RLLib: Ray’s scalable reinforcement learning library

This document describes Ray’s reinforcement learning library. It currently supports the following algorithms:

- Proximal Policy Optimization which is a proximal variant of TRPO.
- Evolution Strategies which is decribed in this paper. Our implementation borrows code from here.
- The Asynchronous Advantage Actor-Critic based on the OpenAI starter agent.

Proximal Policy Optimization scales to hundreds of cores and several GPUs, Evolution Strategies to clusters with thousands of cores and the Asynchronous Advantage Actor-Critic scales to dozens of cores on a single node.

These algorithms can be run on any OpenAI gym MDP, including custom ones written and registered by the user.

Getting Started

You can run training with

```
python ray/python/ray/rllib/train.py --env CartPole-v0 --alg PPO --config
'{"timesteps_per_batch": 10000}'
```

By default, the results will be logged to a subdirectory of `/tmp/ray`. This subdirectory will contain a file `config.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=/tmp/ray
```

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

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

The most important options are for choosing the environment with `--env` (any OpenAI gym environment including ones registered by the user can be used) and for choosing the algorithm with `--alg` (available options are `PPO`, `A3C`, `ES` and `DQN`). Each algorithm has specific hyperparameters that can be set with `--config`, see the `DEFAULT_CONFIG` variable in PPO, A3C, ES and DQN.

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!

The User API

You will be using this part of the API if you run the existing algorithms on a new problem. Note that the API is not considered to be stable yet. Here is an example how to use it:

```python
import ray
import ray.rllib.ppo as ppo

ray.init()

config = ppo.DEFAULT_CONFIG.copy()
alg = ppo.PPOAgent("CartPole-v1", config)

# Can optionally call alg.restore(path) to load a checkpoint.

for i in range(10):
   # Perform one iteration of the algorithm.
   result = alg.train()
   print("result: {}".format(result))
   print("checkpoint saved at path: {}".format(alg.save()))
```

The Developer API

This part of the API will be useful if you need to change existing RL algorithms or implement new ones. Note that the API is not considered to be stable yet.

Agents

Agents implement a particular algorithm and can be used to run some number of iterations of the algorithm, save and load the state of training and evaluate the current policy. All agents inherit from a common base class:

class **`ray.rllib.common.Agent`**(*env_name, config, upload_dir=None*)

All RLlib agents extend this base class.

Agent objects retain internal model state between calls to train(), so you should create a new agent instance for each training session.

`env_name`

str – Name of the OpenAI gym environment to train against.

`config`

obj – Algorithm-specific configuration data.

`logdir`

str – Directory in which training outputs should be placed.

`compute_action`(*observation*)

Computes an action using the current trained policy.

`restore`(*checkpoint_path*)

Restores training state from a given model checkpoint.

These checkpoints are returned from calls to save().

`save`()

Saves the current model state to a checkpoint.

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

`train`()

Runs one logical iteration of training.

Returns: A TrainingResult that describes training progress.

Models

Models are subclasses of the Model class:

class `ray.rllib.models.Model`(*inputs, num_outputs, options*)

Defines an abstract network model for use with RLlib.

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

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

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

`inputs`

Tensor – The input placeholder for this model.

`outputs`

Tensor – The output vector of this model.

`last_layer`

Tensor – The network layer right before the model output.

Currently we support fully connected policies, convolutional policies and LSTMs:

`ray.rllib.models.FullyConnectedNetwork`(*inputs, num_outputs, options*)

Generic fully connected network.

`ray.rllib.models.ConvolutionalNetwork`(*inputs, num_outputs, options*)

Generic convolutional network.

`ray.rllib.models.LSTM`(*inputs, num_outputs, options*)

Action Distributions

Actions can be sampled from different distributions, they have a common base class:

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

The policy action distribution of an agent.

Parameters: **inputs** (*Tensor*) – The input vector to compute samples from.

`entropy`()

The entroy of the action distribution.

`kl`(*other*)

The KL-divergene between two action distributions.

`logp`(*x*)

The log-likelihood of the action distribution.

`sample`()

Draw a sample from the action distribution.

Currently we support the following action distributions:

`ray.rllib.models.Categorical`(*inputs*)

Categorical distribution for discrete action spaces.

`ray.rllib.models.DiagGaussian`(*inputs*)

Action distribution where each vector element is a gaussian.

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

`ray.rllib.models.Deterministic`(*inputs*)

Action distribution that returns the input values directly.

This is similar to DiagGaussian with standard deviation zero.

The Model Catalog

To make picking the right action distribution and models easier, there is a mechanism to pick good default values for various gym environments.

class `ray.rllib.models.ModelCatalog`

Registry of default models and action distributions for envs.

Example

dist_class, dist_dim = ModelCatalog.get_action_dist(env.action_space) model = ModelCatalog.get_model(inputs, dist_dim) dist = dist_class(model.outputs) action_op = dist.sample()

static `get_action_dist`(*action_space*, *dist_type=None*)

Returns action distribution class and size for the given action space.

Parameters:
- **action_space** (*Space*) – Action space of the target gym env.
- **dist_type** (*Optional[str]*) – Identifier of the action distribution.

Returns: Python class of the distribution. dist_dim (int): The size of the input vector to the distribution.

Return type: dist_class (ActionDistribution)

static **`get_model`**(*inputs*, *num_outputs*, *options={}*)

Returns a suitable model conforming to given input and output specs.

Parameters:
- **inputs** (*Tensor*) – The input tensor to the model.
- **num_outputs** (*int*) – The size of the output vector of the model.
- **options** (*dict*) – Optional args to pass to the model constructor.

Returns: Neural network model.

Return type: model (Model)

static **`get_preprocessor`**(*env_name*, *obs_shape*, *options={}*)

Returns a suitable processor for the given environment.

Parameters:
- **env_name** (*str*) – The name of the environment.
- **obs_shape** (*tuple*) – The shape of the env observation space.

Returns: Preprocessor for the env observations.

Return type: preprocessor (Preprocessor)

Using RLLib on a cluster

First create a cluster as described in managing a cluster with parallel ssh. You can then run RLLib on this cluster by passing the address of the main redis shard into `train.py` with `--redis-address`.