiating a `CLIRunner` instance (or `JupyterNotebookRunner` if you're using jupyter notebook). Here's an example:

```
from ray.tune import CLIRunner

# Limit the number of rows.
runner = CLIRunner(max_progress_rows=10)
# Add a custom metric column, in addition to the default metrics.
# Note that this must be a metric that is returned in your training results.
runner.add_metric_column("custom_metric")
tune.run(my_trainable, progress_runner=runner)
```

Extending `CLIRunner` lets you control reporting frequency. For example:

```
class ExperimentTerminationRunner(CLIRunner):
    def should_report(self, trials, done=False):
        """Reports only on experiment termination."""
        return done

tune.run(my_trainable, progress_runner=ExperimentTerminationRunner())

class TrialTerminationRunner(CLIRunner):
    def __init__(self):
        self.num_terminated = 0

    def should_report(self, trials, done=False):
        """Reports only on trial termination events."""
        old_num_terminated = self.num_terminated
        self.num_terminated = len([t for t in trials if t.status == Trial.TERMINATED])
        return self.num_terminated > old_num_terminated

tune.run(my_trainable, progress_runner=TrialTerminationRunner())
```

The default reporting style can also be overridden more broadly by extending the `ProgressReporter` interface directly. Note that you can print to any output stream, file etc.

```
from ray.tune import ProgressReporter

class CustomReporter(ProgressReporter):

    def should_report(self, trials, done=False):
        return True

    def report(self, trials, *sys_info):
        print(*sys_info)
        print("\n".join([str(trial) for trial in trials]))
```

(continues on next page)

(continued from previous page)

```
tune.run(my_trainable, progress_reporter=CustomReporter())
```

## CLIReporter

```
class ray.tune.CLIReporter(metric_columns=None, max_progress_rows=20,
                           max_error_rows=20, max_report_frequency=5)
    Command-line reporter
```

### Parameters

- **metric\_columns** (*dict[str, str]/list[str]*) – Names of metrics to include in progress table. If this is a dict, the keys should be metric names and the values should be the displayed names. If this is a list, the metric name is used directly.
- **max\_progress\_rows** (*int*) – Maximum number of rows to print in the progress table. The progress table describes the progress of each trial. Defaults to 20.
- **max\_error\_rows** (*int*) – Maximum number of rows to print in the error table. The error table lists the error file, if any, corresponding to each trial. Defaults to 20.
- **max\_report\_frequency** (*int*) – Maximum report frequency in seconds. Defaults to 5s.

```
add_metric_column(metric, representation=None)
```

Adds a metric to the existing columns.

### Parameters

- **metric** (*str*) – Metric to add. This must be a metric being returned in training step results.
- **representation** (*str*) – Representation to use in table. Defaults to *metric*.

## JupyterNotebookReporter

```
class ray.tune.JupyterNotebookReporter(overwrite, metric_columns=None,
                                       max_progress_rows=20, max_error_rows=20,
                                       max_report_frequency=5)
```

Jupyter notebook-friendly Reporter that can update display in-place.

### Parameters

- **overwrite** (*bool*) – Flag for overwriting the last reported progress.
- **metric\_columns** (*dict[str, str]/list[str]*) – Names of metrics to include in progress table. If this is a dict, the keys should be metric names and the values should be the displayed names. If this is a list, the metric name is used directly.
- **max\_progress\_rows** (*int*) – Maximum number of rows to print in the progress table. The progress table describes the progress of each trial. Defaults to 20.
- **max\_error\_rows** (*int*) – Maximum number of rows to print in the error table. The error table lists the error file, if any, corresponding to each trial. Defaults to 20.
- **max\_report\_frequency** (*int*) – Maximum report frequency in seconds. Defaults to 5s.

**add\_metric\_column**(*metric*, *representation*=None)

Adds a metric to the existing columns.

**Parameters**

- **metric** (*str*) – Metric to add. This must be a metric being returned in training step results.
- **representation** (*str*) – Representation to use in table. Defaults to *metric*.

**ProgressReporter****class ray.tune.ProgressReporter**

Abstract class for experiment progress reporting.

*should\_report()* is called to determine whether or not *report()* should be called. Tune will call these functions after trial state transitions, receiving training results, and so on.

**should\_report**(*trials*, *done*=False)

Returns whether or not progress should be reported.

**Parameters**

- **trials** (*list [Trial]*) – Trials to report on.
- **done** (*bool*) – Whether this is the last progress report attempt.

**report**(*trials*, *done*, \**sys\_info*)

Reports progress across trials.

**Parameters**

- **trials** (*list [Trial]*) – Trials to report on.
- **done** (*bool*) – Whether this is the last progress report attempt.
- **sys\_info** – System info.

**5.13.4 Analysis (tune.analysis)****Analyzing Results**

You can use the `ExperimentAnalysis` object for analyzing results. It is returned automatically when calling `tune.run`.

```
analysis = tune.run(
    trainable,
    name="example-experiment",
    num_samples=10,
)
```

Here are some example operations for obtaining a summary of your experiment:

```
# Get a dataframe for the last reported results of all of the trials
df = analysis.dataframe()

# Get a dataframe for the max accuracy seen for each trial
df = analysis.dataframe(metric="mean_accuracy", mode="max")
```

(continues on next page)

(continued from previous page)

```
# Get a dict mapping {trial logdir -> dataframes} for all trials in the experiment.
all_dataframes = analysis.trial_dataframes

# Get a list of trials
trials = analysis.trials
```

You may want to get a summary of multiple experiments that point to the same `local_dir`. For this, you can use the `Analysis` class.

```
from ray.tune import Analysis
analysis = Analysis("~/ray_results/example-experiment")
```

## ExperimentAnalysis

**class** `ray.tune.ExperimentAnalysis(experiment_checkpoint_path, trials=None)`  
 Bases: `ray.tune.analysis.experiment_analysis.Analysis`

Analyze results from a Tune experiment.

To use this class, the experiment must be executed with the `JsonLogger`.

### Parameters

- **experiment\_checkpoint\_path** (`str`) – Path to a json file representing an experiment state. Corresponds to `Experiment.local_dir/Experiment.name/experiment_state.json`
- **trials** (`list / None`) – List of trials that can be accessed via `analysis.trials`.

## Example

```
>>> tune.run(my_trainable, name="my_exp", local_dir "~/tune_results")
>>> analysis = ExperimentAnalysis(
>>>     experiment_checkpoint_path "~/tune_results/my_exp/state.json")
```

**get\_best\_trial** (`metric, mode='max', scope='all'`)

Retrieve the best trial object.

Compares all trials' scores on `metric`.

### Parameters

- **metric** (`str`) – Key for trial info to order on.
- **mode** (`str`) – One of [min, max].
- **scope** (`str`) – One of [all, last]. If `scope=last`, only look at each trial's final step for `metric`, and compare across trials based on `mode=[min,max]`. If `scope=all`, find each trial's min/max score for `metric` based on `mode`, and compare trials based on `mode=[min,max]`.

**get\_best\_config** (`metric, mode='max', scope='all'`)

Retrieve the best config corresponding to the trial.

Compares all trials' scores on `metric`.

### Parameters

- **metric** (`str`) – Key for trial info to order on.
- **mode** (`str`) – One of [min, max].

- **scope** (*str*) – One of [all, last]. If *scope=last*, only look at each trial’s final step for *metric*, and compare across trials based on *mode=[min,max]*. If *scope=all*, find each trial’s min/max score for *metric* based on *mode*, and compare trials based on *mode=[min,max]*.

**get\_best\_logdir** (*metric*, *mode='max'*, *scope='all'*)

Retrieve the logdir corresponding to the best trial.

Compares all trials’ scores on *metric*.

#### Parameters

- **metric** (*str*) – Key for trial info to order on.
- **mode** (*str*) – One of [min, max].
- **scope** (*str*) – One of [all, last]. If *scope=last*, only look at each trial’s final step for *metric*, and compare across trials based on *mode=[min,max]*. If *scope=all*, find each trial’s min/max score for *metric* based on *mode*, and compare trials based on *mode=[min,max]*.

**stats()**

Returns a dictionary of the statistics of the experiment.

**runner\_data()**

Returns a dictionary of the TrialRunner data.

## Analysis

**class ray.tune.Analysis(*experiment\_dir*)**

Analyze all results from a directory of experiments.

To use this class, the experiment must be executed with the JsonLogger.

**dataframe** (*metric=None*, *mode=None*)

Returns a pandas.DataFrame object constructed from the trials.

#### Parameters

- **metric** (*str*) – Key for trial info to order on. If None, uses last result.
- **mode** (*str*) – One of [min, max].

**Returns** Constructed from a result dict of each trial.

**Return type** pd.DataFrame

**get\_best\_config** (*metric*, *mode='max'*)

Retrieve the best config corresponding to the trial.

#### Parameters

- **metric** (*str*) – Key for trial info to order on.
- **mode** (*str*) – One of [min, max].

**get\_best\_logdir** (*metric*, *mode='max'*)

Retrieve the logdir corresponding to the best trial.

#### Parameters

- **metric** (*str*) – Key for trial info to order on.
- **mode** (*str*) – One of [min, max].

**get\_all\_configs** (*prefix=False*)

Returns a list of all configurations.

**Parameters** `prefix` (`bool`) – If True, flattens the config dict and prepends `config/`.

**Returns** List of all configurations of trials,

**Return type** List[dict]

**get\_trial\_checkpoints\_paths** (`trial, metric='training_iteration'`)

Gets paths and metrics of all persistent checkpoints of a trial.

**Parameters**

- `trial` (`Trial`) – The log directory of a trial, or a trial instance.

- `metric` (`str`) – key for trial info to return, e.g. “mean\_accuracy”. “training\_iteration” is used by default.

**Returns** List of [path, metric] for all persistent checkpoints of the trial.

**property trial\_dataframes**

List of all dataframes of the trials.

## 5.13.5 Grid/Random Search

### Overview

Tune has a native interface for specifying a grid search or random search. You can specify the search space via `tune.run(config=...)`.

Thereby, you can either use the `tune.grid_search` primitive to specify an axis of a grid search...

```
tune.run(  
    trainable,  
    config={"bar": tune.grid_search([True, False])})
```

... or one of the random sampling primitives to specify distributions (*Random Distributions API*):

```
tune.run(  
    trainable,  
    config={  
        "param1": tune.choice([True, False]),  
        "bar": tune.uniform(0, 10),  
        "alpha": tune.sample_from(lambda _: np.random.uniform(100) ** 2),  
        "const": "hello" # It is also ok to specify constant values.  
    })
```

**Caution:** If you use a Search Algorithm, 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.

To sample multiple times/run multiple trials, specify `tune.run(num_samples=N)`. If `grid_search` is provided as an argument, the *same* grid will be repeated N times.

```
# 13 different configs.  
tune.run(trainable config={  
    "x": tune.choice([0, 1, 2]),  
}  
)
```

(continues on next page)

(continued from previous page)

```

# 13 different configs.
tune.run(trainable, num_samples=13, config={
    "x": tune.choice([0, 1, 2]),
    "y": tune.randn([0, 1, 2]),
}
)

# 4 different configs.
tune.run(trainable, config={"x": tune.grid_search([1, 2, 3, 4])}, num_samples=1)

# 3 different configs.
tune.run(trainable, config={"x": grid_search([1, 2, 3])}, num_samples=1)

# 6 different configs.
tune.run(trainable, config={"x": tune.grid_search([1, 2, 3])}, num_samples=2)

# 9 different configs.
tune.run(trainable, num_samples=1, config={
    "x": tune.grid_search([1, 2, 3]),
    "y": tune.grid_search([a, b, c])
}
)

# 18 different configs.
tune.run(trainable, num_samples=2, config={
    "x": tune.grid_search([1, 2, 3]),
    "y": tune.grid_search([a, b, c])
}
)

# 45 different configs.
tune.run(trainable, num_samples=5, config={
    "x": tune.grid_search([1, 2, 3]),
    "y": tune.grid_search([a, b, c])
}
)

```

Note that grid search and random search primitives are inter-operable. Each can be used independently or in combination with each other.

```

# 6 different configs.
tune.run(trainable, num_samples=2, config={
    "x": tune.sample_from(...),
    "y": tune.grid_search([a, b, c])
}
)

```

In the below example, 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.

```

tune.run(
    my_trainable,
    name="my_trainable",
    # num_samples will repeat the entire config 10 times.
    num_samples=10
    config={
        # ``sample_from`` creates a generator to call the lambda once per trial.
        "alpha": tune.sample_from(lambda spec: np.random.uniform(100)),
        # ``sample_from`` also supports "conditional search spaces"
    }
)

```

(continues on next page)

(continued from previous page)

```

    "beta": tune.sample_from(lambda spec: spec.config.alpha * np.random.
→normal()),
    "nn_layers": [
        # tune.grid_search will make it so that all values are evaluated.
        tune.grid_search([16, 64, 256]),
        tune.grid_search([16, 64, 256]),
    ],
},
)

```

## Custom/Conditional Search Spaces

You'll often run into awkward search spaces (i.e., when one hyperparameter depends on another). Use `tune.sample_from(func)` to provide a **custom** callable function for generating a search space.

The parameter `func` should take in a `spec` object, which has a `config` namespace from which you can access other hyperparameters. This is useful for conditional distributions:

```

tune.run(
    ...,
    config={
        # A random function
        "alpha": tune.sample_from(lambda _: np.random.uniform(100)),
        # Use the `spec.config` namespace to access other hyperparameters
        "beta": tune.sample_from(lambda spec: spec.config.alpha * np.random.normal())
    }
)

```

Here's an example showing a 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.

```

tune.run(
    my_trainable,
    name="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]),
        ],
    }
)

```

## Random Distributions API

### tune.randn

```
ray.tune.randn(*args, **kwargs)
```

Wraps tune.sample\_from around np.random.randn.

tune.randn(10) is equivalent to tune.sample\_from(lambda \_: np.random.randn(10))

### tune.loguniform

```
ray.tune.loguniform(min_bound, max_bound, base=10)
```

Sugar for sampling in different orders of magnitude.

#### Parameters

- **min\_bound** (*float*) – Lower boundary of the output interval (1e-4)
- **max\_bound** (*float*) – Upper boundary of the output interval (1e-2)
- **base** (*float*) – Base of the log. Defaults to 10.

### tune.uniform

```
ray.tune.uniform(*args, **kwargs)
```

Wraps tune.sample\_from around np.random.uniform.

tune.uniform(1, 10) is equivalent to tune.sample\_from(lambda \_: np.random.uniform(1, 10))

### tune.choice

```
ray.tune.choice(*args, **kwargs)
```

Wraps tune.sample\_from around np.random.choice.

tune.choice(10) is equivalent to tune.sample\_from(lambda \_: np.random.choice(10))

### tune.sample\_from

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

Specify that tune should sample configuration values from this function.

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

## Grid Search API

```
ray.tune.grid_search(values)
```

Convenience method for specifying grid search over a value.

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

## Internals

### BasicVariantGenerator

```
class ray.tune.suggest.BasicVariantGenerator(shuffle=False)
```

Uses Tune's variant generation for resolving variables.

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

**Parameters** `shuffle` (`bool`) – Shuffles the generated list of configurations.

User API:

```
from ray import tune
from ray.tune.suggest import BasicVariantGenerator

searcher = BasicVariantGenerator()
tune.run(my_trainable_func, algo=searcher)
```

Internal API:

```
from ray.tune.suggest import BasicVariantGenerator

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

## 5.13.6 Search Algorithms (tune.suggest)

### Repeater

```
class ray.tune.suggest.Repeater(searcher, repeat=1, set_index=True)
```

A wrapper algorithm for repeating trials of same parameters.

Set `tune.run(num_samples=...)` to be a multiple of `repeat`. For example, set `num_samples=15` if you intend to obtain 3 search algorithm suggestions and repeat each suggestion 5 times. Any leftover trials (`num_samples` mod `repeat`) will be ignored.

It is recommended that you do not run an early-stopping TrialScheduler simultaneously.

#### Parameters

- **searcher** (`Searcher`) – Searcher object that the Repeater will optimize. Note that the Searcher will only see 1 trial among multiple repeated trials. The result/metric passed to the Searcher upon trial completion will be averaged among all repeats.
- **repeat** (`int`) – Number of times to generate a trial with a repeated configuration. Defaults to 1.

- **set\_index (bool)** – Sets a tune.suggest.repeater.TRIAL\_INDEX in Trainable/Function config which corresponds to the index of the repeated trial. This can be used for seeds. Defaults to True.

## ConcurrencyLimiter

```
class ray.tune.suggest.ConcurrencyLimiter(searcher, max_concurrent)
```

A wrapper algorithm for limiting the number of concurrent trials.

**Parameters** **searcher** (Searcher) – Searcher object that the ConcurrencyLimiter will manage.

Example:

```
from ray.tune.suggest import ConcurrencyLimiter
search_alg = HyperOptSearch(metric="accuracy")
search_alg = ConcurrencyLimiter(search_alg, max_concurrent=2)
tune.run(trainable, search_alg=search_alg)
```

## AxSearch

```
class ray.tune.suggest.ax.AxSearch(ax_client, mode='max', use_early_stopped_trials=None,
                                     max_concurrent=None)
```

A wrapper around Ax to provide trial suggestions.

Requires Ax to be installed. Ax is an open source tool from Facebook for configuring and optimizing experiments. More information can be found in <https://ax.dev/>.

**Parameters**

- **parameters** (list[dict]) – Parameters in the experiment search space. Required elements in the dictionaries are: “name” (name of this parameter, string), “type” (type of the parameter: “range”, “fixed”, or “choice”, string), “bounds” for range parameters (list of two values, lower bound first), “values” for choice parameters (list of values), and “value” for fixed parameters (single value).
- **objective\_name** (str) – Name of the metric used as objective in this experiment. This metric must be present in *raw\_data* argument to *log\_data*. This metric must also be present in the dict reported/returned by the Trainable.
- **mode** (str) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute. Defaults to “max”.
- **parameter\_constraints** (list[str]) – Parameter constraints, such as “x3 >= x4” or “x3 + x4 >= 2”.
- **outcome\_constraints** (list[str]) – Outcome constraints of form “metric\_name >= bound”, like “m1 <= 3.”
- **max\_concurrent** (int) – Deprecated.
- **use\_early\_stopped\_trials** – Deprecated.

```
from ax.service.ax_client import AxClient
from ray import tune
from ray.tune.suggest.ax import AxSearch

parameters = [
    {"name": "x1", "type": "range", "bounds": [0.0, 1.0]},
```

(continues on next page)

(continued from previous page)

```

        {"name": "x2", "type": "range", "bounds": [0.0, 1.0]},
    ]

def easy_objective(config):
    for i in range(100):
        intermediate_result = config["x1"] + config["x2"] * i
        tune.track.log(score=intermediate_result)

client = AxClient(enforce_sequential_optimization=False)
client.create_experiment(parameters=parameters, objective_name="score")
algo = AxSearch(client)
tune.run(easy_objective, search_alg=algo)

```

## BayesOptSearch

```
class ray.tune.suggest.bayesopt.BayesOptSearch(space, metric='episode_reward_mean',
                                                mode='max', utility_kwargs=None,
                                                random_state=1, verbose=0,
                                                max_concurrent=None,
                                                use_early_stopped_trials=None)
```

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.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **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.
- **max\_concurrent** – Deprecated.
- **use\_early\_stopped\_trials** – Deprecated.

```

from ray import tune
from ray.tune.suggest.bayesopt import BayesOptSearch

space = {
    'width': (0, 20),
    'height': (-100, 100),
}
algo = BayesOptSearch(space, metric="mean_loss", mode="min")
tune.run(my_func, algo=algo)

```

## TuneBOHB

```
class ray.tune.suggest.bohb.TuneBOHB(space, bohb_config=None, max_concurrent=10, metric='neg_mean_loss', mode='max')
```

BOHB suggestion component.

Requires HpBandSter and ConfigSpace to be installed. You can install HpBandSter and ConfigSpace with: pip install hpbandster ConfigSpace.

This should be used in conjunction with HyperBandForBOHB.

### Parameters

- **space** (*ConfigurationSpace*) – Continuous ConfigSpace search space. Parameters will be sampled from this space which will be used to run trials.
- **bohb\_config** (*dict*) – configuration for HpBandSter BOHB algorithm
- **max\_concurrent** (*int*) – Number of maximum concurrent trials. Defaults to 10.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.

Example:

```
import ConfigSpace as CS

config_space = CS.ConfigurationSpace()
config_space.add_hyperparameter(
    CS.UniformFloatHyperparameter('width', lower=0, upper=20))
config_space.add_hyperparameter(
    CS.UniformFloatHyperparameter('height', lower=-100, upper=100))
config_space.add_hyperparameter(
    CS.CategoricalHyperparameter(
        name='activation', choices=['relu', 'tanh']))

algo = TuneBOHB(
    config_space, max_concurrent=4, metric='mean_loss', mode='min')
bohb = HyperBandForBOHB(
    time_attr='training_iteration',
    metric='mean_loss',
    mode='min',
    max_t=100)
run(MyTrainableClass, scheduler=bohb, search_alg=algo)
```

## DragonflySearch

```
class ray.tune.suggest.dragonfly.DragonflySearch(optimizer, metric='episode_reward_mean', mode='max', points_to_evaluate=None, evaluated_rewards=None, **kwargs)
```

A wrapper around Dragonfly to provide trial suggestions.

Requires Dragonfly to be installed via pip install dragonfly-opt.

### Parameters

- **optimizer** (*dragonfly.opt.BlackboxOptimiser*) – Optimizer provided from dragonfly. Choose an optimiser that extends BlackboxOptimiser.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **points\_to\_evaluate** (*list of lists*) – A list of points you'd like to run first before sampling from the optimiser, e.g. these could be parameter configurations you already know work well to help the optimiser select good values. Each point is a list of the parameters using the order definition given by parameter\_names.
- **evaluated\_rewards** (*list*) – If you have previously evaluated the parameters passed in as points\_to\_evaluate you can avoid re-running those trials by passing in the reward attributes as a list so the optimiser can be told the results without needing to re-compute the trial. Must be the same length as points\_to\_evaluate.

```
from ray import tune
from dragonfly.opt.gp_bandit import EuclideanGPBandit
from dragonfly.exd.experiment_caller import EuclideanFunctionCaller
from dragonfly import load_config

domain_vars = [ {
    "name": "LiNO3_vol",
    "type": "float",
    "min": 0,
    "max": 7
}, {
    "name": "Li2SO4_vol",
    "type": "float",
    "min": 0,
    "max": 7
}, {
    "name": "NaClO4_vol",
    "type": "float",
    "min": 0,
    "max": 7
} ]

domain_config = load_config({"domain": domain_vars})
func_caller = EuclideanFunctionCaller(None,
    domain_config.domain.list_of_domains[0])
optimizer = EuclideanGPBandit(func_caller, ask_tell_mode=True)

algo = DragonflySearch(optimizer, metric="objective", mode="max")

tune.run(my_func, algo=algo)
```

## HyperOptSearch

```
class ray.tune.suggest.hyperopt.HyperOptSearch(space, metric='episode_reward_mean',
                                                mode='max', points_to_evaluate=None,
                                                n_initial_points=20, random_state_seed=None, gamma=0.25,
                                                max_concurrent=None,
                                                use_early_stopped_trials=None)
```

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. Trials of the current run can be saved using save method, trials of a previous run can be loaded using restore method, thus enabling a warm start feature.

### 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.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **points\_to\_evaluate** (*list*) – Initial parameter suggestions to be run first. This is for when you already have some good parameters you want hyperopt to run first to help the TPE algorithm make better suggestions for future parameters. Needs to be a list of dict of hyperopt-named variables. Choice variables should be indicated by their index in the list (see example)
- **n\_initial\_points** (*int*) – number of random evaluations of the objective function before starting to approximate it with tree parzen estimators. Defaults to 20.
- **random\_state\_seed** (*int, array\_like, None*) – seed for reproducible results. Defaults to None.
- **gamma** (*float in range (0, 1)*) – parameter governing the tree parzen estimators suggestion algorithm. Defaults to 0.25.
- **max\_concurrent** – Deprecated.
- **use\_early\_stopped\_trials** – Deprecated.

```
space = {
    'width': hp.uniform('width', 0, 20),
    'height': hp.uniform('height', -100, 100),
    'activation': hp.choice("activation", ["relu", "tanh"])
}
current_best_params = [
    {'width': 10,
     'height': 0,
     'activation': 0, # The index of "relu"
    }]
algo = HyperOptSearch(
    space, metric="mean_loss", mode="min",
    points_to_evaluate=current_best_params)
```

## NevergradSearch

```
class ray.tune.suggest.nevergrad.NevergradSearch(optimizer, parameter_names,
                                                metric='episode_reward_mean',
                                                mode='max', max_concurrent=None,
                                                **kwargs)
```

A wrapper around Nevergrad to provide trial suggestions.

Requires Nevergrad to be installed.

Nevergrad is an open source tool from Facebook for derivative free optimization of parameters and/or hyperparameters. It features a wide range of optimizers in a standard ask and tell interface. More information can be found at <https://github.com/facebookresearch/nevergrad>.

### Parameters

- **optimizer** (*nevergrad.optimization.Optimizer*) – Optimizer provided from Nevergrad.
- **parameter\_names** (*list*) – List of parameter names. Should match the dimension of the optimizer output. Alternatively, set to None if the optimizer is already instrumented with kwargs (see nevergrad v0.2.0+).
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **use\_early\_stopped\_trials** – Deprecated.
- **max\_concurrent** – Deprecated.

```
from nevergrad.optimization import optimizerlib

instrumentation = 1
optimizer = optimizerlib.OnePlusOne(instrumentation, budget=100)
algo = NevergradSearch(
    optimizer, ["lr"], metric="mean_loss", mode="min")
```

---

**Note:** In nevergrad v0.2.0+, optimizers can be instrumented. For instance, the following will specifies searching for “lr” from 1 to 2.

```
>>> from nevergrad.optimization import optimizerlib
>>> from nevergrad import instrumentation as inst
>>> lr = inst.var.Array(1).bounded(1, 2).asfloat()
>>> instrumentation = inst.Instrumentation(lr=lr)
>>> optimizer = optimizerlib.OnePlusOne(instrumentation, budget=100)
>>> algo = NevergradSearch(
    optimizer, None, metric="mean_loss", mode="min")
```

---

## SigOptSearch

```
class ray.tune.suggest.sigopt.SigOptSearch(space, name='Default Tune Experiment',
                                            max_concurrent=1, reward_attr=None, metric='episode_reward_mean', mode='max',
                                            **kwargs)
```

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*.

This module manages its own concurrency.

### 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.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.

Example:

```
space = [
    {
        'name': 'width',
        'type': 'int',
        'bounds': {
            'min': 0,
            'max': 20
        },
    },
    {
        'name': 'height',
        'type': 'int',
        'bounds': {
            'min': -100,
            'max': 100
        },
    },
]
algo = SigOptSearch(
    space, name="SigOpt Example Experiment",
    max_concurrent=1, metric="mean_loss", mode="min")
```

## SkOptSearch

```
class ray.tune.suggest.skopt.SkOptSearch(optimizer, parameter_names, metric='episode_reward_mean', mode='max', points_to_evaluate=None, evaluated_rewards=None, max_concurrent=None, use_early_stopped_trials=None)
```

A wrapper around skopt to provide trial suggestions.

Requires skopt to be installed.

### Parameters

- **optimizer** (*skopt.optimizer.Optimizer*) – Optimizer provided from skopt.
- **parameter\_names** (*list*) – List of parameter names. Should match the dimension of the optimizer output.
- **metric** (*str*) – The training result objective value attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **points\_to\_evaluate** (*list of lists*) – A list of points you'd like to run first before sampling from the optimiser, e.g. these could be parameter configurations you already know work well to help the optimiser select good values. Each point is a list of the parameters using the order definition given by parameter\_names.
- **evaluated\_rewards** (*list*) – If you have previously evaluated the parameters passed in as points\_to\_evaluate you can avoid re-running those trials by passing in the reward attributes as a list so the optimiser can be told the results without needing to re-compute the trial. Must be the same length as points\_to\_evaluate. (See tune/examples/skopt\_example.py)
- **max\_concurrent** – Deprecated.
- **use\_early\_stopped\_trials** – Deprecated.

### Example

```
>>> from skopt import Optimizer
>>> optimizer = Optimizer([(0, 20), (-100, 100)])
>>> current_best_params = [[10, 0], [15, -20]]
>>> algo = SkOptSearch(optimizer,
>>>                     ["width", "height"],
>>>                     metric="mean_loss",
>>>                     mode="min",
>>>                     points_to_evaluate=current_best_params)
```

## ZOOptSearch

```
class ray.tune.suggest.zoopt.ZOOptSearch(algo='asracos', budget=None, dim_dict=None,
                                         metric='episode_reward_mean', mode='min',
                                         **kwargs)
```

A wrapper around ZOOpt to provide trial suggestions.

Requires zoopt package ( $\geq 0.4.0$ ) to be installed. You can install it with the command: `pip install -U zoopt`.

### Parameters

- **algo** (*str*) – To specify an algorithm in zoopt you want to use. Only support ASRacos currently.
- **budget** (*int*) – Number of samples.
- **dim\_dict** (*dict*) – Dimension dictionary. For continuous dimensions: (continuous, search\_range, precision); For discrete dimensions: (discrete, search\_range, has\_order). More details can be found in zoopt package.
- **metric** (*str*) – The training result objective value attribute. Defaults to “episode\_reward\_mean”.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute. Defaults to “min”.

```
from ray.tune import run
from ray.tune.suggest.zoopt import ZOOptSearch
from zoopt import ValueType

dim_dict = {
    "height": (ValueType.CONTINUOUS, [-10, 10], 1e-2),
    "width": (ValueType.DISCRETE, [-10, 10], False)
}

config = {
    "num_samples": 200,
    "config": {
        "iterations": 10, # evaluation times
    },
    "stop": {
        "timesteps_total": 10 # custom stop rules
    }
}

zoopt_search = ZOOptSearch(
    algo="Asracos", # only support Asracos currently
    budget=config["num_samples"],
    dim_dict=dim_dict,
    metric="mean_loss",
    mode="min")

run(my_objective,
    search_alg=zoopt_search,
    name="zoopt_search",
    **config)
```

## SearchAlgorithm

```
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*)

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.

**is\_finished** ()

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

Can return True before all trials have finished executing.

**set\_finished** ()

Marks the search algorithm as finished.

## Searcher

```
class ray.tune.suggest.Searcher(metric='episode_reward_mean', mode='max',
                                 max_concurrent=None, use_early_stopped_trials=None)
```

Bases: object

Abstract class for wrapping suggesting algorithms.

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

Any subclass that implements `__init__` must also call the constructor of this class: `super(Subclass, self).__init__(...)`.

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

### Parameters

- **metric** (`str`) – The training result objective value attribute.
- **mode** (`str`) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.

```
class ExampleSearch(Searcher):
    def __init__(self, metric="mean_loss", mode="min", **kwargs):
        super(ExampleSearch, self).__init__(
            metric=metric, mode=mode, **kwargs)
        self.optimizer = Optimizer()
        self.configurations = {}

    def suggest(self, trial_id):
        configuration = self.optimizer.query()
        self.configurations[trial_id] = configuration

    def on_trial_complete(self, trial_id, result, **kwargs):
        configuration = self.configurations[trial_id]
        if result and self.metric in result:
            self.optimizer.update(configuration, result[self.metric])

tune.run(trainable_function, search_alg=ExampleSearch())
```

### on\_trial\_result(`trial_id, result`)

Optional notification for result during training.

Note that by default, the result dict may include NaNs or may not include the optimization metric. It is up to the subclass implementation to preprocess the result to avoid breaking the optimization process.

### Parameters

- **trial\_id** (`str`) – A unique string ID for the trial.
- **result** (`dict`) – Dictionary of metrics for current training progress. Note that the result dict may include NaNs or may not include the optimization metric. It is up to the subclass implementation to preprocess the result to avoid breaking the optimization process.

### on\_trial\_complete(`trial_id, result=None, error=False`)

Notification for the completion of trial.

Typically, this method is used for notifying the underlying optimizer of the result.

### Parameters

- **trial\_id** (*str*) – A unique string ID for the trial.
- **result** (*dict*) – Dictionary of metrics for current training progress. Note that the result dict may include NaNs or may not include the optimization metric. It is up to the subclass implementation to preprocess the result to avoid breaking the optimization process. Upon errors, this may also be None.
- **error** (*bool*) – True if the training process raised an error.

**suggest** (*trial\_id*)

Queries the algorithm to retrieve the next set of parameters.

**Parameters** **trial\_id** (*str*) – Trial ID used for subsequent notifications.

**Returns** Configuration for a trial, if possible.

**Return type** dict|None

**save** (*checkpoint\_dir*)

Save function for this object.

**restore** (*checkpoint\_dir*)

Restore function for this object.

**property metric**

The training result objective value attribute.

**property mode**

Specifies if minimizing or maximizing the metric.

## 5.13.7 Trial Schedulers (tune.schedulers)

### FIFOScheduler

```
class ray.tune.schedulers.FIFOScheduler
    Simple scheduler that just runs trials in submission order.
```

### HyperBandScheduler

```
class ray.tune.schedulers.HyperBandScheduler(time_attr='training_iteration',
                                              reward_attr=None,
                                              metric='episode_reward_mean',
                                              mode='max',
                                              max_t=81,
                                              reduction_factor=3)
```

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*, the name of the reported objective value *metric*, and if *metric* is to be maximized or minimized (*mode*). 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.
- **metric** (*str*) – The training result objective value attribute. Stopping procedures will use this attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric 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.
- **reduction\_factor** (*float*) – Same as *eta*. Determines how sharp the difference is between bracket space-time allocation ratios.

## ASHAScheduler/AsyncHyperBandScheduler

```
class ray.tune.schedulers.AsyncHyperBandScheduler(time_attr='training_iteration',  
                                              reward_attr=None,  
                                              metric='episode_reward_mean',  
                                              mode='max',  
                                              max_t=100,  
                                              grace_period=1,  
                                              reduc-  
tion_factor=4, brackets=1)
```

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://arxiv.org/abs/1810.05934>

## 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.
- **metric** (*str*) – The training result objective value attribute. Stopping procedures will use this attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric 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.

```
ray.tune.schedulers.ASHAScheduler
alias of ray.tune.schedulers.async_hyperband.AsyncHyperBandScheduler
```

## MedianStoppingRule

```
class ray.tune.schedulers.MedianStoppingRule(time_attr='time_total_s',
                                              reward_attr=None,
                                              metric='episode_reward_mean',
                                              mode='max',
                                              grace_period=60.0,
                                              min_samples_required=3,
                                              min_time_slice=0,
                                              hard_stop=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.
- **metric** (*str*) – The training result objective value attribute. Stopping procedures will use this attribute.
- **mode** (*str*) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric attribute.
- **grace\_period** (*float*) – Only stop trials at least this old in time. The mean will only be computed from this time onwards. The units are the same as the attribute named by *time\_attr*.
- **min\_samples\_required** (*int*) – Minimum number of trials to compute median over.
- **min\_time\_slice** (*float*) – Each trial runs at least this long before yielding (assuming it isn't stopped). Note: trials ONLY yield if there are not enough samples to evaluate performance for the current result AND there are other trials waiting to run. The units are the same as the attribute named by *time\_attr*.
- **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.

## PopulationBasedTraining

```
class ray.tune.schedulers.PopulationBasedTraining(time_attr='time_total_s',
                                                 reward_attr=None,
                                                 metric='episode_reward_mean',
                                                 mode='max',
                                                 perturbation_interval=60.0,
                                                 hyparam_mutations={},
                                                 quantile_fraction=0.25,
                                                 reparam_mutations={},
                                                 sample_probability=0.25,
                                                 custom_explore_fn=None,
                                                 log_config=True)
```

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. To run multiple trials, use `tune.run(num_samples=<int>)`.

In `{LOG_DIR}/{MY_EXPERIMENT_NAME}/`, all mutations are logged in `pbt_global.txt` and individual policy perturbations are recorded in `pbt_policy_{i}.txt`. Tune logs: [target trial tag, clone trial tag, target trial iteration, clone trial iteration, old config, new config] on each perturbation step.

### 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.
- **`metric`** (`str`) – The training result objective value attribute. Stopping procedures will use this attribute.
- **`mode`** (`str`) – One of {min, max}. Determines whether objective is minimizing or maximizing the metric 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`.
- **`quantile_fraction`** (`float`) – Parameters are transferred from the top `quantile_fraction` fraction of trials to the bottom `quantile_fraction` fraction. Needs to be between 0 and 0.5. Setting it to 0 essentially implies doing no exploitation at all.
- **`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`.
- **`log_config`** (`bool`) – Whether to log the ray config of each model to `local_dir` at each exploit. Allows config schedule to be reconstructed.

```
import random
from ray import tune
from ray.tune.schedulers import PopulationBasedTraining

pbt = PopulationBasedTraining(
    time_attr="training_iteration",
    metric="episode_reward_mean",
    mode="max",
    perturbation_interval=10, # every 10 `time_attr` units
```

(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],
}

tune.run({...}, num_samples=8, scheduler=pbt)

```

## TrialScheduler

**class** ray.tune.schedulers.TrialScheduler

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.

## 5.13.8 Loggers (tune.logger)

Tune has default loggers for Tensorboard, CSV, and JSON formats.

### Logging Path

Tune will log the results of each trial to a subfolder under a specified local dir, which defaults to `~/ray_results`.

```
# This logs to 2 different trial folders:
# ~/ray_results/trainable_name/trial_name_1 and ~/ray_results/trainable_name/trial_
# trial_name_2
# trainable_name and trial_name are autogenerated.
tune.run(trainable, num_samples=2)
```

You can specify the `local_dir` and `trainable_name`:

```
# This logs to 2 different trial folders:
# ./results/test_experiment/trial_name_1 and ./results/test_experiment/trial_name_2
# Only trial_name is autogenerated.
tune.run(trainable, num_samples=2, local_dir=".//results", name="test_experiment")
```

To specify custom trial folder names, you can pass use the `trial_name_creator` argument to `tune.run`. This takes a function with the following signature:

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

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

tune.run(
    MyTrainableClass,
    name="example-experiment",
    num_samples=1,
    trial_name_creator=trial_name_string
)
```

See the documentation on Trials: [Trial](#).

### Custom Loggers

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

```
from ray.tune.logger import DEFAULT_LOGGERS

tune.run(
    MyTrainableClass,
    name="experiment_name",
    loggers=DEFAULT_LOGGERS + (CustomLogger1, CustomLogger2)
)
```

These loggers will be called along with the default Tune loggers. All loggers must inherit the `Logger` interface ([Logger](#)). You can also check out `logger.py` for implementation details.

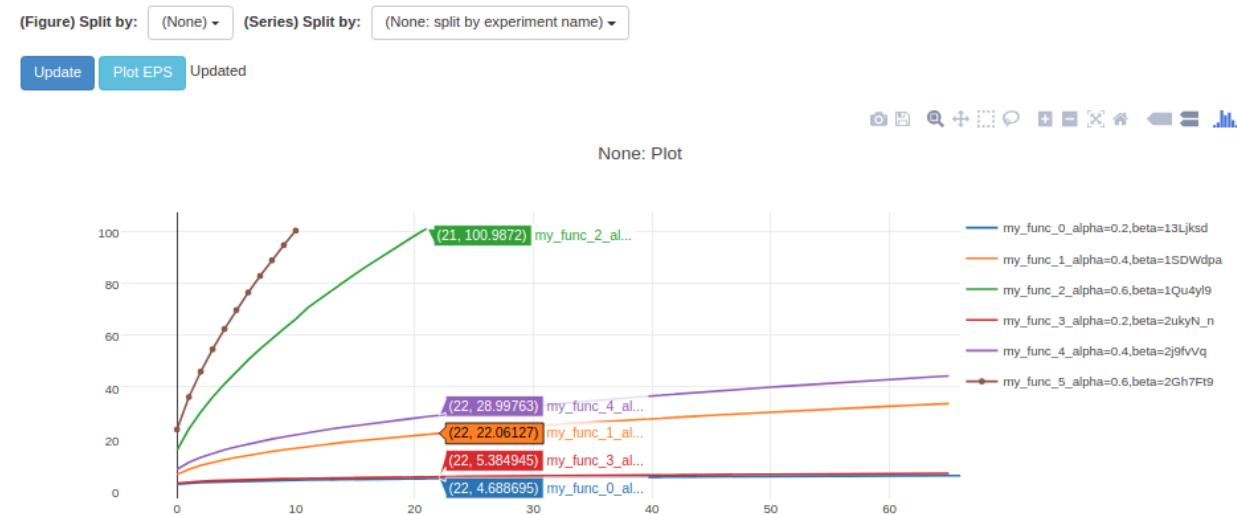
An example can be found in `logging_example.py`.

## Viskit

Tune automatically integrates with Viskit via the `CSVLogger` outputs. To use 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
```

The nonrelevant metrics (like timing stats) can be disabled on the left to show only the relevant ones (like accuracy, loss, etc.).



## Logger

```
class ray.tune.logger.Logger(config, logdir, trial=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.
- `trial` (`Trial`) – Trial object for the logger to access.

## UnifiedLogger

```
class ray.tune.logger.UnifiedLogger(config, logdir, trial=None, loggers=None, sync_function=None)
Unified result logger for TensorBoard, rllab/viskit, plain json.
```

### Parameters

- **config** – Configuration passed to all logger creators.
- **logdir** – Directory for all logger creators to log to.
- **loggers** (*list*) – List of logger creators. Defaults to CSV, Tensorboard, and JSON loggers.
- **sync\_function** (*func/str*) – Optional function for syncer to run. See ray/python/ray/tune/syncer.py

## TBXLogger

```
class ray.tune.logger.TBXLogger(config, logdir, trial=None)
TensorBoardX Logger.
```

Note that hparams will be written only after a trial has terminated. This logger automatically flattens nested dicts to show on TensorBoard:

```
{"a": {"b": 1, "c": 2}} -> {"a/b": 1, "a/c": 2}
```

## JsonLogger

```
class ray.tune.logger.JsonLogger(config, logdir, trial=None)
Logs trial results in json format.
```

Also writes to a results file and param.json file when results or configurations are updated. Experiments must be executed with the JsonLogger to be compatible with the ExperimentAnalysis tool.

## CSVLogger

```
class ray.tune.logger.CSVLogger(config, logdir, trial=None)
Logs results to progress.csv under the trial directory.
```

Automatically flattens nested dicts in the result dict before writing to csv:

```
{"a": {"b": 1, "c": 2}} -> {"a/b": 1, "a/c": 2}
```

## MLFLowLogger

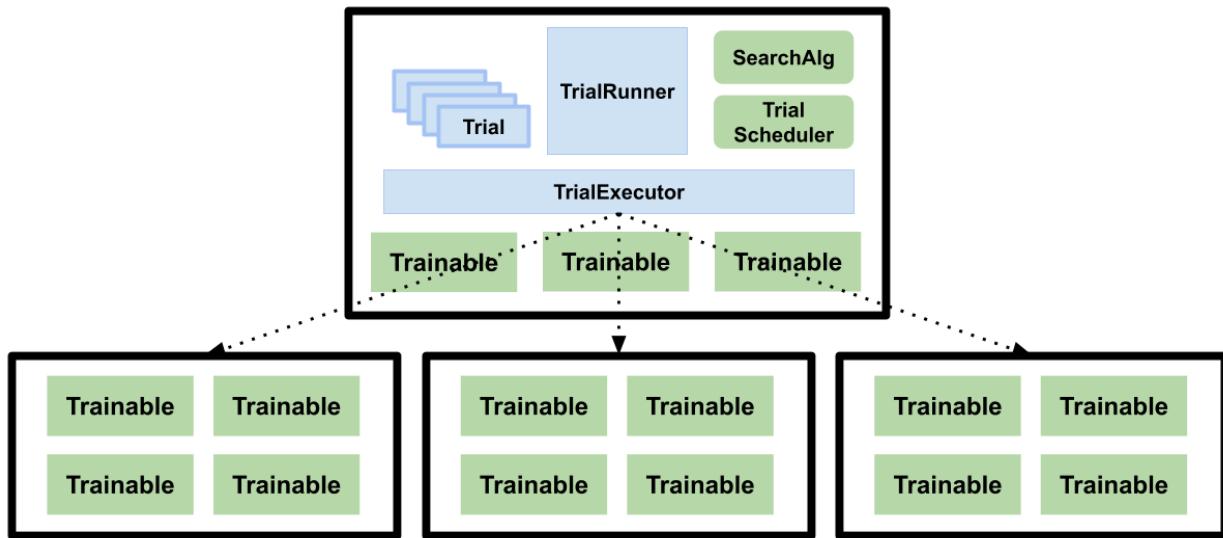
Tune also provides a default logger for [MLFlow](#). You can install MLFlow via `pip install mlflow`. An example can be found [mlflow\\_example.py](#). Note that this currently does not include artifact logging support. For this, you can use the native MLFlow APIs inside your Trainable definition.

```
class ray.tune.logger.MLFLowLogger(config, logdir, trial=None)
MLFlow logger.
```

Requires the experiment configuration to have a MLFlow Experiment ID or manually set the proper environment variables.

### 5.13.9 Tune Internals

This page overviews the design and architectures of Tune and provides docstrings for internal components.



The blue boxes refer to internal components, and green boxes are public-facing.

#### Main Components

Tune's main components consist of TrialRunner, Trial objects, TrialExecutor, SearchAlg, TrialScheduler, and Trainable.

#### TrialRunner

[source code] This is the main driver of the training loop. This component uses the TrialScheduler to prioritize and execute trials, queries the SearchAlgorithm for new configurations to evaluate, and handles the fault tolerance logic.

**Fault Tolerance:** The TrialRunner executes checkpointing if `checkpoint_freq` is set, along with automatic trial restarting in case of trial failures (if `max_failures` is set). For example, if a node is lost while a trial (specifically, the corresponding Trainable of the trial) is still executing on that node and checkpointing is enabled, the trial will then be reverted to a "PENDING" state and resumed from the last available checkpoint when it is run. The TrialRunner is also in charge of checkpointing the entire experiment execution state upon each loop iteration. This allows users to restart their experiment in case of machine failure.

See the docstring at [TrialRunner](#).

## Trial objects

[source code] This is an internal data structure that contains metadata about each training run. Each Trial object is mapped one-to-one with a Trainable object but are not themselves distributed/remote. Trial objects transition among the following states: "PENDING", "RUNNING", "PAUSED", "ERRORED", and "TERMINATED".

See the docstring at [Trial](#).

## TrialExecutor

[source code] The TrialExecutor is a component that interacts with the underlying execution framework. It also manages resources to ensure the cluster isn't overloaded. By default, the TrialExecutor uses Ray to execute trials.

See the docstring at [RayTrialExecutor](#).

## SearchAlg

[source code] The SearchAlgorithm is a user-provided object that is used for querying new hyperparameter configurations to evaluate.

SearchAlgorithms will be notified every time a trial finishes executing one training step (of `train()`), every time a trial errors, and every time a trial completes.

## TrialScheduler

[source code] TrialSchedulers operate over a set of possible trials to run, prioritizing trial execution given available cluster resources.

TrialSchedulers are given the ability to kill or pause trials, and also are given the ability to reorder/prioritize incoming trials.

## Trainables

[source code] These are user-provided objects that are used for the training process. If a class is provided, it is expected to conform to the Trainable interface. If a function is provided, it is wrapped into a Trainable class, and the function itself is executed on a separate thread.

Trainables will execute one step of `train()` before notifying the TrialRunner.

## RayTrialExecutor

```
class ray.tune.ray_trial_executor.RayTrialExecutor(queue_trials=False,
                                                 reuse_actors=False,
                                                 ray_auto_init=False,
                                                 re-
                                                 fresh_period=0.5)
```

Bases: [ray.tune.trial\\_executor.TrialExecutor](#)

An implementation of TrialExecutor based on Ray.

**start\_trial** (`trial, checkpoint=None, train=True`)

Starts the trial.

Will not return resources if trial repeatedly fails on start.

**Parameters**

- **trial** (`Trial`) – Trial to be started.
- **checkpoint** (`Checkpoint`) – A Python object or path storing the state of trial.
- **train** (`bool`) – Whether or not to start training.

**stop\_trial** (`trial, error=False, error_msg=None, stop_logger=True`)

Only returns resources if resources allocated.

**continue\_training** (`trial`)

Continues the training of this trial.

**pause\_trial** (`trial`)

Pauses the trial.

If trial is in-flight, preserves return value in separate queue before pausing, which is restored when Trial is resumed.

**reset\_trial** (`trial, new_config, new_experiment_tag`)

Tries to invoke `Trainable.reset_config()` to reset trial.

**Parameters**

- **trial** (`Trial`) – Trial to be reset.
- **new\_config** (`dict`) – New configuration for Trial trainable.
- **new\_experiment\_tag** (`str`) – New experiment name for trial.

**Returns** True if `reset_config` is successful else False.

**get\_running\_trials** ()

Returns the running trials.

**get\_next\_failed\_trial** ()

Gets the first trial found to be running on a node presumed dead.

**Returns** A Trial object that is ready for failure processing. None if no failure detected.

**get\_next\_available\_trial** ()

Blocking call that waits until one result is ready.

**Returns** Trial object that is ready for intermediate processing.

**fetch\_result** (`trial`)

Fetches one result of the running trials.

**Returns** Result of the most recent trial training run.

**has\_resources** (`resources`)

Returns whether this runner has at least the specified resources.

This refreshes the Ray cluster resources if the time since last update has exceeded `self._refresh_period`.

This also assumes that the cluster is not resizing very frequently.

**debug\_string** ()

Returns a human readable message for printing to the console.

**resource\_string** ()

Returns a string describing the total resources available.

**on\_step\_begin** (`trial_runner`)

Before step() called, update the available resources.

---

**save** (*trial, storage='persistent', result=None*)  
Saves the trial's state to a checkpoint asynchronously.

#### Parameters

- **trial** (`Trial`) – The trial to be saved.
- **storage** (`str`) – Where to store the checkpoint. Defaults to PERSISTENT.
- **result** (`dict`) – The state of this trial as a dictionary to be saved. If result is None, the trial's last result will be used.

**Returns** Checkpoint object, or None if an Exception occurs.

**restore** (*trial, checkpoint=None, block=False*)  
Restores training state from a given model checkpoint.

#### Parameters

- **trial** (`Trial`) – The trial to be restored.
- **checkpoint** (`Checkpoint`) – The checkpoint to restore from. If None, the most recent PERSISTENT checkpoint is used. Defaults to None.
- **block** (`bool`) – Whether or not to block on restore before returning.

#### Raises

- **RuntimeError** – This error is raised if no runner is found.
- **AbortTrialExecution** – This error is raised if the trial is ineligible for restoration, given the Tune input arguments.

**export\_trial\_if\_needed** (*trial*)  
Exports model of this trial based on trial.export\_formats.

**Returns** A dict that maps ExportFormats to successfully exported models.

**has\_gpus** ()  
Returns True if GPUs are detected on the cluster.

**cleanup** ()  
Ensures that trials are cleaned up after stopping.

## TrialExecutor

**class** ray.tune.trial\_executor.TrialExecutor (*queue\_trials=False*)  
Module for interacting with remote trainables.

Manages platform-specific details such as resource handling and starting/stopping trials.

**set\_status** (*trial, status*)  
Sets status and checkpoints metadata if needed.

Only checkpoints metadata if trial status is a terminal condition. PENDING, PAUSED, and RUNNING switches have checkpoints taken care of in the TrialRunner.

#### Parameters

- **trial** (`Trial`) – Trial to checkpoint.
- **status** (`Trial.status`) – Status to set trial to.

**try\_checkpoint\_metadata** (*trial*)  
Checkpoints trial metadata.

**Parameters** `trial` (`Trial`) – Trial to checkpoint.

**get\_checkpoints()**

Returns a copy of mapping of the trial ID to pickled metadata.

**has\_resources(resources)**

Returns whether this runner has at least the specified resources.

**start\_trial(trial, checkpoint=None, train=True)**

Starts the trial restoring from checkpoint if checkpoint is provided.

**Parameters**

- `trial` (`Trial`) – Trial to be started.
- `checkpoint` (`Checkpoint`) – A Python object or path storing the state
- `trial.of` –
- `train` (`bool`) – Whether or not to start training.

**stop\_trial(trial, error=False, error\_msg=None, stop\_logger=True)**

Stops the trial.

Stops this trial, releasing all allocating resources. If stopping the trial fails, the run will be marked as terminated in error, but no exception will be thrown.

**Parameters**

- `error` (`bool`) – Whether to mark this trial as terminated in error.
- `error_msg` (`str`) – Optional error message.
- `stop_logger` (`bool`) – Whether to shut down the trial logger.

**continue\_training(trial)**

Continues the training of this trial.

**pause\_trial(trial)**

Pauses the trial.

We want to release resources (specifically GPUs) when pausing an experiment. This results in PAUSED state that similar to TERMINATED.

**unpause\_trial(trial)**

Sets PAUSED trial to pending to allow scheduler to start.

**resume\_trial(trial)**

Resumes PAUSED trials. This is a blocking call.

**reset\_trial(trial, new\_config, new\_experiment\_tag)**

Tries to invoke `Trainable.reset_config()` to reset trial.

**Parameters**

- `trial` (`Trial`) – Trial to be reset.
- `new_config` (`dict`) – New configuration for Trial trainable.
- `new_experiment_tag` (`str`) – New experiment name for trial.

**Returns** True if `reset_config` is successful else False.

**get\_running\_trials()**

Returns all running trials.

**on\_step\_begin** (*trial\_runner*)  
A hook called before running one step of the trial event loop.

**on\_step\_end** (*trial\_runner*)  
A hook called after running one step of the trial event loop.

**get\_next\_available\_trial** ()  
Blocking call that waits until one result is ready.

**Returns** Trial object that is ready for intermediate processing.

**get\_next\_failed\_trial** ()  
Non-blocking call that detects and returns one failed trial.

**Returns** A Trial object that is ready for failure processing. None if no failure detected.

**fetch\_result** (*trial*)  
Fetches one result for the trial.

Assumes the trial is running.

**Returns** Result object for the trial.

**debug\_string** ()  
Returns a human readable message for printing to the console.

**resource\_string** ()  
Returns a string describing the total resources available.

**restore** (*trial*, *checkpoint=None*, *block=False*)  
Restores training state from a checkpoint.  
If checkpoint is None, try to restore from trial.checkpoint. If restoring fails, the trial status will be set to ERROR.

**Parameters**

- **trial** ([Trial](#)) – Trial to be restored.
- **checkpoint** ([Checkpoint](#)) – Checkpoint to restore from.
- **block** (`bool`) – Whether or not to block on restore before returning.

**Returns** False if error occurred, otherwise return True.

**save** (*trial*, *storage='persistent'*, *result=None*)  
Saves training state of this trial to a checkpoint.  
If result is None, this trial's last result will be used.

**Parameters**

- **trial** ([Trial](#)) – The state of this trial to be saved.
- **storage** (`str`) – Where to store the checkpoint. Defaults to PERSISTENT.
- **result** (`dict`) – The state of this trial as a dictionary to be saved.

**Returns** A Checkpoint object.

**export\_trial\_if\_needed** (*trial*)  
Exports model of this trial based on trial.export\_formats.

**Parameters** **trial** ([Trial](#)) – The state of this trial to be saved.

**Returns** A dict that maps ExportFormats to successfully exported models.

```
has_gpus()  
    Returns True if GPUs are detected on the cluster.  
  
cleanup(trial)  
    Ensures that trials are cleaned up after stopping.
```

## TrialRunner

```
class ray.tune.trial_runner.TrialRunner(search_alg=None, scheduler=None,  
                                         launch_web_server=False, lo-  
                                         cal_checkpoint_dir=None, re-  
                                         mote_checkpoint_dir=None, sync_to_cloud=None,  
                                         stopper=None, resume=False, server_port=4321,  
                                         fail_fast=False, verbose=True, check-  
                                         point_period=10, trial_executor=None)
```

A TrialRunner implements the event loop for scheduling trials on Ray.

The main job of TrialRunner is scheduling trials to efficiently use cluster resources, without overloading the cluster.

While Ray itself provides resource management for tasks and actors, this is not sufficient when scheduling trials that may instantiate multiple actors. This is because if insufficient resources are available, concurrent trials could deadlock waiting for new resources to become available. Furthermore, oversubscribing the cluster could degrade training performance, leading to misleading benchmark results.

### Parameters

- **search\_alg** ([SearchAlgorithm](#)) – SearchAlgorithm for generating Trial objects.
- **scheduler** ([TrialScheduler](#)) – Defaults to FIFO Scheduler.
- **launch\_web\_server** (`bool`) – Flag for starting TuneServer
- **local\_checkpoint\_dir** (`str`) – Path where global checkpoints are stored and restored from.
- **remote\_checkpoint\_dir** (`str`) – Remote path where global checkpoints are stored and restored from. Used if `resume == REMOTE`.
- **stopper** – Custom class for stopping whole experiments. See [Stopper](#).
- **resume** (`str/False`) – see `tune.py:run`.
- **sync\_to\_cloud** (`func/str`) – See `tune.py:run`.
- **server\_port** (`int`) – Port number for launching TuneServer.
- **fail\_fast** (`bool`) – Finishes as soon as a trial fails if True.
- **verbose** (`bool`) – Flag for verbosity. If False, trial results will not be output.
- **checkpoint\_period** (`int`) – Trial runner checkpoint periodicity in seconds. Defaults to 10.
- **trial\_executor** ([TrialExecutor](#)) – Defaults to RayTrialExecutor.

## Trial

```
class ray.tune.trial.Trial(trainable_name, config=None, trial_id=None, lo-
                           cal_dir='/home/docs/ray_results', evaluated_params=None,
                           experiment_tag='', resources=None, stop-
                           ping_criterion=None, remote_checkpoint_dir=None,
                           checkpoint_freq=0, checkpoint_at_end=False,
                           sync_on_checkpoint=True, keep_checkpoints_num=None, check-
                           point_score_attr='training_iteration', export_formats=None,
                           restore_path=None, trial_name_creator=None, loggers=None,
                           sync_to_driver_fn=None, max_failures=0)
```

A trial object holds the state for one model training run.

Trials are themselves managed by the TrialRunner class, which implements the event loop for submitting trial runs to a Ray cluster.

Trials start in the PENDING state, and transition to RUNNING once started. On error it transitions to ERROR, otherwise TERMINATED on success.

### **trainable\_name**

Name of the trainable object to be executed.

**Type** str

### **config**

Provided configuration dictionary with evaluated params.

**Type** dict

### **trial\_id**

Unique identifier for the trial.

**Type** str

### **local\_dir**

Local\_dir as passed to tune.run.

**Type** str

### **logdir**

Directory where the trial logs are saved.

**Type** str

### **evaluated\_params**

Evaluated parameters by search algorithm,

**Type** dict

### **experiment\_tag**

Identifying trial name to show in the console.

**Type** str

### **resources**

Amount of resources that this trial will use.

**Type** *Resources*

### **status**

One of PENDING, RUNNING, PAUSED, TERMINATED, ERROR/

**Type** str

**error\_file**

Path to the errors that this trial has raised.

**Type** str

## Resources

**class ray.tune.resources.Resources**

Ray resources required to schedule a trial.

### Parameters

- **cpu** (*float*) – Number of CPUs to allocate to the trial.
- **gpu** (*float*) – Number of GPUs to allocate to the trial.
- **memory** (*float*) – Memory to reserve for the trial.
- **object\_store\_memory** (*float*) – Object store memory to reserve.
- **extra\_cpu** (*float*) – Extra CPUs to reserve in case the trial needs to launch additional Ray actors that use CPUs.
- **extra\_gpu** (*float*) – Extra GPUs to reserve in case the trial needs to launch additional Ray actors that use GPUs.
- **extra\_memory** (*float*) – Memory to reserve for the trial launching additional Ray actors that use memory.
- **extra\_object\_store\_memory** (*float*) – Object store memory to reserve for the trial launching additional Ray actors that use object store memory.
- **custom\_resources** (*dict*) – Mapping of resource to quantity to allocate to the trial.
- **extra\_custom\_resources** (*dict*) – Extra custom resources to reserve in case the trial needs to launch additional Ray actors that use any of these custom resources.

## Registry

**ray.tune.register\_trainable(name, trainable)**

Register a trainable function or class.

This enables a class or function to be accessed on every Ray process in the cluster.

### 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.register\_env(name, env\_creator)**

Register a custom environment for use with RLlib.

This enables the environment to be accessed on every Ray process in the cluster.

### Parameters

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

### 5.13.10 Tune Client API

You can interact with an ongoing experiment with the Tune Client API. The Tune Client API is organized around REST, which includes resource-oriented URLs, accepts form-encoded requests, returns JSON-encoded responses, and uses standard HTTP protocol.

To allow Tune to receive and respond to your API calls, you have to start your experiment with `with_server=True`:

```
tune.run(..., with_server=True, server_port=4321)
```

The easiest way to use the Tune Client API is with the built-in TuneClient. To use TuneClient, verify that you have the `requests` library installed:

```
$ pip install requests
```

Then, on the client side, you can use the following class. 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, port\_forward)**

Client to interact with an ongoing Tune experiment.

Requires a TuneServer to have started running.

**tune\_address**

Address of running TuneServer

**Type** str

**port\_forward**

Port number of running TuneServer

**Type** int

**get\_all\_trials(timeout=None)**

Returns a list of all trials' information.

**get\_trial(trial\_id, timeout=None)**

Returns trial information by trial\_id.

**add\_trial(name, specification)**

Adds a trial by name and specification (dict).

**stop\_trial(trial\_id)**

Requests to stop trial by trial\_id.

**stop\_experiment()**

Requests to stop the entire experiment.

For an example notebook for using the Client API, see the [Client API Example](#).

The API also supports curl. Here are the examples for getting trials (GET /trials/[::id]):

```
$ curl http://<address>:<port>/trials
$ curl http://<address>:<port>/trials/<trial_id>
```

And stopping a trial (PUT /trials/:id):

```
$ curl -X PUT http://<address>:<port>/trials/<trial_id>
```

### 5.13.11 Tune CLI (Experimental)

tune has an easy-to-use command line interface (CLI) to manage and monitor your experiments on Ray. To do this, verify that you have the `tabulate` library installed:

```
$ pip install tabulate
```

Here are a few examples of command line calls.

- `tune list-trials`: List tabular information about trials within an experiment. Empty columns will be dropped by default. Add the `--sort` flag to sort the output by specific columns. Add the `--filter` flag to filter the output in the format "`<column> <operator> <value>`". Add the `--output` flag to write the trial information to a specific file (CSV or Pickle). Add the `--columns` and `--result-columns` flags to select specific columns to display.

```
$ tune list-trials [EXPERIMENT_DIR] --output note.csv

+-----+-----+-----+
| trainable_name | experiment_tag | trial_id |
+-----+-----+-----+
| MyTrainableClass | 0_height=40, width=37 | 87b54a1d |
| MyTrainableClass | 1_height=21, width=70 | 23b89036 |
| MyTrainableClass | 2_height=99, width=90 | 518dbe95 |
| MyTrainableClass | 3_height=54, width=21 | 7b99a28a |
| MyTrainableClass | 4_height=90, width=69 | ae4e02fb |
+-----+-----+-----+
Dropped columns: ['status', 'last_update_time']
Please increase your terminal size to view remaining columns.
Output saved at: note.csv

$ tune list-trials [EXPERIMENT_DIR] --filter "trial_id == 7b99a28a"

+-----+-----+-----+
| trainable_name | experiment_tag | trial_id |
+-----+-----+-----+
| MyTrainableClass | 3_height=54, width=21 | 7b99a28a |
+-----+-----+-----+
Dropped columns: ['status', 'last_update_time']
Please increase your terminal size to view remaining columns.
```

## 5.14 Contributing to Tune

We welcome (and encourage!) all forms of contributions to Tune, including and not limited to:

- Code reviewing of patches and PRs.
- Pushing patches.
- Documentation and examples.
- Community participation in forums and issues.
- Code readability and code comments to improve readability.
- Test cases to make the codebase more robust.
- Tutorials, blog posts, talks that promote the project.

### 5.14.1 Developing Tune

First, following the instructions in [Developing Ray \(Python Only\)](#) to develop Tune without compiling Ray.

After Ray is set up, run `pip install -r ray/python/ray/tune/requirements-dev.txt` to install all packages required for Tune development.

### 5.14.2 Submitting and Merging a Contribution

There are a couple steps to merge a contribution.

1. First rebase your development branch on the most recent version of master.

```
git remote add upstream https://github.com/ray-project/ray.git
git fetch upstream
git rebase upstream/master # or git pull . upstream/master
```

2. Make sure all existing tests pass.
3. If introducing a new feature or patching a bug, be sure to add new test cases in the relevant file in `tune/tests/`.
4. Document the code. Public functions need to be documented, and remember to provide a usage example if applicable.
5. Request code reviews from other contributors and address their comments. One fast way to get reviews is to help review others' code so that they return the favor. You should aim to improve the code as much as possible before the review. We highly value patches that can get in without extensive reviews.
6. Reviewers will merge and approve the pull request; be sure to ping them if the pull request is getting stale.

### 5.14.3 Testing

Even though we have hooks to run unit tests automatically for each pull request, we recommend you to run unit tests locally beforehand to reduce reviewers' burden and speedup review process.

```
pytest ray/python/ray/tune/tests/
```

Documentation should be documented in [Google style](#) format.

We also have tests for code formatting and linting that need to pass before merge. You can run the following locally:

```
ray/scripts/format.sh
```

### 5.14.4 What can I work on?

We use Github to track issues, feature requests, and bugs. Take a look at the ones labeled “good first issue” and “help wanted” for a place to start. Look for issues with “[tune]” in the title.

---

**Note:** If raising a new issue or PR related to Tune, be sure to include “[tune]” in the title and add a `tune` label.

---

For project organization, Tune maintains a relatively up-to-date organization of issues on the [Tune Github Project Board](#). Here, you can track and identify how issues are organized.

### 5.14.5 Becoming a Reviewer

We identify reviewers from active contributors. Reviewers are individuals who not only actively contribute to the project and are also willing to participate in the code review of new contributions. A pull request to the project has to be reviewed by at least one reviewer in order to be merged. There is currently no formal process, but active contributors to Tune will be solicited by current reviewers.

---

**Note:** These tips are based off of the TVM contributor guide.

---

## 5.15 RLLib: Scalable Reinforcement Learning

RLLib is an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLLib natively supports TensorFlow, TensorFlow Eager, and PyTorch, but most of its internals are framework agnostic.

To get started, take a look over the [custom env example](#) and the [API documentation](#). If you’re looking to develop custom algorithms with RLLib, also check out [concepts](#) and [custom algorithms](#).

---

**Important:** Join our [community slack](#) to discuss Ray/RLLib!

---

### 5.15.1 RLLib in 60 seconds

The following is a whirlwind overview of RLLib. For a more in-depth guide, see also the [full table of contents](#) and [RLLib blog posts](#). You may also want to skim the [list of built-in algorithms](#). Look out for the  and  icons to see which algorithms are [available](#) for each framework.

#### Running RLLib

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 ray[rllib] # also recommended: ray[debug]
```

Then, you can try out training in the following equivalent ways:

```
rllib train --run=PPO --env=CartPole-v0 # -v [-vv] for verbose,  
# --eager [--trace] for eager execution,  
# --torch to use PyTorch
```

```
from ray import tune  
from ray.rllib.agents.ppo import PPOTrainer  
tune.run(PPOTrainer, config={"env": "CartPole-v0"}) # "log_level": "INFO" for  
# verbose,  
# "eager": True for eager  
# execution,  
# "torch": True for PyTorch
```

Next, we'll cover three key concepts in RLlib: Policies, Samples, and Trainers.

## Policies

Policies are a core concept in RLlib. In a nutshell, policies are Python classes that define how an agent acts in an environment. Rollout workers query the policy to determine agent actions. In a gym environment, there is a single agent and policy. In vector envs, policy inference is for multiple agents at once, and in multi-agent, there may be multiple policies, each controlling one or more agents:

Policies can be implemented using any framework. However, for TensorFlow and PyTorch, RLlib has `build_tf_policy` and `build_torch_policy` helper functions that let you define a trainable policy with a functional-style API, for example:

```
def policy_gradient_loss(policy, model, dist_class, train_batch):
    logits, _ = model.from_batch(train_batch)
    action_dist = dist_class(logits, model)
    return -tf.reduce_mean(
        action_dist.logp(train_batch["actions"]) * train_batch["rewards"])

# <class 'ray.rllib.policy.tf_policy_template.MyTFPolicy'>
MyTFPolicy = build_tf_policy(
    name="MyTFPolicy",
    loss_fn=policy_gradient_loss)
```

## Sample Batches

Whether running in a single process or large cluster, all data interchange in RLlib is in the form of sample batches. Sample batches encode one or more fragments of a trajectory. Typically, RLlib collects batches of size `rollout_fragment_length` from rollout workers, and concatenates one or more of these batches into a batch of size `train_batch_size` that is the input to SGD.

A typical sample batch looks something like the following when summarized. Since all values are kept in arrays, this allows for efficient encoding and transmission across the network:

```
{ 'action_logp': np.ndarray((200,), dtype=float32, min=-0.701, max=-0.685, mean=-0.694),
  'actions': np.ndarray((200,), dtype=int64, min=0.0, max=1.0, mean=0.495),
  'dones': np.ndarray((200,), dtype=bool, min=0.0, max=1.0, mean=0.055),
  'infos': np.ndarray((200,), dtype=object, head={}),
  'new_obs': np.ndarray((200, 4), dtype=float32, min=-2.46, max=2.259, mean=0.018),
  'obs': np.ndarray((200, 4), dtype=float32, min=-2.46, max=2.259, mean=0.016),
  'rewards': np.ndarray((200,), dtype=float32, min=1.0, max=1.0, mean=1.0),
  't': np.ndarray((200,), dtype=int64, min=0.0, max=34.0, mean=9.14) }
```

In multi-agent mode, sample batches are collected separately for each individual policy.

## Training

Policies each define a `learn_on_batch()` method that improves the policy given a sample batch of input. For TF and Torch policies, this is implemented using a *loss function* that takes as input sample batch tensors and outputs a scalar loss. Here are a few example loss functions:

- Simple policy gradient loss
- Simple Q-function loss
- Importance-weighted [APPO surrogate loss](#)

RLLib [Trainer classes](#) coordinate the distributed workflow of running rollouts and optimizing policies. They do this by leveraging [policy optimizers](#) that implement the desired computation pattern. The following figure shows *synchronous sampling*, the simplest of [these patterns](#):

Fig. 1: Synchronous Sampling (e.g., A2C, PG, PPO)

RLLib uses [Ray actors](#) to scale training from a single core to many thousands of cores in a cluster. You can [configure the parallelism](#) used for training by changing the `num_workers` parameter. Check out our [scaling guide](#) for more details here.

## Application Support

Beyond environments defined in Python, RLLib supports batch training on [offline datasets](#), and also provides a variety of integration strategies for [external applications](#).

## Customization

RLLib provides ways to customize almost all aspects of training, including the [environment](#), [neural network model](#), [action distribution](#), and [policy definitions](#):

To learn more, proceed to the [table of contents](#).

## 5.16 RLLib Table of Contents

### 5.16.1 Training APIs

- Command-line
  - Evaluating Trained Policies
- Configuration
  - Specifying Parameters
  - Specifying Resources
  - Common Parameters
  - Scaling Guide
  - Tuned Examples
- Basic Python API

- Computing Actions
- Accessing Policy State
- Accessing Model State
- Advanced Python APIs
  - Custom Training Workflows
  - Global Coordination
  - Callbacks and Custom Metrics
  - Customizing Exploration Behavior
  - Customized Evaluation During Training
  - Rewriting Trajectories
  - Curriculum Learning
- Debugging
  - Gym Monitor
  - Eager Mode
  - Episode Traces
  - Log Verbosity
  - Stack Traces
- External Application API

### 5.16.2 Environments

- RLlib Environments Overview
- Feature Compatibility Matrix
- OpenAI Gym
- Vectorized
- Multi-Agent and Hierarchical
- External Agents and Applications
  - External Application Clients
- Advanced Integrations

### 5.16.3 Models, Preprocessors, and Action Distributions

- RLlib Models, Preprocessors, and Action Distributions Overview
- TensorFlow Models
- PyTorch Models
- Custom Preprocessors
- Custom Action Distributions
- Supervised Model Losses

- Self-Supervised Model Losses
- Variable-length / Parametric Action Spaces
- Autoregressive Action Distributions

## 5.16.4 Algorithms

- High-throughput architectures
  -   *Distributed Prioritized Experience Replay (Ape-X)*
  -  *Importance Weighted Actor-Learner Architecture (IMPALA)*
  -   *Asynchronous Proximal Policy Optimization (APPO)*
  -  *Decentralized Distributed Proximal Policy Optimization (DD-PPO)*
  -  *Single-Player AlphaZero (contrib/AlphaZero)*
- 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)*
  -   *Soft Actor Critic (SAC)*
- Derivative-free
  -   *Augmented Random Search (ARS)*
  -   *Evolution Strategies*
- Multi-agent specific
  -  *QMIX Monotonic Value Factorisation (QMIX, VDN, IQN)*
  -  *Multi-Agent Deep Deterministic Policy Gradient (contrib/MADDPG)*
- Offline
  -   *Advantage Re-Weighted Imitation Learning (MARWIL)*
- Contextual bandits
  -  *Linear Upper Confidence Bound (contrib/LinUCB)*
  -  *Linear Thompson Sampling (contrib/LintS)*

## 5.16.5 Offline Datasets

- Working with Offline Datasets
- Input Pipeline for Supervised Losses
- Input API
- Output API

## 5.16.6 Concepts and Custom Algorithms

- Policies
  - Policies in Multi-Agent
  - Building Policies in TensorFlow
  - Building Policies in TensorFlow Eager
  - Building Policies in PyTorch
  - Extending Existing Policies
- Policy Evaluation
- Policy Optimization
- Trainers

## 5.16.7 Examples

- Tuned Examples
- Training Workflows
- Custom Envs and Models
- Serving and Offline
- Multi-Agent and Hierarchical
- Community Examples

## 5.16.8 Development

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

## 5.16.9 Package Reference

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

## 5.16.10 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.

If you encounter out-of-memory errors, consider setting `redis_max_memory` and `object_store_memory` in `ray.init()` to reduce memory usage.

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, `ray timeline` to dump a timeline visualization of tasks to a file, and `ray memory` to list all object references in the cluster.

## TensorFlow 2.0

RLLib currently runs in `tf.compat.v1` mode. This means eager execution is disabled by default, and RLLib imports TF with `import tensorflow.compat.v1 as tf; tf.disable_v2_behavior()`. Eager execution can be enabled manually by calling `tf.enable_eager_execution()` or setting the "eager": `True` trainer config.

# 5.17 RLLib Training APIs

## 5.17.1 Getting Started

At a high level, RLLib provides an `Trainer` class which holds a policy for environment interaction. Through the trainer interface, the policy can be trained, checkpointed, or an action computed. In multi-agent training, the trainer manages the querying and optimization of multiple policies at once.

You can train a simple DQN trainer with the following command:

```
rllib train --run DQN --env CartPole-v0 # --eager [--trace] for eager execution
```

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/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 include SAC, PPO, PG, A2C, A3C, IMPALA, ES, DDPG, DQN, MARWIL, APEX, and APEX\_DDPG).

## Evaluating Trained Policies

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

An example of evaluating a previously trained DQN policy 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 policy 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`.

(Type `rllib rollout --help` to see the available evaluation options.)

For more advanced evaluation functionality, refer to [Customized Evaluation During Training](#).

## 5.17.2 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 algorithms. 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 trainers onto one GPU by setting `num_gpus: 0.2`.

## Scaling Guide

Here are some rules of thumb for scaling training with RLlib.

1. If the environment is slow and cannot be replicated (e.g., since it requires interaction with physical systems), then you should use a sample-efficient off-policy algorithm such as [DQN](#) or [SAC](#). These algorithms default to num\_workers: 0 for single-process operation. Consider also batch RL training with the [offline data API](#).
2. If the environment is fast and the model is small (most models for RL are), use time-efficient algorithms such as [PPO](#), [IMPALA](#), or [APEX](#). These can be scaled by increasing num\_workers to add rollout workers. It may also make sense to enable [vectorization](#) for inference. If the learner becomes a bottleneck, multiple GPUs can be used for learning by setting num\_gpus > 1.
3. If the model is compute intensive (e.g., a large deep residual network) and inference is the bottleneck, consider allocating GPUs to workers by setting num\_gpus\_per\_worker: 1. If you only have a single GPU, consider num\_workers: 0 to use the learner GPU for inference. For efficient use of GPU time, use a small number of GPU workers and a large number of envs per worker.
4. Finally, if both model and environment are compute intensive, then enable [remote worker envs](#) with [async batching](#) by setting remote\_worker\_envs: True and optionally remote\_env\_batch\_wait\_ms. This batches inference on GPUs in the rollout workers while letting envs run asynchronously in separate actors, similar to the [SEED](#) architecture. The number of workers and number of envs per worker should be tuned to maximize GPU utilization. If your env requires GPUs to function, or if multi-node SGD is needed, then also consider [DD-PPO](#).

## Common Parameters

The following is a list of the common algorithm hyperparameters:

```
COMMON_CONFIG = {
    # === Settings for Rollout Worker processes ===
    # Number of rollout worker actors to create for parallel sampling. Setting
    # this to 0 will force rollouts to be done in the trainer actor.
    "num_workers": 2,
    # Number of environments to evaluate vectorwise per worker. This enables
    # model inference batching, which can improve performance for inference
    # bottlenecked workloads.
    "num_envs_per_worker": 1,
    # Divide episodes into fragments of this many steps each during rollouts.
    # Sample batches of this size are collected from rollout workers and
    # combined into a larger batch of `train_batch_size` for learning.
    #
    # For example, given rollout_fragment_length=100 and train_batch_size=1000:
    #   1. RLlib collects 10 fragments of 100 steps each from rollout workers.
    #   2. These fragments are concatenated and we perform an epoch of SGD.
    #
    # When using multiple envs per worker, the fragment size is multiplied by
    # `num_envs_per_worker`. This is since we are collecting steps from
    # multiple envs in parallel. For example, if num_envs_per_worker=5, then
    # rollout workers will return experiences in chunks of 5*100 = 500 steps.
    #
    # The dataflow here can vary per algorithm. For example, PPO further
    # divides the train batch into minibatches for multi-epoch SGD.
    "rollout_fragment_length": 200,
    # Deprecated; renamed to `rollout_fragment_length` in 0.8.4.
    "sample_batch_size": DEPRECATED_VALUE,
    # Whether to rollout "complete_episodes" or "truncate_episodes" to
}
```

(continues on next page)

(continued from previous page)

```

# `rollout_fragment_length` length unrolls. Episode truncation guarantees
# evenly sized batches, but increases variance as the reward-to-go will
# need to be estimated at truncation boundaries.
"batch_mode": "truncate_episodes",

# === Settings for the Trainer process ===
# Number of GPUs to allocate to the trainer process. Note that not all
# algorithms can take advantage of trainer GPUs. This can be fractional
# (e.g., 0.3 GPUs).
"num_gpus": 0,
# Training batch size, if applicable. Should be >= rollout_fragment_length.
# Samples batches will be concatenated together to a batch of this size,
# which is then passed to SGD.
"train_batch_size": 200,
# Arguments to pass to the policy 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": {},

# === Environment Settings ===
# Discount factor of the MDP.
"gamma": 0.99,
# Number of steps after which the episode is forced to terminate. Defaults
# to `env.spec.max_episode_steps` (if present) for Gym envs.
"horizon": None,
# Calculate rewards but don't reset the environment when the horizon is
# hit. This allows value estimation and RNN state to span across logical
# episodes denoted by horizon. This only has an effect if horizon != inf.
"soft_horizon": False,
# Don't set 'done' at the end of the episode. Note that you still need to
# set this if soft_horizon=True, unless your env is actually running
# forever without returning done=True.
"no_done_at_end": False,
# Arguments to pass to the env creator.
"env_config": {},
# Environment name can also be passed via config.
"env": None,
# Unsquash actions to the upper and lower bounds of env's action space
"normalize_actions": False,
# 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",
# The default learning rate.
"lr": 0.0001,

# === Debug Settings ===
# Whether to write episode stats and videos to the agent log dir. This is
# typically located in ~/ray_results.
"monitor": False,
# Set the ray.rllib.* log level for the agent process and its workers.
# Should be one of DEBUG, INFO, WARN, or ERROR. The DEBUG level will also
# periodically print out summaries of relevant internal dataflow (this is

```

(continues on next page)

(continued from previous page)

```

# also printed out once at startup at the INFO level). When using the
# `rllib train` command, you can also use the `-v` and `--vv` flags as
# shorthand for INFO and DEBUG.
"log_level": "WARN",
# Callbacks that will be run during various phases of training. See the
# `DefaultCallbacks` class and `examples/custom_metrics_and_callbacks.py`
# for more usage information.
"callbacks": DefaultCallbacks,
# Whether to attempt to continue training if a worker crashes. The number
# of currently healthy workers is reported as the "num_healthy_workers"
# metric.
"ignore_worker_failures": False,
# Log system resource metrics to results. This requires `psutil` to be
# installed for sys stats, and `gputil` for GPU metrics.
"log_sys_usage": True,

# === Framework Settings ===
# Use PyTorch (instead of tf). If using `rllib train`, this can also be
# enabled with the `--torch` flag.
# NOTE: Some agents may not support `torch` yet and throw an error.
"use_pytorch": False,

# Enable TF eager execution (TF policies only). If using `rllib train`,
# this can also be enabled with the `--eager` flag.
"eager": False,
# Enable tracing in eager mode. This greatly improves performance, but
# makes it slightly harder to debug since Python code won't be evaluated
# after the initial eager pass.
"eager_tracing": False,
# Disable eager execution on workers (but allow it on the driver). This
# only has an effect if eager is enabled.
"no_eager_on_workers": False,

# === Exploration Settings ===
# Default exploration behavior, iff `explore` = None is passed into
# compute_action(s).
# Set to False for no exploration behavior (e.g., for evaluation).
"explore": True,
# Provide a dict specifying the Exploration object's config.
"exploration_config": {
    # The Exploration class to use. In the simplest case, this is the name
    # (str) of any class present in the `rllib.utils.exploration` package.
    # You can also provide the python class directly or the full location
    # of your class (e.g. "ray.rllib.utils.exploration.epsilon_greedy".
    # EpsilonGreedy").
    "type": "StochasticSampling",
    # Add constructor kwargs here (if any).
},
# === Evaluation Settings ===
# Evaluate with 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. If using multiple
# evaluation workers, we will run at least this many episodes total.
"evaluation_num_episodes": 10,

```

(continues on next page)

(continued from previous page)

```

# Internal flag that is set to True for evaluation workers.
"in_evaluation": False,
# Typical usage is to pass extra args to evaluation env creator
# and to disable exploration by computing deterministic actions.
# IMPORTANT NOTE: Policy gradient algorithms are able to find the optimal
# policy, even if this is a stochastic one. Setting "explore=False" here
# will result in the evaluation workers not using this optimal policy!
"evaluation_config": {
    # Example: overriding env_config, exploration, etc:
    # "env_config": {...},
    # "explore": False
},
# Number of parallel workers to use for evaluation. Note that this is set
# to zero by default, which means evaluation will be run in the trainer
# process. If you increase this, it will increase the Ray resource usage
# of the trainer since evaluation workers are created separately from
# rollout workers.
"evaluation_num_workers": 0,
# Customize the evaluation method. This must be a function of signature
# (trainer: Trainer, eval_workers: WorkerSet) -> metrics: dict. See the
# Trainer._evaluate() method to see the default implementation. The
# trainer guarantees all eval workers have the latest policy state before
# this function is called.
"custom_eval_function": None,
# EXPERIMENTAL: use the execution plan based API impl of the algo. Can also
# be enabled by setting RLLIB_EXEC_API=1.
"use_exec_api": False,

# === Advanced Rollout Settings ===
# Use a background thread for sampling (slightly off-policy, usually not
# advisable to turn on unless your env specifically requires it).
"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,
# Configures TF for single-process operation by default.
"tf_session_args": {
    # note: overridden by `local_tf_session_args`
    "intra_op_parallelism_threads": 2,
    "inter_op_parallelism_threads": 2,
    "gpu_options": {
        "allow_growth": True,
    },
    "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 worker
"local_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,
}
,
```

(continues on next page)

(continued from previous page)

```

# Whether to LZ4 compress individual observations
"compress_observations": False,
# Wait for metric batches for at most this many seconds. Those that
# have not returned in time will be collected in the next train iteration.
"collect_metrics_timeout": 180,
# Smooth metrics over this many episodes.
"metrics_smoothing_episodes": 100,
# If using num_envs_per_worker > 1, whether to create those new envs in
# remote processes instead of in the same worker. This adds overheads, but
# can make sense if your envs can take much time to step / reset
# (e.g., for StarCraft). Use this cautiously; overheads are significant.
"remote_worker_envs": False,
# Timeout that remote workers are waiting when polling environments.
# 0 (continue when at least one env is ready) is a reasonable default,
# but optimal value could be obtained by measuring your environment
# step / reset and model inference perf.
"remote_env_batch_wait_ms": 0,
# Minimum time per train iteration (frequency of metrics reporting).
"min_iter_time_s": 0,
# Minimum env steps to optimize for per train call. This value does
# not affect learning, only the length of train iterations.
"timesteps_per_iteration": 0,
# This argument, in conjunction with worker_index, sets the random seed of
# each worker, so that identically configured trials will have identical
# results. This makes experiments reproducible.
"seed": None,
# Any extra python env vars to set in the trainer process, e.g.,
# {"OMP_NUM_THREADS": "16"}
"extra_python_environ_for_driver": {},
# The extra python environments need to set for worker processes.
"extra_python_environ_for_worker": {},


# === Advanced Resource Settings ===
# Number of CPUs to allocate per worker.
"num_cpus_per_worker": 1,
# Number of GPUs to allocate per worker. This can be fractional. This is
# usually needed only if your env itself requires a GPU (i.e., it is a
# GPU-intensive video game), or model inference is unusually expensive.
"num_gpus_per_worker": 0,
# Any custom Ray resources to allocate per worker.
"custom_resources_per_worker": {},
# Number of CPUs to allocate for the trainer. Note: this only takes effect
# when running in Tune. Otherwise, the trainer runs in the main program.
"num_cpus_for_driver": 1,
# You can set these memory quotas to tell Ray to reserve memory for your
# training run. This guarantees predictable execution, but the tradeoff is
# if your workload exceeds the memory quota it will fail.
# Heap memory to reserve for the trainer process (0 for unlimited). This
# can be large if you are using large train batches, replay buffers, etc.
"memory": 0,
# Object store memory to reserve for the trainer process. Being large
# enough to fit a few copies of the model weights should be sufficient.
# This is enabled by default since models are typically quite small.
"object_store_memory": 0,
# Heap memory to reserve for each worker. Should generally be small unless
# your environment is very heavyweight.
"memory_per_worker": 0,

```

(continues on next page)

(continued from previous page)

```

# Object store memory to reserve for each worker. This only needs to be
# large enough to fit a few sample batches at a time. This is enabled
# by default since it almost never needs to be larger than ~200MB.
"object_store_memory_per_worker": 0,

# === Offline Datasets ===
# 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 has an effect when
# reading offline experiences. Available options:
# - "wis": the weighted step-wise importance sampling estimator.
# - "is": the step-wise importance sampling estimator.
# - "simulation": run the environment in the background, but use
#   this data for evaluation only and not for learning.
"input_evaluation": ["is", "wis"],
# Whether to run postprocess_trajectory() on the trajectory fragments from
# offline inputs. Note that postprocessing will be done using the *current*
# policy, not the *behavior* policy, which is typically undesirable for
# on-policy algorithms.
"postprocess_inputs": False,
# If positive, input batches will be shuffled via a sliding window buffer
# of this number of batches. Use this if the input data is not in random
# enough order. Input is delayed until the shuffle buffer is filled.
"shuffle_buffer_size": 0,
# 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,

# === Settings for Multi-Agent Environments ===
"multiagent": {
    # Map from policy ids to tuples of (policy_cls, obs_space,
    # act_space, config). See rollout_worker.py for more info.
    "policies": {},
    # 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
```

### 5.17.3 Basic 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
config["eager"] = False
trainer = ppo.PPOTrainer(config=config, env="CartPole-v0")

# Can optionally call trainer.restore(path) to load a checkpoint.

for i in range(1000):
    # Perform one iteration of training the policy with PPO
    result = trainer.train()
    print(pretty_print(result))

    if i % 100 == 0:
        checkpoint = trainer.save()
        print("checkpoint saved at", checkpoint)

# Also, in case you have trained a model outside of ray/RLlib and have created
# an h5-file with weight values in it, e.g.
# my_keras_model_trained_outside_rllib.save_weights("model.h5")
# (see: https://keras.io/models/about-keras-models/)

# ... you can load the h5-weights into your Trainer's Policy's ModelV2
# (tf or torch) by doing:
trainer.import_model("my_weights.h5")
# NOTE: In order for this to work, your (custom) model needs to implement
# the `import_from_h5` method.
# See https://github.com/ray-project/ray/blob/master/rllib/tests/test\_model\_imports.py
# for detailed examples for tf- and torch trainers/models.
```

---

**Note:** It's recommended that you run RLlib trainers with [Tune](#), for easy experiment management and visualization of results. Just set "run": ALG\_NAME, "env": ENV\_NAME in the experiment config.

---

All RLlib trainers 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
from ray import tune

ray.init()
tune.run(
    "PPO",
    stop={"episode_reward_mean": 200},
    config={
        "env": "CartPole-v0",
        "num_gpus": 0,
        "num_workers": 1,
        "lr": tune.grid_search([0.01, 0.001, 0.0001]),
        "eager": False,
    },
)
```

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
```

## Computing Actions

The simplest way to programmatically compute actions from a trained agent is to use `trainer.compute_action()`. This method preprocesses and filters the observation before passing it to the agent policy. For more advanced usage, you can access the `workers` and policies held by the trainer directly as `compute_action()` does:

```
class Trainer(Trainable):

    @PublicAPI
    def compute_action(self,
                       observation,
                       state=None,
                       prev_action=None,
                       prev_reward=None,
                       info=None,
                       policy_id=DEFAULT_POLICY_ID,
                       full_fetch=False):
        """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.

        Arguments:
            observation (obj): observation from the environment.
```

(continues on next page)

(continued from previous page)

```

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).
full_fetch (bool): whether to return extra action fetch results.
    This is always set to true if RNN state is specified.

>Returns:
Just the computed action if full_fetch=False, or the full output
of policy.compute_actions() otherwise.
"""

if state is None:
    state = []
preprocessed = self.workers.local_worker().preprocessors[
    policy_id].transform(observation)
filtered_obs = self.workers.local_worker().filters[policy_id](
    preprocessed, update=False)
if state:
    return self.get_policy(policy_id).compute_single_action(
        filtered_obs,
        state,
        prev_action,
        prev_reward,
        info,
        clip_actions=self.config["clip_actions"])
res = self.get_policy(policy_id).compute_single_action(
    filtered_obs,
    state,
    prev_action,
    prev_reward,
    info,
    clip_actions=self.config["clip_actions"])
if full_fetch:
    return res
else:
    return res[0] # backwards compatibility

```

## Accessing Policy State

It is common to need to access a trainer's internal state, e.g., to set or get internal weights. In RLLib trainer state is replicated across multiple *rollout workers* (Ray actors) in the cluster. However, you can easily get and update this state between calls to `train()` via `trainer.workers.foreach_worker()` or `trainer.workers.foreach_worker_with_index()`. These functions take a lambda function that is applied with the worker 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 trainer state through `trainer.get_policy()` or `trainer.workers.local_worker()`, 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 `trainer.get_policy().get_weights()`. This is also equivalent to `trainer.workers.`

```

local_worker().policy_map["default_policy"].get_weights():

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

# Same as above
trainer.workers.local_worker().policy_map["default_policy"].get_weights()

# Get list of weights of each worker, including remote replicas
trainer.workers.foreach_worker(lambda ev: ev.get_policy().get_weights())

# Same as above
trainer.workers.foreach_worker_with_index(lambda ev, i: ev.get_policy().get_weights())

```

## Accessing Model State

Similar to accessing policy state, you may want to get a reference to the underlying neural network model being trained. For example, you may want to pre-train it separately, or otherwise update its weights outside of RLlib. This can be done by accessing the `model` of the policy:

### Example: Preprocessing observations for feeding into a model

```

>>> import gym
>>> env = gym.make("Pong-v0")

# RLlib uses preprocessors to implement transforms such as one-hot encoding
# and flattening of tuple and dict observations.
>>> from ray.rllib.models.preprocessors import get_preprocessor
>>> prep = get_preprocessor(env.observation_space)(env.observation_space)
<ray.rllib.models.preprocessors.GenericPixelPreprocessor object at 0x7fc4d049de80>

# Observations should be preprocessed prior to feeding into a model
>>> env.reset().shape
(210, 160, 3)
>>> prep.transform(env.reset()).shape
(84, 84, 3)

```

### Example: Querying a policy's action distribution

```

# Get a reference to the policy
>>> from ray.rllib.agents.ppo import PPOTrainer
>>> trainer = PPOTrainer(env="CartPole-v0", config={"eager": True, "num_workers": 0})
>>> policy = trainer.get_policy()
<ray.rllib.policy.eager_tf_policy.PPOTFPolicy_eager object at 0x7fd020165470>

# Run a forward pass to get model output logits. Note that complex observations
# must be preprocessed as in the above code block.
>>> logits, _ = policy.model.from_batch({"obs": np.array([[0.1, 0.2, 0.3, 0.4]])})
(<tf.Tensor: id=1274, shape=(1, 2), dtype=float32, numpy=..., []>, [])

# Compute action distribution given logits
>>> policy.dist_class
<class 'ray.rllib.models.tf.tf_action_dist.Categorical'>
>>> dist = policy.dist_class(logits, policy.model)
<ray.rllib.models.tf.tf_action_dist.Categorical object at 0x7fd02301d710>

# Query the distribution for samples, sample logs

```

(continues on next page)

(continued from previous page)

```
>>> dist.sample()
<tf.Tensor: id=661, shape=(1,), dtype=int64, numpy=...>
>>> dist.logp([1])
<tf.Tensor: id=1298, shape=(1,), dtype=float32, numpy=...>

# Get the estimated values for the most recent forward pass
>>> policy.model.value_function()
<tf.Tensor: id=670, shape=(1,), dtype=float32, numpy=...>

>>> policy.model.base_model.summary()
Model: "model"

Layer (type)          Output Shape   Param #  Connected to
=====
observations (InputLayer)  [(None, 4)]    0
fc_1 (Dense)         (None, 256)     1280      observations[0][0]
fc_value_1 (Dense)   (None, 256)     1280      observations[0][0]
fc_2 (Dense)         (None, 256)     65792     fc_1[0][0]
fc_value_2 (Dense)   (None, 256)     65792     fc_value_1[0][0]
fc_out (Dense)        (None, 2)       514       fc_2[0][0]
value_out (Dense)    (None, 1)       257       fc_value_2[0][0]
=====
Total params: 134,915
Trainable params: 134,915
Non-trainable params: 0
```

**Example: Getting Q values from a DQN model**

```
# Get a reference to the model through the policy
>>> from ray.rllib.agents.dqn import DQNTrainer
>>> trainer = DQNTrainer(env="CartPole-v0", config={"eager": True})
>>> model = trainer.get_policy().model
<ray.rllib.models.catalog.FullyConnectedNetwork_as_DistributionalQModel ...>

# List of all model variables
>>> model.variables()
[<tf.Variable 'default_policy/fc_1/kernel:0' shape=(4, 256) dtype=float32>, ...]

# Run a forward pass to get base model output. Note that complex observations
# must be preprocessed. An example of preprocessing is examples/saving_experiences.py
>>> model_out = model.from_batch({"obs": np.array([[0.1, 0.2, 0.3, 0.4]])})
(<tf.Tensor: id=832, shape=(1, 256), dtype=float32, numpy=...>)

# Access the base Keras models (all default models have a base)
>>> model.base_model.summary()
Model: "model"

Layer (type)          Output Shape   Param #  Connected to
=====
observations (InputLayer)  [(None, 4)]    0
```

(continues on next page)

(continued from previous page)

fc_1 (Dense)	(None, 256)	1280	observations[0][0]
fc_out (Dense)	(None, 256)	65792	fc_1[0][0]
value_out (Dense)	(None, 1)	257	fc_1[0][0]
<hr/>			
Total params: 67,329			
Trainable params: 67,329			
Non-trainable params: 0			
<hr/>			
# Access the Q value model (specific to DQN)			
>>> model.get_q_value_distributions(model_out)			
[<tf.Tensor: id=891, shape=(1, 2)>, <tf.Tensor: id=896, shape=(1, 2, 1)>]			
>>> model.q_value_head.summary()			
Model: "model_1"			
Layer (type)	Output Shape	Param #	
<hr/>			
model_out (InputLayer)	[None, 256]	0	
<b>lambda</b> (Lambda)	[None, 2], [None, 2, 1],	66306	
<hr/>			
Total params: 66,306			
Trainable params: 66,306			
Non-trainable params: 0			
<hr/>			
# Access the state value model (specific to DQN)			
>>> model.get_state_value(model_out)			
<tf.Tensor: id=913, shape=(1, 1), dtype=float32>			
>>> model.state_value_head.summary()			
Model: "model_2"			
Layer (type)	Output Shape	Param #	
<hr/>			
model_out (InputLayer)	[None, 256]	0	
<b>lambda_1</b> (Lambda)	(None, 1)	66049	
<hr/>			
Total params: 66,049			
Trainable params: 66,049			
Non-trainable params: 0			

This is especially useful when used with [custom model classes](#).

## 5.17.4 Advanced Python APIs

### Custom Training Workflows

In the [basic training example](#), Tune will call `train()` on your trainer once per training 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](#)).

For even finer-grained control over training, you can use RLlib's lower-level [building blocks](#) directly to implement fully customized training workflows.

### 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.util 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 to implement communication patterns such as parameter servers and allreduce.

### Callbacks and Custom Metrics

You can provide callbacks to be called at points during policy evaluation. These callbacks have access to state for the current `episode`. Certain callbacks such as `on_postprocess_trajectory`, `on_sample_end`, and `on_train_result` are also places where custom postprocessing can be applied to intermediate data or results.

User-defined state can be stored for the `episode` in the `episode.user_data` dict, and custom scalar metrics reported by saving values to the `episode.custom_metrics` dict. These custom metrics will be aggregated and reported as part of training results. For a full example, see [custom\\_metrics\\_and\\_callbacks.py](#).

```
class ray.rllib.agents.callbacks.DefaultCallbacks(legacy_callbacks_dict: Dict[str, callable] = None)
```

Abstract base class for RLlib callbacks (similar to Keras callbacks).

These callbacks can be used for custom metrics and custom postprocessing.

By default, all of these callbacks are no-ops. To configure custom training callbacks, subclass DefaultCallbacks and then set {“callbacks”: YourCallbacksClass} in the trainer config.

```
on_episode_start(worker: ray.rllib.evaluation.rollout_worker.RolloutWorker, base_env: ray.rllib.env.base_env.BaseEnv, policies: Dict[str, ray.rllib.policy.policy.Policy], episode: ray.rllib.evaluation.episode.MultiAgentEpisode, **kwargs)
```

Callback run on the rollout worker before each episode starts.

#### Parameters

- **worker** (`RolloutWorker`) – Reference to the current rollout worker.
- **base\_env** (`BaseEnv`) – BaseEnv running the episode. The underlying env object can be gotten by calling `base_env.get_unwrapped()`.
- **policies** (`dict`) – Mapping of policy id to policy objects. In single agent mode there will only be a single “default” policy.
- **episode** (`MultiAgentEpisode`) – Episode object which contains episode state. You can use the `episode.user_data` dict to store temporary data, and `episode.custom_metrics` to store custom metrics for the episode.
- **kwargs** – Forward compatibility placeholder.

```
on_episode_step(worker: ray.rllib.evaluation.rollout_worker.RolloutWorker, base_env: ray.rllib.env.base_env.BaseEnv, episode: ray.rllib.evaluation.episode.MultiAgentEpisode, **kwargs)
```

Runs on each episode step.

#### Parameters

- **worker** (`RolloutWorker`) – Reference to the current rollout worker.
- **base\_env** (`BaseEnv`) – BaseEnv running the episode. The underlying env object can be gotten by calling `base_env.get_unwrapped()`.
- **episode** (`MultiAgentEpisode`) – Episode object which contains episode state. You can use the `episode.user_data` dict to store temporary data, and `episode.custom_metrics` to store custom metrics for the episode.
- **kwargs** – Forward compatibility placeholder.

```
on_episode_end(worker: ray.rllib.evaluation.rollout_worker.RolloutWorker, base_env: ray.rllib.env.base_env.BaseEnv, policies: Dict[str, ray.rllib.policy.policy.Policy], episode: ray.rllib.evaluation.episode.MultiAgentEpisode, **kwargs)
```

Runs when an episode is done.

#### Parameters

- **worker** (`RolloutWorker`) – Reference to the current rollout worker.
- **base\_env** (`BaseEnv`) – BaseEnv running the episode. The underlying env object can be gotten by calling `base_env.get_unwrapped()`.
- **policies** (`dict`) – Mapping of policy id to policy objects. In single agent mode there will only be a single “default” policy.
- **episode** (`MultiAgentEpisode`) – Episode object which contains episode state. You can use the `episode.user_data` dict to store temporary data, and `episode.custom_metrics` to store custom metrics for the episode.
- **kwargs** – Forward compatibility placeholder.

```
on_postprocess_trajectory(worker: ray.rllib.evaluation.rollout_worker.RolloutWorker,
                           episode: ray.rllib.evaluation.episode.MultiAgentEpisode,
                           agent_id: str, policy_id: str, policies: Dict[str,
                           ray.rllib.policy.policy.Policy], postprocessed_batch:
                           ray.rllib.policy.sample_batch.SampleBatch, original_batches:
                           Dict[str, ray.rllib.policy.sample_batch.SampleBatch], **kwargs)
```

Called immediately after a policy's postprocess\_fn is called.

You can use this callback to do additional postprocessing for a policy, including looking at the trajectory data of other agents in multi-agent settings.

#### Parameters

- **worker** (`RolloutWorker`) – Reference to the current rollout worker.
- **episode** (`MultiAgentEpisode`) – Episode object.
- **agent\_id** (`str`) – Id of the current agent.
- **policy\_id** (`str`) – Id of the current policy for the agent.
- **policies** (`dict`) – Mapping of policy id to policy objects. In single agent mode there will only be a single “default” policy.
- **postprocessed\_batch** (`SampleBatch`) – The postprocessed sample batch for this agent. You can mutate this object to apply your own trajectory postprocessing.
- **original\_batches** (`dict`) – Mapping of agents to their unpostprocessed trajectory data. You should not mutate this object.
- **kwargs** – Forward compatibility placeholder.

```
on_sample_end(worker: ray.rllib.evaluation.rollout_worker.RolloutWorker, samples:
               ray.rllib.policy.sample_batch.SampleBatch, **kwargs)
```

Called at the end RolloutWorker.sample().

#### Parameters

- **worker** (`RolloutWorker`) – Reference to the current rollout worker.
- **samples** (`SampleBatch`) – Batch to be returned. You can mutate this object to modify the samples generated.
- **kwargs** – Forward compatibility placeholder.

```
on_train_result(trainer, result: dict, **kwargs)
```

Called at the end of Trainable.train().

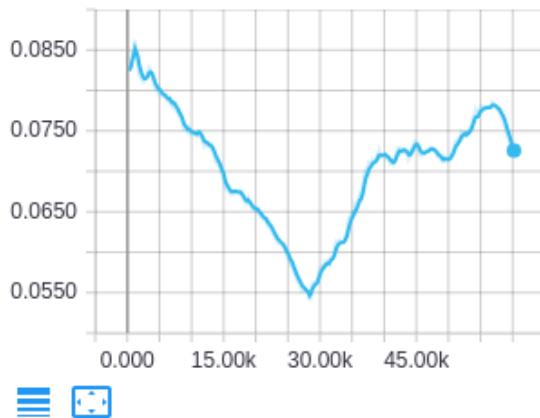
#### Parameters

- **trainer** (`Trainer`) – Current trainer instance.
- **result** (`dict`) – Dict of results returned from trainer.train() call. You can mutate this object to add additional metrics.
- **kwargs** – Forward compatibility placeholder.

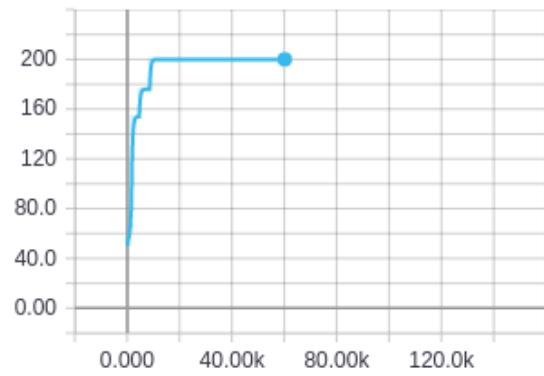
## Visualizing Custom Metrics

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

`ray/tune/custom_metrics/mean_pole_angle`



`ray/tune/episode_reward_max`



## Customizing Exploration Behavior

RLLib offers a unified top-level API to configure and customize an agent's exploration behavior, including the decisions (how and whether) to sample actions from distributions (stochastically or deterministically). The setup can be done via using built-in Exploration classes (see [this package](#)), which are specified (and further configured) inside `Trainer.config["exploration_config"]`. Besides using built-in classes, one can sub-class any of these built-ins, add custom behavior to it, and use that new class in the config instead.

Every policy has-an instantiation of one of the Exploration (sub-)classes. This Exploration object is created from the Trainer's `config["exploration_config"]` dict, which specifies the class to use via the special "type" key, as well as constructor arguments via all other keys, e.g.:

```
# in Trainer.config:
"exploration_config": {
    "type": "StochasticSampling", # <- Special `type` key provides class information
    "[ctor arg]": "[value]", # <- Add any needed constructor args here.
    # etc
}
# ...
```

The following table lists all built-in Exploration sub-classes and the agents that currently used these by default:

An Exploration class implements the `get_exploration_action` method, in which the exact exploratory behavior is defined. It takes the model's output, the action distribution class, the model itself, a timestep (the global env-sampling steps already taken), and an `explore` switch and outputs a tuple of 1) action and 2) log-likelihood:

```
def get_exploration_action(self,
                           distribution_inputs,
                           action_dist_class,
                           model=None,
                           explore=True,
                           timestep=None):
    """Returns a (possibly) exploratory action and its log-likelihood.
```

(continues on next page)

(continued from previous page)

*Given the Model's logits outputs and action distribution, returns an exploratory action.*

**Args:**

```
distribution_inputs (any): The output coming from the model,
    ready for parameterizing a distribution
    (e.g. q-values or PG-logits).
action_dist_class (class): The action distribution class
    to use.
model (ModelV2): The Model object.
explore (bool): True: "Normal" exploration behavior.
    False: Suppress all exploratory behavior and return
    a deterministic action.
timestep (int): The current sampling time step. If None, the
    component should try to use an internal counter, which it
    then increments by 1. If provided, will set the internal
    counter to the given value.
```

**Returns:**

*Tuple:*

- The chosen exploration action or a tf-op to fetch the exploration action from the graph.
- The log-likelihood of the exploration action.

"""

**pass**

On the highest level, the Trainer.compute\_action and Policy.compute\_action(s) methods have a boolean explore switch, which is passed into Exploration.get\_exploration\_action. If None, the value of Trainer.config["explore"] is used. Hence config["explore"] describes the default behavior of the policy and e.g. allows switching off any exploration easily for evaluation purposes (see [Customized Evaluation During Training](#)).

The following are example excerpts from different Trainers' configs (see rllib/agents/trainer.py) to setup different exploration behaviors:

```
# All of the following configs go into Trainer.config.

# 1) Switching *off* exploration by default.
# Behavior: Calling `compute_action(s)` without explicitly setting its `explore`#
# param will result in no exploration.
# However, explicitly calling `compute_action(s)` with `explore=True` will
# still(!) result in exploration (per-call overrides default).
"explore": False,

# 2) Switching *on* exploration by default.
# Behavior: Calling `compute_action(s)` without explicitly setting its
# explore param will result in exploration.
# However, explicitly calling `compute_action(s)` with `explore=False`#
# will result in no(!) exploration (per-call overrides default).
"explore": True,

# 3) Example exploration_config usages:
# a) DQN: see rllib/agents/dqn/dqn.py
"explore": True,
"exploration_config": {
```

(continues on next page)

(continued from previous page)

```

# Exploration sub-class by name or full path to module+class
# (e.g. "ray.rllib.utils.exploration.epsilon_greedy.EpsilonGreedy")
"type": "EpsilonGreedy",
# Parameters for the Exploration class' constructor:
"initial_epsilon": 1.0,
"final_epsilon": 0.02,
"epsilon_timesteps": 10000, # Timesteps over which to anneal epsilon.
},

# b) DQN Soft-Q: In order to switch to Soft-Q exploration, do instead:
"explore": True,
"exploration_config": {
    "type": "SoftQ",
    # Parameters for the Exploration class' constructor:
    "temperature": 1.0,
},
}

# c) PPO: see rllib/agents/ppo/ppo.py
# Behavior: The algo samples stochastically by default from the
# model-parameterized distribution. This is the global Trainer default
# setting defined in trainer.py and used by all PG-type algos.
"explore": True,
"exploration_config": {
    "type": "StochasticSampling",
},
}

```

## Customized Evaluation During Training

RLLib will report online training rewards, however in some cases you may want to compute rewards with different settings (e.g., with exploration turned off, or on a specific set of environment configurations). You can evaluate policies during training by setting the `evaluation_interval` config, and optionally also `evaluation_num_episodes`, `evaluation_config`, `evaluation_num_workers`, and `custom_eval_function` (see `trainer.py` for further documentation).

By default, exploration is left as-is within `evaluation_config`. However, you can switch off any exploration behavior for the evaluation workers via:

```

# Switching off exploration behavior for evaluation workers
# (see rllib/agents/trainer.py)
"evaluation_config": {
    "explore": False
}

```

---

**Note:** Policy gradient algorithms are able to find the optimal policy, even if this is a stochastic one. Setting “`explore=False`” above will result in the evaluation workers not using this stochastic policy.

---

There is an end to end example of how to set up custom online evaluation in `custom_eval.py`. Note that if you only want to eval your policy at the end of training, you can set `evaluation_interval: N`, where N is the number of training iterations before stopping.

Below are some examples of how the custom evaluation metrics are reported nested under the `evaluation` key of normal training results:

```
-----
Sample output for `python custom_eval.py`  
-----  
  
INFO trainer.py:623 -- Evaluating current policy for 10 episodes.  
INFO trainer.py:650 -- Running round 0 of parallel evaluation (2/10 episodes)  
INFO trainer.py:650 -- Running round 1 of parallel evaluation (4/10 episodes)  
INFO trainer.py:650 -- Running round 2 of parallel evaluation (6/10 episodes)  
INFO trainer.py:650 -- Running round 3 of parallel evaluation (8/10 episodes)  
INFO trainer.py:650 -- Running round 4 of parallel evaluation (10/10 episodes)  
  
Result for PG_SimpleCorridor_2c6b27dc:  
...  
evaluation:  
    custom_metrics: {}  
    episode_len_mean: 15.864661654135338  
    episode_reward_max: 1.0  
    episode_reward_mean: 0.49624060150375937  
    episode_reward_min: 0.0  
    episodes_this_iter: 133
```

```
-----
Sample output for `python custom_eval.py --custom-eval`  
-----  
  
INFO trainer.py:631 -- Running custom eval function <function ...>  
Update corridor length to 4  
Update corridor length to 7  
Custom evaluation round 1  
Custom evaluation round 2  
Custom evaluation round 3  
Custom evaluation round 4  
  
Result for PG_SimpleCorridor_0de4e686:  
...  
evaluation:  
    custom_metrics: {}  
    episode_len_mean: 9.15695067264574  
    episode_reward_max: 1.0  
    episode_reward_mean: 0.9596412556053812  
    episode_reward_min: 0.0  
    episodes_this_iter: 223  
    foo: 1
```

## Rewriting Trajectories

Note that in the `on_postprocess_traj` callback you have full access to the trajectory batch (`post_batch`) and other training state. This can be used to rewrite the trajectory, which has a number of uses including:

- Backdating rewards to previous time steps (e.g., based on values in `info`).
- Adding model-based curiosity bonuses to rewards (you can train the model with a `custom` model supervised loss).

To access the policy / model (`policy.model`) in the callbacks, note that `info['pre_batch']` returns a tuple where the first element is a policy and the second one is the batch itself. You can also access all the rollout worker state using the following call:

```
from ray.rllib.evaluation.rollout_worker import get_global_worker

# You can use this from any callback to get a reference to the
# RolloutWorker running in the process, which in turn has references to
# all the policies, etc: see rollout_worker.py for more info.
rollout_worker = get_global_worker()
```

Policy losses are defined over the `post_batch` data, so you can mutate that in the callbacks to change what data the policy loss function sees.

## 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 Trainer API and update the environment between calls to `train()`. This example shows the trainer being run inside a Tune function:

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

def train(config, reporter):
    trainer = PPOTrainer(config=config, env=YourEnv)
    while True:
        result = trainer.train()
        reporter(**result)
        if result["episode_reward_mean"] > 200:
            phase = 2
        elif result["episode_reward_mean"] > 100:
            phase = 1
        else:
            phase = 0
        trainer.workers.foreach_worker(
            lambda ev: ev.foreach_env(
                lambda env: env.set_phase(phase)))

ray.init()
tune.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
```

(continues on next page)

(continued from previous page)

```

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
    trainer = info["trainer"]
    trainer.workers.foreach_worker(
        lambda ev: ev.foreach_env(
            lambda env: env.set_phase(phase)))
ray.init()
tune.run(
    "PPO",
    config={
        "env": YourEnv,
        "callbacks": {
            "on_train_result": on_train_result,
        },
    },
)

```

## 5.17.5 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

```

### Eager Mode

Policies built with `build_tf_policy` (most of the reference algorithms are) can be run in eager mode by setting the "eager": True / "eager\_tracing": True config options or using `rllib train --eager [--trace]`. This will tell RLlib to execute the model forward pass, action distribution, loss, and stats functions in eager mode.

Eager mode makes debugging much easier, since you can now use line-by-line debugging with breakpoints or Python `print()` to inspect intermediate tensor values. However, eager can be slower than graph mode unless tracing is enabled.

## Using PyTorch

Trainers that have an implemented TorchPolicy, will allow you to run `rllib train` using the the command line `--torch` flag. Algorithms that do not have a torch version yet will complain with an error in this case.

## Episode Traces

You can use the [data output API](#) to save episode traces for debugging. For example, the following command will run PPO while saving episode traces to `/tmp/debug`.

```
rllib train --run=PPO --env=CartPole-v0 \
    --config='{"output": "/tmp/debug", "output_compress_columns": []}'

# episode traces will be saved in /tmp/debug, for example
output-2019-02-23_12-02-03_worker-2_0.json
output-2019-02-23_12-02-04_worker-1_0.json
```

## Log Verbosity

You can control the trainer log level via the `"log_level"` flag. Valid values are “DEBUG”, “INFO”, “WARN” (default), and “ERROR”. This can be used to increase or decrease the verbosity of internal logging. You can also use the `-v` and `-vv` flags. For example, the following two commands are about equivalent:

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

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

The default log level is `WARN`. We strongly recommend using at least `INFO` level logging for development.

## 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.

### 5.17.6 External Application 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 it were an independently running service, rather than RLlib hosting the simulations itself. This is possible via RLlib’s external applications interface ([full documentation](#)).

```
class ray.rllib.env.policy_client.PolicyClient(address, inference_mode='local', update_interval=10.0)
    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.env.policy\_server\_input.PolicyServerInput (ioctx, address, port)

REST policy server that acts as an offline data source.

This launches a multi-threaded server that listens on the specified host and port to serve policy requests and forward experiences to RLlib. For high performance experience collection, it implements InputReader.

For an example, run `examples/cartpole_server.py` along with `examples/cartpole_client.py --inference-mode=local|remote`.

## Examples

```
>>> pg = PGTrainer(
...     env="CartPole-v0", config={
...         "input": lambda ioctx:
...             PolicyServerInput(ioctx, addr, port),
...         "num_workers": 0, # Run just 1 server, in the trainer.
...     }
>>> while True:
    pg.train()
```

```
>>> client = PolicyClient("localhost:9900", inference_mode="local")
>>> eps_id = client.start_episode()
>>> action = client.get_action(eps_id, obs)
>>> ...
>>> client.log_returns(eps_id, reward)
>>> ...
>>> client.log_returns(eps_id, reward)
```

### `next()`

Return the next batch of experiences read.

**Returns** SampleBatch or MultiAgentBatch read.

## 5.18 RLLib Environments

RLLib works with several different types of environments, including [OpenAI Gym](#), user-defined, multi-agent, and also batched environments.

---

**Tip:** Not all environments work with all algorithms. Check out the algorithm [feature compatibility matrix](#) for more information.

---

### 5.18.1 Configuring Environments

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 trainer 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>
```

(continues on next page)

(continued from previous page)

```
ray.init()
trainer = ppo.PPOTrainer(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.PPOTrainer(env="my_env")
```

For a full runnable code example using the custom environment API, see [custom\\_env.py](#).

**Warning:** The gym registry is not compatible with Ray. Instead, always use the registration flows documented above to ensure Ray workers can access the environment.

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 trainer. 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))
```

## 5.18.2 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 find the [SimpleCorridor](#) example useful as a reference.

## Performance

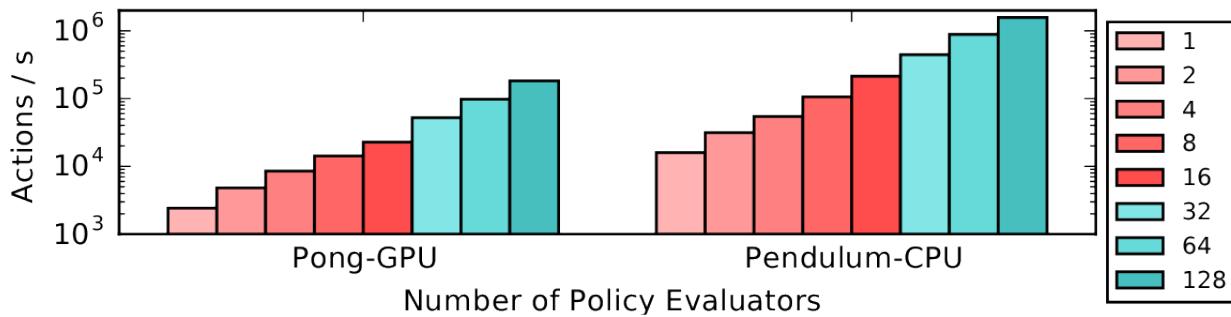
**Tip:** Also check out the [scaling guide](#) for RLLib training.

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.

### 5.18.3 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()`.

Note that auto-vectorization only applies to policy inference by default. This means that policy inference will be batched, but your envs will still be stepped one at a time. If you would like your envs to be stepped in parallel, you can set `"remote_worker_envs": True`. This will create env instances in Ray actors and step them in parallel. These remote processes introduce communication overheads, so this only helps if your env is very expensive to step / reset.

When using remote envs, you can control the batching level for inference with `remote_env_batch_wait_ms`. The default value of 0ms means envs execute asynchronously and inference is only batched opportunistically. Setting

the timeout to a large value will result in fully batched inference and effectively synchronous environment stepping. The optimal value depends on your environment step / reset time, and model inference speed.

### 5.18.4 Multi-Agent and Hierarchical

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": {
        "policies": {
            # the first tuple value is None -> uses default policy
            "car1": (None, car_obs_space, car_act_space, {"gamma": 0.85}),
            "car2": (None, car_obs_space, car_act_space, {"gamma": 0.99}),
            "traffic_light": (None, 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"
},
})
```

(continues on next page)

(continued from previous page)

```
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: 4.99999873689376e-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: 4.99999873689376e-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.

## Rock Paper Scissors Example

The [rock\\_paper\\_scissors\\_multiagent.py](#) example demonstrates several types of policies competing against each other: heuristic policies of repeating the same move, beating the last opponent move, and learned LSTM and feedforward policies.



Fig. 2: TensorBoard output of running the rock-paper-scissors example, where a learned policy faces off between a random selection of the same-move and beat-last-move heuristics. Here the performance of heuristic policies vs the learned policy is compared with LSTM enabled (blue) and a plain feed-forward policy (red). While the feedforward policy can easily beat the same-move heuristic by simply avoiding the last move taken, it takes a LSTM policy to distinguish between and consistently beat both policies.

## Variable-Sharing Between Policies

**Note:** With [ModelV2](#), you can put layers in global variables and straightforwardly share those layer objects between models instead of using variable scopes.

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

Here are two ways to implement a centralized critic compatible with the multi-agent API:

### Strategy 1: Sharing experiences in the trajectory preprocessor:

The most general way of implementing a centralized critic involves modifying the `postprocess_trajectory` method of a custom policy, 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 post-processed trajectory. Here's an example:

```
def postprocess_trajectory(policy, 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(
```

(continues on next page)

(continued from previous page)

```
    self.critic_network, feed_dict={"obs": global_obs_batch})
    return sample_batch
```

To update the critic, you'll also have to modify the loss of the policy. For an end-to-end runnable example, see [examples/centralized\\_critic.py](#).

### Strategy 2: Sharing observations through the environment:

Alternatively, the env itself can be modified to share observations between agents. In this strategy, each observation includes all global state, and policies use a custom model to ignore state they aren't supposed to "see" when computing actions. The advantage of this approach is that it's very simple and you don't have to change the algorithm at all – just use an env wrapper and custom model. However, it is a bit less principled in that you have to change the agent observation spaces and the environment. You can find a runnable example of this strategy at [examples/centralized\\_critic\\_2.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, {
        ...     "group1": ["agent1", "agent2", "agent3"],
        ...     "group2": ["agent4", "agent5"],
        ... })
        """
```

(continues on next page)

(continued from previous page)

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

For environments with multiple groups, or mixtures of agent groups and individual agents, you can use grouping in conjunction with the policy mapping API described in prior sections.

## 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": {
    "policies": {
        "top_level": (custom_policy or None, ...),
        "mid_level": (custom_policy or None, ...),
        "low_level": (custom_policy or None, ...)
    },
    "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`.

## 5.18.5 External Agents and Applications

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.

### Logging off-policy actions

`ExternalEnv` 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).

#### See also:

Offline Datasets provide higher-level interfaces for working with off-policy experience datasets.

### External Application Clients

For applications that are running entirely outside the Ray cluster (i.e., cannot be packaged into a Python environment of any form), RLlib provides the `PolicyServerInput` application connector, which can be connected to over the network using `PolicyClient` instances.

You can configure any Trainer to launch a policy server with the following config:

```
trainer_config = {
    # An environment class is still required, but it doesn't need to be runnable.
    # You only need to define its action and observation space attributes.
    "env": YOUR_ENV_STUB,

    # Use the policy server to generate experiences.
    "input": (
        lambda ioctx: PolicyServerInput(ioctx, SERVER_ADDRESS, SERVER_PORT)
    ),
    # Use the existing trainer process to run the server.
    "num_workers": 0,
    # Disable OPE, since the rollouts are coming from online clients.
    "input_evaluation": [],
}
```

Clients can then connect in either *local* or *remote* inference mode. In local inference mode, copies of the policy are downloaded from the server and cached on the client for a configurable period of time. This allows actions to be computed by the client without requiring a network round trip each time. In remote inference mode, each computed action requires a network call to the server.

Example:

```
client = PolicyClient("http://localhost:9900", inference_mode="local")
episode_id = client.start_episode()
...
action = client.get_action(episode_id, cur_obs)
...
client.end_episode(episode_id, last_obs)
```

To understand the difference between standard envs, external envs, and connecting with a `PolicyClient`, refer to the following figure:

Try it yourself by launching a `cartpole_server.py`, and connecting to it with any number of clients (`cartpole_client.py`):

```
# Start the server by running:  
>>> python rllib/examples/serving/cartpole_server.py --run=PPO  
--  
-- Starting policy server at localhost:9900  
--  
  
# To connect from a client with inference_mode="remote".  
>>> python rllib/examples/serving/cartpole_client.py --inference-mode=remote  
Total reward: 10.0  
Total reward: 58.0  
...  
Total reward: 200.0  
...  
  
# To connect from a client with inference_mode="local" (faster).  
>>> python rllib/examples/serving/cartpole_client.py --inference-mode=local  
Querying server for new policy weights.  
Generating new batch of experiences.  
Total reward: 13.0  
Total reward: 11.0  
...  
Sending batch of 1000 steps back to server.  
Querying server for new policy weights.  
...  
Total reward: 200.0  
...
```

For the best performance, when possible we recommend using `inference_mode="local"` when possible.

### 5.18.6 Advanced Integrations

For more complex / high-performance environment integrations, you can instead extend the low-level `BaseEnv` class. This low-level API models multiple agents executing asynchronously in multiple environments. A call to `BaseEnv:poll()` returns observations from ready agents keyed by their environment and agent ids, and actions for those agents are sent back via `BaseEnv:send_actions()`. `BaseEnv` is used to implement all the other env types in RLLib, so it offers a superset of their functionality. For example, `BaseEnv` is used to implement dynamic batching of observations for inference over multiple simulator actors.

## 5.19 RLLib Models, Preprocessors, and Action Distributions

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.

### 5.19.1 Default Behaviours

#### Built-in Models and Preprocessors

RLLib picks default models based on a simple heuristic: a [vision network](#) for observations that have shape of length larger than 2 (for example, (84 x 84 x 3)), 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,
    # Whether to skip the final linear layer used to resize the hidden layer
    # outputs to size `num_outputs`. If True, then the last hidden layer
    # should already match num_outputs.
    "no_final_linear": False,
    # Whether layers should be shared for the value function.
    "vf_share_layers": True,

    # == 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,
    # When using modelv1 models with a modelv2 algorithm, you may have to
}
```

(continues on next page)

(continued from previous page)

```

# define the state shape here (e.g., [256, 256]).
"state_shape": None,

# == Atari ==
# Whether to enable framestack for Atari envs
"framestack": True,
# Final resized frame dimension
"dim": 84,
# (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 model to use
"custom_model": None,
# Name of a custom action distribution to use.
"custom_action_dist": None,

# Extra options to pass to the custom classes
"custom_options": {},
# Custom preprocessors are deprecated. Please use a wrapper class around
# your environment instead to preprocess observations.
"custom_preprocessor": None,
}

```

## 5.19.2 TensorFlow Models

**Note:** TFModelV2 replaces the previous `rllib.models.Model` class, which did not support Keras-style reuse of variables. The `rllib.models.Model` class is deprecated and should not be used.

Custom TF models should subclass `TFModelV2` to implement the `__init__()` and `forward()` methods. Forward takes in a dict of tensor inputs (the observation `obs`, `prev_action`, and `prev_reward`, `is_training`), optional RNN state, and returns the model output of size `num_outputs` and the new state. You can also override extra methods of the model such as `value_function` to implement a custom value branch. Additional supervised / self-supervised losses can be added via the `custom_loss` method:

```
class ray.rllib.models.tf.tf_modelv2.TFModelV2(obs_space, action_space, num_outputs,
                                              model_config, name)
```

TF version of ModelV2.

Note that this class by itself is not a valid model unless you implement `forward()` in a subclass.

```
__init__(obs_space, action_space, num_outputs, model_config, name)
Initialize a TFModelV2.
```

Here is an example implementation for a subclass `MyModelClass` (`TFModelV2`):

```
def __init__(self, *args, **kwargs):
    super(MyModelClass, self).__init__(*args, **kwargs)
    input_layer = tf.keras.layers.Input(...)
    hidden_layer = tf.keras.layers.Dense(...)(input_layer)
    output_layer = tf.keras.layers.Dense(...)(hidden_layer)
    value_layer = tf.keras.layers.Dense(...)(hidden_layer)
```

(continues on next page)

(continued from previous page)

```
self.base_model = tf.keras.Model(
    input_layer, [output_layer, value_layer])
self.register_variables(self.base_model.variables)
```

**forward**(*input\_dict*, *state*, *seq\_lens*)

Call the model with the given input tensors and state.

Any complex observations (dicts, tuples, etc.) will be unpacked by `__call__` before being passed to `forward()`. To access the flattened observation tensor, refer to `input_dict["obs_flat"]`.

This method can be called any number of times. In eager execution, each call to `forward()` will eagerly evaluate the model. In symbolic execution, each call to `forward` creates a computation graph that operates over the variables of this model (i.e., shares weights).

Custom models should override this instead of `__call__`.

**Parameters**

- **input\_dict** (*dict*) – dictionary of input tensors, including “obs”, “obs\_flat”, “prev\_action”, “prev\_reward”, “is\_training”
- **state** (*list*) – list of state tensors with sizes matching those returned by `get_initial_state` + the batch dimension
- **seq\_lens** (*Tensor*) – 1d tensor holding input sequence lengths

**Returns**

**The model output tensor of size** [BATCH, num\_outputs]

**Return type** (outputs, state)

**Examples**

```
>>> def forward(self, input_dict, state, seq_lens):
...     model_out, self._value_out = self.base_model(
...         input_dict["obs"])
...     return model_out, state
```

**value\_function**()

Returns the value function output for the most recent forward pass.

Note that a `forward` call has to be performed first, before this methods can return anything and thus that calling this method does not cause an extra forward pass through the network.

**Returns** value estimate tensor of shape [BATCH].

**custom\_loss**(*policy\_loss*, *loss\_inputs*)

Override to customize the loss function used to optimize this model.

This can be used to incorporate self-supervised losses (by defining a loss over existing input and output tensors of this model), and supervised losses (by defining losses over a variable-sharing copy of this model’s layers).

You can find an runnable example in `examples/custom_loss.py`.

**Parameters**

- **policy\_loss** (*Union[List[Tensor], Tensor]*) – List of or single policy loss(es) from the policy.

- **loss\_inputs** (*dict*) – map of input placeholders for rollout data.

**Returns**

**List of or scalar tensor for the** customized loss(es) for this model.

**Return type** Union[List[Tensor],Tensor]

**metrics()**

Override to return custom metrics from your model.

**The stats will be reported as part of the learner stats, i.e.,**

**info:**

**learner:**

**model:** key1: metric1 key2: metric2

**Returns** Dict of string keys to scalar tensors.

**update\_ops()**

Return the list of update ops for this model.

For example, this should include any BatchNorm update ops.

**register\_variables(*variables*)**

Register the given list of variables with this model.

**variables(*as\_dict=False*)**

Returns the list (or a dict) of variables for this model.

**Parameters** **as\_dict** (*bool*) – Whether variables should be returned as dict-values (using descriptive keys).

**Returns**

**The list (or dict if *as\_dict* is True)** of all variables of this ModelV2.

**Return type** Union[List[any],Dict[str,any]]

**trainable\_variables(*as\_dict=False*)**

Returns the list of trainable variables for this model.

**Parameters** **as\_dict** (*bool*) – Whether variables should be returned as dict-values (using descriptive keys).

**Returns**

**The list (or dict if *as\_dict* is True)** of all trainable (tf)/requires\_grad (torch) variables of this ModelV2.

**Return type** Union[List[any],Dict[str,any]]

Once implemented, 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
from ray.rllib.models.tf.tf_modelv2 import TFModelV2

class MyModelClass(TFModelV2):
    def __init__(self, obs_space, action_space, num_outputs, model_config, name): ...
    def forward(self, input_dict, state, seq_lens): ...
```

(continues on next page)

(continued from previous page)

```

def value_function(self): ...

ModelCatalog.register_custom_model("my_model", MyModelClass)

ray.init()
trainer = ppo.PPOTrainer(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 [keras model example](#). You can also reference the [unit tests](#) for Tuple and Dict spaces, which show how to access nested observation fields.

## 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. For a RNN model it is preferred to subclass `RecurrentTFModelV2` to implement `__init__()`, `get_initial_state()`, and `forward_rnn()`. You can check out the `custom_keras_rnn_model.py` model as an example to implement your own model:

```

class ray.rllib.models.tf.recurrent_tf_modelv2.RecurrentTFModelV2(obs_space,
                                                               ac-
                                                               tion_space,
                                                               num_outputs,
                                                               model_config,
                                                               name)

```

Helper class to simplify implementing RNN models with TFModelV2.

Instead of implementing `forward()`, you can implement `forward_rnn()` which takes batches with the time dimension added already.

Here is an example implementation for a subclass `MyRNNClass` (`RecurrentTFModelV2`):

```

def __init__(self, *args, **kwargs):
    super(MyModelClass, self). __init__(*args, **kwargs)
    cell_size = 256

    # Define input layers
    input_layer = tf.keras.layers.Input(
        shape=(None, obs_space.shape[0]))
    state_in_h = tf.keras.layers.Input(shape=(256, ))
    state_in_c = tf.keras.layers.Input(shape=(256, ))
    seq_in = tf.keras.layers.Input(shape=(), dtype=tf.int32)

    # Send to LSTM cell
    lstm_out, state_h, state_c = tf.keras.layers.LSTM(
        cell_size, return_sequences=True, return_state=True,
        name="lstm")(
            inputs=input_layer,
            mask=tf.sequence_mask(seq_in),
            initial_state=[state_in_h, state_in_c])
    output_layer = tf.keras.layers.Dense(...)(lstm_out)

```

(continues on next page)

(continued from previous page)

```
# Create the RNN model
self.rnn_model = tf.keras.Model(
    inputs=[input_layer, seq_in, state_in_h, state_in_c],
    outputs=[output_layer, state_h, state_c])
self.register_variables(self.rnn_model.variables)
self.rnn_model.summary()
```

**\_\_init\_\_(obs\_space, action\_space, num\_outputs, model\_config, name)**  
 Initialize a TFModelV2.

Here is an example implementation for a subclass MyModelClass (TFModelV2):

```
def __init__(self, *args, **kwargs):
    super(MyModelClass, self).__init__(*args, **kwargs)
    input_layer = tf.keras.layers.Input(...)
    hidden_layer = tf.keras.layers.Dense(...)(input_layer)
    output_layer = tf.keras.layers.Dense(...)(hidden_layer)
    value_layer = tf.keras.layers.Dense(...)(hidden_layer)
    self.base_model = tf.keras.Model(
        input_layer, [output_layer, value_layer])
    self.register_variables(self.base_model.variables)
```

**forward\_rnn(inputs, state, seq\_lens)**  
 Call the model with the given input tensors and state.

#### Parameters

- **inputs** (*dict*) – observation tensor with shape [B, T, obs\_size].
- **state** (*list*) – list of state tensors, each with shape [B, T, size].
- **seq\_lens** (*Tensor*) – 1d tensor holding input sequence lengths.

#### Returns

The model output tensor of shape [B, T, num\_outputs] and the list of new state tensors each with shape [B, size].

#### Return type

Sample implementation for the MyRNNClass example:

```
def forward_rnn(self, inputs, state, seq_lens):
    model_out, h, c = self.rnn_model([inputs, seq_lens] + state)
    return model_out, [h, c]
```

**get\_initial\_state()**  
 Get the initial recurrent state values for the model.

#### Returns

list of np.array objects, if any

Sample implementation for the MyRNNClass example:

```
def get_initial_state(self):
    return [
        np.zeros(self.cell_size, np.float32),
        np.zeros(self.cell_size, np.float32),
    ]
```

## 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.py` and `multi_gpu_impl.py` for the exact handling of these updates).

In case RLlib does not properly detect the update ops for your custom model, you can override the `update_ops()` method to return the list of ops to run for updates.

### 5.19.3 PyTorch Models

Similarly, you can create and register custom PyTorch models for use with PyTorch-based algorithms (e.g., A2C, PG, QMIX). See these examples of [fully connected](#), [convolutional](#), and [recurrent torch models](#).

```
class ray.rllib.models.torch.torch_modelv2.TorchModelV2(obs_space,
                                                       action_space,
                                                       num_outputs,
                                                       model_config, name)
```

Torch version of ModelV2.

Note that this class by itself is not a valid model unless you inherit from `nn.Module` and implement `forward()` in a subclass.

```
__init__(obs_space, action_space, num_outputs, model_config, name)
        Initialize a TorchModelV2.
```

Here is an example implementation for a subclass `MyModelClass` (`TorchModelV2`, `nn.Module`):

```
def __init__(self, *args, **kwargs):
    TorchModelV2.__init__(self, *args, **kwargs)
    nn.Module.__init__(self)
    self._hidden_layers = nn.Sequential(...)
    self._logits = ...
    self._value_branch = ...
```

`forward(input_dict, state, seq_lens)`

Call the model with the given input tensors and state.

Any complex observations (dicts, tuples, etc.) will be unpacked by `__call__` before being passed to `forward()`. To access the flattened observation tensor, refer to `input_dict["obs_flat"]`.

This method can be called any number of times. In eager execution, each call to `forward()` will eagerly evaluate the model. In symbolic execution, each call to `forward` creates a computation graph that operates over the variables of this model (i.e., shares weights).

Custom models should override this instead of `__call__`.

#### Parameters

- `input_dict` (`dict`) – dictionary of input tensors, including “obs”, “obs\_flat”, “prev\_action”, “prev\_reward”, “is\_training”
- `state` (`list`) – list of state tensors with sizes matching those returned by `get_initial_state` + the batch dimension
- `seq_lens` (`Tensor`) – 1d tensor holding input sequence lengths

#### Returns

The model output tensor of size [BATCH, num\_outputs]

Return type (outputs, state)

## Examples

```
>>> def forward(self, input_dict, state, seq_lens):
>>>     features = self._hidden_layers(input_dict["obs"])
>>>     self._value_out = self._value_branch(features)
>>>     return self._logits(features), state
```

### `value_function()`

Returns the value function output for the most recent forward pass.

Note that a *forward* call has to be performed first, before this methods can return anything and thus that calling this method does not cause an extra forward pass through the network.

**Returns** value estimate tensor of shape [BATCH].

### `custom_loss(policy_loss, loss_inputs)`

Override to customize the loss function used to optimize this model.

This can be used to incorporate self-supervised losses (by defining a loss over existing input and output tensors of this model), and supervised losses (by defining losses over a variable-sharing copy of this model's layers).

You can find an runnable example in examples/custom\_loss.py.

### Parameters

- **policy\_loss** (`Union[List[Tensor], Tensor]`) – List of or single policy loss(es) from the policy.
- **loss\_inputs** (`dict`) – map of input placeholders for rollout data.

### Returns

**List of or scalar tensor for the** customized loss(es) for this model.

**Return type** `Union[List[Tensor], Tensor]`

### `metrics()`

Override to return custom metrics from your model.

**The stats will be reported as part of the learner stats, i.e.,**

**info:**

**learner:**

**model:** key1: metric1 key2: metric2

**Returns** Dict of string keys to scalar tensors.

### `get_initial_state()`

Get the initial recurrent state values for the model.

### Returns

**List of np.array objects containing the initial** hidden state of an RNN, if applicable.

**Return type** `List[np.ndarray]`

## Examples

```
>>> def get_initial_state(self):
>>>     return [
>>>         np.zeros(self.cell_size, np.float32),
>>>         np.zeros(self.cell_size, np.float32),
>>>     ]
```

Once implemented, the model can then be registered and used in place of a built-in model:

```
import torch.nn as nn

import ray
from ray.rllib.agents import a3c
from ray.rllib.models import ModelCatalog
from ray.rllib.models.torch.torch_modelv2 import TorchModelV2

class CustomTorchModel(nn.Module, TorchModelV2):
    def __init__(self, obs_space, action_space, num_outputs, model_config, name): ...
    def forward(self, input_dict, state, seq_lens): ...
    def value_function(self): ...

ModelCatalog.register_custom_model("my_model", CustomTorchModel)

ray.init()
trainer = a3c.A2CTrainer(env="CartPole-v0", config={
    "use_pytorch": True,
    "model": {
        "custom_model": "my_model",
        "custom_options": {}, # extra options to pass to your model
    },
})
```

### 5.19.4 Custom Preprocessors

**Warning:** Custom preprocessors are deprecated, since they sometimes conflict with the built-in preprocessors for handling complex observation spaces. Please use [wrapper classes](#) around your environment instead of preprocessors.

Custom preprocessors should subclass the RLLib [preprocessor class](#) and be registered in the model catalog:

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

class MyPreprocessorClass(Preprocessor):
    def __init__(self, obs_space, options):
        return new_shape # can vary depending on inputs

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

(continues on next page)

(continued from previous page)

```
ModelCatalog.register_custom_preprocessor("my_prep", MyPreprocessorClass)

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

## 5.19.5 Custom Action Distributions

Similar to custom models and preprocessors, you can also specify a custom action distribution class as follows. The action dist class is passed a reference to the model, which you can use to access `model.model_config` or other attributes of the model. This is commonly used to implement *autoregressive action outputs*.

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

class MyActionDist(ActionDistribution):
    @staticmethod
    def required_model_output_shape(action_space, model_config):
        return 7 # controls model output feature vector size

    def __init__(self, inputs, model):
        super(MyActionDist, self).__init__(inputs, model)
        assert model.num_outputs == 7

    def sample(self): ...
    def logp(self, actions): ...
    def entropy(self): ...

ModelCatalog.register_custom_action_dist("my_dist", MyActionDist)

ray.init()
trainer = ppo.PPOTrainer(env="CartPole-v0", config={
    "model": {
        "custom_action_dist": "my_dist",
    },
})
```

## 5.19.6 Supervised Model Losses

You can mix supervised losses into any RLLib algorithm through custom models. For example, you can add an imitation learning loss on expert experiences, or a self-supervised autoencoder loss within the model. These losses can be defined over either policy evaluation inputs, or data read from `offline storage`.

**TensorFlow:** To add a supervised loss to a custom TF model, you need to override the `custom_loss()` method. This method takes in the existing policy loss for the algorithm, which you can add your own supervised loss to before returning. For debugging, you can also return a dictionary of scalar tensors in the `metrics()` method. Here is a runnable example of adding an imitation loss to CartPole training that is defined over a `offline dataset`.

**PyTorch:** There is no explicit API for adding losses to custom torch models. However, you can modify the loss in the policy definition directly. Like for TF models, offline datasets can be incorporated by creating an input reader and calling `reader.next()` in the loss forward pass.

### 5.19.7 Self-Supervised Model Losses

You can also use the `custom_loss()` API to add in self-supervised losses such as VAE reconstruction loss and L2-regularization.

### 5.19.8 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`. 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  $[0, \text{MAX_AVAIL_ACTIONS}]$  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 ParametricActionsModel(TFModelV2):
    def __init__(self,
                 obs_space,
                 action_space,
                 num_outputs,
                 model_config,
                 name,
                 true_obs_shape=(4,),
                 action_embed_size=2):
        super(ParametricActionsModel, self).__init__(
            obs_space, action_space, num_outputs, model_config, name)
        self.action_embed_model = FullyConnectedNetwork(...)

    def forward(self, input_dict, state, seq_lens):
        # Extract the available actions tensor from the observation.
        avail_actions = input_dict["obs"]["avail_actions"]
        action_mask = input_dict["obs"]["action_mask"]

        # Compute the predicted action embedding
        action_embed, _ = self.action_embed_model({
            "obs": input_dict["obs"]["cart"]})
```

(continues on next page)

(continued from previous page)

```

    }

    # 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(action_embed, 1)

    # Batch dot product => 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)
    return action_logits + inf_mask, state

```

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. Not all algorithms support parametric actions; see the [feature compatibility matrix](#).

## 5.19.9 Autoregressive Action Distributions

In an action space with multiple components (e.g., `Tuple(a1, a2)`), you might want `a2` to be conditioned on the sampled value of `a1`, i.e.,  $a2_{\text{sampled}} \sim P(a2 \mid a1_{\text{sampled}}, \text{obs})$ . Normally, `a1` and `a2` would be sampled independently, reducing the expressivity of the policy.

To do this, you need both a custom model that implements the autoregressive pattern, and a custom action distribution class that leverages that model. The `autoregressive_action_dist.py` example shows how this can be implemented for a simple binary action space. For a more complex space, a more efficient architecture such as a `MADE` is recommended. Note that sampling a  $N$ -part action requires  $N$  forward passes through the model, however computing the log probability of an action can be done in one pass:

```

class BinaryAutoregressiveOutput(ActionDistribution):
    """Action distribution  $P(a1, a2) = P(a1) * P(a2 \mid a1)$ """

    @staticmethod
    def required_model_output_shape(self, model_config):
        return 16 # controls model output feature vector size

    def sample(self):
        # first, sample a1
        a1_dist = self._a1_distribution()
        a1 = a1_dist.sample()

        # sample a2 conditioned on a1
        a2_dist = self._a2_distribution(a1)
        a2 = a2_dist.sample()

        # return the action tuple
        return TupleActions([a1, a2])

    def logp(self, actions):
        a1, a2 = actions[:, 0], actions[:, 1]
        a1_vec = tf.expand_dims(tf.cast(a1, tf.float32), 1)
        a1_logits, a2_logits = self.model.action_model([self.inputs, a1_vec])

```

(continues on next page)

(continued from previous page)

```

    return (Categorical(a1_logits, None).logp(a1) + Categorical(
        a2_logits, None).logp(a2))

    def _a1_distribution(self):
        BATCH = tf.shape(self.inputs)[0]
        a1_logits, _ = self.model.action_model(
            [self.inputs, tf.zeros((BATCH, 1))])
        a1_dist = Categorical(a1_logits, None)
        return a1_dist

    def _a2_distribution(self, a1):
        a1_vec = tf.expand_dims(tf.cast(a1, tf.float32), 1)
        _, a2_logits = self.model.action_model([self.inputs, a1_vec])
        a2_dist = Categorical(a2_logits, None)
        return a2_dist

class AutoregressiveActionsModel(TFModelV2):
    """Implements the `action_model` branch required above."""

    def __init__(self, obs_space, action_space, num_outputs, model_config,
                 name):
        super(AutoregressiveActionsModel, self).__init__(
            obs_space, action_space, num_outputs, model_config, name)
        if action_space != Tuple([Discrete(2), Discrete(2)]):
            raise ValueError(
                "This model only supports the [2, 2] action space")

        # Inputs
        obs_input = tf.keras.layers.Input(
            shape=obs_space.shape, name="obs_input")
        a1_input = tf.keras.layers.Input(shape=(1, ), name="a1_input")
        ctx_input = tf.keras.layers.Input(
            shape=(num_outputs, ), name="ctx_input")

        # Output of the model (normally 'logits', but for an autoregressive
        # dist this is more like a context/feature layer encoding the obs)
        context = tf.keras.layers.Dense(
            num_outputs,
            name="hidden",
            activation=tf.nn.tanh,
            kernel_initializer=normc_initializer(1.0))(obs_input)

        # P(a1 | obs)
        a1_logits = tf.keras.layers.Dense(
            2,
            name="a1_logits",
            activation=None,
            kernel_initializer=normc_initializer(0.01))(ctx_input)

        # P(a2 | a1)
        # --note: typically you'd want to implement P(a2 | a1, obs) as follows:
        # a2_context = tf.keras.layers.concatenate(axis=1)(
        #     [ctx_input, a1_input])
        a2_context = a1_input
        a2_hidden = tf.keras.layers.Dense(
            16,
            name="a2_hidden",

```

(continues on next page)

(continued from previous page)

```

activation=tf.nn.tanh,
kernel_initializer=normc_initializer(1.0))(a2_context)
a2_logits = tf.keras.layers.Dense(
    2,
    name="a2_logits",
    activation=None,
    kernel_initializer=normc_initializer(0.01))(a2_hidden)

# Base layers
self.base_model = tf.keras.Model(obs_input, context)
self.register_variables(self.base_model.variables)
self.base_model.summary()

# Autoregressive action sampler
self.action_model = tf.keras.Model([ctx_input, a1_input],
                                   [a1_logits, a2_logits])
self.action_model.summary()
self.register_variables(self.action_model.variables)

```

**Note:** Not all algorithms support autoregressive action distributions; see the feature compatibility matrix.

## 5.20 RLlib Algorithms

**Tip:** Check out the [environments](#) page to learn more about different environment types.

### 5.20.1 Feature Compatibility Matrix

Algorithm	Frameworks	Discrete Actions	Continuous Actions	Multi-Agent	Model Support
<i>A2C, A3C</i>	tf + torch	Yes +parametric	Yes	Yes	+RNN, +autoreg
<i>ARS</i>	tf + torch	Yes	Yes	No	
<i>ES</i>	tf + torch	Yes	Yes	No	
<i>DDPG, TD3</i>	tf + torch	No	Yes	Yes	
<i>APEX-DDPG</i>	tf	No	Yes	Yes	
<i>DQN, Rainbow</i>	tf + torch	Yes +parametric	No	Yes	
<i>APEX-DQN</i>	tf + torch	Yes +parametric	No	Yes	
<i>IMPALA</i>	tf	Yes +parametric	Yes	Yes	+RNN, +autoreg
<i>MARWIL</i>	tf + torch	Yes +parametric	Yes	Yes	+RNN
<i>PG</i>	tf + torch	Yes +parametric	Yes	Yes	+RNN, +autoreg
<i>PPO, APPO</i>	tf + torch	Yes +parametric	Yes	Yes	+RNN, +autoreg
<i>QMIX</i>	torch	Yes	No	Yes	+RNN
<i>SAC</i>	tf + torch	Yes	Yes	Yes	
<i>AlphaZero</i>	torch	Yes +parametric	No	No	
<i>LinUCB, LinTS</i>	torch	Yes +parametric	No	Yes	
<i>MADDPG</i>	tf	No	Yes	Yes	

## 5.20.2 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.

Fig. 3: Ape-X architecture

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

**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": merge_dicts(
        DQN_CONFIG["optimizer"], {
            "max_weight_sync_delay": 400,
            "num_replay_buffer_shards": 4,
            "debug": False
        }),
    "n_step": 3,
    "num_gpus": 1,
    "num_workers": 32,
    "buffer_size": 2000000,
    "learning_starts": 50000,
    "train_batch_size": 512,
    "rollout_fragment_length": 50,
    "target_network_update_freq": 500000,
    "timesteps_per_iteration": 25000,
    "exploration_config": {"type": "PerWorkerEpsilonGreedy"},
    "worker_side_prioritization": True,
    "min_iter_time_s": 30,
}
```

(continues on next page)

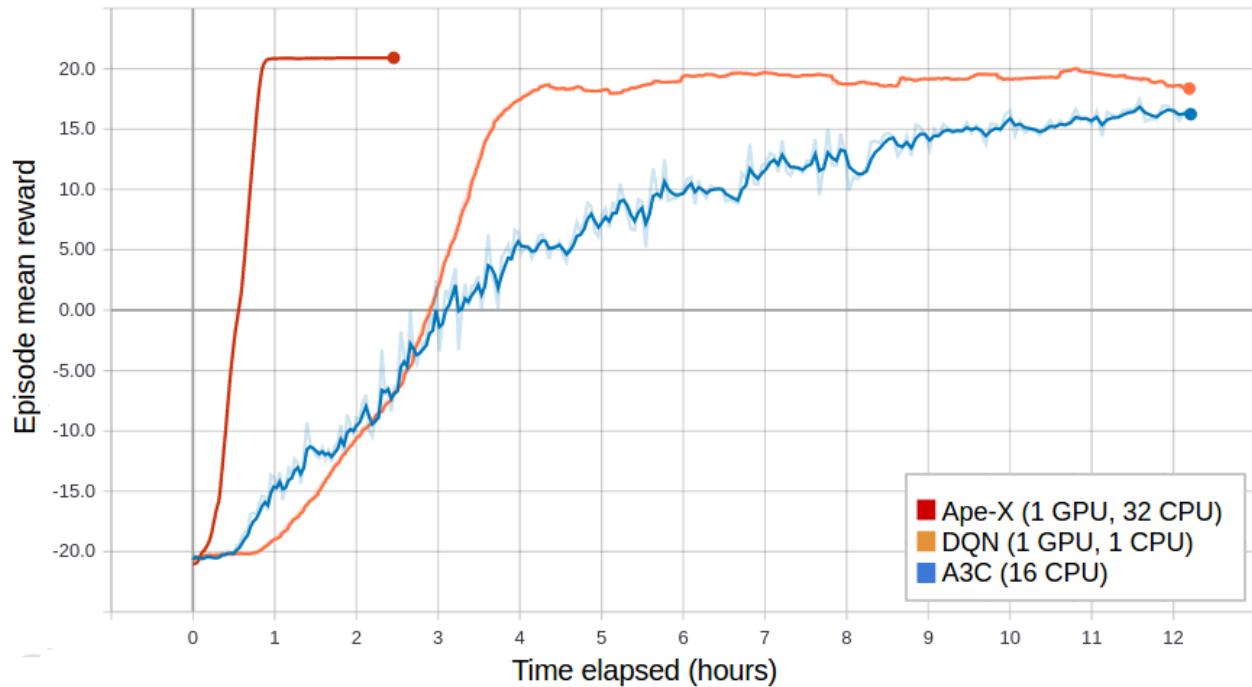


Fig. 4: Ape-X using 32 workers in RLlib vs vanilla DQN (orange) and A3C (blue) on PongNoFrameskip-v4.

(continued from previous page)

```

} ,
)
```

## 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.

Fig. 5: IMPALA architecture

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

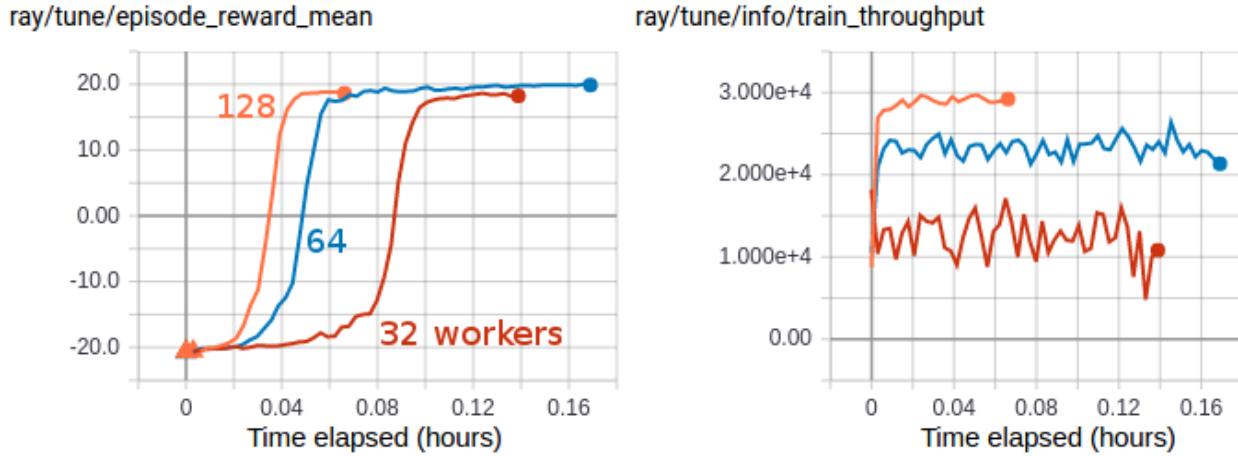


Fig. 6: Multi-GPU IMPALA scales up to solve PongNoFrameskip-v4 in ~3 minutes using a pair of V100 GPUs and 128 CPU workers. The maximum training throughput reached is ~30k transitions per second (~120k environment frames per second).

#### IMPALA-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # V-trace params (see vtrace_tf.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 `rollout_fragment_length * num_envs_per_worker`.
    # 2. If enabled, the replay buffer stores and produces batches of size
    #    `rollout_fragment_length * 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`.
    #
    "rollout_fragment_length": 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
```

(continues on next page)

(continued from previous page)

```

# 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 * rollout_fragment_length).
"replay_buffer_num_slots": 0,
# max queue size for train batches feeding into the learner
"learner_queue_size": 16,
# wait for train batches to be available in minibatch buffer queue
# this many seconds. This may need to be increased e.g. when training
# with a slow environment
"learner_queue_timeout": 300,
# 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,
# use intermediate actors for multi-level aggregation. This can make sense
# if ingesting >2GB/s of samples, or if the data requires decompression.
"num_aggregation_workers": 0,

# Learning params.
"grad_clip": 40.0,
# either "adam" or "rmsprop"
"opt_type": "adam",
"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,
"entropy_coeff_schedule": None,

# use fake (infinite speed) sampler for testing
"_fake_sampler": False,
})

```

## 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.

**Tip:** APPO is not always more efficient; it is often better to use *standard PPO* or *IMPALA*.

Fig. 7: APPo architecture (same as 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 ==
    # Should use a critic as a baseline (otherwise don't use value baseline;
    # required for using GAE).
    "use_critic": True,
    # 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,

    # == PPO KL Loss options ==
    "use_kl_loss": False,
    "kl_coeff": 1.0,
    "kl_target": 0.01,

    # == IMPALA optimizer params (see documentation in impala.py) ==
    "rollout_fragment_length": 50,
    "train_batch_size": 500,
    "min_iter_time_s": 10,
    "num_workers": 2,
    "num_gpus": 0,
    "num_data_loader_buffers": 1,
    "minibatch_buffer_size": 1,
    "num_sgd_iter": 1,
    "replay_proportion": 0.0,
    "replay_buffer_num_slots": 100,
    "learner_queue_size": 16,
    "learner_queue_timeout": 300,
    "max_sample_requests_in_flight_per_worker": 2,
    "broadcast_interval": 1,
    "grad_clip": 40.0,
    "opt_type": "adam",
    "lr": 0.0005,
    "lr_schedule": None,
    "decay": 0.99,
    "momentum": 0.0,
    "epsilon": 0.1,
    "vf_loss_coeff": 0.5,
    "entropy_coeff": 0.01,
    "entropy_coeff_schedule": None,
})
```

## Decentralized Distributed Proximal Policy Optimization (DD-PPO)

 [paper] [implementation] Unlike APPO or PPO, with DD-PPO policy improvement is no longer done centralized in the trainer process. Instead, gradients are computed remotely on each rollout worker and all-reduced at each mini-batch using `torch distributed`. This allows each worker's GPU to be used both for sampling and for training.

---

**Tip:** DD-PPO is best for envs that require GPUs to function, or if you need to scale out SGD to multiple nodes. If you don't meet these requirements, *standard PPO* will be more efficient.

---

Fig. 8: DD-PPO architecture (both sampling and learning are done on worker GPUs)

Tuned examples: `CartPole-v0`, `BreakoutNoFrameskip-v4`

**DDPPO-specific configs** (see also common configs):

```
DEFAULT_CONFIG = with_base_config(ppo.DEFAULT_CONFIG, {
    # During the sampling phase, each rollout worker will collect a batch
    # `rollout_fragment_length * num_envs_per_worker` steps in size.
    "rollout_fragment_length": 100,
    # Vectorize the env (should enable by default since each worker has a GPU).
    "num_envs_per_worker": 5,
    # During the SGD phase, workers iterate over minibatches of this size.
    # The effective minibatch size will be `sgd_minibatch_size * num_workers`.
    "sgd_minibatch_size": 50,
    # Number of SGD epochs per optimization round.
    "num_sgd_iter": 10,

    # *** WARNING: configs below are DDPPO overrides over PPO; you
    # shouldn't need to adjust them. ***
    "use_pytorch": True, # DDPPO requires PyTorch distributed.
    "num_gpus": 0, # Learning is no longer done on the driver process, so
                   # giving GPUs to the driver does not make sense!
    "num_gpus_per_worker": 1, # Each rollout worker gets a GPU.
    "truncate_episodes": True, # Require evenly sized batches. Otherwise,
                             # collective allreduce could fail.
    "train_batch_size": -1, # This is auto set based on sample batch size.
})
```

### 5.20.3 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.

A2C also supports microbatching (i.e., gradient accumulation), which can be enabled by setting the `microbatch_size` config. Microbatching allows for training with a `train_batch_size` much larger than GPU memory. See also the `microbatch` optimizer implementation.

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

Fig. 9: A2C architecture

---

**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({
    # Should use a critic as a baseline (otherwise don't use value baseline;
    # required for using GAE).
    "use_critic": True,
    # If true, use the Generalized Advantage Estimator (GAE)
    # with a value function, see https://arxiv.org/pdf/1506.02438.pdf.
    "use_gae": True,
    # Size of rollout batch
    "rollout_fragment_length": 10,
    # 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
    # rollout_fragment_length by up to 5x due to async buffering of batches.
    "sample_async": True,
    # Use the execution plan API instead of policy optimizers.
    "use_exec_api": 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 as TD3.

Fig. 10: DDPG architecture (same as DQN)

Tuned examples: Pendulum-v0, MountainCarContinuous-v0, HalfCheetah-v2, TD3 Pendulum-v0, TD3 InvertedPendulum-v2, TD3 Mujoco suite (Ant-v2, HalfCheetah-v2, Hopper-v2, Walker2d-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
    # In addition to settings below, you can use "exploration_noise_type" and
    # "exploration_gauss_act_noise" to get IID Gaussian exploration noise
    # instead of OU exploration noise.
    # twin Q-net
    "twin_q": False,
    # delayed policy update
    "policy_delay": 1,
    # target policy smoothing
    # (this also replaces OU exploration noise with IID Gaussian exploration
    # noise, for now)
    "smooth_target_policy": False,
    # gaussian stddev of target action noise for smoothing
    "target_noise": 0.2,
    # target noise limit (bound)
    "target_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 ===
    # Apply a state preprocessor with spec given by the "model" config option
    # (like other RL algorithms). This is mostly useful if you have a weird
    # observation shape, like an image. Disabled by default.
    "use_state_preprocessor": False,
    # Postprocess the policy network model output with these hidden layers. If
    # use_state_preprocessor is False, then these will be the *only* hidden
    # layers in the network.
    "actor_hiddens": [400, 300],
    # Hidden layers activation of the postprocessing stage of the policy
    # network
    "actor_hidden_activation": "relu",
    # Postprocess the critic network model output with these hidden layers;
    # again, if use_state_preprocessor is True, then the state will be
    # preprocessed by the model specified with the "model" config option first.
}
```

(continues on next page)

(continued from previous page)

```

"critic_hiddens": [400, 300],
# Hidden layers activation of the postprocessing state of the critic.
"critic_hidden_activation": "relu",
# N-step Q learning
"n_step": 1,

# === Exploration ===
"exploration_config": {
    # DDPG uses OrnsteinUhlenbeck (stateful) noise to be added to NN-output
    # actions (after a possible pure random phase of n timesteps).
    "type": "OrnsteinUhlenbeckNoise",
    # For how many timesteps should we return completely random actions,
    # before we start adding (scaled) noise?
    "random_timesteps": 1000,
    # The OU-base scaling factor to always apply to action-added noise.
    "ou_base_scale": 0.1,
    # The OU theta param.
    "ou_theta": 0.15,
    # The OU sigma param.
    "ou_sigma": 0.2,
    # The initial noise scaling factor.
    "initial_scale": 1.0,
    # The final noise scaling factor.
    "final_scale": 1.0,
    # Timesteps over which to anneal scale (from initial to final values).
    "scale_timesteps": 10000,
},
# Number of env steps to optimize for before returning
"timesteps_per_iteration": 1000,
# Extra configuration that disables exploration.
"evaluation_config": {
    "explore": False
},
# === 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,
# Time steps over which the beta parameter is annealed.
"prioritized_replay_beta_annealing_timesteps": 20000,
# 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": False,

# === Optimization ===
# Learning rate for the critic (Q-function) optimizer.
"critic_lr": 1e-3,
# Learning rate for the actor (policy) optimizer.
"actor_lr": 1e-3,

```

(continues on next page)

(continued from previous page)

```

# 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,
# 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.
"rollout_fragment_length": 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,

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

# Deprecated keys.
"parameter_noise": DEPRECATED_VALUE,
})

```

## 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.

Fig. 11: DQN architecture

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.

---

---

**Hint:** For a complete `rainbow` setup, make the following changes to the default DQN config: `"n_step": [between 1 and 10], "noisy": True, "num_atoms": [more than 1], "v_min": -10.0, "v_max": 10.0` (set `v_min` and `v_max` according to your expected range of returns).

---

**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,
    # Dense-layer setup for each the advantage branch and the value branch
    # in a dueling architecture.
    "hiddens": [256],
    # Whether to use double dqn
    "double_q": True,
    # N-step Q learning
    "n_step": 1,

    # === Exploration Settings (Experimental) ===
    "exploration_config": {
        # The Exploration class to use.
        "type": "EpsilonGreedy",
        # Config for the Exploration class' constructor:
        "initial_epsilon": 1.0,
        "final_epsilon": 0.02,
        "epsilon_timesteps": 10000, # Timesteps over which to anneal epsilon.

        # For soft_q, use:
        # "exploration_config" = {
        #     "type": "SoftQ"
        #     "temperature": [float, e.g. 1.0]
        # }
    },
    # Switch to greedy actions in evaluation workers.
    "evaluation_config": {
        "explore": False,
    },
})
```

(continues on next page)

(continued from previous page)

```

# Minimum env steps to optimize for per train call. This value does
# not affect learning, only the length of iterations.
"timesteps_per_iteration": 1000,
# 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,
# Final value of beta (by default, we use constant beta=0.4).
"final_prioritized_replay_beta": 0.4,
# Time steps over which the beta parameter is annealed.
"prioritized_replay_beta_annealing_timesteps": 20000,
# 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
"lr": 5e-4,
# Learning rate schedule
"lr_schedule": None,
# Adam epsilon hyper parameter
"adam_epsilon": 1e-8,
# If not None, clip gradients during optimization at this value
"grad_clip": 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.
"rollout_fragment_length": 4,
# Size of a batch 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,
# Whether to compute priorities on workers.
"worker_side_prioritization": False,
# Prevent iterations from going lower than this time span
"min_iter_time_s": 1,

# DEPRECATED VALUES (set to -1 to indicate they have not been overwritten
# by user's config). If we don't set them here, we will get an error
# from the config-key checker.

```

(continues on next page)

(continued from previous page)

```

"schedule_max_timesteps": DEPRECATED_VALUE,
"exploration_final_eps": DEPRECATED_VALUE,
"exploration_fraction": DEPRECATED_VALUE,
"beta_annealing_fraction": DEPRECATED_VALUE,
"per_worker_exploration": DEPRECATED_VALUE,
"softmax_temp": DEPRECATED_VALUE,
"soft_q": DEPRECATED_VALUE,
"parameter_noise": DEPRECATED_VALUE,
"grad_norm_clipping": DEPRECATED_VALUE,

# Use the execution plan API instead of policy optimizers.
"use_exec_api": True,
})

```

## Policy Gradients



[paper] [implementation] We include a vanilla policy gradients implementation as an example algorithm.

Fig. 12: Policy gradients architecture (same as A2C)

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,
    # Use the execution plan API instead of policy optimizers.
    "use_exec_api": True,
})

```

## 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. PPO scales out using multiple workers for experience collection, and also to multiple GPUs for SGD.

---

**Tip:** If you need to scale out with GPUs on multiple nodes, consider using [decentralized PPO](#).

---

Fig. 13: PPO architecture

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

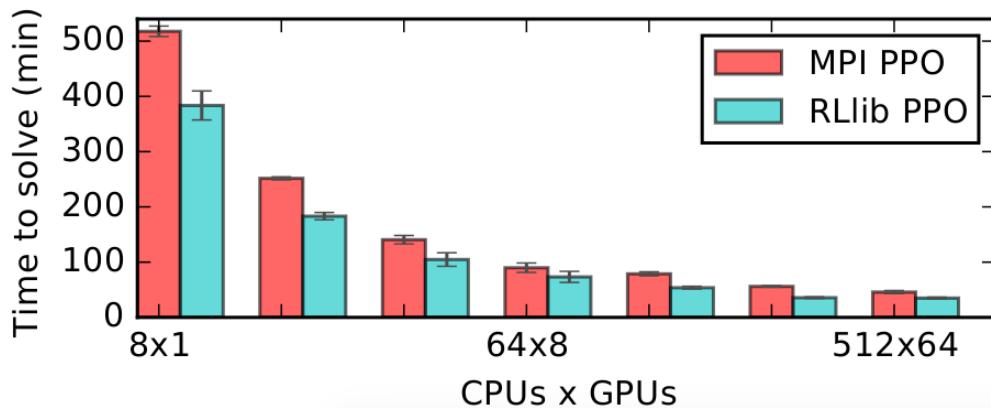


Fig. 14: RLLib’s multi-GPU PPO scales to multiple GPUs and hundreds of CPUs on solving the Humanoid-v1 task. Here we compare against a reference MPI-based implementation.

**PPO-specific configs** (see also [common configs](#)):

```
DEFAULT_CONFIG = with_common_config({
    # Should use a critic as a baseline (otherwise don't use value baseline;
    # required for using GAE).
    "use_critic": True,
    # If true, use the Generalized Advantage Estimator (GAE)
    # with a value function, see https://arxiv.org/pdf/1506.02438.pdf.
    "use_gae": True,
    # The GAE(lambda) parameter.
    "lambda": 1.0,
    # Initial coefficient for KL divergence.
    "kl_coeff": 0.2,
    # Size of batches collected from each worker.
    "rollout_fragment_length": 200,
    # Number of timesteps collected for each SGD round. This defines the size
    # of each SGD epoch.
    "train_batch_size": 4000,
    # Total SGD batch size across all devices for SGD. This defines the
    # minibatch size within each epoch.
    "sgd_minibatch_size": 128,
    # Whether to shuffle sequences in the batch when training (recommended).
    "shuffle_sequences": True,
    # Number of SGD iterations in each outer loop (i.e., number of epochs to
    # execute per train batch).
    "num_sgd_iter": 30,
```

(continues on next page)

(continued from previous page)

```

# Stepsize of SGD.
"lr": 5e-5,
# Learning rate schedule.
"lr_schedule": None,
# Share layers for value function. If you set this to True, it's important
# to tune vf_loss_coeff.
"vf_share_layers": False,
# Coefficient of the value function loss. IMPORTANT: you must tune this if
# you set vf_share_layers: True.
"vf_loss_coeff": 1.0,
# Coefficient of the entropy regularizer.
"entropy_coeff": 0.0,
# Decay schedule for the entropy regularizer.
"entropy_coeff_schedule": None,
# 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": "NoFilter",
# Uses the sync samples optimizer instead of the multi-gpu one. This is
# usually slower, but you might want to try it if you run into issues with
# the default optimizer.
"simple_optimizer": False,
# Whether to fake GPUs (using CPUs).
# Set this to True for debugging on non-GPU machines (set `num_gpus` > 0).
"_fake_gpus": False,
# Use PyTorch as framework?
"use_pytorch": False
})

```

## Soft Actor Critic (SAC)



Fig. 15: SAC architecture (same as DQN)

RLLib's soft-actor critic implementation is ported from the [official SAC repo](#) to better integrate with RLLib APIs. Note that SAC has two fields to configure for custom models: `policy_model` and `Q_model`.

Tuned examples: Pendulum-v0, HalfCheetah-v3

**MuJoCo results @3M steps:** [more details](#)

MuJoCo env	RLLib SAC	Haarnoja et al SAC
HalfCheetah	13000	~15000

**SAC-specific configs** (see also [common configs](#)):

```

DEFAULT_CONFIG = with_common_config({
    # === Model ===
    "twin_q": True,
    "use_state_preprocessor": False,
    # RLlib model options for the Q function(s).
    "Q_model": {
        "fcnet_activation": "relu",
        "fcnet_hiddens": [256, 256],
        "hidden_activation": DEPRECATED_VALUE,
        "hidden_layer_sizes": DEPRECATED_VALUE,
    },
    # RLlib model options for the policy function.
    "policy_model": {
        "fcnet_activation": "relu",
        "fcnet_hiddens": [256, 256],
        "hidden_activation": DEPRECATED_VALUE,
        "hidden_layer_sizes": DEPRECATED_VALUE,
    },
    # Unsquash actions to the upper and lower bounds of env's action space.
    # Ignored for discrete action spaces.
    "normalize_actions": True,

    # === Learning ===
    # Disable setting done=True at end of episode. This should be set to True
    # for infinite-horizon MDPs (e.g., many continuous control problems).
    "no_done_at_end": False,
    # Update the target by \tau * policy + (1-\tau) * target_policy.
    "tau": 5e-3,
    # Initial value to use for the entropy weight alpha.
    "initial_alpha": 1.0,
    # Target entropy lower bound. If "auto", will be set to -|A| (e.g. -2.0 for
    # Discrete(2), -3.0 for Box(shape=(3,))). Note that if done_at_end is set to True,
    # then target_entropy is ignored.
    # This is the inverse of reward scale, and will be optimized automatically.
    "target_entropy": "auto",
    # N-step target updates.
    "n_step": 1,

    # Number of env steps to optimize for before returning.
    "timesteps_per_iteration": 100,

    # === 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": int(1e6),
    # If True prioritized replay buffer will be used.
    "prioritized_replay": False,
    "prioritized_replay_alpha": 0.6,
    "prioritized_replay_beta": 0.4,
    "prioritized_replay_eps": 1e-6,
    "prioritized_replay_beta_annealing_timesteps": 20000,
    "final_prioritized_replay_beta": 0.4,

    "compress_observations": False,

    # === Optimization ===
    "optimization": {
}

```

(continues on next page)

(continued from previous page)

```

    "actor_learning_rate": 3e-4,
    "critic_learning_rate": 3e-4,
    "entropy_learning_rate": 3e-4,
},
# If not None, clip gradients during optimization at this value.
"grad_clip": 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.
"rollout_fragment_length": 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,
# Update the target network every `target_network_update_freq` steps.
"target_network_update_freq": 0,

# === Parallelism ===
# Whether to use a GPU for local optimization.
"num_gpus": 0,
# 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,
# Whether to allocate GPUs for workers (if > 0).
"num_gpus_per_worker": 0,
# Whether to allocate CPUs for workers (if > 0).
"num_cpus_per_worker": 1,
# Whether to compute priorities on workers.
"worker_side_prioritization": False,
# Prevent iterations from going lower than this time span.
"min_iter_time_s": 1,

# Whether the loss should be calculated deterministically (w/o the
# stochastic action sampling step). True only useful for cont. actions and
# for debugging!
"_deterministic_loss": False,
# Use a Beta-distribution instead of a SquashedGaussian for bounded,
# continuous action spaces (not recommended, for debugging only).
"_use_beta_distribution": False,

# DEPRECATED VALUES (set to -1 to indicate they have not been overwritten
# by user's config). If we don't set them here, we will get an error
# from the config-key checker.
"grad_norm_clipping": DEPRECATED_VALUE,
})

```

## 5.20.4 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({
    "action_noise_std": 0.0,
    "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:**

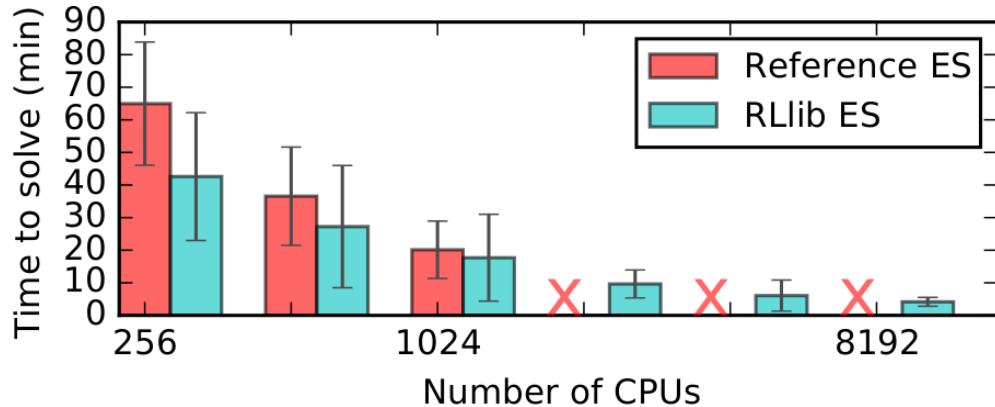


Fig. 16: RLLib's ES implementation scales further and is faster than a reference Redis implementation on solving the Humanoid-v1 task.

**ES-specific configs** (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    "action_noise_std": 0.01,
    "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](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.

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",

    # === 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
})
```

(continues on next page)

(continued from previous page)

```
"exploration_fraction": 0.1,
# Initial value of random action probability.
"exploration_initial_eps": 1.0,
# 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 RMSProp 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.
"rollout_fragment_length": 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,
# 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,

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

## Multi-Agent Deep Deterministic Policy Gradient (contrib/MADDPG)



[paper] [implementation] MADDPG is a specialized multi-agent algorithm. Code here is adapted from <https://github.com/openai/maddpg> to integrate with RLlib multi-agent APIs. Please check [justinkerry/maddpg-rllib](https://github.com/justinkerry/maddpg-rllib) for examples and more information.

**MADDPG-specific configs** (see also [common configs](#)):

Tuned examples: Multi-Agent Particle Environment, Two-step game

```
DEFAULT_CONFIG = with_common_config({
    # === Settings for each individual policy ===
    # ID of the agent controlled by this policy
    "agent_id": None,
    # Use a local critic for this policy.
    "use_local_critic": False,

    # === Evaluation ===
    # Evaluation interval
    "evaluation_interval": None,
    # Number of episodes to run per evaluation period.
    "evaluation_num_episodes": 10,

    # === Model ===
    # Apply a state preprocessor with spec given by the "model" config option
    # (like other RL algorithms). This is mostly useful if you have a weird
    # observation shape, like an image. Disabled by default.
    "use_state_preprocessor": False,
    # Postprocess the policy network model output with these hidden layers. If
    # use_state_preprocessor is False, then these will be the *only* hidden
    # layers in the network.
    "actor_hiddens": [64, 64],
    # Hidden layers activation of the postprocessing stage of the policy
    # network
    "actor_hidden_activation": "relu",
    # Postprocess the critic network model output with these hidden layers;
    # again, if use_state_preprocessor is True, then the state will be
    # preprocessed by the model specified with the "model" config option first.
    "critic_hiddens": [64, 64],
    # Hidden layers activation of the postprocessing stage of the critic.
    "critic_hidden_activation": "relu",
    # N-step Q learning
    "n_step": 1,
    # Algorithm for good policies
    "good_policy": "maddpg",
    # Algorithm for adversary policies
    "adv_policy": "maddpg",

    # === 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": int(1e6),
    # Observation compression. Note that compression makes simulation slow in
    # MPE.
    "compress_observations": False,

    # === Optimization ===
}
```

(continues on next page)

(continued from previous page)

```

# Learning rate for the critic (Q-function) optimizer.
"critic_lr": 1e-2,
# Learning rate for the actor (policy) optimizer.
"actor_lr": 1e-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": 0.01,
# Weights for feature regularization for the actor
"actor_feature_reg": 0.001,
# If not None, clip gradients during optimization at this value
"grad_norm_clipping": 0.5,
# How many steps of the model to sample before learning starts.
"learning_starts": 1024 * 25,
# Update the replay buffer with this many samples at once. Note that this
# setting applies per-worker if num_workers > 1.
"rollout_fragment_length": 100,
# 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": 1024,
# Number of env steps to optimize for before returning
"timesteps_per_iteration": 0,

# === 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": 1,
# Prevent iterations from going lower than this time span
"min_iter_time_s": 0,
})
}

```

## 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({
    # You should override this to point to an offline dataset (see agent.py).
    "input": "sampler",
    # Use importance sampling estimators for reward
    "input_evaluation": ["is", "wis"],

    # 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
})

```

(continues on next page)

(continued from previous page)

```

"postprocess_inputs": True,
# Whether to rollout "complete_episodes" or "truncate_episodes"
"batch_mode": "complete_episodes",
# 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,
# Use PyTorch as framework?
"use_pytorch": False
})

```

## Single-Player Alpha Zero (contrib/AlphaZero)

 [paper] [implementation] AlphaZero is an RL agent originally designed for two-player games. This version adapts it to handle single player games. The code can be used with the SyncSamplesOptimizer as well as with a modified version of the SyncReplayOptimizer, and it scales to any number of workers. It also implements the ranked rewards (R2) strategy to enable self-play even in the one-player setting. The code is mainly purposed to be used for combinatorial optimization.

Tuned examples: [CartPole-v0](#)

**AlphaZero-specific configs** (see also [common configs](#)):

```

DEFAULT_CONFIG = with_common_config({
    # Size of batches collected from each worker
    "rollout_fragment_length": 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,
    # Whether to shuffle sequences in the batch when training (recommended)
    "shuffle_sequences": True,
    # Number of SGD iterations in each outer loop
    "num_sgd_iter": 30,
    # IN case a buffer optimizer is used
    "learning_starts": 1000,
    "buffer_size": 10000,
    # Stepsize of SGD
    "lr": 5e-5,
    # Learning rate schedule
    "lr_schedule": None,
    # Share layers for value function. If you set this to True, it's important
    # to tune vf_loss_coeff.
    "vf_share_layers": False,
    # Whether to rollout "complete_episodes" or "truncate_episodes"
    "batch_mode": "complete_episodes",
    # Which observation filter to apply to the observation
    "observation_filter": "NoFilter",
})

```

(continues on next page)

(continued from previous page)

```

# Uses the sync samples optimizer instead of the multi-gpu one. This does
# not support minibatches.
"simple_optimizer": True,

# === MCTS ===
"mcts_config": {
    "puct_coefficient": 1.0,
    "num_simulations": 30,
    "temperature": 1.5,
    "dirichlet_epsilon": 0.25,
    "dirichlet_noise": 0.03,
    "argmax_tree_policy": False,
    "add_dirichlet_noise": True,
},

# === Ranked Rewards ===
# implement the ranked reward (r2) algorithm
# from: https://arxiv.org/pdf/1807.01672.pdf
"ranked_rewards": {
    "enable": True,
    "percentile": 75,
    "buffer_max_length": 1000,
    # add rewards obtained from random policy to
    # "warm start" the buffer
    "initialize_buffer": True,
    "num_init_rewards": 100,
},

# === Evaluation ===
# Extra configuration that disables exploration.
"evaluation_config": {
    "mcts_config": {
        "argmax_tree_policy": True,
        "add_dirichlet_noise": False,
    },
},
}

# === Callbacks ===
"callbacks": AlphaZeroDefaultCallbacks,
"use_pytorch": True,
})

```

## 5.20.5 Contextual Bandits (contrib/bandits)

The Multi-armed bandit (MAB) problem provides a simplified RL setting that involves learning to act under one situation only, i.e. the state is fixed. Contextual bandit is extension of the MAB problem, where at each round the agent has access not only to a set of bandit arms/actions but also to a context (state) associated with this iteration. The context changes with each iteration, but, is not affected by the action that the agent takes. The objective of the agent is to maximize the cumulative rewards, by collecting enough information about how the context and the rewards of the arms are related to each other. The agent does this by balancing the trade-off between exploration and exploitation.

Contextual bandit algorithms typically consist of an action-value model (Q model) and an exploration strategy (e-

greedy, UCB, Thompson Sampling etc.) RLLib supports the following online contextual bandit algorithms, named after the exploration strategies that they employ:

### Linear Upper Confidence Bound (contrib/LinUCB)

 [paper] [implementation] LinUCB assumes a linear dependency between the expected reward of an action and its context. It estimates the Q value of each action using ridge regression. It constructs a confidence region around the weights of the linear regression model and uses this confidence ellipsoid to estimate the uncertainty of action values.

Tuned examples: `SimpleContextualBandit`, `ParametricItemRecoEnv`.

**LinUCB-specific configs** (see also common configs):

```
UCB_CONFIG = with_common_config({
    # No remote workers by default.
    "num_workers": 0,
    "use_pytorch": True,

    # Do online learning one step at a time.
    "rollout_fragment_length": 1,
    "train_batch_size": 1,

    # Bandits cant afford to do one timestep per iteration as it is extremely
    # slow because of metrics collection overhead. This setting means that the
    # agent will be trained for 100 times in one iteration of Rllib
    "timesteps_per_iteration": 100,

    "exploration_config": {
        "type": "ray.rllib.contrib.bandits.exploration.UCB"
    }
})
```

### Linear Thompson Sampling (contrib/LinTS)

 [paper] [implementation] Like LinUCB, LinTS also assumes a linear dependency between the expected reward of an action and its context and uses online ridge regression to estimate the Q values of actions given the context. It assumes a Gaussian prior on the weights and a Gaussian likelihood function. For deciding which action to take, the agent samples weights for each arm, using the posterior distributions, and plays the arm that produces the highest reward.

Tuned examples: `SimpleContextualBandit`, `WheelBandit`.

**LinTS-specific configs** (see also common configs):

```
TS_CONFIG = with_common_config({
    # No remote workers by default.
    "num_workers": 0,
    "use_pytorch": True,

    # Do online learning one step at a time.
    "rollout_fragment_length": 1,
    "train_batch_size": 1,

    # Bandits cant afford to do one timestep per iteration as it is extremely
})
```

(continues on next page)

(continued from previous page)

```

# slow because of metrics collection overhead. This setting means that the
# agent will be trained for 100 times in one iteration of Rllib
"timesteps_per_iteration": 100,

"exploration_config": {
    "type": "ray.rllib.contrib.bandits.exploration.ThompsonSampling"
}
})

```

## 5.21 RLlib Offline Datasets

### 5.21.1 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.,  $(s, a, r, s', \dots)$  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 environments, 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
```

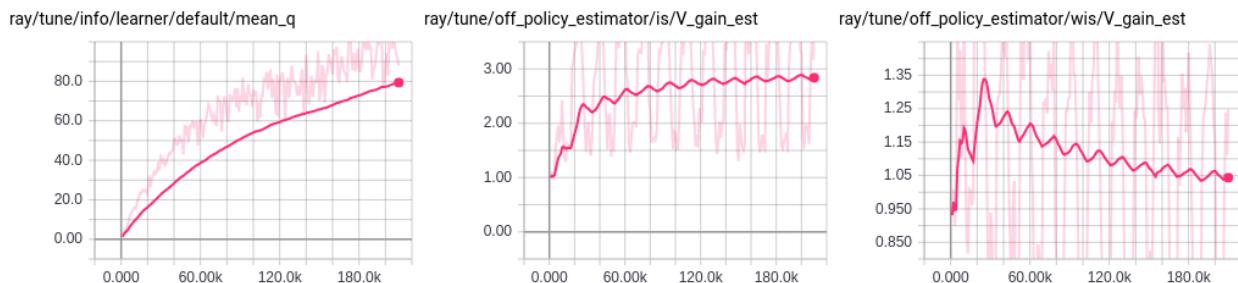
Then, we can tell DQN to train using these previously generated experiences with `"input": "/tmp/cartpole-out"`. We disable exploration since it has no effect on the input:

```
$ rllib train \
--run=DQN \
--env=CartPole-v0 \
--config='{
    "input": "/tmp/cartpole-out",
    "input_evaluation": [],
    "explore": false}'
```

**Off-policy estimation:** 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. Alternatively, [off-policy estimation](#) can be used, which requires both the source and target action probabilities to be available (i.e., the `action_prob` batch key). For DQN, this means enabling soft Q learning so that actions are sampled from a probability distribution:

```
$ rllib train \
--run=DQN \
--env=CartPole-v0 \
--config='{
    "input": "/tmp/cartpole-out",
    "input_evaluation": ["is", "wis"],
    "exploration_config": {
        "type": "SoftQ",
        "temperature": 1.0,
    }
}'
```

This example plot shows the Q-value metric in addition to importance sampling (IS) and weighted importance sampling (WIS) gain estimates (>1.0 means there is an estimated improvement over the original policy):



**Estimator Python API:** For greater control over the evaluation process, you can create off-policy estimators in your Python code and call `estimator.estimate(episode_batch)` to perform counterfactual estimation as needed. The estimators take in a policy object and gamma value for the environment:

```
trainer = DQNTrainer(...)
... # train policy offline

from ray.rllib.offline.json_reader import JsonReader
from ray.rllib.offline.wis_estimator import WeightedImportanceSamplingEstimator

estimator = WeightedImportanceSamplingEstimator(trainer.get_policy(), gamma=0.99)
reader = JsonReader("/path/to/data")
for _ in range(1000):
    batch = reader.next()
    for episode in batch.split_by_episode():
        print(estimator.estimate(episode))
```

**Simulation-based estimation:** 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. Note that in all cases you still need to specify an environment object to define the action and observation spaces. However, you don't need to implement functions like `reset()` and `step()`.

### 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
import os

import ray.utils

from ray.rllib.models.preprocessors import get_preprocessor
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(
        os.path.join(ray.utils.get_user_temp_dir(), "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")

    # RLLib uses preprocessors to implement transforms such as one-hot encoding
    # and flattening of tuple and dict observations. For CartPole a no-op
    # preprocessor is used, but this may be relevant for more complex envs.
    prep = get_preprocessor(env.observation_space)(env.observation_space)
    print("The preprocessor is", prep)

    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()
            new_obs, rew, done, info = env.step(action)
            batch_builder.add_values(
                t=t,
                eps_id=eps_id,
                agent_index=0,
                obs=prep.transform(obs),
                actions=action,
                action_prob=1.0, # put the true action probability here
                rewards=rew,
                prev_actions=prev_action,
                prev_rewards=prev_reward,
                dones=done,
                infos=info,
                new_obs=prep.transform(new_obs))
```

(continues on next page)

(continued from previous page)

```

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,
    },
    "explore": false}'

```

## 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.

## 5.21.2 Input Pipeline for Supervised Losses

You can also define supervised model losses over offline data. This requires defining a [custom model loss](#). We provide a convenience function, `InputReader.tf_input_ops()`, that can be used to convert any input reader to a TF input pipeline. For example:

```
def custom_loss(self, policy_loss):
    input_reader = JsonReader("/tmp/cartpole-out")
    # print(input_reader.next())  # if you want to access imperatively

    input_ops = input_reader.tf_input_ops()
    print(input_ops["obs"])  # -> output Tensor shape=[None, 4]
    print(input_ops["actions"])  # -> output Tensor shape=[None]

    supervised_loss = some_function_of(input_ops)
    return policy_loss + supervised_loss
```

See `custom_loss.py` for a runnable example of using these TF input ops in a custom loss.

## 5.21.3 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 has an effect when
# reading offline experiences. Available options:
# - "wis": the weighted step-wise importance sampling estimator.
# - "is": the step-wise importance sampling estimator.
# - "simulation": run the environment in the background, but use
#   this data for evaluation only and not for learning.
"input_evaluation": ["is", "wis"],
# Whether to run postprocess_trajectory() on the trajectory fragments from
# offline inputs. Note that postprocessing will be done using the *current*
# policy, not the *behavior* policy, which is typically undesirable for
# on-policy algorithms.
"postprocess_inputs": False,
# If positive, input batches will be shuffled via a sliding window buffer
# of this number of batches. Use this if the input data is not in random
# enough order. Input is delayed until the shuffle buffer is filled.
"shuffle_buffer_size": 0,
```

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.

**tf\_input\_ops**(queue\_size=1)

Returns TensorFlow queue ops for reading inputs from this reader.

The main use of these ops is for integration into custom model losses. For example, you can use tf\_input\_ops() to read from files of external experiences to add an imitation learning loss to your model.

This method creates a queue runner thread that will call next() on this reader repeatedly to feed the TensorFlow queue.

**Parameters** `queue_size`(*int*) – Max elements to allow in the TF queue.

### Example

```
>>> class MyModel(rllib.model.Model):
...     def custom_loss(self, policy_loss, loss_inputs):
...         reader = JsonReader(...)
...         input_ops = reader.tf_input_ops()
...         logits, _ = self._build_layers_v2(
...             {"obs": input_ops["obs"]},
...             self.num_outputs, self.options)
...         il_loss = imitation_loss(logits, input_ops["action"])
...         return policy_loss + il_loss
```

You can find a runnable version of this in examples/custom\_loss.py.

**Returns** dict of Tensors, one for each column of the read SampleBatch.

## 5.21.4 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.

## 5.22 RLlib Concepts and Custom Algorithms

This page describes the internal concepts used to implement algorithms in RLlib. You might find this useful if modifying or adding new algorithms to RLlib.

### 5.22.1 Policies

Policy 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 [policy definition](#).

Most interaction with deep learning frameworks is isolated to the [Policy interface](#), allowing RLlib to support multiple frameworks. To simplify the definition of policies, RLlib includes [Tensorflow](#) and [PyTorch-specific](#) templates. You can also write your own from scratch. Here is an example:

```
class CustomPolicy(Policy):
    """Example of a custom policy written from scratch.

    You might find it more convenient to use the `build_tf_policy` and
    `build_torch_policy` helpers instead for a real policy, which are
    described in the next sections.
    """

    def __init__(self, observation_space, action_space, config):
        Policy.__init__(self, observation_space, action_space, config)
        # example parameter
        self.w = 1.0

    def compute_actions(self,
                        obs_batch,
                        state_batches,
                        prev_action_batch=None,
                        prev_reward_batch=None,
                        info_batch=None,
                        episodes=None,
                        **kwargs):
        # return action batch, RNN states, extra values to include in batch
        return [self.action_space.sample() for _ in obs_batch], [], {}

    def learn_on_batch(self, samples):
        # implement your learning code here
        return {} # return stats

    def get_weights(self):
        return {"w": self.w}

    def set_weights(self, weights):
        self.w = weights["w"]
```

The above basic policy, when run, will produce batches of observations with the basic `obs`, `new_obs`, `actions`, `rewards`, `dones`, and `infos` columns. There are two more mechanisms to pass along and emit extra information:

**Policy recurrent state:** Suppose you want to compute actions based on the current timestep of the episode. While it is possible to have the environment provide this as part of the observation, we can instead compute and store it as part of the Policy recurrent state:

```

def get_initial_state(self):
    """Returns initial RNN state for the current policy."""
    return [0] # list of single state element (t=0)
                # you could also return multiple values, e.g., [0, "foo"]

def compute_actions(self,
                    obs_batch,
                    state_batches,
                    prev_action_batch=None,
                    prev_reward_batch=None,
                    info_batch=None,
                    episodes=None,
                    **kwargs):
    assert len(state_batches) == len(self.get_initial_state())
    new_state_batches = [[
        t + 1 for t in state_batches[0]
    ]]
    return ..., new_state_batches, {}

def learn_on_batch(self, samples):
    # can access array of the state elements at each timestep
    # or state_in_1, 2, etc. if there are multiple state elements
    assert "state_in_0" in samples.keys()
    assert "state_out_0" in samples.keys()

```

**Extra action info output:** You can also emit extra outputs at each step which will be available for learning on. For example, you might want to output the behaviour policy logits as extra action info, which can be used for importance weighting, but in general arbitrary values can be stored here (as long as they are convertible to numpy arrays):

```

def compute_actions(self,
                    obs_batch,
                    state_batches,
                    prev_action_batch=None,
                    prev_reward_batch=None,
                    info_batch=None,
                    episodes=None,
                    **kwargs):
    action_info_batch = {
        "some_value": ["foo" for _ in obs_batch],
        "other_value": [12345 for _ in obs_batch],
    }
    return ..., [], action_info_batch

def learn_on_batch(self, samples):
    # can access array of the extra values at each timestep
    assert "some_value" in samples.keys()
    assert "other_value" in samples.keys()

```

## Policies in Multi-Agent

Beyond being agnostic of framework implementation, one of the main reasons to have a Policy abstraction is for use in multi-agent environments. For example, the [rock-paper-scissors](#) example shows how you can leverage the Policy abstraction to evaluate heuristic policies against learned policies.

### Building Policies in TensorFlow

This section covers how to build a TensorFlow RLLib policy using `tf_policy_template.build_tf_policy()`.

To start, you first have to define a loss function. In RLLib, loss functions are defined over batches of trajectory data produced by policy evaluation. A basic policy gradient loss that only tries to maximize the 1-step reward can be defined as follows:

```
import tensorflow as tf
from ray.rllib.policy.sample_batch import SampleBatch

def policy_gradient_loss(policy, model, dist_class, train_batch):
    actions = train_batch[SampleBatch.ACTIONS]
    rewards = train_batch[SampleBatch.REWARDS]
    logits, _ = model.from_batch(train_batch)
    action_dist = dist_class(logits, model)
    return -tf.reduce_mean(action_dist.logp(actions) * rewards)
```

In the above snippet, `actions` is a Tensor placeholder of shape `[batch_size, action_dim...]`, and `rewards` is a placeholder of shape `[batch_size]`. The `action_dist` object is an `ActionDistribution` that is parameterized by the output of the neural network policy model. Passing this loss function to `build_tf_policy` is enough to produce a very basic TF policy:

```
from ray.rllib.policy.tf_policy_template import build_tf_policy

# <class 'ray.rllib.policy.tf_policy_template.MyTFPolicy'>
MyTFPolicy = build_tf_policy(
    name="MyTFPolicy",
    loss_fn=policy_gradient_loss)
```

We can create a `Trainer` and try running this policy on a toy env with two parallel rollout workers:

```
import ray
from ray import tune
from ray.rllib.agents.trainer_template import build_trainer

# <class 'ray.rllib.agents.trainer_template.MyCustomTrainer'>
MyTrainer = build_trainer(
    name="MyCustomTrainer",
    default_policy=MyTFPolicy)

ray.init()
tune.run(MyTrainer, config={"env": "CartPole-v0", "num_workers": 2})
```

If you run the above snippet ([runnable file here](#)), you'll probably notice that CartPole doesn't learn so well:

```
== Status ==
Using FIFO scheduling algorithm.
Resources requested: 3/4 CPUs, 0/0 GPUs
```

(continues on next page)

(continued from previous page)

```
Memory usage on this node: 4.6/12.3 GB
Result logdir: /home/ubuntu/ray_results/MyAlgTrainer
Number of trials: 1 ({'RUNNING': 1})
RUNNING trials:
- MyAlgTrainer_CartPole-v0_0:      RUNNING, [3 CPUs, 0 GPUs], [pid=26784],
                                         32 s, 156 iter, 62400 ts, 23.1 rew
```

Let's modify our policy loss to include rewards summed over time. To enable this advantage calculation, we need to define a *trajectory postprocessor* for the policy. This can be done by defining `postprocess_fn`:

```
from ray.rllib.evaluation.postprocessing import compute_advantages, \
    Postprocessing

def postprocess_advantages(policy,
                           sample_batch,
                           other_agent_batches=None,
                           episode=None):
    return compute_advantages(
        sample_batch, 0.0, policy.config["gamma"], use_gae=False)

def policy_gradient_loss(policy, model, dist_class, train_batch):
    logits, _ = model.from_batch(train_batch)
    action_dist = dist_class(logits, model)
    return -tf.reduce_mean(
        action_dist.logp(train_batch[SampleBatch.ACTIONS]) *
        train_batch[Postprocessing.ADVANTAGES])

MyTFPolicy = build_tf_policy(
    name="MyTFPolicy",
    loss_fn=policy_gradient_loss,
    postprocess_fn=postprocess_advantages)
```

The `postprocess_advantages()` function above uses calls RLLib's `compute_advantages` function to compute advantages for each timestep. If you re-run the trainer with this improved policy, you'll find that it quickly achieves the max reward of 200.

You might be wondering how RLLib makes the advantages placeholder automatically available as `train_batch[Postprocessing.ADVANTAGES]`. When building your policy, RLLib will create a "dummy" trajectory batch where all observations, actions, rewards, etc. are zeros. It then calls your `postprocess_fn`, and generates TF placeholders based on the numpy shapes of the postprocessed batch. RLLib tracks which placeholders that `loss_fn` and `stats_fn` access, and then feeds the corresponding sample data into those placeholders during loss optimization. You can also access these placeholders via `policy.get_placeholder(<name>)` after loss initialization.

### Example 1: Proximal Policy Optimization

In the above section you saw how to compose a simple policy gradient algorithm with RLLib. In this example, we'll dive into how PPO was built with RLLib and how you can modify it. First, check out the [PPO trainer definition](#):

```
PPOTrainer = build_trainer(
    name="PPOTrainer",
    default_config=DEFAULT_CONFIG,
    default_policy=PPOTFPolicy,
    make_policy_optimizer=choose_policy_optimizer,
    validate_config=validate_config,
    after_optimizer_step=update_kl,
```

(continues on next page)

(continued from previous page)

```
before_train_step=warn_about_obs_filter,
after_train_result=warn_about_bad_reward_scales)
```

Besides some boilerplate for defining the PPO configuration and some warnings, there are two important arguments to take note of here: `make_policy_optimizer=choose_policy_optimizer`, and `after_optimizer_step=update_kl`.

The `choose_policy_optimizer` function chooses which *Policy Optimizer* to use for distributed training. You can think of these policy optimizers as coordinating the distributed workflow needed to improve the policy. Depending on the trainer config, PPO can switch between a simple synchronous optimizer, or a multi-GPU optimizer that implements minibatch SGD (the default):

```
def choose_policy_optimizer(workers, config):
    if config["simple_optimizer"]:
        return SyncSamplesOptimizer(
            workers,
            num_sgd_iter=config["num_sgd_iter"],
            train_batch_size=config["train_batch_size"])

    return LocalMultiGPUOptimizer(
        workers,
        sgd_batch_size=config["sgd_minibatch_size"],
        num_sgd_iter=config["num_sgd_iter"],
        num_gpus=config["num_gpus"],
        rollout_fragment_length=config["rollout_fragment_length"],
        num_envs_per_worker=config["num_envs_per_worker"],
        train_batch_size=config["train_batch_size"],
        standardize_fields=["advantages"],
        straggler_mitigation=config["straggler_mitigation"])
```

Suppose we want to customize PPO to use an asynchronous-gradient optimization strategy similar to A3C. To do that, we could define a new function that returns `AsyncGradientsOptimizer` and override the `make_policy_optimizer` component of `PPOTrainer`.

```
from ray.rllib.agents.ppo import PPOTrainer
from ray.rllib.optimizers import AsyncGradientsOptimizer

def make_async_optimizer(workers, config):
    return AsyncGradientsOptimizer(workers, grads_per_step=100)

CustomTrainer = PPOTrainer.with_updates(
    make_policy_optimizer=make_async_optimizer)
```

The `with_updates` method that we use here is also available for Torch and TF policies built from templates.

Now let's take a look at the `update_kl` function. This is used to adaptively adjust the KL penalty coefficient on the PPO loss, which bounds the policy change per training step. You'll notice the code handles both single and multi-agent cases (where there are be multiple policies each with different KL coeffs):

```
def update_kl(trainer, fetches):
    if "kl" in fetches:
        # single-agent
        trainer.workers.local_worker().for_policy(
            lambda pi: pi.update_kl(fetches["kl"]))
    else:
```

(continues on next page)

(continued from previous page)

```

def update(pi, pi_id):
    if pi_id in fetches:
        pi.update_kl(fetches[pi_id] ["kl"])
    else:
        logger.debug("No data for {}, not updating kl".format(pi_id))

    # multi-agent
    trainer.workers.local_worker().foreach_trainable_policy(update)

```

The update\_kl method on the policy is defined in PPOTFPolicy via the KLCoefMixIn, along with several other advanced features. Let's look at each new feature used by the policy:

```

PPOTFPolicy = build_tf_policy(
    name="PPOTFPolicy",
    get_default_config=lambda: ray.rllib.agents.ppo.ppo.DEFAULT_CONFIG,
    loss_fn=ppo_surrogate_loss,
    stats_fn=kl_and_loss_stats,
    extra_action_fetches_fn=vf_preds_and_logits_fetches,
    postprocess_fn=postprocess_ppo_gae,
    gradients_fn=clip_gradients,
    before_loss_init=setup_mixins,
    mixins=[LearningRateSchedule, KLCoefMixIn, ValueNetworkMixIn])

```

stats\_fn: The stats function returns a dictionary of Tensors that will be reported with the training results. This also includes the kl metric which is used by the trainer to adjust the KL penalty. Note that many of the values below reference policy.loss\_obj, which is assigned by loss\_fn (not shown here since the PPO loss is quite complex). RLLib will always call stats\_fn after loss\_fn, so you can rely on using values saved by loss\_fn as part of your statistics:

```

def kl_and_loss_stats(policy, train_batch):
    policy.explained_variance = explained_variance(
        train_batch[Postprocessing.VALUE_TARGETS], policy.model.value_function())

    stats_fetches = {
        "cur_kl_coeff": policy.kl_coeff,
        "cur_lr": tf.cast(policy.cur_lr, tf.float64),
        "total_loss": policy.loss_obj.loss,
        "policy_loss": policy.loss_obj.mean_policy_loss,
        "vf_loss": policy.loss_obj.mean_vf_loss,
        "vf_explained_var": policy.explained_variance,
        "kl": policy.loss_obj.mean_kl,
        "entropy": policy.loss_obj.mean_entropy,
    }

    return stats_fetches

```

extra\_actions\_fetches\_fn: This function defines extra outputs that will be recorded when generating actions with the policy. For example, this enables saving the raw policy logits in the experience batch, which e.g. means it can be referenced in the PPO loss function via batch[BEHAVIOUR\_LOGITS]. Other values such as the current value prediction can also be emitted for debugging or optimization purposes:

```

def vf_preds_and_logits_fetches(policy):
    return {
        SampleBatch.VF_PREDS: policy.model.value_function(),
        BEHAVIOUR_LOGITS: policy.model.last_output(),
    }

```

gradients\_fn: If defined, this function returns TF gradients for the loss function. You'd typically only want to override this to apply transformations such as gradient clipping:

```
def clip_gradients(policy, optimizer, loss):
    if policy.config["grad_clip"] is not None:
        grads = tf.gradients(loss, policy.model.trainable_variables())
        policy.grads, _ = tf.clip_by_global_norm(grads,
                                                policy.config["grad_clip"])
        clipped_grads = list(zip(policy.grads, policy.model.trainable_variables()))
        return clipped_grads
    else:
        return optimizer.compute_gradients(
            loss, colocate_gradients_with_ops=True)
```

mixins: To add arbitrary stateful components, you can add mixin classes to the policy. Methods defined by these mixins will have higher priority than the base policy class, so you can use these to override methods (as in the case of LearningRateSchedule), or define extra methods and attributes (e.g., KLCoefMixIn, ValueNetworkMixin). Like any other Python superclass, these should be initialized at some point, which is what the setup\_mixins function does:

```
def setup_mixins(policy, obs_space, action_space, config):
    ValueNetworkMixin.__init__(policy, obs_space, action_space, config)
    KLCoefMixIn.__init__(policy, config)
    LearningRateSchedule.__init__(policy, config["lr"], config["lr_schedule"])
```

In PPO we run setup\_mixins before the loss function is called (i.e., before\_loss\_init), but other callbacks you can use include before\_init and after\_init.

### Example 2: Deep Q Networks

Let's look at how to implement a different family of policies, by looking at the SimpleQ policy definition:

```
SimpleQPolicy = build_tf_policy(
    name="SimpleQPolicy",
    get_default_config=lambda: ray.rllib.agents.dqn.dqn.DEFAULT_CONFIG,
    make_model=build_q_models,
    action_sampler_fn=build_action_sampler,
    loss_fn=build_q_losses,
    extra_action_feed_fn=exploration_setting_inputs,
    extra_action_fetches_fn=lambda policy: {"q_values": policy.q_values},
    extra_learn_fetches_fn=lambda policy: {"td_error": policy.td_error},
    before_init=setup_early_mixins,
    after_init=setup_late_mixins,
    obs_include_prev_action_reward=False,
    mixins=[ExplorationStateMixin, TargetNetworkMixin],
)
```

The biggest difference from the policy gradient policies you saw previously is that SimpleQPolicy defines its own make\_model and action\_sampler\_fn. This means that the policy builder will not internally create a model and action distribution, rather it will call build\_q\_models and build\_action\_sampler to get the output action tensors.

The model creation function actually creates two different models for DQN: the base Q network, and also a target network. It requires each model to be of type SimpleQModel, which implements a get\_q\_values() method. The model catalog will raise an error if you try to use a custom ModelV2 model that isn't a subclass of SimpleQModel. Similarly, the full DQN policy requires models to subclass DistributionalQModel, which implements get\_q\_value\_distributions() and get\_state\_value():

```
def build_q_models(policy, obs_space, action_space, config):
    ...

    policy.q_model = ModelCatalog.get_model_v2(
        obs_space,
        action_space,
        num_outputs,
        config["model"],
        framework="tf",
        name=Q_SCOPE,
        model_interface=SimpleQModel,
        q_hiddens=config["hiddens"])

    policy.target_q_model = ModelCatalog.get_model_v2(
        obs_space,
        action_space,
        num_outputs,
        config["model"],
        framework="tf",
        name=Q_TARGET_SCOPE,
        model_interface=SimpleQModel,
        q_hiddens=config["hiddens"])

    return policy.q_model
```

The action sampler is straightforward, it just takes the `q_model`, runs a forward pass, and returns the argmax over the actions:

```
def build_action_sampler(policy, q_model, input_dict, obs_space, action_space,
                        config):
    # do max over Q values...
    ...
    return action, action_logp
```

The remainder of DQN is similar to other algorithms. Target updates are handled by a `after_optimizer_step` callback that periodically copies the weights of the Q network to the target.

Finally, note that you do not have to use `build_tf_policy` to define a TensorFlow policy. You can alternatively subclass `Policy`, `TFPolicy`, or `DynamicTFPolicy` as convenient.

## Building Policies in TensorFlow Eager

Policies built with `build_tf_policy` (most of the reference algorithms are) can be run in eager mode by setting the `"eager": True / "eager_tracing": True` config options or using `rllib train --eager [--trace]`. This will tell RLlib to execute the model forward pass, action distribution, loss, and stats functions in eager mode.

Eager mode makes debugging much easier, since you can now use line-by-line debugging with breakpoints or Python `print()` to inspect intermediate tensor values. However, eager can be slower than graph mode unless tracing is enabled.

You can also selectively leverage eager operations within graph mode execution with `tf.py_function`. Here's an example of using eager ops embedded [within a loss function](#).

## Building Policies in PyTorch

Defining a policy in PyTorch is quite similar to that for TensorFlow (and the process of defining a trainer given a Torch policy is exactly the same). Here's a simple example of a trivial torch policy ([runnable file here](#)):

```
from ray.rllib.policy.sample_batch import SampleBatch
from ray.rllib.policy.torch_policy_template import build_torch_policy

def policy_gradient_loss(policy, model, dist_class, train_batch):
    logits, _ = model.from_batch(train_batch)
    action_dist = dist_class(logits)
    log_probs = action_dist.logp(train_batch[SampleBatch.ACTIONS])
    return -train_batch[SampleBatch.REWARDS].dot(log_probs)

# <class 'ray.rllib.policy.torch_policy_template.MyTorchPolicy'>
MyTorchPolicy = build_torch_policy(
    name="MyTorchPolicy",
    loss_fn=policy_gradient_loss)
```

Now, building on the TF examples above, let's look at how the A3C torch policy is defined:

```
A3CTorchPolicy = build_torch_policy(
    name="A3CTorchPolicy",
    get_default_config=lambda: ray.rllib.agents.a3c.a3c.DEFAULT_CONFIG,
    loss_fn=actor_critic_loss,
    stats_fn=loss_and_entropy_stats,
    postprocess_fn=add_advantages,
    extra_action_out_fn=model_value_predictions,
    extra_grad_process_fn=apply_grad_clipping,
    optimizer_fn=torch_optimizer,
    mixins=[ValueNetworkMixin])
```

`loss_fn`: Similar to the TF example, the actor critic loss is defined over batch. We imperatively execute the forward pass by calling `model()` on the observations followed by `dist_class()` on the output logits. The output Tensors are saved as attributes of the policy object (e.g., `policy.entropy = dist.entropy.mean()`), and we return the scalar loss:

```
def actor_critic_loss(policy, model, dist_class, train_batch):
    logits, _ = model.from_batch(train_batch)
    values = model.value_function()
    action_dist = dist_class(logits)
    log_probs = action_dist.logp(train_batch[SampleBatch.ACTIONS])
    policy.entropy = action_dist.entropy().mean()
    ...
    return overall_err
```

`stats_fn`: The stats function references `entropy`, `pi_err`, and `value_err` saved from the call to the loss function, similar in the PPO TF example:

```
def loss_and_entropy_stats(policy, train_batch):
    return {
        "policy_entropy": policy.entropy.item(),
        "policy_loss": policy.pi_err.item(),
        "vf_loss": policy.value_err.item(),
    }
```

`extra_action_out_fn`: We save value function predictions given model outputs. This makes the value function predictions of the model available in the trajectory as `batch[SampleBatch.VF_PREDS]`:

```
def model_value_predictions(policy, input_dict, state_batches, model):
    return {SampleBatch.VF_PREDS: model.value_function().cpu().numpy()}
```

`postprocess_fn` and `mixins`: Similar to the PPO example, we need access to the value function during postprocessing (i.e., `add_advantages` below calls `policy._value()`). The value function is exposed through a mixin class that defines the method:

```
def add_advantages(policy,
                   sample_batch,
                   other_agent_batches=None,
                   episode=None):
    completed = sample_batch[SampleBatch.DONES][-1]
    if completed:
        last_r = 0.0
    else:
        last_r = policy._value(sample_batch[SampleBatch.NEXT_OBS][-1])
    return compute_advantages(sample_batch, last_r, policy.config["gamma"],
                             policy.config["lambda"])

class ValueNetworkMixin(object):
    def __init__(self, obs):
        with self.lock:
            obs = torch.from_numpy(obs).float().unsqueeze(0).to(self.device)
            _, _, vf, _ = self.model({"obs": obs}, [])
        return vf.detach().cpu().numpy().squeeze()
```

You can find the full policy definition in `a3c_torch_policy.py`.

In summary, the main differences between the PyTorch and TensorFlow policy builder functions is that the TF loss and stats functions are built symbolically when the policy is initialized, whereas for PyTorch (or TensorFlow Eager) these functions are called imperatively each time they are used.

## Extending Existing Policies

You can use the `with_updates` method on Trainers and Policy objects built with `make_*` to create a copy of the object with some changes, for example:

```
from ray.rllib.agents.ppo import PPOTrainer
from ray.rllib.agents.ppo.ppo_tf_policy import PPOTFPolicy

CustomPolicy = PPOTFPolicy.with_updates(
    name="MyCustomPPOTFPolicy",
    loss_fn=some_custom_loss_fn)

CustomTrainer = PPOTrainer.with_updates(
    default_policy=CustomPolicy)
```

## 5.22.2 Policy Evaluation

Given an environment and policy, 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 `RolloutWorker` class that manages all of this, and this class is used in most RLlib algorithms.

You can use rollout workers standalone to produce batches of experiences. This can be done by calling `worker.sample()` on a worker instance, or `worker.sample.remote()` in parallel on worker instances created as Ray actors (see [WorkerSet](#)).

Here is an example of creating a set of rollout workers and using them gather experiences in parallel. The trajectories are concatenated, the policy learns on the trajectory batch, and then we broadcast the policy weights to the workers for the next round of rollouts:

```
# Setup policy and rollout workers
env = gym.make("CartPole-v0")
policy = CustomPolicy(env.observation_space, env.action_space, {})
workers = WorkerSet(
    policy=CustomPolicy,
    env_creator=lambda c: gym.make("CartPole-v0"),
    num_workers=10)

while True:
    # Gather a batch of samples
    T1 = SampleBatch.concat_samples(
        ray.get([w.sample.remote() for w in workers.remote_workers()]))

    # Improve the policy using the T1 batch
    policy.learn_on_batch(T1)

    # Broadcast weights to the policy evaluation workers
    weights = ray.put({"default_policy": policy.get_weights()})
    for w in workers.remote_workers():
        w.set_weights.remote(weights)
```

## 5.22.3 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.

For example, in A3C you’d want to compute gradients asynchronously on different workers, and apply them to a central policy 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.

This is how the example in the previous section looks when written using a policy optimizer:

```
# Same setup as before
workers = WorkerSet(
    policy=CustomPolicy,
    env_creator=lambda c: gym.make("CartPole-v0"),
    num_workers=10)

# this optimizer implements the IMPALA architecture
optimizer = AsyncSamplesOptimizer(workers, train_batch_size=500)
```

(continues on next page)

(continued from previous page)

```
while True:
    optimizer.step()
```

## 5.22.4 Trainers

Trainers are the boilerplate classes that put the above components together, making algorithms accessible via Python API and the command line. They manage algorithm configuration, setup of the rollout workers and optimizer, and collection of training metrics. Trainers also implement the [Tune Trainable API](#) for easy experiment management.

Example of three equivalent ways of interacting with the PPO trainer, all of which log results in `~/ray_results`:

```
trainer = PPOTrainer(env="CartPole-v0", config={"train_batch_size": 4000})
while True:
    print(trainer.train())
```

```
rllib train --run=PPO --env=CartPole-v0 --config='{"train_batch_size": 4000}'
```

```
from ray import tune
tune.run(PPOTrainer, config={"env": "CartPole-v0", "train_batch_size": 4000})
```

## 5.23 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.

### 5.23.1 Tuned Examples

- **Tuned examples:** Collection of tuned algorithm hyperparameters.
- **Atari benchmarks:** Collection of reasonably optimized Atari results.

### 5.23.2 Blog Posts

- **Scaling Multi-Agent Reinforcement Learning:** This blog post is a brief tutorial on multi-agent RL and its design in RLlib.
- **Functional RL with Keras and TensorFlow Eager:** Exploration of a functional paradigm for implementing reinforcement learning (RL) algorithms.

### 5.23.3 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.
- **Using rollout workers directly for control over the whole training workflow:** Example of how to use RLlib's lower-level building blocks to implement a fully customized training workflow.

### 5.23.4 Custom Envs and Models

- **Registering a custom env and model:** Example of defining and registering a gym env and model for use with RLlib.
- **Custom Keras model:** Example of using a custom Keras model.
- **Custom Keras RNN model:** Example of using a custom Keras RNN model.
- **Registering a custom model with supervised loss:** Example of defining and registering a custom model with a supervised loss.
- **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.
- **Eager execution:** Example of how to leverage TensorFlow eager to simplify debugging and design of custom models and policies.

### 5.23.5 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.

### 5.23.6 Multi-Agent and Hierarchical

- **Rock-paper-scissors:** Example of different heuristic and learned policies competing against each other in rock-paper-scissors.
- **Two-step game:** Example of the two-step game from the QMIX paper.
- **PPO with centralized critic on two-step game:** Example of customizing PPO to leverage a centralized value function.
- **Centralized critic in the env:** A simpler method of implementing a centralized critic by augmentating agent observations with global information.
- **Hand-coded policy:** Example of running a custom hand-coded policy alongside trainable policies.
- **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.

### 5.23.7 Community Examples

- **CARLA:** Example of training autonomous vehicles with RLlib and CARLA simulator.
- **GFootball:** Example of setting up a multi-agent version of GFootball with RLlib.
- **NeuroCuts:** Example of building packet classification trees using RLlib / multi-agent in a bandit-like setting.
- **NeuroVectorizer:** Example of learning optimal LLVM vectorization compiler pragmas for loops in C and C++ codes using RLlib.
- **Roboschool / SageMaker:** Example of training robotic control policies in SageMaker with RLlib.
- **StarCraft2:** Example of training in StarCraft2 maps with RLlib / multi-agent.
- **Traffic Flow:** Example of optimizing mixed-autonomy traffic simulations with RLlib / multi-agent.
- **Sequential Social Dilemma Games:** Example of using the multi-agent API to model several social dilemma games.

## 5.24 RLlib Package Reference

### 5.24.1 ray.rllib.policy

`class ray.rllib.policy.Policy(observation_space, action_space, config)`

An agent policy and loss, i.e., a TFPolicy 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 Policy, however TensorFlow users may find TFPolicy simpler to implement. TFPolicy 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

#### `exploration`

The exploration object to use for computing actions, or None.

**Type** Exploration

`abstract compute_actions(obs_batch, state_batches=None, prev_action_batch=None, prev_reward_batch=None, info_batch=None, episodes=None, explore=None, timestep=None, **kwargs)`

Computes actions for the current policy.

#### Parameters

- `obs_batch` (`Union[List, np.ndarray]`) – Batch of observations.

- **state\_batches** (*Optional[list]*) – List of RNN state input batches, if any.
- **prev\_action\_batch** (*Optional[List, np.ndarray]*) – Batch of previous action values.
- **prev\_reward\_batch** (*Optional[List, 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.
- **explore** (*bool*) – Whether to pick an exploitation or exploration action (default: None -> use self.config[“explore”]).
- **timestep** (*int*) – The current (sampling) time step.
- **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=None, prev\_action=None, prev\_reward=None, info=None, episode=None, clip\_actions=False, explore=None, timestep=None, \*\*kwargs*)

Unbatched version of compute\_actions.

#### Parameters

- **obs** (*obj*) – Single observation.
- **state** (*list*) – List of RNN state inputs, if any.
- **prev\_action** (*obj*) – Previous action value, if any.
- **prev\_reward** (*float*) – 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.
- **clip\_actions** (*bool*) – Should actions be clipped?
- **explore** (*bool*) – Whether to pick an exploitation or exploration action (default: None -> use self.config[“explore”]).
- **timestep** (*int*) – The current (sampling) time step.
- **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)

---

**compute\_log\_likelihoods** (*actions*, *obs\_batch*, *state\_batches=None*, *prev\_action\_batch=None*, *prev\_reward\_batch=None*)

Computes the log-prob/lielihood for a given action and observation.

#### Parameters

- **actions** (*Union[List, np.ndarray]*) – Batch of actions, for which to retrieve the log-probs/likenesses (given all other inputs: obs, states, ..).
- **obs\_batch** (*Union[List, np.ndarray]*) – Batch of observations.
- **state\_batches** (*Optional[list]*) – List of RNN state input batches, if any.
- **prev\_action\_batch** (*Optional[List, np.ndarray]*) – Batch of previous action values.
- **prev\_reward\_batch** (*Optional[List, np.ndarray]*) – Batch of previous rewards.

#### Returns

**Batch of log probs/likenesses, with** shape: [BATCH\_SIZE].

**Return type** log-likenesses (*np.ndarray*)

**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, agent\_batch) tuples containing the policy and experiences of the other agents.
- **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*

**learn\_on\_batch** (*samples*)

Fused compute gradients and apply gradients call.

Either this or the combination of compute/apply grads must be implemented by subclasses.

**Returns** dictionary of extra metadata from compute\_gradients().

**Return type** *grad\_info*

## Examples

```
>>> batch = ev.sample()  
>>> ev.learn_on_batch(samples)
```

### `compute_gradients` (*postprocessed\_batch*)

Computes gradients against a batch of experiences.

Either this or learn\_on\_batch() must be implemented by subclasses.

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

**Return type** grads (list)

### `apply_gradients` (*gradients*)

Applies previously computed gradients.

Either this or learn\_on\_batch() must be implemented by subclasses.

### `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_exploration_info` ()

Returns the current exploration information of this policy.

This information depends on the policy's Exploration object.

**Returns** Serializable information on the *self.exploration* object.

**Return type** any

### `is_recurrent` ()

Whether this Policy holds a recurrent Model.

**Returns** True if this Policy has-a RNN-based Model.

**Return type** bool

### `num_state_tensors` ()

The number of internal states needed by the RNN-Model of the Policy.

**Returns** The number of RNN internal states kept by this Policy's Model.

**Return type** int

### `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 Policy to local directory for serving.

**Parameters** `export_dir` (*str*) – Local writable directory.

**export\_checkpoint** (*export\_dir*)  
Export Policy checkpoint to local directory.

**Argument:** `export_dir` (*str*): Local writable directory.

**import\_model\_from\_h5** (*import\_file*)  
Imports Policy from local file.

**Parameters** `import_file` (*str*) – Local readable file.

```
class ray.rllib.policy.TorchPolicy(observation_space, action_space, config, *, model, loss,
                                    action_distribution_class,      action_sampler_fn=None,
                                    action_distribution_fn=None,     max_seq_len=20,
                                    get_batch_divisibility_req=None)
```

Template for a PyTorch policy and loss to use with RLLib.  
This is similar to TFPolicy, but for PyTorch.

**observation\_space**  
observation space of the policy.  
**Type** gym.Space

**action\_space**  
action space of the policy.  
**Type** gym.Space

**config**  
config of the policy.  
**Type** dict

**model**  
Torch model instance.  
**Type** TorchModel

**dist\_class**  
Torch action distribution class.  
**Type** type

**compute\_actions** (*obs\_batch*, *state\_batches=None*, *prev\_action\_batch=None*, *prev\_reward\_batch=None*, *info\_batch=None*, *episodes=None*, *explore=None*, *timestep=None*, *\*\*kwargs*)  
Computes actions for the current policy.

**Parameters**

- **obs\_batch** (*Union[List, np.ndarray]*) – Batch of observations.
- **state\_batches** (*Optional[list]*) – List of RNN state input batches, if any.

- **prev\_action\_batch** (*Optional[List, np.ndarray]*) – Batch of previous action values.
- **prev\_reward\_batch** (*Optional[List, 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.
- **explore** (*bool*) – Whether to pick an exploitation or exploration action (default: None -> use self.config[“explore”]).
- **timestep** (*int*) – The current (sampling) time step.
- **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\_log\_likelihoods** (*actions, obs\_batch, state\_batches=None, prev\_action\_batch=None, prev\_reward\_batch=None*)

Computes the log-prob/likeness for a given action and observation.

#### Parameters

- **actions** (*Union[List, np.ndarray]*) – Batch of actions, for which to retrieve the log-probs/likenesses (given all other inputs: obs, states, ..).
- **obs\_batch** (*Union[List, np.ndarray]*) – Batch of observations.
- **state\_batches** (*Optional[list]*) – List of RNN state input batches, if any.
- **prev\_action\_batch** (*Optional[List, np.ndarray]*) – Batch of previous action values.
- **prev\_reward\_batch** (*Optional[List, np.ndarray]*) – Batch of previous rewards.

#### Returns

**Batch of log probs/likenesses, with** shape: [BATCH\_SIZE].

**Return type** log-likenesses (np.ndarray)

**learn\_on\_batch** (*postprocessed\_batch*)

Fused compute gradients and apply gradients call.

Either this or the combination of compute/apply grads must be implemented by subclasses.

**Returns** dictionary of extra metadata from compute\_gradients().

**Return type** grad\_info

## Examples

```
>>> batch = ev.sample()
>>> ev.learn_on_batch(samples)
```

### `compute_gradients` (*postprocessed\_batch*)

Computes gradients against a batch of experiences.

Either this or learn\_on\_batch() must be implemented by subclasses.

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

**Return type** grads (list)

### `apply_gradients` (*gradients*)

Applies previously computed gradients.

Either this or learn\_on\_batch() must be implemented by subclasses.

### `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

### `is_recurrent` ()

Whether this Policy holds a recurrent Model.

**Returns** True if this Policy has-a RNN-based Model.

**Return type** bool

### `num_state_tensors` ()

The number of internal states needed by the RNN-Model of the Policy.

**Returns** The number of RNN internal states kept by this Policy's Model.

**Return type** int

### `get_initial_state` ()

Returns initial RNN state for the current policy.

### `extra_grad_process` (*optimizer, loss*)

Called after each optimizer.zero\_grad() + loss.backward() call.

Called for each self.\_optimizers/loss-value pair. Allows for gradient processing before optimizer.step() is called. E.g. for gradient clipping.

#### Parameters

- `optimizer` (`torch.optim.Optimizer`) – A torch optimizer object.
- `loss` (`torch.Tensor`) – The loss tensor associated with the optimizer.

**Returns** An info dict.

**Return type** dict

### `extra_action_out` (*input\_dict, state\_batches, model, action\_dist*)

Returns dict of extra info to include in experience batch.

**Parameters**

- **input\_dict** (*dict*) – Dict of model input tensors.
- **state\_batches** (*list*) – List of state tensors.
- **model** ([TorchModelV2](#)) – Reference to the model.
- **action\_dist** (*TorchActionDistribution*) – Torch action dist object to get log-probs (e.g. for already sampled actions).

**extra\_grad\_info** (*train\_batch*)

Return dict of extra grad info.

**optimizer()**

Custom PyTorch optimizer to use.

**export\_model** (*export\_dir*)

TODO(sven): implement for torch.

**export\_checkpoint** (*export\_dir*)

TODO(sven): implement for torch.

**import\_model\_from\_h5** (*import\_file*)

Imports weights into torch model.

```
class ray.rllib.policy.TFPolicy(observation_space,      action_space,      config,      sess,
                                 obs_input,      sampled_action,      loss,      loss_inputs,
                                 model=None,      sampled_action_logp=None,      ac-
                                 tion_input=None,      log_likelihood=None,      dist_inputs=None,
                                 dist_class=None,      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,      explore=None,      timestep=None)
```

An agent policy and loss implemented in TensorFlow.

Extending this class enables RLLib to perform TensorFlow specific optimizations on the policy, 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**model**

RLLib model used for the policy.

**Type** [rllib.models.Model](#)

## Examples

```
>>> policy = TFPolicySubclass(
    sess, obs_input, sampled_action, loss, loss_inputs)
```

```
>>> print(policy.compute_actions([1, 0, 2]))
(array([0, 1, 1]), [], {})
```

```
>>> print(policy.postprocess_trajectory(SampleBatch({...})))
SampleBatch({"action": ..., "advantages": ..., ...})
```

### `variables()`

Return the list of all savable variables for this policy.

### `get_placeholder(name)`

Returns the given action or loss input placeholder by name.

If the loss has not been initialized and a loss input placeholder is requested, an error is raised.

### `get_session()`

Returns a reference to the TF session for this policy.

### `loss_initialized()`

Returns whether the loss function has been initialized.

```
compute_actions(obs_batch, state_batches=None, prev_action_batch=None,
                prev_reward_batch=None, info_batch=None, episodes=None, explore=None,
                timestep=None, **kwargs)
```

Computes actions for the current policy.

### Parameters

- `obs_batch` (`Union[List, np.ndarray]`) – Batch of observations.
- `state_batches` (`Optional[list]`) – List of RNN state input batches, if any.
- `prev_action_batch` (`Optional[List, np.ndarray]`) – Batch of previous action values.
- `prev_reward_batch` (`Optional[List, 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.
- `explore` (`bool`) – Whether to pick an exploitation or exploration action (default: `None` -> use `self.config["explore"]`).
- `timestep` (`int`) – The current (sampling) time step.
- `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 [BATCH\_SIZE, ...], “f2”: [BATCH\_SIZE, ...]}.

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

**compute\_log\_likelihoods** (actions, obs\_batch, state\_batches=None, prev\_action\_batch=None, prev\_reward\_batch=None)

Computes the log-prob/likelihood for a given action and observation.

#### Parameters

- **actions** (*Union[List, np.ndarray]*) – Batch of actions, for which to retrieve the log-probs/likelihoods (given all other inputs: obs, states, ..).
- **obs\_batch** (*Union[List, np.ndarray]*) – Batch of observations.
- **state\_batches** (*Optional[list]*) – List of RNN state input batches, if any.
- **prev\_action\_batch** (*Optional[List, np.ndarray]*) – Batch of previous action values.
- **prev\_reward\_batch** (*Optional[List, np.ndarray]*) – Batch of previous rewards.

#### Returns

**Batch of log probs/likelihoods, with** shape: [BATCH\_SIZE].

**Return type** log-likelihoods (np.ndarray)

**compute\_gradients** (postprocessed\_batch)

Computes gradients against a batch of experiences.

Either this or learn\_on\_batch() must be implemented by subclasses.

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

**Return type** grads (list)

**apply\_gradients** (gradients)

Applies previously computed gradients.

Either this or learn\_on\_batch() must be implemented by subclasses.

**learn\_on\_batch** (postprocessed\_batch)

Fused compute gradients and apply gradients call.

Either this or the combination of compute/apply grads must be implemented by subclasses.

**Returns** dictionary of extra metadata from compute\_gradients().

**Return type** grad\_info

## Examples

```
>>> batch = ev.sample()
>>> ev.learn_on_batch(samples)
```

**get\_exploration\_info()**

Returns the current exploration information of this policy.

This information depends on the policy’s Exploration object.

**Returns** Serializable information on the *self.exploration* object.

**Return type** any

**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.

**import\_model\_from\_h5(import\_file)**  
Imports weights into tf model.

**copy(existing\_inputs)**  
Creates a copy of self using existing input placeholders.  
Optional, only required to work with the multi-GPU optimizer.

**is\_recurrent()**  
Whether this Policy holds a recurrent Model.

**Returns** True if this Policy has-a RNN-based Model.

**Return type** bool

**num\_state\_tensors()**  
The number of internal states needed by the RNN-Model of the Policy.

**Returns** The number of RNN internal states kept by this Policy's Model.

**Return type** int

**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().  
By default we return action probability/log-likelihood info and action distribution inputs (if present).

**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().

**optimizer()**  
TF optimizer to use for policy optimization.

**gradients(optimizer, loss)**  
Override for custom gradient computation.

**build\_apply\_op(optimizer, grads\_and\_vars)**  
Override for custom gradient apply computation.

```
ray.rllib.policy.build_torch_policy(name, *, loss_fn, get_default_config=None,
                                    stats_fn=None, postprocess_fn=None, extra_action_out_fn=None, extra_grad_process_fn=None, optimizer_fn=None, before_init=None, after_init=None, action_sampler_fn=None, action_distribution_fn=None, make_model=None, make_model_and_action_dist=None, apply_gradients_fn=None, mixins=None, get_batch_divisibility_req=None)
```

Helper function for creating a torch policy class at runtime.

### Parameters

- **name** (*str*) – name of the policy (e.g., “PPOTorchPolicy”)
- **loss\_fn** (*callable*) – Callable that returns a loss tensor as arguments given (policy, model, dist\_class, train\_batch).
- **get\_default\_config** (*Optional[callable]*) – Optional callable that returns the default config to merge with any overrides.
- **stats\_fn** (*Optional[callable]*) – Optional callable that returns a dict of values given the policy and batch input tensors.
- **postprocess\_fn** (*Optional[callable]*) – Optional experience postprocessing function that takes the same args as Policy.postprocess\_trajectory().
- **extra\_action\_out\_fn** (*Optional[callable]*) – Optional callable that returns a dict of extra values to include in experiences.
- **extra\_grad\_process\_fn** (*Optional[callable]*) – Optional callable that is called after gradients are computed and returns processing info.
- **optimizer\_fn** (*Optional[callable]*) – Optional callable that returns a torch optimizer given the policy and config.
- **before\_init** (*Optional[callable]*) – Optional callable to run at the beginning of Policy.\_\_init\_\_ that takes the same arguments as the Policy constructor.
- **after\_init** (*Optional[callable]*) – Optional callable to run at the end of policy init that takes the same arguments as the policy constructor.
- **action\_sampler\_fn** (*Optional[callable]*) – Optional callable returning a sampled action and its log-likelihood given some (obs and state) inputs.
- **action\_distribution\_fn** (*Optional[callable]*) – A callable that takes the Policy, Model, the observation batch, an explore-flag, a timestep, and an is\_training flag and returns a tuple of a) distribution inputs (parameters), b) a dist-class to generate an action distribution object from, and c) internal-state outputs (empty list if not applicable).
- **make\_model** (*Optional[callable]*) – Optional func that takes the same arguments as Policy.\_\_init\_\_ and returns a model instance. The distribution class will be determined automatically. Note: Only one of *make\_model* or *make\_model\_and\_action\_dist* should be provided.
- **make\_model\_and\_action\_dist** (*Optional[callable]*) – Optional func that takes the same arguments as Policy.\_\_init\_\_ and returns a tuple of model instance and torch action distribution class. Note: Only one of *make\_model* or *make\_model\_and\_action\_dist* should be provided.
- **apply\_gradients\_fn** (*Optional[callable]*) – Optional callable that takes a grads list and applies these to the Model’s parameters.

- **mixins** (*list*) – list of any class mixins for the returned policy class. These mixins will be applied in order and will have higher precedence than the TorchPolicy class.
- **get\_batch\_divisibility\_req** (*Optional[callable]*) – Optional callable that returns the divisibility requirement for sample batches.

**Returns** TorchPolicy child class constructed from the specified args.

**Return type** type

```
ray.rllib.policy.build_tf_policy(name, *, loss_fn, get_default_config=None, postprocess_fn=None, stats_fn=None, optimizer_fn=None, gradients_fn=None, apply_gradients_fn=None, grad_stats_fn=None, extra_action_fetches_fn=None, extra_learn_fetches_fn=None, before_init=None, before_loss_init=None, after_init=None, make_model=None, action_sampler_fn=None, action_distribution_fn=None, mixins=None, get_batch_divisibility_req=None, obs_include_prev_action_reward=True)
```

Helper function for creating a dynamic tf policy at runtime.

**Functions will be run in this order to initialize the policy:**

1. Placeholder setup: postprocess\_fn
2. Loss init: loss\_fn, stats\_fn
3. **Optimizer init: optimizer\_fn, gradients\_fn, apply\_gradients\_fn, grad\_stats\_fn**

This means that you can e.g., depend on any policy attributes created in the running of *loss\_fn* in later functions such as *stats\_fn*.

In eager mode, the following functions will be run repeatedly on each eager execution: loss\_fn, stats\_fn, gradients\_fn, apply\_gradients\_fn, and grad\_stats\_fn.

This means that these functions should not define any variables internally, otherwise they will fail in eager mode execution. Variable should only be created in make\_model (if defined).

### Parameters

- **name** (*str*) – name of the policy (e.g., “PPOTFPolicy”)
- **loss\_fn** (*func*) – function that returns a loss tensor as arguments (policy, model, dist\_class, train\_batch)
- **get\_default\_config** (*func*) – optional function that returns the default config to merge with any overrides
- **postprocess\_fn** (*func*) – optional experience postprocessing function that takes the same args as Policy.postprocess\_trajectory()
- **stats\_fn** (*func*) – optional function that returns a dict of TF fetches given the policy and batch input tensors
- **optimizer\_fn** (*func*) – optional function that returns a tf.Optimizer given the policy and config
- **gradients\_fn** (*func*) – optional function that returns a list of gradients given (policy, optimizer, loss). If not specified, this defaults to optimizer.compute\_gradients(loss)
- **apply\_gradients\_fn** (*func*) – optional function that returns an apply gradients op given (policy, optimizer, grads\_and\_vars)
- **grad\_stats\_fn** (*func*) – optional function that returns a dict of TF fetches given the policy, batch input, and gradient tensors

- **extra\_action\_fetches\_fn** (*func*) – optional function that returns a dict of TF fetches given the policy object
- **extra\_learn\_fetches\_fn** (*func*) – optional function that returns a dict of extra values to fetch and return when learning on a batch
- **before\_init** (*func*) – optional function to run at the beginning of policy init that takes the same arguments as the policy constructor
- **before\_loss\_init** (*func*) – optional function to run prior to loss init that takes the same arguments as the policy constructor
- **after\_init** (*func*) – optional function to run at the end of policy init that takes the same arguments as the policy constructor
- **make\_model** (*func*) – optional function that returns a ModelV2 object given (policy, obs\_space, action\_space, config). All policy variables should be created in this function. If not specified, a default model will be created.
- **action\_sampler\_fn** (*Optional[callable]*) – A callable returning a sampled action and its log-likelihood given some (obs and state) inputs.
- **action\_distribution\_fn** (*Optional[callable]*) – A callable returning distribution inputs (parameters), a dist-class to generate an action distribution object from, and internal-state outputs (or an empty list if not applicable).
- **mixins** (*list*) – list of any class mixins for the returned policy class. These mixins will be applied in order and will have higher precedence than the DynamicTFPolicy class
- **get\_batch\_divisibility\_req** (*func*) – optional function that returns the divisibility requirement for sample batches
- **obs\_include\_prev\_action\_reward** (*bool*) – whether to include the previous action and reward in the model input

**Returns** a DynamicTFPolicy instance that uses the specified args

## 5.24.2 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 RolloutWorker, for example:

```
gym.Env => rllib.VectorEnv => rllib.BaseEnv rllib.MultiAgentEnv => rllib.BaseEnv rl-lib.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_1": {
        "car_0": [8.0, 4.1],
    },
    "env_2": {
        "car_0": [2.3, 3.3],
        "car_1": [1.4, -0.2],
        "car_3": [1.2, 0.1],
    },
}
>>> 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*, *remote\_envs=False*, *remote\_env\_batch\_wait\_ms=0*)  
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)`

**stop()**

Releases all resources used.

**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,
    "car_1": -1,
    "traffic_light_1": 0,
}
```

(continues on next page)

(continued from previous page)

```
>>> 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))  
>>> trainer = DQNTrainer(env="my_env")  
>>> while True:  
    print(trainer.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.ExternalMultiAgentEnv` (*action\_space, observation\_space, max\_concurrent=100*)

This is the multi-agent version of ExternalEnv.

**run ()**

Override this to implement the multi-agent run loop.

**Your loop should continuously:**

1. Call `self.start_episode(episode_id)`
2. Call `self.get_action(episode_id, obs_dict)` -or- `self.log_action(episode_id, obs_dict, action_dict)`

3. Call self.log\_returns(episode\_id, reward\_dict)
4. Call self.end\_episode(episode\_id, obs\_dict)
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\_dict*)

Record an observation and get the on-policy action. observation\_dict is expected to contain the observation of all agents acting in this episode step.

**Parameters**

- **episode\_id** (*str*) – Episode id returned from start\_episode().
- **observation\_dict** (*dict*) – Current environment observation.

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

**Return type** action (dict)

**log\_action** (*episode\_id, observation\_dict, action\_dict*)

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

**Parameters**

- **episode\_id** (*str*) – Episode id returned from start\_episode().
- **observation\_dict** (*dict*) – Current environment observation.
- **action\_dict** (*dict*) – Action for the observation.

**log\_returns** (*episode\_id, reward\_dict, info\_dict=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\_dict** (*dict*) – Reward from the environment agents.
- **info** (*dict*) – Optional info dict.

**end\_episode** (*episode\_id, observation\_dict*)

Record the end of an episode.

**Parameters**

- **episode\_id** (*str*) – Episode id returned from start\_episode().
- **observation\_dict** (*dict*) – 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.

**class** ray.rllib.env.EnvContext (env\_config, worker\_index, vector\_index=0, remote=False)

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

**remote**

Whether environment should be remote or not.

**Type** bool

**class** ray.rllib.env.PolicyClient (address, inference\_mode='local', update\_interval=10.0)  
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.env.PolicyServerInput (ioctx, address, port)  
REST policy server that acts as an offline data source.

This launches a multi-threaded server that listens on the specified host and port to serve policy requests and forward experiences to RLlib. For high performance experience collection, it implements InputReader.

For an example, run `examples/cartpole_server.py` along with `examples/cartpole_client.py --inference-mode=local|remote`.

## Examples

```
>>> pg = PGTrainer(
...     env="CartPole-v0", config={
...         "input": lambda ioctx:
...             PolicyServerInput(ioctx, addr, port),
...         "num_workers": 0, # Run just 1 server, in the trainer.
...     }
>>> while True:
...     pg.train()
```

```
>>> client = PolicyClient("localhost:9900", inference_mode="local")
>>> eps_id = client.start_episode()
>>> action = client.get_action(eps_id, obs)
>>> ...
>>> client.log_returns(eps_id, reward)
>>> ...
>>> client.log_returns(eps_id, reward)
```

`next()`

Return the next batch of experiences read.

**Returns** SampleBatch or MultiAgentBatch read.

### 5.24.3 ray.rllib.evaluation

```
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 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 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())
```

**soft\_reset()**  
Clears rewards and metrics, but retains RNN and other state.  
This is used to carry state across multiple logical episodes in the same env (i.e., if *soft\_horizon* is set).

**policy\_for(agent\_id='agent0')**  
Returns the policy 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='agent0')**  
Returns the last observation for the specified agent.

**last\_raw\_obs\_for(agent\_id='agent0')**  
Returns the last un-preprocessed obs for the specified agent.

**last\_info\_for(agent\_id='agent0')**  
Returns the last info for the specified agent.

**last\_action\_for(agent\_id='agent0')**  
Returns the last action for the specified agent, or zeros.

**prev\_action\_for(agent\_id='agent0')**  
Returns the previous action for the specified agent.

**prev\_reward\_for(agent\_id='agent0')**  
Returns the previous reward for the specified agent.

```
rnn_state_for(agent_id='agent0')
```

Returns the last RNN state for the specified agent.

```
last_pi_info_for(agent_id='agent0')
```

Returns the last info object for the specified agent.

```
class ray.rllib.evaluation.RolloutWorker(env_creator, policy,
                                         policy_mapping_fn=None,
                                         policies_to_train=None, tf_session_creator=None,
                                         rollout_fragment_length=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, num_workers=0, monitor_path=None, log_dir=None, log_level=None,
                                         callbacks=None, input_creator=<function RolloutWorker.<lambda>>, in-
                                         put_evaluation=frozenset({}),
                                         put_creator=<function RolloutWorker.<lambda>>, out-
                                         mote_worker_envs=False,
                                         remote_env_batch_wait_ms=0,
                                         soft_horizon=False, no_done_at_end=False,
                                         seed=None, fake_sampler=False, extra_python_environ=None)
```

Common experience collection class.

This class wraps a policy 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 rollout worker and using it to collect experiences.
>>> worker = RolloutWorker(
...     env_creator=lambda _: gym.make("CartPole-v0"),
...     policy=PGTFFPolicy)
>>> print(worker.sample())
SampleBatch({
    "obs": [...], "actions": [...], "rewards": [...],
    "dones": [...], "new_obs": [...]})
```

```
>>> # Creating a multi-agent rollout worker
>>> worker = RolloutWorker(
...     env_creator=lambda _: MultiAgentTrafficGrid(num_cars=25),
...     policies={
...         # Use an ensemble of two policies for car agents
...         "car_policy1": [
...             (PGTFFPolicy, Box(...), Discrete(...), {"gamma": 0.99}),
...             "car_policy2":
```

(continues on next page)

(continued from previous page)

```

...
    (PGTFFPolicy, Box(...), Discrete(...), {"gamma": 0.95}),
...
    # Use a single shared policy for all traffic lights
...
    "traffic_light_policy":
        (PGTFFPolicy, Box(...), Discrete(...), {}),
...
    },
...
    policy_mapping_fn=lambda agent_id:
        random.choice(["car_policy1", "car_policy2"])
...
    if agent_id.startswith("car_") else "traffic_light_policy")
>>> print(worker.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, global\_vars=None)**

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.

Either this or learn\_on\_batch() 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 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.

Either this or `learn_on_batch()` must be implemented by subclasses.

## Examples

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

### `learn_on_batch` (*samples*)

Update policies based on the given batch.

This is the equivalent to `apply_gradients(compute_gradients(samples))`, but can be optimized to avoid pulling gradients into CPU memory.

Either this or the combination of `compute/apply grads` must be implemented by subclasses.

**Returns** dictionary of extra metadata from `compute_gradients()`.

**Return type** `info`

## Examples

```
>>> batch = ev.sample()
>>> ev.learn_on_batch(samples)
```

### `sample_and_learn` (*expected\_batch\_size*, *num\_sgd\_iter*, *sgd\_minibatch\_size*, *standardize\_fields*)

Sample and batch and learn on it.

This is typically used in combination with distributed allreduce.

#### Parameters

- **`expected_batch_size`** (*int*) – Expected number of samples to learn on.
- **`num_sgd_iter`** (*int*) – Number of SGD iterations.
- **`sgd_minibatch_size`** (*int*) – SGD minibatch size.
- **`standardize_fields`** (*list*) – List of sample fields to normalize.

**Returns** dictionary of extra metadata from `learn_on_batch()`. `count`: number of samples learned on.

**Return type** `info`

### `get_metrics` ()

Returns a list of new `RolloutMetric` objects from evaluation.

### `foreach_env` (*func*)

Apply the given function to each underlying env instance.

**get\_policy** (*policy\_id='default\_policy'*)

Return policy for the specified id, or None.

**Parameters** **policy\_id** (*str*) – id of policy to return.

**for\_policy** (*func, policy\_id='default\_policy'*)

Apply the given function to the specified policy.

**foreach\_policy** (*func*)

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

**foreach\_trainable\_policy** (*func*)

Applies the given function to each (policy, policy\_id) tuple, which can be found in *self.policies\_to\_train*.

**Parameters** **func** (*callable*) – A function - taking a Policy and its ID - that is called on all

Policies within *self.policies\_to\_train*.

#### Returns

**The list of n return values of all** *func([policy], [ID])*-calls.

**Return type** List[any]

**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)

**creation\_args** ()

Returns the args used to create this worker.

**setup\_torch\_data\_parallel** (*url, world\_rank, world\_size, backend*)

Join a torch process group for distributed SGD.

**get\_node\_ip** ()

Returns the IP address of the current node.

**find\_free\_port** ()

Finds a free port on the current node.

ray.rllib.evaluation.PolicyEvaluator

alias of ray.rllib.utils.deprecation.renamed\_class.<locals>.DeprecationWrapper

**class** ray.rllib.evaluation.EvaluatorInterface

This is the interface between policy optimizers and policy evaluation.

See also: RolloutWorker

**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], ...})
```

### `learn_on_batch(samples)`

Update policies based on the given batch.

This is the equivalent to `apply_gradients(compute_gradients(samples))`, but can be optimized to avoid pulling gradients into CPU memory.

Either this or the combination of `compute/apply grads` must be implemented by subclasses.

**Returns** dictionary of extra metadata from `compute_gradients()`.

**Return type** info

## Examples

```
>>> batch = ev.sample()
>>> ev.learn_on_batch(samples)
```

### `compute_gradients(samples)`

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

Either this or `learn_on_batch()` 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 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.

Either this or `learn_on_batch()` 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)
```

**get\_host ()**

Returns the hostname of the process running this evaluator.

**apply (func, \*args)**

Apply the given function to this evaluator instance.

`ray.rllib.evaluation.PolicyGraph`

alias of `ray.rllib.utils.deprecation.renamed_class.<locals>.DeprecationWrapper`

`ray.rllib.evaluation.TFPolicyGraph`

alias of `ray.rllib.utils.deprecation.renamed_class.<locals>.DeprecationWrapper`

`ray.rllib.evaluation.TorchPolicyGraph`

alias of `ray.rllib.utils.deprecation.renamed_class.<locals>.DeprecationWrapper`

**class ray.rllib.evaluation.MultiAgentBatch (\*args, \*\*kw)**

**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, callbacks)**

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\_agent\_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.SyncSampler(worker, env, policies, policy_mapping_fn, preprocessors, obs_filters, clip_rewards, rollout_fragment_length, callbacks, horizon=None, pack=False, tf_sess=None, clip_actions=True, soft_horizon=False, no_done_at_end=False)
```

```
class ray.rllib.evaluation.AsyncSampler(worker, env, policies, policy_mapping_fn, preprocessors, obs_filters, clip_rewards, rollout_fragment_length, callbacks, horizon=None, pack=False, tf_sess=None, clip_actions=True, blackhole_outputs=False, soft_horizon=False, no_done_at_end=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, use_critic=True)
```

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

#### Parameters

- **rollout** ([SampleBatch](#)) – SampleBatch of a single trajectory
- **last\_r** (*float*) – Value estimation for last observation
- **gamma** (*float*) – Discount factor.
- **lambda\_** (*float*) – Parameter for GAE
- **use\_gae** (*bool*) – Using Generalized Advantage Estimation
- **use\_critic** (*bool*) – Whether to use critic (value estimates). Setting this to False will use 0 as baseline.

#### Returns

**Object with experience from rollout and** processed rewards.

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

```
ray.rllib.evaluation.collect_metrics(local_worker=None,           remote_workers=[],  
                                      to_be_collected=[], timeout_seconds=180)
```

Gathers episode metrics from RolloutWorker instances.

```
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.

## 5.24.4 ray.rllib.models

**class** ray.rllib.models.**ActionDistribution** (*inputs*, *model*)

The policy action distribution of an agent.

**inputs**

input vector to compute samples from.

**Type** Tensors

**model**

reference to model producing the inputs.

**Type** ModelV2

**sample()**

Draw a sample from the action distribution.

**deterministic\_sample()**

Get the deterministic “sampling” output from the distribution. This is usually the max likelihood output, i.e. mean for Normal, argmax for Categorical, etc..

**sampled\_action\_logp()**

Returns the log probability of the last sampled action.

**logp** (*x*)

The log-likelihood of the action distribution.

**kl** (*other*)

The KL-divergence between two action distributions.

**entropy()**

The entropy of the action distribution.

**multi\_kl** (*other*)

The KL-divergence between two action distributions.

This differs from kl() in that it can return an array for MultiDiscrete. TODO(ekl) consider removing this.

**multi\_entropy()**

The entropy of the action distribution.

This differs from entropy() in that it can return an array for MultiDiscrete. TODO(ekl) consider removing this.

**static required\_model\_output\_shape** (*action\_space*, *model\_config*)

Returns the required shape of an input parameter tensor for a particular action space and an optional dict of distribution-specific options.

**Parameters**

- **action\_space** (*gym.Space*) – The action space this distribution will be used for, whose shape attributes will be used to determine the required shape of the input parameter tensor.
- **model\_config** (*dict*) – Model’s config dict (as defined in catalog.py)

**Returns**

**size of the** required input vector (minus leading batch dimension).

**Return type** model\_output\_shape (int or np.ndarray of ints)

**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_class, dist_dim = ModelCatalog.get_action_dist(
    env.action_space, {})
>>> model = ModelCatalog.get_model(inputs, dist_dim, options)
>>> dist = dist_class(model.outputs, model)
>>> action = dist.sample()
```

**static get\_action\_dist**(action\_space, config, dist\_type=None, framework='tf', \*\*kwargs)

Returns a distribution class and size for the given action space.

### Parameters

- **action\_space** (*Space*) – Action space of the target gym env.
- **config** (*Optional[dict]*) – Optional model config.
- **dist\_type** (*Optional[str]*) – Identifier of the action distribution.
- **framework** (*str*) – One of “tf” or “torch”.
- **kwargs** (*dict*) – Optional kwargs to pass on to the Distribution’s constructor.

**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\_shape**(action\_space)

Returns action tensor dtype and shape for the action space.

**Parameters** **action\_space** (*Space*) – Action space of the target gym env.

**Returns** Dtype and shape of the actions tensor.

**Return type** (dtype, shape)

**static get\_action\_placeholder**(action\_space, name=None)

Returns an action placeholder 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\_v2**(obs\_space, action\_space, num\_outputs, model\_config, framework='tf', name='default\_model', model\_interface=None, default\_model=None, \*\*model\_kwargs)

Returns a suitable model compatible with given spaces and output.

### Parameters

- **obs\_space** (*Space*) – Observation space of the target gym env. This may have an *original\_space* attribute that specifies how to unflatten the tensor into a ragged tensor.
- **action\_space** (*Space*) – Action space of the target gym env.
- **num\_outputs** (*int*) – The size of the output vector of the model.
- **framework** (*str*) – One of “tf” or “torch”.
- **name** (*str*) – Name (scope) for the model.
- **model\_interface** (*cls*) – Interface required for the model
- **default\_model** (*cls*) – Override the default class for the model. This only has an effect when not using a custom model
- **model\_kwargs** (*dict*) – args to pass to the ModelV2 constructor

**Returns** Model to use for the policy.

**Return type** model (ModelV2)

**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”: *preprocessor\_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.

**static register\_custom\_action\_dist** (*action\_dist\_name*, *action\_dist\_class*)

Register a custom action distribution class by name.

The model can be later used by specifying {“custom\_action\_dist”: *action\_dist\_name*} in the model config.

#### Parameters

- **model\_name** (*str*) – Name to register the action distribution under.
- **model\_class** (*type*) – Python class of the action distribution.

```
static get_model(input_dict, obs_space, action_space, num_outputs, options, state_in=None,
                 seq_lens=None)
```

Deprecated: use `get_model_v2()` instead.

```
class ray.rllib.models.Model(input_dict, obs_space, action_space, num_outputs, options,
                             state_in=None, seq_lens=None)
```

This class is deprecated, please use `TFModelV2` instead.

```
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.

```
custom_loss(policy_loss, loss_inputs)
```

Override to customize the loss function used to optimize this model.

This can be used to incorporate self-supervised losses (by defining a loss over existing input and output tensors of this model), and supervised losses (by defining losses over a variable-sharing copy of this model's layers).

You can find an runnable example in `examples/custom_loss.py`.

#### Parameters

- **policy\_loss** (*Tensor*) – scalar policy loss from the policy.
- **loss\_inputs** (*dict*) – map of input placeholders for rollout data.

**Returns** Scalar tensor for the customized loss for this model.

```
custom_stats()
```

Override to return custom metrics from your model.

**The stats will be reported as part of the learner stats, i.e.,**

**info:**

**learner:**

**model:** key1: metric1 key2: metric2

**Returns** Dict of string keys to scalar tensors.

```
loss()
```

Deprecated: use `self.custom_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.

```
write(observation, array, offset)
```

Alternative to transform for more efficient flattening.

**check\_shape** (*observation*)

Checks the shape of the given observation.

```
class ray.rllib.models.FullyConnectedNetwork(input_dict, obs_space, action_space,  
                                         num_outputs, options, state_in=None,  
                                         seq_lens=None)
```

Generic fully connected network.

```
class ray.rllib.models.VisionNetwork(input_dict, obs_space, action_space, num_outputs, options,  
                                     state_in=None, seq_lens=None)
```

Generic vision network.

## 5.24.5 ray.rllib.optimizers

```
class ray.rllib.optimizers.PolicyOptimizer(workers)
```

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.

**config**

The JSON configuration passed to this optimizer.

**Type** dict

**workers**

The set of rollout workers to use.

**Type** WorkerSet

**num\_steps\_trained**

Number of timesteps trained on so far.

**Type** int

**num\_steps\_sampled**

Number of timesteps sampled so far.

**Type** int

**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_workers=None)
```

Returns worker and optimizer stats.

#### Parameters

- **timeout\_seconds** (*int*) – Max wait time for a worker before dropping its results. This usually indicates a hung worker.
- **min\_history** (*int*) – Min history length to smooth results over.
- **selected\_workers** (*list*) – Override the list of remote workers to collect metrics from.

#### Returns

A training result dict from worker metrics with *info* replaced with stats from self.

**Return type** res (dict)

```
reset(remote_workers)
```

Called to change the set of remote workers being used.

```
foreach_worker(func)
```

Apply the given function to each worker instance.

```
foreach_worker_with_index(func)
```

Apply the given function to each worker instance.

The index will be passed as the second arg to the given function.

```
class ray.rllib.optimizers.AsyncReplayOptimizer(workers, learning_starts=1000,
                                                buffer_size=10000, prioritized_replay=True, prioritized_replay_alpha=0.6, prioritized_replay_beta=0.4,
                                                prioritized_replay_eps=1e-06, train_batch_size=512, rollout_fragment_length=50,
                                                num_replay_buffer_shards=1, max_weight_sync_delay=400, debug=False, batch_replay=False)
```

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

This class coordinates the data transfers between the learner thread, remote workers (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 rollout workers 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.

**reset(*remote\_workers*)**

Called to change the set of remote workers being used.

**stats()**

Returns a dictionary of internal performance statistics.

```
class ray.rllib.optimizers.AsyncSamplesOptimizer(workers, train_batch_size=500,  
                                              rollout_fragment_length=50,  
                                              num_envs_per_worker=1,  
                                              num_gpus=0, lr=0.0005, re-  
                                              play_buffer_num_slots=0,  
                                              replay_proportion=0.0,  
                                              num_data_loader_buffers=1,  
                                              max_sample_requests_in_flight_per_worker=2,  
                                              broadcast_interval=1,  
                                              num_sgd_iter=1, mini-  
                                              batch_buffer_size=1,  
                                              learner_queue_size=16,  
                                              learner_queue_timeout=300,  
                                              num_aggregation_workers=0,  
                                              fake_gpus=False)
```

Main event loop of the IMPALA architecture.

This class coordinates the data transfers between the learner thread and remote workers (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.

**reset(*remote\_workers*)**

Called to change the set of remote workers being used.

**stats()**

Returns a dictionary of internal performance statistics.

```
class ray.rllib.optimizers.AsyncGradientsOptimizer(workers, grads_per_step=100)
```

An asynchronous RL optimizer, e.g. for implementing A3C.

This optimizer asynchronously pulls and applies gradients from remote workers, 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(workers, num_sgd_iter=1,
                                                train_batch_size=1,
                                                sgd_minibatch_size=0, standard-
                                                size_fields=frozenset({}))
```

A simple synchronous RL optimizer.

In each step, this optimizer pulls samples from a number of remote workers, concatenates them, and then updates a local model. The updated model weights are then broadcast to all 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.SyncReplayOptimizer(workers, learning_starts=1000,
                                                buffer_size=10000, prioritized_replay=True,
                                                prioritized_replay_alpha=0.6, prioritized_replay_beta=0.4,
                                                prioritized_replay_eps=1e-06, final_prioritized_replay_beta=0.4,
                                                train_batch_size=32, before_learn_on_batch=None, synchronize_sampling=False,
                                                prioritized_replay_beta_annealing_timesteps=20000.0)
```

Variant of the local sync optimizer that supports replay (for DQN).

This optimizer requires that rollout workers 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.SyncBatchReplayOptimizer(workers, learning_starts=1000,
                                                    buffer_size=10000,
                                                    train_batch_size=32)
```

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.

```
class ray.rllib.optimizers.MicrobatchOptimizer(workers, train_batch_size=10000, microbatch_size=1000)
```

A microbatching synchronous RL optimizer.

This optimizer pulls sample batches from workers until the target microbatch size is reached. Then, it computes and accumulates the policy gradient in a local buffer. This process is repeated until the number of samples collected equals the train batch size. Then, an accumulated gradient update is made.

This allows for training with effective batch sizes much larger than can fit in GPU or host memory.

**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(workers, sgd_batch_size=128, num_sgd_iter=10, roll-out_fragment_length=200, num_envs_per_worker=1, train_batch_size=1024, num_gpus=0, standardize_fields=[], stand-shuffle_sequences=True, fake_gpus=False)
```

A synchronous optimizer that uses multiple local GPUs.

Samples are pulled synchronously from multiple remote workers, 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 requires the underlying Policy to be a TFPolicy instance that implements the `copy()` method for multi-GPU tower generation.

Note that all replicas of the TFPolicy 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.TorchDistributedDataParallelOptimizer(workers, ex-
    pected_batch_size,
    num_sgd_iter=1,
    sgd_minibatch_size=0,
    standard-
    ize_fields=frozenset({}),
    keep_local_weights_in_sync=True,
    back-
    end='gloo')
```

EXPERIMENTAL: torch distributed multi-node SGD.

**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.

## 5.24.6 ray.rllib.utils

**ray.rllib.utils.override(*cls*)**

Annotation for documenting method overrides.

**Parameters** **cls** (*type*) – The superclass that provides the overriden method. If this cls does not actually have the method, an error is raised.

**ray.rllib.utils.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 trainer classes are in public API because Trainer is @PublicAPI).

In addition, you can assume all trainer configurations are part of their public API as well.

**ray.rllib.utils.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).

**ray.rllib.utils.try\_import\_tf(*error=False*)**

**Parameters** **error** (*bool*) – Whether to raise an error if tf cannot be imported.

**Returns** The tf module (either from tf2.0.compat.v1 OR as tf1.x).

---

```
ray.rllib.utils.try_import_tfp(error=False)

Parameters error (bool) – Whether to raise an error if tfp cannot be imported.

Returns The tfp module.

ray.rllib.utils.try_import_torch(error=False)

Parameters error (bool) – Whether to raise an error if torch cannot be imported.

Returns torch AND torch.nn modules.

Return type tuple

ray.rllib.utils.check_framework(framework='tf')
Checks, whether the given framework is “valid”, meaning, whether all necessary dependencies are installed. Errors otherwise.

Parameters framework (str) – Once of “tf”, “torch”, or None.

Returns The input framework string.

Return type str

ray.rllib.utils.deprecation_warning(old, new=None, error=None)
Logs (via the logger object) or throws a deprecation warning/error.

Parameters

- old (str) – A description of the “thing” that is to be deprecated.
- new (Optional[str]) – A description of the new “thing” that replaces it.
- error (Optional[bool, Exception]) – Whether or which exception to throw. If True, throw ValueError.



ray.rllib.utils.renamed_agent(cls)
Helper class for renaming Agent => Trainer with a warning.

ray.rllib.utils.renamed_class(cls, old_name)
Helper class for renaming classes with a warning.

ray.rllib.utils.renamed_function(func, old_name)
Helper function for renaming a function.

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.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

**ray.rllib.utils.sigmoid(*x*, derivative=False)**

Returns the sigmoid function applied to x. Alternatively, can return the derivative or the sigmoid function.

**Parameters**

- **x** (*np.ndarray*) – The input to the sigmoid function.
- **derivative** (*bool*) – Whether to return the derivative or not. Default: False.

**Returns** The sigmoid function (or its derivative) applied to x.

**Return type** *np.ndarray*

**ray.rllib.utils.softmax(*x*, axis=-1)**

Returns the softmax values for x as:  $S(x_i) = e^{x_i} / \sum_j(e^{x_j})$ , where j goes over all elements in x.

**Parameters**

- **x** (*np.ndarray*) – The input to the softmax function.
- **axis** (*int*) – The axis along which to softmax.

**Returns** The softmax over x.

**Return type** *np.ndarray*

**ray.rllib.utils.relu(*x*, alpha=0.0)**

Implementation of the leaky ReLU function:  $y = x * \alpha$  if  $x < 0$  else  $x$

**Parameters**

- **x** (*np.ndarray*) – The input values.
- **alpha** (*float*) – A scaling (“leak”) factor to use for negative x.

**Returns** The leaky ReLU output for x.

**Return type** *np.ndarray*

**ray.rllib.utils.one\_hot(*x*, depth=0, on\_value=1, off\_value=0)**

One-hot utility function for numpy. Thanks to qianyizhang: <https://gist.github.com/qianyizhang/07ee1c15cad08afb03f5de69349efc30>.

**Parameters**

- **x** (*np.ndarray*) – The input to be one-hot encoded.
- **depth** (*int*) – The max. number to be one-hot encoded (size of last rank).
- **on\_value** (*float*) – The value to use for on. Default: 1.0.
- **off\_value** (*float*) – The value to use for off. Default: 0.0.

**Returns** The one-hot encoded equivalent of the input array.

**Return type** *np.ndarray*

---

```
ray.rllib.utils.fc(x, weights, biases=None, framework=None)
```

Calculates the outputs of a fully-connected (dense) layer given weights/biases and an input.

#### Parameters

- **x** (*np.ndarray*) – The input to the dense layer.
- **weights** (*np.ndarray*) – The weights matrix.
- **biases** (*Optional[np.ndarray]*) – The biases vector. All 0s if None.
- **framework** (*Optional[str]*) – An optional framework hint (to figure out, e.g. whether to transpose torch weight matrices).

#### Returns

The dense layer's output.

```
ray.rllib.utils.lstm(x, weights, biases=None, initial_internal_states=None, time_major=False, for-
                     get_bias=1.0)
```

Calculates the outputs of an LSTM layer given weights/biases, internal\_states, and input.

#### Parameters

- **x** (*np.ndarray*) – The inputs to the LSTM layer including time-rank (0th if time-major, else 1st) and the batch-rank (1st if time-major, else 0th).
- **weights** (*np.ndarray*) – The weights matrix.
- **biases** (*Optional[np.ndarray]*) – The biases vector. All 0s if None.
- **initial\_internal\_states** (*Optional[np.ndarray]*) – The initial internal states to pass into the layer. All 0s if None.
- **time\_major** (*bool*) – Whether to use time-major or not. Default: False.
- **forget\_bias** (*float*) – Gets added to first sigmoid (forget gate) output. Default: 1.0.

#### Returns

- The LSTM layer's output.
- Tuple: Last (c-state, h-state).

#### Return type

Tuple

```
class ray.rllib.utils.PolicyClient(address)
```

DEPRECATED: Please use rllib.env.PolicyClient instead.

```
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)

DEPRECATED: Please use rllib.env.PolicyServerInput instead.

**class** ray.rllib.utils.LinearSchedule (\*\*kwargs)

Linear interpolation between initial\_p and final\_p. Simply uses Polynomial with power=1.0.

final\_p + (initial\_p - final\_p) \* (1 - t/t\_max)

**class** ray.rllib.utils.PiecewiseSchedule (endpoints, framework, interpolation=<function linear\_interpolation>, outside\_value=None)

**class** ray.rllib.utils.PolynomialSchedule (schedule\_timesteps, final\_p, framework, initial\_p=1.0, power=2.0)

**class** ray.rllib.utils.ExponentialSchedule (schedule\_timesteps, framework, initial\_p=1.0, decay\_rate=0.1)

**class** ray.rllib.utils.ConstantSchedule (value, framework)

A Schedule where the value remains constant over time.

**ray.rllib.utils.check** (x, y, decimals=5, atol=None, rtol=None, false=False)

Checks two structures (dict, tuple, list, np.array, float, int, etc..) for (almost) numeric identity. All numbers in the two structures have to match up to decimal digits after the floating point. Uses assertions.

**Parameters**

- **x** (any) – The value to be compared (to the expectation: y). This may be a Tensor.
- **y** (any) – The expected value to be compared to x. This must not be a Tensor.
- **decimals** (int) – The number of digits after the floating point up to which all numeric values have to match.

- **atol** (*float*) – Absolute tolerance of the difference between x and y (overrides *decimals* if given).
- **rtol** (*float*) – Relative tolerance of the difference between x and y (overrides *decimals* if given).
- **false** (*bool*) – Whether to check that x and y are NOT the same.

```
ray.rllib.utils.framework_iterator(config=None, frameworks='tf', 'eager', 'torch', session=False)
```

An generator that allows for looping through n frameworks for testing.

Provides the correct config entries (“use\_pytorch” and “eager”) as well as the correct eager/non-eager contexts for tf.

#### Parameters

- **config** (*Optional[dict]*) – An optional config dict to alter in place depending on the iteration.
- **frameworks** (*Tuple[str]*) – A list/tuple of the frameworks to be tested. Allowed are: “tf”, “eager”, and “torch”.
- **session** (*bool*) – If True, enter a tf.Session() and yield that as well in the tf-case (otherwise, yield (fw, None)).

#### Yields *str* –

**If enter\_session is False:** The current framework (“tf”, “eager”, “torch”) used.

**Tuple(str, Union[None, tf.Session]): If enter\_session is True:** A tuple of the current fw and the tf.Session if fw=“tf”.

```
ray.rllib.utils.merge_dicts(d1, d2)
```

#### Parameters

- **d1** (*dict*) – Dict 1.
- **d2** (*dict*) – Dict 2.

**Returns** A new dict that is d1 and d2 deep merged.

#### Return type *dict*

```
ray.rllib.utils.deep_update(original, new_dict, new_keys_allowed=False, whitelist=None, override_all_if_type_changes=None)
```

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** (*Optional[List[str]]*) – List of keys that correspond to dict values where new subkeys can be introduced. This is only at the top level.
- **override\_all\_if\_type\_changes** (*Optional[List[str]]*) – List of top level keys with value=dict, for which we always simply override the entire value (dict), iff the “type” key in that value dict changes.

`ray.rllib.utils.add_mixins(base, mixins)`

Returns a new class with mixins applied in priority order.

`ray.rllib.utils.force_list(elements=None, to_tuple=False)`

Makes sure `elements` is returned as a list, whether `elements` is a single item, already a list, or a tuple.

#### Parameters

- `elements` (*Optional[any]*) – The inputs as single item, list, or tuple to be converted into a list/tuple. If None, returns empty list/tuple.
- `to_tuple` (`bool`) – Whether to use tuple (instead of list).

#### Returns

All given elements in a list/tuple depending on `to_tuple`'s value. If elements is None, returns an empty list/tuple.

#### Return type

 Union[list,tuple]

`ray.rllib.utils.force_tuple(elements=None, *, to_tuple=True)`

Makes sure `elements` is returned as a list, whether `elements` is a single item, already a list, or a tuple.

#### Parameters

- `elements` (*Optional[any]*) – The inputs as single item, list, or tuple to be converted into a list/tuple. If None, returns empty list/tuple.
- `to_tuple` (`bool`) – Whether to use tuple (instead of list).

#### Returns

All given elements in a list/tuple depending on `to_tuple`'s value. If elements is None, returns an empty list/tuple.

#### Return type

 Union[list,tuple]

## 5.25 Contributing to RLlib

### 5.25.1 Development Install

You can develop RLlib locally without needing to compile Ray by using the `setup-dev.py` script. This sets up links between the `rllib` dir in your git repo and the one bundled with the `ray` package. However if you have installed ray from source using [these instructions](<https://docs.ray.io/en/latest/installation.html>) then do not this as these steps should have already created this symlink. 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.)

### 5.25.2 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 trainer classes are in public API because Trainer is `@PublicAPI`).

In addition, you can assume all trainer 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`).

### 5.25.3 Features

Feature development, discussion, and upcoming priorities are tracked on the GitHub issues page (note that this may not include all development efforts).

### 5.25.4 Benchmarks

A number of training run results are available in the [rl-experiments repo](#), and there is also a list of working hyperparameter 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.

### 5.25.5 Contributing Algorithms

These are the guidelines for merging new algorithms into RLlib:

- **Contributed algorithms (`rllib/contrib`):**
  - must subclass Trainer 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 Trainer 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 `Trainer` and implement the `_init` and `_train` methods:

```
class RandomAgent(Trainer):
    """Policy that takes random actions and never learns."""

    _name = "RandomAgent"
    _default_config = with_common_config({
        "rollouts_per_iteration": 10,
    })

    @override(Trainer)
    def __init__(self, config, env_creator):
        self.env = env_creator(config["env_config"])

    @override(Trainer)
    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 {
            "episode_reward_mean": np.mean(rewards),
            "timesteps_this_iter": steps,
        }
```

Second, register the trainer 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 training progress using `rllib train`:

```
rllib train --run=contrib/RandomAgent --env=CartPole-v0
tensorboard --logdir=~/ray_results
```

## 5.26 RaySGD: Distributed Training Wrappers

RaySGD is a lightweight library for distributed deep learning, providing thin wrappers around PyTorch and TensorFlow native modules for data parallel training.

The main features are:

- **Ease of use:** Scale PyTorch’s native `DistributedDataParallel` and TensorFlow’s `tf.distribute.MirroredStrategy` without needing to monitor individual nodes.
- **Composability:** RaySGD is built on top of the Ray Actor API, enabling seamless integration with existing Ray applications such as RLlib, Tune, and Ray.Serve.
- **Scale up and down:** Start on single CPU. Scale up to multi-node, multi-CPU, or multi-GPU clusters by changing 2 lines of code.

**Important:** Join our [community slack](#) to discuss Ray!

### 5.26.1 Getting Started

You can start a `TorchTrainer` with the following:

```
import ray
from ray.util.sgd import TorchTrainer
from ray.util.sgd.torch.examples.train_example import LinearDataset

import torch
from torch.utils.data import DataLoader

def model_creator(config):
    return torch.nn.Linear(1, 1)

def optimizer_creator(model, config):
    """Returns optimizer."""
    return torch.optim.SGD(model.parameters(), lr=1e-2)

def data_creator(config):
    train_loader = DataLoader(LinearDataset(2, 5), config["batch_size"])
    val_loader = DataLoader(LinearDataset(2, 5), config["batch_size"])
    return train_loader, val_loader

ray.init()

trainer1 = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=torch.nn.MSELoss,
    num_workers=2,
    use_gpu=False,
    config={"batch_size": 64})
```

(continues on next page)

(continued from previous page)

```
stats = trainer1.train()
print(stats)
trainer1.shutdown()
print("success!")
```

---

**Tip:** Get in touch with us if you're using or considering using [RaySGD!](#)

---

## 5.27 Distributed PyTorch

The RaySGD TorchTrainer simplifies distributed model training for PyTorch.

---

**Tip:** Get in touch with us if you're using or considering using [RaySGD!](#)

---

The TorchTrainer is a wrapper around `torch.distributed.launch` with a Python API to easily incorporate distributed training into a larger Python application, as opposed to needing to wrap your training code in bash scripts.

For end to end examples leveraging RaySGD TorchTrainer, jump to [TorchTrainer Examples](#).

- *Setting up training*
  - *Model Creator*
  - *Optimizer Creator*
  - *Data Creator*
  - *Loss Creator*
  - *Scheduler Creator*
  - *Putting things together*
  - *Custom Training and Validation (Operators)*
  - *Custom DistributedDataParallel Wrappers*
- *Initialization Functions*
- *Save and Load*
- *Retrieving the model*
- *Mixed Precision (FP16) Training*
- *Distributed Multi-node Training*
- *Advanced: Fault Tolerance*
- *Advanced: Hyperparameter Tuning*
- *Simultaneous Multi-model Training*
- *Benchmarks*

- Debugging/Tips
- Feature Requests
- TorchTrainer Examples

### 5.27.1 Setting up training

The TorchTrainer can be constructed with functions that wrap components of the training script. Specifically, it requires constructors for the Model, Data, Optimizer, Loss, and lr\_scheduler to create replicated copies across different devices and machines.

Under the hood, TorchTrainer will create *replicas* of your model (controlled by num\_workers), each of which is managed by a Ray actor. One of the replicas will be on the main process, which can simplify the debugging and logging experience.

```
from ray.util.sgd import TorchTrainer

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    scheduler_creator=scheduler_creator,
    scheduler_step_freq="epoch", # if scheduler_creator is set
    config={"lr": 0.001, "batch_size": 64})
```

The below section covers the expected signatures of creator functions. Jump to [Putting things together](#).

#### Model Creator

This is the signature needed for TorchTrainer (model\_creator=...).

```
import torch.nn as nn

def model_creator(config):
    """Constructor function for the model(s) to be optimized.

    You will also need to provide a custom training
    function to specify the optimization procedure for multiple models.

    Args:
        config (dict): Configuration dictionary passed into ``TorchTrainer``.

    Returns:
        One or more torch.nn.Module objects.
    """
    return nn.Linear(1, 1)
```

## Optimizer Creator

This is the signature needed for `TorchTrainer(optimizer_creator=...)`.

```
import torch

def optimizer_creator(model, config):
    """Constructor of one or more Torch optimizers.

    Args:
        models: The return values from ``model_creator``. This can be one
            or more torch nn modules.
        config (dict): Configuration dictionary passed into ``TorchTrainer``.

    Returns:
        One or more Torch optimizer objects.
    """
    return torch.optim.SGD(model.parameters(), lr=config.get("lr", 1e-4))
```

## Data Creator

This is the signature needed for `TorchTrainer(data_creator=...)`.

```
from torch.utils.data import DataLoader
from ray.util.sgd.torch.examples.train_example import LinearDataset

def data_creator(config):
    """Constructs Iterables for training and validation.

    Note that even though two Iterable objects can be returned,
    only one Iterable will be used for training.

    Args:
        config: Configuration dictionary passed into ``TorchTrainer``

    Returns:
        One or Two Iterable objects. If only one Iterable object is provided,
        ``trainer.validate()`` will throw a ValueError.
    """
    train_dataset, val_dataset = LinearDataset(2, 5), LinearDataset(2, 5)
    train_loader = DataLoader(train_dataset, batch_size=config["batch_size"])
    val_loader = DataLoader(val_dataset, batch_size=config["batch_size"])
    return train_loader, val_loader
```

---

**Tip:** Setting the batch size: Using a provided `ray.util.sgd.utils.BATCH_SIZE` variable, you can provide a global batch size that will be divided among all workers automatically.

---

```
from torch.utils.data import DataLoader
from ray.util.sgd.utils import BATCH_SIZE

def data_creator(config):
    # config[BATCH_SIZE] == provided BATCH_SIZE // num_workers
    train_dataset, val_dataset = LinearDataset(2, 5), LinearDataset(2, 5)
    train_loader = DataLoader(train_dataset, batch_size=config[BATCH_SIZE])
```

(continues on next page)

(continued from previous page)

```

val_loader = DataLoader(val_dataset, batch_size=config[BATCH_SIZE])
return train_loader, val_loader

trainer = Trainer(
    model_creator=model_creator,
    optimizer_creator=optimizer_creator,
    data_creator=batch_data_creator
    config={BATCH_SIZE: 1024},
    num_workers=128
)

# Each worker will process 1024 // 128 samples per batch
stats = Trainer.train()

```

## Loss Creator

This is the signature needed for `TorchTrainer(loss_creator=...)`.

```

import torch

def loss_creator(config):
    """Constructs the Torch Loss object.

    Note that optionally, you can pass in a Torch Loss constructor directly
    into the TorchTrainer (i.e., ``TorchTrainer(loss_creator=nn.BCELoss, ...)``).

    Args:
        config: Configuration dictionary passed into ``TorchTrainer``

    Returns:
        Torch Loss object.
    """
    return torch.nn.BCELoss()

```

## Scheduler Creator

Optionally, you can provide a creator function for the learning rate scheduler. This is the signature needed for `TorchTrainer(scheduler_creator=...)`.

```

import torch

def scheduler_creator(optimizer, config):
    """Constructor of one or more Torch optimizer schedulers.

    Args:
        optimizers: The return values from ``optimizer_creator``.
            This can be one or more torch optimizer objects.
        config: Configuration dictionary passed into ``TorchTrainer``

    Returns:
        One or more Torch scheduler objects.
    """
    return torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.9)

```

## Putting things together

Before instantiating the trainer, first start or connect to a Ray cluster:

```
import ray

ray.init()
# or ray.init(address="auto") to connect to a running cluster.
```

Instantiate the trainer object:

```
from ray.util.sgd import TorchTrainer

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    scheduler_creator=scheduler_creator,
    scheduler_step_freq="epoch", # if scheduler_creator is set
    config={"lr": 0.001, "batch_size": 64})
```

You can also set the number of workers and whether the workers will use GPUs:

```
trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    scheduler_creator=scheduler_creator,
    config={"lr": 0.001},
    num_workers=100,
    use_gpu=True)
```

Now that the trainer is constructed, here's how to train the model.

```
for i in range(10):
    metrics = trainer.train()
    val_metrics = trainer.validate()
```

Each `train` call makes one pass over the training data, and each `validate` call runs the model on the validation data passed in by the `data_creator`.

You can also obtain profiling information:

```
>>> from ray.tune.logger import pretty_print
>>> print(pretty_print(trainer.train(profile=True)))

batch_count: 16
epoch: 1
last_train_loss: 0.15574650466442108
mean_train_loss: 7.475177114367485
num_samples: 1000
profile:
    mean_apply_s: 2.639293670654297e-05
    mean_fwd_s: 0.00012960433959960938
```

(continues on next page)

(continued from previous page)

```
mean_grad_s: 0.00016553401947021483
train_epoch_s: 0.023712158203125
```

Provide a custom training operator ([Custom Training and Validation \(Operators\)](#)) to calculate custom metrics or customize the training/validation process.

After training, you may want to reappropriate the Ray cluster. To release Ray resources obtained by the Trainer:

```
trainer.shutdown()
```

---

**Note:** Be sure to call `trainer.save()` or `trainer.get_model()` before shutting down.

---

See the documentation on the TorchTrainer here: [TorchTrainer](#).

## Custom Training and Validation (Operators)

TorchTrainer allows you to run a custom training and validation loops in parallel on each worker, providing a flexible interface similar to using PyTorch natively. This is done via the [PyTorch TrainingOperator](#) interface.

For both training and validation, there are two granularities that you can provide customization - per epoch and per batch. These correspond to `train_batch`, `train_epoch`, `validate`, and `validate_batch`. Other useful methods to override include `setup`, `save` and `restore`. You can use these to manage state (like a classifier neural network for calculating inception score, or a heavy tokenizer).

Providing a custom operator is necessary if creator functions return multiple models, optimizers, or schedulers.

Below is a partial example of a custom `TrainingOperator` that provides a `train_batch` implementation for a Deep Convolutional GAN.

```
import torch
from ray.util.sgd.torch import TrainingOperator

class GANOperator(TrainingOperator):
    def setup(self, config):
        """Custom setup for this operator.

        Args:
            config (dict): Custom configuration value to be passed to
                all creator and operator constructors. Same as ``self.config``.
        """
        pass

    def train_batch(self, batch, batch_info):
        """Trains on one batch of data from the data creator.

        Example taken from:
            https://github.com/eriklindernoren/PyTorch-GAN/blob/
            a163b82beff3d01688d8315a3fd39080400e7c01/implementations/dcgan/dcgan.py

        Args:
            batch: One item of the validation iterator.
            batch_info (dict): Information dict passed in from ``train_epoch``.

        Returns:
            ...
        """

    def validate(self, batch, batch_info):
        """Validates on one batch of data from the data creator.

        Args:
            batch: One item of the validation iterator.
            batch_info (dict): Information dict passed in from ``train_epoch``.

        Returns:
            ...
        """

    def validate_batch(self, batch, batch_info):
        """Validates on one batch of data from the data creator.

        Args:
            batch: One item of the validation iterator.
            batch_info (dict): Information dict passed in from ``train_epoch``.

        Returns:
            ...
        """

    def save(self, path):
        """Saves the current state of the operator.

        Args:
            path (str): Path to save the state to.
        """

    def restore(self, path):
        """Restores the state of the operator from a saved path.

        Args:
            path (str): Path to restore the state from.
        """
```

(continues on next page)

(continued from previous page)

```

A dict of metrics. Defaults to "loss" and "num_samples",
corresponding to the total number of datapoints in the batch.
"""
Tensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor
discriminator, generator = self.models
optimizer_D, optimizer_G = self.optimizers

# Adversarial ground truths
valid = Variable(Tensor(batch.shape[0], 1).fill_(1.0), requires_grad=False)
fake = Variable(Tensor(batch.shape[0], 1).fill_(0.0), requires_grad=False)

# Configure input
real_imgs = Variable(batch.type(Tensor))

# -----
# Train Generator
# -----

optimizer_G.zero_grad()

# Sample noise as generator input
z = Variable(Tensor(np.random.normal(0, 1, (
    batch.shape[0], self.config["latent_dim"]))))

# Generate a batch of images
gen_imgs = generator(z)

# Loss measures generator's ability to fool the discriminator
g_loss = adversarial_loss(discriminator(gen_imgs), valid)

g_loss.backward()
optimizer_G.step()

# -----
# Train Discriminator
# -----

optimizer_D.zero_grad()

# Measure discriminator's ability to classify real from generated samples
real_loss = adversarial_loss(discriminator(real_imgs), valid)
fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
d_loss = (real_loss + fake_loss) / 2

d_loss.backward()
optimizer_D.step()

return {
    "loss_g": g_loss.item(),
    "loss_d": d_loss.item(),
    "num_samples": imgs.shape[0]
}

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,

```

(continues on next page)

(continued from previous page)

```

        loss_creator=nn.BCELoss,
        training_operator_cls=GANOperator,
        num_workers=num_workers,
        config=config,
        use_gpu=True
    )

    for i in range(5):
        stats = trainer.train()
        print(stats)

```

See the [DCGAN example](#) for an end to end example. It constructs two models and two optimizers and uses a custom training operator to provide a non-standard training loop.

## Custom DistributedDataParallel Wrappers

TorchTrainer automatically applies a DistributedDataParallel wrapper to your model.

```
DistributedDataParallel(model, device_ids=self.device_ids)
```

By setting `TorchTrainer(wrap_ddp=False)` and providing your own custom training operator, you can change the parameters on the DistributedDataParallel wrapper or provide your own wrapper.

```

from ray.util.sgd.torch import TrainingOperator

class CustomOperator(TrainingOperator):
    def setup(self, config):
        self.new_model = CustomDataParallel(self.model,
                                            device_ids=self.device_ids)

    def train_batch(self, batch, batch_idx):
        output = self.new_model(batch)
        # calculate loss, etc
        return {"loss": loss}

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    training_operator_cls=CustomOperator,
    num_workers=2,
    use_gpu=True
    wrap_ddp=False,
)

```

## 5.27.2 Initialization Functions

Use the `initialization_hook` parameter to initialize state on each worker process when they are started. This is useful when setting an environment variable:

```
def initialization_hook():
    print("NCCL DEBUG SET")
    # Need this for avoiding a connection restart issue
    os.environ["NCCL_SOCKET_IFNAME"] = "^docker0,lo"
    os.environ["NCCL_LL_THRESHOLD"] = "0"
    os.environ["NCCL_DEBUG"] = "INFO"

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    initialization_hook=initialization_hook,
    config={"lr": 0.001},
    num_workers=100,
    use_gpu=True)
```

## 5.27.3 Save and Load

If you want to save or reload the training procedure, you can use `trainer.save` and `trainer.load`, which wraps the relevant `torch.save` and `torch.load` calls. This should work across a distributed cluster even without a NFS because it takes advantage of Ray's distributed object store.

```
checkpoint_path = os.path.join(tempfile.mkdtemp(), "checkpoint")
trainer_1.save(checkpoint_path)
# You can only have 1 trainer alive at a time
trainer_1.shutdown()

trainer_2 = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    num_workers=num_workers)
trainer_2.load(checkpoint_path)
```

## 5.27.4 Retrieving the model

The trained torch model can be extracted for use within the same Python program with `trainer.get_model()`. This will load the state dictionary of the model(s).

```
trainer.train()
model = trainer.get_model() # Returns multiple models if the model_creator does.
```

## 5.27.5 Mixed Precision (FP16) Training

You can enable mixed precision training for PyTorch with the `use_fp16` flag. This automatically converts the model(s) and optimizer(s) to train using mixed-precision. This requires NVIDIA Apex, which can be installed from the [NVIDIA/Apex repository](#):

```
trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    num_workers=4,
    use_fp16=True
)
```

Apex is a Pytorch extension with NVIDIA-maintained utilities to streamline mixed precision and distributed training. When `use_fp16=True`, you should not manually cast your model or data to `.half()`. The flag informs the Trainer to call `amp.initialize` on the created models and optimizers and optimize using the scaled loss: `amp.scale_loss(loss, optimizer)`.

To specify particular parameters for `amp.initialize`, you can use the `apex_args` field for the `TorchTrainer` constructor. Valid arguments can be found on the [Apex documentation](#):

```
trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    num_workers=4,
    use_fp16=True,
    apex_args={
        opt_level="O3",
        num_losses=2,
        verbosity=0
    }
)
```

Note that if using a custom training operator ([Custom Training and Validation \(Operators\)](#)), you will need to manage loss scaling manually.

## 5.27.6 Distributed Multi-node Training

You can scale your training to multiple nodes without making any modifications to your training code.

To train across a cluster, first make sure that the Ray cluster is started. You can start a Ray cluster via the Ray cluster launcher or [manually](#).

Then, in your program, you'll need to connect to this cluster via `ray.init`:

```
ray.init(address="auto") # or a specific redis address of the form "ip-address:port"
```

After connecting, you can scale up the number of workers seamlessly across multiple nodes:

```
trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
```

(continues on next page)

(continued from previous page)

```

        loss_creator=nn.MSELoss,
        num_workers=100
    )
trainer.train()
model = trainer.get_model()

```

## 5.27.7 Advanced: Fault Tolerance

For distributed deep learning, jobs are often run on infrastructure where nodes can be pre-empted frequently (i.e., spot instances in the cloud). To overcome this, RaySGD provides **fault tolerance** features that enable training to continue regardless of node failures.

```
trainer.train(max_retries=N)
```

During each `train` method, each parallel worker iterates through the iterable, synchronizing gradients and parameters at each batch. These synchronization primitives can hang when one or more of the parallel workers becomes unresponsive (i.e., when a node is lost). To address this, we've implemented the following protocol.

1. If any worker node is lost, Ray will mark the training task as complete (`ray.wait` will return).
2. Ray will throw `RayActorException` when fetching the result for any worker, so the Trainer class will call `ray.get` on the “finished” training task.
3. Upon catching this exception, the Trainer class will kill all of its workers.
4. The Trainer will then detect the quantity of available resources (either CPUs or GPUs). It will then restart as many workers as it can, each resuming from the last checkpoint. Note that this may result in fewer workers than initially specified.
5. If there are no available resources, the Trainer will apply an exponential backoff before retrying to create workers.
6. If there are available resources and the Trainer has fewer workers than initially specified, then it will scale up its worker pool until it reaches the initially specified `num_workers`.

Note that we assume the Trainer itself is not on a pre-emptible node. To allow the entire Trainer to recover from failure, you must use Tune to execute the training.

## 5.27.8 Advanced: Hyperparameter Tuning

`TorchTrainer` naturally integrates with Tune via the `BaseTorchTrainable` interface. Without changing any arguments, you can call `TorchTrainer.as_trainable(model_creator...)` to create a Tune-compatible class. See the documentation ([BaseTorchTrainable](#)).

```

def tune_example(num_workers=1, use_gpu=False):
    TorchTrainable = TorchTrainer.as_trainable(
        model_creator=model_creator,
        data_creator=data_creator,
        optimizer_creator=optimizer_creator,
        loss_creator=nn.MSELoss, # Note that we specify a Loss class.
        num_workers=num_workers,
        use_gpu=use_gpu,
        config={BATCH_SIZE: 128}
    )

```

(continues on next page)

(continued from previous page)

```

analysis = tune.run(
    TorchTrainable,
    num_samples=3,
    config={"lr": tune.grid_search([1e-4, 1e-3])},
    stop={"training_iteration": 2},
    verbose=1)

return analysis.get_best_config(metric="validation_loss", mode="min")

```

You can see the [Tune example script](#) for an end-to-end example.

### 5.27.9 Simultaneous Multi-model Training

In certain scenarios, such as training GANs, you may want to use multiple models in the training loop. You can do this in the `TorchTrainer` by allowing the `model_creator`, `optimizer_creator`, and `scheduler_creator` to return multiple values. Provide a custom `TrainingOperator` ([Custom Training and Validation \(Operators\)](#)) to train across multiple models.

You can see the [DCGAN script](#) for an end-to-end example.

```

from ray.util.sgd.torch import TorchTrainer, TrainingOperator

def train(*, model=None, criterion=None, optimizer=None, dataloader=None):
    model.train()
    train_loss = 0
    correct = 0
    total = 0
    for batch_idx, (inputs, targets) in enumerate(dataloader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()
    return {
        "accuracy": correct / total,
        "train_loss": train_loss / (batch_idx + 1)
    }

def model_creator(config):
    return Discriminator(), Generator()

def optimizer_creator(models, config):
    net_d, net_g = models
    discriminator_opt = optim.Adam(
        net_d.parameters(), lr=config.get("lr", 0.01), betas=(0.5, 0.999))
    generator_opt = optim.Adam(
        net_g.parameters(), lr=config.get("lr", 0.01), betas=(0.5, 0.999))
    return discriminator_opt, generator_opt

class CustomOperator(TrainingOperator):

```

(continues on next page)

(continued from previous page)

```

def train_epoch(self, iterator, info):
    result = {}
    for i, (model, optimizer) in enumerate(
        zip(self.models, self.optimizers)):
        result["model_{}".format(i)] = train(
            model=model,
            criterion=self.criterion,
            optimizer=optimizer,
            dataloader=iterator)
    return result

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.BCELoss,
    training_operator_cls=CustomOperator)

stats = trainer.train()

```

## 5.27.10 Benchmarks

RaySGD TorchTrainer provides comparable or better performance than other existing solutions for parallel or distributed training.

### Multi-GPU (Single Node) benchmarks:

```

# Images per second for ResNet50
# Batch size per worker = 128
# GPU Type = V100
# Run on AWS us-east-1c, p3dn.24xlarge instance.

```

Number of GPUs	DataParallel	Ray (PyTorch)	DataParallel + Apex	Ray (PyTorch) + Apex
1	355.5	356	776	770
2	656	701	1303	1346
4	1289	1401	2606	2695
8	2521	2795	4795	5862

### Multi-node benchmarks:

```

# Images per second for ResNet50
# Batch size per worker = 128
# GPU Type = V100
# Run on AWS us-east-1c, p3dn.24xlarge instances.

```

Number of GPUs	Horovod	Ray (PyTorch)	Horovod + Apex	Ray (PyTorch) + Apex
1 * 8	2769.7	2962.7	5143	6172
2 * 8	5492.2	5886.1	9463	10052.8
4 * 8	10733.4	11705.9	18807	20319.5
8 * 8	21872.5	23317.9	36911.8	38642

You can see more details in the [benchmarking README](#).

**DISCLAIMER:** RaySGD does not provide any custom communication primitives. If you see any performance issues, you may need to file them on the PyTorch github repository.

### 5.27.11 Debugging/Tips

Here's some simple tips on how to debug the TorchTrainer.

#### My TorchTrainer implementation is erroring after I ported things over from my previous code.

Try using ipdb, a custom TrainingOperator, and num\_workers=1. This will provide you introspection what is being called and when.

```
# first run pip install ipdb

from ray.util.sgd.torch import TrainingOperator

class CustomOperator(TrainingOperator):
    def setup(self, config):
        import ipdb; ipdb.set_trace()
        ... # custom code if exists?

    def train_batch(self, batch, batch_idx):
        import ipdb; ipdb.set_trace()
        ... # press 'n' or 's' to navigate the session
        ... # custom code if exists?
        ... # or super(CustomOperator, self).train_batch(batch, batch_idx)

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=loss_creator,
    training_operator_cls=GANOperator,
    num_workers=1,
)
```

#### My TorchTrainer implementation is super slow.

Try using a profiler. Either use:

```
trainer.train(profile=True)
trainer.validate(profile=True)
```

or use [Python profiling](#).

#### My creator functions download data, and I don't want multiple processes downloading to the same path at once.

Use filelock within the creator functions to create locks for critical regions. For example:

```
import os
import tempfile
from filelock import FileLock

def create_dataset(config):
```

(continues on next page)

(continued from previous page)

```
dataset_path = config["dataset_path"]

# Create a critical region of the code
# This will take a longer amount of time to download the data at first.
# Other processes will block at the ``with`` statement.
# After downloading, this code block becomes very fast.
with FileLock(os.path.join(tempfile.gettempdir(), "download_data.lock")):
    if not os.path.exists(dataset_path):
        download_data(dataset_path)

# load_data is assumed to safely support concurrent reads.
data = load_data(dataset_path)
return DataLoader(data)
```

### I get a ‘socket timeout’ error during training.

Try increasing the length of the NCCL timeout. The current timeout is 10 seconds.

```
NCCL_TIMEOUT_S=1000 python ray_training_script.py

# or

NCCL_TIMEOUT_S=1000 ray start [--head | --address]
```

## 5.27.12 Feature Requests

Have features that you’d really like to see in RaySGD? Feel free to open an issue.

## 5.27.13 TorchTrainer Examples

Here are some examples of using RaySGD for training PyTorch models. If you’d like to contribute an example, feel free to create a pull request here.

- **Torch training example** Simple example of using Ray’s TorchTrainer.
- **TorchTrainer and RayTune example** Simple example of hyperparameter tuning with Ray’s TorchTrainer.
- **Semantic Segmentation example** Fine-tuning a ResNet50 model on VOC with Batch Norm.
- **Huggingface Transformer GLUE fine tuning example** Fine-tuning a pre-trained Transformer model on GLUE tasks. Based off of the `huggingface/transformers run_glue.py` example.
- **ImageNet Models example** Training state-of-the-art ImageNet models.
- **CIFAR10 example** Training a ResNet18 model on CIFAR10.
- **CIFAR10 RayTune example** Tuning a ResNet18 model on CIFAR10 with Population-based training on Ray-Tune.
- **DCGAN example** Training a Deep Convolutional GAN on MNIST. It constructs two models and two optimizers and uses a custom training operator.

## 5.28 Distributed TensorFlow

RaySGD's `TFTrainer` simplifies distributed model training for Tensorflow. The `TFTrainer` is a wrapper around `MultiWorkerMirroredStrategy` with a Python API to easily incorporate distributed training into a larger Python application, as opposed to write custom logic of setting environments and starting separate processes.

Under the hood, `TFTrainer` will create *replicas* of your model (controlled by `num_replicas`), each of which is managed by a Ray actor.

**Tip:** We need your feedback! RaySGD is currently early in its development, and we're hoping to get feedback from people using or considering it. We'd love [to get in touch!](#)

### With Ray:

Wrap your training with this:

```
ray.init(args.address)

trainer = TFTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    num_replicas=4,
    use_gpu=True,
    verbose=True,
    config={
        "fit_config": {
            "steps_per_epoch": num_train_steps,
        },
        "evaluate_config": {
            "steps": num_eval_steps,
        }
    })

```

Then, start a Ray cluster via autoscaler or manually.

```
ray up CLUSTER.yaml
python train.py --address="localhost:<PORT>"
```

### Before, with Tensorflow:

In your training program, insert the following, and **customize** for each worker:

```
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {
        'worker': ["localhost:12345", "localhost:23456"]
    },
    'task': {'type': 'worker', 'index': 0}
})

...
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
with strategy.scope():
    multi_worker_model = model_creator()
```

And on each machine, launch a separate process that contains the index of the worker and information about all other nodes of the cluster.

### 5.28.1 TFTrainer Example

Below is an example of using Ray's TFTrainer. Under the hood, TFTrainer will create *replicas* of your model (controlled by num\_replicas) which are each managed by a worker.

```
import argparse
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np

import ray
from ray import tune
from ray.util.sgd.tf.tf_trainer import TFTrainer, TFTrainable

NUM_TRAIN_SAMPLES = 1000
NUM_TEST_SAMPLES = 400

def create_config(batch_size):
    return {
        # todo: batch size needs to scale with # of workers
        "batch_size": batch_size,
        "fit_config": {
            "steps_per_epoch": NUM_TRAIN_SAMPLES // batch_size
        },
        "evaluate_config": {
            "steps": NUM_TEST_SAMPLES // batch_size,
        }
    }

def linear_dataset(a=2, size=1000):
    x = np.random.rand(size)
    y = x / 2

    x = x.reshape((-1, 1))
    y = y.reshape((-1, 1))

    return x, y

def simple_dataset(config):
    batch_size = config["batch_size"]
    x_train, y_train = linear_dataset(size=NUM_TRAIN_SAMPLES)
    x_test, y_test = linear_dataset(size=NUM_TEST_SAMPLES)

    train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
    test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
    train_dataset = train_dataset.shuffle(NUM_TRAIN_SAMPLES).repeat().batch(
        batch_size)
    test_dataset = test_dataset.repeat().batch(batch_size)

    return train_dataset, test_dataset
```

(continues on next page)

(continued from previous page)

```

def simple_model(config):
    model = Sequential([Dense(10, input_shape=(1,)), Dense(1)])
    model.compile(
        optimizer="sgd",
        loss="mean_squared_error",
        metrics=["mean_squared_error"])
    return model

def train_example(num_replicas=1, batch_size=128, use_gpu=False):
    trainer = TFTrainer(
        model_creator=simple_model,
        data_creator=simple_dataset,
        num_replicas=num_replicas,
        use_gpu=use_gpu,
        verbose=True,
        config=create_config(batch_size))

    # model baseline performance
    start_stats = trainer.validate()
    print(start_stats)

    # train for 2 epochs
    trainer.train()
    trainer.train()

    # model performance after training (should improve)
    end_stats = trainer.validate()
    print(end_stats)

    # sanity check that training worked
    dloss = end_stats["validation_loss"] - start_stats["validation_loss"]
    dmse = (end_stats["validation_mean_squared_error"] -
            start_stats["validation_mean_squared_error"])
    print(f"dLoss: {dloss}, dMSE: {dmse}")

    if dloss > 0 or dmse > 0:
        print("training sanity check failed. loss increased!")
    else:
        print("success!")

def tune_example(num_replicas=1, use_gpu=False):
    config = {
        "model_creator": tune.function(simple_model),
        "data_creator": tune.function(simple_dataset),
        "num_replicas": num_replicas,
        "use_gpu": use_gpu,
        "trainer_config": create_config(batch_size=128)}
    analysis = tune.run(
        TFTrainable,

```

(continues on next page)

(continued from previous page)

```

    num_samples=2,
    config=config,
    stop={"training_iteration": 2},
    verbose=1)

    return analysis.get_best_config(metric="validation_loss", mode="min")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--address",
        required=False,
        type=str,
        help="the address to use for Ray")
    parser.add_argument(
        "--num-replicas",
        "-n",
        type=int,
        default=1,
        help="Sets number of replicas for training.")
    parser.add_argument(
        "--use-gpu",
        action="store_true",
        default=False,
        help="Enables GPU training")
    parser.add_argument(
        "--tune", action="store_true", default=False, help="Tune training")

    args, _ = parser.parse_known_args()

    ray.init(address=args.address)

    if args.tune:
        tune_example(num_replicas=args.num_replicas, use_gpu=args.use_gpu)
    else:
        train_example(num_replicas=args.num_replicas, use_gpu=args.use_gpu)

```

## 5.29 RaySGD API Documentation

### 5.29.1 TorchTrainer

```

class ray.util.sgd.torch.TorchTrainer(*, model_creator, data_creator, optimizer_creator, loss_creator=None, scheduler_creator=None, training_operator_cls=None, initialization_hook=None, config=None, num_workers=1, use_gpu='auto', backend='auto', wrap_ddp=True, serialize_data_creation=True, use_fp16=False, use_tqdm=False, apex_args=None, add_dist_sampler=True, scheduler_step_freq=None, num_replicas=None, batch_size=None, data_loader_args=None)

```

Train a PyTorch model using distributed PyTorch.

Launches a set of actors which connect via distributed PyTorch and coordinate gradient updates to train the provided model. If Ray is not initialized, TorchTrainer will automatically initialize a local Ray cluster for you. Be sure to run `ray.init(address="auto")` to leverage multi-node training.

```
def model_creator(config):
    return nn.Linear(1, 1)

def optimizer_creator(model, config):
    return torch.optim.SGD(
        model.parameters(), lr=config.get("lr", 1e-4))

def data_creator(config):
    batch_size = config["batch_size"]
    train_data, val_data = LinearDataset(2, 5), LinearDataset(2, 5)
    train_loader = DataLoader(train_data, batch_size=batch_size)
    val_loader = DataLoader(val_data, batch_size=batch_size)
    return train_loader, val_loader

trainer = TorchTrainer(
    model_creator=model_creator,
    data_creator=data_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.MSELoss,
    config={"batch_size": 32},
    use_gpu=True
)
for i in range(4):
    trainer.train()
```

The creator functions will execute before distributed coordination and training is setup. This is so that creator functions that download large datasets will not trigger any timeouts.

The order of operations for creator functions are:

```
data_creator -> model_creator -> optimizer_creator -> scheduler_creator ->
loss_creator.
```

### Parameters

- **model\_creator** (`dict -> Model(s)`) – Constructor function that takes in config and returns the model(s) to be optimized. These must be `torch.nn.Module` objects. If multiple models are returned, a `training_operator_cls` must be specified. You do not need to handle GPU/devices in this function; RaySGD will do that under the hood.
- **data\_creator** (`dict -> Iterable(s)`) – Constructor function that takes in the passed config and returns one or two Iterable objects. Note that even though two Iterable objects can be returned, only one will be used for training, and the other will be used for validation. If not provided, you must provide a custom TrainingOperator.
- **optimizer\_creator** (`(models, dict) -> optimizers`) – Constructor function that takes in the return values from `model_creator` and the passed config and returns One or more Torch optimizer objects. You do not need to handle GPU/devices in this function; RaySGD will do that for you.
- **loss\_creator** (`torch.nn.*Loss class / dict -> loss`) – A constructor function for the training loss. This can be either a function that takes in the provided config for customization or a subclass of `torch.nn.modules.loss._Loss`, which is most

Pytorch loss classes. For example, `loss_creator=torch.nn.BCELoss`. If not provided, you must provide a custom TrainingOperator.

- **scheduler\_creator** (*optimizers, dict*) – A constructor function for the torch scheduler. This is a function that takes in the generated optimizers (from `optimizer_creator`) provided config for customization. Be sure to set `scheduler_step_freq` to increment the scheduler correctly.
- **training\_operator\_cls** (*type*) – Custom training operator class that subclasses the TrainingOperator class. This class will be copied onto all remote workers and used to specify custom training and validation operations. Defaults to `TrainingOperator`.
- **config** (*dict*) – Custom configuration value to be passed to all creator and operator constructors.
- **num\_workers** (*int*) – the number of workers used in distributed training. If 1, the worker will not be wrapped with `DistributedDataParallel`.
- **use\_gpu** (*bool*) – Sets resource allocation for workers to 1 GPU if true, and automatically moves both the model and optimizer to the available CUDA device.
- **backend** (*string*) – backend used by distributed PyTorch. Currently support “nccl”, “gloo”, and “auto”. If “auto”, RaySGD will automatically use “nccl” if `use_gpu` is True, and “gloo” otherwise.
- **serialize\_data\_creation** (*bool*) – A filelock will be used to ensure no race conditions in data downloading among different workers on the same node (using the local file system). Defaults to True.
- **wrap\_ddp** (*bool*) – Whether to automatically wrap `DistributedDataParallel` over each model. If False, you are expected to call it yourself.
- **add\_dist\_sampler** (*bool*) – Whether to automatically add a `DistributedSampler` to all created dataloaders. Only applicable if `num_workers > 1`.
- **use\_fp16** (*bool*) – Enables mixed precision training via apex if apex is installed. This is automatically done after the model and optimizers are constructed and will work for multi-model training. Please see <https://github.com/NVIDIA/apex> for more details.
- **apex\_args** (*dict / None*) – Dict containing keyword args for `amp.initialize`. See <https://nvidia.github.io/apex/amp.html#module-apex.amp>. By default, the models and optimizers are passed in. Consider using “`num_losses`” if operating over multiple models and optimizers.
- **scheduler\_step\_freq** – “batch”, “epoch”, “manual”, or None. This will determine when `scheduler.step` is called. If “batch”, step will be called after every optimizer step. If “epoch”, step will be called after one pass of the `DataLoader`. If “manual”, the scheduler will not be incremented automatically - you are expected to call `trainer.update_schedulers` manually. If a scheduler is passed in, this value is expected to not be None.

**train** (*num\_steps=None, profile=False, reduce\_results=True, max\_retries=3, info=None*)

Runs a training epoch.

Calls `operator.train_epoch()` on N parallel workers simultaneously underneath the hood.

Set `max_retries` to enable fault handling in case of instance preemption.

### Parameters

- **num\_steps** (*int*) – Number of batches to compute update steps on. This corresponds also to the number of times `TrainingOperator.train_batch` is called.

- **profile** (*bool*) – Returns time stats for the training procedure.
- **reduce\_results** (*bool*) – Whether to average all metrics across all workers into one dict. If a metric is a non-numerical value (or nested dictionaries), one value will be randomly selected among the workers. If False, returns a list of dicts.
- **max\_retries** (*int*) – Must be non-negative. If set to N, TorchTrainer will detect and recover from training failure. The recovery process will kill all current workers, query the Ray global state for total available resources, and re-launch up to the available resources. Behavior is not well-defined in case of shared cluster usage. Defaults to 3.
- **info** (*dict*) – Optional dictionary passed to the training operator for `train_epoch` and `train_batch`.

**Returns**

(*dict* | *list*) **A dictionary of metrics for training.** You can provide custom metrics by passing in a custom `training_operator_cls`. If `reduce_results=False`, this will return a list of metric dictionaries whose length will be equal to `num_workers`.

**apply\_all\_workers** (*fn*)

Run a function on all operators on the workers.

**Parameters** *fn* (*Callable*) – A function that takes in no arguments.

**Returns** A list of objects returned by *fn* on each worker.

**apply\_all\_operators** (*fn*)

Run a function on all operators on the workers.

**Parameters** *fn* (*Callable[TrainingOperator]*) – A function that takes in a `TrainingOperator`.

**Returns** A list of objects returned by *fn* on each operator.

**validate** (*num\_steps=None*, *profile=False*, *reduce\_results=True*, *info=None*)

Evaluates the model on the validation data set.

**Parameters**

- **num\_steps** (*int*) – Number of batches to compute update steps on. This corresponds also to the number of times `TrainingOperator.validate_batch` is called.
- **profile** (*bool*) – Returns time stats for the evaluation procedure.
- **reduce\_results** (*bool*) – Whether to average all metrics across all workers into one dict. If a metric is a non-numerical value (or nested dictionaries), one value will be randomly selected among the workers. If False, returns a list of dicts.
- **info** (*dict*) – Optional dictionary passed to the training operator for `validate` and `validate_batch`.

**Returns**

**A dictionary of metrics for validation.** You can provide custom metrics by passing in a custom `training_operator_cls`.

**update\_scheduler** (*metric*)

Calls `scheduler.step(metric)` on all schedulers.

This is useful for lr\_schedulers such as `ReduceLROnPlateau`.

**get\_model** ()

Returns the learned model(s).

**get\_local\_operator()**

Returns the local TrainingOperator object.

Be careful not to perturb its state, or else you can cause the system to enter an inconsistent state.

**Returns** The local TrainingOperator object.

**Return type** *TrainingOperator*

**save(*checkpoint*)**

Saves the Trainer state to the provided checkpoint path.

**Parameters** **checkpoint** (*str*) – Path to target checkpoint file.

**load(*checkpoint*)**

Loads the Trainer and all workers from the provided checkpoint.

**Parameters** **checkpoint** (*str*) – Path to target checkpoint file.

**shutdown(*force=False*)**

Shuts down workers and releases resources.

**classmethod as\_trainable(\*args, \*\*kwargs)**

Creates a BaseTorchTrainable class compatible with Tune.

Any configuration parameters will be overridden by the Tune Trial configuration. You can also subclass the provided Trainable to implement your own iterative optimization routine.

```
TorchTrainable = TorchTrainer.as_trainable(  
    model_creator=ResNet18,  
    data_creator=cifar_creator,  
    optimizer_creator=optimizer_creator,  
    loss_creator=nn.CrossEntropyLoss,  
    num_gpus=2  
)  
analysis = tune.run(  
    TorchTrainable,  
    config={"lr": tune.grid_search([0.01, 0.1])}  
)
```

## 5.29.2 PyTorch TrainingOperator

```
class ray.util.sgd.torch.TrainingOperator(config, models, optimizers, train_loader, validation_loader, world_rank, criterion=None, schedulers=None, device_ids=None, use_gpu=False, use_fp16=False, use_tqdm=False)
```

Abstract class for custom training or validation loops.

The scheduler will only be called at a batch or epoch frequency, depending on the user parameter. Be sure to set `scheduler_step_freq` in `TorchTrainer` to either “batch” or “epoch” to increment the scheduler correctly during training. If using a learning rate scheduler that depends on validation loss, you can use `trainer.update_scheduler`.

For both training and validation, there are two granularities that you can provide customization: per epoch or per batch. You do not need to override both.

**train\_epoch/validate: (1 epoch)**

- Validate every 100 batches
- Persist RNN state across batches
- Use a tokenizer for validation

**train\_batch/validate\_batch: (for each batch)**

- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>• Encode, decode</li> <li>• Custom metrics</li> <li>• Image Generation</li> </ul> | <ul style="list-style-type: none"> <li>• Loss Scaling</li> <li>• Gradient Clipping</li> <li>• LR scheduling</li> </ul> |
|--|--|

**Raises ValueError if multiple models/optimizers/schedulers are provided.** – You are expected to subclass this class if you wish to train over multiple models/optimizers/schedulers.

**setup(config)**

Override this method to implement custom operator setup.

**Parameters config (dict)** – Custom configuration value to be passed to all creator and operator constructors. Same as self.config.

**train\_epoch(iterator, info)**

Runs one standard training pass over the training dataloader.

By default, this method will iterate over the given iterator and call self.train\_batch over each batch. If scheduler\_step\_freq is set, this default method will also step the scheduler accordingly.

You do not need to call train\_batch in this method if you plan to implement a custom optimization/training routine here.

You may find ray.util.sgd.utils.AverageMeterCollection useful when overriding this method. See example below:

```
def train_epoch(self, ...):
    meter_collection = AverageMeterCollection()
    self.model.train()
    for batch in iterator:
        # do some processing
        metrics = {"metric_1": 1, "metric_2": 3} # dict of metrics

        # This keeps track of all metrics across multiple batches
        meter_collection.update(metrics, n=len(batch))

        # Returns stats of the meters.
        stats = meter_collection.summary()
    return stats
```

**Parameters**

- **iterator** (*iter*) – Iterator over the training data for the entire epoch. This iterator is expected to be entirely consumed.
- **info** (*dict*) – Dictionary for information to be used for custom training operations.

**Returns** A dict of metrics from training.

#### **train\_batch** (*batch, batch\_info*)

Computes loss and updates the model over one batch.

This method is responsible for computing the loss and gradient and updating the model.

By default, this method implementation assumes that batches are in (features, labels) format. If using amp/fp16 training, it will also scale the loss automatically.

You can provide custom loss metrics and training operations if you override this method. If overriding this method, you can access model, optimizer, criterion via `self.model`, `self.optimizer`, and `self.criterion`.

You do not need to override this method if you plan to override `train_epoch`.

#### **Parameters**

- **batch** – One item of the validation iterator.
- **batch\_info** (*dict*) – Information dict passed in from `train_epoch`.

#### **Returns**

**A dictionary of metrics.** By default, this dictionary contains “loss” and “num\_samples”.

“num\_samples” corresponds to number of datapoints in the batch. However, you can provide any number of other values. Consider returning “num\_samples” in the metrics because by default, `train_epoch` uses “num\_samples” to calculate averages.

#### **validate** (*val\_iterator, info*)

Runs one standard validation pass over the `val_iterator`.

This will call `model.eval()` and `torch.no_grad` when iterating over the validation dataloader.

If overriding this method, you can access model, criterion via `self.model` and `self.criterion`. You also do not need to call `validate_batch` if overriding this method.

#### **Parameters**

- **val\_iterator** (*iter*) – Iterable constructed from the validation dataloader.
- **info** – (dict): Dictionary for information to be used for custom validation operations.

#### **Returns**

**A dict of metrics from the evaluation.** By default, returns “val\_accuracy” and “val\_loss” which is computed by aggregating “loss” and “correct” values from `validate_batch` and dividing it by the sum of `num_samples` from all calls to `self.validate_batch`.

#### **validate\_batch** (*batch, batch\_info*)

Calculates the loss and accuracy over a given batch.

You can override this method to provide arbitrary metrics.

#### **Parameters**

- **batch** – One item of the validation iterator.
- **batch\_info** (*dict*) – Contains information per batch from `validate()`.

**Returns**

**A dict of metrics.** By default, returns “val\_loss”, “val\_accuracy”, and “num\_samples”. When overriding, consider returning “num\_samples” in the metrics because by default, validate uses “num\_samples” to calculate averages.

**state\_dict()**

Override this to return a representation of the operator state.

**Returns** The state dict of the operator.

**Return type** dict

**load\_state\_dict(state\_dict)**

Override this to load the representation of the operator state.

**Parameters** `state_dict (dict)` – State dict as returned by the operator.

**property device**

The appropriate torch device, at your convenience.

**Type** torch.device

**property config**

Provided into TorchTrainer.

**Type** dict

**property model**

First or only model created by the provided `model_creator`.

**property models**

List of models created by the provided `model_creator`.

**property optimizer**

First or only optimizer(s) created by the `optimizer_creator`.

**property optimizers**

List of optimizers created by the `optimizer_creator`.

**property train\_loader**

1st Dataloader from `data_creator`.

**Type** Iterable

**property validation\_loader**

2nd Dataloader from `data_creator`.

**Type** Iterable

**property world\_rank**

The rank of the parent runner. Always 0 if not distributed.

**Type** int

**property criterion**

Criterion created by the provided `loss_creator`.

**property scheduler**

First or only scheduler(s) created by the `scheduler_creator`.

**property schedulers**

List of schedulers created by the `scheduler_creator`.

**property use\_gpu**

Returns True if cuda is available and `use_gpu` is True.

**property use\_fp16**

Whether the model and optimizer have been FP16 enabled.

**Type** bool

**property use\_tqdm**

Whether tqdm progress bars are enabled.

**Type** bool

**property device\_ids**

Device IDs for the model.

This is useful for using batch norm with DistributedDataParallel.

**Type** List[int]

### 5.29.3 BaseTorchTrainable

```
class ray.util.sgd.torch.BaseTorchTrainable(config=None, logger_creator=None)
```

Base class for converting TorchTrainer to a Trainable class.

This class is produced when you call `TorchTrainer.as_trainable(...)`.

You can override the produced Trainable to implement custom iterative training procedures:

```
TorchTrainable = TorchTrainer.as_trainable(
    model_creator=ResNet18,
    data_creator=cifar_creator,
    optimizer_creator=optimizer_creator,
    loss_creator=nn.CrossEntropyLoss,
    num_gpus=2
)
# TorchTrainable is subclass of BaseTorchTrainable.

class CustomTrainable(TorchTrainable):
    def _train(self):
        for i in range(5):
            train_stats = self.trainer.train()
            validation_stats = self.trainer.validate()
            train_stats.update(validation_stats)
            return train_stats

analysis = tune.run(
    CustomTrainable,
    config={"lr": tune.grid_search([0.01, 0.1])}
)
```

**\_setup(config)**

Constructs a TorchTrainer object as `self.trainer`.

**\_train()**

Calls `self.trainer.train()` and `self.trainer.validate()` once.

You may want to override this if using a custom LR scheduler.

**\_save(checkpoint\_dir)**

Returns a path containing the trainer state.

**\_restore(checkpoint\_path)**

Restores the trainer state.

Override this if you have state external to the Trainer object.

**\_stop()**  
Shuts down the trainer.

**property trainer**  
An instantiated TorchTrainer object.

Use this when specifying custom training procedures for Tune.

## 5.29.4 TFTrainer

```
class ray.util.sgd.tf.TFTrainer(model_creator, data_creator, config=None, num_replicas=1,
                                use_gpu=False, verbose=False)
```

**\_\_init\_\_(model\_creator, data\_creator, config=None, num\_replicas=1, use\_gpu=False, verbose=False)**  
Sets up the TensorFlow trainer.

### Parameters

- **model\_creator** (*dict* → *Model*) – This function takes in the *config* dict and returns a compiled TF model.
- **data\_creator** (*dict* → *tf.Dataset*, *tf.Dataset*) – Creates the training and validation data sets using the *config*. *config* dict is passed into the function.
- **config** (*dict*) – configuration passed to ‘model\_creator’, ‘data\_creator’. Also contains *fit\_config*, which is passed into *model.fit(data, \*\*fit\_config)* and *evaluate\_config* which is passed into *model.evaluate*.
- **num\_replicas** (*int*) – Sets number of workers used in distributed training. Workers will be placed arbitrarily across the cluster.
- **use\_gpu** (*bool*) – Enables all workers to use GPU.
- **verbose** (*bool*) – Prints output of one model if true.

**train()**  
Runs a training epoch.

**validate()**  
Evaluates the model on the validation data set.

**get\_model()**  
Returns the learned model.

**save(checkpoint)**  
Saves the model at the provided checkpoint.

**Parameters** **checkpoint** (*str*) – Path to target checkpoint file.

**restore(checkpoint)**  
Restores the model from the provided checkpoint.

**Parameters** **checkpoint** (*str*) – Path to target checkpoint file.

**shutdown()**  
Shuts down workers and releases resources.

## 5.30 RayServe: Scalable and Programmable Serving

### 5.30.1 Overview

RayServe is a scalable model-serving library built on Ray.

For users RayServe is:

- **Framework Agnostic:** Use the same toolkit to serve everything from deep learning models built with frameworks like PyTorch or TensorFlow to scikit-learn models or arbitrary business logic.
- **Python First:** Configure your model serving with pure Python code - no more YAMLs or JSON configs.

RayServe enables:

- **A/B test models** with zero downtime by decoupling routing logic from response handling logic.
- **Batching** built-in to help you meet your performance objectives.

Since Ray is built on Ray, RayServe also allows you to **scale to many machines** and allows you to leverage all of the other Ray frameworks so you can deploy and scale on any cloud.

---

**Note:** If you want to try out Serve, join our [community slack](#) and discuss in the #serve channel.

---

### RayServe in 90 Seconds

Serve a stateless function:

```
from ray import serve
import requests

serve.init()

def echo(flask_request):
    return "hello " + flask_request.args.get("name", "serve!")

serve.create_endpoint("hello", "/hello")
serve.create_backend("hello", echo)
serve.set_traffic("hello", {"hello": 1.0})

requests.get("http://127.0.0.1:8000/hello").text
# > "hello serve!"
```

Serve a stateful class:

```
from ray import serve
import requests

serve.init()
```

(continues on next page)

(continued from previous page)

```

class Counter:
    def __init__(self):
        self.count = 0

    def __call__(self, flask_request):
        return {"current_counter": self.count}

serve.create_endpoint("counter", "/counter")
serve.create_backend("counter", Counter)
serve.set_traffic("counter", {"counter": 1.0})

requests.get("http://127.0.0.1:8000/counter").json()
# > {"current_counter": self.count}

```

See [Key Concepts](#) for more information about working with RayServe.

## Why RayServe?

There are generally two ways of serving machine learning applications, both with serious limitations: you can build using a **traditional webserver** - your own Flask app or you can use a cloud hosted solution.

The first approach is easy to get started with, but it's hard to scale each component. The second approach requires vendor lock-in (SageMaker), framework specific tooling (TFServing), and a general lack of flexibility.

RayServe solves these problems by giving a user the ability to leverage the simplicity of deployment of a simple webserver but handles the complex routing, scaling, and testing logic necessary for production deployments.

For more on the motivation behind RayServe, check out these [meetup slides](#).

## When should I use Ray Serve?

RayServe should be used when you need to deploy at least one model, preferably many models. RayServe **won't work well** when you need to run batch prediction over a dataset. Given this use case, we recommend looking into [multiprocessing with Ray](#).

### 5.30.2 Key Concepts

RayServe focuses on **simplicity** and only has two core concepts: endpoints and backends.

To follow along, you'll need to make the necessary imports.

```

from ray import serve
serve.init() # initializes serve and Ray

```

## Endpoints

Endpoints allow you to name the “entity” that you’ll be exposing, the HTTP path that your application will expose. Endpoints are “logical” and decoupled from the business logic or model that you’ll be serving. To create one, we’ll simply specify the name, route, and methods.

```
serve.create_endpoint("simple_endpoint", "/simple")
```

You can also delete an endpoint using `serve.delete_endpoint`. Note that this will not delete any associated backends, which can be reused for other endpoints.

```
serve.delete_endpoint("simple_endpoint")
```

## Backends

Backends are the logical structures for your business logic or models and how you specify what should happen when an endpoint is queried. To define a backend, first you must define the “handler” or the business logic you’d like to respond with. The input to this request will be a [Flask Request object](#). Once you define the function (or class) that will handle a request. You’d use a function when your response is stateless and a class when you might need to maintain some state (like a model). For both functions and classes (that take as input Flask Requests), you’ll need to define them as backends to RayServe.

It’s important to note that RayServe places these backends in individual workers, which are replicas of the model.

```
def handle_request(flask_request):
    return "hello world"

class RequestHandler:
    def __init__(self):
        self.msg = "hello, world!"

    def __call__(self, flask_request):
        return self.msg

serve.create_backend("simple_backend", handle_request)
serve.create_backend("simple_backend_class", RequestHandler)
```

Lastly, we need to link the particular backend to the server endpoint. To do that we’ll use the `link` capability. A link is essentially a load-balancer and allow you to define queuing policies for how you would like backends to be served via an endpoint. For instance, you can route 50% of traffic to Model A and 50% of traffic to Model B.

```
serve.set_traffic("simple_backend", {"simple_endpoint": 1.0})
```

Once we’ve done that, we can now query our endpoint via HTTP (we use `requests` to make HTTP calls here).

```
import requests
print(requests.get("http://127.0.0.1:8000/-/routes", timeout=0.5).text)
```

To delete a backend, we can use `serve.delete_backend`. Note that the backend must not be used by any endpoints in order to be deleted. Once a backend is deleted, its tag can be reused.

```
serve.delete_backend("simple_backend")
```

## Configuring Backends

There are a number of things you'll likely want to do with your serving application including scaling out, splitting traffic, or batching input for better response performance. To do all of this, you will create a `BackendConfig`, a configuration object that you'll use to set the properties of a particular backend.

### Scaling Out

To scale out a backend to multiple workers, simply configure the number of replicas.

```
config = {"num_replicas": 2}
serve.create_backend("my_scaled_endpoint_backend", handle_request, config=config)
```

This will scale out the number of workers that can accept requests.

### Splitting Traffic

It's trivial to also split traffic, simply specify the endpoint and the backends that you want to split.

```
serve.create_endpoint("endpoint_identifier_split", "/split", methods=["GET", "POST"])

# splitting traffic 70/30
serve.set_traffic("endpoint_identifier_split", {"my_endpoint_backend": 0.7, "my_
↪endpoint_backend_class": 0.3})
```

### Batching

You can also have RayServe batch requests for performance. You'll configure this in the backend config.

```
class BatchingExample:
    def __init__(self):
        self.count = 0

    @serve.accept_batch
    def __call__(self, flask_request):
        self.count += 1
        batch_size = serve.context.batch_size
        return [self.count] * batch_size

serve.create_endpoint("counter1", "/increment")

config = {"max_batch_size": 5}
serve.create_backend("counter1", BatchingExample, config=config)
serve.set_traffic("counter1", {"counter1": 1.0})
```

### 5.30.3 Other Resources

More coming soon!

## 5.31 Distributed multiprocessing.Pool

Ray supports running distributed python programs with the `multiprocessing.Pool` API using Ray Actors instead of local processes. This makes it easy to scale existing applications that use `multiprocessing.Pool` from a single node to a cluster.

---

**Note:** This API is new and may be revised in future Ray releases. If you encounter any bugs, please file an issue on [GitHub](#).

---

### 5.31.1 Quickstart

To get started, first [install Ray](#), then use `ray.util.multiprocessing.Pool` in place of `multiprocessing.Pool`. This will start a local Ray cluster the first time you create a Pool and distribute your tasks across it. See the [Run on a Cluster](#) section below for instructions to run on a multi-node Ray cluster instead.

```
from ray.util.multiprocessing import Pool

def f(index):
    return index

pool = Pool()
for result in pool.map(f, range(100)):
    print(result)
```

The full `multiprocessing.Pool` API is currently supported. Please see the [multiprocessing documentation](#) for details.

**Warning:** The `context` argument in the `Pool` constructor is ignored when using Ray.

### 5.31.2 Run on a Cluster

This section assumes that you have a running Ray cluster. To start a Ray cluster, please refer to the [cluster setup](#) instructions.

To connect a Pool to a running Ray cluster, you can specify the address of the head node in one of two ways:

- By setting the `RAY_ADDRESS` environment variable.
- By passing the `ray_address` keyword argument to the `Pool` constructor.

```
from ray.util.multiprocessing import Pool

# Starts a new local Ray cluster.
pool = Pool()
```

(continues on next page)

(continued from previous page)

```
# Connects to a running Ray cluster, with the current node as the head node.
# Alternatively, set the environment variable RAY_ADDRESS="auto".
pool = Pool(ray_address="auto")

# Connects to a running Ray cluster, with a remote node as the head node.
# Alternatively, set the environment variable RAY_ADDRESS=<ip_address>:<port>.
pool = Pool(ray_address=<ip_address>:<port>)
```

You can also start Ray manually by calling `ray.init()` (with any of its supported configuration options) before creating a `Pool`.

## 5.32 Distributed Scikit-learn / Joblib

Ray supports running distributed `scikit-learn` programs by implementing a Ray backend for `joblib` using Ray Actors instead of local processes. This makes it easy to scale existing applications that use `scikit-learn` from a single node to a cluster.

---

**Note:** This API is new and may be revised in future Ray releases. If you encounter any bugs, please file an issue on [GitHub](#).

---

### 5.32.1 Quickstart

To get started, first `install Ray`, then use `from ray.util.joblib import register_ray` and run `register_ray()`. This will register Ray as a `joblib` backend for `scikit-learn` to use. Then run your original `scikit-learn` code inside with `joblib.parallel_backend('ray')`. This will start a local Ray cluster. See the *Run on a Cluster* section below for instructions to run on a multi-node Ray cluster instead.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC
digits = load_digits()
param_space = {
    'C': np.logspace(-6, 6, 30),
    'gamma': np.logspace(-8, 8, 30),
    'tol': np.logspace(-4, -1, 30),
    'class_weight': [None, 'balanced'],
}
model = SVC(kernel='rbf')
search = RandomizedSearchCV(model, param_space, cv=5, n_iter=300, verbose=10)

import joblib
from ray.util.joblib import register_ray
register_ray()
with joblib.parallel_backend('ray'):
    search.fit(digits.data, digits.target)
```

### 5.32.2 Run on a Cluster

This section assumes that you have a running Ray cluster. To start a Ray cluster, please refer to the [cluster setup instructions](#).

To connect a scikit-learn to a running Ray cluster, you have to specify the address of the head node by setting the `RAY_ADDRESS` environment variable.

You can also start Ray manually by calling `ray.init()` (with any of its supported configuration options) before calling with `joblib.parallel_backend('ray')`.

**Warning:** If you do not set the `RAY_ADDRESS` environment variable and do not provide address in `ray.init(address=<address>)` then scikit-learn will run on a SINGLE node!

## 5.33 Parallel Iterators

`ray.util.iter` provides a parallel iterator API for simple data ingest and processing. It can be thought of as syntactic sugar around Ray actors and `ray.wait` loops.

Parallel iterators are lazy and can operate over infinite sequences of items. Iterator transformations are only executed when the user calls `next()` to fetch the next output item from the iterator.

---

**Note:** This API is new and may be revised in future Ray releases. If you encounter any bugs, please file an issue on [GitHub](#).

---

### 5.33.1 Concepts

**Parallel Iterators:** You can create a `ParallelIterator` object from an existing set of items, range of numbers, set of iterators, or set of worker actors. Ray will create a worker actor that produces the data for each shard of the iterator:

```
# Create an iterator with 2 worker actors over the list [1, 2, 3, 4].
>>> it = ray.util.iter.from_items([1, 2, 3, 4], num_shards=2)
ParallelIterator[from_items[int, 4, shards=2]]

# Create an iterator with 32 worker actors over range(1000000).
>>> it = ray.util.iter.from_range(1000000, num_shards=32)
ParallelIterator[from_range[1000000, shards=32]]

# Create an iterator over two range(10) generators.
>>> it = ray.util.iter.from_iterators([range(10), range(10)])
ParallelIterator[from_iterators[shards=2]]

# Create an iterator from existing worker actors. These actors must
# implement the ParallelIteratorWorker interface.
>>> it = ray.util.iter.from_actors([a1, a2, a3, a4])
ParallelIterator[from_actors[shards=4]]
```

Simple transformations can be chained on the iterator, such as mapping, filtering, and batching. These will be executed in parallel on the workers:

```
# Apply a transformation to each element of the iterator.
>>> it = it.for_each(lambda x: x ** 2)
ParallelIterator[...].for_each()

# Batch together items into a lists of 32 elements.
>>> it = it.batch(32)
ParallelIterator[...].for_each().batch(32)

# Filter out items with odd values.
>>> it = it.filter(lambda x: x % 2 == 0)
ParallelIterator[...].for_each().batch(32).filter()
```

**Local Iterators:** To read elements from a parallel iterator, it has to be converted to a LocalIterator by calling `gather_sync()` or `gather_async()`. These correspond to `ray.get` and `ray.wait` loops over the actors respectively:

```
# Gather items synchronously (deterministic round robin across shards):
>>> it = ray.util.iter.from_range(1000000, 1)
>>> it = it.gather_sync()
LocalIterator[ParallelIterator[from_range[1000000, shards=1]].gather_sync()]

# Local iterators can be used as any other Python iterator.
>>> it.take(5)
[0, 1, 2, 3, 4]

# They also support chaining of transformations. Unlike transformations
# applied on a ParallelIterator, they will be executed in the current process.
>>> it.filter(lambda x: x % 2 == 0).take(5)
[0, 2, 4, 6, 8]

# Async gather can be used for better performance, but it is non-deterministic.
>>> it = ray.util.iter.from_range(1000, 4).gather_async()
>>> it.take(5)
[0, 250, 500, 750, 1]
```

**Passing iterators to remote functions:** Both `ParallelIterator` and `LocalIterator` are serializable. They can be passed to any Ray remote function. However, note that each shard should only be read by one process at a time:

```
# Get local iterators representing the shards of this ParallelIterator:
>>> it = ray.util.iter.from_range(10000, 3)
>>> [s0, s1, s2] = it.shards()
[LocalIterator[from_range[10000, shards=3].shard[0]],
 LocalIterator[from_range[10000, shards=3].shard[1]],
 LocalIterator[from_range[10000, shards=3].shard[2]]]

# Iterator shards can be passed to remote functions.
>>> @ray.remote
... def do_sum(it):
...     return sum(it)
...
>>> ray.get([do_sum.remote(s) for s in it.shards()])
[5552778, 16661667, 27780555]
```

## Semantic Guarantees

The parallel iterator API guarantees the following semantics:

**Fetch ordering:** When using `it.gather_sync().foreach(fn)` or `it.gather_async().foreach(fn)` (or any other transformation after a gather), `fn(x_i)` will be called on the element `x_i` before the next element `x_{i+1}` is fetched from the source actor. This is useful if you need to update the source actor between iterator steps. Note that for async gather, this ordering only applies per shard.

**Operator state:** Operator state is preserved for each shard. This means that you can pass a stateful callable to `.foreach()`:

```
class CumulativeSum:
    def __init__(self):
        self.total = 0

    def __call__(self, x):
        self.total += x
        return (self.total, x)

it = ray.util.iter.from_range(5, 1)
for x in it.foreach(CumulativeSum()).gather_sync():
    print(x)

## This prints:
#(0, 0)
#(1, 1)
#(3, 2)
#(6, 3)
#(10, 4)
```

## Example: Streaming word frequency count

Parallel iterators can be used for simple data processing use cases such as streaming grep:

```
import ray
import glob
import gzip
import numpy as np

ray.init()

file_list = glob.glob("/var/log/syslog*.gz")
it = (
    ray.util.iter.from_items(file_list, num_shards=4)
    .for_each(lambda f: gzip.open(f).readlines())
    .flatten()
    .for_each(lambda line: line.decode("utf-8"))
    .for_each(lambda line: 1 if "cron" in line else 0)
    .batch(1024)
    .for_each(np.mean)
)

# Show the probability of a log line containing "cron", with a
# sliding window of 1024 lines.
for freq in it.gather_async():
    print(freq)
```

### Example: Passing iterator shards to remote functions

Both parallel iterators and local iterators are fully serializable, so once created you can pass them to Ray tasks and actors. This can be useful for distributed training:

```
import ray
import numpy as np

ray.init()

@ray.remote
def train(data_shard):
    for batch in data_shard:
        print("train on", batch) # perform model update with batch

it = (
    ray.util.iter.from_range(1000000, num_shards=4, repeat=True)
    .batch(1024)
    .for_each(np.array)
)

work = [train.remote(shard) for shard in it.shards()]
ray.get(work)
```

## 5.33.2 API Reference

`ray.util.iter.from_items(items: List[T], num_shards: int = 2, repeat: bool = False) → ray.util.iter.ParallelIterator[~T][T]`

Create a parallel iterator from an existing set of objects.

The objects will be divided round-robin among the number of shards.

#### Parameters

- **items** (*list*) – The list of items to iterate over.
- **num\_shards** (*int*) – The number of worker actors to create.
- **repeat** (*bool*) – Whether to cycle over the items forever.

`ray.util.iter.from_range(n: int, num_shards: int = 2, repeat: bool = False) → ray.util.iter.ParallelIterator[int][int]`

Create a parallel iterator over the range 0..n.

The range will be partitioned sequentially among the number of shards.

#### Parameters

- **n** (*int*) – The max end of the range of numbers.
- **num\_shards** (*int*) – The number of worker actors to create.
- **repeat** (*bool*) – Whether to cycle over the range forever.

`ray.util.iter.from_iterators(generators: List[Iterable[T]], repeat: bool = False, name=None) → ray.util.iter.ParallelIterator[~T][T]`

Create a parallel iterator from a set of iterators.

An actor will be created for each iterator.

## Examples

```
>>> # Create using a list of generators.  
>>> from_iterators([range(100), range(100)])
```

```
>>> # Equivalent to the above.  
>>> from_iterators([lambda: range(100), lambda: range(100)])
```

## Parameters

- **generators** (*list*) – A list of Python generator objects or lambda functions that produced a generator when called. We allow lambda functions since the generator itself might not be serializable, but a lambda that returns it can be.
- **repeat** (*bool*) – Whether to cycle over the iterators forever.
- **name** (*str*) – Optional name to give the iterator.

`ray.util.iter.from_actors` (*actors*: *List[ray.actor.ActorHandle]*, *name=None*) →  
`ray.util.iter.ParallelIterator[~T][T]`

Create a parallel iterator from an existing set of actors.

Each actor must subclass the ParallelIteratorWorker interface.

## Parameters

- **actors** (*list*) – List of actors that each implement ParallelIteratorWorker.
- **name** (*str*) – Optional name to give the iterator.

`class ray.util.iter.ParallelIterator` (*actor\_sets*: *List[\_ActorSet]*, *name: str*, *parent\_iterators: List[ParallelIterator[Any]]*)

Bases: `typing.Generic`

A parallel iterator over a set of remote actors.

This can be used to iterate over a fixed set of task results (like an actor pool), or a stream of data (e.g., a fixed range of numbers, an infinite stream of RLlib rollout results).

This class is **serializable** and can be passed to other remote tasks and actors. However, each shard should be read from at most one process at a time.

## Examples

```
>>> # Applying a function over items in parallel.  
>>> it = ray.util.iter.from_items([1, 2, 3], num_shards=2)  
... <__main__.ParallelIterator object>  
>>> it = it.for_each(lambda x: x * 2).gather_sync()  
... <__main__.LocalIterator object>  
>>> print(list(it))  
... [2, 4, 6]
```

```
>>> # Creating from generators.  
>>> it = ray.util.iter.from_iterators([range(3), range(3)])  
... <__main__.ParallelIterator object>  
>>> print(list(it.gather_sync()))  
... [0, 0, 1, 1, 2, 2]
```

```
>>> # Accessing the individual shards of an iterator.
>>> it = ray.util.iter.from_range(10, num_shards=2)
... <__main__.ParallelIterator object>
>>> it0 = it.get_shard(0)
... <__main__.LocalIterator object>
>>> print(list(it0))
... [0, 1, 2, 3, 4]
>>> it1 = it.get_shard(1)
... <__main__.LocalIterator object>
>>> print(list(it1))
... [5, 6, 7, 8, 9]
```

```
>>> # Gathering results from actors synchronously in parallel.
>>> it = ray.util.iter.from_actors(workers)
... <__main__.ParallelIterator object>
>>> it = it.batch_across_shards()
... <__main__.LocalIterator object>
>>> print(next(it))
... [worker_1_result_1, worker_2_result_1]
>>> print(next(it))
... [worker_1_result_2, worker_2_result_2]
```

**for\_each**(fn: Callable[[T], U]) → ray.util.iter.ParallelIterator[~U][U]

Remotely apply fn to each item in this iterator.

**Parameters** **fn** (*func*) – function to apply to each item.

**Examples**

```
>>> next(from_range(4).for_each(lambda x: x * 2).gather_sync())
... [0, 2, 4, 8]
```

**filter**(fn: Callable[[T], bool]) → ray.util.iter.ParallelIterator[~T][T]

Remotely filter items from this iterator.

**Parameters** **fn** (*func*) – returns False for items to drop from the iterator.

**Examples**

```
>>> it = from_items([0, 1, 2]).filter(lambda x: x > 0)
>>> next(it.gather_sync())
... [1, 2]
```

**batch**(n: int) → ray.util.iter.ParallelIterator[typing.List[~T]][List[T]]

Remotely batch together items in this iterator.

**Parameters** **n** (*int*) – Number of items to batch together.

## Examples

```
>>> next(from_range(10, 1).batch(4).gather_sync())
... [0, 1, 2, 3]
```

**flatten()** → ParallelIterator[T[0]]  
Flatten batches of items into individual items.

## Examples

```
>>> next(from_range(10, 1).batch(4).flatten())
... 0
```

**combine** (*fn: Callable[[T], List[U]]*) → ray.util.iter.ParallelIterator[~U][U]  
Transform and then combine items horizontally.

This is the equivalent of `for_each(fn).flatten()` (flat map).

**local\_shuffle** (*shuffle\_buffer\_size: int, seed: int = None*) → ray.util.iter.ParallelIterator[~T][T]  
Remotely shuffle items of each shard independently

### Parameters

- **shuffle\_buffer\_size** (*int*) – The algorithm fills a buffer with `shuffle_buffer_size` elements and randomly samples elements from this buffer, replacing the selected elements with new elements. For perfect shuffling, this argument should be greater than or equal to the largest iterator size.
- **seed** (*int*) – Seed to use for randomness. Default value is `None`.

**Returns** A ParallelIterator with a local shuffle applied on the base iterator

## Examples

```
>>> it = from_range(10, 1).local_shuffle(shuffle_buffer_size=2)
>>> it = it.gather_sync()
>>> next(it)
0
>>> next(it)
2
>>> next(it)
3
>>> next(it)
1
```

**repartition** (*num\_partitions: int*) → ray.util.iter.ParallelIterator[~T][T]  
Returns a new ParallelIterator instance with `num_partitions` shards.

The new iterator contains the same data in this instance except with `num_partitions` shards. The data is split in round-robin fashion for the new ParallelIterator.

**Parameters** **num\_partitions** (*int*) – The number of shards to use for the new ParallelIterator

**Returns** A ParallelIterator with `num_partitions` number of shards and the data of this ParallelIterator split round-robin among the new number of shards.

## Examples

```
>>> it = from_range(8, 2)
>>> it = it.repartition(3)
>>> list(it.get_shard(0))
[0, 4, 3, 7]
>>> list(it.get_shard(1))
[1, 5]
>>> list(it.get_shard(2))
[2, 6]
```

**gather\_sync()** → ray.util.iter.LocalIterator[~T][T]

Returns a local iterable for synchronous iteration.

New items will be fetched from the shards on-demand as the iterator is stepped through.

This is the equivalent of `batch_across_shards().flatten()`.

## Examples

```
>>> it = from_range(100, 1).gather_sync()
>>> next(it)
...
0
>>> next(it)
...
1
>>> next(it)
...
2
```

**batch\_across\_shards()** → ray.util.iter.LocalIterator[typing.List[~T]][List[T]]

Iterate over the results of multiple shards in parallel.

## Examples

```
>>> it = from_iterators([range(3), range(3)])
>>> next(it.batch_across_shards())
...
[0, 0]
```

**gather\_async(*async\_queue\_depth=1*)** → ray.util.iter.LocalIterator[~T][T]

Returns a local iterable for asynchronous iteration.

New items will be fetched from the shards asynchronously as soon as the previous one is computed. Items arrive in non-deterministic order.

**Parameters** `async_queue_depth (int)` – The max number of async requests in flight per actor. Increasing this improves the amount of pipeline parallelism in the iterator.

## Examples

```
>>> it = from_range(100, 1).gather_async()
>>> next(it)
...
3
>>> next(it)
...
0
>>> next(it)
...
1
```

**take** (*n: int*) → List[T]

Return up to the first n items from this iterator.

**show** (*n: int = 20*)

Print up to the first n items from this iterator.

**union** (*other: ray.util.iter.ParallelIterator[~T][T]*) → ray.util.iter.ParallelIterator[~T][T]

Return an iterator that is the union of this and the other.

**num\_shards** () → int

Return the number of worker actors backing this iterator.

**shards** () → List[ray.util.iter.LocalIterator[~T][T]]

Return the list of all shards.

**get\_shard** (*shard\_index: int*) → ray.util.iter.LocalIterator[~T][T]

Return a local iterator for the given shard.

The iterator is guaranteed to be serializable and can be passed to remote tasks or actors.

```
class ray.util.iter.LocalIterator(base_iterator: Callable[], Iterable[T]), shared_metrics:
    ray.util.iter_metrics.SharedMetrics, local_transforms:
    List[Callable[[Iterable], Any]] = None, timeout: int =
    None, name=None)
```

Bases: typing.Generic

An iterator over a single shard of data.

It implements similar transformations as ParallelIterator[T], but the transforms will be applied locally and not remotely in parallel.

This class is **Serializable** and can be passed to other remote tasks and actors. However, it should be read from at most one process at a time.

**static get\_metrics** () → ray.util.iter\_metrics.MetricsContext

Return the current metrics context.

This can only be called within an iterator function.

**shuffle** (*shuffle\_buffer\_size: int, seed: int = None*) → ray.util.iter.LocalIterator[~T][T]

Shuffle items of this iterator

### Parameters

- **shuffle\_buffer\_size** (*int*) – The algorithm fills a buffer with shuffle\_buffer\_size elements and randomly samples elements from this buffer, replacing the selected elements with new elements. For perfect shuffling, this argument should be greater than or equal to the largest iterator size.
- **seed** (*int*) – Seed to use for randomness. Default value is None.

**Returns** A new LocalIterator with shuffling applied

**take** (*n: int*) → List[T]

Return up to the first *n* items from this iterator.

**show** (*n: int = 20*)

Print up to the first *n* items from this iterator.

**duplicate** (*n*) → List[ray.util.iter.LocalIterator[~T][T]]

Copy this iterator *n* times, duplicating the data.

#### Returns

**multiple iterators that each have a copy** of the data of this iterator.

**Return type** List[[LocalIterator](#)[T]]

**union** (\**others*: ray.util.iter.LocalIterator[~T][T], *deterministic*: bool = False) →

ray.util.iter.LocalIterator[~T][T]

Return an iterator that is the union of this and the others.

If deterministic=True, we alternate between reading from one iterator and the others. Otherwise we return items from iterators as they become ready.

**class** ray.util.iter.**ParallelIteratorWorker** (*item\_generator: Any, repeat: bool*)

Bases: object

Worker actor for a ParallelIterator.

Actors that are passed to iter.from\_actors() must subclass this interface.

**par\_iter\_init** (*transforms*)

Implements ParallelIterator worker init.

**par\_iter\_next** ()

Implements ParallelIterator worker item fetch.

**par\_iter\_slice** (*step: int, start: int*)

Iterates in increments of step starting from start.

## 5.34 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.

## 5.35 Development Tips

---

**Note:** Unless otherwise stated, directory and file paths are relative to the project root directory.

---

### 5.35.1 Compilation

To speed up compilation, be sure to install Ray with the following commands:

```
cd 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/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/anaconda3` (substituting in the appropriate path).

If you make changes to the C++ or Python files, you will need to run the build so C++ code is recompiled and/or Python files are redeployed in the `python` directory. However, you do not need to rerun `pip install -e ..`. Instead, you can recompile much more quickly by running the following:

```
bash build.sh
```

This command is not enough to recompile all C++ unit tests. To do so, see [Testing locally](#).

### 5.35.2 Developing Ray (Python Only)

---

**Note:** Unless otherwise stated, directory and file paths are relative to the project root directory.

---

RLLib, Tune, Autoscaler, and most Python files do not require you to build and compile Ray. Follow these instructions to develop Ray’s Python files locally.

1. Pip install the **latest Ray wheels**. See [Latest Snapshots \(Nightlies\)](#) for instructions.
2. Fork and clone the project to your machine. Connect your repository to the upstream (main project) ray repository.

```
git clone https://github.com/[your username]/ray.git  
cd ray  
git remote add upstream https://github.com/ray-project/ray.git  
# Make sure you are up-to-date on master.
```

4. Run `python python/ray/setup-dev.py`. This sets up links between the `tune` dir (among other directories) in your local repo and the one bundled with the `ray` package.

**Warning:** Do not run `pip uninstall ray` or `pip install -U` (for Ray or Ray wheels) if setting up your environment this way. To uninstall or upgrade, you must `rm -rf` the installation site (usually a `site-packages/ray` location).

### 5.35.3 Using a local repository for dependencies

If you'd like to build Ray with custom dependencies (for example, with a different version of Cython), you can modify your `.bzl` file as follows:

```
http_archive(
    name = "cython",
    ...
) if False else native.new_local_repository(
    name = "cython",
    build_file = "bazel/BUILD.cython",
    path = ".../cython",
)
```

This replaces the existing `http_archive` rule with one that references a sibling of your Ray directory (named `cython`) using the build file provided in the Ray repository (`bazel/BUILD.cython`). If the dependency already has a Bazel build file in it, you can use `native.local_repository` instead, and omit `build_file`.

To test switching back to the original rule, change `False` to `True`.

### 5.35.4 Debugging

#### Starting processes in a debugger

When processes are crashing, it is often useful to start them in a debugger. Ray currently allows processes to be started in the following:

- valgrind
- the valgrind profiler
- the perfetto profiler
- gdb
- tmux

To use any of these tools, please make sure that you have them installed on your machine first (`gdb` and `valgrind` on MacOS are known to have issues). Then, you can launch a subset of ray processes by adding the environment variable `RAY_{PROCESS_NAME}_{DEBUGGER}=1`. For instance, if you wanted to start the raylet in `valgrind`, then you simply need to set the environment variable `RAY_RAYLET_VALGRIND=1`.

To start a process inside of `gdb`, the process must also be started inside of `tmux`. So if you want to start the raylet in `gdb`, you would start your Python script with the following:

```
RAY_RAYLET_GDB=1 RAY_RAYLET_TMUX=1 python
```

You can then list the `tmux` sessions with `tmux ls` and attach to the appropriate one.

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 global state 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.nodes()  
# Returns current information about the nodes in the cluster, such as:  
# [{"ClientID": "2a9d2b34ad24a37ed54e4fc32bf19f915742f5b",  
#  "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](#).

## 5.35.5 Testing locally

### Testing for Python development

Suppose that one of the tests in a file of tests, e.g., `python/ray/tests/test_basic.py`, is failing. You can run just that test file locally as follows:

```
python -m pytest -v python/ray/tests/test_basic.py
```

However, this will run all of the tests in the file, which can take some time. To run a specific test that is failing, you can do the following instead:

```
python -m pytest -v python/ray/tests/test_basic.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 file.

## Testing for C++ development

To compile and run all C++ tests, you can run:

```
bazel test $(bazel query 'kind(cc_test, ...)')
```

Alternatively, you can also run one specific C++ test. You can use:

```
bazel test $(bazel query 'kind(cc_test, ...)') --test_filter=ClientConnectionTest --  
--test_output=streamed
```

## 5.35.6 Building the Docs

If you make changes that require documentation changes, don't forget to update the documentation!

When you make documentation changes, build them locally to verify they render correctly. [Sphinx](#) is used to generate the documentation.

```
cd doc  
pip install -r requirements-doc.txt  
make html
```

Once done, the docs will be in `doc/_build/html`. For example, on Mac OSX, you can open the docs (assuming you are still in the `doc` directory) using `open _build/html/index.html`.

## 5.35.7 Creating a pull request

To create a pull request (PR) for your change, first go through the [PR template](#) checklist and ensure you've completed all the steps.

When you push changes to GitHub, the formatting and verification script `ci/travis/format.sh` is run first. For pushing to your fork, you can skip this step with `git push --no-verify`.

Before submitting the PR, you should run this script. If it fails, the push operation will not proceed. This script requires *specific versions* of the following tools. Installation commands are shown for convenience:

- `yapf` version 0.23.0 (`pip install yapf==0.23.0`)
- `flake8` version 3.7.7 (`pip install flake8==3.7.7`)
- `flake8-quotes` (`pip install flake8-quotes`)
- `clang-format` version 7.0.0 (download this version of Clang from [here](#))

**Note:** On MacOS X, don't use HomeBrew to install `clang-format`, as the only version available is too new.

The Ray project automatically runs continuous integration (CI) tests once a PR is opened using [Travis-CI](#) with multiple CI test jobs.

### 5.35.8 Understand CI test jobs

The [Travis CI](#) test folder contains all integration test scripts and they invoke other test scripts via `pytest`, `bazel`-based test or other bash scripts. Some of the examples include:

- **Raylet integration tests commands:**

```
- src/ray/test/run_core_worker_tests.sh  
- src/ray/test/run_object_manager_tests.sh
```

- **Bazel test command:**

```
- bazel test --build_tests_only //:all
```

- **Ray serving test commands:**

```
- python -m pytest python/ray/serve/tests  
- python python/ray/serve/examples/echo_full.py
```

If a Travis-CI build exception doesn't appear to be related to your change, please visit this [link](#) to check recent tests known to be flaky.

### 5.35.9 Format and Linting

Installation instructions for the tools mentioned here are discussed above in [Creating a pull request](#).

**Running the linter locally:** To run the Python linter on a specific file, run `flake8` as in this example, `flake8 python/ray/worker.py`.

**Autoformatting code.** We use [yapf](#) for linting. The config file is `.style.yapf`. We recommend running `scripts/yapf.sh` prior to pushing a PR to format any changed files. Note that some projects, such as `dataframes` and `rllib`, are currently excluded.

**Running CI linter:** The Travis CI linter script has multiple components to run. We recommend running `ci/travis/format.sh`, which runs both linters for Python and C++ codes. In addition, there are other formatting checkers for components like the following:

- Python REAME format:

```
cd python  
python setup.py check --restructuredtext --strict --metadata
```

- Bazel format:

```
./ci/travis/bazel-format.sh
```

## 5.36 Profiling for Ray Developers

This document details, for Ray developers, how to use `pprof` to profile Ray binaries.

### 5.36.1 Installation

These instructions are for Ubuntu only. Attempts to get pprof to correctly symbolize on Mac OS have failed.

```
sudo apt-get install google-perf-tools libgoogle-perf-tools-dev
```

### 5.36.2 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.

### 5.36.3 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:

### 5.36.4 Running Microbenchmarks

To run a set of single-node Ray microbenchmarks, use:

```
ray microbenchmark
```

You can find the microbenchmark results for Ray releases in the [GitHub release logs](#).

## 5.36.5 References

- The [pprof documentation](#).
- A [Go version](#) of pprof.
- The [gperftools](#), including libprofiler, tcmalloc, and other goodies.

## 5.37 Fault Tolerance

This document describes how Ray handles machine and process failures.

### 5.37.1 Tasks

When a worker is executing a task, if the worker dies unexpectedly, either because the process crashed or because the machine failed, Ray will rerun the task (after a delay of several seconds) until either the task succeeds or the maximum number of retries is exceeded. The default number of retries is 4.

You can experiment with this behavior by running the following code.

```
import numpy as np
import os
import ray
import time

ray.init(ignore_reinit_error=True)

@ray.remote(max_retries=1)
def potentially_fail(failure_probability):
    time.sleep(0.2)
    if np.random.random() < failure_probability:
        os._exit(0)
    return 0

for _ in range(3):
    try:
        # If this task crashes, Ray will retry it up to one additional
        # time. If either of the attempts succeeds, the call to ray.get
        # below will return normally. Otherwise, it will raise an
        # exception.
        ray.get(potentially_fail.remote(0.5))
        print('SUCCESS')
    except ray.exceptions.RayWorkerError:
        print('FAILURE')
```

## 5.37.2 Actors

If an actor process crashes unexpectedly, Ray will attempt to reconstruct the actor process up to a maximum number of times. This value can be specified with the `max_reconstructions` keyword, which by default is 0. If the maximum number of reconstructions has been used up, then subsequent actor methods will raise exceptions.

When an actor is reconstructed, its state will be recreated by rerunning its constructor.

You can experiment with this behavior by running the following code.

```
import os
import ray
import time

ray.init(ignore_reinit_error=True)

@ray.remote(max_reconstructions=5)
class Actor:
    def __init__(self):
        self.counter = 0

    def increment_and_possibly_fail(self):
        self.counter += 1
        time.sleep(0.2)
        if self.counter == 10:
            os._exit(0)
        return self.counter

actor = Actor.remote()

# The actor will be reconstructed up to 5 times. After that, methods will
# raise exceptions. The actor is reconstructed by rerunning its
# constructor. Methods that were executing when the actor died will also
# raise exceptions.
for _ in range(100):
    try:
        counter = ray.get(actor.increment_and_possibly_fail.remote())
        print(counter)
    except ray.exceptions.RayActorError:
        print('FAILURE')
```

## 5.38 Getting Involved

We welcome (and encourage!) all forms of contributions to Ray, including and not limited to:

- Code reviewing of patches and PRs.
- Pushing patches.
- Documentation and examples.
- Community participation in forums and issues.
- Code readability and code comments to improve readability.
- Test cases to make the codebase more robust.
- Tutorials, blog posts, talks that promote the project.

### 5.38.1 What can I work on?

We use Github to track issues, feature requests, and bugs. Take a look at the ones labeled “good first issue” and “help wanted” for a place to start.

### 5.38.2 Submitting and Merging a Contribution

There are a couple steps to merge a contribution.

1. First rebase your development branch on the most recent version of master.

```
git remote add upstream https://github.com/ray-project/ray.git
git fetch upstream
git rebase upstream/master
```

2. Make sure all existing tests pass.
3. If introducing a new feature or patching a bug, be sure to add new test cases in the relevant file in `ray/python/ray/tests/`.
4. Document the code. Public functions need to be documented, and remember to provide an usage example if applicable.
5. Request code reviews from other contributors and address their comments. One fast way to get reviews is to help review others’ code so that they return the favor. You should aim to improve the code as much as possible before the review. We highly value patches that can get in without extensive reviews.
6. Reviewers will merge and approve the pull request; be sure to ping them if the pull request is getting stale.

### 5.38.3 Testing

Even though we have hooks to run unit tests automatically for each pull request, we recommend you to run unit tests locally beforehand to reduce reviewers’ burden and speedup review process.

```
pytest ray/python/ray/tests/
```

Documentation should be documented in [Google style](#) format.

We also have tests for code formatting and linting that need to pass before merge. Install `yapf==0.23`, `flake8`, `flake8-quotes`. You can run the following locally:

```
ray/scripts/format.sh
```

### 5.38.4 Becoming a Reviewer

We identify reviewers from active contributors. Reviewers are individuals who not only actively contribute to the project and are also willing to participate in the code review of new contributions. A pull request to the project has to be reviewed by at least one reviewer in order to be merged. There is currently no formal process, but active contributors to Ray will be solicited by current reviewers.

### 5.38.5 More Resources for Getting Involved

- [ray-dev@googlegroups.com](mailto:ray-dev@googlegroups.com): For discussions about development or any general questions.
- [StackOverflow](#): For questions about how to use Ray.
- [GitHub Issues](#): For reporting bugs and feature requests.
- [Pull Requests](#): For submitting code contributions.
- [Meetup Group](#): Join our meetup group.
- [Community Slack](#): Join our Slack workspace.
- [Twitter](#): Follow updates on Twitter.

---

**Note:** These tips are based off of the TVM [contributor guide](#).

---



## PYTHON MODULE INDEX

### r

ray.experimental, 132  
ray.rllib.env, 370  
ray.rllib.evaluation, 379  
ray.rllib.models, 389  
ray.rllib.optimizers, 393  
ray.rllib.policy, 357  
ray.rllib.utils, 398  
ray.tune.integration.keras, 216  
ray.tune.track, 215  
ray.util.iter, 445



# INDEX

## Symbols

`_call_()` (*ray.tune.Stopper method*), 207  
`_call_()` (*ray.tune.function\_runner.StatusReporter method*), 216  
`_init_()` (*ray.rllib.models.tf.recurrent\_tf\_modelv2.RecurrentTFModelV2 method*), 304  
`_init_()` (*ray.rllib.models.tf.tf\_modelv2.TFModelV2 method*), 300  
`_init_()` (*ray.rllib.models.torch.torch\_modelv2.TorchModelV2 method*), 305  
`_init_()` (*ray.util.sgd.tf.TFTrainer method*), 435  
`_export_model()` (*ray.tune.Trainable method*), 210  
`_log_result()` (*ray.tune.Trainable method*), 211  
`_restore()` (*ray.tune.Trainable method*), 211  
`_restore()` (*ray.util.sgd.torch.BaseTorchTrainable method*), 434  
`_save()` (*ray.tune.Trainable method*), 211  
`_save()` (*ray.util.sgd.torch.BaseTorchTrainable method*), 434  
`_setup()` (*ray.tune.Trainable method*), 212  
`_setup()` (*ray.util.sgd.torch.BaseTorchTrainable method*), 434  
`_stop()` (*ray.tune.Trainable method*), 212  
`_stop()` (*ray.util.sgd.torch.BaseTorchTrainable method*), 435  
`_train()` (*ray.tune.Trainable method*), 212  
`_train()` (*ray.util.sgd.torch.BaseTorchTrainable method*), 434

`-N`  
    `ray-attach` command line option, 138  
`--address <address>`  
    `ray-globalgc` command line option, 139  
    `ray-memory` command line option, 139  
    `ray-start` command line option, 133  
    `ray-stat` command line option, 139  
    `ray-timeline` command line option, 140  
`--args <args>`  
    `ray-submit` command line option, 137  
`--autoscaling-config <autoscaling_config>`

`ray-start` command line option, 134  
`--block`  
    `ray-start` command line option, 134  
`--cluster-name <cluster_name>`  
`ray-attach` command line option, 138  
    `ray-down` command line option, 136  
    `ray-exec` command line option, 136  
    `ray-get_head_ip` command line  
        `ray-model` option, 139  
    `ray-submit` command line option, 137  
    `ray-up` command line option, 135  
`--docker`  
    `ray-exec` command line option, 136  
    `ray-submit` command line option, 137  
`--force`  
    `ray-stop` command line option, 135  
`--head`  
    `ray-start` command line option, 134  
`--huge-pages`  
    `ray-start` command line option, 134  
`--include-java`  
    `ray-start` command line option, 134  
`--include-webui <include_webui>`  
    `ray-start` command line option, 134  
`--internal-config <internal_config>`  
    `ray-start` command line option, 134  
`--java-worker-options <java_worker_options>`  
    `ray-start` command line option, 134  
`--keep-min-workers`  
    `ray-down` command line option, 136  
`--load-code-from-local`  
    `ray-start` command line option, 134  
`--max-workers <max_workers>`  
    `ray-up` command line option, 135  
`--memory <memory>`  
    `ray-start` command line option, 133  
`--min-workers <min_workers>`  
    `ray-up` command line option, 135  
`--new`  
    `ray-attach` command line option, 138  
`--no-redirect-output`

```
    ray-start command line option, 134
--no-redirect-worker-output
    ray-start command line option, 134
--no-restart
    ray-up command line option, 135
--node-ip-address <node_ip_address>
    ray-start command line option, 133
--node-manager-port
    <node_manager_port>
    ray-start command line option, 133
--num-cpus <num_cpus>
    ray-start command line option, 134
--num-gpus <num_gpus>
    ray-start command line option, 134
--num-redis-shards <num_redis_shards>
    ray-start command line option, 133
--object-manager-port
    <object_manager_port>
    ray-start command line option, 133
--object-store-memory
    <object_store_memory>
    ray-start command line option, 133
--plasma-directory <plasma_directory>
    ray-start command line option, 134
--plasma-store-socket-name
    <plasma_store_socket_name>
    ray-start command line option, 134
--port-forward <port_forward>
    ray-attach command line option, 138
    ray-exec command line option, 136
    ray-submit command line option, 137
--raylet-socket-name
    <raylet_socket_name>
    ray-start command line option, 134
--redis-address <redis_address>
    ray-start command line option, 133
--redis-max-clients
    <redis_max_clients>
    ray-start command line option, 133
--redis-max-memory <redis_max_memory>
    ray-start command line option, 133
--redis-password <redis_password>
    ray-start command line option, 133
--redis-port <redis_port>
    ray-start command line option, 133
--redis-shard-ports
    <redis_shard_ports>
    ray-start command line option, 133
--resources <resources>
    ray-start command line option, 134
--restart-only
    ray-up command line option, 135
--screen
    ray-attach command line option, 138
    ray-exec command line option, 136
    ray-submit command line option, 137
--start
    ray-attach command line option, 138
    ray-exec command line option, 136
    ray-submit command line option, 137
--stop
    ray-exec command line option, 136
    ray-submit command line option, 137
--temp-dir <temp_dir>
    ray-start command line option, 134
--tmux
    ray-attach command line option, 138
    ray-exec command line option, 136
    ray-submit command line option, 137
--verbose
    ray-stop command line option, 135
--webui-host <webui_host>
    ray-start command line option, 134
--workers-only
    ray-down command line option, 136
--yes
    ray-down command line option, 136
    ray-up command line option, 135
-f
    ray-stop command line option, 135
-n
    ray-attach command line option, 138
    ray-down command line option, 136
    ray-exec command line option, 136
    ray-get_head_ip command line
        option, 139
    ray-submit command line option, 137
    ray-up command line option, 135
-p
    ray-attach command line option, 138
    ray-exec command line option, 136
    ray-submit command line option, 137
-v
    ray-stop command line option, 135
-y
    ray-down command line option, 136
    ray-up command line option, 135
```

## A

action\_space (*ray.rllib.env.BaseEnv attribute*), 370  
action\_space (*ray.rllib.env.ExternalEnv attribute*), 374  
action\_space (*ray.rllib.env.VectorEnv attribute*), 377  
action\_space (*ray.rllib.policy.Policy attribute*), 357  
action\_space (*ray.rllib.policy.TFPolicy attribute*), 364  
action\_space (*ray.rllib.policy.TorchPolicy attribute*), 361

ActionDistribution (*class in ray.rllib.models*), 389  
add\_batch () (*ray.rllib.evaluation.SampleBatchBuilder method*), 386  
add\_configurations ()  
(ray.tune.suggest.SearchAlgorithm method), 236  
add\_extra\_batch (*ray.rllib.evaluation.MultiAgentEpisode attribute*), 379  
add\_metric\_column ()  
(ray.tune.CLIReporter method), 218  
add\_metric\_column ()  
(ray.tune.JupyterNotebookReporter method), 218  
add\_mixins () (*in module ray.rllib.utils*), 403  
add\_trial ()  
(ray.tune.web\_server.TuneClient method), 255  
add\_values () (*ray.rllib.evaluation.MultiAgentSampleBatchBuilder method*), 386  
add\_values () (*ray.rllib.evaluation.SampleBatchBuilder method*), 386  
agent\_rewards (*ray.rllib.evaluation.MultiAgentEpisode attribute*), 380  
Analysis (*class in ray.tune*), 221  
apply ()  
(ray.rllib.evaluation.EvaluatorInterface method), 386  
apply\_all\_operators ()  
(ray.util.sgd.torch.TorchTrainer 429  
apply\_all\_workers ()  
(ray.util.sgd.torch.TorchTrainer 429  
apply\_changes () (*ray.rllib.utils.Filter method*), 399  
apply\_gradients ()  
(ray.rllib.evaluation.EvaluatorInterface method), 385  
apply\_gradients ()  
(ray.rllib.evaluation.RolloutWorker method), 383  
apply\_gradients ()  
(ray.rllib.policy.Policy method), 360  
apply\_gradients ()  
(ray.rllib.policy.TFPolicy method), 366  
apply\_gradients ()  
(ray.rllib.policy.TorchPolicy method), 363  
as\_trainable ()  
(ray.util.sgd.torch.TorchTrainer class method), 430  
ASHAScheduler (*in module ray.tune.schedulers*), 239  
assignment\_nodes (*ray.experimental.tf\_utils.TensorFlowVariables attribute*), 54  
AsyncGradientsOptimizer  
(ray.rllib.optimizers), 395  
AsyncHyperBandScheduler  
(ray.tune.schedulers), 239  
AsyncReplayOptimizer  
(ray.rllib.optimizers), 394  
AsyncSampler (*class in ray.rllib.evaluation*), 387  
AsyncSamplesOptimizer  
(ray.rllib.optimizers), 395  
available\_resources () (*in module ray*), 132  
AxSearch (*class in ray.tune.suggest.ax*), 227

**B**

BaseEnv (*class in ray.rllib.env*), 370  
BaseTorchTrainable (*class in ray.util.sgd.torch*), 434  
BasicVariantGenerator  
(ray.tune.suggest), 226  
batch () (*ray.util.iter.ParallelIterator method*), 447  
batch\_across\_shards ()  
(ray.util.iter.ParallelIterator method), 449  
BatchBuilder (*ray.rllib.evaluation.MultiAgentEpisode attribute*), 379  
BayesOptSearch (*class in ray.tune.suggest.bayesoft*), 228  
build\_and\_reset ()  
(ray.rllib.evaluation.MultiAgentSampleBatchBuilder method), 387  
build\_and\_reset ()  
(ray.rllib.evaluation.SampleBatchBuilder method), 386  
build\_apply\_op ()  
(ray.rllib.policy.TFPolicy method), 367  
build\_tf\_policy () (*in module ray.rllib.policy*), 369  
build\_torch\_policy ()  
(in module ray.rllib.policy), 367

**C**

cancel () (*in module ray*), 129  
check () (*in module ray.rllib.utils*), 402  
check\_framework () (*in module ray.rllib.utils*), 399  
check\_shape ()  
(ray.rllib.models.Preprocessor method), 392  
choice () (*in module ray.tune*), 225  
choose\_trial\_to\_run ()  
(ray.tune.schedulers.TrialScheduler method), 242  
cleanup () (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 249  
cleanup ()  
(ray.tune.trial\_executor.TrialExecutor method), 252  
clear\_buffer () (*ray.rllib.utils.Filter method*), 400  
CLIReporter (*class in ray.tune*), 218  
CLUSTER\_CONFIG\_FILE  
ray-attach command line option, 138  
ray-down command line option, 136  
ray-exec command line option, 137

ray-get\_head\_ip command line option, 139  
ray-submit command line option, 138  
ray-up command line option, 135  
cluster\_resources() (*in module* ray), 131  
CMD  
    ray-exec command line option, 137  
collect\_metrics() (*in module* ray.rllib.evaluation), 388  
collect\_metrics() (*ray.rllib.optimizers.PolicyOptimizer method*), 393  
columns() (*ray.rllib.evaluation.SampleBatch method*), 388  
combine() (*ray.util.iter.ParallelIterator method*), 448  
compute\_actions() (*ray.rllib.policy.Policy method*), 357  
compute\_actions() (*ray.rllib.policy.TFPolicy method*), 365  
compute\_actions() (*ray.rllib.policy.TorchPolicy method*), 361  
compute\_advantages() (*in module* ray.rllib.evaluation), 387  
compute\_gradients() (*ray.rllib.evaluation.EvaluatorInterface method*), 385  
compute\_gradients() (*ray.rllib.evaluation.RolloutWorker method*), 382  
compute\_gradients() (*ray.rllib.policy.Policy method*), 360  
compute\_gradients() (*ray.rllib.policy.TFPolicy method*), 366  
compute\_gradients() (*ray.rllib.policy.TorchPolicy method*), 363  
compute\_log\_likelihoods() (*ray.rllib.policy.Policy method*), 358  
compute\_log\_likelihoods() (*ray.rllib.policy.TFPolicy method*), 366  
compute\_log\_likelihoods() (*ray.rllib.policy.TorchPolicy method*), 362  
compute\_single\_action() (*ray.rllib.policy.Policy method*), 358  
concat() (*ray.rllib.evaluation.SampleBatch method*), 388  
ConcurrencyLimiter (*class in* ray.tune.suggest), 227  
config (*ray.rllib.optimizers.PolicyOptimizer attribute*), 393  
config (*ray.rllib.policy.TorchPolicy attribute*), 361  
config (*ray.tune.trial.Trial attribute*), 253  
config() (*ray.util.sgd.torch.TrainingOperator property*), 433  
ConstantSchedule (*class in* ray.rllib.utils), 402  
CONTINUE (*ray.tune.schedulers.TrialScheduler attribute*), 242  
continue\_training() (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248  
continue\_training() (*ray.tune.trial\_executor.TrialExecutor method*), 250  
copy() (*ray.rllib.policy.TFPolicy method*), 367  
copy() (*ray.rllib.utils.Filter method*), 399  
creation\_args() (*ray.rllib.evaluation.RolloutWorker method*), 384  
criterion() (*ray.util.sgd.torch.TrainingOperator property*), 433  
CSVLogger (*class in* ray.tune.logger), 245  
custom\_loss() (*ray.rllib.models.Model method*), 392  
custom\_loss() (*ray.rllib.models.tf\_tf\_modelv2.TFModelV2 method*), 301  
custom\_loss() (*ray.rllib.models.torch.torch\_modelv2.TorchModelV2 method*), 306  
custom\_metrics (*ray.rllib.evaluation.MultiAgentEpisode attribute*), 380  
custom\_stats() (*ray.rllib.models.Model method*), 392

## D

dataframe() (*ray.tune.Analysis method*), 221  
debug\_string() (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248  
debug\_string() (*ray.tune.schedulers.TrialScheduler method*), 242  
debug\_string() (*ray.tune.trial\_executor.TrialExecutor method*), 251  
deep\_update() (*in module* ray.rllib.utils), 403  
default\_resource\_request() (*ray.tune.Trainable class method*), 212  
DefaultCallbacks (*class in* ray.rllib.agents.callbacks), 278  
delete\_checkpoint() (*ray.tune.Trainable method*), 212  
deprecation\_warning() (*in module* ray.rllib.utils), 399  
deterministic\_sample() (*ray.rllib.models.ActionDistribution method*), 389  
DeveloperAPI() (*in module* ray.rllib.utils), 398  
DeveloperAPI() (*in module* ray.rllib.utils.annotations), 405  
device() (*ray.util.sgd.torch.TrainingOperator property*), 433  
device\_ids() (*ray.util.sgd.torch.TrainingOperator property*), 434  
dist\_class (*ray.rllib.policy.TorchPolicy attribute*), 361

DragonflySearch (class in `ray.tune.suggest.dragonfly`), 229

`duplicate()` (`ray.util.iter.LocalIterator` method), 451

DurableTrainable (class in `ray.tune`), 215

## E

`end_episode()` (`ray.rllib.env.ExternalEnv` method), 375

`end_episode()` (`ray.rllib.env.ExternalMultiAgentEnv` method), 376

`end_episode()` (`ray.rllib.env.policy_client.PolicyClient` method), 288

`end_episode()` (`ray.rllib.env.PolicyClient` method), 378

`end_episode()` (`ray.rllib.utils.PolicyClient` method), 402

`entropy()` (`ray.rllib.models.ActionDistribution` method), 389

EnvContext (class in `ray.rllib.env`), 377

`episode_id` (`ray.rllib.evaluation.MultiAgentEpisode` attribute), 380

`error_file` (`ray.tune.trial.Trial` attribute), 253

`errors()` (in module `ray`), 132

`evaluated_params` (`ray.tune.trial.Trial` attribute), 253

EvaluatorInterface (class in `ray.rllib.evaluation`), 384

`Experiment()` (in module `ray.tune`), 206

`experiment_tag` (`ray.tune.trial.Trial` attribute), 253

ExperimentAnalysis (class in `ray.tune`), 220

`exploration` (`ray.rllib.policy.Policy` attribute), 357

ExponentialSchedule (class in `ray.rllib.utils`), 402

`export_checkpoint()` (`ray.rllib.policy.Policy` method), 361

`export_checkpoint()` (`ray.rllib.policy.TFPolicy` method), 367

`export_checkpoint()` (`ray.rllib.policy.TorchPolicy` method), 364

`export_model()` (`ray.rllib.policy.Policy` method), 361

`export_model()` (`ray.rllib.policy.TFPolicy` method), 367

`export_model()` (`ray.rllib.policy.TorchPolicy` method), 364

`export_model()` (`ray.tune.Trainable` method), 212

`export_trial_if_needed()` (`ray.tune.ray_trial_executor.RayTrialExecutor` method), 249

`export_trial_if_needed()` (`ray.tune.trial_executor.TrialExecutor` method), 251

ExternalEnv (class in `ray.rllib.env`), 374

ExternalMultiAgentEnv (class in `ray.rllib.env`), 375

in `extra_action_out()` (`ray.rllib.policy.TorchPolicy` method), 363

`extra_compute_action_feed_dict()` (`ray.rllib.policy.TFPolicy` method), 367

`extra_compute_action_fetches()` (`ray.rllib.policy.TFPolicy` method), 367

`extra_compute_grad_feed_dict()` (`ray.rllib.policy.TFPolicy` method), 367

`extra_compute_grad_fetches()` (`ray.rllib.policy.TFPolicy` method), 367

`extra_grad_info()` (`ray.rllib.policy.TorchPolicy` method), 364

`extra_grad_process()` (`ray.rllib.policy.TorchPolicy` method), 363

## F

`fc()` (in module `ray.rllib.utils`), 400

`fetch_result()` (`ray.tune.ray_trial_executor.RayTrialExecutor` method), 248

`fetch_result()` (`ray.tune.trial_executor.TrialExecutor` method), 251

FIFOScheduler (class in `ray.tune.schedulers`), 238

Filter (class in `ray.rllib.utils`), 399

`filter()` (`ray.util.iter.ParallelIterator` method), 447

FilterManager (class in `ray.rllib.utils`), 399

`find_free_port()` (`ray.rllib.evaluation.RolloutWorker` method), 384

`flatten()` (`ray.util.iter.ParallelIterator` method), 448

`for_each()` (`ray.util.iter.ParallelIterator` method), 447

`for_policy()` (`ray.rllib.evaluation.RolloutWorker` method), 384

`force_list()` (in module `ray.rllib.utils`), 404

`force_tuple()` (in module `ray.rllib.utils`), 404

`foreach_env()` (`ray.rllib.evaluation.RolloutWorker` method), 383

`foreach_policy()` (`ray.rllib.evaluation.RolloutWorker` method), 384

`foreach_trainable_policy()` (`ray.rllib.evaluation.RolloutWorker` method), 384

`foreach_worker()` (`ray.rllib.optimizers.PolicyOptimizer` method), 394

`foreach_worker_with_index()` (`ray.rllib.optimizers.PolicyOptimizer` method), 394

`forward()` (`ray.rllib.models.tf.tf_modelv2.TFModelV2` method), 301

`forward()` (`ray.rllib.models.torch.torch_modelv2.TorchModelV2` method), 305

`forward_rnn()` (`ray.rllib.models.tf.recurrent_tf_modelv2.RecurrentTFRNN` method), 304

`framework_iterator()` (in module `ray.rllib.utils`), 403

`from_actors()` (in module `ray.util.iter`), 446

from\_items () (*in module ray.util.iter*), 445  
from\_iterators () (*in module ray.util.iter*), 445  
from\_range () (*in module ray.util.iter*), 445  
FullyConnectedNetwork (class) in  
    *ray.rllib.models*, 393

**G**

gather\_async () (*ray.util.iter.ParallelIterator method*), 449  
gather\_sync () (*ray.util.iter.ParallelIterator method*), 449  
get () (*in module ray*), 127  
get () (*in module ray.experimental*), 132  
get\_action () (*ray.rllib.env.ExternalEnv method*), 375  
get\_action () (*ray.rllib.env.ExternalMultiAgentEnv method*), 376  
get\_action () (*ray.rllib.env.policy\_client.PolicyClient method*), 288  
get\_action () (*ray.rllib.env.PolicyClient method*), 378  
get\_action () (*ray.rllib.utils.PolicyClient method*), 401  
get\_action\_dist ()  
    (*ray.rllib.models.ModelCatalog static method*), 390  
get\_action\_placeholder ()  
    (*ray.rllib.models.ModelCatalog static method*), 390  
get\_action\_shape ()  
    (*ray.rllib.models.ModelCatalog static method*), 390  
get\_all\_configs () (*ray.tune.Analysis method*), 221  
get\_all\_trials () (*ray.tune.web\_server.TuneClient method*), 255  
get\_best\_config () (*ray.tune.Analysis method*), 221  
get\_best\_config () (*ray.tune.ExperimentAnalysis method*), 220  
get\_best\_logdir () (*ray.tune.Analysis method*), 221  
get\_best\_logdir () (*ray.tune.ExperimentAnalysis method*), 221  
get\_best\_trial () (*ray.tune.ExperimentAnalysis method*), 220  
get\_checkpoints ()  
    (*ray.tune.trial\_executor.TrialExecutor method*), 250  
get\_config () (*ray.tune.Trainable method*), 213  
get\_exploration\_info () (*ray.rllib.policy.Policy method*), 360  
get\_exploration\_info ()  
    (*ray.rllib.policy.TFPolicy method*), 366

get\_filters () (*ray.rllib.evaluation.RolloutWorker method*), 384  
get\_flat () (*ray.experimental.tf\_utils.TensorFlowVariables method*), 54  
get\_flat\_size () (*ray.experimental.tf\_utils.TensorFlowVariables method*), 54  
get\_gpu\_ids () (*in module ray*), 129  
get\_host () (*ray.rllib.evaluation.EvaluatorInterface method*), 386  
get\_initial\_state ()  
    (*ray.rllib.models.tf.recurrent\_tf\_modelv2.RecurrentTFModelV2 method*), 304  
get\_initial\_state ()  
    (*ray.rllib.models.torch.torch\_modelv2.TorchModelV2 method*), 306  
get\_initial\_state ()  
    (*ray.rllib.policy.Policy method*), 360  
get\_initial\_state () (*ray.rllib.policy.TorchPolicy method*), 363  
get\_local\_operator ()  
    (*ray.util.sgd.torch.TorchTrainer method*), 429  
get\_metrics () (*ray.rllib.evaluation.RolloutWorker method*), 383  
get\_metrics () (*ray.util.iter.LocalIterator static method*), 450  
get\_model () (*ray.rllib.models.ModelCatalog static method*), 392  
get\_model () (*ray.util.sgd.tf.TFTrainer method*), 435  
get\_model ()  
    (*ray.util.sgd.torch.TorchTrainer method*), 429  
get\_model\_v2 () (*ray.rllib.models.ModelCatalog static method*), 390  
get\_next\_available\_trial ()  
    (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248  
get\_next\_available\_trial ()  
    (*ray.tune.trial\_executor.TrialExecutor method*), 251  
get\_next\_failed\_trial ()  
    (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248  
get\_next\_failed\_trial ()  
    (*ray.tune.trial\_executor.TrialExecutor method*), 251  
get\_node\_ip () (*ray.rllib.evaluation.RolloutWorker method*), 384  
get\_placeholder ()  
    (*ray.rllib.policy.TFPolicy method*), 365  
get\_policy () (*ray.rllib.evaluation.RolloutWorker method*), 383  
get\_preprocessor ()  
    (*ray.rllib.models.ModelCatalog static method*), 391

get\_preprocessor\_for\_space()  
     (*ray.rllib.models.ModelCatalog static method*),  
     391

get\_resource\_ids() (*in module ray*), 129

get\_running\_trials()  
     (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248

get\_running\_trials()  
     (*ray.tune.trial\_executor.TrialExecutor method*),  
     250

get\_session() (*ray.rllib.policy.TFPolicy method*),  
     365

get\_shard() (*ray.util.iter.ParallelIterator method*),  
     450

get\_state() (*ray.rllib.policy.Policy method*), 360

get\_trial() (*ray.tune.web\_server.TuneClient method*), 255

get\_trial\_checkpoints\_paths()  
     (*ray.tune.Analysis method*), 222

get\_unwrapped() (*ray.rllib.env.BaseEnv method*),  
     372

get\_unwrapped() (*ray.rllib.env.VectorEnv method*),  
     377

get\_webui\_url() (*in module ray*), 129

get\_weights() (*ray.experimental.tf\_utils.TensorFlowVariables method*), 54

get\_weights() (*ray.rllib.evaluation.EvaluatorInterface method*), 385

get\_weights() (*ray.rllib.evaluation.RolloutWorker method*), 382

get\_weights() (*ray.rllib.policy.Policy method*), 360

get\_weights() (*ray.rllib.policy.TFPolicy method*),  
     367

get\_weights() (*ray.rllib.policy.TorchPolicy method*),  
     363

gradients() (*ray.rllib.policy.TFPolicy method*), 367

grid\_search() (*in module ray.tune*), 226

**H**

has\_gpus() (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 249

has\_gpus() (*ray.tune.trial\_executor.TrialExecutor method*), 251

has\_pending\_agent\_data()  
     (*ray.rllib.evaluation.MultiAgentSampleBatchBuilder method*), 386

has\_resources() (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248

has\_resources() (*ray.tune.trial\_executor.TrialExecutor method*), 250

HyperBandScheduler (*class in ray.tune.schedulers*),  
     238

HyperOptSearch (*class in ray.tune.suggest.hyperopt*),  
     231

**I**

import\_model\_from\_h5() (*ray.rllib.policy.Policy method*), 361

import\_model\_from\_h5()  
     (*ray.rllib.policy.TFPolicy method*), 367

import\_model\_from\_h5()  
     (*ray.rllib.policy.TorchPolicy method*), 364

init() (*in module ray*), 124

InputReader (*class in ray.rllib.offline*), 342

inputs (*ray.rllib.models.ActionDistribution attribute*),  
     389

is\_finished() (*ray.tune.suggest.SearchAlgorithm method*), 236

is\_initialized() (*in module ray*), 126

is\_recurrent() (*ray.rllib.policy.Policy method*), 360

is\_recurrent() (*ray.rllib.policy.TFPolicy method*),  
     367

is\_recurrent() (*ray.rllib.policy.TorchPolicy method*), 363

iteration() (*ray.tune.Trainable property*), 214

**J**

JsonLogger (*class in ray.tune.logger*), 245

JupyterNotebookReporter (*class in ray.tune*),  
     218

**K**

kill() (*in module ray*), 129

kl() (*ray.rllib.models.ActionDistribution method*), 389

**L**

last\_action\_for()  
     (*ray.rllib.evaluation.MultiAgentEpisode method*), 380

last\_info\_for() (*ray.rllib.evaluation.MultiAgentEpisode method*), 380

last\_observation\_for()  
     (*ray.rllib.evaluation.MultiAgentEpisode method*), 380

last\_pi\_info\_for()  
     (*ray.rllib.evaluation.MultiAgentEpisode method*), 381

last\_raw\_obs\_for()  
     (*ray.rllib.evaluation.MultiAgentEpisode method*), 380

learn\_on\_batch() (*ray.rllib.evaluation.EvaluatorInterface method*), 385

learn\_on\_batch() (*ray.rllib.evaluation.RolloutWorker method*), 383

learn\_on\_batch() (*ray.rllib.policy.Policy method*),  
     359

learn\_on\_batch() (*ray.rllib.policy.TFPolicy method*), 366

learn\_on\_batch() (*ray.rllib.policy.TorchPolicy method*), 362  
length (*ray.rllib.evaluation.MultiAgentEpisode attribute*), 379  
LinearSchedule (*class in ray.rllib.utils*), 402  
load() (*ray.util.sgd.torch.TorchTrainer method*), 430  
load\_state\_dict()  
    (*ray.util.sgd.torch.TrainingOperator method*), 433  
local\_dir (*ray.tune.trial.Trial attribute*), 253  
local\_shuffle()  
    (*ray.util.iter.ParallelIterator method*), 448  
LocalIterator (*class in ray.util.iter*), 450  
LocalMultiGPUOptimizer (*class in ray.rllib.optimizers*), 397  
log() (*in module ray.tune.track*), 215  
log\_action()  
    (*ray.rllib.env.ExternalEnv method*), 375  
log\_action()  
    (*ray.rllib.env.ExternalMultiAgentEnv method*), 376  
log\_action()  
    (*ray.rllib.env.policy\_client.PolicyClient method*), 288  
log\_action()  
    (*ray.rllib.env.PolicyClient method*), 378  
log\_action()  
    (*ray.rllib.utils.PolicyClient method*), 402  
log\_returns()  
    (*ray.rllib.env.ExternalEnv method*), 375  
log\_returns()  
    (*ray.rllib.env.ExternalMultiAgentEnv method*), 376  
log\_returns()  
    (*ray.rllib.env.policy\_client.PolicyClient method*), 288  
log\_returns()  
    (*ray.rllib.env.PolicyClient method*), 378  
log\_returns()  
    (*ray.rllib.utils.PolicyClient method*), 402  
logdir (*ray.tune.trial.Trial attribute*), 253  
logdir() (*ray.tune.Trainable property*), 214  
Logger (*class in ray.tune.logger*), 244  
logp()  
    (*ray.rllib.models.ActionDistribution method*), 389  
loguniform()  
    (*in module ray.tune*), 225  
loss()  
    (*ray.rllib.models.Model method*), 392  
loss\_initialized()  
    (*ray.rllib.policy.TFPolicy method*), 365  
lstm()  
    (*in module ray.rllib.utils*), 401

**M**

MedianStoppingRule (*class in ray.tune.schedulers*), 240  
merge\_dicts()  
    (*in module ray.rllib.utils*), 403  
method()  
    (*in module ray*), 130  
metric()  
    (*ray.tune.suggest.Searcher property*), 238

metrics()  
    (*ray.rllib.models.tf.tf\_modelv2.TFModelV2 method*), 302  
metrics()  
    (*ray.rllib.models.torch.torch\_modelv2.TorchModelV2 method*), 306  
MicrobatchOptimizer (*class in ray.rllib.optimizers*), 397  
MLFLowLogger (*class in ray.tune.logger*), 245  
mode()  
    (*ray.tune.suggest.Searcher property*), 238  
Model (*class in ray.rllib.models*), 392  
model (*ray.rllib.models.ActionDistribution attribute*), 389  
model (*ray.rllib.policy.TFPolicy attribute*), 364  
model (*ray.rllib.policy.TorchPolicy attribute*), 361  
model()  
    (*ray.util.sgd.torch.TrainingOperator property*), 433  
ModelCatalog (*class in ray.rllib.models*), 390  
models()  
    (*ray.util.sgd.torch.TrainingOperator property*), 433  
module  
    ray.experimental, 132  
    ray.rllib.env, 370  
    ray.rllib.evaluation, 379  
    ray.rllib.models, 389  
    ray.rllib.optimizers, 393  
    ray.rllib.policy, 357  
    ray.rllib.utils, 398  
    ray.tune.integration.keras, 216  
    ray.tune.track, 215  
    ray.util.iter, 445  
    multi\_entropy()  
        (*ray.rllib.models.ActionDistribution method*), 389  
    multi\_kl()  
        (*ray.rllib.models.ActionDistribution method*), 389  
MultiAgentBatch (*class in ray.rllib.evaluation*), 386  
MultiAgentEnv (*class in ray.rllib.env*), 372  
MultiAgentEpisode (*class in ray.rllib.evaluation*), 379  
MultiAgentSampleBatchBuilder (*class in ray.rllib.evaluation*), 386

**N**

NevergradSearch (*class in ray.tune.suggest.nevergrad*), 232  
new\_batch\_builder()  
    (*ray.rllib.evaluation.MultiAgentEpisode attribute*), 379  
next()  
    (*ray.rllib.env.policy\_server\_input.PolicyServerInput method*), 289  
next()  
    (*ray.rllib.env.PolicyServerInput method*), 379  
next()  
    (*ray.rllib.offline.InputReader method*), 342  
next\_trials()  
    (*ray.tune.suggest.SearchAlgorithm method*), 236  
nodes()  
    (*in module ray*), 131  
num\_envs (*ray.rllib.env.VectorEnv attribute*), 377

num\_shards () (*ray.util.iter.ParallelIterator method*),  
     450  
 num\_state\_tensors ()     (*ray.rllib.policy.Policy method*), 360  
 num\_state\_tensors ()     (*ray.rllib.policy.TFPolicy method*), 367  
 num\_state\_tensors ()     (*ray.rllib.policy.TorchPolicy method*), 363  
 num\_steps\_sampled  
     (*ray.rllib.optimizers.PolicyOptimizer attribute*),  
     393  
 num\_steps\_trained  
     (*ray.rllib.optimizers.PolicyOptimizer attribute*),  
     393

**O**

object\_transfer\_timeline () (*in module ray*),  
     131  
 objects () (*in module ray*), 131  
 observation\_space     (*ray.rllib.env.BaseEnv attribute*), 370  
 observation\_space     (*ray.rllib.env.ExternalEnv attribute*), 374  
 observation\_space     (*ray.rllib.env.VectorEnv attribute*), 377  
 observation\_space     (*ray.rllib.policy.Policy attribute*), 357  
 observation\_space     (*ray.rllib.policy.TFPolicy attribute*), 364  
 observation\_space     (*ray.rllib.policy.TorchPolicy attribute*), 361  
 on\_episode\_end () (*ray.rllib.agents.callbacks.DefaultCallbacks method*), 279  
 on\_episode\_start ()  
     (*ray.rllib.agents.callbacks.DefaultCallbacks method*), 279  
 on\_episode\_step ()  
     (*ray.rllib.agents.callbacks.DefaultCallbacks method*), 279  
 on\_global\_var\_update () (*ray.rllib.policy.Policy method*), 361  
 on\_postprocess\_trajectory ()  
     (*ray.rllib.agents.callbacks.DefaultCallbacks method*), 279  
 on\_sample\_end () (*ray.rllib.agents.callbacks.DefaultCallbacks method*), 280  
 on\_step\_begin () (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248  
 on\_step\_begin () (*ray.tune.trial\_executor.TrialExecutor method*), 250  
 on\_step\_end () (*ray.tune.trial\_executor.TrialExecutor method*), 251  
 on\_train\_result ()  
     (*ray.rllib.agents.callbacks.DefaultCallbacks*

*method*), 280  
 on\_trial\_add () (*ray.tune.schedulers.TrialScheduler method*), 242  
 on\_trial\_complete ()  
     (*ray.tune.schedulers.TrialScheduler method*),  
     242  
 on\_trial\_complete ()  
     (*ray.tune.suggest.SearchAlgorithm method*),  
     236  
 on\_trial\_complete ()     (*ray.tune.suggest.Searcher method*), 237  
 on\_trial\_error () (*ray.tune.schedulers.TrialScheduler method*), 242  
 on\_trial\_remove ()  
     (*ray.tune.schedulers.TrialScheduler method*),  
     242  
 on\_trial\_result ()  
     (*ray.tune.schedulers.TrialScheduler method*),  
     242  
 on\_trial\_result ()  
     (*ray.tune.suggest.SearchAlgorithm method*),  
     236  
 on\_trial\_result ()     (*ray.tune.suggest.Searcher method*), 237  
 one\_hot () (*in module ray.rllib.utils*), 400  
 optimizer () (*ray.rllib.policy.TFPolicy method*), 367  
 optimizer () (*ray.rllib.policy.TorchPolicy method*),  
     364  
 optimizer ()     (*ray.util.sgd.torch.TrainingOperator property*), 433  
 optimizers ()     (*ray.util.sgd.torch.TrainingOperator property*), 433  
 OutputWriter (*class in ray.rllib.offline*), 343  
 override () (*in module ray.rllib.utils*), 398

**P**

par\_iter\_init () (*ray.util.iter.ParallelIteratorWorker method*), 451  
 par\_iter\_next () (*ray.util.iter.ParallelIteratorWorker method*), 451  
 par\_iter\_slice () (*ray.util.iter.ParallelIteratorWorker method*), 451  
 ParallelIterator (*class in ray.util.iter*), 446  
 ParallelIteratorWorker (*class in ray.util.iter*),  
     451  
 PAUSE (*ray.tune.schedulers.TrialScheduler attribute*),  
     242  
 pause\_trial () (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248  
 pause\_trial () (*ray.tune.trial\_executor.TrialExecutor method*), 250  
 PiecewiseSchedule (*class in ray.rllib.utils*), 402  
 placeholders (*ray.experimental.tf\_utils.TensorFlowVariables attribute*), 54

Policy (*class in ray.rllib.policy*), 357  
policy\_for () (*ray.rllib.evaluation.MultiAgentEpisode method*), 380  
PolicyClient (*class in ray.rllib.env*), 378  
PolicyClient (*class in ray.rllib.env.policy\_client*), 287  
PolicyClient (*class in ray.rllib.utils*), 401  
PolicyEvaluator (*in module ray.rllib.evaluation*), 384  
PolicyGraph (*in module ray.rllib.evaluation*), 386  
PolicyOptimizer (*class in ray.rllib.optimizers*), 393  
PolicyServer (*class in ray.rllib.utils*), 402  
PolicyServerInput (*class in ray.rllib.env*), 378  
PolicyServerInput (*class in ray.rllib.env.policy\_server\_input*), 288  
poll () (*ray.rllib.env.BaseEnv method*), 371  
PolynomialSchedule (*class in ray.rllib.utils*), 402  
PopulationBasedTraining (*class in ray.tune.schedulers*), 240  
port\_forward (*ray.tune.web\_server.TuneClient attribute*), 255  
postprocess\_batch\_so\_far ()  
    (*ray.rllib.evaluation.MultiAgentSampleBatchBuilder method*), 387  
postprocess\_trajectory ()  
    (*ray.rllib.policy.Policy method*), 359  
Preprocessor (*class in ray.rllib.models*), 392  
prev\_action\_for ()  
    (*ray.rllib.evaluation.MultiAgentEpisode method*), 380  
prev\_reward\_for ()  
    (*ray.rllib.evaluation.MultiAgentEpisode method*), 380  
profile () (*in module ray*), 130  
ProgressReporter (*class in ray.tune*), 219  
PublicAPI () (*in module ray.rllib.utils*), 398  
PublicAPI () (*in module ray.rllib.utils.annotations*), 404  
put () (*in module ray*), 128

**R**

randn () (*in module ray.tune*), 225  
ray.experimental  
    module, 132  
ray.rllib.env  
    module, 370  
ray.rllib.evaluation  
    module, 379  
ray.rllib.models  
    module, 389  
ray.rllib.optimizers  
    module, 393  
ray.rllib.policy  
    module, 357

ray.rllib.utils  
    module, 398  
ray.tune.integration.keras  
    module, 216  
ray.tune.track  
    module, 215  
ray.util.iter  
    module, 445  
ray-attach command line option  
    -N, 138  
    --cluster-name <cluster\_name>, 138  
    --new, 138  
    --port-forward <port\_forward>, 138  
    --screen, 138  
    --start, 138  
    --tmux, 138  
    -n, 138  
    -p, 138  
    CLUSTER\_CONFIG\_FILE, 138  
ray-down command line option  
    --cluster-name <cluster\_name>, 136  
    --keep-min-workers, 136  
    --workers-only, 136  
    --yes, 136  
    -n, 136  
    -y, 136  
    CLUSTER\_CONFIG\_FILE, 136  
ray-exec command line option  
    --cluster-name <cluster\_name>, 136  
    --docker, 136  
    --port-forward <port\_forward>, 136  
    --screen, 136  
    --start, 136  
    --stop, 136  
    --tmux, 136  
    -n, 136  
    -p, 136  
    CLUSTER\_CONFIG\_FILE, 137  
    CMD, 137  
ray-get\_head\_ip command line option  
    --cluster-name <cluster\_name>, 139  
    -n, 139  
    CLUSTER\_CONFIG\_FILE, 139  
ray-globalgc command line option  
    --address <address>, 139  
ray-memory command line option  
    --address <address>, 139  
ray-start command line option  
    --address <address>, 133  
    --autoscaling-config  
        <autoscaling\_config>, 134  
    --block, 134  
    --head, 134  
    --huge-pages, 134

```
--include-java, 134
--include-webui <include_webui>, 134
--internal-config
    <internal_config>, 134
--java-worker-options
    <java_worker_options>, 134
--load-code-from-local, 134
--memory <memory>, 133
--no-redirect-output, 134
--no-redirect-worker-output, 134
--node-ip-address
    <node_ip_address>, 133
--node-manager-port
    <node_manager_port>, 133
--num-cpus <num_cpus>, 134
--num-gpus <num_gpus>, 134
--num-redis-shards
    <num_redis_shards>, 133
--object-manager-port
    <object_manager_port>, 133
--object-store-memory
    <object_store_memory>, 133
--plasma-directory
    <plasma_directory>, 134
--plasma-store-socket-name
    <plasma_store_socket_name>, 134
--raylet-socket-name
    <raylet_socket_name>, 134
--redis-address <redis_address>, 133
--redis-max-clients
    <redis_max_clients>, 133
--redis-max-memory
    <redis_max_memory>, 133
--redis-password <redis_password>,
    133
--redis-port <redis_port>, 133
--redis-shard-ports
    <redis_shard_ports>, 133
--resources <resources>, 134
--temp-dir <temp_dir>, 134
--webui-host <webui_host>, 134
ray-stat command line option
    --address <address>, 139
ray-stop command line option
    --force, 135
    --verbose, 135
    -f, 135
    -v, 135
ray-submit command line option
    --args <args>, 137
    --cluster-name <cluster_name>, 137
    --docker, 137
    --port-forward <port_forward>, 137
    --screen, 137
    --start, 137
    --stop, 137
    --tmux, 137
    -n, 137
    -p, 137
    CLUSTER_CONFIG_FILE, 138
    SCRIPT, 138
    SCRIPT_ARGS, 138
ray-timeline command line option
    --address <address>, 140
ray-up command line option
    --cluster-name <cluster_name>, 135
    --max-workers <max_workers>, 135
    --min-workers <min_workers>, 135
    --no-restart, 135
    --restart-only, 135
    --yes, 135
    -n, 135
    -y, 135
    CLUSTER_CONFIG_FILE, 135
RayTrialExecutor      (class)      in
    ray.tune.ray_trial_executor), 247
RecurrentTFModelV2     (class)      in
    ray.rllib.models.tf.recurrent_tf_modelv2),
    303
register_custom_action_dist()
    (ray.rllib.models.ModelCatalog static method),
    391
register_custom_model()
    (ray.rllib.models.ModelCatalog static method),
    391
register_custom_preprocessor()
    (ray.rllib.models.ModelCatalog static method),
    391
register_custom_serializer() (in module
    ray), 130
register_env() (in module ray.tune), 254
register_trainable() (in module ray.tune), 254
register_variables()
    (ray.rllib.models.tf.tf_modelv2.TFModelV2
    method), 302
relu() (in module ray.rllib.utils), 400
remote (ray.rllib.env.EnvContext attribute), 377
remote() (in module ray), 126
remote_checkpoint_dir
    (ray.tune.DurableTrainable attribute), 215
renamed_agent() (in module ray.rllib.utils), 399
renamed_class() (in module ray.rllib.utils), 399
renamed_function() (in module ray.rllib.utils), 399
repartition() (ray.util.iter.ParallelIterator method),
    448
Repeater (class in ray.tune.suggest), 226
report() (ray.tune.ProgressReporter method), 219
required_model_output_shape()
```

```

        (ray.rllib.models.ActionDistribution      static sample () (ray.rllib.evaluation.RolloutWorker method),
method), 389                         382
reset () (ray.rllib.env.MultiAgentEnv method), 373 sample () (ray.rllib.models.ActionDistribution
reset () (ray.rllib.optimizers.AsyncReplayOptimizer method), 395 method), 389
reset () (ray.rllib.optimizers.AsyncSamplesOptimizer method), 395 sample_and_learn()
reset () (ray.rllib.optimizers.PolicyOptimizer method), 394 (ray.rllib.evaluation.RolloutWorker method),
383
reset_at () (ray.rllib.env.VectorEnv method), 377 sample_from (class in ray.tune), 225
reset_config () (ray.tune.Trainable method), 213 sample_with_count ()
reset_trial () (ray.tune.ray_trial_executor.RayTrialExecutor (ray.rllib.evaluation.RolloutWorker method),
method), 248 382
reset_trial () (ray.tune.trial_executor.TrialExecutor exampleBatch (class in ray.rllib.evaluation), 388
method), 250 SampleBatchBuilder (class in ray.rllib.evaluation),
386
resource_help () (ray.tune.Trainable class method), sampled_action_logp ()
213 (ray.rllib.models.ActionDistribution method),
389
resource_string () save () (ray.rllib.optimizers.PolicyOptimizer method),
(ray.tune.ray_trial_executor.RayTrialExecutor 393
method), 248 save () (ray.tune.ray_trial_executor.RayTrialExecutor
method), 248
resource_string () save () (ray.tune.suggest.Searcher method), 238
(ray.tune.trial_executor.TrialExecutor method), 251 save () (ray.tune.Trainable method), 213
251 save () (ray.tune.trial_executor.TrialExecutor method),
251 save () (ray.util.sgd.tf.TFTrainer method), 435
Resources (class in ray.tune.resources), 254 save () (ray.util.sgd.torch.TorchTrainer method), 430
resources (ray.tune.trial.Trial attribute), 253 save_to_object () (ray.tune.Trainable method), 213
restore () (ray.rllib.optimizers.PolicyOptimizer scheduler () (ray.util.sgd.torch.TrainingOperator
method), 393 property), 433
restore () (ray.tune.ray_trial_executor.RayTrialExecutor schedulers () (ray.util.sgd.torch.TrainingOperator
method), 249 property), 433
restore () (ray.tune.suggest.Searcher method), 238 SCRIPT
restore () (ray.tune.Trainable method), 213 ray-submit command line option, 138
restore () (ray.tune.trial_executor.TrialExecutor SCRIPT_ARGS
method), 251 ray-submit command line option, 138
resume_trial () (ray.tune.trial_executor.TrialExecutor SearchAlgorithm (class in ray.tune.suggest), 236
method), 250 Searcher (class in ray.tune.suggest), 237
rnn_state_for () (ray.rllib.evaluation.MultiAgentEpisode end_actions () (ray.rllib.env.BaseEnv method), 372
method), 380 sess (ray.experimental.tf_utils.TensorFlowVariables
attribute), 54
RolloutWorker (class in ray.rllib.evaluation), 381 set_finished () (ray.tune.suggest.SearchAlgorithm
rows () (ray.rllib.evaluation.SampleBatch method), 388 method), 236
run () (in module ray.tune), 203 set_flat () (ray.experimental.tf_utils.TensorFlowVariables
run () (ray.rllib.env.ExternalEnv method), 374 method), 54
run () (ray.rllib.env.ExternalMultiAgentEnv method), 375 set_resource () (in module ray.experimental), 132
run () (ray.rllib.evaluation.AsyncSampler method), 387 set_session () (ray.experimental.tf_utils.TensorFlowVariables
run_experiments () (in module ray.tune), 205 method), 54
runner_data () (ray.tune.ExperimentAnalysis set_state () (ray.rllib.policy.Policy method), 360
method), 221 set_status () (ray.tune.trial_executor.TrialExecutor
method), 249
S set_weights () (ray.experimental.tf_utils.TensorFlowVariables
sample () (ray.rllib.evaluation.EvaluatorInterface method), 384 method), 55
set_weights () (ray.rllib.evaluation.EvaluatorInterface

```

*method), 386*  
*set\_weights() (ray.rllib.evaluation.RolloutWorker method), 382*  
*set\_weights() (ray.rllib.policy.Policy method), 360*  
*set\_weights() (ray.rllib.policy.TFPolicy method), 367*  
*set\_weights() (ray.rllib.policy.TorchPolicy method), 363*  
*setup() (ray.util.sgd.torch.TrainingOperator method), 431*  
*setup\_torch\_data\_parallel() (ray.rllib.evaluation.RolloutWorker method), 384*  
*shape (ray.rllib.models.Preprocessor attribute), 392*  
*shards() (ray.util.iter.ParallelIterator method), 450*  
*should\_report() (ray.tune.ProgressReporter method), 219*  
*show() (ray.util.iter.LocalIterator method), 451*  
*show() (ray.util.iter.ParallelIterator method), 450*  
*shuffle() (ray.rllib.evaluation.SampleBatch method), 388*  
*shuffle() (ray.util.iter.LocalIterator method), 450*  
*shutdown() (in module ray), 129*  
*shutdown() (in module ray.tune.track), 215*  
*shutdown() (ray.util.sgd.tf.TFTrainer method), 435*  
*shutdown() (ray.util.sgd.torch.TorchTrainer method), 430*  
*sigmoid() (in module ray.rllib.utils), 400*  
*SigOptSearch (class in ray.tune.suggest.sigopt), 233*  
*SkOptSearch (class in ray.tune.suggest.skopt), 234*  
*slice() (ray.rllib.evaluation.SampleBatch method), 388*  
*soft\_reset() (ray.rllib.evaluation.MultiAgentEpisode method), 380*  
*softmax() (in module ray.rllib.utils), 400*  
*split\_by\_episode() (ray.rllib.evaluation.SampleBatch method), 388*  
*start\_episode() (ray.rllib.env.ExternalEnv method), 374*  
*start\_episode() (ray.rllib.env.ExternalMultiAgentEnv method), 376*  
*start\_episode() (ray.rllib.env.policy\_client.PolicyClient method), 287*  
*start\_episode() (ray.rllib.env.PolicyClient method), 378*  
*start\_episode() (ray.rllib.utils.PolicyClient method), 401*  
*start\_trial() (ray.tune.ray\_trial\_executor.RayTrialExecutor method), 247*  
*start\_trial() (ray.tune.trial\_executor.TrialExecutor method), 250*  
*state\_dict() (ray.util.sgd.torch.TrainingOperator method), 433*  
*stats() (ray.rllib.optimizers.AsyncGradientsOptimizer method), 395*  
*stats() (ray.rllib.optimizers.AsyncReplayOptimizer method), 395*  
*stats() (ray.rllib.optimizers.AsyncSamplesOptimizer method), 395*  
*stats() (ray.rllib.optimizers.LocalMultiGPUOptimizer method), 397*  
*stats() (ray.rllib.optimizers.MicrobatchOptimizer method), 397*  
*stats() (ray.rllib.optimizers.PolicyOptimizer method), 393*  
*stats() (ray.rllib.optimizers.SyncBatchReplayOptimizer method), 397*  
*stats() (ray.rllib.optimizers.SyncReplayOptimizer method), 396*  
*stats() (ray.rllib.optimizers.SyncSamplesOptimizer method), 396*  
*stats() (ray.rllib.optimizers.TorchDistributedDataParallelOptimizer method), 398*  
*stats() (ray.tune.ExperimentAnalysis method), 221*  
*status (ray.tune.trial.Trial attribute), 253*  
*StatusReporter (class in ray.tune.function\_runner), 216*  
*step() (ray.rllib.env.MultiAgentEnv method), 373*  
*step() (ray.rllib.optimizers.AsyncGradientsOptimizer method), 395*  
*step() (ray.rllib.optimizers.AsyncReplayOptimizer method), 394*  
*step() (ray.rllib.optimizers.AsyncSamplesOptimizer method), 395*  
*step() (ray.rllib.optimizers.LocalMultiGPUOptimizer method), 397*  
*step() (ray.rllib.optimizers.MicrobatchOptimizer method), 397*  
*step() (ray.rllib.optimizers.PolicyOptimizer method), 393*  
*step() (ray.rllib.optimizers.SyncBatchReplayOptimizer method), 396*  
*step() (ray.rllib.optimizers.SyncReplayOptimizer method), 396*  
*step() (ray.rllib.optimizers.SyncSamplesOptimizer method), 396*  
*step() (ray.rllib.optimizers.TorchDistributedDataParallelOptimizer method), 398*  
*STOP (ray.tune.schedulers.TrialScheduler attribute), 242*  
*stop() (ray.rllib.env.BaseEnv method), 372*  
*stop() (ray.rllib.optimizers.AsyncReplayOptimizer method), 395*  
*stop() (ray.rllib.optimizers.AsyncSamplesOptimizer method), 395*  
*stop() (ray.rllib.optimizers.PolicyOptimizer method), 393*  
*stop() (ray.tune.Trainable method), 213*

stop\_all() (*ray.tune.Stopper method*), 207  
 stop\_experiment()  
     (*ray.tune.web\_server.TuneClient method*), 255  
 stop\_trial() (*ray.tune.ray\_trial\_executor.RayTrialExecutor method*), 248  
 stop\_trial() (*ray.tune.trial\_executor.TrialExecutor method*), 250  
 stop\_trial() (*ray.tune.web\_server.TuneClient method*), 255  
 Stopper (*class in ray.tune*), 206  
 storage\_client (*ray.tune.DurableTrainable attribute*), 215  
 suggest() (*ray.tune.suggest.Searcher method*), 238  
 sync() (*ray.rllib.utils.Filter method*), 400  
 sync\_filters() (*ray.rllib.evaluation.RolloutWorker method*), 384  
 SyncBatchReplayOptimizer (*class in ray.rllib.optimizers*), 396  
 synchronize() (*ray.rllib.utils.FilterManager static method*), 399  
 SyncReplayOptimizer (*class in ray.rllib.optimizers*), 396  
 SyncSampler (*class in ray.rllib.evaluation*), 387  
 SyncSamplesOptimizer (*class in ray.rllib.optimizers*), 396

**T**

take() (*ray.util.iter.LocalIterator method*), 450  
 take() (*ray.util.iter.ParallelIterator method*), 450  
 TBXLogger (*class in ray.tune.logger*), 245  
 TensorFlowVariables (*class in ray.experimental.tf\_utils*), 54  
 tf\_input\_ops() (*ray.rllib.offline.InputReader method*), 342  
 TFModelV2 (*class in ray.rllib.models.tf\_tf\_modelv2*), 300  
 TFPolicy (*class in ray.rllib.policy*), 364  
 TFPolicyGraph (*in module ray.rllib.evaluation*), 386  
 TFTTrainer (*class in ray.util.sgd.tf*), 435  
 timeline() (*in module ray*), 131  
 to\_base\_env() (*ray.rllib.env.BaseEnv static method*), 371  
 TorchDistributedDataParallelOptimizer (*class in ray.rllib.optimizers*), 398  
 TorchModelV2 (*class in ray.rllib.models.torch.torch\_modelv2*), 305  
 TorchPolicy (*class in ray.rllib.policy*), 361  
 TorchPolicyGraph (*in module ray.rllib.evaluation*), 386  
 TorchTrainer (*class in ray.util.sgd.torch*), 426  
 total() (*ray.rllib.evaluation.MultiAgentSampleBatchBuilder method*), 386

total\_reward (*ray.rllib.evaluation.MultiAgentEpisode attribute*), 379  
 train() (*ray.tune.Trainable method*), 213  
 train() (*ray.util.sgd.tf.TFTrainer method*), 435  
 train\_in() (*ray.util.sgd.torch.TorchTrainer method*), 428  
 train\_batch() (*ray.util.sgd.torch.TrainingOperator method*), 432  
 train\_epoch() (*ray.util.sgd.torch.TrainingOperator method*), 431  
 train\_loader() (*ray.util.sgd.torch.TrainingOperator property*), 433  
 Trainable (*class in ray.tune*), 210  
 trainable\_name (*ray.tune.trial.Trial attribute*), 253  
 trainable\_variables()  
     (*ray.rllib.models.tf\_tf\_modelv2.TFModelV2 method*), 302  
 trainer() (*ray.util.sgd.torch.BaseTorchTrainable property*), 435  
 training\_iteration() (*ray.tune.Trainable property*), 214  
 TrainingOperator (*class in ray.util.sgd.torch*), 430  
 transform() (*ray.rllib.models.Preprocessor method*), 392  
 Trial (*class in ray.tune.trial*), 253  
 trial\_dataframes() (*ray.tune.Analysis property*), 222  
 trial\_dir() (*in module ray.tune.track*), 215  
 trial\_id (*ray.tune.trial.Trial attribute*), 253  
 trial\_id() (*in module ray.tune.track*), 215  
 trial\_id() (*ray.tune.Trainable property*), 214  
 trial\_name() (*in module ray.tune.track*), 215  
 trial\_name() (*ray.tune.Trainable property*), 214  
 TrialExecutor (*class in ray.tune.trial\_executor*), 249  
 TrialRunner (*class in ray.tune.trial\_runner*), 252  
 TrialScheduler (*class in ray.tune.schedulers*), 242  
 try\_checkpoint\_metadata()  
     (*ray.tune.trial\_executor.TrialExecutor method*), 249  
 try\_import\_tf() (*in module ray.rllib.utils*), 398  
 try\_import\_tfp() (*in module ray.rllib.utils*), 398  
 try\_import\_torch() (*in module ray.rllib.utils*), 399  
 try\_reset() (*ray.rllib.env.BaseEnv method*), 372  
 tune\_address (*ray.tune.web\_server.TuneClient attribute*), 255  
 TuneBOHB (*class in ray.tune.suggest.bohb*), 229  
 TuneClient (*class in ray.tune.web\_server*), 255

**U**

UnifiedLogger (*class in ray.tune.logger*), 245  
 uniform() (*in module ray.tune*), 225  
 union() (*ray.util.iter.LocalIterator method*), 451  
 union() (*ray.util.iter.ParallelIterator method*), 450  
 unpause\_trial() (*ray.tune.trial\_executor.TrialExecutor method*), 250

update\_ops () (*ray.rllib.models.tf.tf\_modelv2.TFModelV2*.*world\_rank* () (ray.util.sgd.torch.TrainingOperator property), 433  
 update\_scheduler () (ray.util.sgd.torch.TorchTrainer method), 429  
 use\_fp16 () (ray.util.sgd.torch.TrainingOperator property), 433  
 use\_gpu () (ray.util.sgd.torch.TrainingOperator property), 433  
 use\_tqdm () (ray.util.sgd.torch.TrainingOperator property), 434  
 user\_data (*ray.rllib.evaluation.MultiAgentEpisode* attribute), 380

**V**

validate () (ray.util.sgd.tf.TFTrainer method), 435  
 validate () (ray.util.sgd.torch.TorchTrainer method), 429  
 validate () (ray.util.sgd.torch.TrainingOperator method), 432  
 validate\_batch () (ray.util.sgd.torch.TrainingOperator method), 432  
 validation\_loader () (ray.util.sgd.torch.TrainingOperator property), 433  
 value\_function () (*ray.rllib.models.Model* method), 392  
 value\_function () (*ray.rllib.models.tf.tf\_modelv2.TFModelV2* method), 301  
 value\_function () (*ray.rllib.models.torch.torch\_modelv2.TorchModelV2* method), 306  
 variables (*ray.experimental.tf\_utils.TensorFlowVariables* attribute), 54  
 variables () (*ray.rllib.models.tf.tf\_modelv2.TFModelV2* method), 302  
 variables () (ray.rllib.policy.TFPolicy method), 365  
 vector\_index (*ray.rllib.env.EnvContext* attribute), 377  
 vector\_reset () (*ray.rllib.env.VectorEnv* method), 377  
 vector\_step () (*ray.rllib.env.VectorEnv* method), 377  
 VectorEnv (*class in ray.rllib.env*), 376  
 VisionNetwork (*class in ray.rllib.models*), 393

**W**

wait () (*in module ray*), 128  
 wait () (*in module ray.experimental*), 132  
 with\_agent\_groups () (*ray.rllib.env.MultiAgentEnv* method), 373  
 worker\_index (*ray.rllib.env.EnvContext* attribute), 377  
 workers (*ray.rllib.optimizers.PolicyOptimizer* attribute), 393

**Z**

ZooptSearch (*class in ray.tune.suggest.zoopt*), 235