Ray Documentation

Release 0.8.0.dev3

The Ray Team

Aug 22, 2019

Installation

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

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

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

Install Ray with: `pip install ray`. For nightly wheels, see the Installation page.

View the codebase on GitHub.

CHAPTER 1

Quick Start

```
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:

```
ray.init()

@ray.remote
class Counter():
    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))
```

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
```

See more details in the Cluster Launch page.

CHAPTER 2

Tune Quick Start

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

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.

```python
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
```

CHAPTER 3

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,
```

(continues on next page)

(continued from previous page)

```
        "num_workers": 4,
        "env_config": {"corridor_length": 5}})
```

CHAPTER 4

Contact

The following are good places to discuss Ray.

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

4.1 Installing Ray

Ray supports Python 2 and Python 3 as well as MacOS and Linux. Windows support is planned for the future.

4.1.1 Latest stable version

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

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

4.1.2 Trying snapshots from master

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

4.2 Walkthrough

This walkthrough will overview the core concepts of Ray:

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

With Ray, your code will work on a single machine and can be easily scaled to a large cluster. To run this walkthrough, install Ray with `pip install -U ray`.

```
import ray

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

See the Configuration documentation for the various ways to configure Ray. To start a multi-node Ray cluster, see the cluster setup page. You can stop ray by calling `ray.shutdown()`. To check if Ray is initialized, you can call `ray.is_initialized()`.

4.2.1 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 things 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`.

```
>>> regular_function()
1

>>> remote_function.remote()
ObjectID(1c80d6937802cd7786ad25e50caf2f023c95e350)

>>> ray.get(remote_function.remote())
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()
```

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**.

```
>>> y1_id = f.remote(x1_id)
>>> ray.get(y1_id)
1

>>> y2_id = f.remote(x2_id)
>>> ray.get(y2_id)
[1, 2, 3]
```

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 `x1_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.
@ray.remote(resources={'Custom': 1})
def f():
    return 1
```

Further, remote function can return multiple object IDs.

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

a_id, b_id, c_id = return_multiple.remote()
```

4.2.2 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 **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
>>> y_id = ray.put(y)
>>> print(y_id)
ObjectID(0369a14bc595e08cfbd508dfaa162cb7feffffff)
```

Here is the docstring for `ray.put`:

`ray.`**`put`**`(`*value*`)`

Store an object in the object store.

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

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.

4.2.3 Fetching Results

The command `ray.get(x_id)` takes an object ID and creates a Python object from the corresponding remote object. For some objects like arrays, we can use shared memory and avoid copying the object.

```
>>> y = 1
>>> obj_id = ray.put(y)
>>> print(obj_id)
ObjectID(0369a14bc595e08cfbd508dfaa162cb7feffffff)
>>> ray.get(obj_id)
1
```

Here is the docstring for `ray.get`:

`ray.`**`get`**(*object_ids*)

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

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

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

Returns A Python object or a list of Python objects.

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

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)
```

Here is the docstring for `ray.wait`:

`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.

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

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

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

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

Parameters

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

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

4.2.4 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.

4.3 How-to: 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.

4.3.1 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.

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
```

4.3.2 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.

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

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

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

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

4.3.3 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.

4.3.4 Passing Around Actor Handles

Actor handles can be passed into other tasks. To see an example of this, take a look at the asynchronous parameter server example. To illustrate this with a simple example, consider a simple actor definition.

```
@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()))
```

4.4 How-to: Using Ray with GPUs

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

4.4.1 Starting Ray with GPUs

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

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

If you don't pass in the `num_gpus` argument, Ray will automatically detect the number of GPUs available.

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

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

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

4.4.2 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.

4.4.3 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)

# 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.

4.4.4 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()
```

4.4.5 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()
```

4.5 How-to: Profile Ray Programs

Profiling the performance of your code can be very helpful to determine performance bottlenecks or to find out where your code may not be parallelized properly.

4.5.1 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`.

4.5.2 A Basic Example to Profile

Let's try to profile a simple example, and compare how different ways to write a simple loop can affect performance.

As a proxy for a computationally intensive and possibly slower function, let's define our remote function to just sleep for 0.5 seconds:

```
import ray
import time

# Our time-consuming remote function
@ray.remote
def func():
    time.sleep(0.5)
```

In our example setup, we wish to call our remote function `func()` five times, and store the result of each call into a list. To compare the performance of different ways of looping our calls to our remote function, we can define each loop version as a separate function on the driver script.

For the first version **ex1**, each iteration of the loop calls the remote function, then calls `ray.get` in an attempt to store the current result into the list, as follows:

```
# This loop is suboptimal in Ray, and should only be used for the sake of this example
def ex1():
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))
```

For the second version **ex2**, each iteration of the loop calls the remote function, and stores it into the list **without** calling `ray.get` each time. `ray.get` is used after the loop has finished, in preparation for processing `func()`'s results:

```
# This loop is more proper in Ray
def ex2():
    list2 = []
    for i in range(5):
        list2.append(func.remote())
    ray.get(list2)
```

Finally, for an example that's not so parallelizable, let's create a third version **ex3** where the driver has to call a local function in between each call to the remote function `func()`:

```
# A local function executed on the driver, not on Ray
def other_func():
    time.sleep(0.3)

def ex3():
    list3 = []
    for i in range(5):
        other_func()
        list3.append(func.remote())
    ray.get(list3)
```

4.5.3 Timing Performance Using Python's Timestamps

One way to sanity-check the performance of the three loops is simply to time how long it takes to complete each loop version. We can do this using python's built-in `time` module.

The `time` module contains a useful `time()` function that returns the current timestamp in unix time whenever it's called. We can create a generic function wrapper to call `time()` right before and right after each loop function to print out how long each loop takes overall:

```
# This is a generic wrapper for any driver function you want to time
def time_this(f):
    def timed_wrapper(*args, **kw):
        start_time = time.time()
        result = f(*args, **kw)
        end_time = time.time()

        # Time taken = end_time - start_time
        print('| func:%r args:[%r, %r] took: %2.4f seconds |' % \
              (f.__name__, args, kw, end_time - start_time))
        return result
    return timed_wrapper
```

To always print out how long the loop takes to run each time the loop function `ex1()` is called, we can evoke our `time_this` wrapper with a function decorator. This can similarly be done to functions `ex2()` and `ex3()`:

```
@time_this  # Added decorator
def ex1():
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))

def main():
    ray.init()
    ex1()
    ex2()
    ex3()

if __name__ == "__main__":
    main()
```

Then, running the three timed loops should yield output similar to this:

```
| func:'ex1' args:[(), {}] took: 2.5083 seconds |
| func:'ex2' args:[(), {}] took: 1.0032 seconds |
| func:'ex3' args:[(), {}] took: 2.0039 seconds |
```

Let's interpret these results.

Here, `ex1()` took substantially more time than `ex2()`, where their only difference is that `ex1()` calls `ray.get` on the remote function before adding it to the list, while `ex2()` waits to fetch the entire list with `ray.get` at once.

```
@ray.remote
def func(): # A single call takes 0.5 seconds
    time.sleep(0.5)

def ex1():  # Took Ray 2.5 seconds
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))

def ex2():  # Took Ray 1 second
    list2 = []
    for i in range(5):
```

(continues on next page)

(continued from previous page)

```
        list2.append(func.remote())
    ray.get(list2)
```

Notice how `ex1()` took 2.5 seconds, exactly five times 0.5 seconds, or the time it would take to wait for our remote function five times in a row.

By calling `ray.get` after each call to the remote function, `ex1()` removes all ability to parallelize work, by forcing the driver to wait for each `func()`'s result in succession. We are not taking advantage of Ray parallelization here!

Meanwhile, `ex2()` takes about 1 second, much faster than it would normally take to call `func()` five times iteratively. Ray is running each call to `func()` in parallel, saving us time.

`ex1()` is actually a common user mistake in Ray. `ray.get` is not necessary to do before adding the result of `func()` to the list. Instead, the driver should send out all parallelizable calls to the remote function to Ray before waiting to receive their results with `ray.get`. `ex1()`'s suboptimal behavior can be noticed just using this simple timing test.

Realistically, however, many applications are not as highly parallelizable as `ex2()`, and the application includes sections where the code must run in serial. `ex3()` is such an example, where the local function `other_func()` must run first before each call to `func()` can be submitted to Ray.

```python
# A local function that must run in serial
def other_func():
    time.sleep(0.3)

def ex3():  # Took Ray 2 seconds, vs. ex1 taking 2.5 seconds
    list3 = []
    for i in range(5):
        other_func()
        list2.append(func.remote())
    ray.get(list3)
```

What results is that while `ex3()` still gained 0.5 seconds of speedup compared to the completely serialized `ex1()` version, this speedup is still nowhere near the ideal speedup of `ex2()`.

The dramatic speedup of `ex2()` is possible because `ex2()` is theoretically completely parallelizable: if we were given 5 CPUs, all 5 calls to `func()` can be run in parallel. What is happening with `ex3()`, however, is that each parallelized call to `func()` is staggered by a wait of 0.3 seconds for the local `other_func()` to finish.

`ex3()` is thus a manifestation of Amdahls Law: the fastest theoretically possible execution time from parallelizing an application is limited to be no better than the time it takes to run all serial parts in serial.

Due to Amdahl's Law, `ex3()` must take at least 1.5 seconds – the time it takes for 5 serial calls to `other_func()` to finish! After an additional 0.5 seconds to execute func and get the result, the computation is done.

4.5.4 Profiling Using An External Profiler (Line Profiler)

One way to profile the performance of our code using Ray is to use a third-party profiler such as Line_profiler. Line_profiler is a useful line-by-line profiler for pure Python applications that formats its output side-by-side with the profiled code itself.

Alternatively, another third-party profiler (not covered in this documentation) that you could use is Pyflame, which can generate profiling graphs.

First install `line_profiler` with pip:

```
pip install line_profiler
```

line_profiler requires each section of driver code that you want to profile as its own independent function. Conveniently, we have already done so by defining each loop version as its own function. To tell line_profiler which functions to profile, just add the @profile decorator to ex1(), ex2() and ex3(). Note that you do not need to import line_profiler into your Ray application:

```
@profile  # Added decorator
def ex1():
    list1 = []
    for i in range(5):
        list1.append(ray.get(func.remote()))

def main():
    ray.init()
    ex1()
    ex2()
    ex3()

if __name__ == "__main__":
    main()
```

Then, when we want to execute our Python script from the command line, instead of python your_script_here.py, we use the following shell command to run the script with line_profiler enabled:

```
kernprof -l your_script_here.py
```

This command runs your script and prints only your script's output as usual. Line_profiler instead outputs its profiling results to a corresponding binary file called your_script_here.py.lprof.

To read line_profiler's results to terminal, use this shell command:

```
python -m line_profiler your_script_here.py.lprof
```

In our loop example, this command outputs results for ex1() as follows. Note that execution time is given in units of 1e-06 seconds:

```
Timer unit: 1e-06 s

Total time: 2.50883 s
File: your_script_here.py
Function: ex1 at line 28

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    29                                           @profile
    30                                           def ex1():
    31         1          3.0      3.0      0.0   list1 = []
    32         6         18.0      3.0      0.0   for i in range(5):
    33         5    2508805.0 501761.0    100.0     list1.append(ray.get(func.
↪remote()))
```

Notice that each hit to list1.append(ray.get(func.remote())) at line 33 takes the full 0.5 seconds waiting for func() to finish. Meanwhile, in ex2() below, each call of func.remote() at line 40 only takes 0.127 ms, and the majority of the time (about 1 second) is spent on waiting for ray.get() at the end:

```
Total time: 1.00357 s
File: your_script_here.py
Function: ex2 at line 35
```

(continues on next page)

(continued from previous page)

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    36                                           @profile
    37                                           def ex2():
    38         1          2.0      2.0      0.0   list2 = []
    39         6         13.0      2.2      0.0   for i in range(5):
    40         5        637.0    127.4      0.1     list2.append(func.remote())
    41         1    1002919.0 1002919.0     99.9     ray.get(list2)
```

And finally, `line_profiler`'s output for `ex3()`. Each call to `func.remote()` at line 50 still take magnitudes faster than 0.5 seconds, showing that Ray is successfully parallelizing the remote calls. However, each call to the local function `other_func()` takes the full 0.3 seconds, totalling up to the guaranteed minimum application execution time of 1.5 seconds:

```
Total time: 2.00446 s
File: basic_kernprof.py
Function: ex3 at line 44

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    44                                           @profile
    45                                           #@time_this
    46                                           def ex3():
    47         1          2.0      2.0      0.0   list3 = []
    48         6         13.0      2.2      0.0   for i in range(5):
    49         5    1501934.0 300386.8     74.9     other_func()
    50         5        917.0    183.4      0.0     list3.append(func.remote())
    51         1     501589.0 501589.0     25.0   ray.get(list3)
```

4.5.5 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()')
```

(continues on next page)

(continued from previous page)

```python

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:

```python
# 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:

```python
def ex4():
    # This is suboptimal in Ray, and should only be used for the sake of this example
```

(continues on next page)

(continued from previous page)

```
    actor_example = Sleeper.remote()

    five_results = []
    for i in range(5):
        five_results.append(actor_example.actor_func.remote())

    # Wait until the end to call ray.get()
    ray.get(five_results)
```

We enable cProfile on this example as follows:

```
def main():
    ray.init()
    cProfile.run('ex4()')

if __name__ == "__main__":
    main()
```

Running our new Actor example, cProfile's abbreviated output is as follows:

```
12519 function calls (11956 primitive calls) in 2.525 seconds

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
...
1     0.000    0.000    0.015    0.015 actor.py:546(remote)
1     0.000    0.000    0.015    0.015 actor.py:560(_submit)
1     0.000    0.000    0.000    0.000 actor.py:697(__init__)
...
1     0.000    0.000    2.525    2.525 your_script_here.py:63(ex4)
...
9     0.000    0.000    0.000    0.000 worker.py:2459(__init__)
1     0.000    0.000    2.509    2.509 worker.py:2535(get)
9     0.000    0.000    0.000    0.000 worker.py:2695(get_global_worker)
4     0.000    0.000    2.508    0.627 worker.py:374(retrieve_and_deserialize)
1     0.000    0.000    2.509    2.509 worker.py:424(get_object)
8     0.000    0.000    0.001    0.000 worker.py:514(submit_task)
...
```

It turns out that the entire example still took 2.5 seconds to execute, or the time for five calls to `actor_func()` to run in serial. We remember in `ex1` that this behavior was because we did not wait until after submitting all five remote function tasks to call `ray.get()`, but we can verify on cProfile's output line `worker.py:2535(get)` that `ray.get()` was only called once at the end, for 2.509 seconds. What happened?

It turns out Ray cannot parallelize this example, because we have only initialized a single `Sleeper` actor. Because each actor is a single, stateful worker, our entire code is submitted and ran on a single worker the whole time.

To better parallelize the actors in `ex4`, we can take advantage that each call to `actor_func()` is independent, and instead create five `Sleeper` actors. That way, we are creating five workers that can run in parallel, instead of creating a single worker that can only handle one call to `actor_func()` at a time.

```
def ex4():
    # Modified to create five separate Sleepers
    five_actors = [Sleeper.remote() for i in range(5)]

    # Each call to actor_func now goes to a different Sleeper
    five_results = []
    for actor_example in five_actors:
```

(continues on next page)

(continued from previous page)

```
        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)
...
```

4.6 How-To: Inspect 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.

4.6.1 Context: Ray Processes

For context, when using Ray, several processes are involved.

- Multiple **worker** processes execute tasks and store results in object stores. Each worker is a separate process.
- One **object store** per node stores immutable objects in shared memory and allows workers to efficiently share objects on the same node with minimal copying and deserialization.
- One **raylet** per node assigns tasks to workers on the same node.
- A **driver** is the Python process that the user controls. For example, if the user is running a script or using a Python shell, then the driver is the Python process that runs the script or the shell. A driver is similar to a worker in that it can submit tasks to its raylet and get objects from the object store, but it is different in that the raylet will not assign tasks to the driver to be executed.
- A **Redis server** maintains much of the system's state. For example, it keeps track of which objects live on which machines and of the task specifications (but not data). It can also be queried directly for debugging purposes.

4.6.2 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()
>>> ray.nodes()
[{'ClientID': 'a9e430719685f3862ed7ba411259d4138f8afb1e',
  '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.

4.6.3 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.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.

4.6.4 Object Information

To get information about the current objects that have been placed in the Ray object store across the cluster, you can use `ray.objects()`.

`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.

4.7 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.

4.7.1 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-cpus` 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 --redis-address=<redis-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(redis_address=<redis-address>)
```

4.7.2 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 subdirectory 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.
```

4.7.3 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.

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.

See the Redis security documentation for more information.

4.7.4 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.

4.8 Advanced Usage

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

4.8.1 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.

4.8.2 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.

4.8.3 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).

4.8.4 Serialization

There are a number of situations in which Ray will place objects in the object store. Once an object is placed in the object store, it is immutable. Situations include:

1. The return values of a remote function.
2. The value `x` in a call to `ray.put(x)`.
3. Arguments to remote functions (except for simple arguments like ints or floats).

A Python object may have an arbitrary number of pointers with arbitrarily deep nesting. To place an object in the object store or send it between processes, it must first be converted to a contiguous string of bytes. Serialization and deserialization can often be a bottleneck.

Pickle is standard Python serialization library. However, for numerical workloads, pickling and unpickling can be inefficient. For example, if multiple processes want to access a Python list of numpy arrays, each process must unpickle the list and create its own new copies of the arrays. This can lead to high memory overheads, even when all processes are read-only and could easily share memory.

In Ray, we optimize for numpy arrays by using the Apache Arrow data format. When we deserialize a list of numpy arrays from the object store, we still create a Python list of numpy array objects. However, rather than copy each numpy array, each numpy array object holds a pointer to the relevant array held in shared memory. There are some advantages to this form of serialization.

- Deserialization can be very fast.
- Memory is shared between processes so worker processes can all read the same data without having to copy it.

What Objects Does Ray Handle

Ray does not currently support serialization of arbitrary Python objects. The set of Python objects that Ray can serialize using Arrow includes the following.

1. Primitive types: ints, floats, longs, bools, strings, unicode, and numpy arrays.

2. Any list, dictionary, or tuple whose elements can be serialized by Ray.

For a more general object, Ray will first attempt to serialize the object by unpacking the object as a dictionary of its fields. This behavior is not correct in all cases. If Ray cannot serialize the object as a dictionary of its fields, Ray will fall back to using pickle. However, using pickle will likely be inefficient.

Notes and limitations

- We currently handle certain patterns incorrectly, according to Python semantics. For example, a list that contains two copies of the same list will be serialized as if the two lists were distinct.

```
l1 = [0]
l2 = [l1, l1]
l3 = ray.get(ray.put(l2))

l2[0] is l2[1]  # True.
l3[0] is l3[1]  # False.
```

- For reasons similar to the above example, we also do not currently handle objects that recursively contain themselves (this may be common in graph-like data structures).

```
l = []
l.append(l)

# Try to put this list that recursively contains itself in the object store.
ray.put(l)
```

This will throw an exception with a message like the following.

```
This object exceeds the maximum recursion depth. It may contain itself␣
↪recursively.
```

- Whenever possible, use numpy arrays for maximum performance.

Last Resort Workaround

If you find cases where Ray serialization doesn't work or does something unexpected, please let us know so we can fix it. In the meantime, you may have to resort to writing custom serialization and deserialization code (e.g., calling pickle by hand).

```
import pickle

@ray.remote
def f(complicated_object):
    # Deserialize the object manually.
    obj = pickle.loads(complicated_object)
    return "Successfully passed {} into f.".format(obj)

# Define a complicated object.
l = []
l.append(l)

# Manually serialize the object and pass it in as a string.
ray.get(f.remote(pickle.dumps(l)))  # prints 'Successfully passed [[...]] into f.'
```

Note: If you have trouble with pickle, you may have better luck with cloudpickle.

4.9 Troubleshooting and FAQs

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.

4.9.1 No Speedup

You just ran an application using Ray, but it wasn't as fast as you expected it to be. Or worse, perhaps it was slower than the serial version of the application! The most common reasons are the following.

- **Number of cores:** How many cores is Ray using? When you start Ray, it will determine the number of CPUs on each machine with `psutil.cpu_count()`. Ray usually will not schedule more tasks in parallel than the number of CPUs. So if the number of CPUs is 4, the most you should expect is a 4x speedup.
- **Physical versus logical CPUs:** Do the machines you're running on have fewer **physical** cores than **logical** cores? You can check the number of logical cores with `psutil.cpu_count()` and the number of physical cores with `psutil.cpu_count(logical=False)`. This is common on a lot of machines and especially on EC2. For many workloads (especially numerical workloads), you often cannot expect a greater speedup than the number of physical CPUs.
- **Small tasks:** Are your tasks very small? Ray introduces some overhead for each task (the amount of overhead depends on the arguments that are passed in). You will be unlikely to see speedups if your tasks take less than ten milliseconds. For many workloads, you can easily increase the sizes of your tasks by batching them together.
- **Variable durations:** Do your tasks have variable duration? If you run 10 tasks with variable duration in parallel, you shouldn't expect an N-fold speedup (because you'll end up waiting for the slowest task). In this case, consider using `ray.wait` to begin processing tasks that finish first.
- **Multi-threaded libraries:** Are all of your tasks attempting to use all of the cores on the machine? If so, they are likely to experience contention and prevent your application from achieving a speedup. 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.

4.9.2 Crashes

If Ray crashed, you may wonder what happened. Currently, this can occur for some of the following reasons.

- **Stressful workloads:** Workloads that create many many tasks in a short amount of time can sometimes interfere with the heartbeat mechanism that we use to check that processes are still alive. On the head node in the cluster, you can check the files `/tmp/ray/session_*/logs/monitor*`. They will indicate which processes Ray has marked as dead (due to a lack of heartbeats). However, it is currently possible for a process to get marked as dead without actually having died.
- **Starting many actors:** Workloads that start a large number of actors all at once may exhibit problems when the processes (or libraries that they use) contend for resources. Similarly, a script that starts many actors over the lifetime of the application will eventually cause the system to run out of file descriptors. This is addressable, but currently we do not garbage collect actor processes until the script finishes.

- **Running out of file descriptors:** As a workaround, you may be able to increase the maximum number of file descriptors with a command like `ulimit -n 65536`. If that fails, double check that the hard limit is sufficiently large by running `ulimit -Hn`. If it is too small, you can increase the hard limit as follows (these instructions work on EC2).
  - Increase the hard ulimit for open file descriptors system-wide by running the following.

    ```
    sudo bash -c "echo $USER hard nofile 65536 >> /etc/security/limits.conf"
    ```

  - Logout and log back in.

4.9.3 Hanging

Tip: You can run `ray stack` to dump the stack traces of all Ray workers on the current node. This requires py-spy to be installed.

If a workload is hanging and not progressing, the problem may be one of the following.

- **Reconstructing an object created with put:** When an object that is needed has been evicted or lost, Ray will attempt to rerun the task that created the object. However, there are some cases that currently are not handled. For example, if the object was created by a call to `ray.put` on the driver process, then the argument that was passed into `ray.put` is no longer available and so the call to `ray.put` cannot be rerun (without rerunning the driver).
- **Reconstructing an object created by actor task:** Ray currently does not reconstruct objects created by actor methods.

4.9.4 Serialization Problems

Ray's serialization is currently imperfect. If you encounter an object that Ray does not serialize/deserialize correctly, please let us know. For example, you may want to bring it up on this thread.

- Objects with multiple references to the same object.
- Subtypes of lists, dictionaries, or tuples.

4.9.5 Outdated Function Definitions

Due to subtleties of Python, if you redefine a remote function, you may not always get the expected behavior. In this case, it may be that Ray is not running the newest version of the function.

Suppose you define a remote function `f` and then redefine it. Ray should use the newest version.

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

@ray.remote
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`.

4.10 Ray Package Reference

`ray.`**`init`**(*redis_address=None, address=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=None, 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=None, plasma_directory=None, huge_pages=False, include_webui=False, job_id=None, configure_logging=True, logging_level=20, logging_format='%(asctime)s\t%(levelname)s %(filename)s:%(lineno)s – %(message)s', plasma_store_socket_name=None, raylet_socket_name=None, temp_dir=None, load_code_from_local=False, _internal_config=None*)

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

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

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

```
ray.init()
```

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

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

Parameters

- **`redis_address`** (*str*) – The address of the Redis server to connect to. If this address is not provided, then this command will start Redis, a raylet, a plasma store, a plasma manager, and some workers. It will also kill these processes when Python exits.
- **`address`** (*str*) – Same as redis_address.
- **`num_cpus`** (*int*) – Number of cpus the user wishes all raylets to be configured with.
- **`num_gpus`** (*int*) – Number of gpus the user wishes all raylets to be configured with.
- **`resources`** – A dictionary mapping the name of a resource to the quantity of that resource available.
- **`memory`** – The amount of memory (in bytes) that is available for use by workers requesting memory resources. By default, this is autoset based on available system memory.
- **`object_store_memory`** – The amount of memory (in bytes) to start the object store with. By default, this is autoset 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, then 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*) – True if the code should be executed serially without Ray. 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** – True if we should suppress errors from calling ray.init() a second time.
- **num_redis_shards** – The number of Redis shards to start in addition to the primary Redis shard.
- **redis_max_clients** – If provided, attempt to configure Redis with this maxclients number.
- **redis_password** (*str*) – Prevents external clients without the password from connecting to Redis if provided.
- **plasma_directory** – A directory where the Plasma memory mapped files will be created.
- **huge_pages** – Boolean flag indicating whether to start the Object Store with hugetlbfs support. Requires plasma_directory.
- **include_webui** – Boolean flag indicating whether to start the web UI, which displays the status of the Ray cluster.
- **job_id** – The ID of this job.
- **configure_logging** – True if allow the logging cofiguration here. Otherwise, the users may want to configure it by their own.
- **logging_level** – Logging level, default will be logging.INFO.
- **logging_format** – Logging format, default contains a timestamp, filename, line number, and message. See ray_constants.py.
- **plasma_store_socket_name** (*str*) – If provided, it will specify the socket name used by the plasma store.
- **raylet_socket_name** (*str*) – If provided, it will specify the socket path used by the raylet process.
- **temp_dir** (*str*) – If provided, it will specify the root temporary directory for the Ray process.
- **load_code_from_local** – Whether code should be loaded from a local module or from the GCS.
- **_internal_config** (*str*) – JSON configuration for overriding RayConfig defaults. For testing purposes ONLY.

Returns Address information about the started processes.

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

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

Check if ray.init has been called yet.

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

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

Define a remote function or an actor class.

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

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

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

It can also be used with specific keyword arguments:

- **num_return_vals:** This is only for *remote functions*. It specifies the number of object IDs returned by the remote function invocation.
- **num_cpus:** The quantity of CPU cores to reserve for this task or for the lifetime of the actor.
- **num_gpus:** The quantity of GPUs to reserve for this task or for the lifetime of the actor.
- **resources:** The quantity of various custom resources to reserve for this task or for the lifetime of the actor. This is a dictionary mapping strings (resource names) to numbers.
- **max_calls:** Only for *remote functions*. This specifies the maximum number of times that a given worker can execute the given remote function before it must exit (this can be used to address memory leaks in third-party libraries or to reclaim resources that cannot easily be released, e.g., GPU memory that was acquired by TensorFlow). By default this is infinite.
- **max_reconstructions**: Only for *actors*. This specifies the maximum number of times that the actor should be reconstructed when it dies unexpectedly. The minimum valid value is 0 (default), which indicates that the actor doesn't need to be reconstructed. And the maximum valid value is ray.ray_constants.INFINITE_RECONSTRUCTIONS.

This can be done as follows:

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

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

`ray.`**`get`**(*object_ids*)

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

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

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

Returns A Python object or a list of Python objects.

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

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

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

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

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

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

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

Parameters

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

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

`ray.`**`put`**`(`*value*`)`

Store an object in the object store.

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

Returns The object ID assigned to this value.

`ray.`**`get_gpu_ids`**`()`

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

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

Returns A list of GPU IDs.

`ray.`**`get_resource_ids`**`()`

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

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

`ray.`**`get_webui_url`**`()`

Get the URL to access the web UI.

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

Returns The URL of the web UI as a string.

`ray.`**`shutdown`**`(`*exiting_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 **exiting_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*, *use_pickle=False*, *use_dict=False*, *serializer=None*, *deserializer=None*, *local=False*, *job_id=None*, *class_id=None*)

Enable serialization and deserialization for a particular class.

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

Parameters

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

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

ray.**profile**(*event_type*, *extra_data=None*)

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

Note that this only works in the raylet code path.

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

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

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

Parameters

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

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

`ray.method(*args, **kwargs)`
Annotate an actor method.

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

f = Foo.remote()

_, _ = f.bar.remote()
```

Parameters **`num_return_vals`** – The number of object IDs that should be returned by invocations of this actor method.

4.10.1 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.tasks(task_id=None)`
Fetch and parse the task table information for one or more task IDs.

Parameters **`task_id`** – A hex string of the task ID to fetch information about. If this is None, then the task object table is fetched.

Returns Information from the task table.

`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`**`(`*include_cluster_errors=True*`)`

Get error messages from the cluster.

Parameters **`include_cluster_errors`** – True if we should include error messages for all drivers, and false if we should only include error messages for this specific driver.

Returns Error messages pushed from the cluster.

4.10.2 The Ray Command Line API

ray start

```
ray start [OPTIONS]
```

Options

`--node-ip-address` `<node_ip_address>`

the IP address of this node

`--redis-address` `<redis_address>`

the address to use for connecting to Redis

`--address` `<address>`

same as –redis-address

`--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
provide this argument if the UI should be started

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

--plasma-directory `<plasma_directory>`
object store directory for memory mapped files

--huge-pages
enable support for huge pages in the object store

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

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

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

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

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

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

--include-java
Enable Java worker support.

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

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

--load-code-from-local
Specify whether load code from local file or GCS serialization.

ray stop

```
ray stop [OPTIONS]
```

ray up

Create or update a Ray cluster.

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

Options

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

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

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

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

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

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

Arguments

CLUSTER_CONFIG_FILE
Required argument

ray down

Tear down the Ray cluster.

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

Options

--workers-only
Only destroy the workers.

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

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

Arguments

CLUSTER_CONFIG_FILE
Required argument

ray exec

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

Options

--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.

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

Arguments

CLUSTER_CONFIG_FILE
Required argument

CMD
Required argument

ray attach

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

Options

--start
Start the cluster if needed.

--tmux
Run the command in tmux.

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

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

Arguments

CLUSTER_CONFIG_FILE
Required argument

ray get_head_ip

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

Options

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

Arguments

CLUSTER_CONFIG_FILE
Required argument

ray stack

```
ray stack [OPTIONS]
```

ray timeline

```
ray timeline [OPTIONS]
```

Options

--redis-address `<redis_address>`
Override the redis address to connect to.

4.11 Built-in Autoscaling

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

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

4.11.1 Quick start (AWS)

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

Then you're ready to go. The provided ray/python/ray/autoscaler/aws/example-full.yaml cluster config file will create a small cluster with a m5.large head node (on-demand) configured to autoscale up to two m5.large spot workers.

Try it out by running these commands from your personal computer. Once the cluster is started, you can then SSH into the head node, `source activate tensorflow_p36`, and then run Ray programs with `ray.init(redis_address="localhost:6379")`.

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

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

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

4.11.2 Quick start (GCP)

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

Then you're ready to go. The provided ray/python/ray/autoscaler/gcp/example-full.yaml cluster config file will create a small cluster with a n1-standard-2 head node (on-demand) configured to autoscale up to two n1-standard-2 preemptible workers. Note that you'll need to fill in your project id in those templates.

Try it out by running these commands from your personal computer. Once the cluster is started, you can then SSH into the head node and then run Ray programs with `ray.init(redis_address="localhost:6379")`.

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

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

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

4.11.3 Quick start (Private Cluster)

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

Try it out by running these commands from your personal computer. Once the cluster is started, you can then SSH into the head node and then run Ray programs with `ray.init(redis_address="localhost:6379")`.

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

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

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

4.11.4 Running commands on new and existing clusters

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

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

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

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

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

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

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

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

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

4.11.5 Attaching to the cluster

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

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

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

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

4.11.6 Port-forwarding applications

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

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

4.11.7 Manually synchronizing files

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

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

4.11.8 Updating your cluster

When you run `ray up` with an existing cluster, the command checks if the local configuration differs from the applied configuration of the cluster. This includes any changes to synced files specified in the `file_mounts` section of the config. If so, the new files and config will be uploaded to the cluster. Following that, Ray services will be restarted.

You can also run `ray up` to restart a cluster if it seems to be in a bad state (this will restart all Ray services even if there are no config changes).

If you don't want the update to restart services (e.g. because the changes don't require a restart), pass `--no-restart` to the update call.

4.11.9 Security

By default, the nodes will be launched into their own security group, with traffic allowed only between nodes in the same group. A new SSH key will also be created and saved to your local machine for access to the cluster.

4.11.10 Autoscaling

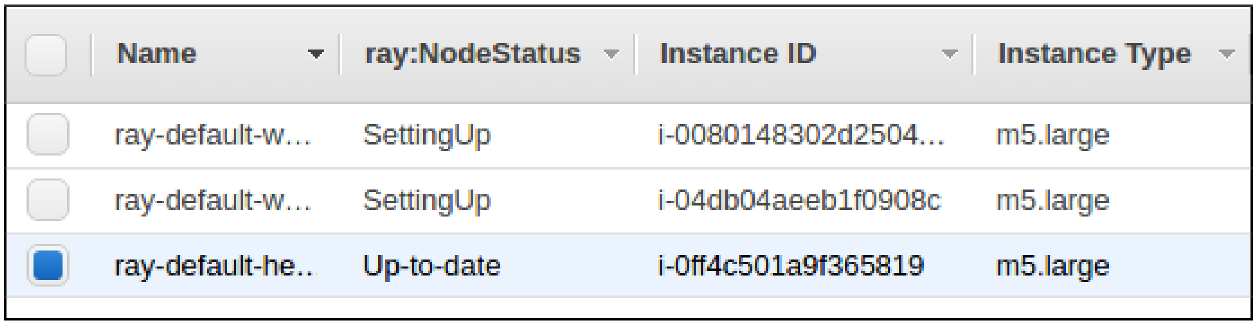
Ray clusters come with a load-based auto-scaler. When cluster resource usage exceeds a configurable threshold (80% by default), new nodes will be launched up the specified `max_workers` limit. When nodes are idle for more than a timeout, they will be removed, down to the `min_workers` limit. The head node is never removed.

The default idle timeout is 5 minutes. This is to prevent excessive node churn which could impact performance and increase costs (in AWS / GCP there is a minimum billing charge of 1 minute per instance, after which usage is billed by the second).

4.11.11 Monitoring cluster status

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

The Ray autoscaler also reports per-node status in the form of instance tags. In your cloud provider console, you can click on a Node, go the the "Tags" pane, and add the `ray-node-status` tag as a column. This lets you see per-node statuses at a glance:



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

4.11.12 Customizing cluster setup

You are encouraged to copy the example YAML file and modify it to your needs. This may include adding additional setup commands to install libraries or sync local data files.

Note: After you have customized the nodes, it is also a good idea to create a new machine image and use that in the config file. This reduces worker setup time, improving the efficiency of auto-scaling.

The setup commands you use should ideally be *idempotent*, that is, can be run more than once. This allows Ray to update nodes after they have been created. You can usually make commands idempotent with small modifications, e.g. `git clone foo` can be rewritten as `test -e foo || git clone foo` which checks if the repo is already cloned first.

Most of the example YAML file is optional. Here is a reference minimal YAML file, and you can find the defaults for optional fields in this YAML file.

4.11.13 Syncing git branches

A common use case is syncing a particular local git branch to all workers of the cluster. However, if you just put a *git checkout <branch>* in the setup commands, the autoscaler won't know when to rerun the command to pull in updates. There is a nice workaround for this by including the git SHA in the input (the hash of the file will change if the branch is updated):

```
file_mounts: {
    "/tmp/current_branch_sha": "/path/to/local/repo/.git/refs/heads/<YOUR_BRANCH_NAME>
↪",
}

setup_commands:
    - test -e <REPO_NAME> || git clone https://github.com/<REPO_ORG>/<REPO_NAME>.git
    - cd <REPO_NAME> && git fetch && git checkout `cat /tmp/current_branch_sha`
```

This tells `ray up` to sync the current git branch SHA from your personal computer to a temporary file on the cluster (assuming you've pushed the branch head already). Then, the setup commands read that file to figure out which SHA they should checkout on the nodes. Note that each command runs in its own session. The final workflow to update the cluster then becomes just this:

1. Make local changes to a git branch
2. Commit the changes with `git commit` and `git push`
3. Update files on your Ray cluster with `ray up`

4.11.14 Common cluster configurations

The `example-full.yaml` configuration is enough to get started with Ray, but for more compute intensive workloads you will want to change the instance types to e.g. use GPU or larger compute instance by editing the yaml file. Here are a few common configurations:

GPU single node: use Ray on a single large GPU instance.

```
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:
```

(continues on next page)

(continued from previous page)

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

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

```
min_workers: 1  # must have at least 1 GPU worker (issue #2106)
max_workers: 10
head_node:
    InstanceType: m4.large
worker_nodes:
    InstanceMarketOptions:
        MarketType: spot
    InstanceType: p2.xlarge
```

4.11.15 External Node Provider

Ray also supports external node providers (check node_provider.py implementation). You can specify the external node provider using the yaml config:

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

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

4.11.16 Additional Cloud providers

To use Ray autoscaling on other Cloud providers or cluster management systems, you can implement the `NodeProvider` interface (~100 LOC) and register it in node_provider.py. Contributions are welcome!

4.11.17 Questions or Issues?

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

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

4.12 Manual Cluster Setup

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

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

4.12.1 Deploying Ray on a Cluster

This section assumes that you have a cluster running and that the nodes in the cluster can communicate with each other. It also assumes that Ray is installed on each machine. To install Ray, follow the installation instructions.

Starting Ray on each machine

On the head node (just choose some node to be the head node), run the following. If the `--redis-port` argument is omitted, Ray will choose a port at random.

```
ray start --head --redis-port=6379
```

The command will print out the address of the Redis server that was started (and some other address information).

Then on all of the other nodes, run the following. Make sure to replace `<redis-address>` with the value printed by the command on the head node (it should look something like `123.45.67.89:6379`).

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

If you wish to specify that a machine has 10 CPUs and 1 GPU, you can do this with the flags `--num-cpus=10` and `--num-gpus=1`. 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(redis_address="<redis-address>")
```

Now you can define remote functions and execute tasks. For example, to verify that the correct number of nodes have joined the cluster, you can run the following.

```
import time

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

# Get a list of the IP addresses of the nodes that have joined the cluster.
set(ray.get([f.remote() for _ in range(1000)]))
```

Stopping Ray

When you want to stop the Ray processes, run `ray stop` on each node.

4.13 Deploying on Kubernetes

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

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

Start by cloning the Ray repository.

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

4.13.1 Work Interactively on the Cluster

To work interactively, first start Ray on Kubernetes.

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

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

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

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

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

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

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

```
from collections import Counter
import time
import ray

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

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

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

4.13.2 Submitting a Script to the Cluster

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

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

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

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

```
$ kubectl get all -n ray
NAME                                  READY   STATUS    RESTARTS   AGE
pod/ray-head-5486648dc9-c6hz2         1/1     Running   0          11s
pod/ray-worker-5c49b7cc57-2jz4l       1/1     Running   0          11s
pod/ray-worker-5c49b7cc57-8nwjk       1/1     Running   0          11s
pod/ray-worker-5c49b7cc57-xlksn       1/1     Running   0          11s

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

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

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

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

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

4.13.3 Cleaning Up

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

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

4.13.4 Customization

You will probably need to do some amount of customization.

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

4.13.5 TODO

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

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

4.14 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=20
#SBATCH --mem-per-cpu=1GB
#SBATCH --nodes=5
#SBATCH --tasks-per-node 1

worker_num=4 # Must be one less that 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 $node1 hostname --ip-address) # Making redis-
↪address
suffix=':6379'
ip_head=$ip_prefix$suffix

export ip_head # Exporting for latter access by trainer.py

srun --nodes=1 --ntasks=1 -w $node1 ray start --block --head --redis-port=6379 & #␣
↪Starting the head
sleep 5

for ((  i=1; i<=$worker_num; i++ ))
do
  node2=${nodes_array[$i]}
  srun --nodes=1 --ntasks=1 -w $node2 ray start --block --redis-address=$ip_head & #␣
↪Starting the workers
  sleep 5
done

python trainer.py 100 # Pass the total number of allocated CPUs
```

```
# trainer.py
import os
import sys
import time
import ray

ray.init(redis_address=os.environ["ip_head"])

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

# The following takes one second (assuming that ray was able to access all of the␣
↪allocated nodes).
start = time.time()
num_cpus = int(sys.argv[1])
ray.get([f.remote() for _ in range(num_cpus)])
end = time.time()
print(end - start)
```

4.15 Tune: Scalable Hyperparameter Search

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

- Scale to running on a large distributed cluster without changing your code.
- Launch a multi-node Tune experiment in less than 10 lines of code.
- Supports any deep learning framework, including PyTorch, TensorFlow, and Keras.

- Visualize results with TensorBoard.
- Choose among scalable SOTA algorithms such as Population Based Training (PBT), Vizier's Median Stopping Rule, HyperBand/ASHA.

4.15.1 Quick Start

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

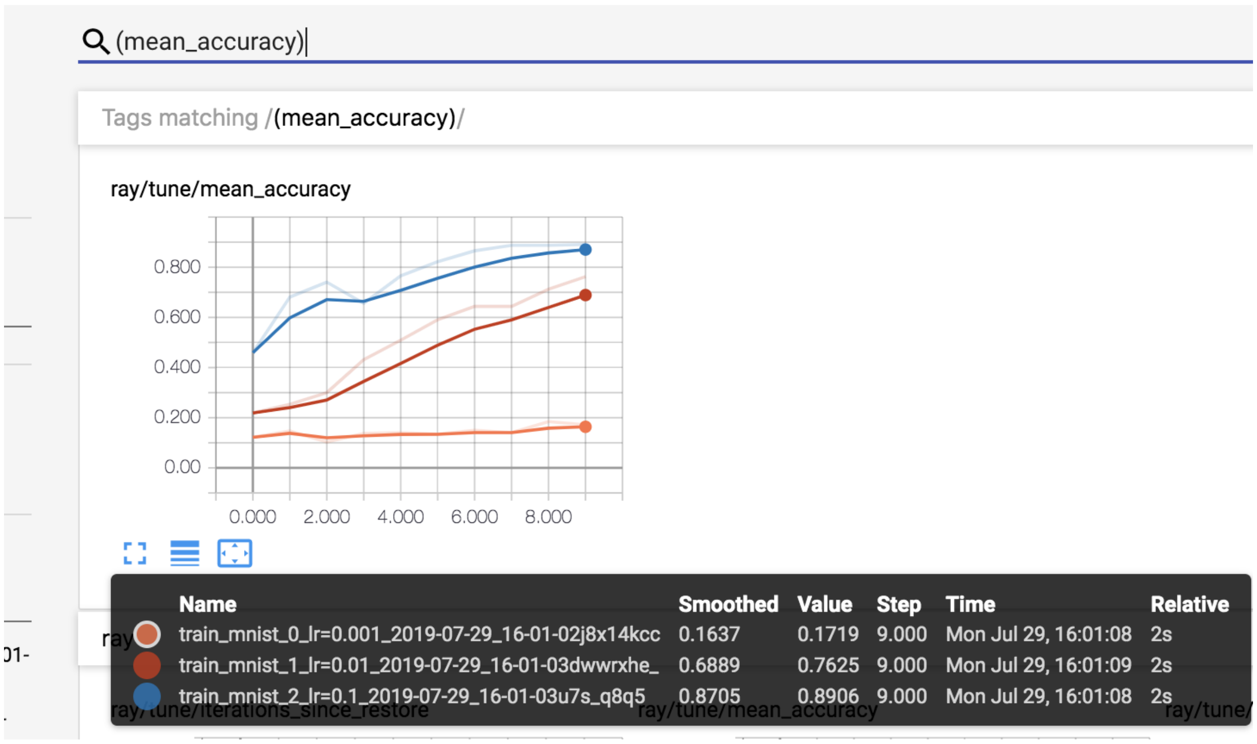
```
$ pip install ray torch torchvision filelock
```

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
```



	Name	Smoothed	Value	Step	Time	Relative
●	train_mnist_0_lr=0.001_2019-07-29_16-01-02j8x14kcc	0.1637	0.1719	9.000	Mon Jul 29, 16:01:08	2s
●	train_mnist_1_lr=0.01_2019-07-29_16-01-03dwwrxhe_	0.6889	0.7625	9.000	Mon Jul 29, 16:01:09	2s
●	train_mnist_2_lr=0.1_2019-07-29_16-01-03u7s_q8q5	0.8705	0.8906	9.000	Mon Jul 29, 16:01:08	2s

4.15.2 Distributed Quick Start

1. Import and initialize Ray by appending the following to your example script.

```
# Append to top of your script
import ray
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--ray-redis-address")
args = parser.parse_args()
ray.init(redis_address=args.ray_redis_address)
```

Alternatively, download a full example script here: `mnist_pytorch.py`

2. Download the following example Ray cluster configuration as `tune-local-default.yaml` and replace the appropriate fields:

```
cluster_name: local-default
provider:
    type: local
    head_ip: YOUR_HEAD_NODE_HOSTNAME
    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
```

Alternatively, download it here: `tune-local-default.yaml`. See Ray cluster docs here.

3. Run `ray submit` like the following.

```
ray submit tune-local-default.yaml mnist_pytorch.py --args="--ray-redis-
↪address=localhost:6379" --start
```

This will start Ray on all of your machines and run a distributed hyperparameter search across them.

To summarize, here are the full set of commands:

```
wget https://raw.githubusercontent.com/ray-project/ray/master/python/ray/tune/
↪examples/mnist_pytorch.py
wget https://raw.githubusercontent.com/ray-project/ray/master/python/ray/tune/tune-
↪local-default.yaml
ray submit tune-local-default.yaml mnist_pytorch.py --args="--ray-redis-
↪address=localhost:6379" --start
```

Take a look at the Distributed Experiments documentation for more details, including:

1. Setting up distributed experiments on your local cluster
2. Using AWS and GCP
3. Spot instance usage/pre-emptible instances, and more.

4.15.3 Getting Started

- Code: GitHub repository for Tune.
- User Guide: A comprehensive overview on how to use Tune's features.
- Tutorial Notebook: Our tutorial notebooks of using Tune with Keras or PyTorch.

4.15.4 Contribute to Tune

Take a look at our Contributor Guide for guidelines on contributing.

4.15.5 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}
}
```

4.16 Tune Example Walkthrough

This tutorial will walk you through the following process to setup a Tune experiment. 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 filelock
```

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.

```
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)
```

(continues on next page)

(continued from previous page)

```
}

# Uncomment this to enable distributed execution
# `ray.init(redis_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()]
```

4.16.1 Early Stopping with ASHA

Let's integrate an early stopping algorithm to our search - ASHA, a scalable algorithm for principled early stopping.

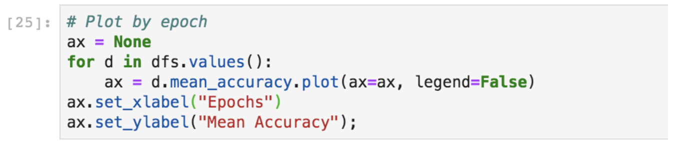
How does it work? On a high level, it terminates trials that are less promising and allocates more time and resources to more promising trials. See this blog post for more details.

We can afford to **increase the search space by 5x**, by adjusting the parameter `num_samples`. See the Trial Scheduler section for more details of available schedulers and library integrations.

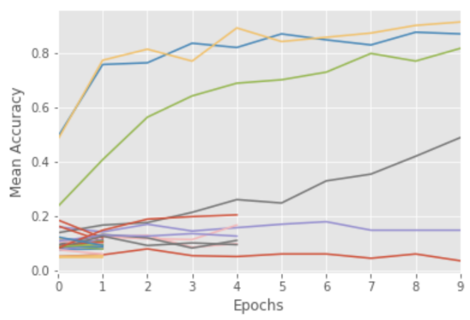
```
analysis = tune.run(
    train_mnist,
    num_samples=30,
    scheduler=ASHAScheduler(metric="mean_accuracy", mode="max"),
    config=search_space)

# 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}
```

4.16.2 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 the Search Algorithm section 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, max_concurrent=2, reward_attr="mean_accuracy")

analysis = tune.run(train_mnist, num_samples=10, search_alg=hyperopt_search)
```

4.16.3 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 Usage Guide for more comprehensive overview of Tune features.

4.17 Tune User Guide

4.17.1 Tune Overview

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

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

4.17.2 Experiment Configuration

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

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

Training API

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

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

```
class Example(Trainable):
    def _setup(self, config):
        ...

    def _train(self):
        # run training code
        result_dict = {"accuracy": 0.5, "f1": 0.1, ...}
        return result_dict
```

Python functions will need to have the following signature and call `tune.track.log`, which will allow you to report metrics used for scheduling, search, or early stopping.:

```
def trainable(config):
    """
    Args:
        config (dict): Parameters provided from the search algorithm
            or variant generation.
    """

    while True:
        # ...
        tune.track.log(**kwargs)
```

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. `tune.track` documentation can be found here.

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

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

Note: See previous versions of the documentation for the `reporter` API.

Launching an Experiment

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

```
tune.run(
    trainable,
    name="example-experiment",
    num_samples=10,
)
```

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

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

All results reported by the trainable will be logged locally to a unique directory per experiment, e.g. `~/ray_results/example-experiment` in the above example. On a cluster, incremental results will be synced to local disk on the head node.

4.17.3 Analyzing Results

Tune provides an `ExperimentAnalysis` object for analyzing results from `tune.run`.

```
analysis = tune.run(
    trainable,
    name="example-experiment",
    num_samples=10,
)
```

You can use the `ExperimentAnalysis` object to obtain the best configuration of the experiment:

```
>>> print("Best config is", analysis.get_best_config(metric="mean_accuracy"))
Best config is: {'lr': 0.011537575723482687, 'momentum': 0.8921971713692662}
```

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")

# 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")
```

See the full documentation for the `Analysis` object.

4.17.4 Training Features

Tune Search Space (Default)

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

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

The following shows grid search over two nested parameters combined with random sampling from two lambda functions, generating 9 different trials. Note that the value of `beta` depends on the value of `alpha`, which is represented by referencing `spec.config.alpha` in the lambda function. This lets you specify conditional parameter distributions.

```
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]),
        ],
    }
)
```

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

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

Custom Trial Names

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

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

    Returns:
        trial_name (str): String representation of Trial.
```

(continues on next page)

(continued from previous page)

```
    """
    return str(trial)

tune.run(
    MyTrainableClass,
    name="example-experiment",
    num_samples=1,
    trial_name_creator=tune.function(trial_name_string)
)
```

An example can be found in logging_example.py.

Sampling Multiple Times

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

```
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]),
        ],
    },
    num_samples=10
)
```

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

Using GPUs (Resource Allocation)

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

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

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

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

```
tune.run(
    my_trainable,
    name="my_trainable",
    resources_per_trial={
        "cpu": 1,
        "gpu": 1,
        "extra_gpu": 4
    }
)
```

4.17.5 Save and Restore

When running a hyperparameter search, Tune can automatically and periodically save/checkpoint your model. Checkpointing is used for

- saving a model at the end of training
- modifying a model in the middle of training
- fault-tolerance in experiments with pre-emptible machines.
- enables certain Trial Schedulers such as HyperBand and PBT.

To enable checkpointing, you must implement a Trainable class (Trainable functions are not checkpointable, since they never return control back to their caller). The easiest way to do this is to subclass the pre-defined `Trainable` class and implement `_save`, and `_restore` abstract methods, as seen in this example.

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

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

    def _train(self):
        return {"mean_accuracy: self.sess.run(...)}

    def _save(self, checkpoint_dir):
        return self.saver.save(self.sess, os.path.join(checkpoint_dir, save))

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

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>)`. To test if your Trainable will checkpoint and restore correctly, you can use `tune.util.validate_save_restore` as follows:

```
from ray.tune.util import validate_save_restore

validate_save_restore(MyTrainableClass)
validate_save_restore(MyTrainableClass, use_object_store=True)
```

4.17.6 Trainable (Trial) Checkpointing

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"},
)
```

4.17.7 Fault Tolerance

Tune will automatically restart trials from the last checkpoint in case of trial failures/error (if `max_failures` is set), both in the single node and distributed setting.

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.

Take a look at an example.

Recovering From Failures

Tune automatically persists the progress of your entire experiment (a `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 makes no effect: 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 preserved. Only `FIFOScheduler` and `BasicVariantGenerator` will be supported.

4.17.8 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.util import pin_in_object_store, get_pinned_object

import numpy as np

ray.init()

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

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

tune.run(f)
```

4.17.9 Auto-Filled Results

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

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

# (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 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 experiment.
EPISODES_TOTAL = "episodes_total"

# Number of timesteps in this iteration.
```

(continues on next page)

(continued from previous page)

```
TIMESTEPS_THIS_ITER = "timesteps_this_iter"

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

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

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

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

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

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

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

4.17.10 Visualizing Results

To visualize learning in tensorboard, install TensorFlow:

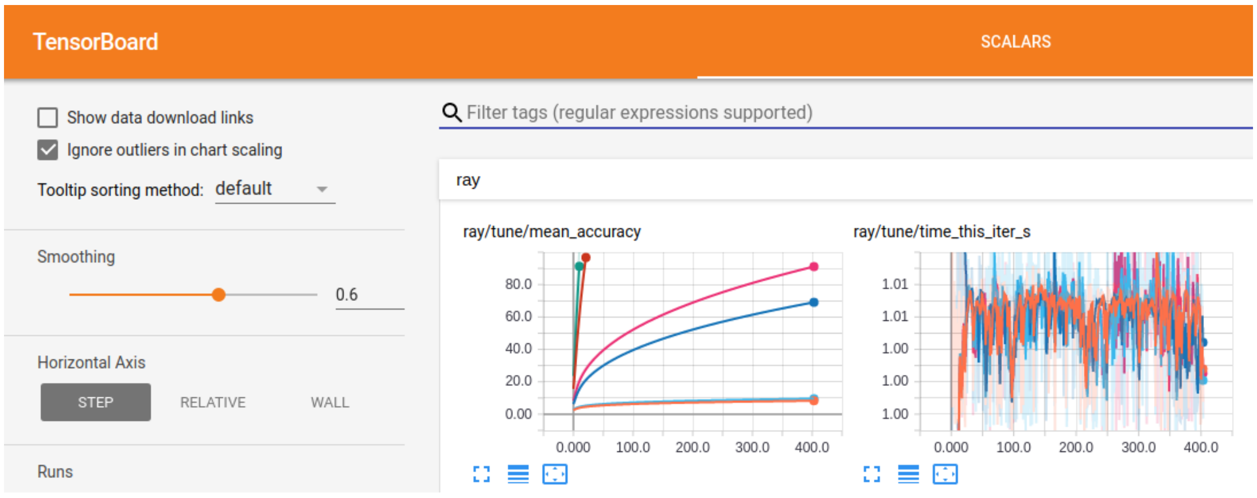
```
$ pip install tensorflow
```

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

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

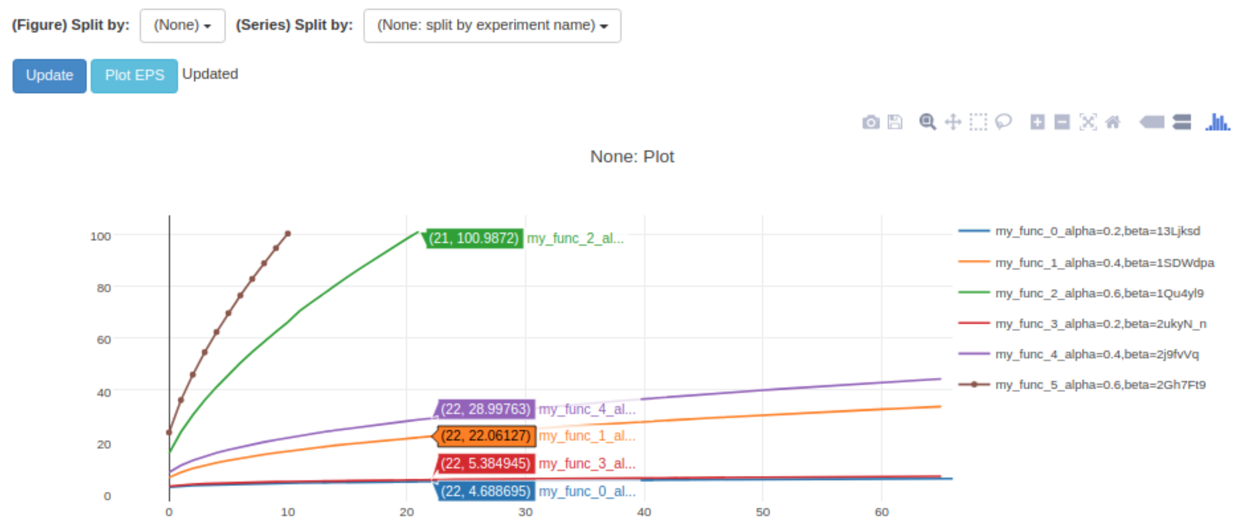
If you are running Ray on a remote multi-user cluster where you do not have sudo access, you can run the following commands to make sure tensorboard is able to write to the tmp directory:

```
$ export TMPDIR=/tmp/$USER; mkdir -p $TMPDIR; tensorboard --logdir=~/ray_results
```


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

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

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


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

4.17.11 Logging

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. Tune enables default loggers for Tensorboard, CSV, and JSON formats. You can also check out logger.py for implementation details. An example can be found in logging_example.py.

MLFlow

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.

4.17.12 Uploading/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 (and must be wrapped with `tune.function`):

```python
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=tune.function(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 would be also be used in node failures, where the source argument would be the path to the trial directory and the target would be a remote path. The *sync_to_driver* would be invoked to push a checkpoint to new node for a queued trial to resume.

If an upload directory is provided, Tune will automatically sync results to the given directory, natively supporting standard S3/gsutil commands. You can customize this to specify arbitrary storages with the `sync_to_cloud` argument. This argument is similar to `sync_to_cloud` in that it supports strings with the same replacement fields and arbitrary functions. See syncer.py for implementation details.

```python
tune.run(
    MyTrainableClass,
    name="experiment_name",
    sync_to_cloud=tune.function(custom_sync_func),
)
```

4.17.13 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`()

Returns a list of all trials' information.

`get_trial`(*trial_id*)

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.

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>
```

4.17.14 Debugging (Single Process)

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)
```

Note that some behavior such as writing to files by depending on the current working directory in a Trainable and setting global process variables may not work as expected. Local mode with multiple configuration evaluations will interleave computation, so it is most naturally used when running a single configuration evaluation.

4.17.15 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.
```

- `tune list-experiments`: List tabular information about experiments within a project. 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` flag to select specific columns to display.

```
$ tune list-experiments [PROJECT_DIR] --output note.csv

+----------------------+----------------+------------------+---------------------+
| name                 |   total_trials |   running_trials |   terminated_trials |
|----------------------+----------------+------------------+---------------------|
| pbt_test             |             10 |                0 |                   0 |
| test                 |              1 |                0 |                   0 |
| hyperband_test       |              1 |                0 |                   1 |
+----------------------+----------------+------------------+---------------------+
Dropped columns: ['error_trials', 'last_updated']
Please increase your terminal size to view remaining columns.
```

(continues on next page)

(continued from previous page)

```
Output saved at: note.csv

$ tune list-experiments [PROJECT_DIR] --filter "total_trials <= 1" --sort name

+-----------------------+----------------+------------------+---------------------+
| name                  |   total_trials |   running_trials |   terminated_trials |
|-----------------------+----------------+------------------+---------------------|
| hyperband_test        |              1 |                0 |                   1 |
| test                  |              1 |                0 |                   0 |
+-----------------------+----------------+------------------+---------------------+
Dropped columns: ['error_trials', 'last_updated']
Please increase your terminal size to view remaining columns.
```

4.17.16 Further Questions or Issues?

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

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

4.18 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(redis_address=...)` to connect to the existing Ray cluster.
2. If a ray cluster does not exist, start a Ray cluster (instructions for local machines, cloud).
3. Run the script on the head node (or use `ray submit`).

4.18.1 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 (instructions for local machines, 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(redis_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("--ray-redis-address")
args = parser.parse_args()
ray.init(redis_address=args.ray_redis_address)

tune.run(...)
```

```
# On the head node, connect to an existing ray cluster
$ python tune_script.py --ray-redis-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 --args="--ray-redis-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.

4.18.2 Local Cluster Setup

If you have 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
    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 --args="--ray-redis-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 `<redis-address>` with the value printed by the command on the head node (it should look something like `123.45.67.89:6379`).

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

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-redis-address=<redis-address>
```

4.18.3 Launching a cloud cluster

Tip: If you have already have a list of nodes, go to the *Local Cluster Setup* section.

Ray currently supports AWS and GCP. Below, we will launch nodes on AWS that will default to using the Deep Learning AMI. See the cluster setup documentation. Save the below cluster configuration (`tune-default.yaml`):

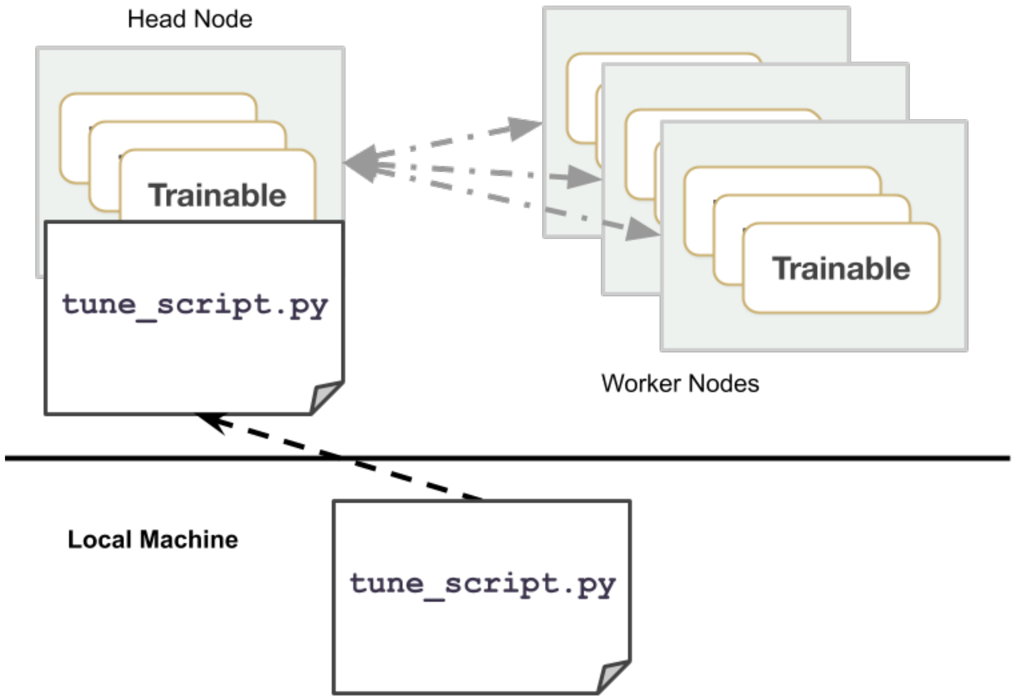
```
cluster_name: tune-default
provider: {type: aws, region: us-west-2}
auth: {ssh_user: ubuntu}
min_workers: 3
max_workers: 3
# Deep Learning AMI (Ubuntu) Version 21.0
head_node: {InstanceType: c5.xlarge, ImageId: ami-0b294f219d14e6a82}
worker_nodes: {InstanceType: c5.xlarge, ImageId: ami-0b294f219d14e6a82}
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]`.

```
ray submit tune-default.yaml tune_script.py --start --args="--ray-redis-
↪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.

4.18.4 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:
```

(continues on next page)

(continued from previous page)

```
      - 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

    # Run workers on preemtible 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 \
    --args="--ray-redis-address=localhost:6379" \
    --start
```

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 --args="--ray-redis-
↪address=localhost:6379" --start

# 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 Save and Restore section for more details.

You can also specify `tune.run(upload_dir=...)` to sync results with a cloud storage like S3, persisting results in case you want to start and stop your cluster automatically.

4.18.5 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 --redis-address=localhost:6379` on the remote␣
↪machine.
$ ray submit CLUSTER.YAML tune_experiment.py --args="--redis-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  --args="--redis-
↪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"
```

4.18.6 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
```

4.19 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:

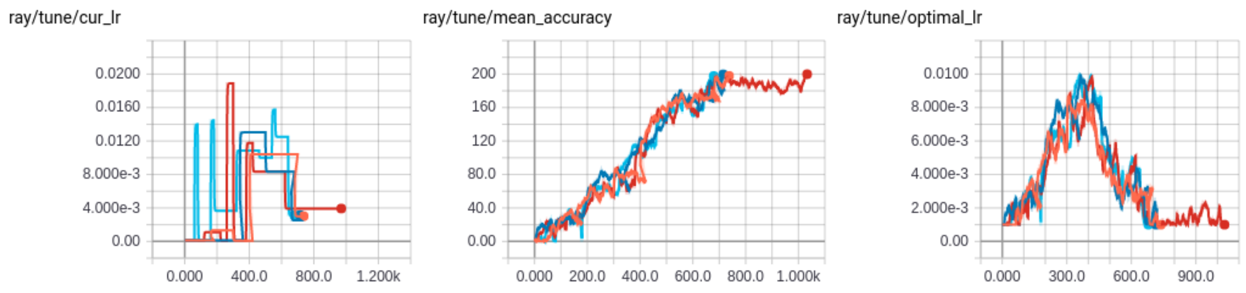
4.19.1 Population Based Training (PBT)

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],
            "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 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, resample_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>)*.

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.

Example

```
>>> 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)
```

4.19.2 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*, *metric='episode_reward_mean'*, *mode='max'*, *max_t=100*, *grace_period=1*, *reduction_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.

4.19.3 HyperBand

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

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

```
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
```

(continues on next page)

(continued from previous page)

```
 - 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

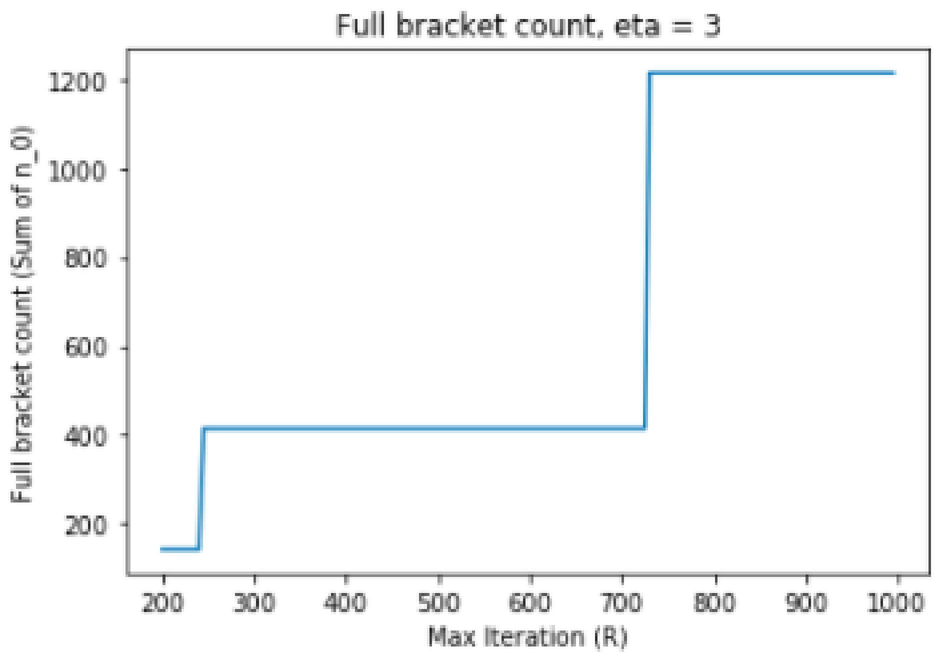
Parameters

- **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.

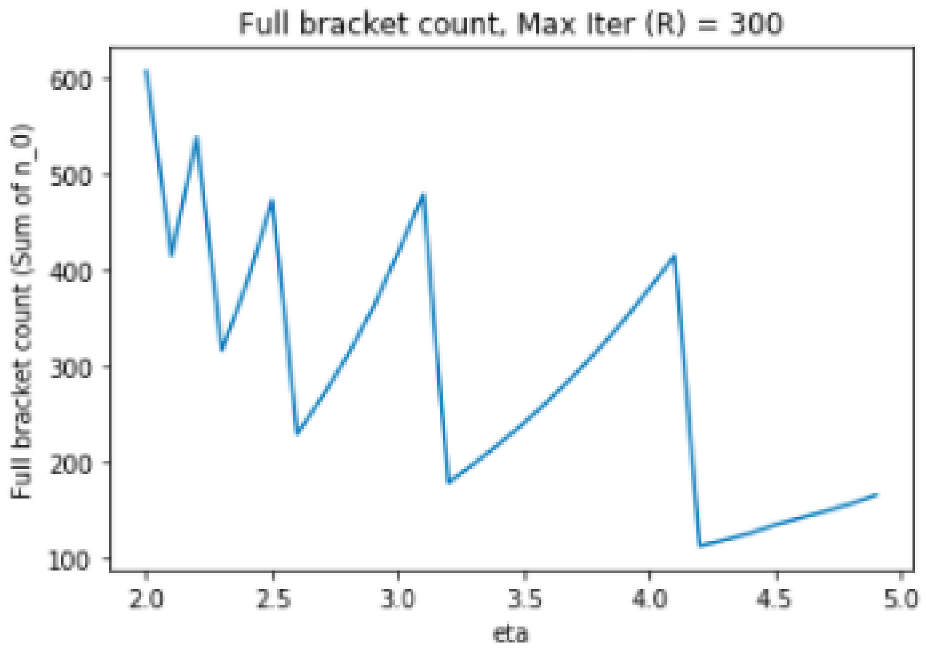
HyperBand Implementation Details

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 >= 200`. For simplicity, assume `eta = 3`. Varying `R` between `R = 200` and `R = 1000` creates a huge range of the number of trials needed to fill up all brackets.

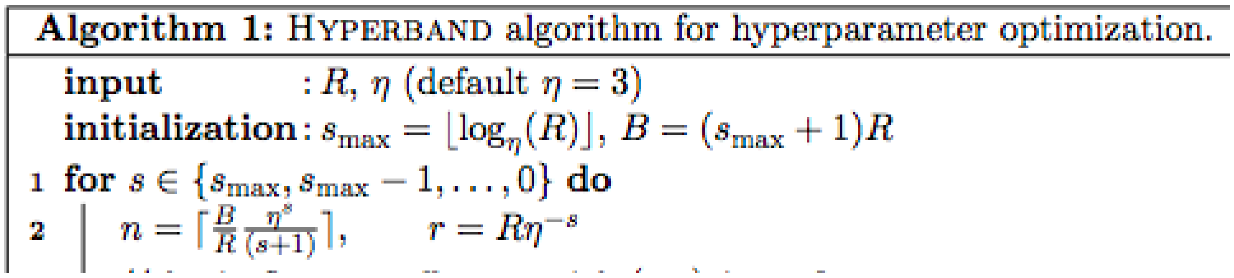


On the other hand, holding `R` constant at `R = 300` and varying `eta` also leads to HyperBand configurations that are not very intuitive:



The implementation takes the same configuration as the example given in the paper and exposes `max_t`, which is not a parameter in the paper.

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



```
Algorithm 1: HYPERBAND algorithm for hyperparameter optimization.
   input          : $R$, $\eta$ (default $\eta = 3$)
   initialization : $s_{\max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{\max} + 1)R$
1  for $s \in \{s_{\max}, s_{\max} - 1, \ldots, 0\}$ 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.

4.19.4 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.

4.19.5 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*, *hard_stop=True*, *verbose=True*)

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

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

Parameters

- **time_attr** (*str*) – The training result attr to use for comparing time. Note that you can pass in something non-temporal such as *training_iteration* as a measure of progress, the only requirement is that the attribute should increase monotonically.
- **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 units are the same as the attribute named by *time_attr*.
- **min_samples_required** (*int*) – Min samples to compute median over.
- **hard_stop** (*bool*) – If False, pauses trials instead of stopping them. When all other trials are complete, paused trials will be resumed and allowed to run FIFO.
- **verbose** (*bool*) – If True, will output the median and best result each time a trial reports. Defaults to True.

4.20 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

4.20.1 Variant Generation (Grid Search/Random Search)

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

Example

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

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

4.20.2 BayesOpt Search

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

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

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

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

```
tune.run(... , search_alg=BayesOptSearch(bayesopt_space, utility_kwargs=utility_
↪params, ... ))
```

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

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

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

A wrapper around BayesOpt to provide trial suggestions.

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

Parameters

- **space** (*dict*) – Continuous search space. Parameters will be sampled from this space which will be used to run trials.
- **max_concurrent** (*int*) – Number of maximum concurrent trials. Defaults to 10.
- **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.

Example

```
>>> space = {
>>>     'width': (0, 20),
>>>     'height': (-100, 100),
>>> }
>>> algo = BayesOptSearch(
>>>     space, max_concurrent=4, metric="mean_loss", mode="min")
```

4.20.3 HyperOpt Search (Tree-structured Parzen Estimators)

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

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

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

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

```
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*, *max_concurrent=10*, *reward_attr=None*, *metric='episode_reward_mean'*, *mode='max'*, *points_to_evaluate=None*, *n_initial_points=20*, *random_state_seed=None*, *gamma=0.25*, ***kwargs*)

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

A wrapper around HyperOpt to provide trial suggestions.

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

Example

```
>>> 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, max_concurrent=4, metric="mean_loss", mode="min",
>>>     points_to_evaluate=current_best_params)
```

4.20.4 SigOpt Search

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

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

```
$ pip install sigopt
```

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

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

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

```
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.SuggestionAlgorithm`

A wrapper around SigOpt to provide trial suggestions.

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

Parameters

- **space** (*list of dict*) – SigOpt configuration. Parameters will be sampled from this configuration and will be used to override parameters generated in the variant generation process.
- **name** (*str*) – Name of experiment. Required by SigOpt.
- **max_concurrent** (*int*) – Number of maximum concurrent trials supported based on the user's SigOpt plan. Defaults to 1.
- **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")
```

4.20.5 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*, *max_concurrent=10*, *reward_attr=None*, *metric='episode_reward_mean'*, *mode='max'*, ***kwargs*)

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

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+).
- **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

```
>>> from nevergrad.optimization import optimizerlib
>>> instrumentation = 1
>>> optimizer = optimizerlib.OnePlusOne(instrumentation, budget=100)
>>> algo = NevergradSearch(optimizer, ["lr"], max_concurrent=4,
>>>                        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)
```

(continues on next page)

(continued from previous page)

```
>>> algo = NevergradSearch(optimizer, None, max_concurrent=4,
>>>                        metric="mean_loss", mode="min")
```

4.20.6 Scikit-Optimize Search

The `SkOptSearch` is a SearchAlgorithm that is backed by Scikit-Optimize 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 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*, *max_concurrent=10*, *reward_attr=None*, *metric='episode_reward_mean'*, *mode='max'*, *points_to_evaluate=None*, *evaluated_rewards=None*, ***kwargs*)

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

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.
- **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.
- **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)

Example

```
>>> from skopt import Optimizer
>>> optimizer = Optimizer([(0,20),(-100,100)])
>>> current_best_params = [[10, 0], [15, -20]]
>>> algo = SkOptSearch(optimizer,
>>>     ["width", "height"],
>>>     max_concurrent=4,
>>>     metric="mean_loss",
>>>     mode="min",
>>>     points_to_evaluate=current_best_params)
```

4.20.7 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*, *max_concurrent=10*, ***kwargs*)
Bases: `ray.tune.suggest.suggestion.SuggestionAlgorithm`

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.
- **`max_concurrent`** (*int*) – Number of maximum concurrent trials. Defaults to 10.
- **`minimize`** (*bool*) – Whether this experiment represents a minimization problem. Defaults to False.

- **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."

Example

```
>>> parameters = [
>>>     {"name": "x1", "type": "range", "bounds": [0.0, 1.0]},
>>>     {"name": "x2", "type": "range", "bounds": [0.0, 1.0]},
>>> ]
>>> algo = AxSearch(parameters=parameters,
>>>     objective_name="hartmann6", max_concurrent=4)
```

4.20.8 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.SuggestionAlgorithm`

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)
```

4.20.9 Contributing a New Algorithm

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

class `ray.tune.suggest.`**`SearchAlgorithm`**

Interface of an event handler API for hyperparameter search.

Unlike TrialSchedulers, SearchAlgorithms will not have the ability to modify the execution (i.e., stop and pause trials).

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

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

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

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

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

Returns Returns a list of trials.

Return type trials (list)

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

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

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

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

Parameters

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

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

Can return True before all trials have finished executing.

Model-Based Suggestion Algorithms

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

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

Abstract class for suggestion-based algorithms.

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

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

Example

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

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

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

Returns

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

Return type dict|None

Example

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

4.21 Tune Package Reference

4.21.1 ray.tune

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

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

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

Parameters

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

`ray.tune.`**`register_trainable`**(*name*, *trainable*)
Register a trainable function or class.

Parameters

- **`name`** (*`str`*) – Name to register.

- **trainable** (*obj*) – Function or tune.Trainable class. Functions must take (config, status_reporter) as arguments and will be automatically converted into a class during registration.

ray.tune.**run**(*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*, *keep_checkpoints_num=None*, *checkpoint_score_attr=None*, *global_checkpoint_period=10*, *export_formats=None*, *max_failures=3*, *restore=None*, *search_alg=None*, *scheduler=None*, *with_server=False*, *server_port=4321*, *verbose=2*, *resume=False*, *queue_trials=False*, *reuse_actors=False*, *trial_executor=None*, *raise_on_failed_trial=True*, *return_trials=False*, *ray_auto_init=True*, *sync_function=None*)

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*) – The stopping criteria. The keys may be any field in the return result of 'train()', whichever is reached first. Defaults to empty dict.
- **config** (*dict*) – Algorithm-specific configuration for Tune variant generation (e.g. env, hyperparams). Defaults to empty dict. Custom search algorithms may ignore this.
- **resources_per_trial** (*dict*) – Machine resources to allocate per trial, e.g. `{"cpu": 64, "gpu": 8}`. Note that GPUs will not be assigned unless you specify them here. Defaults to 1 CPU and 0 GPUs in `Trainable.default_resource_request()`.
- **num_samples** (*int*) – Number of times to sample from the hyperparameter space. Defaults to 1. If *grid_search* is provided as an argument, the grid will be repeated *num_samples* of times.
- **local_dir** (*str*) – Local dir to save training results to. Defaults to `~/ray_results`.
- **upload_dir** (*str*) – Optional URI to sync training results to (e.g. `s3://bucket`).
- **trial_name_creator** (*func*) – Optional function for generating the trial string representation.
- **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 comamnds.
- **sync_to_driver** (*func|str*) – 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 not provided, defaults to using rsync.
- **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.
- **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 from its last checkpoint at least this many times. Only applies if checkpointing is enabled. Setting to -1 will lead to infinite recovery retries. Defaults to 3.
- **restore** (*str*) – Path to checkpoint. Only makes sense to set if running 1 trial. Defaults to None.
- **search_alg** (SearchAlgorithm) – Search Algorithm. Defaults to BasicVariantGenerator.
- **scheduler** (TrialScheduler) – Scheduler for executing the experiment. Choose among FIFO (default), MedianStopping, AsyncHyperBand, and HyperBand.
- **with_server** (*bool*) – Starts a background Tune server. Needed for using the Client API.
- **server_port** (*int*) – Port number for launching TuneServer.
- **verbose** (*int*) – 0, 1, or 2. Verbosity mode. 0 = silent, 1 = only status updates, 2 = status and trial results.
- **resume** (*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.
- **sync_function** – Deprecated. See *sync_to_cloud* and *sync_to_driver*.

Returns List of Trial objects.

Raises TuneError if any trials failed and *raise_on_failed_trial* is True.

Examples

```
>>> tune.run(mytrainable, scheduler=PopulationBasedTraining())
```

```
>>> tune.run(mytrainable, num_samples=5, reuse_actors=True)
```

```
>>> tune.run(
        "PG",
        num_samples=5,
        config={
            "env": "CartPole-v0",
            "lr": tune.sample_from(lambda _: np.random.rand())
        }
    )
```

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

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

Bases: `object`

Tracks experiment specifications.

Implicitly registers the Trainable if needed.

Examples

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

classmethod from_json(*name*, *spec*)
Generates an Experiment object from JSON.

Parameters

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

class ray.tune.**function**(*func*)
Bases: object

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

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

Parameters **func** – A function literal.

class ray.tune.**sample_from**(*func*)
Bases: object

Specify that tune should sample configuration values from this function.

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

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

ray.tune.**uniform**(**args*, ***kwargs*)
A wrapper around np.random.uniform.

ray.tune.**choice**(**args*, ***kwargs*)
A wrapper around np.random.choice.

ray.tune.**randint**(**args*, ***kwargs*)
A wrapper around np.random.randint.

ray.tune.**randn**(**args*, ***kwargs*)
A wrapper around np.random.randn.

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.

class ray.tune.**ExperimentAnalysis**(*experiment_checkpoint_path*, *trials=None*)

Bases: `ray.tune.analysis.experiment_analysis.Analysis`

Analyze results from a Tune experiment.

Parameters **experiment_checkpoint_path** (*str*) – Path to a json file representing an experiment state. Corresponds to Experiment.local_dir/Experiment.name/experiment_state.json

Example

```
>>> tune.run(my_trainable, name="my_exp", local_dir="~/tune_results")
>>> analysis = ExperimentAnalysis(
>>>     experiment_checkpoint_path="~/tune_results/my_exp/state.json")
```

stats()

Returns a dictionary of the statistics of the experiment.

runner_data()

Returns a dictionary of the TrialRunner data.

class ray.tune.**Analysis**(*experiment_dir*)

Bases: `object`

Analyze all results from a directory of experiments.

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].

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/*.

trial_dataframes
List of all dataframes of the trials.

class `ray.tune.`**Trainable**(*config=None*, *logger_creator=None*)
Abstract class for trainable models, functions, etc.

A call to `train()` on a trainable will execute one logical iteration of training. As a rule of thumb, the execution time of one train call should be large enough to avoid overheads (i.e. more than a few seconds), but short enough to report progress periodically (i.e. at most a few minutes).

Calling `save()` should save the training state of a trainable to disk, and `restore(path)` should restore a trainable to the given state.

Generally you only need to implement `_train`, `_save`, and `_restore` here when subclassing Trainable.

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

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*.

classmethod default_resource_request(*config*)
Returns the resource requirement for the given configuration.

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

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

train()
Runs one logical iteration of training.

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

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

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

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

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

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

delete_checkpoint(*checkpoint_dir*)
Removes subdirectory within checkpoint_folder

Parameters **`checkpoint_dir`** – path to checkpoint

`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().

`save_to_object`()
Saves the current model state to a Python object.

It also saves to disk but does not return the checkpoint path.

Returns Object holding checkpoint data.

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

These checkpoints are returned from calls to save().

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

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

These checkpoints are returned from calls to save_to_object().

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

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

Parameters

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

Returns A dict that maps ExportFormats to successfully exported models.

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

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.

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

Returns True if reset was successful else False.

`stop`()
Releases all resources used by this trainable.

`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.

iteration
Current training iteration.

This value is automatically incremented every time *train()* is called and is automatically inserted into the training result dict.

get_config()
Returns configuration passed in by Tune.

_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* to manually trigger termination of this trial or checkpointing of this trial. Note that manual checkpointing only works when subclassing Trainables.

Returns A dict that describes training progress.

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

Parameters **checkpoint_dir** (*str*) – The directory where the checkpoint file must be stored. In a Tune run, this defaults to *<self.logdir>/checkpoint_<ITER>* (which is the same as *local_dir/exp_name/trial_name/checkpoint_<ITER>*).

Returns

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

Return type checkpoint (str | dict)

Examples

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

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

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

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

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

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

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

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

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

Parameters

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

Returns A dict that maps ExportFormats to successfully exported models.

class ray.tune.function_runner.**StatusReporter**(*result_queue*, *continue_semaphore*, *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.

4.21.2 ray.tune.schedulers

class ray.tune.schedulers.**TrialScheduler**

Bases: object

Interface for implementing a Trial Scheduler class.

CONTINUE = 'CONTINUE'

Status for continuing trial execution

PAUSE = 'PAUSE'

Status for pausing trial execution

STOP = 'STOP'

Status for stopping trial execution

on_trial_add(*trial_runner*, *trial*)

Called when a new trial is added to the trial runner.

on_trial_error(*trial_runner*, *trial*)

Notification for the error of trial.

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

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

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

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

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

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

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

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

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

If no trial is ready, return None.

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

class `ray.tune.schedulers.`**HyperBandScheduler**(*time_attr='training_iteration'*, *reward_attr=None*, *metric='episode_reward_mean'*, *mode='max'*, *max_t=81*, *reduction_factor=3*)
Bases: `ray.tune.schedulers.trial_scheduler.FIFOScheduler`

Implements the HyperBand early stopping algorithm.

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

To use this implementation of HyperBand with Tune, all you need to do is specify the max length of time a trial can run *max_t*, the time units *time_attr*, 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

class ray.tune.schedulers.**AsyncHyperBandScheduler**(*time_attr='training_iteration'*, *reward_attr=None*, *metric='episode_reward_mean'*, *mode='max'*, *max_t=100*, *grace_period=1*, *reduction_factor=4*, *brackets=1*)
Bases: ray.tune.schedulers.trial_scheduler.FIFOScheduler

Implements the Async Successive Halving.

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

See https://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.

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

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

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

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

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

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

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

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

ray.tune.schedulers.**ASHAScheduler**
alias of ray.tune.schedulers.async_hyperband.AsyncHyperBandScheduler

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*, *hard_stop=True*, *verbose=True*)

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

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

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

Parameters

- **`time_attr`** (*`str`*) – The training result attr to use for comparing time. Note that you can pass in something non-temporal such as *training_iteration* as a measure of progress, the only requirement is that the attribute should increase monotonically.
- **`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 units are the same as the attribute named by *time_attr*.
- **`min_samples_required`** (*`int`*) – Min samples to compute median over.
- **`hard_stop`** (*`bool`*) – If False, pauses trials instead of stopping them. When all other trials are complete, paused trials will be resumed and allowed to run FIFO.
- **`verbose`** (*`bool`*) – If True, will output the median and best result each time a trial reports. Defaults to True.

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

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

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

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

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

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

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

Simple scheduler that just runs trials in submission order.

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

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

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

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

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

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

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

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

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

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

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

If no trial is ready, return None.

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

class `ray.tune.schedulers.`**PopulationBasedTraining**(*time_attr='time_total_s'*, *reward_attr=None*, *metric='episode_reward_mean'*, *mode='max'*, *perturbation_interval=60.0*, *hyperparam_mutations={}*, *quantile_fraction=0.25*, *resample_probability=0.25*, *custom_explore_fn=None*, *log_config=True*)
Bases: `ray.tune.schedulers.trial_scheduler.FIFOScheduler`

Implements the Population Based Training (PBT) algorithm.

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

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

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

This Tune PBT implementation considers all trials added as part of the PBT population. If the number of trials exceeds the cluster capacity, they will be time-multiplexed as to balance training progress across the population. To run multiple trials, use *tune.run(num_samples=<int>)*.

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.

Example

```
>>> 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)
```

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

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

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

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

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

debug_string()

Returns a human readable message for printing to the console.

class ray.tune.schedulers.**HyperBandForBOHB**(*time_attr='training_iteration'*, *reward_attr=None*, *metric='episode_reward_mean'*, *mode='max'*, *max_t=81*, *reduction_factor=3*)

Bases: `ray.tune.schedulers.hyperband.HyperBandScheduler`

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.

on_trial_add(*trial_runner*, *trial*)

Adds new trial.

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

on_trial_result(*trial_runner*, *trial*, *result*)

If bracket is finished, all trials will be stopped.

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

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

choose_trial_to_run(*trial_runner*)

Fair scheduling within iteration by completion percentage.

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

4.21.3 ray.tune.suggest

class ray.tune.suggest.**SearchAlgorithm**

Bases: `object`

Interface of an event handler API for hyperparameter search.

Unlike TrialSchedulers, SearchAlgorithms will not have the ability to modify the execution (i.e., stop and pause trials).

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

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

add_configurations(*experiments*)

Tracks given experiment specifications.

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

next_trials()

Provides Trial objects to be queued into the TrialRunner.

Returns Returns a list of trials.

Return type trials (list)

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

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

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

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

Parameters

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

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

Can return True before all trials have finished executing.

class ray.tune.suggest.**BasicVariantGenerator**(*shuffle=False*)
Bases: ray.tune.suggest.search.SearchAlgorithm

Uses Tune's variant generation for resolving variables.

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

Example

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

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

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

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

Returns Returns a list of trials.

Return type trials (list)

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

Can return True before all trials have finished executing.

class ray.tune.suggest.**TuneBOHB**(*space*, *bohb_config=None*, *max_concurrent=10*, *metric='neg_mean_loss'*, *mode='max'*)
Bases: ray.tune.suggest.suggestion.SuggestionAlgorithm

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)
```

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

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

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

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

Parameters

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

- **early_terminated** (*bool*) – Defaults to False. True if the trial is stopped while in PAUSED or PENDING state.

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

Bases: ray.tune.suggest.search.SearchAlgorithm

Abstract class for suggestion-based algorithms.

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

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

Example

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

add_configurations(*experiments*)

Chains generator given experiment specifications.

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

next_trials()

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

A batch ends when self._trial_generator returns None.

Returns Returns a list of trials.

Return type trials (list)

_generate_trials(*experiment_spec*, *output_path=''*)

Generates trials with configurations from *_suggest*.

Creates a trial_id that is passed into *_suggest*.

Yields Trial objects constructed according to *spec*

is_finished()

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

Can return True before all trials have finished executing.

_suggest(*trial_id*)

Queries the algorithm to retrieve the next set of parameters.

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

Returns

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

Return type dict|None

Example

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

4.21.4 ray.tune.track

`ray.tune.track.`**`init`**(*ignore_reinit_error=True*, ***session_kwargs*)
Initializes the global trial context for this process.

This creates a TrackSession object and the corresponding hooks for logging.

Examples

```
>>> from ray.tune import track
>>> track.init()
```

`ray.tune.track.`**`shutdown`**()
Cleans up the trial and removes it from the global context.

`ray.tune.track.`**`log`**(***kwargs*)
Applies TrackSession.log to the trial in the current context.

`ray.tune.track.`**`trial_dir`**()
Returns the directory where trial results are saved.

This includes json data containing the session's parameters and metrics.

`class` `ray.tune.track.`**`TrackSession`**(*trial_name=”*, *experiment_dir=None*, *upload_dir=None*, *trial_config=None*, *_tune_reporter=None*)
Manages results for a single session.

Represents a single Trial in an experiment.

`trial_name`
Custom trial name.

Type str

`experiment_dir`
Directory where results for all trials are stored. Each session is stored into a unique directory inside experiment_dir.

Type str

`upload_dir`
Directory to sync results to.

Type str

`trial_config`
Parameters that will be logged to disk.

Type dict

`_tune_reporter`
For rerouting when using Tune. Will not instantiate logging if not None.

Type *StatusReporter*

`log`(***metrics*)
Logs all named arguments specified in *metrics*.

This will log trial metrics locally, and they will be synchronized with the driver periodically through ray.

Parameters **`metrics`** – named arguments with corresponding values to log.

`logdir`
Trial logdir (subdir of given experiment directory)

4.21.5 ray.tune.logger

class `ray.tune.logger.`**`Logger`**(*config*, *logdir*)
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.

`on_result`(*result*)
Given a result, appends it to the existing log.

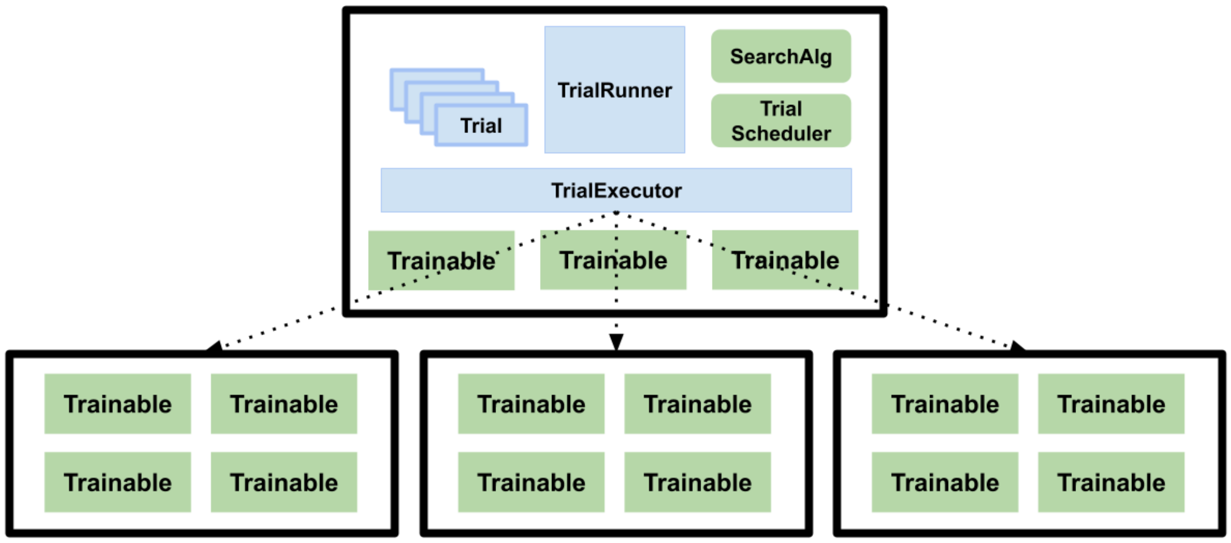
`update_config`(*config*)
Updates the config for logger.

`close`()
Releases all resources used by this logger.

`flush`()
Flushes all disk writes to storage.

4.22 Tune Design Guide

In this part of the documentation, we overview the design and architecture of Tune.



The blue boxes refer to internal components, and green boxes are public-facing. Please refer to the package reference for user-facing APIs.

4.22.1 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.

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"`.

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.

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.

4.23 Tune Examples

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

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

4.23.1 General Examples

- async_hyperband_example: Example of using a Trainable class with AsyncHyperBandScheduler.
- hyperband_example: Example of using a Trainable class with HyperBandScheduler. Also uses the Experiment class API for specifying the experiment configuration. 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.

4.23.2 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.

4.23.3 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.

4.23.4 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.

4.23.5 TensorFlow Examples

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

4.23.6 XGBoost Example

- xgboost_example: Trains a basic XGBoost model with Tune with the function-based API and a XGBoost callback.

4.23.7 LightGBM Example

- lightgbm_example: Trains a basic LightGBM model with Tune with the function-based API and a LightGBM callback.

4.23.8 Contributed Examples

- pbt_tune_cifar10_with_keras: A contributed example of tuning a Keras model on CIFAR10 with the PopulationBasedTraining scheduler.
- genetic_example: Optimizing the michalewicz function using the contributed GeneticSearch search algorithm with AsyncHyperBandScheduler.
- tune_cifar10_gluon: MXNet Gluon example to use Tune with the function-based API on CIFAR-10 dataset.

4.24 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.

4.24.1 Setting up a development environment

If you have Ray installed via pip (`pip install -U [link to wheel]` - you can find the link to the latest wheel here), you can develop Tune locally without needing to compile Ray.

First, you will need your own fork to work on the code. Press the Fork button on the ray project page. Then, clone the project to your machine and connect your repository to the upstream (main project) ray repository.

```
git clone https://github.com/[your username]/ray.git [path to ray directory]
cd [path to ray directory]
git remote add upstream https://github.com/ray-project/ray.git
```

Before continuing, 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.)

Then, run *[path to ray directory]/python/ray/setup-dev.py* (also here on Github) script. This sets up links between the `tune` dir (among other directories) in your local repo and the one bundled with the `ray` package.

As a last step make sure to install all packages required for development of tune. This can be done by running:

```
pip install -r [path to ray directory]/python/ray/tune/requirements-dev.txt
```

4.24.2 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 beginning of the title.

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.

4.24.3 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 *tune/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.

4.24.4 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. Install *yapf==0.23, flake8, flake8-quotes* (these are also in the *requirements-dev.txt* found in `python/ray/tune`). You can run the following locally:

```
ray/scripts/format.sh
```

4.24.5 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.

4.25 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.

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.

4.25.1 RLlib in 60 seconds

The following is a whirlwind overview of RLlib. See also the full table of contents for a more in-depth guide including the list of built-in algorithms.

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
```

```
from ray import tune
from ray.rllib.agents.ppo import PPOTrainer
tune.run(PPOTrainer, config={"env": "CartPole-v0"})
```

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, batch_tensors):
    actions = batch_tensors[SampleBatch.ACTIONS]
    rewards = batch_tensors[SampleBatch.REWARDS]
    return -tf.reduce_mean(policy.action_dist.logp(actions) * 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 `sample_batch_size` 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),
```

(continues on next page)

(continued from previous page)

```
'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 (i.e., synchronous or asynchronous sampling, distributed replay, etc):

Fig. 1: Synchronous Sampling (e.g., A2C, PG, PPO)

Fig. 2: Synchronous Replay (e.g., DQN, DDPG, TD3)

Fig. 3: Asynchronous Sampling (e.g., IMPALA, APPO)

RLlib uses Ray actors to scale these architectures 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.

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.

4.26 RLlib Table of Contents

4.26.1 Training APIs

Fig. 4: Asynchronous Replay (e.g., Ape-X)

  - 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)

4.26.5 Offline Datasets

- Working with Offline Datasets
- Input Pipeline for Supervised Losses
- Input API
- Output API

4.26.6 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

4.26.7 Examples

- Tuned Examples
- Training Workflows
- Custom Envs and Models
- Serving and Offline
- Multi-Agent and Hierarchical
- Community Examples

4.26.8 Development

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

4.26.9 Package Reference

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

4.26.10 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, and `ray timeline` to dump a timeline visualization of tasks to a file.

4.27 RLlib Training APIs

4.27.1 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
```

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 are `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`.

4.27.2 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`.

Common Parameters

The following is a list of the common algorithm hyperparameters:

```
COMMON_CONFIG = {
    # === Debugging ===
    # Whether to write episode stats and videos to the agent log dir
    "monitor": False,
    # Set the ray.rllib.* log level for the agent process and its 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
    # also printed out once at startup at the INFO level).
    "log_level": "INFO",
    # Callbacks that will be run during various phases of training. These all
    # take a single "info" dict as an argument. For episode callbacks, custom
    # metrics can be attached to the episode by updating the episode object's
    # custom metrics dict (see examples/custom_metrics_and_callbacks.py). You
    # may also mutate the passed in batch data in your callback.
    "callbacks": {
        "on_episode_start": None,     # arg: {"env": .., "episode": ...}
        "on_episode_step": None,      # arg: {"env": .., "episode": ...}
        "on_episode_end": None,       # arg: {"env": .., "episode": ...}
        "on_sample_end": None,        # arg: {"samples": .., "worker": ...}
        "on_train_result": None,      # arg: {"trainer": ..., "result": ...}
        "on_postprocess_traj": None,  # arg: {
                                      #   "agent_id": ..., "episode": ...,
                                      #   "pre_batch": (before processing),
                                      #   "post_batch": (after processing),
                                      #   "all_pre_batches": (other agent ids),
                                      # }
    },
    # Whether to attempt to continue training if a worker crashes.
    "ignore_worker_failures": False,
    # Log system resource metrics to results.
    "log_sys_usage": True,

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

    # === Environment ===
    # 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,
```

(continues on next page)

(continued from previous page)

```
    # 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,

    # === Evaluation ===
    # 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.
    "evaluation_num_episodes": 10,
    # Extra arguments to pass to evaluation workers.
    # Typical usage is to pass extra args to evaluation env creator
    # and to disable exploration by computing deterministic actions
    # TODO(kismuz): implement determ. actions and include relevant keys hints
    "evaluation_config": {},

    # === Resources ===
    # Number of actors used for parallelism
    "num_workers": 2,
    # 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,
    # Number of CPUs to allocate per worker.
    "num_cpus_per_worker": 1,
    # Number of GPUs to allocate per worker. This can be fractional.
    "num_gpus_per_worker": 0,
    # Any custom resources to allocate per worker.
    "custom_resources_per_worker": {},
    # Number of CPUs to allocate for the trainer. Note: this only takes effect
    # when running in Tune.
    "num_cpus_for_driver": 1,

    # === Memory quota ===
    # 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 exceeeds the memory quota it will fail.
    # Heap memory to reserve for the trainer process (0 for unlimited). This
    # can be large if your 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,
    # 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
```

(continues on next page)

(continued from previous page)

```
    # by default since it almost never needs to be larger than ~200MB.
    "object_store_memory_per_worker": 0,

    # === Execution ===
    # Number of environments to evaluate vectorwise per worker.
    "num_envs_per_worker": 1,
    # Default sample batch size (unroll length). Batches of this size are
    # collected from workers until train_batch_size is met. When using
    # multiple envs per worker, this is multiplied by num_envs_per_worker.
    "sample_batch_size": 200,
    # Training batch size, if applicable. Should be >= sample_batch_size.
    # Samples batches will be concatenated together to this size for training.
    "train_batch_size": 200,
    # Whether to rollout "complete_episodes" or "truncate_episodes"
    "batch_mode": "truncate_episodes",
    # 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,
    # Configure TF for single-process operation by default
    "tf_session_args": {
        # note: overriden 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,
    },
    # 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 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,
```

(continues on next page)

(continued from previous page)

```
    # 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 iteration
    "min_iter_time_s": 0,
    # Minimum env steps to optimize for per train call. This value does
    # not affect learning, only the length of 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,

    # === 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 *behaviour* 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,

    # === Multiagent ===
    "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,
```

(continues on next page)

(continued from previous page)

```
        # 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
```

4.27.3 Python API

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

Here is an example of the basic usage (for a more complete example, see custom_env.py):

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

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

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()
```

(continues on next page)

(continued from previous page)

```
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]),
    },
)
```

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

```
== Status ==
Using FIFO scheduling algorithm.
Resources requested: 4/4 CPUs, 0/0 GPUs
Result logdir: ~/ray_results/my_experiment
PENDING trials:
 - PPO_CartPole-v0_2_lr=0.0001:     PENDING
RUNNING trials:
 - PPO_CartPole-v0_0_lr=0.01:       RUNNING [pid=21940], 16 s, 4013 ts, 22 rew
 - PPO_CartPole-v0_1_lr=0.001:      RUNNING [pid=21942], 27 s, 8111 ts, 54.7 rew
```

Custom Training Workflows

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

For even finer-grained control over training, you can use RLlib's lower-level building blocks directly to implement fully customized training workflows.

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())
```

(continues on next page)

(continued from previous page)

```
# Same as above
trainer.workers.foreach_worker_with_index(lambda ev, i: ev.get_policy().get_weights())
```

Global Coordination

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

```
from ray.experimental import named_actors

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

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

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

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

Callbacks and Custom Metrics

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

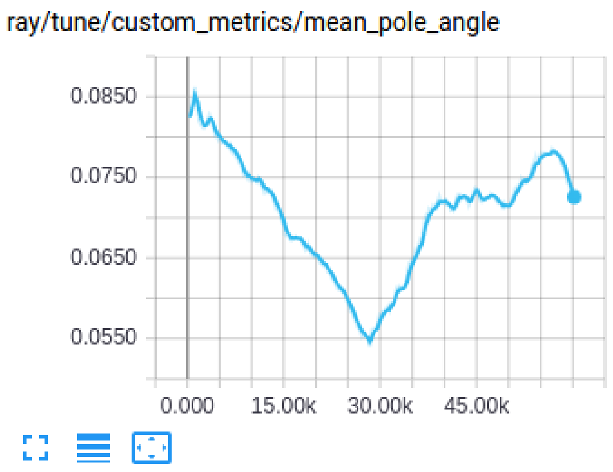
```
def on_episode_start(info):
    print(info.keys())  # -> "env", 'episode"
    episode = info["episode"]
    print("episode {} started".format(episode.episode_id))
    episode.user_data["pole_angles"] = []

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

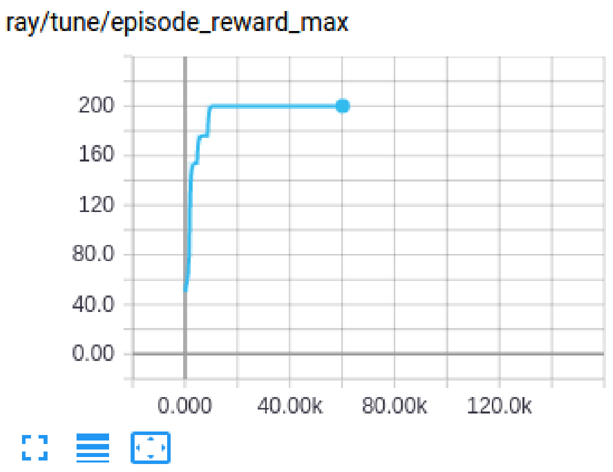
(continues on next page)

(continued from previous page)

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

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

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

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





Example: Curriculum Learning

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

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

```python
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:

```python
import ray
from ray import tune

def on_train_result(info):
    result = info["result"]
    if result["episode_reward_mean"] > 200:
        phase = 2
    elif result["episode_reward_mean"] > 100:
        phase = 1
    else:
        phase = 0
    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": {
```

(continues on next page)

(continued from previous page)

```
            "on_train_result": tune.function(on_train_result),
        },
    },
)
```

4.27.4 Debugging

Gym Monitor

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

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

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

TensorFlow Eager

While RLlib uses TF graph mode for all computations, you can still leverage TF eager to inspect the intermediate state of computations using tf.py_function. Here's an example of using eager mode in a custom RLlib model and loss.

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 "INFO" (default), "DEBUG", "WARN", and "ERROR". This can be used to increase or decrease the verbosity of internal logging. For example:

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

Stack Traces

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

4.27.5 REST API

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

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

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

Parameters

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

Returns Unique string id for the episode.

Return type episode_id (str)

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

Parameters

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

Returns Action from the env action space.

Return type action (obj)

`log_action`(*episode_id*, *observation*, *action*)
Record an observation and (off-policy) action taken.

Parameters

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

`log_returns`(*episode_id*, *reward*, *info=None*)
Record returns from the environment.

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

Parameters

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

`end_episode`(*episode_id*, *observation*)
Record the end of an episode.

Parameters

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

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

REST server than can be launched from a ExternalEnv.

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

Examples

```
>>> class CartpoleServing(ExternalEnv):
       def __init__(self):
           ExternalEnv.__init__(
               self, spaces.Discrete(2),
               spaces.Box(
                   low=-10,
                   high=10,
                   shape=(4,),
                   dtype=np.float32))
       def run(self):
           server = PolicyServer(self, "localhost", 8900)
           server.serve_forever()
>>> register_env("srv", lambda _: CartpoleServing())
>>> pg = PGTrainer(env="srv", config={"num_workers": 0})
>>> while True:
        pg.train()
```

```
>>> client = PolicyClient("localhost:8900")
>>> eps_id = client.start_episode()
>>> action = client.get_action(eps_id, obs)
>>> ...
>>> client.log_returns(eps_id, reward)
>>> ...
>>> client.log_returns(eps_id, reward)
```

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

4.28 RLlib Environments

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

4.28.1 Feature Compatibility Matrix

Algorithm	Discrete Actions	Continuous	Multi-Agent	Model Support
A2C, A3C	**Yes** +parametric	**Yes**	**Yes**	+RNN, +autoreg
PPO, APPO	**Yes** +parametric	**Yes**	**Yes**	+RNN, +autoreg
PG	**Yes** +parametric	**Yes**	**Yes**	+RNN, +autoreg
IMPALA	**Yes** +parametric	**Yes**	**Yes**	+RNN, +autoreg
DQN, Rainbow	**Yes** +parametric	No	**Yes**	
DDPG, TD3	No	**Yes**	**Yes**	
APEX-DQN	**Yes** +parametric	No	**Yes**	
APEX-DDPG	No	**Yes**	**Yes**	
SAC	(todo)	**Yes**	**Yes**	
ES	**Yes**	**Yes**	No	
ARS	**Yes**	**Yes**	No	
QMIX	**Yes**	No	**Yes**	+RNN
MARWIL	**Yes** +parametric	**Yes**	**Yes**	+RNN

4.28.2 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>

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))
```

4.28.3 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.

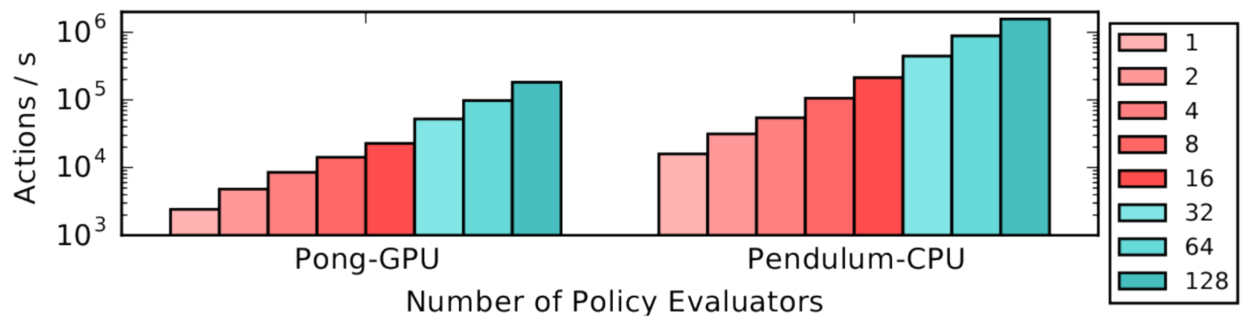
Performance

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.

4.28.4 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.

4.28.5 Multi-Agent and Hierarchical

Note: Learn more about multi-agent reinforcement learning in RLlib by checking out some of the code examples or reading the blog post.

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

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

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

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

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

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

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

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

```
trainer = pg.PGAgent(env="my_multiagent_env", config={
    "multiagent": {
        "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
    },
})

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
```

(continues on next page)

(continued from previous page)

```
    info:
      policy_0:
        cur_lr: 4.999999873689376e-05
        entropy: 0.6833480000495911
        kl: 0.010264254175126553
        policy_loss: -11.95590591430664
        total_loss: 197.7039794921875
        vf_explained_var: 0.0010995268821716309
        vf_loss: 209.6578826904297
      policy_1:
        cur_lr: 4.999999873689376e-05
        entropy: 0.6827034950256348
        kl: 0.01119876280426979
        policy_loss: -8.787769317626953
        total_loss: 88.26161193847656
        vf_explained_var: 0.0005457401275634766
        vf_loss: 97.0471420288086
    policy_reward_mean:
      policy_0: 21.194444444444443
      policy_1: 21.798387096774192
```

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

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.

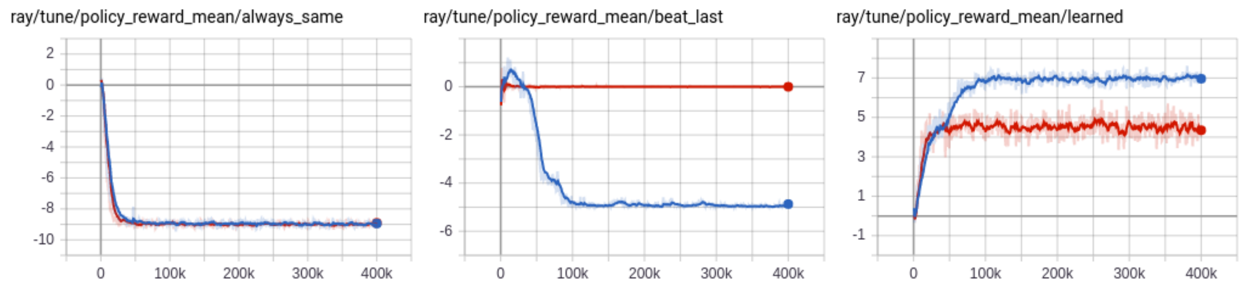


Fig. 5: TensorBoard output of running the rock-paper-scissors example, where a learned policy faces off between a random selection of the same-move and beat-last-move heuristics. Here the performance of heuristic policies vs the learned policy is compared with LSTM enabled (blue) and a plain feed-forward 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.

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.

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(
        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
```

(continues on next page)

(continued from previous page)

```
                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"],
            ... })
        """

        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.

4.28.6 Interfacing with External Agents

In many situations, it does not make sense for an environment to be "stepped" by RLlib. For example, if a policy is to be used in a web serving system, then it is more natural for an agent to query a service that serves policy decisions, and for that service to learn from experience over time. This case also naturally arises with **external simulators** that run independently outside the control of RLlib, but may still want to leverage RLlib for training.

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

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

Logging off-policy actions

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

Data ingest

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

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

See also:

Offline Datasets provide higher-level interfaces for working with offline experience datasets.

4.28.7 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.

4.29 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.

4.29.1 Default Behaviours

Built-in Models and Preprocessors

RLlib picks default models based on a simple heuristic: a vision network for image observations, and a fully connected network for everything else. These models can be configured via the `model` config key, documented in the model catalog. Note that you'll probably have to configure `conv_filters` if your environment observations have custom sizes, e.g., `"model": {"dim": 42, "conv_filters": [[16, [4, 4], 2], [32, [4, 4], 2], [512, [11, 11], 1]]}` for 42x42 observations.

In addition, if you set `"model": {"use_lstm": true}`, then the model output will be further processed by a LSTM cell. More generally, RLlib supports the use of recurrent models for its policy gradient algorithms (A3C, PPO, PG, IMPALA), and RNN support is built into its policy evaluation utilities.

For preprocessors, RLlib tries to pick one of its built-in preprocessor based on the environment's observation space. Discrete observations are one-hot encoded, Atari observations downscaled, and Tuple and Dict observations flattened (these are unflattened and accessible via the `input_dict` parameter in custom models). Note that for Atari, RLlib defaults to using the DeepMind preprocessors, which are also used by the OpenAI baselines library.

Built-in Model Parameters

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

```
MODEL_DEFAULTS = {
    # === Built-in options ===
    # Filter config. List of [out_channels, kernel, stride] for each filter
    "conv_filters": None,
    # Nonlinearity for built-in convnet
    "conv_activation": "relu",
    # Nonlinearity for fully connected net (tanh, relu)
```

(continues on next page)

(continued from previous page)

```
    "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
    # 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 preprocessor to use
    "custom_preprocessor": None,
    # 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": {},
}
```

4.29.2 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)
    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)

Sample implementation for the `MyModelClass` example:

```
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()

Return the value function estimate for the most recent forward pass.

Returns value estimate tensor of shape [BATCH].

Sample implementation for the `MyModelClass` example:

```
def value_function(self):
    return self._value_out
```

`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.

`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`()
Returns the list of variables for this model.

`trainable_variables`()
Returns the list of trainable variables for this model.

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): ...
    def value_function(self): ...

ModelCatalog.register_custom_model("my_model", MyModelClass)

ray.init()
```

(continues on next page)

(continued from previous page)

```
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*, *action_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.

`__init__`(*obs_space*, *action_space*, *num_outputs*, *model_config*, *name*)

Initialize a TFModelV2.

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=())

    # 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)

    # Create the RNN model
    self.rnn_model = tf.keras.Model(
```

(continues on next page)

(continued from previous page)

```
            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()
```

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 (outputs, new_state)

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.

4.29.3 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)

Sample implementation for the `MyModelClass` example:

```
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()

Return the value function estimate for the most recent forward pass.

Returns value estimate tensor of shape [BATCH].

Sample implementation for the `MyModelClass` example:

```
def value_function(self):
    return self._value_out
```

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.

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, if any

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
    },
})
```

4.29.4 Custom Preprocessors

Custom preprocessors should subclass the RLlib preprocessor class and be registered in the model catalog. Note that you can alternatively use gym wrapper classes around your environment instead of preprocessors.

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

class MyPreprocessorClass(Preprocessor):
    def _init_shape(self, obs_space, options):
        return new_shape  # can vary depending on inputs

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

ModelCatalog.register_custom_preprocessor("my_prep", MyPreprocessorClass)

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

4.29.5 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": {
```

(continues on next page)

(continued from previous page)

```
        "custom_action_dist": "my_dist",
    },
})
```

4.29.6 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.

4.29.7 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, 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):
```

(continues on next page)

(continued from previous page)

```
        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"]
        })

        # 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.

4.29.8 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_sampled ~ P(a2 | a1_sampled, 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 | 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()
```

(continues on next page)

(continued from previous page)

```
        # 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])
        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,
```

(continues on next page)

(continued from previous page)

```
            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",
            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.

4.30 RLlib Algorithms

4.30.1 High-throughput architectures

Distributed Prioritized Experience Replay (Ape-X)

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

Fig. 6: 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

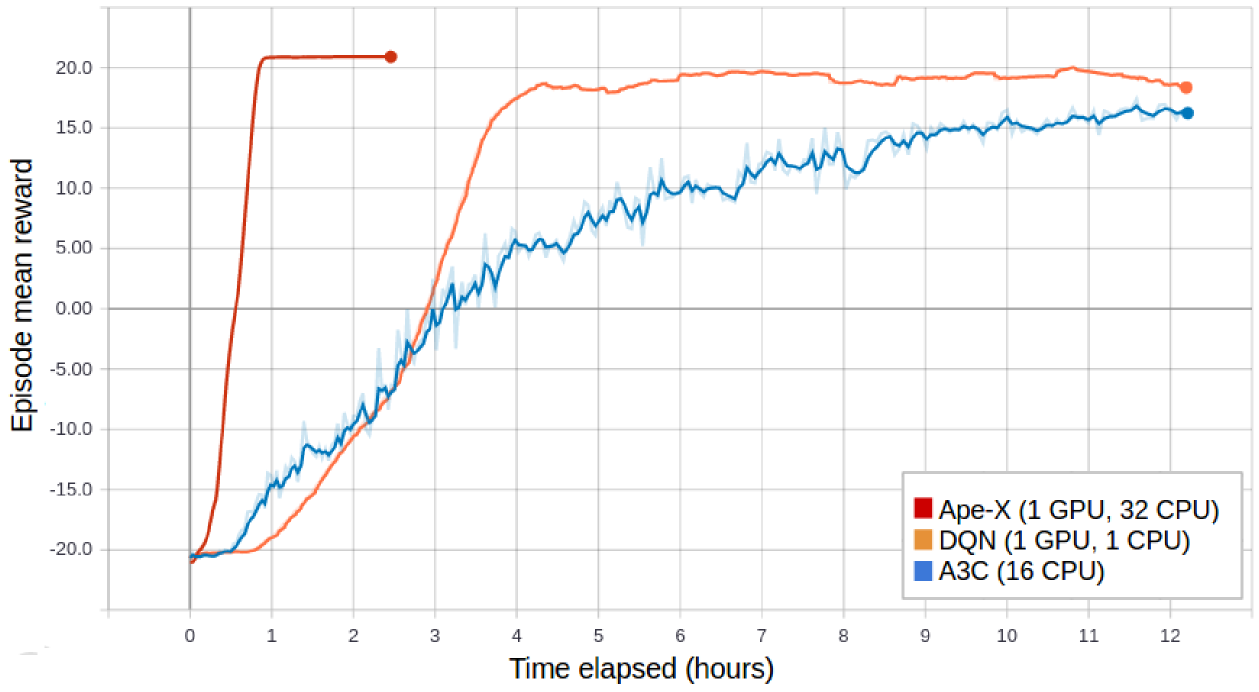


Fig. 7: Ape-X using 32 workers in RLlib vs vanilla DQN (orange) and A3C (blue) on PongNoFrameskip-v4.

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

```
APEX_DEFAULT_CONFIG = merge_dicts(
    DQN_CONFIG,  # see also the options in dqn.py, which are also supported
    {
        "optimizer": 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,
```

(continues on next page)

(continued from previous page)

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

Importance Weighted Actor-Learner Architecture (IMPALA)

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

Fig. 8: 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

IMPALA-specific configs (see also common configs):

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

    # System params.
    #
    # == Overview of data flow in IMPALA ==
```

(continues on next page)

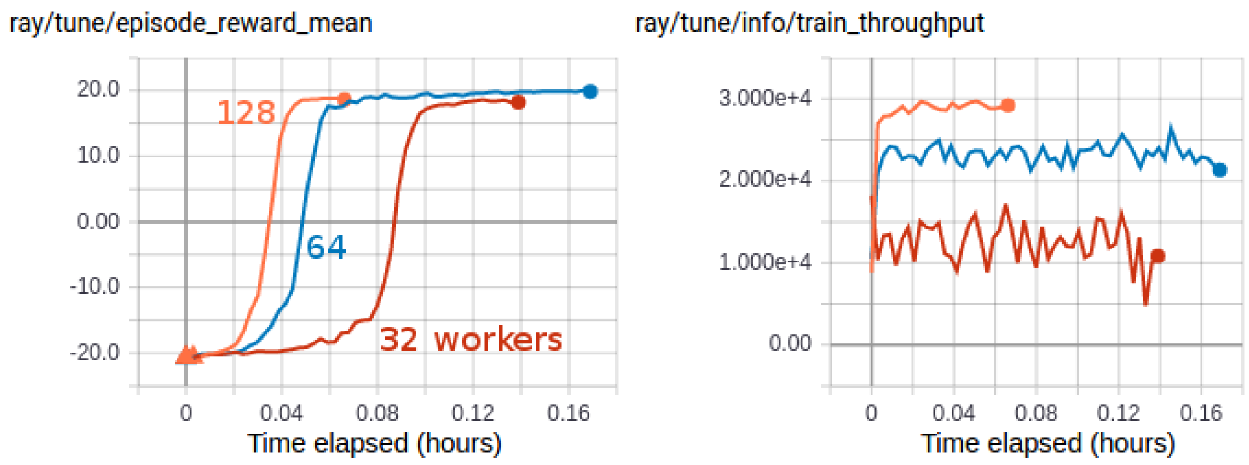


Fig. 9: Multi-GPU IMPALA scales up to solve PongNoFrameskip-v4 in ~3 minutes using a pair of V100 GPUs and 128 CPU workers. The maximum training throughput reached is ~30k transitions per second (~120k environment frames per second).

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
    # 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.

APPO is not always more efficient; it is often better to simply use PPO or IMPALA.

Fig. 10: 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 ==
    # 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,
```

(continues on next page)

(continued from previous page)

```
    # == 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) ==
    "sample_batch_size": 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,
})
```

4.30.2 Gradient-based

Advantage Actor-Critic (A2C, A3C)

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

Fig. 11: A2C architecture

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

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

Atari results @10M steps: more details

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

A3C-specific configs (see also common configs):

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

Deep Deterministic Policy Gradients (DDPG, TD3)

[paper] [implementation] DDPG is implemented similarly to DQN (below). The algorithm can be scaled by increasing the number of workers, switching to AsyncGradientsOptimizer, or using Ape-X. The improvements from TD3 are available as `TD3`.

Fig. 12: 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,
```

(continues on next page)

(continued from previous page)

```
    # 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. Auto-enabled if a custom model is set.
    "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.
    "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 ===
    # Turns on annealing schedule for exploration noise. Exploration is
    # annealed from 1.0 to exploration_final_eps over schedule_max_timesteps
    # scaled by exploration_fraction. Original DDPG and TD3 papers do not
    # anneal noise, so this is False by default.
    "exploration_should_anneal": False,
    # Max num timesteps for annealing schedules.
    "schedule_max_timesteps": 100000,
    # Number of env steps to optimize for before returning
    "timesteps_per_iteration": 1000,
    # Fraction of entire training period over which the exploration rate is
    # annealed
    "exploration_fraction": 0.1,
    # Final scaling multiplier for action noise (initial is 1.0)
    "exploration_final_scale": 0.02,
    # valid values: "ou" (time-correlated, like original DDPG paper),
```

(continues on next page)

(continued from previous page)

```
    # "gaussian" (IID, like TD3 paper)
    "exploration_noise_type": "ou",
    # OU-noise scale; this can be used to scale down magnitude of OU noise
    # before adding to actions (requires "exploration_noise_type" to be "ou")
    "exploration_ou_noise_scale": 0.1,
    # theta for OU
    "exploration_ou_theta": 0.15,
    # sigma for OU
    "exploration_ou_sigma": 0.2,
    # gaussian stddev of act noise for exploration (requires
    # "exploration_noise_type" to be "gaussian")
    "exploration_gaussian_sigma": 0.1,
    # If True parameter space noise will be used for exploration
    # See https://blog.openai.com/better-exploration-with-parameter-noise/
    "parameter_noise": False,
    # Until this many timesteps have elapsed, the agent's policy will be
    # ignored & it will instead take uniform random actions. Can be used in
    # conjunction with learning_starts (which controls when the first
    # optimization step happens) to decrease dependence of exploration &
    # optimization on initial policy parameters. Note that this will be
    # disabled when the action noise scale is set to 0 (e.g during evaluation).
    "pure_exploration_steps": 1000,
    # Extra configuration that disables exploration.
    "evaluation_config": {
        "exploration_fraction": 0,
        "exploration_final_eps": 0,
    },

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

(continues on next page)

(continued from previous page)

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

    # === 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,
})
```

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. 13: 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.

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,
    # Whether to use double dqn
    "double_q": True,
    # Postprocess model outputs with these hidden layers to compute the
    # state and action values. See also the model config in catalog.py.
    "hiddens": [256],
    # N-step Q learning
    "n_step": 1,

    # === Exploration ===
    # Max num timesteps for annealing schedules. Exploration is annealed from
    # 1.0 to exploration_fraction over this number of timesteps scaled by
    # exploration_fraction
    "schedule_max_timesteps": 100000,
    # Minimum env steps to optimize for per train call. This value does
    # not affect learning, only the length of iterations.
    "timesteps_per_iteration": 1000,
    # Fraction of entire training period over which the exploration rate is
    # annealed
    "exploration_fraction": 0.1,
    # Final value of random action probability
    "exploration_final_eps": 0.02,
    # Update the target network every `target_network_update_freq` steps.
    "target_network_update_freq": 500,
    # Use softmax for sampling actions. Required for off policy estimation.
    "soft_q": False,
    # Softmax temperature. Q values are divided by this value prior to softmax.
    # Softmax approaches argmax as the temperature drops to zero.
    "softmax_temp": 1.0,
    # If True parameter space noise will be used for exploration
    # See https://blog.openai.com/better-exploration-with-parameter-noise/
    "parameter_noise": False,
    # Extra configuration that disables exploration.
    "evaluation_config": {
        "exploration_fraction": 0,
```

(continues on next page)

(continued from previous page)

```
        "exploration_final_eps": 0,
    },

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

    # === Optimization ===
    # Learning rate for adam optimizer
    "lr": 5e-4,
    # 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_norm_clipping": 40,
    # How many steps of the model to sample before learning starts.
    "learning_starts": 1000,
    # Update the replay buffer with this many samples at once. Note that
    # this setting applies per-worker if num_workers > 1.
    "sample_batch_size": 4,
    # Size of a batched sampled from replay buffer for training. Note that
    # if async_updates is set, then each worker returns gradients for a
    # batch of this size.
    "train_batch_size": 32,

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

Policy Gradients

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

Fig. 14: 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 PyTorch as backend
    "use_pytorch": False,
})
```

Proximal Policy Optimization (PPO)

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

Fig. 15: 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

PPO-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # If true, use the Generalized Advantage Estimator (GAE)
    # with a value function, see https://arxiv.org/pdf/1506.02438.pdf.
    "use_gae": True,
```

(continues on next page)

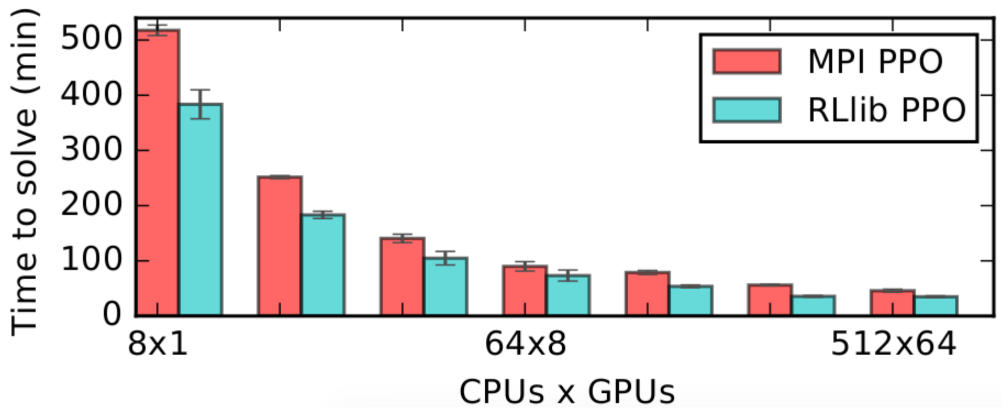


Fig. 16: RLlib's multi-GPU PPO scales to multiple GPUs and hundreds of CPUs on solving the Humanoid-v1 task. Here we compare against a reference MPI-based implementation.

(continued from previous page)

```
    # GAE(lambda) parameter
    "lambda": 1.0,
    # Initial coefficient for KL divergence
    "kl_coeff": 0.2,
    # Size of batches collected from each worker
    "sample_batch_size": 200,
    # Number of timesteps collected for each SGD round
    "train_batch_size": 4000,
    # Total SGD batch size across all devices for SGD
    "sgd_minibatch_size": 128,
    # 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,
    # 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. It's important to 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"
```

(continues on next page)

(continued from previous page)

```
    "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 does
    # not support minibatches.
    "simple_optimizer": False,
})
```

Soft Actor Critic (SAC)

[paper] [implementation]

Fig. 17: 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`, and currently has no support for non-continuous action distributions. It is also currently *experimental*.

Tuned examples: Pendulum-v0

SAC-specific configs (see also common configs):

```
DEFAULT_CONFIG = with_common_config({
    # === Model ===
    "twin_q": True,
    "use_state_preprocessor": False,
    "policy": "GaussianLatentSpacePolicy",
    # RLlib model options for the Q function
    "Q_model": {
        "hidden_activation": "relu",
        "hidden_layer_sizes": (256, 256),
    },
    # RLlib model options for the policy function
    "policy_model": {
        "hidden_activation": "relu",
        "hidden_layer_sizes": (256, 256),
    },

    # === Learning ===
    # Update the target by \tau * policy + (1-\tau) * target_policy
    "tau": 5e-3,
    # Target entropy lower bound. This is the inverse of reward scale,
    # and will be optimized automatically.
    "target_entropy": "auto",
    # Disable setting done=True at end of episode.
    "no_done_at_end": True,
    # N-step target updates
    "n_step": 1,

    # === Evaluation ===
    # The evaluation stats will be reported under the "evaluation" metric key.
    "evaluation_interval": 1,
    # Number of episodes to run per evaluation period.
    "evaluation_num_episodes": 1,
```

(continues on next page)

(continued from previous page)

```
    # Extra configuration that disables exploration.
    "evaluation_config": {
        "exploration_enabled": False,
    },

    # === Exploration ===
    # Number of env steps to optimize for before returning
    "timesteps_per_iteration": 1000,
    "exploration_enabled": True,

    # === 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.
    # TODO(hartikainen): Make sure this works or remove the option.
    "prioritized_replay": False,
    "prioritized_replay_alpha": 0.6,
    "prioritized_replay_beta": 0.4,
    "prioritized_replay_eps": 1e-6,
    "beta_annealing_fraction": 0.2,
    "final_prioritized_replay_beta": 0.4,
    "compress_observations": False,

    # === Optimization ===
    "optimization": {
        "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_norm_clipping": None,
    # How many steps of the model to sample before learning starts.
    "learning_starts": 1500,
    # Update the replay buffer with this many samples at once. Note that this
    # setting applies per-worker if num_workers > 1.
    "sample_batch_size": 1,
    # Size of a batched sampled from replay buffer for training. Note that
    # if async_updates is set, then each worker returns gradients for a
    # batch of this size.
    "train_batch_size": 256,
    # 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,
```

(continues on next page)

(continued from previous page)

```
    # Prevent iterations from going lower than this time span
    "min_iter_time_s": 1,

    # TODO(ekl) these are unused; remove them from sac config
    "per_worker_exploration": False,
    "exploration_fraction": 0.1,
    "schedule_max_timesteps": 100000,
    "exploration_final_eps": 0.02,
})
```

4.30.3 Derivative-free

Augmented Random Search (ARS)

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

Tuned examples: CartPole-v0, Swimmer-v2

ARS-specific configs (see also common configs):

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

Evolution Strategies

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

Tuned examples: Humanoid-v1

Scalability:

ES-specific configs (see also common configs):

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

(continues on next page)

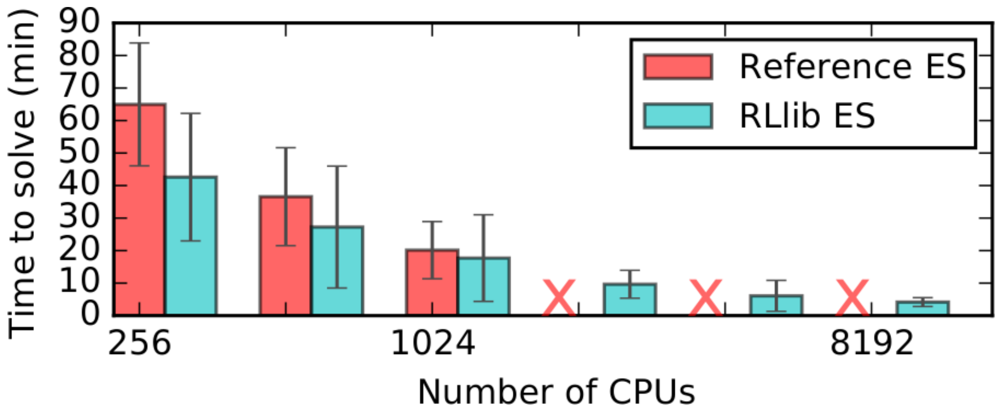


Fig. 18: RLlib's ES implementation scales further and is faster than a reference Redis implementation on solving the Humanoid-v1 task.

(continued from previous page)

```
    "noise_size": 250000000,
    "report_length": 10,
})
```

QMIX Monotonic Value Factorisation (QMIX, VDN, IQN)

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

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

Tuned examples: Two-step game

QMIX-specific configs (see also common configs):

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

    # === 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,

```

(continues on next page)

(continued from previous page)

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

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

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

    # === Parallelism ===
    # Number of workers for collecting samples with. This only makes sense
    # to increase if your environment is particularly slow to sample, or if
    # you"re using the Async or Ape-X optimizers.
    "num_workers": 0,
    # 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 wsjeon/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 state 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 ===
    # Learning rate for the critic (Q-function) optimizer.
```

(continues on next page)

(continued from previous page)

```
    "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.
    "sample_batch_size": 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
    "postprocess_inputs": True,
    # Whether to rollout "complete_episodes" or "truncate_episodes"
```

(continues on next page)

(continued from previous page)

```
    "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,
})
```

4.31 RLlib Offline Datasets

4.31.1 Working with Offline Datasets

RLlib's offline dataset APIs enable working with experiences read from offline storage (e.g., disk, cloud storage, streaming systems, HDFS). For example, you might want to read experiences saved from previous training runs, or gathered from policies deployed in web applications. You can also log new agent experiences produced during online training for future use.

RLlib represents trajectory sequences (i.e., `(s, a, r, s', ...)` tuples) with SampleBatch objects. Using a batch format enables efficient encoding and compression of experiences. During online training, RLlib uses policy evaluation actors to generate batches of experiences in parallel using the current policy. RLlib also uses this same batch format for reading and writing experiences to offline storage.

Example: Training on previously saved experiences

Note: For custom models and enviroments, you'll need to use the Python API.

In this example, we will save batches of experiences generated during online training to disk, and then leverage this saved data to train a policy offline using DQN. First, we run a simple policy gradient algorithm for 100k steps with `"output": "/tmp/cartpole-out"` to tell RLlib to write simulation outputs to the `/tmp/cartpole-out` directory.

```
$ rllib train
    --run=PG \
    --env=CartPole-v0 \
    --config='{"output": "/tmp/cartpole-out", "output_max_file_size": 5000000}' \
    --stop='{"timesteps_total": 100000}'
```

The experiences will be saved in compressed JSON batch format:

```
$ ls -l /tmp/cartpole-out
total 11636
-rw-rw-r-- 1 eric eric 5022257 output-2019-01-01_15-58-57_worker-0_0.json
-rw-rw-r-- 1 eric eric 5002416 output-2019-01-01_15-59-22_worker-0_1.json
-rw-rw-r-- 1 eric eric 1881666 output-2019-01-01_15-59-47_worker-0_2.json
```

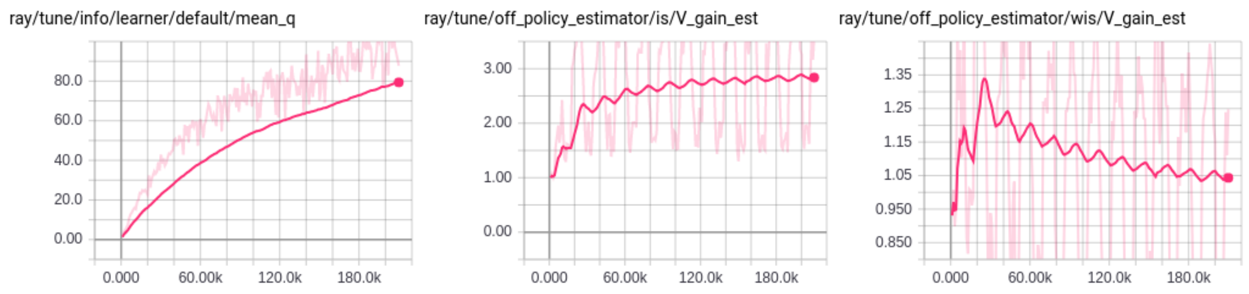
Then, we can tell DQN to train using these previously generated experiences with `"input": "/tmp/cartpole-out"`. We disable exploration since it has no effect on the input:

```
$ rllib train \
    --run=DQN \
    --env=CartPole-v0 \
    --config='{
        "input": "/tmp/cartpole-out",
        "input_evaluation": [],
        "exploration_final_eps": 0,
        "exploration_fraction": 0}'
```

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"],
        "soft_q": true,
        "softmax_temp": 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:

```python
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

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("/tmp/demo-out")

    # You normally wouldn't want to manually create sample batches if a
    # simulator is available, but let's do it anyways for example purposes:
    env = gym.make("CartPole-v0")

    # 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))
            obs = new_obs
```

(continues on next page)

(continued from previous page)

```
            prev_action = action
            prev_reward = rew
            t += 1
        writer.write(batch_builder.build_and_reset())
```

On-policy algorithms and experience postprocessing

RLlib assumes that input batches are of postprocessed experiences. This isn't typically critical for off-policy algorithms (e.g., DQN's post-processing is only needed if `n_step > 1` or `worker_side_prioritization: True`). For off-policy algorithms, you can also safely set the `postprocess_inputs:  True` config to auto-postprocess data.

However, for on-policy algorithms like PPO, you'll need to pass in the extra values added during policy evaluation and postprocessing to `batch_builder.add_values()`, e.g., `logits`, `vf_preds`, `value_target`, and `advantages` for PPO. This is needed since the calculation of these values depends on the parameters of the *behaviour* policy, which RLlib does not have access to in the offline setting (in online training, these values are automatically added during policy evaluation).

Note that for on-policy algorithms, you'll also have to throw away experiences generated by prior versions of the policy. This greatly reduces sample efficiency, which is typically undesirable for offline training, but can make sense for certain applications.

Mixing simulation and offline data

RLlib supports multiplexing inputs from multiple input sources, including simulation. For example, in the following example we read 40% of our experiences from `/tmp/cartpole-out`, 30% from `hdfs:/archive/cartpole`, and the last 30% is produced via policy evaluation. Input sources are multiplexed using np.random.choice:

```
$ rllib train \
    --run=DQN \
    --env=CartPole-v0 \
    --config='{
        "input": {
            "/tmp/cartpole-out": 0.4,
            "hdfs:/archive/cartpole": 0.3,
            "sampler": 0.3,
        },
        "exploration_final_eps": 0,
        "exploration_fraction": 0}'
```

Scaling I/O throughput

Similar to scaling online training, you can scale offline I/O throughput by increasing the number of RLlib workers via the `num_workers` config. Each worker accesses offline storage independently in parallel, for linear scaling of I/O throughput. Within each read worker, files are chosen in random order for reads, but file contents are read sequentially.

4.31.2 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.

4.31.3 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 *behaviour* 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.

4.31.4 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.

4.32 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.

4.32.1 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,
```

(continues on next page)

(continued from previous page)

```
                    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, batch_tensors):
    actions = batch_tensors[SampleBatch.ACTIONS]
    rewards = batch_tensors[SampleBatch.REWARDS]
    return -tf.reduce_mean(policy.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 `policy.action_dist` object is an ActionDistribution that represents 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
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(
```

(continues on next page)

(continued from previous page)

```
        sample_batch, 0.0, policy.config["gamma"], use_gae=False)

def policy_gradient_loss(policy, batch_tensors):
    actions = batch_tensors[SampleBatch.ACTIONS]
    advantages = batch_tensors[Postprocessing.ADVANTAGES]
    return -tf.reduce_mean(policy.action_dist.logp(actions) * 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 `batch_tensors[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,
    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,
```

(continues on next page)

(continued from previous page)

```
        sgd_batch_size=config["sgd_minibatch_size"],
        num_sgd_iter=config["num_sgd_iter"],
        num_gpus=config["num_gpus"],
        sample_batch_size=config["sample_batch_size"],
        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:

        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 `KLCoeffMixin`, 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, KLCoeffMixin, 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, batch_tensors):
    policy.explained_variance = explained_variance(
        batch_tensors[Postprocessing.VALUE_TARGETS], policy.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_tensors[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.value_function,
        BEHAVIOUR_LOGITS: policy.model.outputs,
    }
```

`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:
        policy.var_list = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                            tf.get_variable_scope().name)
        grads = tf.gradients(loss, policy.var_list)
        policy.grads, _ = tf.clip_by_global_norm(grads,
                                                 policy.config["grad_clip"])
        clipped_grads = list(zip(policy.grads, policy.var_list))
        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., `KLCoeffMixin`, `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)
    KLCoeffMixin.__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 SimpleQ-Model. 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"],
```

(continues on next page)

(continued from previous page)

```
        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

While RLlib runs all TF operations in graph mode, you can still leverage TensorFlow eager using tf.py_function. However, note that eager and non-eager tensors cannot be mixed within the `py_function`. Here's an example of embedding eager execution within a policy loss function:

```
def eager_loss(policy, batch_tensors):
    """Example of using embedded eager execution in a custom loss.

    Here `compute_penalty` prints the actions and rewards for debugging, and
    also computes a (dummy) penalty term to add to the loss.
    """

    def compute_penalty(actions, rewards):
        penalty = tf.reduce_mean(tf.cast(actions, tf.float32))
        if random.random() > 0.9:
            print("The eagerly computed penalty is", penalty, actions, rewards)
        return penalty

    actions = batch_tensors[SampleBatch.ACTIONS]
    rewards = batch_tensors[SampleBatch.REWARDS]
    penalty = tf.py_function(
        compute_penalty, [actions, rewards], Tout=tf.float32)

    return penalty - tf.reduce_mean(policy.action_dist.logp(actions) * rewards)
```

You can find a runnable file for the above eager execution example here.

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):

```python
from ray.rllib.policy.sample_batch import SampleBatch
from ray.rllib.policy.torch_policy_template import build_torch_policy

def policy_gradient_loss(policy, batch_tensors):
    logits, _, values, _ = policy.model({
        SampleBatch.CUR_OBS: batch_tensors[SampleBatch.CUR_OBS]
    }, [])
    action_dist = policy.dist_class(logits)
    log_probs = action_dist.logp(batch_tensors[SampleBatch.ACTIONS])
    return -batch_tensors[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:

```python
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_tensors`. We imperatively execute the forward pass by calling `policy.model()` on the observations followed by `policy.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:

```python
def actor_critic_loss(policy, batch_tensors):
    logits, _, values, _ = policy.model({
        SampleBatch.CUR_OBS: batch_tensors[SampleBatch.CUR_OBS]
    }, [])
    dist = policy.dist_class(logits)
    log_probs = dist.logp(batch_tensors[SampleBatch.ACTIONS])
    policy.entropy = 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:

```python
def loss_and_entropy_stats(policy, batch_tensors):
    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_tensors[SampleBatch.VF_PREDS]`:

```
def model_value_predictions(policy, input_dict, state_batches, model_out):
    return {SampleBatch.VF_PREDS: model_out[2].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 _value(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 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_policy import PPOTFPolicy

CustomPolicy = PPOTFPolicy.with_updates(
    name="MyCustomPPOTFPolicy",
    loss_fn=some_custom_loss_fn)

CustomTrainer = PPOTrainer.with_updates(
    default_policy=CustomPolicy)
```

4.32.2 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)
```

4.32.3 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)

while True:
    optimizer.step()
```

4.32.4 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 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})
```

4.33 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.

4.33.1 Tuned Examples

- **Tuned examples**: Collection of tuned algorithm hyperparameters.
- **Atari benchmarks**: Collection of reasonably optimized Atari results.

4.33.2 Training Workflows

- **Custom training workflows**: Example of how to use Tune's support for custom training functions to implement custom training workflows.
- **Curriculum learning**: Example of how to adjust the configuration of an environment over time.
- **Custom metrics**: Example of how to output custom training metrics to TensorBoard.
- **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.

4.33.3 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.

4.33.4 Serving and Offline

- **CartPole server**: Example of online serving of predictions for a simple CartPole policy.
- **Saving experiences**: Example of how to externally generate experience batches in RLlib-compatible format.

4.33.5 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.

4.33.6 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.
- **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.

4.34 RLlib Development

4.34.1 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. 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.)

4.34.2 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`).

4.34.3 Features

Feature development and upcoming priorities are tracked on the RLlib project board (note that this may not include all development efforts). For discussion of issues and new features, we use the Ray dev list and GitHub issues page.

4.34.4 Benchmarks

A number of training run results are available in the rl-experiments repo, and there is also a list of working hyper-parameter configurations in tuned_examples. Benchmark results are extremely valuable to the community, so if you happen to have results that may be of interest, consider making a pull request to either repo.

4.34.5 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
```

(continues on next page)

(continued from previous page)

```

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_trainer,
    "contrib/RandomAgent2": _import_random_trainer_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
```

4.35 RLlib Package Reference

4.35.1 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

`compute_actions`(*obs_batch*, *state_batches*, *prev_action_batch=None*, *prev_reward_batch=None*, *info_batch=None*, *episodes=None*, ***kwargs*)

Compute actions for the current policy.

Parameters

- **obs_batch** (*np.ndarray*) – batch of observations
- **state_batches** (*list*) – list of RNN state input batches, if any
- **prev_action_batch** (*np.ndarray*) – batch of previous action values
- **prev_reward_batch** (*np.ndarray*) – batch of previous rewards
- **info_batch** (*info*) – batch of info objects

- **episodes** (*list*) – MultiAgentEpisode for each obs in obs_batch. This provides access to all of the internal episode state, which may be useful for model-based or multiagent algorithms.
- **kwargs** – forward compatibility placeholder

Returns

batch of output actions, with shape like [BATCH_SIZE, ACTION_SHAPE].

state_outs (list): list of RNN state output batches, if any, with shape like [STATE_SIZE, BATCH_SIZE].

info (dict): dictionary of extra feature batches, if any, with shape like {"f1": [BATCH_SIZE, . . .], "f2": [BATCH_SIZE, . . .]}.

Return type actions (np.ndarray)

compute_single_action(*obs*, *state*, *prev_action=None*, *prev_reward=None*, *info=None*, *episode=None*, *clip_actions=False*, ***kwargs*)

Unbatched version of compute_actions.

Parameters

- **obs** (*obj*) – single observation
- **state_batches** (*list*) – list of RNN state inputs, if any
- **prev_action** (*obj*) – previous action value, if any
- **prev_reward** (*int*) – previous reward, if any
- **info** (*dict*) – info object, if any
- **episode** (*MultiAgentEpisode*) – this provides access to all of the internal episode state, which may be useful for model-based or multi-agent algorithms.
- **clip_actions** (*bool*) – should the action be clipped
- **kwargs** – forward compatibility placeholder

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

Return type actions (obj)

postprocess_trajectory(*sample_batch*, *other_agent_batches=None*, *episode=None*)

Implements algorithm-specific trajectory postprocessing.

This will be called on each trajectory fragment computed during policy evaluation. Each fragment is guaranteed to be only from one episode.

Parameters

- **sample_batch** (*SampleBatch*) – batch of experiences for the policy, which will contain at most one episode trajectory.
- **other_agent_batches** (*dict*) – In a multi-agent env, this contains a mapping of agent ids to (policy, 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_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.

class ray.rllib.policy.**TFPolicy**(*observation_space*, *action_space*, *sess*, *obs_input*, *action_sampler*, *loss*, *loss_inputs*, *model=None*, *action_logp=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*)
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, action_sampler, loss, loss_inputs)
```

```
>>> print(policy.compute_actions([1, 0, 2]))
(array([0, 1, 1]), [], {})
```

```
>>> print(policy.postprocess_trajectory(SampleBatch({...})))
SampleBatch({"action": ..., "advantages": ..., ...})
```

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*, ***kwargs*)
Compute actions for the current policy.

Parameters

- **obs_batch** (*np.ndarray*) – batch of observations
- **state_batches** (*list*) – list of RNN state input batches, if any
- **prev_action_batch** (*np.ndarray*) – batch of previous action values
- **prev_reward_batch** (*np.ndarray*) – batch of previous rewards
- **info_batch** (*info*) – batch of info objects
- **episodes** (*list*) – MultiAgentEpisode for each obs in obs_batch. This provides access to all of the internal episode state, which may be useful for model-based or multiagent algorithms.
- **kwargs** – forward compatibility placeholder

Returns

batch of output actions, with shape like [BATCH_SIZE, ACTION_SHAPE].

state_outs (list): list of RNN state output batches, if any, with shape like [STATE_SIZE, BATCH_SIZE].

info (dict): dictionary of extra feature batches, if any, with shape like {"f1": [BATCH_SIZE, . . .], "f2": [BATCH_SIZE, . . .]}.

Return type actions (np.ndarray)

compute_gradients(*postprocessed_batch*)

Computes gradients against a batch of experiences.

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_weights()

Returns model weights.

Returns Serializable copy or view of model weights

Return type weights (obj)

set_weights(*weights*)

Sets model weights.

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

export_model(*export_dir*)
Export tensorflow graph to export_dir for serving.

export_checkpoint(*export_dir*, *filename_prefix='model'*)
Export tensorflow checkpoint to export_dir.

copy(*existing_inputs*)
Creates a copy of self using existing input placeholders.

Optional, only required to work with the multi-GPU optimizer.

extra_compute_action_feed_dict()
Extra dict to pass to the compute actions session run.

extra_compute_action_fetches()
Extra values to fetch and return from compute_actions().

By default we only return action probability info (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.

class ray.rllib.policy.**TorchPolicy**(*observation_space*, *action_space*, *model*, *loss*, *action_distribution_class*)
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

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

Type Lock

compute_actions(*obs_batch*, *state_batches=None*, *prev_action_batch=None*, *prev_reward_batch=None*, *info_batch=None*, *episodes=None*, ***kwargs*)
Compute actions for the current policy.

Parameters

- **obs_batch** (*np.ndarray*) – batch of observations
- **state_batches** (*list*) – list of RNN state input batches, if any

- **prev_action_batch** (*np.ndarray*) – batch of previous action values
- **prev_reward_batch** (*np.ndarray*) – batch of previous rewards
- **info_batch** (*info*) – batch of info objects
- **episodes** (*list*) – MultiAgentEpisode for each obs in obs_batch. This provides access to all of the internal episode state, which may be useful for model-based or multiagent algorithms.
- **kwargs** – forward compatibility placeholder

Returns

batch of output actions, with shape like [BATCH_SIZE, ACTION_SHAPE].

state_outs (list): list of RNN state output batches, if any, with shape like [STATE_SIZE, BATCH_SIZE].

info (dict): dictionary of extra feature batches, if any, with shape like {"f1": [BATCH_SIZE, . . .], "f2": [BATCH_SIZE, . . .]}.

Return type actions (np.ndarray)

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

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

extra_grad_process()
Allow subclass to do extra processing on gradients and return processing info.

extra_action_out(*input_dict*, *state_batches*, *model*)
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.

extra_grad_info(*batch_tensors*)
Return dict of extra grad info.

optimizer()
Custom PyTorch optimizer to use.

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_action_feed_fn=None*, *extra_learn_fetches_fn=None*, *extra_learn_feed_fn=None*, *before_init=None*, *before_loss_init=None*, *after_init=None*, *make_model=None*, *action_sampler_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 (to be implemented), the following functions will be run repeatedly on each eager execution: loss_fn, 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 the policy, and dict of experience tensor placeholdes
- **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_action_feed_fn** (*func*) – optional function that returns a feed dict to also feed to TF when computing actions
- **extra_learn_fetches_fn** (*func*) – optional function that returns a dict of extra values to fetch and return when learning on a batch
- **extra_learn_feed_fn** (*func*) – optional function that returns a feed dict to also feed to TF 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** (*func*) – optional function that returns a tuple of action and action prob tensors given (policy, model, input_dict, obs_space, action_space, config). If not specified, a default action distribution will be used.
- **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

`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*, *make_model_and_action_dist=None*, *mixins=None*)

Helper function for creating a torch policy at runtime.

Parameters

- **name** (*str*) – name of the policy (e.g., "PPOTorchPolicy")
- **loss_fn** (*func*) – function that returns a loss tensor the policy, and dict of experience tensor placeholders
- **get_default_config** (*func*) – optional function that returns the default config to merge with any overrides
- **stats_fn** (*func*) – optional function that returns a dict of values given the policy and batch input tensors
- **postprocess_fn** (*func*) – optional experience postprocessing function that takes the same args as Policy.postprocess_trajectory()
- **extra_action_out_fn** (*func*) – optional function that returns a dict of extra values to include in experiences
- **extra_grad_process_fn** (*func*) – optional function that is called after gradients are computed and returns processing info
- **optimizer_fn** (*func*) – optional function that returns a torch optimizer given the policy and config
- **before_init** (*func*) – optional function to run at the beginning of policy 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_and_action_dist** (*func*) – optional func that takes the same arguments as policy init and returns a tuple of model instance and torch action distribution class. If not specified, the default model and action dist from the catalog will be used
- **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

Returns a TorchPolicy instance that uses the specified args

4.35.2 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 rllib.ExternalEnv => rllib.BaseEnv

action_space

Action space. This must be defined for single-agent envs. Multi-agent envs can set this to None.

Type gym.Space

observation_space

Observation space. This must be defined for single-agent envs. Multi-agent envs can set this to None.

Type gym.Space

Examples

```
>>> env = MyBaseEnv()
>>> obs, rewards, dones, infos, off_policy_actions = env.poll()
>>> print(obs)
{
    "env_0": {
        "car_0": [2.4, 1.6],
        "car_1": [3.4, -3.2],
    },
    "env_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.**VectorEnv**
An environment that supports batch evaluation.

Subclasses must define the following attributes:

action_space
Action space of individual envs.

Type gym.Space

observation_space
Observation space of individual envs.

Type gym.Space

num_envs
Number of envs in this vector env.

Type int

vector_reset()
Resets all environments.

Returns Vector of observations from each environment.

Return type obs (list)

reset_at(*index*)
Resets a single environment.

Returns Observations from the resetted environment.

Return type obs (obj)

vector_step(*actions*)
Vectorized step.

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

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

Return type obs (list)

get_unwrapped()
Returns the underlying env instances.

ray.rllib.env.**ServingEnv**
alias of `ray.rllib.env.external_env.ExternalEnv`

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

4.35.3 ray.rllib.evaluation

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.

class ray.rllib.evaluation.**RolloutWorker**(*env_creator*, *policy*, *policy_mapping_fn=None*, *policies_to_train=None*, *tf_session_creator=None*, *batch_steps=100*, *batch_mode='truncate_episodes'*, *episode_horizon=None*, *preprocessor_pref='deepmind'*, *sample_async=False*, *compress_observations=False*, *num_envs=1*, *observation_filter='NoFilter'*, *clip_rewards=None*, *clip_actions=True*, *env_config=None*, *model_config=None*, *policy_config=None*, *worker_index=0*, *monitor_path=None*, *log_dir=None*, *log_level=None*, *callbacks=None*, *input_creator=<function RolloutWorker.<lambda>>*, *input_evaluation=frozenset()*, *output_creator=<function RolloutWorker.<lambda>>*, *remote_worker_envs=False*, *remote_env_batch_wait_ms=0*, *soft_horizon=False*, *no_done_at_end=False*, *seed=None*, *_fake_sampler=False*)

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=PGTFPolicy)
>>> 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":
...         (PGTFPolicy, Box(...), Discrete(...), {"gamma": 0.99}),
...       "car_policy2":
...         (PGTFPolicy, Box(...), Discrete(...), {"gamma": 0.95}),
...       # Use a single shared policy for all traffic lights
...       "traffic_light_policy":
...         (PGTFPolicy, 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*)
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)
```

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

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

sync_filters(*new_filters*)
Changes self's filter to given and rebases any accumulated delta.

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

get_filters(*flush_after=False*)
Returns a snapshot of filters.

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

Returns Dict for serializable filters

Return type return_filters (dict)

`ray.rllib.evaluation.`**PolicyGraph**
alias of `ray.rllib.utils.renamed_class.<locals>.DeprecationWrapper`

`ray.rllib.evaluation.`**TFPolicyGraph**
alias of `ray.rllib.utils.renamed_class.<locals>.DeprecationWrapper`

`ray.rllib.evaluation.`**TorchPolicyGraph**
alias of `ray.rllib.utils.renamed_class.<locals>.DeprecationWrapper`

class `ray.rllib.evaluation.`**SampleBatch**(**args*, ***kw*)

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*, *postp_callback*)
Util to build SampleBatches for each policy in a multi-agent env.

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

total()
Returns summed number of steps across all agent buffers.

has_pending_data()
Returns whether there is pending unprocessed data.

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

Parameters

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

postprocess_batch_so_far(*episode*)
Apply policy postprocessors to any unprocessed rows.

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

Parameters episode – current MultiAgentEpisode object or None

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

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

Parameters episode – current MultiAgentEpisode object or None

class ray.rllib.evaluation.**SyncSampler**(*env*, *policies*, *policy_mapping_fn*, *preprocessors*, *obs_filters*, *clip_rewards*, *unroll_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**(*env*, *policies*, *policy_mapping_fn*, *preprocessors*, *obs_filters*, *clip_rewards*, *unroll_length*, *callbacks*, *horizon=None*, *pack=False*, *tf_sess=None*, *clip_actions=True*, *blackhole_outputs=False*, *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*)

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

Parameters

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

Returns

Object with experience from rollout and processed rewards.

Return type *SampleBatch* (*SampleBatch*)

`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.`**`MultiAgentEpisode`**(*policies*, *policy_mapping_fn*, *batch_builder_factory*, *extra_batch_callback*)

Tracks the current state of a (possibly multi-agent) episode.

`new_batch_builder`

Create a new MultiAgentSampleBatchBuilder.

Type func

`add_extra_batch`

Return a built MultiAgentBatch to the sampler.

Type func

`batch_builder`

Batch builder for the current episode.

Type obj

`total_reward`

Summed reward across all agents in this episode.

Type float

`length`

Length of this episode.

Type int

`episode_id`

Unique id identifying this trajectory.

Type int

`agent_rewards`

Summed rewards broken down by agent.

Type dict

`custom_metrics`

Dict where the you can add custom metrics.

Type dict

user_data
Dict that you can use for temporary storage.

Type dict

Use case 1: Model-based rollouts in multi-agent: A custom compute_actions() function in a policy 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.

ray.rllib.evaluation.**PolicyEvaluator**
alias of `ray.rllib.utils.renamed_class.<locals>.DeprecationWrapper`

4.35.4 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.

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*, *torch=False*)
Returns action distribution class and size for the given action space.

Parameters

- **action_space** (*Space*) – Action space of the target gym env.
- **config** (*dict*) – Optional model config.
- **dist_type** (*str*) – Optional identifier of the action distribution.
- **torch** (*bool*) – Optional whether to return PyTorch distribution.

Returns Python class of the distribution. dist_dim (int): The size of the input vector to the distribution.

Return type dist_class (*ActionDistribution*)

static get_action_placeholder(*action_space*)
Returns an action placeholder that is consistent with the action space

Parameters **action_space** (*Space*) – Action space of the target gym env.

Returns A placeholder for the actions

Return type action_placeholder (Tensor)

static get_model_v2(*obs_space*, *action_space*, *num_outputs*, *model_config*, *framework*, *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*) – Either “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”: preprocesor_name} in the model config.

Parameters

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

static register_custom_model(*model_name*, *model_class*)
Register a custom model class by name.

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

Parameters

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

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.

4.35.5 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*, *sample_batch_size=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*, *sample_batch_size=50*, *num_envs_per_worker=1*, *num_gpus=0*, *lr=0.0005*, *replay_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*, *minibatch_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*)

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*, *schedule_max_timesteps=100000*, *beta_annealing_fraction=0.2*, *final_prioritized_replay_beta=0.4*, *train_batch_size=32*, *sample_batch_size=4*, *before_learn_on_batch=None*, *synchronize_sampling=False*)

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.**LocalMultiGPUOptimizer**(*workers*, *sgd_batch_size=128*, *num_sgd_iter=10*, *sample_batch_size=200*, *num_envs_per_worker=1*, *train_batch_size=1024*, *num_gpus=0*, *standardize_fields=[]*, *shuffle_sequences=True*)

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 require the underlying Policy to be a TFPolicy instance that support *.copy()*.

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.**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.

4.35.6 ray.rllib.utils

ray.rllib.utils.**renamed_class**(*cls*, *old_name*)
Helper class for renaming classes with a warning.

class ray.rllib.utils.**Filter**
Processes input, possibly statefully.

apply_changes(*other*, **args*, ***kwargs*)
Updates self with "new state" from other filter.

copy()
Creates a new object with same state as self.

Returns A copy of self.

sync(*other*)
Copies all state from other filter to self.

clear_buffer()
Creates copy of current state and clears accumulated state

class ray.rllib.utils.**FilterManager**
Manages filters and coordination across remote evaluators that expose *get_filters* and *sync_filters*.

static synchronize(*local_filters*, *remotes*, *update_remote=True*)
Aggregates all filters from remote evaluators.

Local copy is updated and then broadcasted to all remote evaluators.

Parameters

- **local_filters** (*dict*) – Filters to be synchronized.
- **remotes** (*list*) – Remote evaluators with filters.
- **update_remote** (*bool*) – Whether to push updates to remote filters.

class ray.rllib.utils.**PolicyClient**(*address*)

REST client to interact with a RLlib policy server.

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

Record the start of an episode.

Parameters

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

Returns Unique string id for the episode.

Return type episode_id (str)

get_action(*episode_id*, *observation*)

Record an observation and get the on-policy action.

Parameters

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

Returns Action from the env action space.

Return type action (obj)

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

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

Parameters

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

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

Record returns from the environment.

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

Parameters

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

end_episode(*episode_id*, *observation*)

Record the end of an episode.

Parameters

- **episode_id** (*str*) – Episode id returned from start_episode().

- **observation** (*obj*) – Current environment observation.

class ray.rllib.utils.**PolicyServer**(*external_env*, *address*, *port*)

REST server than can be launched from a ExternalEnv.

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

Examples

```
>>> class CartpoleServing(ExternalEnv):
       def __init__(self):
           ExternalEnv.__init__(
               self, spaces.Discrete(2),
               spaces.Box(
                   low=-10,
                   high=10,
                   shape=(4,),
                   dtype=np.float32))
       def run(self):
           server = PolicyServer(self, "localhost", 8900)
           server.serve_forever()
>>> register_env("srv", lambda _: CartpoleServing())
>>> pg = PGTrainer(env="srv", config={"num_workers": 0})
>>> while True:
        pg.train()
```

```
>>> client = PolicyClient("localhost:8900")
>>> eps_id = client.start_episode()
>>> action = client.get_action(eps_id, obs)
>>> ...
>>> client.log_returns(eps_id, reward)
>>> ...
>>> client.log_returns(eps_id, reward)
```

ray.rllib.utils.**merge_dicts**(*d1*, *d2*)

Returns a new dict that is d1 and d2 deep merged.

ray.rllib.utils.**deep_update**(*original*, *new_dict*, *new_keys_allowed*, *whitelist*)

Updates original dict with values from new_dict recursively. If new key is introduced in new_dict, then if new_keys_allowed is not True, an error will be thrown. Further, for sub-dicts, if the key is in the whitelist, then new subkeys can be introduced.

Parameters

- **original** (*dict*) – Dictionary with default values.
- **new_dict** (*dict*) – Dictionary with values to be updated
- **new_keys_allowed** (*bool*) – Whether new keys are allowed.
- **whitelist** (*list*) – List of keys that correspond to dict values where new subkeys can be introduced. This is only at the top level.

4.36 Distributed Training (Experimental)

Ray includes abstractions for distributed model training that integrate with deep learning frameworks, such as PyTorch.

Ray Train is built on top of the Ray task and actor abstractions to provide seamless integration into existing Ray applications.

4.36.1 PyTorch Interface

To use Ray Train with PyTorch, pass model and data creator functions to the `ray.experimental.sgd.pytorch.PyTorchTrainer` class. To drive the distributed training, `trainer.train()` can be called repeatedly.

```
model_creator = lambda config: YourPyTorchModel()
data_creator = lambda config: YourTrainingSet(), YourValidationSet()

trainer = PyTorchTrainer(
    model_creator,
    data_creator,
    optimizer_creator=utils.sgd_mse_optimizer,
    config={"lr": 1e-4},
    num_replicas=2,
    resources_per_replica=Resources(num_gpus=1),
    batch_size=16,
    backend="auto")

for i in range(NUM_EPOCHS):
    trainer.train()
```

Under the hood, Ray Train will create *replicas* of your model (controlled by `num_replicas`) which are each managed by a worker. Multiple devices (e.g. GPUs) can be managed by each replica (controlled by `resources_per_replica`), which allows training of lage models across multiple GPUs. The `PyTorchTrainer` class coordinates the distributed computation and training to improve the model.

The full documentation for `PyTorchTrainer` is as follows:

class `ray.experimental.sgd.pytorch.`**`PyTorchTrainer`**(*model_creator*, *data_creator*, *optimizer_creator=<function sgd_mse_optimizer>*, *config=None*, *num_replicas=1*, *use_gpu=False*, *batch_size=16*, *backend='auto'*)

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.

`__init__`(*model_creator*, *data_creator*, *optimizer_creator=<function sgd_mse_optimizer>*, *config=None*, *num_replicas=1*, *use_gpu=False*, *batch_size=16*, *backend='auto'*)

Sets up the PyTorch trainer.

Parameters

- **`model_creator`** (*`dict -> torch.nn.Module`*) – creates the model using the config.
- **`data_creator`** (*`dict -> Dataset, Dataset`*) – creates the training and validation data sets using the config.
- **`optimizer_creator`** (*`torch.nn.Module, dict -> loss, optimizer`*) – creates the loss and optimizer using the model and the config.
- **`config`** (*`dict`*) – configuration passed to 'model_creator', 'data_creator', and 'optimizer_creator'.

- **num_replicas** (*int*) – the number of workers used in distributed training.
- **use_gpu** (*bool*) – Sets resource allocation for workers to 1 GPU if true.
- **batch_size** (*int*) – batch size for an update.
- **backend** (*string*) – backend used by distributed PyTorch.

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.

4.37 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.

4.38 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.

4.38.1 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
# .rayproject subdirectory of the current directory.
$ ray project create <project-name>

# Create a new session from the given project.
# Launch a cluster and run the appropriate command.
$ ray session start

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

(continues on next page)

(continued from previous page)

```
# Stop the given session and all of its worker nodes. The nodes/clusters
# are not actually terminated.
$ ray session stop
```

4.38.2 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)

4.38.3 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:

```yaml
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: .rayproject/cluster.yaml

# Commands/information to build the environment, once the cluster is
# 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: test
    command: python test.py
```

Project files have to adhere to the following schema:

type	*object*	
properties		
• **name**	The name of the project	
	type	*string*
• description	A short description of the project	
	type	*string*
• repo	The URL of the repo this project is part of	
	type	*string*
• **cluster**	Path to a .yaml cluster configuration file (relative to the project root)	
	type	*string*

Continued on next page

Table 1 – continued from previous page

• environment	The environment that needs to be set up to run the project			
	type	*object*		
	properties			
	• dockerimage	URL to a docker image that can be pulled to run the project in		
		type	*string*	
	• dockerfile	Path to a Dockerfile to set up an image the project can run in (relative to the project root)		
		type	*string*	
	• requirements	Path to a Python requirements.txt file to set up project dependencies (relative to the project root)		
		type	*string*	
	• shell	A sequence of shell commands to run to set up the project environment		
		type	*array*	
		items		
		•	type	*string*
• commands	type	*array*		
	items			
	•	Possible commands to run to start a session		
		type	*object*	
		properties		
		• name	Name of the command	
			type	*string*
		• command	Shell command to run on the cluster	
			type	*string*

4.38.4 Cluster file format (cluster.yaml)

This is the same as for the autoscaler, see Cluster Launch page.

4.39 Signal API (Experimental)

This experimental API allows tasks and actors to generate signals which can be received by other tasks and actors. In addition, task failures and actor method failures generate error signals. The error signals enable applications to detect failures and potentially recover from failures.

`ray.experimental.signal.send(signal)`

Send signal.

The signal has a unique identifier that is computed from (1) the id of the actor or task sending this signal (i.e., the actor or task calling this function), and (2) an index that is incremented every time this source sends a signal. This index starts from 1.

Parameters **`signal`** – Signal to be sent.

Here is a simple example of a remote function that sends a user-defined signal.

```
import ray.experimental.signal as signal

# Define an application level signal.
```

(continues on next page)

(continued from previous page)

```
class UserSignal(signal.Signal):
    def __init__(self, value):
          self.value = value

    def get_value(self):
          return self.value

# Define a remote function that sends a user-defined signal.
@ray.remote
def send_signal(value):
    signal.send(UserSignal(value))
```

ray.experimental.signal.**receive**(*sources*, *timeout=None*)

Get all outstanding signals from sources.

A source can be either (1) an object ID returned by the task (we want to receive signals from), or (2) an actor handle.

When invoked by the same entity E (where E can be an actor, task or driver), for each source S in sources, this function returns all signals generated by S since the last receive() was invoked by E on S. If this is the first call on S, this function returns all past signals generated by S so far. Note that different actors, tasks or drivers that call receive() on the same source S will get independent copies of the signals generated by S.

Parameters

- **sources** – List of sources from which the caller waits for signals. A source is either an object ID returned by a task (in this case the object ID is used to identify that task), or an actor handle. If the user passes the IDs of multiple objects returned by the same task, this function returns a copy of the signals generated by that task for each object ID.
- **timeout** – Maximum time (in seconds) this function waits to get a signal from a source in sources. If None, the timeout is infinite.

Returns

A list of pairs (S, sig), where S is a source in the sources argument, and sig is a signal generated by S since the last time receive() was called on S. Thus, for each S in sources, the return list can contain zero or multiple entries.

Here is a simple example of how to receive signals from an actor or task identified by `a`. Note that an actor is identified by its handle, and a task by one of its object ID return values.

```
import ray.experimental.signal as signal

# This returns a possibly empty list of all signals that have been sent by 'a'
# since the last invocation of signal.receive from within this process. If 'a'
# did not send any signals, then this will wait for up to 10 seconds to receive
# a signal from 'a'.
signal_list = signal.receive([a], timeout=10)
```

ray.experimental.signal.**reset**()

Reset the worker state associated with any signals that this worker has received so far.

If the worker calls receive() on a source next, it will get all the signals generated by that source starting with index = 1.

4.39.1 Example: sending a user signal

The code below show a simple example in which a task, called `send_signal()` sends a user signal and the driver gets it by invoking `signal.receive()`.

```
import ray.experimental.signal as signal

# Define a user signal.
class UserSignal(signal.Signal):
    def __init__(self, value):
          self.value = value

    def get_value(self):
          return self.value

@ray.remote
def send_signal(value):
    signal.send(UserSignal(value))
    return

signal_value = 'simple signal'
object_id = send_signal.remote(signal_value)
# Wait up to 10sec to receive a signal from the task. Note the task is
# identified by the object_id it returns.
result_list = signal.receive([object_id], timeout=10)
# Print signal values. This should print "simple_signal".
# Note that result_list[0] is the signal we expect from the task.
# The signal is a tuple where the first element is the first object ID
# returned by the task and the second element is the signal object.
print(result_list[0][1].get_value())
```

4.39.2 Example: Getting an error signals

This is a simple example in which a driver gets an error signal caused by the failure of `task()`.

```
@ray.remote
def task():
    raise Exception('exception message')

object_id = task.remote()
try:
    ray.get(object_id)
except Exception as e:
    pass
finally:
    result_list = signal.receive([object_id], timeout=10)
    # Expected signal is 'ErrorSignal'.
    assert type(result_list[0][1]) == signal.ErrorSignal
    # Print the error.
    print(result_list[0][1].get_error())
```

4.39.3 Example: Sending signals between multiple actors

This is a more involved example in which two actors `a1` and `a2` each generate five signals, and another actor `b` waits to receive all signals generated by `a1` and `a2`, respectively. Note that `b` recursively calls its own method get_signals() until it gets all signals it expects.

```
@ray.remote
class ActorSendSignals(object):
    def send_signals(self, value, count):
        for i in range(count):
            signal.send(UserSignal(value + str(i)))

@ray.remote
class ActorGetAllSignals(object):
    def __init__(self, num_expected_signals, *source_ids):
        self.received_signals = []
        self.num_expected_signals = num_expected_signals
        self.source_ids = source_ids

    def register_handle(self, handle):
        self.this_actor = handle

    def get_signals(self):
        new_signals = signal.receive(self.source_ids, timeout=10)
        self.received_signals.extend(new_signals)
        if len(self.received_signals) < self.num_expected_signals:
            self.this_actor.get_signals.remote()

    def get_count(self):
        return len(self.received_signals)

# Create two actors to send signals.
a1 = ActorSendSignals.remote()
a2 = ActorSendSignals.remote()
signal_value = 'simple signal'
count = 5
# Each actor sends five signals.
a1.send_signals.remote(signal_value, count)
a2.send_signals.remote(signal_value, count)

# Create an actor that waits for all five signals sent by each actor.
b = ActorGetAllSignals.remote(2 * count, *[a1, a2])
# Provide actor to its own handle, so it can recursively call itself
# to get all signals from a1, and a2, respectively. This enables the actor
# execute other methods if needed.
ray.get(b.register_handle.remote(b))
b.get_signals.remote()
# Print total number of signals. This should be 2*count = 10.
print(ray.get(b.get_count.remote()))
```

4.39.4 Note

A failed actor (e.g., an actor that crashed) generates an error message only when another actor or task invokes one of its methods.

Please let us know any issues you encounter.

4.40 Async API (Experimental)

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

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

4.40.1 Starting Ray

You must initialize Ray first.

Please refer to Starting Ray for instructions.

4.40.2 Converting Ray objects into asyncio futures

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

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

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

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

`ray.experimental.async_api.`**`as_future`**`(`*object_id*`)`

Turn an object_id into a Future object.

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

Returns A future object that waits the object_id.

Return type PlasmaObjectFuture

4.40.3 Example Usage

Basic Python	Distributed with Ray
`# Execute f serially.`	`# Execute f in parallel.`
`def f():`	`@ray.remote`
`  time.sleep(1)`	`def f():`
`  return 1`	`    time.sleep(1)`
	`    return 1`
	`ray.init()`
`results = [f() for i in range(4)]`	`results = ray.get([f.remote() for i in␣`
	`↪range(4)])`
Async Python	**Async Ray**
`# Execute f asynchronously.`	`# Execute f asynchronously with Ray/`
	`↪asyncio.`
	`from ray.experimental import async_api`
	`@ray.remote`
`async def f():`	`def f():`
`    await asyncio.sleep(1)`	`    time.sleep(1)`
`    return 1`	`    return 1`
	`ray.init()`
`loop = asyncio.get_event_loop()`	`loop = asyncio.get_event_loop()`
`tasks = [f() for i in range(4)]`	`tasks = [async_api.as_future(f.remote())`
	`         for i in range(4)]`
`results = loop.run_until_complete(`	`results = loop.run_until_complete(`
`    asyncio.gather(tasks))`	`    asyncio.gather(tasks))`

4.40.4 Known Issues

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

4.41 Learning to Play Pong

In this example, we'll train a **very simple** neural network to play Pong using the OpenAI Gym. This application is adapted, with minimal modifications, from Andrej Karpathy's code (see the accompanying blog post).

You can view the code for this example.

To run the application, first install some dependencies.

```
pip install gym[atari]
```

Then you can run the example as follows.

```
python ray/doc/examples/rl_pong/driver.py --batch-size=10
```

To run the example on a cluster, simply pass in the flag `--redis-address=<redis-address>`.

At the moment, on a large machine with 64 physical cores, computing an update with a batch of size 1 takes about 1 second, a batch of size 10 takes about 2.5 seconds. A batch of size 60 takes about 3 seconds. On a cluster with 11 nodes, each with 18 physical cores, a batch of size 300 takes about 10 seconds. If the numbers you see differ from these by much, take a look at the **Troubleshooting** section at the bottom of this page and consider submitting an issue.

Note that these times depend on how long the rollouts take, which in turn depends on how well the policy is doing. For example, a really bad policy will lose very quickly. As the policy learns, we should expect these numbers to increase.

4.41.1 The distributed version

At the core of Andrej's code, a neural network is used to define a "policy" for playing Pong (that is, a function that chooses an action given a state). In the loop, the network repeatedly plays games of Pong and records a gradient from each game. Every ten games, the gradients are combined together and used to update the network.

This example is easy to parallelize because the network can play ten games in parallel and no information needs to be shared between the games.

We define an **actor** for the Pong environment, which includes a method for performing a rollout and computing a gradient update. Below is pseudocode for the actor.

```
@ray.remote
class PongEnv(object):
    def __init__(self):
        # Tell numpy to only use one core. If we don't do this, each actor may try
        # to use all of the cores and the resulting contention may result in no
        # speedup over the serial version. Note that if numpy is using OpenBLAS,
        # then you need to set OPENBLAS_NUM_THREADS=1, and you probably need to do
        # it from the command line (so it happens before numpy is imported).
        os.environ["MKL_NUM_THREADS"] = "1"
        self.env = gym.make("Pong-v0")

    def compute_gradient(self, model):
        # Reset the game.
        observation = self.env.reset()
        while not done:
            # Choose an action using policy_forward.
            # Take the action and observe the new state of the world.
        # Compute a gradient using policy_backward. Return the gradient and reward.
        return [gradient, reward_sum]
```

We then create a number of actors, so that we can perform rollouts in parallel.

```
actors = [PongEnv() for _ in range(batch_size)]
```

Calling this remote function inside of a for loop, we launch multiple tasks to perform rollouts and compute gradients in parallel.

```
model_id = ray.put(model)
actions = []
# Launch tasks to compute gradients from multiple rollouts in parallel.
for i in range(batch_size):
    action_id = actors[i].compute_gradient.remote(model_id)
    actions.append(action_id)
```

4.41.2 Troubleshooting

If you are not seeing any speedup from Ray (and assuming you're using a multicore machine), the problem may be that numpy is trying to use multiple threads. When many processes are each trying to use multiple threads, the result is often no speedup. When running this example, try opening up `top` and seeing if some python processes are using more than 100% CPU. If yes, then this is likely the problem.

The example tries to set `MKL_NUM_THREADS=1` in the actor. However, that only works if the numpy on your machine is actually using MKL. If it's using OpenBLAS, then you'll need to set `OPENBLAS_NUM_THREADS=1`. In fact, you may have to do this **before** running the script (it may need to happen before numpy is imported).

```
export OPENBLAS_NUM_THREADS=1
```

4.42 Parameter Server

This document walks through how to implement simple synchronous and asynchronous parameter servers using actors. To run the application, first install some dependencies.

```
pip install tensorflow
```

You can view the code for this example.

The examples can be run as follows.

```
# Run the asynchronous parameter server.
python ray/doc/examples/parameter_server/async_parameter_server.py --num-workers=4

# Run the synchronous parameter server.
python ray/doc/examples/parameter_server/sync_parameter_server.py --num-workers=4
```

Note that this examples uses distributed actor handles, which are still considered experimental.

4.42.1 Asynchronous Parameter Server

The asynchronous parameter server itself is implemented as an actor, which exposes the methods `push` and `pull`.

```
@ray.remote
class ParameterServer(object):
    def __init__(self, keys, values):
        values = [value.copy() for value in values]
        self.weights = dict(zip(keys, values))

    def push(self, keys, values):
        for key, value in zip(keys, values):
            self.weights[key] += value

    def pull(self, keys):
        return [self.weights[key] for key in keys]
```

We then define a worker task, which take a parameter server as an argument and submits tasks to it. The structure of the code looks as follows.

```python
@ray.remote
def worker_task(ps):
    while True:
        # Get the latest weights from the parameter server.
        weights = ray.get(ps.pull.remote(keys))

        # Compute an update.
        ...

        # Push the update to the parameter server.
        ps.push.remote(keys, update)
```

Then we can create a parameter server and initiate training as follows.

```python
ps = ParameterServer.remote(keys, initial_values)
worker_tasks = [worker_task.remote(ps) for _ in range(4)]
```

4.42.2 Synchronous Parameter Server

The parameter server is implemented as an actor, which exposes the methods `apply_gradients` and `get_weights`. A constant linear scaling rule is applied by scaling the learning rate by the number of workers.

```python
@ray.remote
class ParameterServer(object):
    def __init__(self, learning_rate):
        self.net = model.SimpleCNN(learning_rate=learning_rate)

    def apply_gradients(self, *gradients):
        self.net.apply_gradients(np.mean(gradients, axis=0))
        return self.net.variables.get_flat()

    def get_weights(self):
        return self.net.variables.get_flat()
```

Workers are actors which expose the method `compute_gradients`.

```python
@ray.remote
class Worker(object):
    def __init__(self, worker_index, batch_size=50):
        self.worker_index = worker_index
        self.batch_size = batch_size
        self.mnist = input_data.read_data_sets("MNIST_data", one_hot=True,
                                               seed=worker_index)
        self.net = model.SimpleCNN()

    def compute_gradients(self, weights):
        self.net.variables.set_flat(weights)
        xs, ys = self.mnist.train.next_batch(self.batch_size)
        return self.net.compute_gradients(xs, ys)
```

Training alternates between computing the gradients given the current weights from the parameter server and updating the parameter server's weights with the resulting gradients.

```python
while True:
    gradients = [worker.compute_gradients.remote(current_weights)
```

(continues on next page)

(continued from previous page)

```
                 for worker in workers]
    current_weights = ps.apply_gradients.remote(*gradients)
```

Both of these examples implement the parameter server using a single actor, however they can be easily extended to **split the parameters across multiple actors**.

4.43 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;`.

4.44 ResNet

This code adapts the TensorFlow ResNet example to do data parallel training across multiple GPUs using Ray. View the code for this example.

To run the example, you will need to install TensorFlow (at least version `1.0.0`). Then you can run the example as follows.

First download the CIFAR-10 or CIFAR-100 dataset.

```
# Get the CIFAR-10 dataset.
curl -o cifar-10-binary.tar.gz https://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz
tar -xvf cifar-10-binary.tar.gz

# Get the CIFAR-100 dataset.
curl -o cifar-100-binary.tar.gz https://www.cs.toronto.edu/~kriz/cifar-100-binary.tar.
↪gz
tar -xvf cifar-100-binary.tar.gz
```

Then run the training script that matches the dataset you downloaded.

```
# Train Resnet on CIFAR-10.
python ray/doc/examples/resnet/resnet_main.py \
    --eval_dir=/tmp/resnet-model/eval \
```

(continues on next page)

(continued from previous page)

```
    --train_data_path=cifar-10-batches-bin/data_batch* \
    --eval_data_path=cifar-10-batches-bin/test_batch.bin \
    --dataset=cifar10 \
    --num_gpus=1

# Train Resnet on CIFAR-100.
python ray/doc/examples/resnet/resnet_main.py \
    --eval_dir=/tmp/resnet-model/eval \
    --train_data_path=cifar-100-binary/train.bin \
    --eval_data_path=cifar-100-binary/test.bin \
    --dataset=cifar100 \
    --num_gpus=1
```

To run the training script on a cluster with multiple machines, you will need to also pass in the flag `--redis-address=<redis_address>`, where `<redis-address>` is the address of the Redis server on the head node.

The script will print out the IP address that the log files are stored on. In the single-node case, you can ignore this and run tensorboard on the current machine.

```
python -m tensorflow.tensorboard --logdir=/tmp/resnet-model
```

If you are running Ray on multiple nodes, you will need to go to the node at the IP address printed, and run the command.

The core of the script is the actor definition.

```
@ray.remote(num_gpus=1)
class ResNetTrainActor(object):
    def __init__(self, data, dataset, num_gpus):
        # data is the preprocessed images and labels extracted from the dataset.
        # Thus, every actor has its own copy of the data.
        # Set the CUDA_VISIBLE_DEVICES environment variable in order to restrict
        # which GPUs TensorFlow uses. Note that this only works if it is done before
        # the call to tf.Session.
        os.environ['CUDA_VISIBLE_DEVICES'] = ','.join([str(i) for i in ray.get_gpu_
↪ids()])
        with tf.Graph().as_default():
            with tf.device('/gpu:0'):
                # We omit the code here that actually constructs the residual network
                # and initializes it. Uses the definition in the Tensorflow Resnet␣
↪Example.

    def compute_steps(self, weights):
        # This method sets the weights in the network, runs some training steps,
        # and returns the new weights. self.model.variables is a TensorFlowVariables
        # class that we pass the train operation into.
        self.model.variables.set_weights(weights)
        for i in range(self.steps):
            self.model.variables.sess.run(self.model.train_op)
        return self.model.variables.get_weights()
```

The main script first creates one actor for each GPU, or a single actor if `num_gpus` is zero.

```
train_actors = [ResNetTrainActor.remote(train_data, dataset, num_gpus) for _ in␣
↪range(num_gpus)]
```

Then the main loop passes the same weights to every model, performs updates on each model, averages the updates, and puts the new weights in the object store.

```
while True:
    all_weights = ray.get([actor.compute_steps.remote(weight_id) for actor in train_
↪actors])
    mean_weights = {k: sum([weights[k] for weights in all_weights]) / num_gpus for k␣
↪in all_weights[0]}
    weight_id = ray.put(mean_weights)
```

4.45 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}'
```

4.45.1 Reinforcement Learning

Reinforcement Learning is an area of machine learning concerned with **learning how an agent should act in an environment** so as to maximize some form of cumulative reward. Typically, an agent will observe the current state of the environment and take an action based on its observation. The action will change the state of the environment and will provide some numerical reward (or penalty) to the agent. The agent will then take in another observation and the process will repeat. **The mapping from state to action is a policy**, and in reinforcement learning, this policy is often represented with a deep neural network.

The **environment** is often a simulator (for example, a physics engine), and reinforcement learning algorithms often involve trying out many different sequences of actions within these simulators. These **rollouts** can often be done in parallel.

Policies are often initialized randomly and incrementally improved via simulation within the environment. To improve a policy, gradient-based updates may be computed based on the sequences of states and actions that have been observed. The gradient calculation is often delayed until a termination condition is reached (that is, the simulation has finished) so that delayed rewards have been properly accounted for. However, in the Actor Critic model, we can begin the gradient calculation at any point in the simulation rollout by predicting future rewards with a Value Function approximator.

In our A3C implementation, each worker, implemented as a Ray actor, continuously simulates the environment. The driver will create a task that runs some steps of the simulator using the latest model, computes a gradient update, and returns the update to the driver. Whenever a task finishes, the driver will use the gradient update to update the model and will launch a new task with the latest model.

There are two main parts to the implementation - the driver and the worker.

4.45.2 Worker Code Walkthrough

We use a Ray Actor to simulate the environment.

```python
import numpy as np
import ray

@ray.remote
class Runner(object):
    """Actor object to start running simulation on workers.
        Gradient computation is also executed on this object."""
    def __init__(self, env_name, actor_id):
        # starts simulation environment, policy, and thread.
        # Thread will continuously interact with the simulation environment
        self.env = env = create_env(env_name)
        self.id = actor_id
        self.policy = LSTMPolicy()
        self.runner = RunnerThread(env, self.policy, 20)
        self.start()

    def start(self):
        # starts the simulation thread
        self.runner.start_runner()

    def pull_batch_from_queue(self):
        # Implementation details removed - gets partial rollout from queue
        return rollout

    def compute_gradient(self, params):
        self.policy.set_weights(params)
        rollout = self.pull_batch_from_queue()
        batch = process_rollout(rollout, gamma=0.99, lambda_=1.0)
        gradient = self.policy.compute_gradients(batch)
        info = {"id": self.id,
                "size": len(batch.a)}
        return gradient, info
```

4.45.3 Driver Code Walkthrough

The driver manages the coordination among workers and handles updating the global model parameters. The main training script looks like the following.

```python
import numpy as np
import ray

def train(num_workers, env_name="PongDeterministic-v4"):
    # Setup a copy of the environment
    # 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)
```

(continues on next page)

(continued from previous page)

```
    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(parameters)])
    return policy
```

4.45.4 Benchmarks and Visualization

For the `PongDeterministic-v4` and an Amazon EC2 m4.16xlarge instance, we are able to train the agent with 16 workers in around 15 minutes. With 8 workers, we can train the agent in around 25 minutes.

You can visualize performance by running `tensorboard --logdir [directory]` in a separate screen, where `[directory]` is defaulted to `~/ray_results/`. If you are running multiple experiments, be sure to vary the directory to which Tensorflow saves its progress (found in `a3c.py`).

4.46 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.

4.46.1 The serial version

First we load the data in batches. Here, each element in `batches` is a tuple whose first component is a batch of `100` images and whose second component is a batch of the `100` corresponding labels. For simplicity, we use TensorFlow's built in methods for loading the data.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
batch_size = 100
num_batches = mnist.train.num_examples // batch_size
batches = [mnist.train.next_batch(batch_size) for _ in range(num_batches)]
```

Now, suppose we have defined a function which takes a set of model parameters `theta` and a batch of data (both images and labels) and computes the loss for that choice of model parameters on that batch of data. Similarly, suppose we've also defined a function that takes the same arguments and computes the gradient of the loss for that choice of model parameters.

```
def loss(theta, xs, ys):
    # compute the loss on a batch of data
    return loss

def grad(theta, xs, ys):
    # compute the gradient on a batch of data
    return grad

def full_loss(theta):
    # compute the loss on the full data set
    return sum([loss(theta, xs, ys) for (xs, ys) in batches])

def full_grad(theta):
    # compute the gradient on the full data set
    return sum([grad(theta, xs, ys) for (xs, ys) in batches])
```

Since we are working with a small dataset, we don't actually need to separate these methods into the part that operates on a batch and the part that operates on the full dataset, but doing so will make the distributed version clearer.

Now, if we wish to optimize the loss function using L-BFGS, we simply plug these functions, along with an initial choice of model parameters, into `scipy.optimize.fmin_l_bfgs_b`.

```
theta_init = 1e-2 * np.random.normal(size=dim)
result = scipy.optimize.fmin_l_bfgs_b(full_loss, theta_init, fprime=full_grad)
```

4.46.2 The distributed version

In this example, the computation of the gradient itself can be done in parallel on a number of workers or machines.

First, let's turn the data into a collection of remote objects.

```
batch_ids = [(ray.put(xs), ray.put(ys)) for (xs, ys) in batches]
```

We can load the data on the driver and distribute it this way because MNIST easily fits on a single machine. However, for larger data sets, we will need to use remote functions to distribute the loading of the data.

Now, lets turn `loss` and `grad` into methods of an actor that will contain our network.

```python
class Network(object):
    def __init__():
        # Initialize network.

    def loss(theta, xs, ys):
        # compute the loss
        return loss

    def grad(theta, xs, ys):
        # compute the gradient
        return grad
```

Now, it is easy to speed up the computation of the full loss and the full gradient.

```python
def full_loss(theta):
    theta_id = ray.put(theta)
    loss_ids = [actor.loss(theta_id) for actor in actors]
    return sum(ray.get(loss_ids))

def full_grad(theta):
    theta_id = ray.put(theta)
    grad_ids = [actor.grad(theta_id) for actor in actors]
    return sum(ray.get(grad_ids)).astype("float64") # This conversion is necessary␣
↪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

```python
[actor.loss(theta_id) for actor in actors]
```

instead of

```python
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.

```python
theta_init = 1e-2 * np.random.normal(size=dim)
result = scipy.optimize.fmin_l_bfgs_b(full_loss, theta_init, fprime=full_grad)
```

4.47 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
```

(continues on next page)

(continued from previous page)

```
   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.

4.48 Using Ray with TensorFlow

This document describes best practices for using Ray with TensorFlow.

To see more involved examples using TensorFlow, take a look at A3C, ResNet, and LBFGS.

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

A straightforward attempt to pickle a TensorFlow graph gives mixed results. Some examples fail, and some succeed (but produce very large strings). The results are similar with other pickling libraries as well.

Furthermore, creating a TensorFlow graph can take tens of seconds, and so serializing a graph and recreating it in another process will be inefficient. The better solution is to 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).

```
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.

```
# 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.
```

4.48.1 Complete Example for Weight Averaging

Putting this all together, we would first embed the graph in an actor. Within the actor, we would use the `get_weights` and `set_weights` methods of the `TensorFlowVariables` class. We would then use those methods to ship the weights (as a dictionary of variable names mapping to numpy arrays) between the processes without shipping the actual TensorFlow graphs, which are much more complex Python objects.

```
import tensorflow as tf
import numpy as np
import ray
import ray.experimental.tf_utils

ray.init()

BATCH_SIZE = 100
NUM_BATCHES = 1
NUM_ITERS = 201

class Network(object):
    def __init__(self, x, y):
        # Seed TensorFlow to make the script deterministic.
        tf.set_random_seed(0)
        # Define the inputs.
        self.x_data = tf.constant(x, dtype=tf.float32)
        self.y_data = tf.constant(y, dtype=tf.float32)
        # Define the weights and computation.
        w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
        b = tf.Variable(tf.zeros([1]))
        y = w * self.x_data + b
        # Define the loss.
        self.loss = tf.reduce_mean(tf.square(y - self.y_data))
        optimizer = tf.train.GradientDescentOptimizer(0.5)
        self.grads = optimizer.compute_gradients(self.loss)
        self.train = optimizer.apply_gradients(self.grads)
        # Define the weight initializer and session.
        init = tf.global_variables_initializer()
        self.sess = tf.Session()
        # Additional code for setting and getting the weights
        self.variables = ray.experimental.tf_utils.TensorFlowVariables(self.loss,␣
↪self.sess)
        # Return all of the data needed to use the network.
        self.sess.run(init)

    # Define a remote function that trains the network for one step and returns the
    # new weights.
    def step(self, weights):
        # Set the weights in the network.
        self.variables.set_weights(weights)
        # Do one step of training.
        self.sess.run(self.train)
        # Return the new weights.
        return self.variables.get_weights()
```

(continues on next page)

(continued from previous page)

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

# Define a remote function for generating fake data.
@ray.remote(num_return_vals=2)
def generate_fake_x_y_data(num_data, seed=0):
    # Seed numpy to make the script deterministic.
    np.random.seed(seed)
    x = np.random.rand(num_data)
    y = x * 0.1 + 0.3
    return x, y

# Generate some training data.
batch_ids = [generate_fake_x_y_data.remote(BATCH_SIZE, seed=i) for i in range(NUM_
↪BATCHES)]
x_ids = [x_id for x_id, y_id in batch_ids]
y_ids = [y_id for x_id, y_id in batch_ids]
# Generate some test data.
x_test, y_test = ray.get(generate_fake_x_y_data.remote(BATCH_SIZE, seed=NUM_BATCHES))

# Create actors to store the networks.
remote_network = ray.remote(Network)
actor_list = [remote_network.remote(x_ids[i], y_ids[i]) for i in range(NUM_BATCHES)]

# Get initial weights of some actor.
weights = ray.get(actor_list[0].get_weights.remote())

# Do some steps of training.
for iteration in range(NUM_ITERS):
    # Put the weights in the object store. This is optional. We could instead pass
    # the variable weights directly into step.remote, in which case it would be
    # placed in the object store under the hood. However, in that case multiple
    # copies of the weights would be put in the object store, so this approach is
    # more efficient.
    weights_id = ray.put(weights)
    # Call the remote function multiple times in parallel.
    new_weights_ids = [actor.step.remote(weights_id) for actor in actor_list]
    # Get all of the weights.
    new_weights_list = ray.get(new_weights_ids)
    # Add up all the different weights. Each element of new_weights_list is a dict
    # of weights, and we want to add up these dicts component wise using the keys
    # of the first dict.
    weights = {variable: sum(weight_dict[variable] for weight_dict in new_weights_
↪list) / NUM_BATCHES for variable in new_weights_list[0]}
    # Print the current weights. They should converge to roughly to the values 0.1
    # and 0.3 used in generate_fake_x_y_data.
    if iteration % 20 == 0:
        print("Iteration {}: weights are {}".format(iteration, weights))
```

4.48.2 How to Train in Parallel using Ray and Gradients

In some cases, you may want to do data-parallel training on your network. We use the network above to illustrate how to do this in Ray. The only differences are in the remote function `step` and the driver code.

In the function `step`, we run the grad operation rather than the train operation to get the gradients. Since Tensorflow pairs the gradients with the variables in a tuple, we extract the gradients to avoid needless computation.

Extracting numerical gradients

Code like the following can be used in a remote function to compute numerical gradients.

```
x_values = [1] * 100
y_values = [2] * 100
numerical_grads = sess.run([grad[0] for grad in grads], feed_dict={x_data: x_values,␣
↪y_data: y_values})
```

Using the returned gradients to train the network

By pairing the symbolic gradients with the numerical gradients in a feed_dict, we can update the network.

```
# We can feed the gradient values in using the associated symbolic gradient
# operation defined in tensorflow.
feed_dict = {grad[0]: numerical_grad for (grad, numerical_grad) in zip(grads,␣
↪numerical_grads)}
sess.run(train, feed_dict=feed_dict)
```

You can then run `variables.get_weights()` to see the updated weights of the network.

For reference, the full code is below:

```
import tensorflow as tf
import numpy as np
import ray

ray.init()

BATCH_SIZE = 100
NUM_BATCHES = 1
NUM_ITERS = 201

class Network(object):
    def __init__(self, x, y):
        # Seed TensorFlow to make the script deterministic.
        tf.set_random_seed(0)
        # Define the inputs.
        x_data = tf.constant(x, dtype=tf.float32)
        y_data = tf.constant(y, dtype=tf.float32)
        # Define the weights and computation.
        w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
        b = tf.Variable(tf.zeros([1]))
        y = w * x_data + b
        # Define the loss.
        self.loss = tf.reduce_mean(tf.square(y - y_data))
        optimizer = tf.train.GradientDescentOptimizer(0.5)
        self.grads = optimizer.compute_gradients(self.loss)
        self.train = optimizer.apply_gradients(self.grads)
        # Define the weight initializer and session.
        init = tf.global_variables_initializer()
        self.sess = tf.Session()
        # Additional code for setting and getting the weights
        self.variables = ray.experimental.tf_utils.TensorFlowVariables(self.loss,␣
↪self.sess)
```

(continues on next page)

(continued from previous page)

```
        # Return all of the data needed to use the network.
        self.sess.run(init)

    # Define a remote function that trains the network for one step and returns the
    # new weights.
    def step(self, weights):
        # Set the weights in the network.
        self.variables.set_weights(weights)
        # Do one step of training. We only need the actual gradients so we filter␣
↪over the list.
        actual_grads = self.sess.run([grad[0] for grad in self.grads])
        return actual_grads

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

# Define a remote function for generating fake data.
@ray.remote(num_return_vals=2)
def generate_fake_x_y_data(num_data, seed=0):
    # Seed numpy to make the script deterministic.
    np.random.seed(seed)
    x = np.random.rand(num_data)
    y = x * 0.1 + 0.3
    return x, y

# Generate some training data.
batch_ids = [generate_fake_x_y_data.remote(BATCH_SIZE, seed=i) for i in range(NUM_
↪BATCHES)]
x_ids = [x_id for x_id, y_id in batch_ids]
y_ids = [y_id for x_id, y_id in batch_ids]
# Generate some test data.
x_test, y_test = ray.get(generate_fake_x_y_data.remote(BATCH_SIZE, seed=NUM_BATCHES))

# Create actors to store the networks.
remote_network = ray.remote(Network)
actor_list = [remote_network.remote(x_ids[i], y_ids[i]) for i in range(NUM_BATCHES)]
local_network = Network(x_test, y_test)

# Get initial weights of local network.
weights = local_network.get_weights()

# Do some steps of training.
for iteration in range(NUM_ITERS):
    # Put the weights in the object store. This is optional. We could instead pass
    # the variable weights directly into step.remote, in which case it would be
    # placed in the object store under the hood. However, in that case multiple
    # copies of the weights would be put in the object store, so this approach is
    # more efficient.
    weights_id = ray.put(weights)
    # Call the remote function multiple times in parallel.
    gradients_ids = [actor.step.remote(weights_id) for actor in actor_list]
    # Get all of the weights.
    gradients_list = ray.get(gradients_ids)

    # Take the mean of the different gradients. Each element of gradients_list is a␣
↪list
    # of gradients, and we want to take the mean of each one.
```

(continues on next page)

(continued from previous page)

```
    mean_grads = [sum([gradients[i] for gradients in gradients_list]) / len(gradients_
↪list) for i in range(len(gradients_list[0]))]

    feed_dict = {grad[0]: mean_grad for (grad, mean_grad) in zip(local_network.grads,␣
↪mean_grads)}
    local_network.sess.run(local_network.train, feed_dict=feed_dict)
    weights = local_network.get_weights()

    # Print the current weights. They should converge to roughly to the values 0.1
    # and 0.3 used in generate_fake_x_y_data.
    if iteration % 20 == 0:
        print("Iteration {}: weights are {}".format(iteration, weights))
```

class ray.experimental.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.

4.48.3 Troubleshooting

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

```python
import ray
import tensorflow as tf

class Network(object):
    def __init__(self):
        a = tf.Variable(1)
        b = tf.Variable(1)
        c = tf.add(a, b)
        sess = tf.Session()
        init = tf.global_variables_initializer()
        sess.run(init)
        self.variables = ray.experimental.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:

```python
a = Network()
b = Network()
b.set_weights(a.get_weights())
```

the code would fail. If we instead defined each network in its own TensorFlow graph, then it would work:

```python
with tf.Graph().as_default():
    a = Network()
with tf.Graph().as_default():
    b = Network()
b.set_weights(a.get_weights())
```

This issue does not occur between actors that contain a network, as each actor is in its own process, and thus is in its own graph. This also does not occur when using `set_flat`.

Another issue to keep in mind is that `TensorFlowVariables` needs to add new operations to the graph. If you close the graph and make it immutable, e.g. creating a `MonitoredTrainingSession` the initialization will fail. To resolve this, simply create the instance before you close the graph.

4.49 Installing Ray from Source

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

4.49.1 Dependencies

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

For Ubuntu, run the following commands:

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

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

pip install cython==0.29.0
```

For MacOS, run the following commands:

```
brew update
brew install wget

pip install cython==0.29.0
```

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

```
conda install libgcc
```

4.49.2 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

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

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

```
pip install git+https://github.com/ray-project/ray.git#subdirectory=python
```

4.49.3 Cleaning the source tree

The source tree can be cleaned by running

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

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

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

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

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

under `ray/`.

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

4.49.4 Launch Ray in Docker

Start out by launching the deployment container.

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

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

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

You should now see a prompt that looks something like:

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

4.49.5 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
```

Troubleshooting installing Arrow

Some candidate possibilities.

4.49.6 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.

4.49.7 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.

4.50 Development Tips

4.50.1 Compilation

To speed up compilation, be sure to install Ray with

```
cd ray/python
pip install -e . --verbose
```

The `-e` means "editable", so changes you make to files in the Ray directory will take effect without reinstalling the package. In contrast, if you do `python setup.py install`, files will be copied from the Ray directory to a directory of Python packages (often something like `/home/ubuntu/anaconda3/lib/python3.6/site-packages/ray`). This means that changes you make to files in the Ray directory will not have any effect.

If you run into **Permission Denied** errors when running `pip install`, you can try adding `--user`. You may also need to run something like `sudo chown -R $USER /home/ubuntu/anaconda3` (substituting in the appropriate path).

If you make changes to the C++ files, you will need to recompile them. However, you do not need to rerun `pip install -e .`. Instead, you can recompile much more quickly by doing

```
cd ray
bazel build //:ray_pkg
```

This command is not enough to recompile all C++ unit tests. To do so, see *Testing locally*.

4.50.2 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 perftools 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': '2a9d2b34ad24a37ed54e4fcd32bf19f915742f5b',
#   'IsInsertion': True,
#   'NodeManagerAddress': '1.2.3.4',
#   'NodeManagerPort': 43280,
#   'ObjectManagerPort': 38062,
#   'ObjectStoreSocketName': '/tmp/ray/session_2019-01-21_16-28-05_4216/sockets/
↪plasma_store',
#   'RayletSocketName': '/tmp/ray/session_2019-01-21_16-28-05_4216/sockets/raylet',
#   'Resources': {'CPU': 8.0, 'GPU': 1.0}}]
```

To inspect the primary Redis shard manually, you can also query with commands like the following.

```
r_primary = ray.worker.global_worker.redis_client
r_primary.keys("*")
```

To inspect other Redis shards, you will need to create a new Redis client. For example (assuming the relevant IP address is `127.0.0.1` and the relevant port is `1234`), you can do this as follows.

```
import redis
r = redis.StrictRedis(host='127.0.0.1', port=1234)
```

You can find a list of the relevant IP addresses and ports by running

```
r_primary.lrange('RedisShards', 0, -1)
```

Backend logging

The `raylet` process logs detailed information about events like task execution and object transfers between nodes. To set the logging level at runtime, you can set the `RAY_BACKEND_LOG_LEVEL` environment variable before starting Ray. For example, you can do:

```
export RAY_BACKEND_LOG_LEVEL=debug
ray start
```

This will print any `RAY_LOG(DEBUG)` lines in the source code to the `raylet.err` file, which you can find in the Temporary Files.

4.50.3 Testing locally

Suppose that one of the tests (e.g., `test_basic.py`) is failing. You can run that test locally by running `python -m pytest -v python/ray/tests/test_basic.py`. However, doing so will run all of the tests which can take a while. To run a specific test that is failing, you can do

```
cd ray
python -m pytest -v 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 script.

To compile and run all C++ tests, you can run:

```
cd ray
bazel test $(bazel query 'kind(cc_test, ...)')
```

4.50.4 Linting

Running linter locally: To run the Python linter on a specific file, run something like `flake8 ray/python/ray/worker.py`. You may need to first run `pip install flake8`.

Autoformatting code. We use yapf for linting, and the config file is located at `.style.yapf`. We recommend running `scripts/yapf.sh` prior to pushing to format changed files. Note that some projects such as dataframes and rllib are currently excluded.

4.51 Profiling for Ray Developers

This document details, for Ray developers, how to use `pprof` to profile Ray binaries.

4.51.1 Installation

These instructions are for Ubuntu only. Attempts to get `pprof` to correctly symbolize on Mac OS have failed.

```
sudo apt-get install google-perftools libgoogle-perftools-dev
```

4.51.2 Launching the to-profile binary

If you want to launch Ray in profiling mode, define the following variables:

```
export RAYLET_PERFTOOLS_PATH=/usr/lib/x86_64-linux-gnu/libprofiler.so
export RAYLET_PERFTOOLS_LOGFILE=/tmp/pprof.out
```

The file `/tmp/pprof.out` will be empty until you let the binary run the target workload for a while and then `kill` it via `ray stop` or by letting the driver exit.

4.51.3 Visualizing the CPU profile

The output of `pprof` can be visualized in many ways. Here we output it as a zoomable `.svg` image displaying the call graph annotated with hot paths.

```
# Use the appropriate path.
RAYLET=ray/python/ray/core/src/ray/raylet/raylet

google-pprof -svg $RAYLET /tmp/pprof.out > /tmp/pprof.svg
# Then open the .svg file with Chrome.

# If you realize the call graph is too large, use -focus=<some function> to zoom
# into subtrees.
google-pprof -focus=epoll_wait -svg $RAYLET /tmp/pprof.out > /tmp/pprof.svg
```

Here's a snapshot of an example svg output, taken from the official documentation:

4.51.4 References

- The pprof documentation.
- A Go version of pprof.
- The gperftools, including libprofiler, tcmalloc, and other goodies.

4.52 An Overview of the Internals

In this document, we overview the internal architecture of Ray.

4.52.1 Connecting to Ray

There are two ways that a Ray script can be initiated. It can either be run in a standalone fashion or it can be connect to an existing Ray cluster.

Running Ray standalone

Ray can be used standalone by calling `ray.init()` within a script. When the call to `ray.init()` happens, all of the relevant processes are started. These include a raylet, an object store and manager, a Redis server, and a number of worker processes.

When the script exits, these processes will be killed.

Note: This approach is limited to a single machine.

Connecting to an existing Ray cluster

To connect to an existing Ray cluster, simply pass the argument address of the Redis server as the `redis_address=` keyword argument into `ray.init`. In this case, no new processes will be started when `ray.init` is called, and similarly the processes will continue running when the script exits. In this case, all processes except workers that correspond to actors are shared between different driver processes.

4.52.2 Ray Processes

When using Ray, several processes are involved.

- Multiple **worker** processes execute tasks and store results in object stores. Each worker is a separate process.
- One **object store** per node stores immutable objects in shared memory and allows workers to efficiently share objects on the same node with minimal copying and deserialization.
- One **raylet** per node assigns tasks to workers on the same node.
- A **driver** is the Python process that the user controls. For example, if the user is running a script or using a Python shell, then the driver is the Python process that runs the script or the shell. A driver is similar to a worker in that it can submit tasks to its raylet and get objects from the object store, but it is different in that the raylet will not assign tasks to the driver to be executed.
- A **Redis server** maintains much of the system's state. For example, it keeps track of which objects live on which machines and of the task specifications (but not data). It can also be queried directly for debugging purposes.

4.52.3 Defining a remote function

A central component of this system is the **centralized control plane**. This is implemented using one or more Redis servers. Redis is an in-memory key-value store.

We use the centralized control plane in two ways. First, as persistent store of the system's control state. Second, as a message bus for communication between processes (using Redis's publish-subscribe functionality).

Now, consider a remote function definition as below.

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

When the remote function is defined as above, the function is immediately pickled, assigned a unique ID, and stored in a Redis server.

Each worker process has a separate thread running in the background that listens for the addition of remote functions to the centralized control state. When a new remote function is added, the thread fetches the pickled remote function, unpickles it, and can then execute that function.

4.52.4 Calling a remote function

When a driver or worker invokes a remote function, a number of things happen.

- First, a task object is created. The task object includes the following.
  - The ID of the function being called.
  - The IDs or values of the arguments to the function. Python primitives like integers or short strings will be pickled and included as part of the task object. Larger or more complex objects will be put into the object store with an internal call to `ray.put`, and the resulting IDs are included in the task object. Object IDs that are passed directly as arguments are also included in the task object.
  - The ID of the task. This is generated uniquely from the above content.
  - The IDs for the return values of the task. These are generated uniquely from the above content.
- The task object is then sent to the raylet on the same node as the driver or worker.
- The raylet makes a decision to either schedule the task locally or to pass the task on to another raylet.
  - If all of the task's object dependencies are present in the local object store and there are enough CPU and GPU resources available to execute the task, then the raylet will assign the task to one of its available workers.
  - If those conditions are not met, the task will be forwarded to another raylet. This is done by peer-to-peer connection between raylets. The task table can be inspected as follows.

`ray.`**`tasks`**`(`*task_id=None*`)`

Fetch and parse the task table information for one or more task IDs.

Parameters **`task_id`** – A hex string of the task ID to fetch information about. If this is None, then the task object table is fetched.

Returns Information from the task table.

- Once a task has been scheduled to a raylet, the raylet queues the task for execution. A task is assigned to a worker when enough resources become available and the object dependencies are available locally, in first-in, first-out order.
- When the task has been assigned to a worker, the worker executes the task and puts the task's return values into the object store. The object store will then update the **object table**, which is part of the centralized control state, to reflect the fact that it contains the newly created objects. The object table can be viewed as follows.

`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.

- When the task's return values are placed into the object store, they are first serialized into a contiguous blob of bytes using the Apache Arrow data layout, which is helpful for efficiently sharing data between processes using shared memory.

Notes and limitations

- When an object store on a particular node fills up, it will begin evicting objects in a least-recently-used manner. If an object that is needed later is evicted, then the call to `ray.get` for that object will initiate the reconstruction of the object. The raylet will attempt to reconstruct the object by replaying its task lineage.

4.52.5 Getting an object ID

Several things happen when a driver or worker calls `ray.get` on an object ID.

```
ray.get(x_id)
```

- The driver or worker goes to the object store on the same node and requests the relevant object. Each object store consists of two components, a shared-memory key-value store of immutable objects, and a manager to coordinate the transfer of objects between nodes.
  - If the object is not present in the object store, the manager checks the object table to see which other object stores, if any, have the object. It then requests the object directly from one of those object stores, via its manager. If the object doesn't exist anywhere, then the centralized control state will notify the requesting manager when the object is created. If the object doesn't exist anywhere because it has been evicted from all object stores, the worker will also request reconstruction of the object from the raylet. These checks repeat periodically until the object is available in the local object store, whether through reconstruction or through object transfer.
- Once the object is available in the local object store, the driver or worker will map the relevant region of memory into its own address space (to avoid copying the object), and will deserialize the bytes into a Python object. Note that any numpy arrays that are part of the object will not be copied.

4.53 Fault Tolerance

This document describes the handling of failures in Ray.

4.53.1 Machine and Process Failures

Each **raylet** (the scheduler process) sends heartbeats to a **monitor** process. If the monitor does not receive any heartbeats from a given raylet for some period of time (about ten seconds), then it will mark that process as dead.

4.53.2 Lost Objects

If an object is needed but is lost or was never created, then the task that created the object will be re-executed to create the object. If necessary, tasks needed to create the input arguments to the task being re-executed will also be re-executed. This is the standard *lineage-based fault tolerance* strategy used by other systems like Spark.

4.53.3 Actors

When an actor dies (either because the actor process crashed or because the node that the actor was on died), by default any attempt to get an object from that actor that cannot be created will raise an exception. Subsequent releases will include an option for automatically restarting actors.

4.53.4 Current Limitations

At the moment, Ray does not handle all failure scenarios. We are working on addressing these known problems.

Process Failures

1. Ray does not recover from the failure of any of the following processes: a Redis server and the monitor process.
2. If a driver fails, that driver will not be restarted and the job will not complete.

Lost Objects

1. If an object is constructed by a call to `ray.put` on the driver, is then evicted, and is later needed, Ray will not reconstruct this object.
2. If an object is constructed by an actor method, is then evicted, and is later needed, Ray will not reconstruct this object.

4.54 Contributing to Ray

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.

4.54.1 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.

4.54.2 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.

4.54.3 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/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
```

4.54.4 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.

Note: These tips are based off of the TVM contributor guide.

Python Module Index

Index

Symbols