rlpyt

Sep 14, 2020

Contents:

rlpyt includes modular, optimized implementations of common deep RL algorithms in PyTorch, with unified infrastructure supporting all three major families of model-free algorithms: policy gradient, deep-q learning, and q-function policy gradient. It is intended to be a high-throughput code-base for small- to medium-scale research (large-scale meaning like OpenAI Dota with 100's GPUs). A conceptual overview is provided in the white paper, and the code (with examples) in the github repository.

This documentation aims to explain the intent of the code structure, to make it easier to use and modify (it might not detail every keyword argument as in a fixed library). See the github README for installation instructions and other introductory notes. Please share any questions or comments to do with documenantation on the github issues.

The sections are organized as follows. First, several of the base classes are introduced. Then, each algorithm family and associated agents and models are grouped together. Infrastructure code such as the runner classes and sampler classes are covered next. All the remaining components are covered thereafter, in no particular order.

CHAPTER 1

Base Classes and Interfaces

This page describes the base classes for three main components: algorithm, agent, and environment. These are the most likely to need modification for a new project. Intended interfaces to the infrastructure code (i.e. runner and sampler) are specified here. More details on specific instances of these components appear in following pages.

Commonly, these classes will simply store their keyword arguments when instantiated, and actual initialization occurs in methods to be called later by the runner or sampler.

1.1 Algorithms

class `rlpyt.algos.base.`**`RlAlgorithm`**

Trains the agent using gathered samples, for example by constructing TD-errors and performing gradient descent on the agent's model parameters. Includes pre-processing of samples e.g. discounting returns.

`initialize`(*agent*, *n_itr*, *batch_spec*, *mid_batch_reset*, *examples*, *world_size=1*, *rank=0*)

Typically called in the runner during startup.

Parameters

- **`agent`** – The learning agent instance.
- **`n_itr`** (*`int`*) – Number of training loop iterations which will be run (e.g. corresponds to each call of `optimize_agent()`)
- **`batch_spec`** – Holds sampler batch dimensions.
- **`mid_batch_reset`** (*`bool`*) – Whether the sampler resets environments during a sampling batch (`True`) or only between batches (`False`). Affects whether some samples are invalid for training.
- **`examples`** – Structure of example RL quantities, e.g. observation, action, agent_info, env_info, e.g. in case needed to allocate replay buffer.
- **`world_size`** (*`int`*) – Number of separate optimizing processes (e.g. multi-GPU).
- **`rank`** (*`int`*) – Unique index for each optimizing process.

async_initialize(*agent*, *sampler_n_itr*, *batch_spec*, *mid_batch_reset*, *examples*, *world_size=1*)
Called instead of `initialize()` in async runner (not needed unless using async runner). Should return async replay_buffer using shared memory.

optim_initialize(*rank=0*)
Called in async runner which requires two stages of initialization; might also be used in `initialize()` to avoid redundant code.

optimize_agent(*itr*, *samples=None*, *sampler_itr=None*)
Train the agent for some number of parameter updates, e.g. either using new samples or a replay buffer.

Typically called in the runner's training loop.

Parameters

- **itr** (*int*) – Iteration of the training loop.
- **samples** – New samples from the sampler (for `None` case, see async runner).
- **sampler_itr** – For case other than `None`, see async runner.

optim_state_dict()
Return the optimizer state dict (e.g. Adam); overwrite if using multiple optimizers.

load_optim_state_dict(*state_dict*)
Load an optimizer state dict; should expect the format returned from `optim_state_dict()`.

1.2 Environments

Environments are expected to input/output numpy arrays.

class rlpyt.envs.base.**Env**
The learning task, e.g. an MDP containing a transition function T(state, action)–>state'. Has a defined observation space and action space.

step(*action*)
Run on timestep of the environment's dynamics using the input action, advancing the internal state; T(state,action)–>state'.

Parameters **action** – An element of this environment's action space.

Returns An element of this environment's observation space corresponding to the next state. reward (float): A scalar reward resulting from the state transition. done (bool): Indicates whether the episode has ended. info (namedtuple): Additional custom information.

Return type observation

reset()
Resets the state of the environment.

Returns The initial observation of the new episode.

Return type observation

1.3 Agents

Agents are expected to input/output torch tensors.

class `rlpyt.agents.base.`**`BaseAgent`**(*ModelCls=None*, *model_kwargs=None*, *initial_model_state_dict=None*)

The agent performs many functions, including: action-selection during sampling, returning policy-related values to use in training (e.g. action probabilities), storing recurrent state during sampling, managing model device, and performing model parameter communication between processes. The agent is both interfaces: sampler<->neural network<->algorithm. Typically, each algorithm and environment combination will require at least some of its own agent functionality.

The base agent automatically carries out some of these roles. It assumes there is one neural network model. Agents using multiple models might need to extend certain funcionality to include those models, depending on how they are used.

`__init__`(*ModelCls=None*, *model_kwargs=None*, *initial_model_state_dict=None*)

Arguments are saved but no model initialization occurs.

Parameters

- **`ModelCls`** – The model class to be used.
- **`model_kwargs`** (*`optional`*) – Any keyword arguments to pass when instantiating the model.
- **`initial_model_state_dict`** (*`optional`*) – Initial model parameter values.

`__call__`(*observation*, *prev_action*, *prev_reward*)

Returns values from model forward pass on training data (i.e. used in algorithm).

`initialize`(*env_spaces*, *share_memory=False*, ***kwargs*)

Instantiates the neural net model(s) according to the environment interfaces.

Uses shared memory as needed–e.g. in CpuSampler, workers have a copy of the agent for action-selection. The workers automatically hold up-to-date parameters in `model`, because they exist in shared memory, constructed here before worker processes fork. Agents with additional model components (beyond `self.model`) for action-selection should extend this method to share those, as well.

Typically called in the sampler during startup.

Parameters

- **`env_spaces`** – passed to `make_env_to_model_kwargs()`, typically namedtuple of 'observation' and 'action'.
- **`share_memory`** (*`bool`*) – whether to use shared memory for model parameters.

`make_env_to_model_kwargs`(*env_spaces*)

Generate any keyword args to the model which depend on environment interfaces.

`to_device`(*cuda_idx=None*)

Moves the model to the specified cuda device, if not `None`. If sharing memory, instantiates a new model to preserve the shared (CPU) model. Agents with additional model components (beyond `self.model`) for action-selection or for use during training should extend this method to move those to the device, as well.

Typically called in the runner during startup.

`data_parallel`()

Wraps the model with PyTorch's DistributedDataParallel. The intention is for rlpyt to create a separate Python process to drive each GPU (or CPU-group for CPU-only, MPI-like configuration). Agents with additional model components (beyond `self.model`) which will have gradients computed through them should extend this method to wrap those, as well.

Typically called in the runner during startup.

async_cpu(*share_memory=True*)
Used in async runner only; creates a new model instance to be used in the sampler, separate from the model shared with the optimizer process. The sampler can operate asynchronously, and choose when to copy the optimizer's (shared) model parameters into its model (under read-write lock). The sampler model may be stored in shared memory, as well, to instantly share values with sampler workers. Agents with additional model components (beyond `self.model`) should extend this method to do the same with those, if using in asynchronous mode.

Typically called in the runner during startup.

TODO: double-check wording if this happens in sampler and optimizer.

step(*observation*, *prev_action*, *prev_reward*)
Returns selected actions for environment instances in sampler.

state_dict()
Returns model parameters for saving.

load_state_dict(*state_dict*)
Load model parameters, should expect format returned from `state_dict()`.

train_mode(*itr*)
Go into training mode (e.g. see PyTorch's `Module.train()`).

sample_mode(*itr*)
Go into sampling mode.

eval_mode(*itr*)
Go into evaluation mode. Example use could be to adjust epsilon-greedy.

sync_shared_memory()
Copies model parameters into shared_model, e.g. to make new values available to sampler workers. If running CPU-only, these will be the same object–no copy necessary. If model is on GPU, copy to CPU is performed. (Requires `initialize(share_memory=True)` called previously. Not used in async mode.

Typically called in the XXX during YY.

send_shared_memory()
Used in async mode only, in optimizer process; copies parameters from trained model (maybe GPU) to shared model, which the sampler can access. Does so under write-lock, and increments send-count which sampler can check.

Typically called in the XXX during YY.

recv_shared_memory()
Used in async mode, in sampler process; copies parameters from model shared with optimizer into local model, if shared model has been updated. Does so under read-lock. (Local model might also be shared with sampler workers).

Typically called in the XXX during YY.

1.3.1 Recurrent Agents

class `rlpyt.agents.base.`**RecurrentAgentMixin**(**args*, ***kwargs*)
Mixin class to manage recurrent state during sampling (so the sampler remains agnostic). To be used like `class MyRecurrentAgent(RecurrentAgentMixin, MyAgent):`.

reset()
Sets the recurrent state to `None`, which built-in PyTorch modules conver to zeros.

reset_one(*idx*)
Sets the recurrent state corresponding to one environment instance to zero. Assumes rnn state is in cudnn-compatible shape: [N,B,H], where B corresponds to environment index.

advance_rnn_state(*new_rnn_state*)
Sets the recurrent state to the newly computed one (i.e. recurrent agents should call this at the end of their `step()`).

train_mode(*itr*)
If coming from sample mode, store the rnn state elsewhere and clear it.

sample_mode(*itr*)
If coming from non-sample modes, restore the last sample-mode rnn state.

eval_mode(*itr*)
If coming from sample mode, store the rnn state elsewhere and clear it.

class `rlpyt.agents.base.`**AlternatingRecurrentAgentMixin**(**args*, ***kwargs*)
Maintain an alternating pair of recurrent states to use when stepping in the sampler. Automatically swap them out when `advance_rnn_state()` is called, so it otherwise behaves like regular recurrent agent. Should use only in alternating samplers, where two sets of environment instances take turns stepping (no special class needed for feedforward agents). Use in place of `RecurrentAgentMixin`.

CHAPTER 2

Policy Gradient Implementations

This page documents the implemented policy gradient / actor-critic algorithms, agents, and models.

2.1 Algorithms

class `rlpyt.algos.pg.base.`**`PolicyGradientAlgo`**

Bases: *`rlpyt.algos.base.RlAlgorithm`*

Base policy gradient / actor-critic algorithm, which includes initialization procedure and processing of data samples to compute advantages.

`initialize`(*agent*, *n_itr*, *batch_spec*, *mid_batch_reset=False*, *examples=None*, *world_size=1*, *rank=0*)

Build the torch optimizer and store other input attributes. Params `batch_spec` and `examples` are unused.

`process_returns`(*samples*)

Compute bootstrapped returns and advantages from a minibatch of samples. Uses either discounted returns (if `self.gae_lambda==1`) or generalized advantage estimation. Mask out invalid samples according to `mid_batch_reset` or for recurrent agent. Optionally, normalize advantages.

class `rlpyt.algos.pg.a2c.`**`A2C`**(*discount=0.99*, *learning_rate=0.001*, *value_loss_coeff=0.5*, *entropy_loss_coeff=0.01*, *OptimCls=<sphinx.ext.autodoc.importer._MockObject object>*, *optim_kwargs=None*, *clip_grad_norm=1.0*, *initial_optim_state_dict=None*, *gae_lambda=1*, *normalize_advantage=False*)

Bases: *`rlpyt.algos.pg.base.PolicyGradientAlgo`*

Advantage Actor Critic algorithm (synchronous). Trains the agent by taking one gradient step on each iteration of samples, with advantages computed by generalized advantage estimation.

`__init__`(*discount=0.99, learning_rate=0.001, value_loss_coeff=0.5, entropy_loss_coeff=0.01, OptimCls=<sphinx.ext.autodoc.importer._MockObject object>, optim_kwargs=None, clip_grad_norm=1.0, initial_optim_state_dict=None, gae_lambda=1, normalize_advantage=False*)
Saves the input settings.

`optimize_agent`(*itr, samples*)
Train the agent on input samples, by one gradient step.

`loss`(*samples*)
Computes the training loss: policy_loss + value_loss + entropy_loss. Policy loss: log-likelihood of actions * advantages Value loss: 0.5 * (estimated_value - return) ^ 2 Organizes agent inputs from training samples, calls the agent instance to run forward pass on training data, and uses the `agent.distribution` to compute likelihoods and entropies. Valid for feedforward or recurrent agents.

class `rlpyt.algos.pg.ppo.`**`PPO`**(*discount=0.99, learning_rate=0.001, value_loss_coeff=1.0, entropy_loss_coeff=0.01, OptimCls=<sphinx.ext.autodoc.importer._MockObject object>, optim_kwargs=None, clip_grad_norm=1.0, initial_optim_state_dict=None, gae_lambda=1, minibatches=4, epochs=4, ratio_clip=0.1, linear_lr_schedule=True, normalize_advantage=False*)
Bases: *`rlpyt.algos.pg.base.PolicyGradientAlgo`*

Proximal Policy Optimization algorithm. Trains the agent by taking multiple epochs of gradient steps on minibatches of the training data at each iteration, with advantages computed by generalized advantage estimation. Uses clipped likelihood ratios in the policy loss.

`__init__`(*discount=0.99, learning_rate=0.001, value_loss_coeff=1.0, entropy_loss_coeff=0.01, OptimCls=<sphinx.ext.autodoc.importer._MockObject object>, optim_kwargs=None, clip_grad_norm=1.0, initial_optim_state_dict=None, gae_lambda=1, minibatches=4, epochs=4, ratio_clip=0.1, linear_lr_schedule=True, normalize_advantage=False*)
Saves input settings.

`initialize`(**args, **kwargs*)
Extends base `initialize()` to initialize learning rate schedule, if applicable.

`optimize_agent`(*itr, samples*)
Train the agent, for multiple epochs over minibatches taken from the input samples. Organizes agent inputs from the training data, and moves them to device (e.g. GPU) up front, so that minibatches are formed within device, without further data transfer.

`loss`(*agent_inputs, action, return_, advantage, valid, old_dist_info, init_rnn_state=None*)
Compute the training loss: policy_loss + value_loss + entropy_loss Policy loss: min(likelhood-ratio * advantage, clip(likelihood_ratio, 1-eps, 1+eps) * advantage) Value loss: 0.5 * (estimated_value - return) ^ 2 Calls the agent to compute forward pass on training data, and uses the `agent.distribution` to compute likelihoods and entropies. Valid for feedforward or recurrent agents.

2.2 Agents

2.2.1 Continuous Actions

class `rlpyt.agents.pg.gaussian.`**`GaussianPgAgent`**(*ModelCls=None, model_kwargs=None, initial_model_state_dict=None*)
Bases: *`rlpyt.agents.base.BaseAgent`*

Agent for policy gradient algorithm using Gaussian action distribution.

`__call__`(*observation*, *prev_action*, *prev_reward*)
: Performs forward pass on training data, for algorithm.

`initialize`(*env_spaces*, *share_memory=False*, *global_B=1*, *env_ranks=None*)
: Extends base method to build Gaussian distribution.

`step`(*observation*, *prev_action*, *prev_reward*)
: Compute policy's action distribution from inputs, and sample an action. Calls the model to produce mean, log_std, and value estimate. Moves inputs to device and returns outputs back to CPU, for the sampler. (no grad)

`value`(*observation*, *prev_action*, *prev_reward*)
: Compute the value estimate for the environment state, e.g. for the bootstrap value, V(s_{T+1}), in the sampler. (no grad)

class `rlpyt.agents.pg.gaussian.`**`RecurrentGaussianPgAgentBase`**(*ModelCls=None*, *model_kwargs=None*, *initial_model_state_dict=None*)

Bases: *`rlpyt.agents.base.BaseAgent`*

`__call__`(*observation*, *prev_action*, *prev_reward*, *init_rnn_state*)
: Performs forward pass on training data, for algorithm (requires recurrent state input).

`step`(*observation*, *prev_action*, *prev_reward*)
: Compute policy's action distribution from inputs, and sample an action. Calls the model to produce mean, log_std, value estimate, and next recurrent state. Moves inputs to device and returns outputs back to CPU, for the sampler. Advances the recurrent state of the agent. (no grad)

`value`(*observation*, *prev_action*, *prev_reward*)
: Compute the value estimate for the environment state using the currently held recurrent state, without advancing the recurrent state, e.g. for the bootstrap value V(s_{T+1}), in the sampler. (no grad)

class `rlpyt.agents.pg.gaussian.`**`RecurrentGaussianPgAgent`**(**args*, ***kwargs*)

Bases: *`rlpyt.agents.base.RecurrentAgentMixin`*, *`rlpyt.agents.pg.gaussian.RecurrentGaussianPgAgentBase`*

class `rlpyt.agents.pg.gaussian.`**`AlternatingRecurrentGaussianPgAgent`**(**args*, ***kwargs*)

Bases: *`rlpyt.agents.base.AlternatingRecurrentAgentMixin`*, *`rlpyt.agents.pg.gaussian.RecurrentGaussianPgAgentBase`*

class `rlpyt.agents.pg.mujoco.`**`MujocoMixin`**

Mixin class defining which environment interface properties are given to the model. Now supports observation normalization, including multi-GPU.

`make_env_to_model_kwargs`(*env_spaces*)
: Extract observation_shape and action_size.

class `rlpyt.agents.pg.mujoco.`**`MujocoFfAgent`**(*ModelCls=<class 'rlpyt.models.pg.mujoco_ff_model.MujocoFfModel'>*, ***kwargs*)

Bases: *`rlpyt.agents.pg.mujoco.MujocoMixin`*, *`rlpyt.agents.pg.gaussian.GaussianPgAgent`*

`__init__`(*ModelCls=<class 'rlpyt.models.pg.mujoco_ff_model.MujocoFfModel'>*, ***kwargs*)
: Initialize self. See help(type(self)) for accurate signature.

class `rlpyt.agents.pg.mujoco.`**`MujocoLstmAgent`**(*ModelCls=<class 'rlpyt.models.pg.mujoco_lstm_model.MujocoLstmModel'>*, ***kwargs*)

Bases: *rlpyt.agents.pg.mujoco.MujocoMixin*, *rlpyt.agents.pg.gaussian.RecurrentGaussianPgAgent*

__init__(*ModelCls=<class 'rlpyt.models.pg.mujoco_lstm_model.MujocoLstmModel'>*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

2.2.2 Discrete Actions

class rlpyt.agents.pg.categorical.**CategoricalPgAgent**(*ModelCls=None*, *model_kwargs=None*, *initial_model_state_dict=None*)
Bases: *rlpyt.agents.base.BaseAgent*

Agent for policy gradient algorithm using categorical action distribution. Same as `GausssianPgAgent` and related classes, except uses `Categorical` distribution, and has a different interface to the model (model here outputs discrete probabilities in place of means and log_stds, while both output the value estimate).

class rlpyt.agents.pg.atari.**AtariMixin**
Mixin class defining which environment interface properties are given to the model.

make_env_to_model_kwargs(*env_spaces*)
Extract image shape and action size.

class rlpyt.agents.pg.atari.**AtariFfAgent**(*ModelCls=<class 'rlpyt.models.pg.atari_ff_model.AtariFfModel'>*, ***kwargs*)
Bases: *rlpyt.agents.pg.atari.AtariMixin*, *rlpyt.agents.pg.categorical.CategoricalPgAgent*

__init__(*ModelCls=<class 'rlpyt.models.pg.atari_ff_model.AtariFfModel'>*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class rlpyt.agents.pg.atari.**AtariLstmAgent**(*ModelCls=<class 'rlpyt.models.pg.atari_lstm_model.AtariLstmModel'>*, ***kwargs*)
Bases: *rlpyt.agents.pg.atari.AtariMixin*, `rlpyt.agents.pg.categorical.RecurrentCategoricalPgAgent`

__init__(*ModelCls=<class 'rlpyt.models.pg.atari_lstm_model.AtariLstmModel'>*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

2.3 Models

class rlpyt.models.pg.mujoco_ff_model.**MujocoFfModel**(*observation_shape*, *action_size*, *hidden_sizes=None*, *hidden_nonlinearity=<sphinx.ext.autodoc.importer._MockO object>*, *mu_nonlinearity=<sphinx.ext.autodoc.importer._MockOb object>*, *init_log_std=0.0*, *normalize_observation=False*, *norm_obs_clip=10*, *norm_obs_var_clip=1e-06*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Model commonly used in Mujoco locomotion agents: an MLP which outputs distribution means, separate parameter for learned log_std, and separate MLP for state-value estimate.

__init__(*observation_shape*, *action_size*, *hidden_sizes=None*, *hidden_nonlinearity=<sphinx.ext.autodoc.importer._MockObject object>*, *mu_nonlinearity=<sphinx.ext.autodoc.importer._MockObject object>*, *init_log_std=0.0*, *normalize_observation=False*, *norm_obs_clip=10*, *norm_obs_var_clip=1e-06*)
Instantiate neural net modules according to inputs.

forward(*observation*, *prev_action*, *prev_reward*)
Compute mean, log_std, and value estimate from input state. Infers leading dimensions of input: can be [T,B], [B], or []; provides returns with same leading dims. Intermediate feedforward layers process as [T*B,H], with T=1,B=1 when not given. Used both in sampler and in algorithm (both via the agent).

class `rlpyt.models.pg.mujoco_lstm_model.`**MujocoLstmModel**(*observation_shape*, *action_size*, *hidden_sizes=None*, *lstm_size=256*, *nonlinearity=<sphinx.ext.autodoc.importer._MockObject object>*, *normalize_observation=False*, *norm_obs_clip=10*, *norm_obs_var_clip=1e-06*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Recurrent model for Mujoco locomotion agents: an MLP into an LSTM which outputs distribution means, log_std, and state-value estimate.

__init__(*observation_shape*, *action_size*, *hidden_sizes=None*, *lstm_size=256*, *nonlinearity=<sphinx.ext.autodoc.importer._MockObject object>*, *normalize_observation=False*, *norm_obs_clip=10*, *norm_obs_var_clip=1e-06*)
Initialize self. See help(type(self)) for accurate signature.

forward(*observation*, *prev_action*, *prev_reward*, *init_rnn_state*)
Compute mean, log_std, and value estimate from input state. Infer leading dimensions of input: can be [T,B], [B], or []; provides returns with same leading dims. Intermediate feedforward layers process as [T*B,H], and recurrent layers as [T,B,H], with T=1,B=1 when not given. Used both in sampler and in algorithm (both via the agent). Also returns the next RNN state.

class `rlpyt.models.pg.atari_ff_model.`**AtariFfModel**(*image_shape*, *output_size*, *fc_sizes=512*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Feedforward model for Atari agents: a convolutional network feeding an MLP with outputs for action probabilities and state-value estimate.

__init__(*image_shape*, *output_size*, *fc_sizes=512*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)
Instantiate neural net module according to inputs.

forward(*image*, *prev_action*, *prev_reward*)
Compute action probabilities and value estimate from input state. Infers leading dimensions of input: can be [T,B], [B], or []; provides returns with same leading dims. Convolution layers process as [T*B, *image_shape], with T=1,B=1 when not given. Expects uint8 images in [0,255] and converts them to float32 in [0,1] (to minimize image data storage and transfer). Used in both sampler and in algorithm (both via the agent).

class `rlpyt.models.pg.atari_lstm_model.`**`AtariLstmModel`**(*image_shape*, *output_size*, *fc_sizes=512*, *lstm_size=512*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Recurrent model for Atari agents: a convolutional network into an FC layer into an LSTM which outputs action probabilities and state-value estimate.

`__init__`(*image_shape*, *output_size*, *fc_sizes=512*, *lstm_size=512*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)

Instantiate neural net module according to inputs.

`forward`(*image*, *prev_action*, *prev_reward*, *init_rnn_state*)

Compute action probabilities and value estimate from input state. Infers leading dimensions of input: can be [T,B], [B], or []; provides returns with same leading dims. Convolution layers process as [T*B, *image_shape], with T=1,B=1 when not given. Expects uint8 images in [0,255] and converts them to float32 in [0,1] (to minimize image data storage and transfer). Recurrent layers processed as [T,B,H]. Used in both sampler and in algorithm (both via the agent). Also returns the next RNN state.

CHAPTER 3

Deep Q-Learning Implementations

This page documents the implemented deep Q-learning algorithms, agents, and models. Up to Rainbow, minus noisy nets, can be run using Categorical-DQN with the options for double-DQN, dueling heads, n-step returns, and prioritized replay.

3.1 DQN

class `rlpyt.algos.dqn.dqn.`**`DQN`**(*discount=0.99, batch_size=32, min_steps_learn=50000, delta_clip=1.0, replay_size=1000000, replay_ratio=8, target_update_tau=1, target_update_interval=312, n_step_return=1, learning_rate=0.00025, OptimCls=<sphinx.ext.autodoc.importer._MockObject object>, optim_kwargs=None, initial_optim_state_dict=None, clip_grad_norm=10.0, eps_steps=1000000, double_dqn=False, prioritized_replay=False, pri_alpha=0.6, pri_beta_init=0.4, pri_beta_final=1.0, pri_beta_steps=50000000, default_priority=None, ReplayBufferCls=None, updates_per_sync=1*)

Bases: `rlpyt.algos.base.RlAlgorithm`

DQN algorithm trainig from a replay buffer, with options for double-dqn, n-step returns, and prioritized replay.

`__init__`(*discount=0.99, batch_size=32, min_steps_learn=50000, delta_clip=1.0, replay_size=1000000, replay_ratio=8, target_update_tau=1, target_update_interval=312, n_step_return=1, learning_rate=0.00025, OptimCls=<sphinx.ext.autodoc.importer._MockObject object>, optim_kwargs=None, initial_optim_state_dict=None, clip_grad_norm=10.0, eps_steps=1000000, double_dqn=False, prioritized_replay=False, pri_alpha=0.6, pri_beta_init=0.4, pri_beta_final=1.0, pri_beta_steps=50000000, default_priority=None, ReplayBufferCls=None, updates_per_sync=1*)

Saves input arguments.

`delta_clip` selects the Huber loss; if `None`, uses MSE.

`replay_ratio` determines the ratio of data-consumption to data-generation. For example, original DQN sampled 4 environment steps between each training update with batch-size 32, for a replay ratio of 8.

initialize(*agent*, *n_itr*, *batch_spec*, *mid_batch_reset*, *examples*, *world_size=1*, *rank=0*)
Stores input arguments and initializes replay buffer and optimizer. Use in non-async runners. Computes number of gradient updates per optimization iteration as *(replay_ratio * sampler-batch-size / training-batch_size)*.

async_initialize(*agent*, *sampler_n_itr*, *batch_spec*, *mid_batch_reset*, *examples*, *world_size=1*)
Used in async runner only; returns replay buffer allocated in shared memory, does not instantiate optimizer.

optim_initialize(*rank=0*)
Called in initilize or by async runner after forking sampler.

initialize_replay_buffer(*examples*, *batch_spec*, *async_=False*)
Allocates replay buffer using examples and with the fields in *SamplesToBuffer* namedarraytuple. Uses frame-wise buffers, so that only unique frames are stored, using less memory (usual observations are 4 most recent frames, with only newest frame distince from previous observation).

optimize_agent(*itr*, *samples=None*, *sampler_itr=None*)
Extracts the needed fields from input samples and stores them in the replay buffer. Then samples from the replay buffer to train the agent by gradient updates (with the number of updates determined by replay ratio, sampler batch size, and training batch size). If using prioritized replay, updates the priorities for sampled training batches.

samples_to_buffer(*samples*)
Defines how to add data from sampler into the replay buffer. Called in optimize_agent() if samples are provided to that method. In asynchronous mode, will be called in the memory_copier process.

loss(*samples*)
Computes the Q-learning loss, based on: 0.5 * (Q - target_Q) ^ 2. Implements regular DQN or Double-DQN for computing target_Q values using the agent's target network. Computes the Huber loss using `delta_clip`, or if `None`, uses MSE. When using prioritized replay, multiplies losses by importance sample weights.

Input `samples` have leading batch dimension [B,..] (but not time).

Calls the agent to compute forward pass on training inputs, and calls `agent.target()` to compute target values.

Returns loss and TD-absolute-errors for use in prioritization.

> **Warning:** If not using mid_batch_reset, the sampler will only reset environments between iterations, so some samples in the replay buffer will be invalid. This case is not supported here currently.

class rlpyt.agents.dqn.epsilon_greedy.**EpsilonGreedyAgentMixin**(*eps_init=1*, *eps_final=0.01*, *eps_final_min=None*, *eps_itr_min=50*, *eps_itr_max=1000*, *eps_eval=0.001*, **args*, ***kwargs*)
Mixin class to operate all epsilon-greedy agents. Includes epsilon annealing, switching between sampling and evaluation epsilons, and vector-valued epsilons. The agent subclass must use a compatible epsilon-greedy distribution.

`__init__`(*eps_init=1*, *eps_final=0.01*, *eps_final_min=None*, *eps_itr_min=50*, *eps_itr_max=1000*, *eps_eval=0.001*, **args*, ***kwargs*)
Saves input arguments. `eps_final_min` other than `None` will use vector-valued epsilon, log-spaced.

`collector_initialize`(*global_B=1*, *env_ranks=None*)
For vector-valued epsilon, the agent inside the sampler worker process must initialize with its own epsilon values.

`make_vec_eps`(*global_B*, *env_ranks*)
Construct log-spaced epsilon values and select local assignments from the global number of sampler environment instances (for SyncRl and AsyncRl).

`sample_mode`(*itr*)
Extend method to set epsilon for sampling (including annealing).

`eval_mode`(*itr*)
Extend method to set epsilon for evaluation, using 1 for pre-training eval.

class `rlpyt.agents.dqn.dqn_agent.`**`DqnAgent`**(*eps_init=1*, *eps_final=0.01*, *eps_final_min=None*, *eps_itr_min=50*, *eps_itr_max=1000*, *eps_eval=0.001*, **args*, ***kwargs*)

Bases: *`rlpyt.agents.dqn.epsilon_greedy.EpsilonGreedyAgentMixin`*, *`rlpyt.agents.base.BaseAgent`*

Standard agent for DQN algorithms with epsilon-greedy exploration.

`__call__`(*observation*, *prev_action*, *prev_reward*)
Returns Q-values for states/observations (with grad).

`initialize`(*env_spaces*, *share_memory=False*, *global_B=1*, *env_ranks=None*)
Along with standard initialization, creates vector-valued epsilon for exploration, if applicable, with a different epsilon for each environment instance.

`step`(*observation*, *prev_action*, *prev_reward*)
Computes Q-values for states/observations and selects actions by epsilon-greedy. (no grad)

`target`(*observation*, *prev_action*, *prev_reward*)
Returns the target Q-values for states/observations.

`update_target`(*tau=1*)
Copies the model parameters into the target model.

class `rlpyt.models.dqn.atari_dqn_model.`**`AtariDqnModel`**(*image_shape*, *output_size*, *fc_sizes=512*, *dueling=False*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Standard convolutional network for DQN. 2-D convolution for multiple video frames per observation, feeding an MLP for Q-value outputs for the action set.

`__init__`(*image_shape*, *output_size*, *fc_sizes=512*, *dueling=False*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)
Instantiates the neural network according to arguments; network defaults stored within this method.

`forward`(*observation*, *prev_action*, *prev_reward*)
Compute action Q-value estimates from input state. Infers leading dimensions of input: can be [T,B], [B], or []; provides returns with same leading dims. Convolution layers process as [T*B, image_shape[0], image_shape[1],…,image_shape[-1]], with T=1,B=1 when not given. Expects uint8 images in [0,255]

and converts them to float32 in [0,1] (to minimize image data storage and transfer). Used in both sampler and in algorithm (both via the agent).

3.2 Categorical-DQN

class `rlpyt.algos.dqn.cat_dqn.`**`CategoricalDQN`**(*V_min=-10*, *V_max=10*, ***kwargs*)

Bases: `rlpyt.algos.dqn.dqn.DQN`

Distributional DQN with fixed probability bins for the Q-value of each action, a.k.a. categorical.

`__init__`(*V_min=-10*, *V_max=10*, ***kwargs*)

Standard __init__() plus Q-value limits; the agent configures the number of atoms (bins).

`loss`(*samples*)

Computes the Distributional Q-learning loss, based on projecting the discounted rewards + target Q-distribution into the current Q-domain, with cross-entropy loss.

Returns loss and KL-divergence-errors for use in prioritization.

class `rlpyt.agents.dqn.catdqn_agent.`**`CatDqnAgent`**(*n_atoms=51*, ***kwargs*)

Bases: `rlpyt.agents.dqn.dqn_agent.DqnAgent`

Agent for Categorical DQN algorithm.

`__init__`(*n_atoms=51*, ***kwargs*)

Standard init, and set the number of probability atoms (bins).

`step`(*observation*, *prev_action*, *prev_reward*)

Compute the discrete distribution for the Q-value for each action for each state/observation (no grad).

class `rlpyt.models.dqn.atari_catdqn_model.`**`AtariCatDqnModel`**(*image_shape*, *output_size*, *n_atoms=51*, *fc_sizes=512*, *dueling=False*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

2D conlutional network feeding into MLP with `n_atoms` outputs per action, representing a discrete probability distribution of Q-values.

`__init__`(*image_shape*, *output_size*, *n_atoms=51*, *fc_sizes=512*, *dueling=False*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)

Instantiates the neural network according to arguments; network defaults stored within this method.

`forward`(*observation*, *prev_action*, *prev_reward*)

Returns the probability masses `num_atoms x num_actions` for the Q-values for each state/observation, using softmax output nonlinearity.

3.3 Recurrent DQN (R2D1)

class rlpyt.algos.dqn.r2d1.**R2D1**(*discount=0.997, batch_T=80, batch_B=64, warmup_T=40, store_rnn_state_interval=40, min_steps_learn=100000, delta_clip=None, replay_size=1000000, replay_ratio=1, target_update_interval=2500, n_step_return=5, learning_rate=0.0001, OptimCls=<sphinx.ext.autodoc.importer._MockObject object>, optim_kwargs=None, initial_optim_state_dict=None, clip_grad_norm=80.0, eps_steps=1000000, double_dqn=True, prioritized_replay=True, pri_alpha=0.6, pri_beta_init=0.9, pri_beta_final=0.9, pri_beta_steps=50000000, pri_eta=0.9, default_priority=None, input_priorities=True, input_priority_shift=None, value_scale_eps=0.001, ReplayBufferCls=None, updates_per_sync=1*)

Bases: `rlpyt.algos.dqn.dqn.DQN`

Recurrent-replay DQN with options for: Double-DQN, Dueling Architecture, n-step returns, prioritized_replay.

__init__(*discount=0.997, batch_T=80, batch_B=64, warmup_T=40, store_rnn_state_interval=40, min_steps_learn=100000, delta_clip=None, replay_size=1000000, replay_ratio=1, target_update_interval=2500, n_step_return=5, learning_rate=0.0001, OptimCls=<sphinx.ext.autodoc.importer._MockObject object>, optim_kwargs=None, initial_optim_state_dict=None, clip_grad_norm=80.0, eps_steps=1000000, double_dqn=True, prioritized_replay=True, pri_alpha=0.6, pri_beta_init=0.9, pri_beta_final=0.9, pri_beta_steps=50000000, pri_eta=0.9, default_priority=None, input_priorities=True, input_priority_shift=None, value_scale_eps=0.001, ReplayBufferCls=None, updates_per_sync=1*)

Saves input arguments.

Parameters **store_rnn_state_interval** (`int`) – store RNN state only once this many steps, to reduce memory usage; replay sequences will only begin at the steps with stored recurrent state.

Note: Typically ran with `store_rnn_state_interval` equal to the sampler's `batch_T`, 40. Then every 40 steps can be the beginning of a replay sequence, and will be guaranteed to start with a valid RNN state. Only reset the RNN state (and env) at the end of the sampler batch, so that the beginnings of episodes are trained on.

initialize_replay_buffer(*examples, batch_spec, async_=False*)

Similar to DQN but uses replay buffers which return sequences, and stores the agent's recurrent state.

optimize_agent(*itr, samples=None, sampler_itr=None*)

Similar to DQN, except allows to compute the priorities of new samples as they enter the replay buffer (input priorities), instead of only once they are used in training (important because the replay-ratio is quite low, about 1, so must avoid un-informative samples).

compute_input_priorities(*samples*)

Used when putting new samples into the replay buffer. Computes n-step TD-errors using recorded Q-values from online network and value scaling. Weights the max and the mean TD-error over each sequence to make a single priority value for that sequence.

Note: Although the original R2D2 implementation used the entire 80-step sequence to compute the input priorities, we ran R2D1 with 40 time-step sample batches, and so computed the priority for each 80-step training sequence based on one of the two 40-step halves. Algorithm argument `input_priority_shift` determines which 40-step half is used as the priority for the 80-step sequence. (Since this method might get executed by alternating memory copiers in async mode, don't carry internal state here, do all computation with only the samples available in input. Could probably reduce to one memory copier and keep state there, if needed.)

---

loss(*samples*)
Samples have leading Time and Batch dimentions [T,B,..]. Move all samples to device first, and then slice for sub-sequences. Use same init_rnn_state for agent and target; start both at same t. Warmup the RNN state first on the warmup subsequence, then train on the remaining subsequence.

Returns loss (usually use MSE, not Huber), TD-error absolute values, and new sequence-wise priorities, based on weighted sum of max and mean TD-error over the sequence.

value_scale(*x*)
Value scaling function to handle raw rewards across games (not clipped).

inv_value_scale(*z*)
Invert the value scaling.

class rlpyt.agents.dqn.r2d1_agent.**R2d1AgentBase**(*eps_init=1*, *eps_final=0.01*, *eps_final_min=None*, *eps_itr_min=50*, *eps_itr_max=1000*, *eps_eval=0.001*, **args*, ***kwargs*)
Bases: `rlpyt.agents.dqn.dqn_agent.DqnAgent`

Base agent for recurrent DQN (to add recurrent mixin).

step(*observation*, *prev_action*, *prev_reward*)
Computes Q-values for states/observations and selects actions by epsilon-greedy (no grad). Advances RNN state.

class rlpyt.agents.dqn.r2d1_agent.**R2d1Agent**(**args*, ***kwargs*)
Bases: `rlpyt.agents.base.RecurrentAgentMixin`, `rlpyt.agents.dqn.r2d1_agent.R2d1AgentBase`

R2D1 agent.

class rlpyt.models.dqn.atari_r2d1_model.**AtariR2d1Model**(*image_shape*, *output_size*, *fc_size=512*, *lstm_size=512*, *head_size=512*, *dueling=False*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

2D convolutional neural network (for multiple video frames per observation) feeding into an LSTM and MLP output for Q-value outputs for the action set.

__init__(*image_shape*, *output_size*, *fc_size=512*, *lstm_size=512*, *head_size=512*, *dueling=False*, *use_maxpool=False*, *channels=None*, *kernel_sizes=None*, *strides=None*, *paddings=None*)
Instantiates the neural network according to arguments; network defaults stored within this method.

3.4 Miscellaneous

class `rlpyt.models.dqn.dueling.`**`DuelingHeadModel`**(*input_size*, *hidden_sizes*, *output_size*, *grad_scale=0.7071067811865476*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Model component for dueling DQN. For each state Q-value, uses a scalar output for mean (bias), and vector output for relative advantages associated with each action, so the Q-values are computed as: Mean + (Advantages - mean(Advantages)). Uses a shared bias for all Advantage outputs.Gradient scaling can be applied, affecting preceding layers in the backward pass.

`forward`(*input*)

Computes Q-values through value and advantage heads; applies gradient scaling.

`advantage`(*input*)

Computes shared-bias advantages.

class `rlpyt.models.dqn.dueling.`**`DistributionalDuelingHeadModel`**(*input_size*, *hidden_sizes*, *output_size*, *n_atoms*, *grad_scale=0.7071067811865476*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Model component for Dueling Distributional (Categorical) DQN, like `DuelingHeadModel`, but handles *n_atoms* outputs for each state-action Q-value distribution.

class `rlpyt.models.dqn.atari_catdqn_model.`**`DistributionalHeadModel`**(*input_size*, *layer_sizes*, *output_size*, *n_atoms*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

An MLP head which reshapes output to [B, output_size, n_atoms].

CHAPTER 4

Q-Value Policy Gradient Implementations

This page documents algorithms, agents, and models implemented for Q-value policy gradient methods. (Much of the functionality around training and replay buffers looks similar to DQN.)

4.1 Deep Deterministc Policy Gradient (DDPG)

class rlpyt.algos.qpg.ddpg.**DDPG**(*discount=0.99*, *batch_size=64*, *min_steps_learn=10000*, *replay_size=1000000*, *replay_ratio=64*, *target_update_tau=0.01*, *target_update_interval=1*, *policy_update_interval=1*, *learning_rate=0.0001*, *q_learning_rate=0.001*, *OptimCls=<sphinx.ext.autodoc.importer._MockObject object>*, *optim_kwargs=None*, *initial_optim_state_dict=None*, *clip_grad_norm=100000000.0*, *q_target_clip=1000000.0*, *n_step_return=1*, *updates_per_sync=1*, *bootstrap_timelimit=True*, *ReplayBufferCls=None*)

Bases: *rlpyt.algos.base.RlAlgorithm*

Deep deterministic policy gradient algorithm, training from a replay buffer.

__init__(*discount=0.99*, *batch_size=64*, *min_steps_learn=10000*, *replay_size=1000000*, *replay_ratio=64*, *target_update_tau=0.01*, *target_update_interval=1*, *policy_update_interval=1*, *learning_rate=0.0001*, *q_learning_rate=0.001*, *OptimCls=<sphinx.ext.autodoc.importer._MockObject object>*, *optim_kwargs=None*, *initial_optim_state_dict=None*, *clip_grad_norm=100000000.0*, *q_target_clip=1000000.0*, *n_step_return=1*, *updates_per_sync=1*, *bootstrap_timelimit=True*, *ReplayBufferCls=None*)

Saves input arguments.

initialize(*agent*, *n_itr*, *batch_spec*, *mid_batch_reset*, *examples*, *world_size=1*, *rank=0*)

Stores input arguments and initializes replay buffer and optimizer. Use in non-async runners. Computes number of gradient updates per optimization iteration as *(replay_ratio * sampler-batch-size / training-batch_size)*.

async_initialize(*agent*, *sampler_n_itr*, *batch_spec*, *mid_batch_reset*, *examples*, *world_size=1*)
Used in async runner only; returns replay buffer allocated in shared memory, does not instantiate optimizer.

optim_initialize(*rank=0*)
Called in initilize or by async runner after forking sampler.

initialize_replay_buffer(*examples*, *batch_spec*, *async_=False*)
Allocates replay buffer using examples and with the fields in *SamplesToBuffer* namedarraytuple.

optimize_agent(*itr*, *samples=None*, *sampler_itr=None*)
Extracts the needed fields from input samples and stores them in the replay buffer. Then samples from the replay buffer to train the agent by gradient updates (with the number of updates determined by replay ratio, sampler batch size, and training batch size).

samples_to_buffer(*samples*)
Defines how to add data from sampler into the replay buffer. Called in optimize_agent() if samples are provided to that method.

mu_loss(*samples*, *valid*)
Computes the mu_loss as the Q-value at that action.

q_loss(*samples*, *valid*)
Constructs the n-step Q-learning loss using target Q. Input samples have leading batch dimension [B,..] (but not time).

class rlpyt.agents.qpg.ddpg_agent.**DdpgAgent**(*ModelCls=<class 'rlpyt.models.qpg.mlp.MuMlpModel'>*, *QModelCls=<class 'rlpyt.models.qpg.mlp.QofMuMlpModel'>*, *model_kwargs=None*, *q_model_kwargs=None*, *initial_model_state_dict=None*, *initial_q_model_state_dict=None*, *action_std=0.1*, *action_noise_clip=None*)
Bases: `rlpyt.agents.base.BaseAgent`

Agent for deep deterministic policy gradient algorithm.

__init__(*ModelCls=<class 'rlpyt.models.qpg.mlp.MuMlpModel'>*, *QModelCls=<class 'rlpyt.models.qpg.mlp.QofMuMlpModel'>*, *model_kwargs=None*, *q_model_kwargs=None*, *initial_model_state_dict=None*, *initial_q_model_state_dict=None*, *action_std=0.1*, *action_noise_clip=None*)
Saves input arguments; default network sizes saved here.

initialize(*env_spaces*, *share_memory=False*, *global_B=1*, *env_ranks=None*)
Instantiates mu and q, and target_mu and target_q models.

q(*observation*, *prev_action*, *prev_reward*, *action*)
Compute Q-value for input state/observation and action (with grad).

q_at_mu(*observation*, *prev_action*, *prev_reward*)
Compute Q-value for input state/observation, through the mu_model (with grad).

target_q_at_mu(*observation*, *prev_action*, *prev_reward*)
Compute target Q-value for input state/observation, through the target mu_model.

step(*observation*, *prev_action*, *prev_reward*)
Computes distribution parameters (mu) for state/observation, returns (gaussian) sampled action.

class rlpyt.models.qpg.mlp.**MuMlpModel**(*observation_shape*, *hidden_sizes*, *action_size*, *output_max=1*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

MLP neural net for action mean (mu) output for DDPG agent.

`__init__`(*observation_shape*, *hidden_sizes*, *action_size*, *output_max=1*)
Instantiate neural net according to inputs.

class `rlpyt.models.qpg.mlp.`**`QofMuMlpModel`**(*observation_shape*, *hidden_sizes*, *action_size*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Q portion of the model for DDPG, an MLP.

`__init__`(*observation_shape*, *hidden_sizes*, *action_size*)
Instantiate neural net according to inputs.

4.2 Twin Delayed Deep Deterministic Policy Gradient (TD3)

class `rlpyt.algos.qpg.td3.`**`TD3`**(*batch_size=100*, *replay_ratio=100*, *target_update_tau=0.005*, *target_update_interval=2*, *policy_update_interval=2*, *mu_learning_rate=0.001*, *q_learning_rate=0.001*, ***kwargs*)
Bases: *`rlpyt.algos.qpg.ddpg.DDPG`*

Twin delayed deep deterministic policy gradient algorithm.

`__init__`(*batch_size=100*, *replay_ratio=100*, *target_update_tau=0.005*, *target_update_interval=2*, *policy_update_interval=2*, *mu_learning_rate=0.001*, *q_learning_rate=0.001*, ***kwargs*)
Saved input arguments.

`q_loss`(*samples*, *valid*)
Computes MSE Q-loss for twin Q-values and min of target-Q values.

class `rlpyt.agents.qpg.td3_agent.`**`Td3Agent`**(*pretrain_std=0.5*, *target_noise_std=0.2*, *target_noise_clip=0.5*, *initial_q2_model_state_dict=None*, ***kwargs*)
Bases: *`rlpyt.agents.qpg.ddpg_agent.DdpgAgent`*

Agent for TD3 algorithm, using two Q-models and two target Q-models.

`__init__`(*pretrain_std=0.5*, *target_noise_std=0.2*, *target_noise_clip=0.5*, *initial_q2_model_state_dict=None*, ***kwargs*)
Saves input arguments.

`q`(*observation*, *prev_action*, *prev_reward*, *action*)
Compute twin Q-values for state/observation and input action (with grad).

`target_q_at_mu`(*observation*, *prev_action*, *prev_reward*)
Compute twin target Q-values for state/observation, through target mu model.

4.3 Soft Actor Critic (SAC)

class `rlpyt.algos.qpg.sac.`**SAC**(*discount=0.99*, *batch_size=256*, *min_steps_learn=10000*, *replay_size=1000000*, *replay_ratio=256*, *target_update_tau=0.005*, *target_update_interval=1*, *learning_rate=0.0003*, *fixed_alpha=None*, *OptimCls=<sphinx.ext.autodoc.importer._MockObject object>*, *optim_kwargs=None*, *initial_optim_state_dict=None*, *action_prior='uniform'*, *reward_scale=1*, *target_entropy='auto'*, *reparameterize=True*, *clip_grad_norm=1000000000.0*, *n_step_return=1*, *updates_per_sync=1*, *bootstrap_timelimit=True*, *ReplayBufferCls=None*)

Bases: *rlpyt.algos.base.RlAlgorithm*

Soft actor critic algorithm, training from a replay buffer.

__init__(*discount=0.99*, *batch_size=256*, *min_steps_learn=10000*, *replay_size=1000000*, *replay_ratio=256*, *target_update_tau=0.005*, *target_update_interval=1*, *learning_rate=0.0003*, *fixed_alpha=None*, *OptimCls=<sphinx.ext.autodoc.importer._MockObject object>*, *optim_kwargs=None*, *initial_optim_state_dict=None*, *action_prior='uniform'*, *reward_scale=1*, *target_entropy='auto'*, *reparameterize=True*, *clip_grad_norm=1000000000.0*, *n_step_return=1*, *updates_per_sync=1*, *bootstrap_timelimit=True*, *ReplayBufferCls=None*)

Save input arguments.

initialize(*agent*, *n_itr*, *batch_spec*, *mid_batch_reset*, *examples*, *world_size=1*, *rank=0*)

Stores input arguments and initializes replay buffer and optimizer. Use in non-async runners. Computes number of gradient updates per optimization iteration as *(replay_ratio * sampler-batch-size / training-batch_size)*.

optim_initialize(*rank=0*)

Called in initilize or by async runner after forking sampler.

initialize_replay_buffer(*examples*, *batch_spec*, *async_=False*)

Allocates replay buffer using examples and with the fields in *SamplesToBuffer* namedarraytuple.

optimize_agent(*itr*, *samples=None*, *sampler_itr=None*)

Extracts the needed fields from input samples and stores them in the replay buffer. Then samples from the replay buffer to train the agent by gradient updates (with the number of updates determined by replay ratio, sampler batch size, and training batch size).

samples_to_buffer(*samples*)

Defines how to add data from sampler into the replay buffer. Called in optimize_agent() if samples are provided to that method.

loss(*samples*)

Computes losses for twin Q-values against the min of twin target Q-values and an entropy term. Computes reparameterized policy loss, and loss for tuning entropy weighting, alpha.

Input samples have leading batch dimension [B,..] (but not time).

class `rlpyt.agents.qpg.sac_agent.`**SacAgent**(*ModelCls=<class 'rlpyt.models.qpg.mlp.PiMlpModel'>*, *QModelCls=<class 'rlpyt.models.qpg.mlp.QofMuMlpModel'>*, *model_kwargs=None*, *q_model_kwargs=None*, *v_model_kwargs=None*, *initial_model_state_dict=None*, *action_squash=1.0*, *pretrain_std=0.75*)

Bases: *rlpyt.agents.base.BaseAgent*

Agent for SAC algorithm, including action-squashing, using twin Q-values.

__init__(*ModelCls=<class 'rlpyt.models.qpg.mlp.PiMlpModel'>*, *QModelCls=<class 'rlpyt.models.qpg.mlp.QofMuMlpModel'>*, *model_kwargs=None*, *q_model_kwargs=None*, *v_model_kwargs=None*, *initial_model_state_dict=None*, *action_squash=1.0*, *pretrain_std=0.75*)
Saves input arguments; network defaults stored within.

q(*observation*, *prev_action*, *prev_reward*, *action*)
Compute twin Q-values for state/observation and input action (with grad).

target_q(*observation*, *prev_action*, *prev_reward*, *action*)
Compute twin target Q-values for state/observation and input action.

pi(*observation*, *prev_action*, *prev_reward*)
Compute action log-probabilities for state/observation, and sample new action (with grad). Uses special `sample_loglikelihood()` method of Gaussian distriution, which handles action squashing through this process.

class rlpyt.models.qpg.mlp.**PiMlpModel**(*observation_shape*, *hidden_sizes*, *action_size*)
Action distrubition MLP model for SAC agent.

CHAPTER 5

Runners

class rlpyt.runners.base.**BaseRunner**

Orchestrates sampler and algorithm to run the training loop. The runner should also manage logging to record agent performance during training. Different runner classes may be used depending on the overall RL procedure and the hardware configuration (e.g. multi-GPU).

train()

Entry point to conduct an entire RL training run, to be called in a launch script after instantiating all components: algo, agent, sampler.

All of the existing runners implement loops which collect minibatches of samples and provide them to the algorithm. The distinguishing features of the following classes are: a) online vs offline performance logging, b) single- vs multi-GPU training, and c) synchronous vs asynchronous operation of sampling and training. Most RL workflows should be able to use the desired class without modification.

5.1 Single-GPU Runners

class rlpyt.runners.minibatch_rl.**MinibatchRlBase**(*algo*, *agent*, *sampler*, *n_steps*, *seed=None*, *affinity=None*, *log_interval_steps=100000.0*)

Bases: *rlpyt.runners.base.BaseRunner*

Implements startup, logging, and agent checkpointing functionality, to be called in the *train()* method of the subclassed runner. Subclasses will modify/extend many of the methods here.

Parameters

- **algo** – The algorithm instance.
- **agent** – The learning agent instance.
- **sampler** – The sampler instance.
- **n_steps** (*int*) – Total number of environment steps to run in training loop.
- **seed** (*int*) – Random seed to use, if None will generate randomly.

- **`affinity`** (*`dict`*) – Hardware component assignments for sampler and algorithm.
- **`log_interval_steps`** (*`int`*) – Number of environment steps between logging to csv.

`startup`()

Sets hardware affinities, initializes the following: 1) sampler (which should initialize the agent), 2) agent device and data-parallel wrapper (if applicable), 3) algorithm, 4) logger.

`get_traj_info_kwargs`()

Pre-defines any TrajInfo attributes needed from elsewhere e.g. algorithm discount factor.

`get_n_itr`()

Determine number of train loop iterations to run. Converts logging interval units from environment steps to iterations.

`get_itr_snapshot`(*itr*)

Returns all state needed for full checkpoint/snapshot of training run, including agent parameters and optimizer parameters.

`save_itr_snapshot`(*itr*)

Calls the logger to save training checkpoint/snapshot (logger itself may or may not save, depending on mode selected).

`store_diagnostics`(*itr*, *traj_infos*, *opt_info*)

Store any diagnostic information from a training iteration that should be kept for the next logging iteration.

`log_diagnostics`(*itr*, *traj_infos=None*, *eval_time=0*, *prefix='Diagnostics/'*)

Write diagnostics (including stored ones) to csv via the logger.

`_log_infos`(*traj_infos=None*)

Writes trajectory info and optimizer info into csv via the logger. Resets stored optimizer info.

class `rlpyt.runners.minibatch_rl.`**`MinibatchRl`**(*log_traj_window=100*, ***kwargs*)

Bases: *`rlpyt.runners.minibatch_rl.MinibatchRlBase`*

Runs RL on minibatches; tracks performance online using learning trajectories.

`__init__`(*log_traj_window=100*, ***kwargs*)

Parameters **`log_traj_window`** (*`int`*) – How many trajectories to hold in deque for computing performance statistics.

`train`()

Performs startup, then loops by alternating between `sampler.obtain_samples()` and `algo.optimize_agent()`, logging diagnostics at the specified interval.

class `rlpyt.runners.minibatch_rl.`**`MinibatchRlEval`**(*algo*, *agent*, *sampler*, *n_steps*, *seed=None*, *affinity=None*, *log_interval_steps=100000.0*)

Bases: *`rlpyt.runners.minibatch_rl.MinibatchRlBase`*

Runs RL on minibatches; tracks performance offline using evaluation trajectories.

`train`()

Performs startup, evaluates the initial agent, then loops by alternating between `sampler.obtain_samples()` and `algo.optimize_agent()`. Pauses to evaluate the agent at the specified log interval.

`evaluate_agent`(*itr*)

Record offline evaluation of agent performance, by `sampler.evaluate_agent()`.

5.2 Multi-GPU Runners

class `rlpyt.runners.sync_rl.`**`SyncRlMixin`**

Mixin class to extend runner functionality to multi-GPU case. Creates a full replica of the sampler-algorithm-agent stack in a separate Python process for each GPU. Initializes `torch.distributed` to support data-parallel training of the agent. The main communication point among processes is to all-reduce gradients during backpropagation, which is handled implicitly within PyTorch. There is one agent, with the same parameters copied in all processes. No data samples are communicated in the implemented runners.

On GPU, uses the *NCCL* backend to communicate directly among GPUs. Can also be used without GPU, as multi-CPU (MPI-like, but using the *gloo* backend).

The parallelism in the sampler is independent from the parallelism here–each process will initialize its own sampler, and any one can be used (serial, cpu-parallel, gpu-parallel).

The name "Sync" refers to the fact that the sampler and algorithm still operate synchronously within each process (i.e. they alternate, running one at a time).

Note: Weak scaling is implemented for batch sizes. The batch size input argument to the sampler and to the algorithm classes are used in each process, so the actual batch sizes are *(world_size * batch_size)*. The world size is readily available from `torch.distributed`, so can change this if desired.

Note: The `affinities` input is expected to be a list, with a seprate affinity dict for each process. The number of processes is taken from the length of the affinities list.

`launch_workers``()`

As part of startup, fork a separate Python process for each additional GPU; the master process runs on the first GPU. Initialize `torch.distributed` so the `DistributedDataParallel` wrapper can work–also makes `torch.distributed` avaiable for other communication.

class `rlpyt.runners.sync_rl.`**`SyncRl`**`(`*log_traj_window=100*, ***kwargs*`)`

Bases: *`rlpyt.runners.sync_rl.SyncRlMixin`*, *`rlpyt.runners.minibatch_rl.MinibatchRl`*

Multi-process RL with online agent performance tracking. Trajectory info is collected from all processes and is included in logging.

class `rlpyt.runners.sync_rl.`**`SyncRlEval`**`(`*algo*, *agent*, *sampler*, *n_steps*, *seed=None*, *affinity=None*, *log_interval_steps=100000.0*`)`

Bases: *`rlpyt.runners.sync_rl.SyncRlMixin`*, *`rlpyt.runners.minibatch_rl.MinibatchRlEval`*

Multi-process RL with offline agent performance evaluation. Only the master process runs agent evaluation.

5.3 Asynchronous Runners

class `rlpyt.runners.async_rl.`**`AsyncRlBase`**`(`*algo*, *agent*, *sampler*, *n_steps*, *affinity*, *seed=None*, *log_interval_steps=100000.0*`)`

Bases: *`rlpyt.runners.base.BaseRunner`*

Runs sampling and optimization asynchronously in separate Python processes. May be useful to achieve higher hardware utilization, e.g. CPUs fully busy simulating the environment while GPU fully busy training the agent (there's no reason to use this CPU-only). This setup is significantly more complicated than the synchronous (single- or multi-GPU) runners, requires use of the asynchronous sampler, and may require special methods in the algorithm.

Further parallelization within the sampler and optimizer are independent. The asynchronous sampler can be serial, cpu-parallel, gpu-parallel, or multi-gpu-parallel. The optimizer can be single- or multi-gpu.

The algorithm must initialize a replay buffer on OS shared memory. The asynchronous sampler will allocate minibatch buffers on OS shared memory, and yet another Python process is run to copy the completed minibatches over to the algorithm's replay buffer. While that memory copy is underway, the sampler immediately begins gathering the next minibatch.

Care should be taken to balance the rate at which the algorithm runs against the rate of the sampler, as this can affect learning performance. In the existing implementations, the sampler runs at full speed, and the algorithm may be throttled not to exceed the specified relative rate. This is set by the algorithm's `replay_ratio`, which becomes the upper bound on the amount of training samples used in ratio with the amount of samples generated. (In synchronous mode, the replay ratio is enforced more precisely by running a fixed batch size and number of updates per iteration.)

The master process runs the (first) training GPU and performs all logging.

Within the optimizer, one agent exists. If multi-GPU, the same parameter values are copied across all GPUs, and PyTorch's DistributedDataParallel is used to all-reduce gradients (as in the synchronous multi-GPU runners). Within the sampler, one agent exists. If new agent parameters are available from the optimizer between sampler minibatches, then those values are copied into the sampler before gathering the next minibatch.

Note: The `affinity` argument should be a structure with `sampler` and `optimizer` attributes holding the respective hardware allocations. Optimizer and sampler parallelization is determined from this.

`train`()
Run the optimizer in a loop. Check whether enough new samples have been generated, and throttle down if necessary at each iteration. Log at an interval in the number of sampler iterations, not optimizer iterations.

`startup`()
Calls `sampler.async_initialize()` to get a double buffer for minibatches, followed by `algo.async_initialize()` to get a replay buffer on shared memory, then launches all workers (sampler, optimizer, memory copier).

`optim_startup`()
Sets the hardware affinity, moves the agent's model parameters onto device and initialize data-parallel agent, if applicable. Computes optimizer throttling settings.

`build_ctrl`(*world_size*)
Builds several parallel communication mechanisms for controlling the workflow across processes.

`launch_optimizer_workers`(*n_itr*)
If multi-GPU optimization, launches an optimizer worker for each GPU and initializes `torch.distributed.`

`launch_memcpy`(*sample_buffers*, *replay_buffer*)
Fork a Python process for each of the sampler double buffers. (It may be overkill to use two separate processes here, may be able to simplify to one and still get good performance.)

class `rlpyt.runners.async_rl.`**`AsyncRl`**(**args*, *log_traj_window=100*, ***kwargs*)
Bases: *`rlpyt.runners.async_rl.AsyncRlBase`*

Asynchronous RL with online agent performance tracking.

class `rlpyt.runners.async_rl.`**`AsyncRlEval`**(*algo*, *agent*, *sampler*, *n_steps*, *affinity*, *seed=None*, *log_interval_steps=100000.0*)
Bases: *`rlpyt.runners.async_rl.AsyncRlBase`*

Asynchronous RL with offline agent performance evaluation.

5.3.1 Asynchronous Worker Processes

`rlpyt.runners.async_rl.`**`run_async_sampler`**`(`*sampler*, *affinity*, *ctrl*, *traj_infos_queue*, *n_itr*`)`

Target function for the process which will run the sampler, in the case of online performance logging. Toggles the sampler's double-buffer for each iteration, waits for the memory copier to finish before writing into that buffer, and signals the memory copier when the sampler is done writing a minibatch.

`rlpyt.runners.async_rl.`**`run_async_sampler_eval`**`(`*sampler*, *affinity*, *ctrl*, *traj_infos_queue*, *n_itr*, *eval_itrs*`)`

Target function running the sampler with offline performance evaluation.

`rlpyt.runners.async_rl.`**`memory_copier`**`(`*sample_buffer*, *samples_to_buffer*, *replay_buffer*, *ctrl*`)`

Target function for the process which will copy the sampler's minibatch buffer into the algorithm's main replay buffer.

Parameters

- **`sample_buffer`** – The (single) minibatch buffer from the sampler, on shared memory.
- **`samples_to_buffer`** – A function/method from the algorithm to process samples from the minibatch buffer into the replay buffer (e.g. select which fields, compute some prioritization).
- **`replay_buffer`** – Algorithm's main replay buffer, on shared memory.
- **`ctrl`** – Structure for communicating when the minibatch is ready to copy/done copying.

Warning: Although this function may use the algorithm's `samples_to_buffer()` method, here it is running in a separate process, so will not be aware of changes in the algorithm from the optimizer process. Furthermore, it may not be able to store state across iterations–in the implemented setup, two separate memory copier processes are used, so each one only sees every other minibatch. (Could easily change to single copier if desired, and probably without peformance loss.)

CHAPTER 6

Samplers

Several sampler classes are implemented for different parallelization schemes, with multiple environment instances running on CPU resources and agent forward passes happening on either CPU or GPU. The implemented samplers execute a fixed number of time-steps at each call to `obtain_samples()`, which returns a batch of data with leading dimensions `[batch_T, batch_B]`.

Something about choosing which sampler based on parallel needs/availability, and different for each case, but try them out.

class `rlpyt.samplers.base.`**`BaseSampler`**(*EnvCls*, *env_kwargs*, *batch_T*, *batch_B*, *CollectorCls*, *max_decorrelation_steps=100*, *TrajInfoCls=<class 'rlpyt.samplers.collections.TrajInfo'>*, *eval_n_envs=0*, *eval_CollectorCls=None*, *eval_env_kwargs=None*, *eval_max_steps=None*, *eval_max_trajectories=None*)

Class which interfaces with the Runner, in master process only.

Parameters

- **`EnvCls`** – class (or factory function) callable to instantiate an environment object
- **`env_kwargs`** (*`dict`*) – keyword arguments passed to `EnvCls()` to instantiate
- **`batch_T`** (*`int`*) – number of time-steps per sample batch
- **`batch_B`** (*`int`*) – number of environment instances to run (in parallel), becomes second batch dimension
- **`CollectorCls`** – callable to instantiate the collector, which manages agent-environment interaction loop
- **`max_decorrelation_steps`** (*`int`*) – if taking random number of steps before start of training, to decorrelate batch states
- **`TrajInfoCls`** – callable to instantiate object for tracking trajectory-wise info
- **`eval_n_envs`** (*`int`*) – number of environment instances for agent evaluation (0 for no separate evaluation)

- **eval_CollectorCls** – callable to instantiate evaluation collector
- **eval_env_kwargs** – keyword arguments passed to `EnvCls()` for eval envs
- **eval_max_steps** – max total number of steps (time * n_envs) per evaluation call
- **eval_max_trajectories** – optional earlier cutoff for evaluation phase

initialize(**args*, ***kwargs*)
Should instantiate all components, including setup of parallel process if applicable.

obtain_samples(*itr*)
Execute agent-environment interactions and return data batch.

evaluate_agent(*itr*)
Run offline agent evaluation, if applicable.

6.1 Serial Sampler

class rlpyt.samplers.serial.sampler.**SerialSampler**(**args*, *CollectorCls=<class 'rlpyt.samplers.parallel.cpu.collectors.CpuResetCollector'>*, *eval_CollectorCls=<class 'rlpyt.samplers.serial.collectors.SerialEvalCollector'>*, ***kwargs*)

Bases: `rlpyt.samplers.base.BaseSampler`

The simplest sampler; no parallelism, everything occurs in same, master Python process. This can be easier for debugging (e.g. can use `breakpoint()` in master process) and might be fast enough for experiment purposes. Should be used with collectors which generate the agent's actions internally, i.e. CPU-based collectors but not GPU-based ones.

initialize(*agent*, *affinity=None*, *seed=None*, *bootstrap_value=False*, *traj_info_kwargs=None*, *rank=0*, *world_size=1*)
Store the input arguments. Instantiate the specified number of environment instances (`batch_B`). Initialize the agent, and pre-allocate a memory buffer to hold the samples collected in each batch. Applies `traj_info_kwargs` settings to the *TrajInfoCls* by direct class attribute assignment. Instantiates the Collector and, if applicable, the evaluation Collector.

Returns a structure of inidividual examples for data fields such as *observation*, *action*, etc, which can be used to allocate a replay buffer.

obtain_samples(*itr*)
Call the collector to execute a batch of agent-environment interactions. Return data in torch tensors, and a list of trajectory-info objects from episodes which ended.

evaluate_agent(*itr*)
Call the evaluation collector to execute agent-environment interactions.

6.2 Parallel Samplers

class rlpyt.samplers.parallel.base.**ParallelSamplerBase**(*EnvCls*, *env_kwargs*, *batch_T*, *batch_B*, *CollectorCls*, *max_decorrelation_steps=100*, *TrajInfoCls=<class 'rlpyt.samplers.collections.TrajInfo'>*, *eval_n_envs=0*, *eval_CollectorCls=None*, *eval_env_kwargs=None*, *eval_max_steps=None*, *eval_max_trajectories=None*)

Bases: `rlpyt.samplers.base.BaseSampler`

Base class for samplers which use worker processes to run environment steps in parallel, across CPU resources.

initialize(*agent*, *affinity*, *seed*, *bootstrap_value=False*, *traj_info_kwargs=None*, *world_size=1*, *rank=0*, *worker_process=None*)

Creates an example instance of the environment for agent initialization (which may differ by sub-class) and to pre-allocate batch buffers, then deletes the environment instance. Batch buffers are allocated on shared memory, so that worker processes can read/write directly to them.

Computes the number of parallel processes based on the `affinity` argument. Forks worker processes, which instantiate their own environment and collector objects. Waits for the worker process to complete all initialization (such as decorrelating environment states) before returning. Barriers and other parallel indicators are constructed to manage worker processes.

Warning: If doing offline agent evaluation, will use at least one evaluation environment instance per parallel worker, which might increase the total number of evaluation instances over what was requested. This may result in bias towards shorter episodes if the episode length is variable, and if the max number of evalution steps divided over the number of eval environments (*eval_max_steps / actual_eval_n_envs*), is not large relative to the max episode length.

obtain_samples(*itr*)

Signal worker processes to collect samples, and wait until they finish. Workers will write directly to the pre-allocated samples buffer, which this method returns. Trajectory-info objects from completed trajectories are retrieved from workers through a parallel queue object and are also returned.

evaluate_agent(*itr*)

Signal worker processes to perform agent evaluation. If a max number of evaluation trajectories was specified, keep watch over the number of trajectories finished and signal an early end if the limit is reached. Return a list of trajectory-info objects from the completed episodes.

6.2.1 CPU-Agent

class rlpyt.samplers.parallel.cpu.sampler.**CpuSampler**(**args*, *CollectorCls=<class 'rlpyt.samplers.parallel.cpu.collectors.CpuResetCollecto*
eval_CollectorCls=<class 'rlpyt.samplers.parallel.cpu.collectors.CpuEvalCollecto
***kwargs*)

Parallel sampler for using the CPU resource of each worker to compute agent forward passes; for use with CPU-based collectors.

obtain_samples(*itr*)
First, have the agent sync shared memory; in case training uses a GPU, the agent needs to copy its (new) GPU model parameters to the shared-memory CPU model which all the workers use. Then call super class's method.

evaluate_agent(*itr*)
Like in `obtain_samples()`, first sync agent shared memory.

6.2.2 GPU-Agent

class rlpyt.samplers.parallel.gpu.sampler.**GpuSamplerBase**(**args, CollectorCls=<class 'rlpyt.samplers.parallel.gpu.collectors.GpuResetC eval_CollectorCls=<class 'rlpyt.samplers.parallel.gpu.collectors.GpuEvalC **kwargs*)

Bases: *rlpyt.samplers.parallel.base.ParallelSamplerBase*

Base class for parallel samplers which use worker processes to execute environment steps on CPU resources but the master process to execute agent forward passes for action selection, presumably on GPU. Use GPU-based collecter classes.

In addition to the usual batch buffer for data samples, allocates a step buffer over shared memory, which is used for communication with workers. The step buffer includes *observations*, which the workers write and the master reads, and *actions*, which the master write and the workers read. (The step buffer has leading dimension [*batch_B*], for the number of parallel environments, and each worker gets its own slice along that dimension.) The step buffer object is held in both numpy array and torch tensor forms over the same memory; e.g. workers write to the numpy array form, and the agent is able to read the torch tensor form.

(Possibly more information about how the stepping works, but write in action-server or smwr like that.)

obtain_samples(*itr*)
Signals worker to begin environment step execution loop, and drops into `serve_actions()` method to provide actions to workers based on the new observations at each step.

evaluate_agent(*itr*)
Signals workers to begin agent evaluation loop, and drops into `serve_actions_evaluation()` to provide actions to workers at each step.

_agent_init(*agent, env, global_B=1, env_ranks=None*)
Initializes the agent, having it *not* share memory, because all agent functions (training and sampling) happen in the master process, presumably on GPU.

class rlpyt.samplers.parallel.gpu.action_server.**ActionServer**
Mixin class with methods for serving actions to worker processes which execute environment steps.

serve_actions(*itr*)
Called in master process during `obtain_samples()`.

Performs agent action- selection loop in concert with workers executing environment steps. Uses shared memory buffers to communicate agent/environment data at each time step. Uses semaphores for synchronization: one per worker to acquire when they finish writing the next step of observations, one per worker to release when master has written the next actions. Resets the agent one B-index at a time when the corresponding environment resets (i.e. agent's recurrent state, with leading dimension `batch_B`).

Also communicates `agent_info` to workers, which are responsible for recording all data into the batch buffer.

If requested, collects additional agent value estimation of final observation for bootstrapping (the one thing written to the batch buffer here).

> **Warning:** If trying to modify, must be careful to keep correct logic of the semaphores, to make sure they drain properly. If a semaphore ends up with an extra release, synchronization can be lost silently, leading to wrong and confusing results.

serve_actions_evaluation(*itr*)

Similar to `serve_actions()`. If a maximum number of eval trajectories was specified, keeps track of the number completed and terminates evaluation if the max is reached. Returns a list of completed trajectory-info objects.

class `rlpyt.samplers.parallel.gpu.sampler.`**GpuSampler**(**args*, *CollectorCls=<class 'rlpyt.samplers.parallel.gpu.collectors.GpuResetCollect* *eval_CollectorCls=<class 'rlpyt.samplers.parallel.gpu.collectors.GpuEvalCollecto* ***kwargs*)

Bases: *rlpyt.samplers.parallel.gpu.action_server.ActionServer*, *rlpyt.samplers.parallel.gpu.sampler.GpuSamplerBase*

6.2.3 GPU-Agent, Alternating Workers

class `rlpyt.samplers.parallel.gpu.alternating_sampler.`**AlternatingSamplerBase**(**args*, ***kwargs*)

Twice the standard number of worker processes are forked, and they may share CPU resources in pairs. Environment instances are divided evenly among the two sets. While one set of workers steps their environments, the action-server process computes the actions for the other set of workers, which are paused until their new actions are ready (this pause happens in the GpuSampler). The two sets of workers alternate in this procedure, keeping the CPU maximally busy. The intention is to hide the time to compute actions from the critical path of execution, which can provide up to a 2x speed boost in theory, if the environment-step time and agent-step time were othewise equal.

If the latency in computing and returning the agent's action is longer than environment stepping, then this alternation might not be faster, because it calls agent action selection twice as many times.

initialize(*agent*, **args*, ***kwargs*)

Like the super class's `initialize()`, but creates additional set of synchronization and communication objects for the alternate workers.

class `rlpyt.samplers.parallel.gpu.action_server.`**AlternatingActionServer**

Mixin class for serving actions in the alternating GPU sampler. The synchronization format in this class allows the two worker groups to execute partially simultaneously; workers wait to step for their new action to be ready but do not wait for the other set of workers to be done stepping.

class `rlpyt.samplers.parallel.gpu.action_server.`**NoOverlapAlternatingActionServer**

Mixin class for serving actions in the alternating GPU sampler. The synchronization format in this class disallows the two worker groups from executing simultaneously; workers wait to step for their new action to be ready and also wait for the other set of workers to be done stepping.

> **Warning:** Not sure the logic around semaphores is correct for all cases at the end of `serve_actions_evaluation()` (see TODO comment).

class `rlpyt.samplers.parallel.gpu.alternating_sampler.`**`AlternatingSampler`**(**args*, ***kwargs*)

Bases: *`rlpyt.samplers.parallel.gpu.action_server.AlternatingActionServer`*, *`rlpyt.samplers.parallel.gpu.alternating_sampler.AlternatingSamplerBase`*

class `rlpyt.samplers.parallel.gpu.alternating_sampler.`**`NoOverlapAlternatingSampler`**(**args*, ***kwargs*)

Bases: *`rlpyt.samplers.parallel.gpu.action_server.NoOverlapAlternatingActionServer`*, *`rlpyt.samplers.parallel.gpu.alternating_sampler.AlternatingSamplerBase`*

6.3 Parallel Sampler Worker

The same function is used as the target for forking worker processes in all parallel samplers.

`rlpyt.samplers.parallel.worker.`**`sampling_process`**(*common_kwargs*, *worker_kwargs*)

Target function used for forking parallel worker processes in the samplers. After `initialize_worker()`, it creates the specified number of environment instances and gives them to the collector when instantiating it. It then calls collector startup methods for environments and agent. If applicable, instantiates evaluation environment instances and evaluation collector.

Then enters infinite loop, waiting for signals from master to collect training samples or else run evaluation, until signaled to exit.

`rlpyt.samplers.parallel.worker.`**`initialize_worker`**(*rank*, *seed=None*, *cpu=None*, *torch_threads=None*)

Assign CPU affinity, set random seed, set torch_threads if needed to prevent MKL deadlock.

CHAPTER 7

Asynchronous Samplers

Separate sampler classes are needed for asynchronous sampling-optimization mode, and they closely match the options for the other samplers. In asynchronous mode, the sampler will run in a separate process forked from the main (training) process. Parallel asynchronous samplers will fork additional processes.

7.1 Base Components

class `rlpyt.samplers.async_.base.`**`AsyncSamplerMixin`**

Mixin class defining master runner initialization method for all asynchronous samplers.

`async_initialize`(*agent*, *bootstrap_value=False*, *traj_info_kwargs=None*, *seed=None*)

Instantiate an example environment and use it to initialize the agent (on shared memory). Pre-allocate a double-buffer for sample batches, and return that buffer along with example data (e.g. *observation*, *action*, etc.)

class `rlpyt.samplers.async_.base.`**`AsyncParallelSamplerMixin`**

Bases: *`rlpyt.samplers.async_.base.AsyncSamplerMixin`*

Mixin class defining methods for the asynchronous sampler main process (which is forked from the overall master process).

`obtain_samples`(*itr*, *db_idx*)

Communicates to workers which batch buffer to use, and signals them to start collection. Waits until workers finish, and then retrieves completed trajectory-info objects from the workers and returns them in a list.

7.2 Serial

class rlpyt.samplers.async_.serial_sampler.**AsyncSerialSampler**(**args*, *CollectorCls=<class 'rlpyt.samplers.async_.collectors.DbCpuR*, *eval_CollectorCls=<class 'rlpyt.samplers.serial.collectors.SerialEva*, ***kwargs*)

Bases: *rlpyt.samplers.async_.base.AsyncSamplerMixin*, *rlpyt.samplers.base.BaseSampler*

Sampler which runs asynchronously in a python process forked from the master (training) process, but with no further parallelism.

initialize(*affinity*)
Initialization inside the main sampler process. Sets process hardware affinities, creates specified number of environment instances and instantiates the collector with them. If applicable, does the same for evaluation environment instances. Moves the agent to device (could be GPU), and calls on `agent.async_cpu()` initialization. Starts up collector.

obtain_samples(*itr*, *db_idx*)
First calls the agent to retrieve new parameter values from the training process's agent. Then passes the double-buffer index to the collector and collects training sample batch. Returns list of completed trajectory-info objects.

evaluate_agent(*itr*)
First calls the agent to retrieve new parameter values from the training process's agent.

7.3 CPU-Agent

class rlpyt.samplers.async_.cpu_sampler.**AsyncCpuSampler**(**args*, *CollectorCls=<class 'rlpyt.samplers.async_.collectors.DbCpuResetColle*, *eval_CollectorCls=<class 'rlpyt.samplers.parallel.cpu.collectors.CpuEvalCol*, ***kwargs*)

Bases: *rlpyt.samplers.async_.base.AsyncParallelSamplerMixin*, *rlpyt.samplers.parallel.base.ParallelSamplerBase*

Parallel sampler for agent action-selection on CPU, to use in asynchronous runner. The master (training) process will have forked the main sampler process, which here will fork sampler workers from itself, and otherwise will run similarly to the `CpuSampler`.

initialize(*affinity*)
Runs inside the main sampler process. Sets process hardware affinity and calls the `agent.async_cpu()` initialization. Then proceeds with usual parallel sampler initialization.

obtain_samples(*itr*, *db_idx*)
Calls the agent to retrieve new parameter values from the training process, then proceeds with base async parallel method.

evaluate_agent(*itr*)
Calls the agent to retrieve new parameter values from the training process, then proceeds with base async parallel method.

7.4 GPU-Agent

7.4.1 Main Class

class rlpyt.samplers.async_.gpu_sampler.**AsyncGpuSampler**(**args, CollectorCls=<class 'rlpyt.samplers.async_.collectors.DbGpuResetColle eval_CollectorCls=<class 'rlpyt.samplers.parallel.gpu.collectors.GpuEvalCol **kwargs*)

Bases: *rlpyt.samplers.async_.action_server.AsyncActionServer*, *rlpyt.samplers.async_.gpu_sampler.AsyncGpuSamplerBase*

7.4.2 Component Definitions

class rlpyt.samplers.async_.gpu_sampler.**AsyncGpuSamplerBase**(**args, CollectorCls=<class 'rlpyt.samplers.async_.collectors.DbGpuRese eval_CollectorCls=<class 'rlpyt.samplers.parallel.gpu.collectors.GpuEv **kwargs*)

Bases: *rlpyt.samplers.async_.base.AsyncParallelSamplerMixin*, *rlpyt.samplers.parallel.base.ParallelSamplerBase*

Main definitions for asynchronous parallel sampler using GPU(s) for action selection. The main sampler process (forked from the overall master), forks action-server processes, one per GPU to be used, and the action-server process(es) fork their own parallel CPU workers. This same sampler object is used in the main sampler process and in the action server process(es), but for different methods, labeled by comments in the code (easier way to pass arguments along).

initialize(*affinity*)

Initialization inside the main sampler process. Builds one level of parallel synchronization objects, and forks action-server processes, one per GPU to be used.

action_server_process(*rank, env_ranks, double_buffer_slice, affinity, seed, n_envs_list*)

Target method used for forking action-server process(es) from the main sampler process. By inheriting the sampler object from the sampler process, can more easily pass args to the environment worker processes, which are forked from here.

Assigns hardware affinity, and then forks parallel worker processes and moves agent model to device. Then enters infinite loop: waits for signals from main sampler process to collect training samples or perform evaluation, and then serves actions during collection. At every loop, calls agent to retrieve new parameter values from the training process, which are communicated through shared CPU memory.

class rlpyt.samplers.async_.action_server.**AsyncActionServer**

Bases: *rlpyt.samplers.parallel.gpu.action_server.ActionServer*

serve_actions_evaluation(*itr*)

Similar to normal action-server, but with different signaling logic for ending evaluation early; receive signal from main sampler process and pass it along to my workers.

7.5 GPU-Agent, Alternating Workers

7.5.1 Main Classes

class rlpyt.samplers.async_.alternating_sampler.**AsyncAlternatingSampler**(**args, **kwargs*)

Bases: *rlpyt.samplers.async_.action_server.AsyncAlternatingActionServer*, *rlpyt.samplers.async_.alternating_sampler.AsyncAlternatingSamplerBase*

class rlpyt.samplers.async_.alternating_sampler.**AsyncNoOverlapAlternatingSampler**(**args, **kwargs*)

Bases: *rlpyt.samplers.async_.action_server.AsyncNoOverlapAlternatingActionServer*, *rlpyt.samplers.async_.alternating_sampler.AsyncAlternatingSamplerBase*

7.5.2 Component Definitions

class rlpyt.samplers.async_.alternating_sampler.**AsyncAlternatingSamplerBase**(**args, **kwargs*)

Bases: *rlpyt.samplers.async_.gpu_sampler.AsyncGpuSamplerBase*

Defines several methods to extend the asynchronous GPU sampler to use two alternating sets of environment workers.

class rlpyt.samplers.async_.action_server.**AsyncAlternatingActionServer**

Bases: *rlpyt.samplers.parallel.gpu.action_server.AlternatingActionServer*

serve_actions_evaluation(*itr*)

Similar to normal action-server, but with different signaling logic for ending evaluation early; receive signal from main sampler process and pass it along to my workers.

class rlpyt.samplers.async_.action_server.**AsyncNoOverlapAlternatingActionServer**

Bases: *rlpyt.samplers.parallel.gpu.action_server.NoOverlapAlternatingActionServer*

Not tested, possibly faulty corner cases for synchronization.

serve_actions_evaluation(*itr*)

Similar to normal action-server, but with different signaling logic for ending evaluation early; receive signal from main sampler process and pass it along to my workers.

CHAPTER 8

Collectors

Collectors run the environment-agent interaction loop and record sampled data to the batch buffer. The serial sampler runs one collector, and in parallel samplers, each worker process runs one collector. Different collectors are needed for CPU-agent vs GPU-agent samplers.

In general, collectors will execute a for loop over time steps, and and inner for loop over environments, and step each environment one at a time. At every step, all information (e.g. *observation*, *env_info*, etc.) is recorded to its place in the pre-allocated batch buffer. All information is also fed to the trajectory-info object for each environment, for tracking trajectory-wise measures.

Evaluation collectors only record trajectory-wise results.

8.1 Training Collectors

8.1.1 Base Components

class `rlpyt.samplers.collectors.`**`BaseCollector`**(*rank*, *envs*, *samples_np*, *batch_T*, *TrajInfoCls*, *agent=None*, *sync=None*, *step_buffer_np=None*, *global_B=1*, *env_ranks=None*)

Class that steps environments, possibly in worker process.

`start_envs`()

e.g. calls reset() on every env.

`start_agent`()

In CPU-collectors, call `agent.collector_initialize()` e.g. to set up vector epsilon-greedy, and reset the agent.

`collect_batch`(*agent_inputs*, *traj_infos*)

Main data collection loop.

`reset_if_needed`(*agent_inputs*)

Reset agent and or env as needed, if doing between batches.

class rlpyt.samplers.collectors.**DecorrelatingStartCollector**(*rank*, *envs*, *samples_np*, *batch_T*, *TrajInfoCls*, *agent=None*, *sync=None*, *step_buffer_np=None*, *global_B=1*, *env_ranks=None*)

Bases: *rlpyt.samplers.collectors.BaseCollector*

Collector which can step all environments through a random number of random actions during startup, to decorrelate the states in training batches.

start_envs(*max_decorrelation_steps=0*)

Calls `reset()` on every environment instance, then steps each one through a random number of random actions, and returns the resulting agent_inputs buffer (*observation*, *prev_action*, *prev_reward*).

8.1.2 CPU-Agent Collectors

class rlpyt.samplers.parallel.cpu.collectors.**CpuResetCollector**(*rank*, *envs*, *samples_np*, *batch_T*, *TrajInfoCls*, *agent=None*, *sync=None*, *step_buffer_np=None*, *global_B=1*, *env_ranks=None*)

Bases: *rlpyt.samplers.collectors.DecorrelatingStartCollector*

Collector which executes `agent.step()` in the sampling loop (i.e. use in CPU or serial samplers.)

It immediately resets any environment which finishes an episode. This is typically indicated by the environment returning `done=True`. But this collector defers to the `done` signal only after looking for `env_info["traj_done"]`, so that RL episodes can end without a call to `env_reset()` (e.g. used for episodic lives in the Atari env). The agent gets reset based solely on `done`.

class rlpyt.samplers.parallel.cpu.collectors.**CpuWaitResetCollector**(**args*, ***kwargs*)

Bases: *rlpyt.samplers.collectors.DecorrelatingStartCollector*

Collector which executes `agent.step()` in the sampling loop.

It waits to reset any environments with completed episodes until after the end of collecting the batch, i.e. the `done` environment is bypassed in remaining timesteps, and zeros are recorded into the batch buffer.

Waiting to reset can be beneficial for two reasons. One is for training recurrent agents; PyTorch's built-in LSTM cannot reset in the middle of a training sequence, so any samples in a batch after a reset would be ignored and the beginning of new episodes would be missed in training. The other reason is if the environment's reset function is very slow compared to its step function; it can be faster overall to leave invalid samples after a reset, and perform the environment resets in the workers while the master process is training the agent (this was true for massively parallelized Atari).

8.1.3 GPU-Agent Collectors

class rlpyt.samplers.parallel.gpu.collectors.**GpuResetCollector**(*rank*, *envs*, *samples_np*, *batch_T*, *TrajInfoCls*, *agent=None*, *sync=None*, *step_buffer_np=None*, *global_B=1*, *env_ranks=None*)

Bases: *rlpyt.samplers.collectors.DecorrelatingStartCollector*

Collector which communicates observations to an action-server, which in turn provides the agent's actions (i.e. use in GPU samplers).

Environment reset logic is the same as in `CpuResetCollector`.

class rlpyt.samplers.parallel.gpu.collectors.**GpuWaitResetCollector**(**args*, ***kwargs*)

Bases: *rlpyt.samplers.collectors.DecorrelatingStartCollector*

Collector which communicates observations to an action-server, which in turn provides the agent's actions (i.e. use in GPU samplers).

Environment reset logic is the same as in `CpuWaitResetCollector`.

8.2 Evaluation Collectors

class rlpyt.samplers.collectors.**BaseEvalCollector**(*rank*, *envs*, *TrajInfoCls*, *traj_infos_queue*, *max_T*, *agent=None*, *sync=None*, *step_buffer_np=None*)

Collectors for offline agent evalution; not to record intermediate samples.

collect_evaluation()

Run agent evaluation in environment and return completed trajectory infos.

class rlpyt.samplers.parallel.cpu.collectors.**CpuEvalCollector**(*rank*, *envs*, *TrajInfoCls*, *traj_infos_queue*, *max_T*, *agent=None*, *sync=None*, *step_buffer_np=None*)

Bases: *rlpyt.samplers.collectors.BaseEvalCollector*

Offline agent evaluation collector which calls `agent.step()` in sampling loop. Immediately resets any environment which finishes a trajectory. Stops when the max time-steps have been reached, or when signaled by the master process (i.e. if enough trajectories have completed).

class rlpyt.samplers.parallel.gpu.collectors.**GpuEvalCollector**(*rank*, *envs*, *TrajInfoCls*, *traj_infos_queue*, *max_T*, *agent=None*, *sync=None*, *step_buffer_np=None*)

Bases: *rlpyt.samplers.collectors.BaseEvalCollector*

Offline agent evaluation collector which communicates observations to an action-server, which in turn provides the agent's actions.

CHAPTER 9

Distributions

Distributions are used to select randomized actions during sampling, and for some algorithms to compute likelihood and related values for training. Typically, the distribution is owned by the agent. This page documents the implemented distributions and some methods–see the code for details.

class `rlpyt.distributions.base.`**`Distribution`**

Base distribution class. Not all subclasses will impelement all methods.

`sample`(*dist_info*)

Generate random sample(s) from distribution informations.

`kl`(*old_dist_info*, *new_dist_info*)

Compute the KL divergence of two distributions at each datum; should maintain leading dimensions (e.g. [T,B]).

`mean_kl`(*old_dist_info*, *new_dist_info*, *valid*)

Compute the mean KL divergence over a data batch, possible ignoring data marked as invalid.

`log_likelihood`(*x*, *dist_info*)

Compute log-likelihood of samples `x` at distributions described in `dist_info` (i.e. can have same leading dimensions [T, B]).

`likelihood_ratio`(*x*, *old_dist_info*, *new_dist_info*)

Compute likelihood ratio of samples `x` at new distributions over old distributions (usually `new_dist_info` is variable for differentiation); should maintain leading dimensions.

`entropy`(*dist_info*)

Compute entropy of distributions contained in `dist_info`; should maintain any leading dimensions.

`perplexity`(*dist_info*)

Exponential of the entropy, maybe useful for logging.

`mean_entropy`(*dist_info*, *valid=None*)

In case some sophisticated mean is needed (e.g. internally ignoring select parts of action space), can override.

`mean_perplexity`(*dist_info*, *valid=None*)

Exponential of the entropy, maybe useful for logging.

class `rlpyt.distributions.discrete.`**`DiscreteMixin`**(*dim*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*, *one-hot_dtype=<sphinx.ext.autodoc.importer._MockObject object>*)

Conversions to and from one-hot.

`to_onehot`(*indexes*, *dtype=None*)
Convert from integer indexes to one-hot, preserving leading dimensions.

`from_onehot`(*onehot*, *dtype=None*)
Convert from one-hot to integer indexes, preserving leading dimensions.

class `rlpyt.distributions.categorical.`**`Categorical`**(*dim*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*, *one-hot_dtype=<sphinx.ext.autodoc.importer._MockObject object>*)

Bases: *`rlpyt.distributions.discrete.DiscreteMixin`*, *`rlpyt.distributions.base.Distribution`*

Multinomial distribution over a discrete domain.

`sample`(*dist_info*)
Sample from `torch.multiomial` over trailing dimension of `dist_info.prob`.

class `rlpyt.distributions.epsilon_greedy.`**`EpsilonGreedy`**(*epsilon=1*, ***kwargs*)

Bases: *`rlpyt.distributions.discrete.DiscreteMixin`*, *`rlpyt.distributions.base.Distribution`*

For epsilon-greedy exploration from state-action Q-values.

`sample`(*q*)
Input can be shaped [T,B,Q] or [B,Q], and vector epsilon of length B will apply across the Batch dimension (same epsilon for all T).

`set_epsilon`(*epsilon*)
Assign value for epsilon (can be vector).

class `rlpyt.distributions.epsilon_greedy.`**`CategoricalEpsilonGreedy`**(*z=None*, ***kwargs*)

Bases: *`rlpyt.distributions.epsilon_greedy.EpsilonGreedy`*

For epsilon-greedy exploration from distributional (categorical) representation of state-action Q-values.

`sample`(*p*, *z=None*)
Input p to be shaped [T,B,A,P] or [B,A,P], A: number of actions, P: number of atoms. Optional input z is domain of atom-values, shaped [P]. Vector epsilon of lenght B will apply across Batch dimension.

`set_z`(*z*)
Assign vector of bin locations, distributional domain.

class `rlpyt.distributions.gaussian.`**`Gaussian`**(*dim*, *std=None*, *clip=None*, *noise_clip=None*, *min_std=None*, *max_std=None*, *squash=None*)

Multivariate Gaussian with independent variables (diagonal covariance). Standard deviation can be provided, as scalar or value per dimension, or it will be drawn from the dist_info (possibly learnable), where it is expected to have a value per each dimension. Noise clipping or sample clipping optional during sampling, but not accounted for in formulas (e.g. entropy). Clipping of standard deviation optional and accounted in formulas. Squashing of samples to squash * tanh(sample) is optional and accounted for in log_likelihood formula but not entropy.

`entropy`(*dist_info*)
Uses `self.std` unless that is None, then will get log_std from dist_info. Not implemented for squashing.

log_likelihood(*x*, *dist_info*)
Uses `self.std` unless that is None, then uses log_std from dist_info. When squashing: instead of numerically risky arctanh, assume param 'x' is pre-squash action, see `sample_loglikelihood()` below.

sample_loglikelihood(*dist_info*)
Special method for use with SAC algorithm, which returns a new sampled action and its log-likelihood for training use. Temporarily turns OFF squashing, so that log_likelihood can be computed on non-squashed sample, and then restores squashing and applies it to the sample before output.

sample(*dist_info*)
Generate random samples using `torch.normal`, from `dist_info.mean`. Uses `self.std` unless it is `None`, then uses `dist_info.log_std`.

set_clip(*clip*)
Input value or `None` to turn OFF.

set_squash(*squash*)
Input multiplicative factor for `squash * tanh(sample)` (usually will be 1), or `None` to turn OFF.

set_noise_clip(*noise_clip*)
Input value or `None` to turn OFF.

set_std(*std*)
Input value, which can be same shape as action space, or else broadcastable up to that shape, or `None` to turn OFF and use `dist_info.log_std` in other methods.

CHAPTER 10

Spaces

Spaces are used to specify the interfaces from the environment to the agent (model); the describe the observations and actions.

class `rlpyt.spaces.base.`**`Space`**

Common definitions for observations and actions.

`sample``()`

Uniformly randomly sample a random element of this space.

`null_value``()`

Return a null value used to fill for absence of element.

class `rlpyt.spaces.int_box.`**`IntBox`**`(`*low*, *high*, *shape=None*, *dtype='int32'*, *null_value=None*`)`

Bases: *`rlpyt.spaces.base.Space`*

A box in *J^n*, with specificiable bound and dtype.

`__init__``(`*low*, *high*, *shape=None*, *dtype='int32'*, *null_value=None*`)`

Params `low` and `high` are scalars, applied across all dimensions of shape; valid values will be those in `range(low, high)`.

`sample``()`

Return a single sample from `np.random.randint`.

`n`

The number of elements in the space.

class `rlpyt.spaces.float_box.`**`FloatBox`**`(`*low*, *high*, *shape=None*, *null_value=0.0*, *dtype='float32'*`)`

Bases: *`rlpyt.spaces.base.Space`*

A box in *R^n*, with specifiable bounds and dtype.

`__init__``(`*low*, *high*, *shape=None*, *null_value=0.0*, *dtype='float32'*`)`

Two kinds of valid input:

- low and high are scalars, and shape is provided: Box(-1.0, 1.0, (3,4))

- low and high are arrays of the same shape: Box(np.array([-1.0,-2.0]), np.array([2.0,4.0]))

sample()
Return a single sample from `np.random.uniform`.

class rlpyt.spaces.composite.**Composite**(*spaces*, *NamedTupleCls*)
Bases: *rlpyt.spaces.base.Space*

A space for composing arbitrary combinations of spaces together.

__init__(*spaces*, *NamedTupleCls*)
Must input the instantiated sub-spaces in order (e.g. list or tuple), and a named tuple class with whch to organize the sub-spaces and resulting samples. The `NamedTupleCls` should be defined in the module (file) which defines the composite space.

sample()
Return a single sample which is a named tuple composed of samples from all sub-spaces.

null_value()
Return a null value which is a named tuple composed of null values from all sub-spaces.

shape
Return a named tuple composed of shapes of every sub-space.

names
Return names of sub-spaces.

spaces
Return the bare sub-spaces.

class rlpyt.spaces.gym_wrapper.**GymSpaceWrapper**(*space*, *null_value=0*, *name='obs'*, *force_float32=True*)
Wraps a gym space to match the rlpyt interface; most of the functionality is for automatically converting a GymDict (dictionary) space into an rlpyt Composite space (and converting between the two). Use inside the initialization of the environment wrapper for a gym environment.

__init__(*space*, *null_value=0*, *name='obs'*, *force_float32=True*)
Input `space` is a gym space instance.

Input `name` is used to disambiguate different gym spaces being wrapped, which is necessary if more than one GymDict space is to be wrapped in the same file. The reason is that the associated namedtuples must be defined in the globals of this file, so they must have distinct names.

sample()
Returns a single sample in a namedtuple (for composite) or numpy array using the the `sample()` method of the underlying gym space(s).

null_value()
Similar to `sample()` but returning a null value.

convert(*value*)
For dictionary space, use to convert wrapped env's dict to rlpyt namedtuple, i.e. inside the environment wrapper's `step()`, for observation output to the rlpyt sampler (see helper function in file)

revert(*value*)
For dictionary space, use to revert namedtuple action into wrapped env's dict, i.e. inside the environment wrappers `step()`, for input to the underlying gym environment (see helper function in file).

CHAPTER 11

Model Components

This page documents the implemented neural network components. These are intended as building blocks for the agent model, but not to be used as standalone models (should probably disambiguate the name from *model*).

Complete models which actually function as the agent model have additional functionality in the `forward()` method for handling of leading dimensions of inputs/outputs. See `infer_leading_dims()` and `restore_leading_dims` until utilities, and see the documentation for each algorithm for associated complete models.

class `rlpyt.models.mlp.`**`MlpModel`**(*input_size*, *hidden_sizes*, *output_size=None*, *nonlinearity=<sphinx.ext.autodoc.importer._MockObject object>*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Multilayer Perceptron with last layer linear.

Parameters

- **`input_size`** (*`int`*) – number of inputs
- **`hidden_sizes`** (*`list`*) – can be empty list for none (linear model).
- **`output_size`** – linear layer at output, or if `None`, the last hidden size will be the output size and will have nonlinearity applied
- **`nonlinearity`** – torch nonlinearity Module (not Functional).

`forward`(*input*)

Compute the model on the input, assuming input shape [B,input_size].

`output_size`

Retuns the output size of the model.

class `rlpyt.models.conv2d.`**`Conv2dModel`**(*in_channels*, *channels*, *kernel_sizes*, *strides*, *paddings=None*, *nonlinearity=<sphinx.ext.autodoc.importer._MockObject object>*, *use_maxpool=False*, *head_sizes=None*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

2-D Convolutional model component, with option for max-pooling vs downsampling for strides > 1. Requires number of input channels, but not input shape. Uses `torch.nn.Conv2d`.

forward(*input*)
Computes the convolution stack on the input; assumes correct shape already: [B,C,H,W].

conv_out_size(*h*, *w*, *c=None*)
Helper function ot return the output size for a given input shape, without actually performing a forward pass through the model.

class `rlpyt.models.conv2d.`**Conv2dHeadModel**(*image_shape*, *channels*, *kernel_sizes*, *strides*, *hidden_sizes*, *output_size=None*, *paddings=None*, *nonlinearity=<sphinx.ext.autodoc.importer._MockObject object>*, *use_maxpool=False*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Model component composed of a `Conv2dModel` component followed by a fully-connected `MlpModel` head. Requires full input image shape to instantiate the MLP head.

forward(*input*)
Compute the convolution and fully connected head on the input; assumes correct input shape: [B,C,H,W].

output_size
Returns the final output size after MLP head.

11.1 Utilities

`rlpyt.models.utils.`**conv2d_output_shape**(*h*, *w*, *kernel_size=1*, *stride=1*, *padding=0*, *dilation=1*)
Returns output H, W after convolution/pooling on input H, W.

class `rlpyt.models.utils.`**ScaleGrad**(**args*, ***kwargs*)
Model component to scale gradients back from layer, without affecting the forward pass. Used e.g. in dueling heads DQN models.

static forward(*ctx*, *tensor*, *scale*)
Stores the `scale` input to `ctx` for application in `backward()`; simply returns the input `tensor`.

static backward(*ctx*, *grad_output*)
Return the `grad_output` multiplied by `ctx.scale`. Also returns a `None` as placeholder corresponding to (non-existent) gradient of the input `scale` of `forward()`.

`rlpyt.models.utils.`**update_state_dict**(*model*, *state_dict*, *tau=1*, *strip_ddp=True*)
Update the state dict of `model` using the input `state_dict`, which must match format. `tau==1` applies hard update, copying the values, `0<tau<1` applies soft update: `tau * new + (1 - tau) * old`.

`rlpyt.models.utils.`**strip_ddp_state_dict**(*state_dict*)
Workaround the fact that DistributedDataParallel prepends 'module.' to every key, but the sampler models will not be wrapped in DistributedDataParallel. (Solution from PyTorch forums.)

CHAPTER 12

Environments

The Atari Environment and a Gym Env Wrapper are included in rlpyt.

12.1 Atari

class `rlpyt.envs.atari.atari_env.`**`AtariTrajInfo`**`(`*`**kwargs`*`)`

Bases: `rlpyt.samplers.collections.TrajInfo`

TrajInfo class for use with Atari Env, to store raw game score separate from clipped reward signal.

class `rlpyt.envs.atari.atari_env.`**`AtariEnv`**`(`*game='pong'*, *frame_skip=4*, *num_img_obs=4*, *clip_reward=True*, *episodic_lives=True*, *fire_on_reset=False*, *max_start_noops=30*, *repeat_action_probability=0.0*, *horizon=27000*`)`

Bases: *`rlpyt.envs.base.Env`*

An efficient implementation of the classic Atari RL envrionment using the Arcade Learning Environment (ALE).

Output *env_info* includes:

- *game_score*: raw game score, separate from reward clipping.
- *traj_done*: special signal which signals game-over or timeout, so that sampler doesn't reset the environment when `done==True` but `traj_done==False`, which can happen when `episodic_lives==True`.

Always performs 2-frame max to avoid flickering (this is pretty fast).

Screen size downsampling is done by cropping two rows and then downsampling by 2x using *cv2*: (210, 160) –> (80, 104). Downsampling by 2x is much faster than the old scheme to (84, 84), and the (80, 104) shape is fairly convenient for convolution filter parameters which don't cut off edges.

The action space is an *IntBox* for the number of actions. The observation space is an *IntBox* with `dtype=uint8` to save memory; conversion to float should happen inside the agent's model's `forward()` method.

(See the file for implementation details.)

Parameters

- **game** (*str*) – game name
- **frame_skip** (*int*) – frames per step (>=1)
- **num_img_obs** (*int*) – number of frames in observation (>=1)
- **clip_reward** (*bool*) – if `True`, clip reward to np.sign(reward)
- **episodic_lives** (*bool*) – if `True`, output `done=True` but `env_info[traj_done]=False` when a life is lost
- **max_start_noops** (*int*) – upper limit for random number of noop actions after reset
- **repeat_action_probability** (*0-1*) – probability for sticky actions
- **horizon** (*int*) – max number of steps before timeout / `traj_done=True`

reset()
Performs hard reset of ALE game.

12.2 Gym Wrappers

class rlpyt.envs.gym.**GymEnvWrapper**(*env*, *act_null_value=0*, *obs_null_value=0*, *force_float32=True*)
Gym-style wrapper for converting the Openai Gym interface to the rlpyt interface. Action and observation spaces are wrapped by rlpyt's `GymSpaceWrapper`.

Output *env_info* is automatically converted from a dictionary to a corresponding namedtuple, which the rlpyt sampler expects. For this to work, every key that might appear in the gym environments *env_info* at any step must appear at the first step after a reset, as the *env_info* entries will have sampler memory pre-allocated for them (so they also cannot change dtype or shape). (see *EnvInfoWrapper*, *build_info_tuples*, and *info_to_nt* in file or more help/details)

Warning: Unrecognized keys in *env_info* appearing later during use will be silently ignored.

This wrapper looks for gym's `TimeLimit` env wrapper to see whether to add the field `timeout` to env info.

step(*action*)
Reverts the action from rlpyt format to gym format (i.e. if composite-to- dictionary spaces), steps the gym environment, converts the observation from gym to rlpyt format (i.e. if dict-to-composite), and converts the env_info from dictionary into namedtuple.

reset()
Returns converted observation from gym env reset.

spaces
Returns the rlpyt spaces for the wrapped env.

class rlpyt.envs.gym.**EnvInfoWrapper**(*env*, *info_example*)
Gym-style environment wrapper to infill the *env_info* dict of every `step()` with a pre-defined set of examples, so that *env_info* has those fields at every step and they are made available to the algorithm in the sampler's batch of data.

step(*action*)
If need be, put extra fields into the *env_info* dict returned. See file for function `infill_info()` for details.

`rlpyt.envs.gym.`**`make`**`(`**args*, *info_example=None*, ***kwargs*`)`

Use as factory function for making instances of gym environment with rlpyt's `GymEnvWrapper`, using `gym.make(*args, **kwargs)`. If `info_example` is not `None`, will include the `EnvInfoWrapper`.

CHAPTER 13

Replay Buffers

Several variants of replay buffers are included in rlpyt. Options include: n-step returns (computed by the replay buffer), prioritized replay (sum-tree), frame-based observation storage (for memory savings), and replay of sequences.

All buffers are based on pre-allocated a size of memory with leading dimensions [T,B], where B is the expected (and required) corresponding dimension in the input sample batches (which will be the number of parallel environmnets in the sampler), and T is chosen to attain the total requested buffer size. A universal time cursor tracks the position of latest inputs along the T dimension of the buffer, and it wraps automatically. Use of namedarraytuples makes it straightforward to write data of arbitrary structure to the buffer's next indexes. Further benefits are that pre-allocated storage doesn't grow and is more easily shared across processes (async mode). But this format does require accounting for which samples are currently invalid due to partial memory overwrite, based on n-step returns or needing to replay sequences. If memory and performance optimization are less of a concern, it might be preferable to write a simpler buffer which, for example, stores a rotating list of complete sequences to replay.

Hint: The implemented replay buffers share a lot of components, and sub-classing with multiple inheritances is used to prevent redundant code. If modifying a replay buffer, it might be easier to first copy all desired components into one monolithic class, and then work from there.

13.1 Replay Buffer Components

13.1.1 Base Buffers

class `rlpyt.replays.base.`**`BaseReplayBuffer`**

> **`append_samples`**(*samples*)
> Add new data to the replay buffer, possibly ejecting old data.
>
> **`sample_batch`**(*batch_B*, *batch_T=None*)
> Returns a data batch, e.g. for training.

class rlpyt.replays.n_step.**BaseNStepReturnBuffer**(*example*, *size*, *B*, *discount=1*, *n_step_return=1*)

Bases: `rlpyt.replays.base.BaseReplayBuffer`

Stores the most recent data and computes n_step returns. Operations are all vectorized, as data is stored with leading dimensions [T,B]. Cursor is next idx to be written.

For now, Assume all incoming samples are "valid" (i.e. must have mid_batch_reset=True in sampler). Can relax this later by tracking valid for each data sample.

Subclass this with specific batch sampling scheme.

Latest n_step timesteps up to cursor are temporarily invalid because all future empirical rewards not yet sampled (*off_backward*). The current cursor position is also an invalid sample, because the previous action and previous reward have been overwritten (off_forward).

Input `example` should be a namedtuple with the structure of data (and one example each, no leading dimensions), which will be input every time samples are appended.

If `n_step_return>1`, then additional buffers `samples_return_` and `samples_done_n` will also be allocated. n-step returns for a given sample will be stored at that same index (e.g. samples_return_[T,B] will store reward[T,B] + discount * reward[T+1,B], + discount ** 2 * reward[T+2,B],. . .). `done_n` refers to whether a `done=True` signal appears in any of the n-step future, such that the following value should *not* be bootstrapped.

append_samples(*samples*)
Write the samples into the buffer and advance the time cursor. Handle wrapping of the cursor if necessary (boundary doesn't need to align with length of `samples`). Compute and store returns with newly available rewards.

compute_returns(*T*)
Compute the n-step returns using the new rewards just written into the buffer, but before the buffer cursor is advanced. Input `T` is the number of new timesteps which were just written. Does nothing if *n-step==1*. e.g. if 2-step return, t-1 is first return written here, using reward at t-1 and new reward at t (up through t-1+T from t+T).

class rlpyt.replays.frame.**FrameBufferMixin**(*example*, ***kwargs*)
Like n-step return buffer but expects multi-frame input observation where each new observation has one new frame and the rest old; stores only unique frames to save memory. Samples observation should be shaped: [T,B,C,..] with C the number of frames. Expects frame order: OLDEST to NEWEST.

A special method for replay will be required to piece the frames back together into full observations.

Latest n_steps up to cursor temporarilty invalid because "next" not yet written. Cursor timestep invalid because previous action and reward overwritten. NEW: Next n_frames-1 invalid because observation history frames overwritten.

append_samples(*samples*)
Appends all samples except for the *observation* as normal. Only the new frame in each observation is recorded.

class rlpyt.replays.async_.**AsyncReplayBufferMixin**(**args*, ***kwargs*)
Mixin class which manages the buffer (shared) memory under a read-write lock (multiple-reader, single-writer), for use with the asynchronous runner. Wraps the `append_samples()`, `sample_batch()`, and `update_batch_priorities()` methods. Maintains a universal buffer cursor, communicated asynchronously. Supports multiple buffer-writer processes and multiple replay processes.

13.1.2 Non-Sequence Replays

class `rlpyt.replays.non_sequence.n_step.`**`NStepReturnBuffer`**(*example*, *size*, *B*, *discount=1*, *n_step_return=1*)

Bases: *`rlpyt.replays.n_step.BaseNStepReturnBuffer`*

Definition of what fields are replayed from basic n-step return buffer.

`extract_batch`(*T_idxs*, *B_idxs*)
From buffer locations *[T_idxs,B_idxs]*, extract data needed for training, including target values at *T_idxs* + *n_step_return*. Returns namedarraytuple of torch tensors (see file for all fields). Each tensor has leading batch dimension `len(T_idxs)==len(B_idxs)`, but individual samples are drawn, so no leading time dimension.

`extract_observation`(*T_idxs*, *B_idxs*)
Simply `observation[T_idxs,B_idxs]`; generalization anticipating frame-based buffer.

class `rlpyt.replays.non_sequence.frame.`**`NStepFrameBuffer`**(*example*, ***kwargs*)
Bases: *`rlpyt.replays.frame.FrameBufferMixin`*, *`rlpyt.replays.non_sequence.n_step.NStepReturnBuffer`*

Special method for re-assembling observations from frames.

`extract_observation`(*T_idxs*, *B_idxs*)
Assembles multi-frame observations from frame-wise buffer. Frames are ordered OLDEST to NEWEST along C dim: [B,C,H,W]. Where `done=True` is found, the history is not full due to recent environment reset, so these frames are zero-ed.

class `rlpyt.replays.non_sequence.uniform.`**`UniformReplay`**
Replay of individual samples by uniform random selection.

`sample_batch`(*batch_B*)
Randomly select desired batch size of samples to return, uses `sample_idxs()` and `extract_batch()`.

`sample_idxs`(*batch_B*)
Randomly choose the indexes of data to return using `np.random.randint()`. Disallow samples within certain proximity to the current cursor which hold invalid data.

class `rlpyt.replays.non_sequence.prioritized.`**`PrioritizedReplay`**(*alpha=0.6*, *beta=0.4*, *default_priority=1*, *unique=False*, *input_priorities=False*, *input_priority_shift=0*, ***kwargs*)
Prioritized experience replay using sum-tree prioritization.

The priority tree must configure at instantiation if priorities will be input with samples in `append_samples()`, by parameter `input_priorities=True`, else the default value will be applied to all new samples.

`append_samples`(*samples*)
Looks for `samples.priorities`; if not found, uses default priority. Writes samples using super class's `append_samples`, and advances matching cursor in priority tree.

sample_batch(*batch_B*)
Calls on the priority tree to generate random samples. Returns samples data and normalized importance-sampling weights: `is_weights=priorities ** -beta`

update_batch_priorities(*priorities*)
Takes in new priorities (i.e. from the algorithm after a training step) and sends them to priority tree as `priorities ** alpha`; the tree internally remembers which indexes were sampled for this batch.

class `rlpyt.replays.non_sequence.time_limit.`**NStepTimeLimitBuffer**(**args*, ***kwargs*)
Bases: *`rlpyt.replays.non_sequence.n_step.NStepReturnBuffer`*

For use in e.g. SAC when bootstrapping when env *done* due to timeout. Expects input samples to include `timeout` field, and returns `timeout` and `timeout_n` similar to `done` and `done_n`.

13.1.3 Sequence Replays

class `rlpyt.replays.sequence.n_step.`**SequenceNStepReturnBuffer**(*example*, *size*, *B*, *rnn_state_interval*, *batch_T=None*, ***kwargs*)
Bases: *`rlpyt.replays.n_step.BaseNStepReturnBuffer`*

Base n-step return buffer for sequences replays. Includes storage of agent's recurrent (RNN) state.

Use of `rnn_state_interval>1` only periodically stores RNN state, to save memory. The replay mechanism must account for the fact that only time-steps with saved RNN state are valid first states for replay. (`rnn_state_interval<1` does not store RNN state.)

append_samples(*samples*)
Special handling for RNN state storage, and otherwise uses superclass's `append_samples()`.

extract_batch(*T_idxs*, *B_idxs*, *T*)
Return full sequence of each field in *agent_inputs* (e.g. *observation*), including all timesteps for the main sequence and for the target sequence in one array; many timesteps will likely overlap, so the algorithm and make sub-sequences by slicing on device, for reduced memory usage.

Enforces that input *T_idxs* align with RNN state interval.

Uses helper function `extract_sequences()` to retrieve samples of length `T` starting at locations `[T_idxs,B_idxs]`, so returned data batch has leading dimensions `[T,len(B_idxs)]`.

class `rlpyt.replays.sequence.frame.`**SequenceNStepFrameBuffer**(*example*, ***kwargs*)
Bases: *`rlpyt.replays.frame.FrameBufferMixin`*, *`rlpyt.replays.sequence.n_step.SequenceNStepReturnBuffer`*

Includes special method for extracting observation sequences from a frame-wise buffer, where each time-step includes multiple frames. Each returned sequence will contain many redundant frames (A more efficient way would be to turn the Conv2D into a Conv3D and only return unique frames.)

extract_observation(*T_idxs*, *B_idxs*, *T*)
Observations are re-assembled from frame-wise buffer as [T,B,C,H,W], where C is the frame-history channels, which will have redundancy across the T dimension. Frames are returned OLDEST to NEWEST along the C dimension.

Frames are zero-ed after environment resets.

class `rlpyt.replays.sequence.uniform.`**UniformSequenceReplay**
Replays sequences with starting state chosen uniformly randomly.

sample_batch(*batch_B*, *batch_T=None*)
Can dynamically input length of sequences to return, by `batch_T`, else if `None` will use interanlly set value. Returns batch with leading dimensions `[batch_T, batch_B]`.

sample_idxs(*batch_B*, *batch_T*)
Randomly choose the indexes of starting data to return using `np.random.randint()`. Disallow samples within certain proximity to the current cursor which hold invalid data, including accounting for sequence length (so every state returned in sequence will hold valid data). If the RNN state is only stored periodically, only choose starting states with stored RNN state.

class `rlpyt.replays.sequence.prioritized.`**PrioritizedSequenceReplay**(*alpha=0.6*, *beta=0.4*, *default_priority=1*, *unique=False*, *input_priorities=False*, *input_priority_shift=0*, ***kwargs*)

Prioritized experience replay of sequences using sum-tree prioritization. The size of the sum-tree is based on the number of RNN states stored, since valid sequences must start with an RNN state. Hence using periodic storage with `rnn_state_inveral>1` results in a faster tree using less memory. Replay buffer priorities are indexed to the start of the whole sequence to be returned, regardless of whether the initial part is used only as RNN warmup.

Requires `batch_T` to be set and fixed at instantiation, so that the priority tree has a fixed scheme for which samples are temporarilty invalid due to the looping cursor (the tree must set and propagate 0-priorities for those samples, so dynamic `batch_T` could require additional tree operations for every sampling event).

Parameter `input_priority_shift` is used to assign input priorities to a starting time-step which is shifted from the samples input to `append_samples()`. For example, in R2D1, using replay sequences of 120 time-steps, with 40 steps for warmup and 80 steps for training, we might run the sampler with 40-step batches, and store the RNN state only at the beginning of each batch: `rnn_state_interval=40`. In this scenario, we would use `input_priority_shift=2`, so that the input priorities which are provided with each batch of samples are assigned to sequence start-states at the beginning of warmup (shifted 2 entries back in the priority tree). This way, the input priorities can be computed after seeing all 80 training steps. In the meantime, the partially-written sequences are marked as temporarily invalid for replay anyway, according to buffer cursor position and the fixed `batch_T` replay setting. (If memory and performance optimization are less of a concern, the indexing effort might all be simplified by writing a replay buffer which manages a list of valid trajectories to sample, rather than a monolithic, pre-allocated buffer.)

append_samples(*samples*)
Like non-sequence prioritized, except also stores RNN state, and advances the priority tree cursor according to the number of RNN states stored (which might be less than overall number of time-steps).

sample_batch(*batch_B*)
Returns batch with leading dimensions `[self.batch_T, batch_B]`, with each sequence sampled randomly according to priority. (`self.batch_T` should not be changed).

13.1.4 Priority Tree

class `rlpyt.replays.sum_tree.`**SumTree**(*T*, *B*, *off_backward*, *off_forward*, *default_value=1*, *enable_input_priorities=False*, *input_priority_shift=0*)
Sum tree for matrix of values stored as [T,B], updated in chunks along T dimension, applying to the full B dimension at each update. Priorities represented as first T*B leaves of binary tree. Turns on/off entries in vicinity

of cursor position according to "off_backward" (e.g. n_step_return) and "off_forward" (e.g. 1 for prev_action or max(1, frames-1) for frame-wise buffer). Provides efficient sampling from non-uniform probability masses.

Note: Tried single precision (float32) tree, and it sometimes returned samples with priority 0.0, because subtraction during tree cascade left random value larger than the remaining sum; suggest keeping float64.

---

advance (*T*, *priorities=None*)
Cursor advances by T: set priorities to zero in vicinity of new cursor position and turn priorities on for new samples since previous cursor position. Optional param `priorities` can be None for default, or of dimensions [T, B], or [B] or scalar will broadcast. (Must have enabled `input_priorities=True` when instantiating the tree.) These will be stored at the current cursor position, meaning these priorities correspond to the current values being added to the buffer, even though their priority might temporarily be set to zero until future advances.

sample (*n*, *unique=False*)
Get *n* samples, with replacement (default) or without. Use `np.random.rand()` to generate random values with which to descend the tree to each sampled leaf node. Returns *T_idxs* and *B_idxs*, and sample priorities.

update_batch_priorities (*priorities*)
Apply new priorities to tree at the leaf positions where the last batch was returned from the `sample()` method.

print_tree (*level=None*)
Print values for whole tree or at specified level.

class rlpyt.replays.sum_tree.**AsyncSumTree** (**args*, ***kwargs*)
Bases: *rlpyt.replays.sum_tree.SumTree*

Allocates the tree into shared memory, and manages asynchronous cursor position, for different read and write processes. Assumes that writing to tree values is lock protected elsewhere, i.e. by the replay buffer.

13.2 Full Replay Buffer Classes

These are all defined purely as sub-classes with above components.

13.2.1 Non-Sequence Replay

class rlpyt.replays.non_sequence.uniform.**UniformReplayBuffer** (*example*, *size*, *B*, *discount=1*, *n_step_return=1*)
Bases: *rlpyt.replays.non_sequence.uniform.UniformReplay*, *rlpyt.replays.non_sequence.n_step.NStepReturnBuffer*

class rlpyt.replays.non_sequence.uniform.**AsyncUniformReplayBuffer** (**args*, ***kwargs*)
Bases: *rlpyt.replays.async_.AsyncReplayBufferMixin*, *rlpyt.replays.non_sequence.uniform.UniformReplayBuffer*

class rlpyt.replays.non_sequence.prioritized.**PrioritizedReplayBuffer**(*alpha=0.6, beta=0.4, default_priority=1, unique=False, input_priorities=False, input_priority_shift=0, **kwargs*)

Bases: *rlpyt.replays.non_sequence.prioritized.PrioritizedReplay*, *rlpyt.replays.non_sequence.n_step.NStepReturnBuffer*

class rlpyt.replays.non_sequence.prioritized.**AsyncPrioritizedReplayBuffer**(**args, **kwargs*)

Bases: *rlpyt.replays.async_.AsyncReplayBufferMixin*, *rlpyt.replays.non_sequence.prioritized.PrioritizedReplayBuffer*

class rlpyt.replays.non_sequence.frame.**UniformReplayFrameBuffer**(*example, **kwargs*)

Bases: *rlpyt.replays.non_sequence.uniform.UniformReplay*, *rlpyt.replays.non_sequence.frame.NStepFrameBuffer*

class rlpyt.replays.non_sequence.frame.**PrioritizedReplayFrameBuffer**(*alpha=0.6, beta=0.4, default_priority=1, unique=False, input_priorities=False, input_priority_shift=0, **kwargs*)

Bases: *rlpyt.replays.non_sequence.prioritized.PrioritizedReplay*, *rlpyt.replays.non_sequence.frame.NStepFrameBuffer*

class rlpyt.replays.non_sequence.frame.**AsyncUniformReplayFrameBuffer**(**args, **kwargs*)

Bases: *rlpyt.replays.async_.AsyncReplayBufferMixin*, *rlpyt.replays.non_sequence.frame.UniformReplayFrameBuffer*

class rlpyt.replays.non_sequence.frame.**AsyncPrioritizedReplayFrameBuffer**(**args, **kwargs*)

Bases: *rlpyt.replays.async_.AsyncReplayBufferMixin*, *rlpyt.replays.non_sequence.frame.PrioritizedReplayFrameBuffer*

class rlpyt.replays.non_sequence.time_limit.**TlUniformReplayBuffer**(**args, **kwargs*)

Bases: *rlpyt.replays.non_sequence.uniform.UniformReplay*, *rlpyt.replays.non_sequence.time_limit.NStepTimeLimitBuffer*

class `rlpyt.replays.non_sequence.time_limit.`**`TlPrioritizedReplayBuffer`**(*alpha=0.6, beta=0.4, default_priority=1, unique=False, input_priorities=False, input_priority_shift=0, **kwargs*)

Bases: *`rlpyt.replays.non_sequence.prioritized.PrioritizedReplay`*, *`rlpyt.replays.non_sequence.time_limit.NStepTimeLimitBuffer`*

class `rlpyt.replays.non_sequence.time_limit.`**`AsyncTlUniformReplayBuffer`**(**args, **kwargs*)

Bases: *`rlpyt.replays.async_.AsyncReplayBufferMixin`*, *`rlpyt.replays.non_sequence.time_limit.TlUniformReplayBuffer`*

class `rlpyt.replays.non_sequence.time_limit.`**`AsyncTlPrioritizedReplayBuffer`**(**args, **kwargs*)

Bases: *`rlpyt.replays.async_.AsyncReplayBufferMixin`*, *`rlpyt.replays.non_sequence.time_limit.TlPrioritizedReplayBuffer`*

13.2.2 Sequence Replay

class `rlpyt.replays.sequence.uniform.`**`UniformSequenceReplayBuffer`**(*example, size, B, rnn_state_interval, batch_T=None, **kwargs*)

Bases: *`rlpyt.replays.sequence.uniform.UniformSequenceReplay`*, *`rlpyt.replays.sequence.n_step.SequenceNStepReturnBuffer`*

class `rlpyt.replays.sequence.uniform.`**`AsyncUniformSequenceReplayBuffer`**(**args, **kwargs*)

Bases: *`rlpyt.replays.async_.AsyncReplayBufferMixin`*, *`rlpyt.replays.sequence.uniform.UniformSequenceReplayBuffer`*

class `rlpyt.replays.sequence.prioritized.`**`PrioritizedSequenceReplayBuffer`**(*alpha=0.6, beta=0.4, default_priority=1, unique=False, input_priorities=False, input_priority_shift=0, **kwargs*)

Bases: *`rlpyt.replays.sequence.prioritized.PrioritizedSequenceReplay`*, *`rlpyt.replays.sequence.n_step.SequenceNStepReturnBuffer`*

class `rlpyt.replays.sequence.prioritized.`**`AsyncPrioritizedSequenceReplayBuffer`**(**args, **kwargs*)

Bases: *`rlpyt.replays.async_.AsyncReplayBufferMixin`*, *`rlpyt.replays.sequence.prioritized.PrioritizedSequenceReplayBuffer`*

class `rlpyt.replays.sequence.frame.`**`UniformSequenceReplayFrameBuffer`**(*example, **kwargs*)

Bases: *rlpyt.replays.sequence.uniform.UniformSequenceReplay*, *rlpyt.replays.sequence.frame.SequenceNStepFrameBuffer*

class rlpyt.replays.sequence.frame.**PrioritizedSequenceReplayFrameBuffer**(*alpha=0.6*, *beta=0.4*, *default_priority=1*, *unique=False*, *input_priorities=False*, *input_priority_shift=0*, ***kwargs*)

Bases: *rlpyt.replays.sequence.prioritized.PrioritizedSequenceReplay*, *rlpyt.replays.sequence.frame.SequenceNStepFrameBuffer*

class rlpyt.replays.sequence.frame.**AsyncUniformSequenceReplayFrameBuffer**(**args*, ***kwargs*)

Bases: *rlpyt.replays.async_.AsyncReplayBufferMixin*, *rlpyt.replays.sequence.frame.UniformSequenceReplayFrameBuffer*

class rlpyt.replays.sequence.frame.**AsyncPrioritizedSequenceReplayFrameBuffer**(**args*, ***kwargs*)

Bases: *rlpyt.replays.async_.AsyncReplayBufferMixin*, *rlpyt.replays.sequence.frame.PrioritizedSequenceReplayFrameBuffer*

CHAPTER 14

Named Array Tuples

rlpyt.utils.collections.**namedarraytuple**(*typename*, *field_names*, *return_namedtuple_cls=False*, *classname_suffix=False*)

Returns a new subclass of a namedtuple which exposes indexing / slicing reads and writes applied to all contained objects, which must share indexing (__getitem__) behavior (e.g. numpy arrays or torch tensors).

(Code follows pattern of collections.namedtuple.)

```
>>> PointsCls = namedarraytuple('Points', ['x', 'y'])
>>> p = PointsCls(np.array([0, 1]), y=np.array([10, 11]))
>>> p
Points(x=array([0, 1]), y=array([10, 11]))
>>> p.x                         # fields accessible by name
array([0, 1])
>>> p[0]                        # get location across all fields
Points(x=0, y=10)               # (location can be index or slice)
>>> p.get(0)                    # regular tuple-indexing into field
array([0, 1])
>>> x, y = p                    # unpack like a regular tuple
>>> x
array([0, 1])
>>> p[1] = 2                    # assign value to location of all fields
>>> p
Points(x=array([0, 2]), y=array([10, 2]))
>>> p[1] = PointsCls(3, 30)     # assign to location field-by-field
>>> p
Points(x=array([0, 3]), y=array([10, 30]))
>>> 'x' in p                    # check field name instead of object
True
```

class rlpyt.utils.collections.**DocExampleNat**(*field_1*, *field_2*)

__contains__(*key*)
Checks presence of field name (unlike tuple; like dict).

__getitem__(*loc*)
Return a new DocExampleNat instance containing the selected index or slice from each field.

__setitem__(*loc*, *value*)
If input value is the same named[array]tuple type, iterate through its fields and assign values into selected index or slice of corresponding field. Else, assign whole of value to selected index or slice of all fields. Ignore fields that are both None.

get(*index*)
Retrieve value as if indexing into regular tuple.

items()
Iterate ordered (field_name, value) pairs (like OrderedDict).

`rlpyt.utils.collections.`**is_namedtuple_class**(*obj*)
Heuristic, might be spoofed. Returns False if obj is namedarraytuple class.

`rlpyt.utils.collections.`**is_namedarraytuple_class**(*obj*)
Heuristic, might be spoofed. Returns False if obj is namedtuple class.

`rlpyt.utils.collections.`**is_namedtuple**(*obj*)
Heuristic, might be spoofed. Returns False if obj is namedarraytuple.

`rlpyt.utils.collections.`**is_namedarraytuple**(*obj*)
Heuristic, might be spoofed. Returns False if obj is namedtuple.

`rlpyt.utils.collections.`**namedarraytuple_like**(*namedtuple_or_class*, *classname_suffix=False*)
Returns namedarraytuple class with same name and fields as input namedtuple or namedarraytuple instance or class. If input is namedarraytuple instance or class, the same class is returned directly. Input can also be from the new Schema format, instances of the four: Named[Array]Tuple[Schema].

14.1 Alternative Implementation

Classes for creating objects which closely follow the interfaces for namedtuple and namedarraytuple types and instances, except without defining a new class for each type. (May be easier to use with regards to pickling under spawn, or dynamically creating types, by avoiding module-level definitions.

class `rlpyt.utils.collections.`**NamedTupleSchema**(*typename*, *fields*)
Instances of this class act like a type returned by namedtuple().

__call__(**args*, ***kwargs*)
Allows instances to act like *namedtuple* constructors.

_make(*iterable*)
Allows instances to act like *namedtuple* constructors.

class `rlpyt.utils.collections.`**NamedTuple**
Bases: `tuple`

Instances of this class act like instances of namedtuple types, but this same class is used for all namedtuple-like types created. Unlike true namedtuples, this mock avoids defining a new class for each configuration of typename and field names. Methods from namedtuple source are copied here.

Implementation differences from *namedtuple*:

- The individual fields don't show up in *dir(obj)*, but they do still show up as *hasattr(obj, field) => True*, because of *__getattr__()*.

- These objects have a *__dict__* (by ommitting *__slots__ = ()*), intended to hold only the typename and list of field names, which are now instance attributes instead of class attributes.
- Since *property(itemgetter(i))* only works on classes, *__getattr__()* is modified instead to look for field names.
- Attempts to enforce call signatures are included, might not exactly match.

`__getattr__`(*name*)
Look in *_fields* when *name* is not in *dir(self)*.

`_make`(*iterable*)
Make a new object of same typename and fields from a sequence or iterable.

`_replace`(***kwargs*)
Return a new object of same typename and fields, replacing specified fields with new values.

`_asdict`()
Return an ordered dictionary mapping field names to their values.

class `rlpyt.utils.collections.`**`NamedArrayTupleSchema`**(**args*, ***kwargs*)
Bases: *`rlpyt.utils.collections.NamedTupleSchema`*

Instances of this class act like a type returned by rlpyt's namedarraytuple().

class `rlpyt.utils.collections.`**`NamedArrayTuple`**
Bases: *`rlpyt.utils.collections.NamedTuple`*

`rlpyt.utils.collections.`**`NamedArrayTupleSchema_like`**(*example*)
Returns a NamedArrayTupleSchema instance with the same name and fields as input, which can be a class or instance of namedtuple or namedarraytuple, or an instance of NamedTupleSchema, NamedTuple, NamedArrayTupleSchema, or NamedArrayTuple.

CHAPTER 15

Utilities

Here are listed number of miscellaneous utilities used in rlpyt.

15.1 Array

Miscellaneous functions for manipulating numpy arrays.

`rlpyt.utils.array.`**`select_at_indexes`**(*indexes*, *array*)
: Returns the contents of `array` at the multi-dimensional integer array `indexes`. Leading dimensions of `array` must match the dimensions of `indexes`.

`rlpyt.utils.array.`**`to_onehot`**(*indexes*, *dim*, *dtype=None*)
: Converts integer values in multi-dimensional array `indexes` to one-hot values of size `dim`; expanded in an additional trailing dimension.

`rlpyt.utils.array.`**`valid_mean`**(*array*, *valid=None*, *axis=None*)
: Mean of `array`, accounting for optional mask `valid`, optionally along an axis.

`rlpyt.utils.array.`**`infer_leading_dims`**(*array*, *dim*)
: Determine any leading dimensions of `array`, which can have up to two leading dimensions more than the number of data dimensions, `dim`. Used to check for [B] or [T,B] leading. Returns size of leading dimensions (or 1 if they don't exist), the data shape, and whether the leading dimensions where found.

15.2 Tensor

Miscellaneous functions for manipulating torch tensors.

`rlpyt.utils.tensor.`**`select_at_indexes`**(*indexes*, *tensor*)
: Returns the contents of `tensor` at the multi-dimensional integer array `indexes`. Leading dimensions of `tensor` must match the dimensions of `indexes`.

`rlpyt.utils.tensor.`**`to_onehot`**(*indexes*, *num*, *dtype=None*)
Converts integer values in multi-dimensional tensor `indexes` to one-hot values of size `num`; expanded in an additional trailing dimension.

`rlpyt.utils.tensor.`**`from_onehot`**(*onehot*, *dim=-1*, *dtype=None*)
Argmax over trailing dimension of tensor `onehot`. Optional return dtype specification.

`rlpyt.utils.tensor.`**`valid_mean`**(*tensor*, *valid=None*, *dim=None*)
Mean of `tensor`, accounting for optional mask `valid`, optionally along a dimension.

`rlpyt.utils.tensor.`**`infer_leading_dims`**(*tensor*, *dim*)
Looks for up to two leading dimensions in `tensor`, before the data dimensions, of which there are assumed to be `dim` number. For use at beginning of model's `forward()` method, which should finish with `restore_leading_dims()` (see that function for help.) Returns: lead_dim: int –number of leading dims found. T: int –size of first leading dim, if two leading dims, o/w 1. B: int –size of first leading dim if one, second leading dim if two, o/w 1. shape: tensor shape after leading dims.

`rlpyt.utils.tensor.`**`restore_leading_dims`**(*tensors*, *lead_dim*, *T=1*, *B=1*)
Reshapes `tensors` (one or *tuple*, *list*) to to have `lead_dim` leading dimensions, which will become [], [B], or [T,B]. Assumes input tensors already have a leading Batch dimension, which might need to be removed. (Typically the last layer of model will compute with leading batch dimension.) For use in model `forward()` method, so that output dimensions match input dimensions, and the same model can be used for any such case. Use with outputs from `infer_leading_dims()`.

15.3 Miscellaneous Array / Tensor

`rlpyt.utils.misc.`**`iterate_mb_idxs`**(*data_length*, *minibatch_size*, *shuffle=False*)
Yields minibatches of indexes, to use as a for-loop iterator, with option to shuffle.

`rlpyt.utils.misc.`**`zeros`**(*shape*, *dtype*)
Attempt to return torch tensor of zeros, or if numpy dtype provided, return numpy array or zeros.

`rlpyt.utils.misc.`**`empty`**(*shape*, *dtype*)
Attempt to return empty torch tensor, or if numpy dtype provided, return empty numpy array.

`rlpyt.utils.misc.`**`extract_sequences`**(*array_or_tensor*, *T_idxs*, *B_idxs*, *T*)
Assumes *array_or_tensor* has [T,B] leading dims. Returns new array/tensor which contains sequences of length [T] taken from the starting indexes [T_idxs, B_idxs], where T_idxs (and B_idxs) is a list or vector of integers. Handles wrapping automatically. (Return shape: [T, len(B_idxs),. . .]).

15.4 Collections

(see Named Array Tuple page)

class `rlpyt.utils.collections.`**`AttrDict`**(**args*, ***kwargs*)
Bases: `dict`

Behaves like a dictionary but additionally has attribute-style access for both read and write. e.g. x["key"] and x.key are the same, e.g. can iterate using: for k, v in x.items(). Can sublcass for specific data classes; must call AttrDict's __init__().

`copy`()
Provides a "deep" copy of all unbroken chains of types AttrDict, but shallow copies otherwise, (e.g. numpy arrays are NOT copied).

15.5 Buffers

`rlpyt.utils.buffer.`**`buffer_from_example`**(*example*, *leading_dims*, *share_memory=False*, *use_NatSchema=None*)

Allocates memory and returns it in *namedarraytuple* with same structure as `examples`, which should be a *namedtuple* or *namedarraytuple*. Applies the same leading dimensions `leading_dims` to every entry, and otherwise matches their shapes and dtypes. The examples should have no leading dimensions. `None` fields will stay `None`. Optionally allocate on OS shared memory. Uses `build_array()`.

New: can use NamedArrayTuple types by the *use_NatSchema* flag, which may be easier for pickling/unpickling when using spawn instead of fork. If use_NatSchema is None, the type of `example` will be used to infer what type to return (this is the default)

`rlpyt.utils.buffer.`**`build_array`**(*example*, *leading_dims*, *share_memory=False*)

Allocate a numpy array matchin the dtype and shape of example, possibly with additional leading dimensions. Optionally allocate on OS shared memory.

`rlpyt.utils.buffer.`**`np_mp_array`**(*shape*, *dtype*)

Allocate a numpy array on OS shared memory.

`rlpyt.utils.buffer.`**`torchify_buffer`**(*buffer_*)

Convert contents of `buffer_` from numpy arrays to torch tensors. `buffer_` can be an arbitrary structure of tuples, namedtuples, namedarraytuples, NamedTuples, and NamedArrayTuples, and a new, matching structure will be returned. `None` fields remain `None`, and torch tensors are left alone.

`rlpyt.utils.buffer.`**`numpify_buffer`**(*buffer_*)

Convert contents of `buffer_` from torch tensors to numpy arrays. `buffer_` can be an arbitrary structure of tuples, namedtuples, namedarraytuples, NamedTuples, and NamedArrayTuples, and a new, matching structure will be returned. `None` fields remain `None`, and numpy arrays are left alone.

`rlpyt.utils.buffer.`**`buffer_to`**(*buffer_*, *device=None*)

Send contents of `buffer_` to specified device (contents must be torch tensors.). `buffer_` can be an arbitrary structure of tuples, namedtuples, namedarraytuples, NamedTuples and NamedArrayTuples, and a new, matching structure will be returned.

`rlpyt.utils.buffer.`**`buffer_method`**(*buffer_*, *method_name*, **args*, ***kwargs*)

Call method `method_name(*args, **kwargs)` on all contents of `buffer_`, and return the results. `buffer_` can be an arbitrary structure of tuples, namedtuples, namedarraytuples, NamedTuples, and NamedArrayTuples, and a new, matching structure will be returned. `None` fields remain `None`.

`rlpyt.utils.buffer.`**`buffer_func`**(*buffer_*, *func*, **args*, ***kwargs*)

Call function `func(buf, *args, **kwargs)` on all contents of `buffer_`, and return the results. `buffer_` can be an arbitrary structure of tuples, namedtuples, namedarraytuples, NamedTuples, and NamedArrayTuples, and a new, matching structure will be returned. `None` fields remain `None`.

`rlpyt.utils.buffer.`**`get_leading_dims`**(*buffer_*, *n_dim=1*)

Return the `n_dim` number of leading dimensions of the contents of `buffer_`. Checks to make sure the leading dimensions match for all tensors/arrays, except ignores `None` fields.

15.6 Algorithms

`rlpyt.algos.utils.`**`discount_return`**(*reward*, *done*, *bootstrap_value*, *discount*, *return_dest=None*)

Time-major inputs, optional other dimensions: [T], [T,B], etc. Computes discounted sum of future rewards from each time-step to the end of the batch, including bootstrapping value. Sum resets where *done* is 1. Optionally, writes to buffer *return_dest*, if provided. Operations vectorized across all trailing dimensions after the first [T,].

`rlpyt.algos.utils.`**`generalized_advantage_estimation`**`(`*reward*, *value*, *done*, *bootstrap_value*, *discount*, *gae_lambda*, *advantage_dest=None*, *return_dest=None*`)`

Time-major inputs, optional other dimensions: [T], [T,B], etc. Similar to *discount_return()* but using Generalized Advantage Estimation to compute advantages and returns.

`rlpyt.algos.utils.`**`discount_return_n_step`**`(`*reward*, *done*, *n_step*, *discount*, *return_dest=None*, *done_n_dest=None*, *do_truncated=False*`)`

Time-major inputs, optional other dimension: [T], [T,B], etc. Computes n-step discounted returns within the timeframe of the of given rewards. If *do_truncated==False*, then only compute at time-steps with full n-step future rewards are provided (i.e. not at last n-steps–output shape will change!). Returns n-step returns as well as n-step done signals, which is True if *done=True* at any future time before the n-step target bootstrap would apply (bootstrap in the algo, not here).

`rlpyt.algos.utils.`**`valid_from_done`**`(`*done*`)`

Returns a float mask which is zero for all time-steps after a *done=True* is signaled. This function operates on the leading dimension of *done*, assumed to correspond to time [T,. . .], other dimensions are preserved.

`rlpyt.algos.utils.`**`discount_return_tl`**`(`*reward*, *done*, *bootstrap_value*, *discount*, *timeout*, *value*, *return_dest=None*`)`

Like discount_return(), above, except uses bootstrapping where 'done' is due to env horizon time-limit (tl=Time-Limit). (In the algo, should not train on samples where *timeout=True*.)

`rlpyt.algos.utils.`**`generalized_advantage_estimation_tl`**`(`*reward*, *value*, *done*, *bootstrap_value*, *discount*, *gae_lambda*, *timeout*, *advantage_dest=None*, *return_dest=None*`)`

Like generalized_advantage_estimation(), above, except uses bootstrapping where 'done' is due to env horizon time-limit (tl=Time-Limit). (In the algo, should not train on samples where *timeout=True*.)

15.7 Synchronize

`class` `rlpyt.utils.synchronize.`**`RWLock`**

Multiple simultaneous readers, one writer.

`rlpyt.utils.synchronize.`**`drain_queue`**`(`*queue_obj*, *n_sentinel=0*, *guard_sentinel=False*`)`

Empty a multiprocessing queue object, with options to protect against the delay between `queue.put()` and `queue.get()`. Returns a list of the queue contents.

With `n_sentinel=0`, simply call `queue.get(block=False)` until `queue.Empty` exception (which can still happen slightly *after* another process called `queue.put()`).

With `n_sentinel>1`, call `queue.get()` until *n_sentinel* `None` objects have been returned (marking that each `put()` process has finished).

With `guard_sentinel=True` (need `n_sentinel=0`), stops if a `None` is retrieved, and puts it back into the queue, so it can do a blocking drain later with `n_sentinel>1`.

`rlpyt.utils.synchronize.`**`find_port`**`(`*offset*`)`

Find a unique open port, for initializing *torch.distributed* in multiple separate multi-GPU runs on one machine.

15.8 Quick Arguments

`rlpyt.utils.quick_args.`**`save__init__args`**(*values*, *underscore=False*, *overwrite=False*, *subclass_only=False*)

Use in *__init__()* only; assign all args/kwargs to instance attributes. To maintain precedence of args provided to subclasses, call this in the subclass before *super().__init__()* if *save__init__args()* also appears in base class, or use *overwrite=True*. With *subclass_only==True*, only args/kwargs listed in current subclass apply.

15.9 Progress Bar

`class` `rlpyt.utils.prog_bar.`**`ProgBarCounter`**(*total_count*)

Dynamic display of progress bar in terminal, for example to mark progress (and estimate time to completion) of RL iterations toward the next logging update. credit: *rllab*.

15.10 Seed

`rlpyt.utils.seed.`**`set_seed`**(*seed*)

Sets random.seed, np.random.seed, torch.manual_seed, torch.cuda.manual_seed.

`rlpyt.utils.seed.`**`make_seed`**()

Returns a random number between [0, 10000], using timing jitter.

This has a white noise spectrum and gives unique values for multiple simultaneous processes… some simpler attempts did not achieve that, but there's probably a better way.

CHAPTER 16

Logger

The logger is nearly a direct copy from *rllab*, which implemented it as a module. It provides convenient recording of diagnostics to the terminal, which is also saved to *debug.log*, tabular diagnostics to comma-separated file, *progress.csv*, and training snapshot files (e.g. agent parameters), *params.pkl*. The logger is not extensively documented here; its usage is mostly exposed in the examples.

`rlpyt.utils.logging.context.`**`logger_context`**`(`*log_dir*, *run_ID*, *name*, *log_params=None*, *snapshot_mode='none'*, *override_prefix=False*, *use_summary_writer=False*`)`

Use as context manager around calls to the runner's `train()` method. Sets up the logger directory and filenames. Unless override_prefix is True, this function automatically prepends `log_dir` with the rlpyt logging directory and the date: *path-to-rlpyt/data/yyyymmdd/hhmmss* (*data/* is in the gitignore), and appends with */run_{run_ID}* to separate multiple runs of the same settings. Saves hyperparameters provided in `log_params` to *params.json*, along with experiment *name* and *run_ID*.

Input `snapshot_mode` refers to how often the logger actually saves the snapshot (e.g. may include agent parameters). The runner calls on the logger to save the snapshot at every iteration, but the input `snapshot_mode` sets how often the logger actually saves (e.g. snapshot may include agent parameters). Possible modes include (but check inside the logger itself):

- "none": don't save at all
- "last": always save and overwrite the previous
- "all": always save and keep each iteration
- "gap": save periodically and keep each (will also need to set the gap, not done here)

The cleanup operations after the `yield` close files but might not be strictly necessary if not launching another training session in the same python process.

`rlpyt.utils.logging.context.`**`add_exp_param`**`(`*param_name*, *param_val*, *exp_dir=None*, *overwrite=False*`)`

Puts a param in all experiments in immediate subdirectories. So you can write a new distinguising param after the fact, perhaps reflecting a combination of settings.

`rlpyt.utils.logging.context.`**`check_progress`**`(`*exp_dir=None*`)`

Print to stdout the number of lines in all `progress.csv` files in the directory. Call like:

```
python -c 'from rlpyt.util.logging.context import check_progress;
check_progress('path_to_dir')
```

CHAPTER 17

Creating and Launching Experiments

Some utilities are included for creating and launching experiments comprised of multiple individual learning runs, e.g. for hyperparameter sweeps. To date, these include functions for launching locally on a machine, so launching into the cloud may require different tooling. Many experiments can be queued on a given hardware resource, and they will be cycled through to run in sequence (e.g. a desktop with 4 GPUs and each run getting exclusive use of 2 GPUs).

17.1 Launching

`rlpyt.utils.launching.exp_launcher.`**`run_experiments`**(*script*, *affinity_code*, *experiment_title*, *runs_per_setting*, *variants*, *log_dirs*, *common_args=None*, *runs_args=None*, *set_egl_device=False*)

Call in a script to run a set of experiments locally on a machine. Uses the `launch_experiment()` function for each individual run, which is a call to the `script` file. The number of experiments to run at the same time is determined from the `affinity_code`, which expresses the hardware resources of the machine and how much resource each run gets (e.g. 4 GPU machine, 2 GPUs per run). Experiments are queued and run in sequence, with the intention to avoid hardware overlap. Inputs `variants` and `log_dirs` should be lists of the same length, containing each experiment configuration and where to save its log files (which have the same name, so can't exist in the same folder).

Hint: To monitor progress, view the *num_launched.txt* file and *experiments_tree.txt* file in the experiment root directory, and also check the length of each *progress.csv* file, e.g. `wc -l experiment-directory/.../run_*/progress.csv`.

`rlpyt.utils.launching.exp_launcher.`**`launch_experiment`**(*script*, *run_slot*, *affinity_code*, *log_dir*, *variant*, *run_ID*, *args*, *python_executable=None*, *set_egl_device=False*)

Launches one learning run using `subprocess.Popen()` to call the python script. Calls the script as:

```
python {script} {slot_affinity_code} {log_dir} {run_ID} {*args}
```

If `affinity_code["all_cpus"]` is provided, then the call is prepended with `tasket -c ..` and the listed cpus (this is the most sure way to keep the run limited to these CPU cores). Also saves the *variant* file. Returns the process handle, which can be monitored.

Use `set_egl_device=True` to set an environment variable `EGL_DEVICE_ID` equal to the same value as the cuda index for the algorithm. For example, can use with DMControl environment modified to look for this environment variable when selecting a GPU for headless rendering.

17.2 Variants

Some simple tools are provided for creating hyperparameter value variants.

class `rlpyt.utils.launching.variant.`**`VariantLevel`**(*keys*, *values*, *dir_names*)

A *namedtuple* which describes a set of hyperparameter settings.

Input `keys` should be a list of tuples, where each tuple is the sequence of keys to navigate down the configuration dictionary to the value.

Input `values` should be a list of lists, where each element of the outer list is a complete set of values, and position in the inner list corresponds to the key at that position in the keys list, i.e. each combination must be explicitly written.

Input `dir_names` should have the same length as `values`, and includeunique paths for logging results from each value combination.

`rlpyt.utils.launching.variant.`**`make_variants`**(**variant_levels*)

Takes in any number of `VariantLevel` objects and crosses them in order. Returns the resulting lists of full variant and log directories. Every set of values in one level is paired with every set of values in the next level, e.g. if two combinations are specified in one level and three combinations in the next, then six total configuations will result.

Use in the script to create and run a set of learning runs.

`rlpyt.utils.launching.variant.`**`_cross_variants`**(*prev_variants*, *prev_log_dirs*, *variant_level*)

For every previous variant, make all combinations with new values.

`rlpyt.utils.launching.variant.`**`load_variant`**(*log_dir*)

Loads the *variant.json* file from the directory.

`rlpyt.utils.launching.variant.`**`save_variant`**(*variant*, *log_dir*)

Saves a *variant.json* file to the directory.

`rlpyt.utils.launching.variant.`**`update_config`**(*default*, *variant*)

Performs deep update on all dict structures from `variant`, updating only individual fields. Any field in `variant` must be present in `default`, else raises `KeyError` (helps prevent mistakes). Operates recursively to return a new dictionary.

17.3 Affinity

The hardware affinity is used for several purposes: 1) the experiment launcher uses it to determine how many concurrent experiments to run, 2) runners use it to determine GPU device selection, 3) parallel samplers use it to determine the number of worker processes, and 4) multi-GPU and asynchronous runners use it to determine the number of parallel processes. The main intent of the implemented utilities is to take as input the total amount of hardware resources in the computer (CPU & GPU) and the amount of resources to be dedicated to each job, and then to divide resources evenly.

Example

An 8-GPU, 40-CPU machine would have 5 CPU assigned to each GPU. 1 GPU per run would set up 8 concurrent experiments, with each sampler using the 5 CPU. 2 GPU per run with synchronous runner would set up 4 concurrent experiments.

`rlpyt.utils.launching.affinity.`**`encode_affinity`**(*n_cpu_core=1*, *n_gpu=0*, *contexts_per_gpu=1*, *gpu_per_run=1*, *cpu_per_run=1*, *cpu_per_worker=1*, *cpu_reserved=0*, *hyperthread_offset=None*, *n_socket=None*, *run_slot=None*, *async_sample=False*, *sample_gpu_per_run=0*, *optim_sample_share_gpu=False*, *alternating=False*, *set_affinity=True*)

Encodes the hardware configuration into a string (with meanings defined in this file) which can be passed as a command line argument to call the training script. Use in overall experiments setup script to specify computer and experiment resources into `run_experiments()`.

We refer to an "experiment" as an individual learning run, i.e. one set of hyperparameters and which does not interact with other runs.

Parameters

- **`n_cpu_core`** (*`int`*) – Total number of phyical cores to use on machine (not virtual)
- **`n_gpu`** (*`int`*) – Total number of GPUs to use on machine
- **`contexts_per_gpu`** (*`int`*) – How many experiment to share each GPU
- **`gpu_per_run`** (*`int`*) – How many GPUs to use per experiment (for multi-GPU optimization)
- **`cpu_per_run`** (*`int`*) – If not using GPU, specify how macores per experiment
- **`cpu_per_worker`** (*`int`*) – CPU cores per sampler worker; 1 unless environment is multi-threaded
- **`cpu_reserved`** (*`int`*) – Number of CPUs to reserve per GPU, and not allow sampler to use them
- **`hyperthread_offset`** (*`int`*) – Typically the number of physical cores, since they are labeled 0-x, and hyperthreads as (x+1)-2x; use 0 to disable hyperthreads, None to auto-detect
- **`n_socket`** (*`int`*) – Number of CPU sockets in machine; tries to keep CPUs grouped on same socket, and match socket-to-GPU affinity
- **`run_slot`** (*`int`*) – Which hardware slot to use; leave `None` into `run_experiments()`, but specified for inidividual train script
- **`async_sample`** (*`bool`*) – True if asynchronous sampling/optimization mode; different affinity structure needed
- **`sample_gpu_per_run`** (*`int`*) – In asynchronous mode only, number of action-server GPUs per experiment

- **optim_sample_share_gpu** (*bool*) – In asynchronous mode only, whether to use same GPU(s) for both training and sampling
- **alternating** (*bool*) – True if using alternating sampler (will make more worker assignments)
- **set_affinity** (*bool*) – False to disable runner and sampler from setting cpu affinity via *psutil*, maybe inappropriate in cloud machines.

`rlpyt.utils.launching.affinity.`**`encode_affinity`**(*n_cpu_core=1*, *n_gpu=0*, *contexts_per_gpu=1*, *gpu_per_run=1*, *cpu_per_run=1*, *cpu_per_worker=1*, *cpu_reserved=0*, *hyperthread_offset=None*, *n_socket=None*, *run_slot=None*, *async_sample=False*, *sample_gpu_per_run=0*, *optim_sample_share_gpu=False*, *alternating=False*, *set_affinity=True*)

Encodes the hardware configuration into a string (with meanings defined in this file) which can be passed as a command line argument to call the training script. Use in overall experiments setup script to specify computer and experiment resources into `run_experiments()`.

We refer to an "experiment" as an individual learning run, i.e. one set of hyperparameters and which does not interact with other runs.

Parameters

- **n_cpu_core** (*int*) – Total number of phyical cores to use on machine (not virtual)
- **n_gpu** (*int*) – Total number of GPUs to use on machine
- **contexts_per_gpu** (*int*) – How many experiment to share each GPU
- **gpu_per_run** (*int*) – How many GPUs to use per experiment (for multi-GPU optimization)
- **cpu_per_run** (*int*) – If not using GPU, specify how macores per experiment
- **cpu_per_worker** (*int*) – CPU cores per sampler worker; 1 unless environment is multi-threaded
- **cpu_reserved** (*int*) – Number of CPUs to reserve per GPU, and not allow sampler to use them
- **hyperthread_offset** (*int*) – Typically the number of physical cores, since they are labeled 0-x, and hyperthreads as (x+1)-2x; use 0 to disable hyperthreads, None to auto-detect
- **n_socket** (*int*) – Number of CPU sockets in machine; tries to keep CPUs grouped on same socket, and match socket-to-GPU affinity
- **run_slot** (*int*) – Which hardware slot to use; leave `None` into `run_experiments()`, but specified for inidividual train script
- **async_sample** (*bool*) – True if asynchronous sampling/optimization mode; different affinity structure needed
- **sample_gpu_per_run** (*int*) – In asynchronous mode only, number of action-server GPUs per experiment
- **optim_sample_share_gpu** (*bool*) – In asynchronous mode only, whether to use same GPU(s) for both training and sampling

- **alternating** (*bool*) – True if using alternating sampler (will make more worker assignments)
- **set_affinity** (*bool*) – False to disable runner and sampler from setting cpu affinity via *psutil*, maybe inappropriate in cloud machines.

rlpyt.utils.launching.affinity.**make_affinity**(*run_slot=0*, ***kwargs*)
Input same kwargs as `encode_affinity()`, returns the AttrDict form.

rlpyt.utils.launching.affinity.**affinity_from_code**(*run_slot_affinity_code*)
Use in individual experiment script; pass output to Runner.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Index

Symbols