RLHive

Release 1.0.1

RLHive Authors

Jun 01, 2023

CONTENTS:

CHAPTER

ONE

QUICKSTART

1.1 Installation

RLHive is available through pip! For the basic RLHive package, simply run `pip install rlhive`.

You can also install dependencies necessary for the environments that RLHive comes with by running `pip install rlhive[<env_names>]` where `<env_names>` is a comma separated list made up of the following:

- atari
- gym_minigrid
- pettingzoo

In addition to these environments, Minatar and Marlgrid are also supported, but need to be installed separately.

To install Minatar, run `pip install MinAtar@git+https://github.com/kenjyoung/MinAtar.git@8b39a18a60248ede15ce70142b557f3897c4e1eb`

To install Marlgrid, run `pip install marlgrid@https://github.com/kandouss/marlgrid/archive/refs/heads/master.zip`

1.2 Running an experiment

There are several ways to run an experiment with RLHive. If you want to just run a preset config, you can directly run your experiment from the command line, with a config file path relative to the hive/configs folder. These examples run a DQN on the Atari game Asterix according to the Dopamine configuration and a simplified Rainbow agent for Hanabi trained using self-play according to the DeepMind's configuration

```
hive_single_agent_loop -p atari/dqn.yml
hive_multi_agent_loop -p hanabi/rainbow.yml
```

If you want to run an experiment with components that are all available in RLHive, but not presets, you can create your own config file, and run that instead! Make sure you look at the examples here and the tutorial *here* to properly format it:

```
hive_single_agent_loop -c <config-file>
hive_multi_agent_loop -c <config-file>
```

Finally, if instead you want to use your own custom custom components you can simply register it with RLHive and run your config normally:

```
import hive
from hive.runners.utils import load_config
from hive.runners.single_agent_loop import set_up_experiment

class CustomAgent(hive.agents.Agent):
    # Definition of Agent
    pass

hive.registry.register('CustomAgent', CustomAgent, CustomAgent)

# Either load your custom full config file with that includes CustomAgent
config = load_config(config='custom_full_config.yml')
runner = set_up_experiment(config)
runner.run_training()

# Or load a preset config and just replace the agent config
config = load_config(preset_config='atari/dqn.yml', agent_config='custom_agent_config.yml
↪')
runner = set_up_experiment(config)
runner.run_training()
```

CHAPTER

TWO

TUTORIALS

2.1 Agent

2.1.1 Agent API

Interacting with the agent happens primarily through two functions: `agent.act()` and `agent.update()`. `agent.act()` takes in an observation and returns an action, while `agent.update()` takes in a dictionary consisting of the information relevant to the most recent transition and updates the agent.

2.1.2 Creating an Agent

Let's create a new tabular Q-learning agent with discount factor `gamma` and learning rate `alpha`, for some environment with one hot observations. We want this agent to have an epsilon greedy policy, with the exploration rate decaying over `explore_steps` from `1.0` to some value `final_epsilon`. First, we define the constructor:

```
import numpy as np
import os

import hive
from hive.agents.agent import Agent
from hive.utils.schedule import LinearSchedule

class TabularQLearningAgent(Agent):
    def __init__(self, obs_dim, act_dim, gamma, alpha, explore_steps, final_epsilon,␣
↪id=0):
        super().__init__(obs_dim, act_dim, id=id)
        self._q_values = np.zeros(obs_dim, act_dim)
        self._gamma = gamma
        self._alpha = alpha
        self._epsilon_schedule = LinearSchedule(1.0, final_epsilon, explore_steps)
```

In this constructor, we created a numpy array to keep track of the Q-values for every state-action pair, and a linear decay schedule for the epsilon exploration rate. Next, let's create the act function:

```
def act(self, observation):
    # Return a random action if exploring
    if np.random.rand() < self._epsilon_schedule.update():
        return np.random.randint(self._act_dim)
    else:
        state = np.argmax(observation) # Convert from one-hot
```

(continues on next page)

(continued from previous page)

```
        # Break ties randomly between all actions with max values
        max_value = np.amax(self._q_values[state])
        best_actions = np.where(self._q_values[state] == max_value)[0]
        return np.random.choice(best_actions)
```

Now, we write our update function, which updates the state of our agent:

```
def update(self, update_info):
    state = np.argmax(update_info["observation"])
    next_state = np.argmax(update_info["next_observation"])

    self._q_values[state, update_info["action"]] += self._alpha * (
        update_info["reward"]
        + self._gamma * np.amax(self._q_values[next_state])
        - self._q_values[state, action]
    )
```

Now, we can directly use this environment with the single agent or multi-agent runners. Note `act` and `update` are framework agnostic, so you could implement it with any (deep) learning framework, although most of our implemented agents are written in PyTorch.

If we write a save and load function for this agent, we can also take advantage of checkpointing and resuming in the runner:

```
def save(self, dname):
    np.save(os.path.join(dname, "qvalues.npy"), self._q_values)
    pickle.dump({"schedule": self._epsilon_schedule}, open("state.p", "wb"))

def load(self, dname):
    self._q_values = np.load(os.path.join(dname, "qvalues.npy"))
    self._epsilon_schedule = pickle.load(open("state.p", "rb"))["schedule"]
```

Finally, we *register* our agent class, so that it can be found when setting up experiments through the yaml config files and command line.

```
hive.registry.register('TabularQLearningAgent', TabularQLearningAgent, Agent)
```

2.2 Configuration

RLHive was written to allow for fast configuration and iteration on configuration. The base configuration for all parameters for the experiment is done through YAML files. The majority of these parameters can be overriden through the command line.

Any object *registered* with RLHive can be configured directly through YAML files or through the command line, without having to add any extra argument parsers.

2.2.1 YAML files

Let's do a basic example of configuring an agent through YAML files. Specifically, let's create a `DQNAgent`.

```yaml
agent:
  name: DQNAgent
  kwargs:
    representation_net:
      name: MLPNetwork
      kwargs:
        hidden_units: [256, 256]
    discount_rate: .9
    replay_buffer:
      name: CircularReplayBuffer
    reward_clip: 1.0
```

In this example, `DQNAgent` , *MLPNetwork* , and *CircularReplayBuffer* are all classes registered with RLHive. Thus, we can do this configuration directly. When the `registry` getter function for agents, *get_agent()* is then called with this config dictionary (with the missing required arguments such as `obs_dim` and `act_dim`, filled in), it will build all the inner RLHive objects automatically. This works by using the type annotations on the constructors of the objects, so to recursively create the internal objects, those arguments need to be annotated correctly.

2.2.2 Overriding from command lines

When using the `registry` getter functions, RLHive automatically checks any command line arguments passed to see if they match/override any default or yaml configured arguments. With `getter` functionyou provide a config and a prefix. That prefix is added prepended to any argument names when searching the command line. For example, with the above config, if it were loaded and the *get_agent()* method was called as follows:

```
agent = get_agent(config['agent'], 'ag')
```

Then, to override the discount_rate, you could pass the following argument to your python script: `--ag.discount_rate .95`. This can go arbitrarily deep into registered RLHive class. For example, if you wanted to change the capacity of the replay buffer, you could pass `--ag.replay_buffer.capacity 100000`.

If the type annotation the argument `arg` is `List[C]` where C is a registered RLHive class, then you can override the argument of an individual object, `foo`, configured through YAML by passing `--arg.0.foo <value>`.

Note as of this version, you must have configured the object in the YAML file in order to override its parameters through the command line.

2.3 Using the DQN/Rainbow Agents

The *DQNAgent* and *RainbowDQNAgent* are written to allow for easy extensions and adaptation to your applications. We outline a few different use cases here.

2.3.1 Using a different network architecture

Using different types of network architectures with *DQNAgent* and *RainbowDQNAgent* is done using the `representation_net` parameter in the constructor. This network should not include the final layer which computes the final Q-values. It computes the representations that are fed into the layer which will compute the final Q-values. This is because often the only difference between different variations of the DQN algorithms is how the final Q-values are computed, with the rest of the architecture not changing.

You can modify the architecture of the representation network from the config, or create a completely new architecture better suited to your needs. From the config, two different types of network architectures are supported:

- *ConvNetwork*: Networks with convolutional layers, followed by an MLP
- *MLPNetwork*: An MLP with only linear layers

See *this page* for details on how to configure the network.

To use an architecture not supported by the above classes, simply write the Pytorch module implementing the architecture, and register the class wrapped with *FunctionApproximator* wrapper. The only requirement is that this class should take in the input dimension as the first positional argument:

```
import torch

import hive
from hive.agents.qnets import FunctionApproximator

class CustomArchitecture(torch.nn.Module):
    def __init__(self, in_dim, hidden_units):
        super().__init__()
        self.network = torch.nn.Sequential(
            torch.nn.Linear(in_dim, hidden_units),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_units, hidden_units)
        )

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        return self.network(x)

hive.registry.register(
    'CustomArchitecture',
    FunctionApproximator(CustomArchitecture),
    FunctionApproximator
)
```

2.3.2 Adding in different Rainbow components

The Rainbow architecture is composed of several different components, namely:

- Double Q-learning
- Prioritized Replay
- Dueling Networks
- Multi-step Learning
- Distributional RL

- Noisy Networks

Each of these components can be independently used with our *RainbowDQNAgent* class. To use Prioritized Replay, you must pass a *PrioritizedReplayBuffer* to the `replay_buffer` parameter of *RainbowDQNAgent*. The details for how to use the other components of rainbow are found in the API documentation of *RainbowDQNAgent*.

2.3.3 Custom Input Observations

The current implementations of *DQNAgent* and *RainbowDQNAgent* handle the standard case of observations being a single numpy array, and no extra inputs being necessary during the update phase other than `action`, `reward`, and `done`. In the situation where this is not the case, and you need to handle more complex inputs, you can do so by overriding the methods of *DQNAgent*. Let's walk through the example of *LegalMovesRainbowAgent*. This agent takes in a list of legal moves on each turn and only selects from those.

```
class LegalMovesHead(torch.nn.Module):
    def __init__(self, base_network):
        super().__init__()
        self.base_network = base_network

    def forward(self, x, legal_moves):
        x = self.base_network(x)
        return x + legal_moves

    def dist(self, x, legal_moves):
        return self.base_network.dist(x)

class LegalMovesRainbowAgent(RainbowDQNAgent):
    """A Rainbow agent which supports games with legal actions."""

    def create_q_networks(self, representation_net):
        """Creates the qnet and target qnet."""
        super().create_q_networks(representation_net)
        self._qnet = LegalMovesHead(self._qnet)
        self._target_qnet = LegalMovesHead(self._target_qnet)
```

This defines a wrapper around the Q-networks used by agent that takes an encoding of the legal moves where illegal moves have value $-\infty$ and legal moves have value 0. The wrapper then adds this encoding to the values generated by the base Q-networks. Overriding *create_q_networks()* allows you to modify the base Q-networks by adding this wrapper.

```
def preprocess_update_batch(self, batch):
    for key in batch:
        batch[key] = torch.tensor(batch[key], device=self._device)
    return (
        (batch["observation"], batch["action_mask"]),
        (batch["next_observation"], batch["next_action_mask"]),
        batch,
    )
```

Now, since the Q-networks expect an extra parameter (the legal moves action mask), we override the *preprocess_update_batch()* method, which takes a batch sampled from the replay buffer and defines the inputs that will be used to compute the values of the current state and the next state during the update step.

```
def preprocess_update_info(self, update_info):
    preprocessed_update_info = {
        "observation": update_info["observation"]["observation"],
        "action": update_info["action"],
        "reward": update_info["reward"],
        "done": update_info["done"],
        "action_mask": action_encoding(update_info["observation"]["action_mask"]),
    }
    if "agent_id" in update_info:
        preprocessed_update_info["agent_id"] = int(update_info["agent_id"])
    return preprocessed_update_info
```

We must also make sure that the action encoding for each transition is added to the replay buffer in the first place. To do that, we override the *preprocess_update_info()* method, which should return a dictionary with keys and values corresponding to the items you wish to store into the replay buffer. Note, these keys need to be specified when you create the replay buffer, see *Replays* for more information.

```
@torch.no_grad()
def act(self, observation):
    if self._training:
        if not self._learn_schedule.get_value():
            epsilon = 1.0
        elif not self._use_eps_greedy:
            epsilon = 0.0
        else:
            epsilon = self._epsilon_schedule.update()
        if self._logger.update_step(self._timescale):
            self._logger.log_scalar("epsilon", epsilon, self._timescale)
    else:
        epsilon = self._test_epsilon

    vectorized_observation = torch.tensor(
        np.expand_dims(observation["observation"], axis=0), device=self._device
    ).float()
    legal_moves_as_int = [
        i for i, x in enumerate(observation["action_mask"]) if x == 1
    ]
    encoded_legal_moves = torch.tensor(
        action_encoding(observation["action_mask"]), device=self._device
    ).float()
    qvals = self._qnet(vectorized_observation, encoded_legal_moves).cpu()

    if self._rng.random() < epsilon:
        action = np.random.choice(legal_moves_as_int).item()
    else:
        action = torch.argmax(qvals).item()

    return action
```

Finally, you also need to override the *act()* method to extract and use the extra information.

2.4 Environments

2.4.1 Installing Environments

We support several environments in RLHive, namely:

- Atari
- Gym classic control
- Minatar (simplified Atari)
- Minigrid (single-agent grid world)
- Marlgrid (multi-agent)
- Pettingzoo (multi-agent)

While gym comes installed with the base package, you need to install the other environments. See *Installation* for more details.

2.4.2 Creating an Environment

RLHive Environments

Every environment used in RLHive should be a subclass of *~hive.envs.base.BaseEnv*. It should provide a `reset` function that resets the environment to a new episode and returns a tuple of `(observation, turn)` and a `step` function that takes in an action, performs the step in the environment, and returns a tuple of `(observation, reward, done, turn, info)`. All these values correspond to their canonical meanings, and `turn` corresponds to the index of the agent whose turn it is (in multi-agent environments).

The `reward` return value can be a single number, an array, or a dictionary. If it's a number, then that same reward will be given to every single agent. If it's an array, the agents get the reward corresponding to their index in the runner. If it's a dictionary, the keys should be the agent ids, and the value the reward for that agent.

Each environment should also provide an *EnvSpec* environment that will provide information about the environment such as the expected observation shape and action dimension for each agent. These should be lists with one element for each agent. See `GymEnv` for an example.

Gym environments

If your environment is a gym environment, and you do not need to preprocess the observations generated by the environment, then you can directly use the `GymEnv`. Just make sure you register your environment with `gym`, and pass the name of the environment to the `GymEnv` constructor.

If you need to add extra preprocessing or change the default way that environment/EnvSpec creation is done, you can simply subclass this class and override either `create_env()` and/or `create_env_spec()`, as in `AtariEnv`.

Parallel Step Environments

Multi-agent environments usually come in two flavors: sequential step environments, where each agent takes it's action one at a time, and parallel step environments, where each agent steps at the same time. The *MultiAgentRunner* class expects only sequential step environments. Fortunately, we can convert between parallel step environments and single step environments by simply generating the action for each agent one at a time and passing all the action to the parallel step environment all at once. To facilitate this, we provide a utility class *ParallelEnv*. Simply write the logic for your parallel step environment as normal, and then create a single step version of the environment by subclassing *ParallelEnv* and the parallel step environment, **making sure to put *ParallelEnv* first in the superclass list.**

```
from hive.envs.base import BaseEnv, ParallelEnv

class ParallelStepEnvironment(BaseEnv):
    # Write the logic needed for the parallel step environment. Assume the step
    # function gets an array actions as it's input, and should return an array
    # containing the observations for each agent, as well as the other return
    # values expected by the environment.

class SequentialStepEnvironment(ParallelEnv, ParallelStepEnvironment):
    # Any other logic needed to create the environmnet.
```

2.5 Loggers

2.5.1 Using a Logger

RLHive currently provides 3 types of loggers:

- *ChompLogger*: Logs metrics to a dictionary like object that can be saved.
- *WandbLogger*: Logs metrics to WandB.
- *NullLogger*: Does not log metrics.

All of these loggers are `ScheduledLoggers`. When creating these loggers, you provide the different timescales that you want to track with the logger. Timescales could correspond to any loop variable, such as `"train_step"`, `"agent_step"`, `"test_iteration"`, etc.

With `ScheduledLoggers`, each timescale is associated with a *Schedule* object. By updating these schedules in some loop, you can control the logging frequency for that timescale. The loggers also keep track of how many times that timescale was updated, and those values are logged alongside any metric you log (i.e. if `"timescale_1"` was updated 7 times, then the logger will log an additional key value pair of `"timescale_1": 7` with each metric. This allows you to see the trends of any metric across any timescale.

Timescales can be registered when creating the logger, or later on.

For an example:

```
from hive.utils.loggers import ChompLogger, CompositeLogger, WandbLogger
from hive.utils.schedule import ConstantSchedule, PeriodicSchedule

logger = ChompLogger(
    ['train_step', 'test_step'], # Timescales to track
    [ # How often to log train_step and test_step
        PeriodicSchedule(False, True, 10), # train_step is logged once every 10 times
        ConstantSchedule(True) # test_step is always logged
```

(continues on next page)

(continued from previous page)

```
    ]
)

# You can register timescales after logger creation as well. This particular timescale
# is never scheduled to log.
logger.register_timescale('dummy_timescale', ConstantSchedule(False))

for _ in range(total_training_time):
    metrics_to_log = run_training_step()
    if logger.update_step('train_step'): # Evaluates to True once every 10 times it's hit.
        logger.log_metrics(metrics_to_log, 'training_metrics')


    other_metric = do_something_else()
    if logger.should_log('train_step'): # Checks schedule for this timescale without␣
↪updating
        logger.log_scalar('other_metric', other_metric, 'training_metrics')


    if should_run_testing():
        test_metrics = run_testing()
        if logger.update_step('tets_step'): # Evaluates to True every time
            logger.log_metrics(test_metrics, 'testing_metrics')
```

Note, the schedules associated with each timescale are not hard constraints on when you can log. They are merely a convenience to help you keep track of when to log.

2.5.2 Composite Logger

If you want to log to multiple sources, without the hassle of keeping track of multiple loggers, you can create a *CompositeLogger* object initialized with each of the individual loggers that you want to use. For example:

```
from hive.utils.loggers import ChompLogger, CompositeLogger, WandbLogger

logger = CompositeLogger([
    ChompLogger('train'),
    WandbLogger('train')
])
```

You can now use this logger as above, and it will log to both *ChompLogger* and *WandbLogger*.

2.6 Registration

2.6.1 Registering new Classes

To register a new class with the Registry, you need to make sure that it or one of its ancestors subclassed *hive.utils.registry.Registrable* and provided a definition for *hive.utils.registry.Registrable.type_name()*. The return value of this function is used to create the getter function for that type (`registry.get_{type_name}`).

You can either register several different classes at once:

```
registry.register_all(
    Agent,
    {
        "DQNAgent": DQNAgent,
        "LegalMovesRainbowAgent": LegalMovesRainbowAgent,
        "RainbowDQNAgent": RainbowDQNAgent,
        "RandomAgent": RandomAgent,
    },
)
```

or one at a time:

```
registry.register("DQNAgent", DQNAgent, Agent)
registry.register("LegalMovesRainbowAgent", LegalMovesRainbowAgent, Agent)
registry.register("RainbowDQNAgent", RainbowDQNAgent, Agent)
registry.register("RandomAgent", RandomAgent, Agent)
```

After a class has been registered, you can use pass a config dictionary to the getter function for that type to create the object.

2.6.2 Callables

There are several cases where we want to parameterize some function or constructor partway, but not pass the fully created object in as an argument. One example is optimizers. You might want to pass a learning rate, but you cannot create the final optimizer object until you've created the parameters you want to optimize. To deal with such cases, we provide a `CallableType` class, which can be used to register and wrap any callable. For example, with optimizers, we have:

```
class OptimizerFn(CallableType):
    """A wrapper for callables that produce optimizer functions.

    These wrapped callables can be partially initialized through configuration
    files or command line arguments.
    """

    @classmethod
    def type_name(cls):
        """
        Returns:
            "optimizer_fn"
        """
        return "optimizer_fn"

registry.register_all(
    OptimizerFn,
    {
        "Adadelta": OptimizerFn(optim.Adadelta),
        "Adagrad": OptimizerFn(optim.Adagrad),
        "Adam": OptimizerFn(optim.Adam),
        "Adamax": OptimizerFn(optim.Adamax),
        "AdamW": OptimizerFn(optim.AdamW),
        "ASGD": OptimizerFn(optim.ASGD),
```

(continues on next page)

(continued from previous page)

```
        "LBFGS": OptimizerFn(optim.LBFGS),
        "RMSprop": OptimizerFn(optim.RMSprop),
        "RMSpropTF": OptimizerFn(RMSpropTF),
        "Rprop": OptimizerFn(optim.Rprop),
        "SGD": OptimizerFn(optim.SGD),
        "SparseAdam": OptimizerFn(optim.SparseAdam),
    },
)
```

With this, we can now make use of the configurability of RLHive objects while still passing callables as arguments.

2.7 Replays

RLHive currently provides 4 types of Replays:

- *CircularReplayBuffer*: An implementation of a FIFO circular replay buffer. Stores individual observations and constructs transitions on the fly when sampling to save space.
- *SimpleReplayBuffer*: A simplified version of a FIFO circular replay buffer that stores individual transitions directly.
- *PrioritizedReplayBuffer*: A subclass of *CircularReplayBuffer* that adds prioritized sampling.
- `LegalMovesReplayBuffer`: A subclass of *PrioritizedReplayBuffer* that stores/handles legal moves.

The main replay buffer classes that you will likely use/extend are *CircularReplayBuffer* and *PrioritizedReplayBuffer*. By default, these classes expect the arguments `"observation"`, `"action"`, `"reward"`, and `"done"` when adding to the buffer. You can also provide alternative shapes/dtypes for these keys, and the buffer will try to automatically cast the objects you add to the buffer.

Along with these default keys, you can also store extra keys in the buffer. When creating the buffer, provide a dictionary with key-value pairs `key: (type, shape)`. When adding, you can directly provide this key as an argument to the *add()* method, and it will automatically be added to the batch dictionary that you sample.

2.8 Runners

We provide two different *Runner* classes: *SingleAgentRunner* and *MultiAgentRunner*. The setup for both Runner classes can be viewed in their respective files with the `set_up_experiment()` functions. The *get_parsed_args()* function can be used to get any arguments from the command line are not part of the signatures of already registered RLHive class constructors.

2.8.1 Metrics and TransitionInfo

The *Metrics* class can be used to keep track of metrics for single/multiple agents across an episode.

```
# Create the Metrics object. The first set of metrics is individual to
# each agent, the second is common for all agents. The metrics can
# be initialized either with a value or with callable with no arguments
metrics = Metrics(
    [agent1, agent2],
    [("reward", 0), ("episode_traj", lambda: [])],
```

(continues on next page)

(continued from previous page)

```
    [("full_episode_length", 0)],
)

# Add metrics
metrics[agent1.id]["reward"] += 1
metrics[agent2.id]["episode_traj"].append(0)
metrics["full_episode_length"] += 1

# Convert to flat dictionary for easy logging. Adds agent id's as prefixes
# for agent_specific metrics
flat_metrics = metrics.get_flat_dict()

# Reinitialize/reset all metrics
metrics.reset_metrics()
```

The *TransitionInfo* class can be used to keep track of the information needed by the agent to construct it's next state for acting or next transition for updating. It also handles state stacking and padding.

```
transition_info = TransitionInfo([agent1, agent2], stack_size)

# Set the start flag for agent1.
transition_info.start(agent1)

# Get stacked observation for agent1. If not enough observations have been
# recorded, it will pad with 0s
stacked_observation = transition_info.get_stacked_state(
    agent1, observation
)

# Record update information about the agent
transition_info.record_info(agent, info)

# Get the update information for the agent, with done set to the value passed
info = transition_info.get_info(agent, done=done)
```

CHAPTER

THREE

RLHIVE API

3.1 hive package

3.1.1 Subpackages

hive.agents package

Subpackages

hive.agents.qnets package

Subpackages

hive.agents.qnets.atari package

Submodules

hive.agents.qnets.atari.nature_atari_dqn module

Module contents

Submodules

hive.agents.qnets.base module

class hive.agents.qnets.base.**FunctionApproximator**

Bases: *Registrable*

A wrapper for callables that produce function approximators.

For example, `FunctionApproximator(create_neural_network)` or `FunctionApproximator(MyNeuralNetwork)` where `create_neural_network` is a function that creates a neural network module and `MyNeuralNetwork` is a class that defines your function approximator.

These wrapped callables can be partially initialized through configuration files or command line arguments.

classmethod type_name()

Returns
"function"

hive.agents.qnets.conv module

class hive.agents.qnets.conv.**ConvNetwork**(*in_dim*, *channels=None*, *mlp_layers=None*, *kernel_sizes=1*, *strides=1*, *paddings=0*, *normalization_factor=255*, *noisy=False*, *std_init=0.5*)

Bases: Module

Basic convolutional neural network architecture. Applies a number of convolutional layers (each followed by a ReLU activation), and then feeds the output into an *hive.agents.qnets.mlp.MLPNetwork*.

Note, if `channels` is `None`, the network created for the convolution portion of the architecture is simply an `torch.nn.Identity` module. If `mlp_layers` is `None`, the mlp portion of the architecture is an `torch.nn.Identity` module.

Parameters

- **in_dim** (*tuple*) – The tuple of observations dimension (channels, width, height).
- **channels** (*list*) – The size of output channel for each convolutional layer.
- **mlp_layers** (*list*) – The number of neurons for each mlp layer after the convolutional layers.
- **kernel_sizes** (*list* | *int*) – The kernel size for each convolutional layer
- **strides** (*list* | *int*) – The stride used for each convolutional layer.
- **paddings** (*list* | *int*) – The size of the padding used for each convolutional layer.
- **normalization_factor** (*float* | *int*) – What the input is divided by before the forward pass of the network.
- **noisy** (*bool*) – Whether the MLP part of the network will use *NoisyLinear* layers or `torch.nn.Linear` layers.
- **std_init** (*float*) – The range for the initialization of the standard deviation of the weights in *NoisyLinear*.

training: bool

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

hive.agents.qnets.mlp module

class hive.agents.qnets.mlp.**MLPNetwork**(*in_dim*, *hidden_units=256*, *activation_fn=None*, *noisy=False*, *std_init=0.5*, *initialization_fn=<function MLPNetwork.<lambda>>*)

Bases: Module

Basic MLP neural network architecture.

Contains a series of torch.nn.Linear or *NoisyLinear* layers, each of which is followed by a ReLU.

Parameters

- **in_dim** (*tuple[int]*) – The shape of input observations.
- **hidden_units** (*int | list[int]*) – The number of neurons for each mlp layer.
- **noisy** (*bool*) – Whether the MLP should use *NoisyLinear* layers or normal torch.nn.Linear layers.
- **std_init** (*float*) – The range for the initialization of the standard deviation of the weights in *NoisyLinear*.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

hive.agents.qnets.noisy_linear module

class hive.agents.qnets.noisy_linear.**NoisyLinear**(*in_dim*, *out_dim*, *std_init=0.5*)

Bases: Module

NoisyLinear Layer. Implements the layer described in https://arxiv.org/abs/1706.10295.

Parameters

- **in_dim** (*int*) – The dimension of the input.
- **out_dim** (*int*) – The desired dimension of the output.
- **std_init** (*float*) – The range for the initialization of the standard deviation of the weights.

forward(*inp*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

hive.agents.qnets.qnet_heads module

class hive.agents.qnets.qnet_heads.**DQNNetwork**(*base_network*, *hidden_dim*, *out_dim*, *linear_fn=None*)

Bases: Module

Implements the standard DQN value computation. Transforms output from base_network with output dimension hidden_dim to dimension out_dim, which should be equal to the number of actions.

Parameters

- **base_network** (*torch.nn.Module*) – Backbone network that computes the representations that are used to compute action values.
- **hidden_dim** (*int*) – Dimension of the output of the network.
- **out_dim** (*int*) – Output dimension of the DQN. Should be equal to the number of actions that you are computing values for.
- **linear_fn** (*torch.nn.Module*) – Function that will create the torch.nn.Module that will take the output of network and produce the final action values. If None, a torch.nn.Linear layer will be used.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class hive.agents.qnets.qnet_heads.**DuelingNetwork**(*base_network*, *hidden_dim*, *out_dim*, *linear_fn=None*, *atoms=1*)

Bases: Module

Computes action values using Dueling Networks (https://arxiv.org/abs/1511.06581). In dueling, we have two heads—one for estimating advantage function and one for estimating value function.

Parameters

- **base_network** (*torch.nn.Module*) – Backbone network that computes the representations that are shared by the two estimators.
- **hidden_dim** (*int*) – Dimension of the output of the base_network.
- **out_dim** (*int*) – Output dimension of the Dueling DQN. Should be equal to the number of actions that you are computing values for.
- **linear_fn** (*torch.nn.Module*) – Function that will create the torch.nn.Module that will take the output of network and produce the final action values. If None, a torch.nn.Linear layer will be used.
- **atoms** (*int*) – Multiplier for the dimension of the output. For standard dueling networks, this should be 1. Used by *DistributionalNetwork*.

init_networks()

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class hive.agents.qnets.qnet_heads.**DistributionalNetwork**(*base_network*, *out_dim*, *vmin=0*, *vmax=200*, *atoms=51*)

Bases: `Module`

Computes a categorical distribution over values for each action (https://arxiv.org/abs/1707.06887).

Parameters

- **base_network** (*torch.nn.Module*) – Backbone network that computes the representations that are used to compute the value distribution.
- **out_dim** (*int*) – Output dimension of the Distributional DQN. Should be equal to the number of actions that you are computing values for.
- **vmin** (*float*) – The minimum of the support of the categorical value distribution.
- **vmax** (*float*) – The maximum of the support of the categorical value distribution.
- **atoms** (*int*) – Number of atoms discretizing the support range of the categorical value distribution.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

dist(*x*)

Computes a categorical distribution over values for each action.

training: bool

hive.agents.qnets.rnn module

hive.agents.qnets.sequence_models module

class hive.agents.qnets.sequence_models.**SequenceFn**

Bases: *Registrable*, Module

A wrapper for callables that produce sequence functions.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod type_name()

This should represent a string that denotes the which type of class you are creating. For example, "logger", "agent", or "env".

abstract init_hidden(*batch_size*)

get_hidden_spec()

training: bool

class hive.agents.qnets.sequence_models.**LSTMModel**(*rnn_input_size*, *rnn_hidden_size=128*, *num_rnn_layers=1*, *batch_first=True*)

Bases: *SequenceFn*

A multi-layer long short-term memory (LSTM) RNN.

Parameters

- **rnn_input_size** (*int*) – The number of expected features in the input x.
- **rnn_hidden_size** (*int*) – The number of features in the hidden state h.
- **num_rnn_layers** (*int*) – Number of recurrent layers.
- **batch_first** (*bool*) – If True, then the input and output tensors are
- **as** (*provided*) –

forward(*x*, *hidden_state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

init_hidden(*batch_size*)

get_hidden_spec()

training: bool

class hive.agents.qnets.sequence_models.**GRUModel**(*rnn_input_size*, *rnn_hidden_size=128*, *num_rnn_layers=1*, *batch_first=True*)

Bases: *SequenceFn*

A multi-layer gated recurrent unit (GRU) RNN.

Parameters

- **rnn_input_size** (*int*) – The number of expected features in the input x.
- **rnn_hidden_size** (*int*) – The number of features in the hidden state h.
- **num_rnn_layers** (*int*) – Number of recurrent layers.
- **batch_first** (*bool*) – If True, then the input and output tensors are
- **as** (*provided*) –

forward(*x*, *hidden_state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

init_hidden(*batch_size*)

get_hidden_spec()

training: bool

class hive.agents.qnets.sequence_models.**SequenceModel**(*in_dim*, *representation_network*, *sequence_fn*, *mlp_layers=None*, *normalization_factor=255*, *noisy=False*, *std_init=0.5*)

Bases: *Registrable*, Module

Basic convolutional recurrent neural network architecture. Applies a number of convolutional layers (each followed by a ReLU activation), recurrent layers, and then feeds the output into an *hive.agents.qnets.mlp.MLPNetwork*.

Note, if `channels` is `None`, the network created for the convolution portion of the architecture is simply an `torch.nn.Identity` module. If `mlp_layers` is `None`, the mlp portion of the architecture is an `torch.nn.Identity` module.

Parameters

- **in_dim** (*tuple*) – The tuple of observations dimension (channels, width, height).
- **sequence_fn** (SequenceFn) – A sequence neural network that learns recurrent representation. Usually placed between the convolutional layers and mlp layers.
- **normalization_factor** (*float | int*) – What the input is divided by before the forward pass of the network.
- **noisy** (*bool*) – Whether the MLP part of the network will use *NoisyLinear* layers or `torch.nn.Linear` layers.
- **std_init** (*float*) – The range for the initialization of the standard deviation of the weights in *NoisyLinear*.

classmethod type_name()

This should represent a string that denotes the which type of class you are creating. For example, "logger", "agent", or "env".

forward(*x*, *hidden_state=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_hidden_spec()

training: bool

class hive.agents.qnets.sequence_models.**DRQNNetwork**(*base_network*, *hidden_dim*, *out_dim*, *linear_fn=None*)

Bases: Module

Implements the standard DRQN value computation. This module returns two outputs, which correspond to the two outputs from `base_network`. In particular, it transforms the first output from `base_network` with output dimension `hidden_dim` to dimension `out_dim`, which should be equal to the number of actions. The second output of this module is the second output from `base_network`, which is the hidden state that will be used as the initial hidden state when computing the next action in the trajectory.

Parameters

- **base_network** (*torch.nn.Module*) – Backbone network that returns two outputs, one is the representation used to compute action values, and the other one is the hidden state used as input hidden state later.
- **hidden_dim** (*int*) – Dimension of the output of the `network`.
- **out_dim** (*int*) – Output dimension of the DRQN. Should be equal to the number of actions that you are computing values for.
- **linear_fn** (*torch.nn.Module*) – Function that will create the `torch.nn.Module` that will take the output of `network` and produce the final action values. If `None`, a `torch.nn.Linear` layer will be used.

forward(*x*, *hidden_state=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_hidden_spec()

training: bool

hive.agents.qnets.td3_heads module

class hive.agents.qnets.td3_heads.**TD3ActorNetwork**(*representation_network*, *actor_net*, *network_output_shape*, *action_shape*, *use_tanh=True*)

Bases: Module

A module that implements the TD3 actor computation. It puts together the `representation_network` and `actor_net`, and adds a final `Linear` layer to compute the action.

Parameters

- **representation_network** (*torch.nn.Module*) – Network that encodes the observations.
- **actor_net** (FunctionApproximator) – Function that takes in the shape of the encoded observations and creates a network. This network takes the encoded observations from representation_net and outputs the representations used to compute the actions (ie everything except the last layer).
- **network_output_shape** (Union[int, Tuple[int]]) – Expected output shape of the representation network.
- **action_shape** (Tuple[int]) – Requiured shape of the output action.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class hive.agents.qnets.td3_heads.**TD3CriticNetwork**(*representation_network*, *critic_net*, *network_output_shape*, *n_critics*, *action_shape*)

Bases: Module

Parameters

- **representation_network** (*torch.nn.Module*) – Network that encodes the observations.
- **critic_net** (FunctionApproximator) – Function that takes in the shape of the encoded observations and creates a network. This network takes two inputs: the encoded observations from representation_net and actions. It outputs the representations used to compute the values of the actions (ie everything except the last layer).
- **network_output_shape** (Union[int, Tuple[int]]) – Expected output shape of the representation network.
- **n_critics** (int) – How many copies of the critic to create. They will all use the shared representation from the representation_network.
- **action_shape** (Tuple[int]) – Expected shape of actions.

forward(*obs*, *actions*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

q1(*obs*, *actions*)

Returns the value according to only the first critic.

training: *bool*

hive.agents.qnets.utils module

hive.agents.qnets.utils.**calculate_output_dim**(*net*, *input_shape*)

Calculates the resulting output shape for a given input shape and network.

Parameters

- **net** (*torch.nn.Module*) – The network which you want to calculate the output dimension for.
- **input_shape** (*int | tuple[int]*) – The shape of the input being fed into the `net`. Batch dimension should not be included.

Returns

The shape of the output of a network given an input shape. Batch dimension is not included.

hive.agents.qnets.utils.**apply_to_tensor**(*x*, *fn*)

Applies a function to a tensor or a tuple/list of tensors.

Parameters

- **x** (*torch.Tensor | tuple | list | dict*) – The tensor or tuple/list/dict of tensors to apply the function to.
- **fn** (*callable*) – The function to apply to the tensor or tuple/list/dict of tensors.

Returns

The result of applying the function to the tensor or tuple/list/dict of tensors.

hive.agents.qnets.utils.**create_init_weights_fn**(*initialization_fn*)

Returns a function that wraps `initialization_function()` and applies it to modules that have the `weight` attribute.

Parameters

initialization_fn (*callable*) – A function that takes in a tensor and initializes it.

Returns

Function that takes in PyTorch modules and initializes their weights. Can be used as follows:

```
init_fn = create_init_weights_fn(variance_scaling_)
network.apply(init_fn)
```

hive.agents.qnets.utils.**calculate_correct_fan**(*tensor*, *mode*)

Calculate fan of tensor.

Parameters

- **tensor** (*torch.Tensor*) – Tensor to calculate fan of.

- **mode** (*str*) – Which type of fan to compute. Must be one of *"fan_in"*, *"fan_out"*, and *"fan_avg"*.

Returns
Fan of the tensor based on the mode.

hive.agents.qnets.utils.**variance_scaling_**(*tensor*, *scale=1.0*, *mode='fan_in'*, *distribution='uniform'*)

Implements the `tf.keras.initializers.VarianceScaling` initializer in PyTorch.

Parameters

- **tensor** (*torch.Tensor*) – Tensor to initialize.
- **scale** (*float*) – Scaling factor (must be positive).
- **mode** (*str*) – Must be one of *"fan_in"*, *"fan_out"*, and *"fan_avg"*.
- **distribution** – Random distribution to use, must be one of "truncated_normal", "untruncated_normal" and "uniform".

Returns
Initialized tensor.

class hive.agents.qnets.utils.**InitializationFn**

Bases: *Registrable*

A wrapper for callables that produce initialization functions.

These wrapped callables can be partially initialized through configuration files or command line arguments.

classmethod type_name()

Returns
"init_fn"

Module contents

Submodules

hive.agents.agent module

class hive.agents.agent.**Agent**(*observation_space*, *action_space*, *id=0*)

Bases: ABC, *Registrable*

Base class for agents. Every implemented agent should be a subclass of this class.

Parameters

- **observation_space** (*gym.Space*) – Observation space for agent.
- **action_space** (*gym.Space*) – Action space for agent.
- **id** – Identifier for the agent.

property id

abstract act(*observation*, *agent_traj_state*)

Returns an action for the agent to perform based on the observation.

Parameters

- **observation** – Current observation that agent should act on.

- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- Action for the current timestep.
- Agent trajectory state.

abstract update(*update_info*, *agent_traj_state*)

Updates the agent.

Parameters

- **update_info** (*dict*) – Contains information from the environment agent needs to update itself.
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

Agent trajectory state.

train()

Changes the agent to training mode.

eval()

Changes the agent to evaluation mode

abstract save(*dname*)

Saves agent checkpointing information to file for future loading.

Parameters

dname (*str*) – directory where agent should save all relevant info.

abstract load(*dname*)

Loads agent information from file.

Parameters

dname (*str*) – directory where agent checkpoint info is stored.

classmethod type_name()

Returns

"agent"

hive.agents.ddpg module

class hive.agents.ddpg.**DDPG**(*observation_space*, *action_space*, *representation_net=None*, *actor_net=None*, *critic_net=None*, *init_fn=None*, *actor_optimizer_fn=None*, *critic_optimizer_fn=None*, *critic_loss_fn=None*, *stack_size=1*, *replay_buffer=None*, *discount_rate=0.99*, *n_step=1*, *grad_clip=None*, *reward_clip=None*, *soft_update_fraction=0.005*, *batch_size=64*, *logger=None*, *log_frequency=100*, *update_frequency=1*, *action_noise=0*, *min_replay_history=1000*, *device='cpu'*, *id=0*)

Bases: *TD3*

An agent implementing the DDPG algorithm. It is implemented by fixing the n_critics, policy_update_frequency, target_noise, and target_noise_clip parameters of the *TD3* agent.

Parameters

- **observation_space** (*gym.spaces.Box*) – Observation space for the agent.
- **action_space** (*gym.spaces.Box*) – Action space for the agent.
- **representation_net** (FunctionApproximator) – The network that encodes the observations that are then fed into the actor_net and critic_net. If None, defaults to Identity.
- **actor_net** (FunctionApproximator) – The network that takes the encoded observations from representation_net and outputs the representations used to compute the actions (ie everything except the last layer).
- **critic_net** (FunctionApproximator) – The network that takes two inputs: the encoded observations from representation_net and actions. It outputs the representations used to compute the values of the actions (ie everything except the last layer).
- **init_fn** (InitializationFn) – Initializes the weights of agent networks using create_init_weights_fn.
- **actor_optimizer_fn** (OptimizerFn) – A function that takes in the list of parameters of the actor returns the optimizer for the actor. If None, defaults to Adam.
- **critic_optimizer_fn** (OptimizerFn) – A function that takes in the list of parameters of the critic returns the optimizer for the critic. If None, defaults to Adam.
- **critic_loss_fn** (LossFn) – The loss function used to optimize the critic. If None, defaults to MSELoss.
- **stack_size** (*int*) – Number of observations stacked to create the state fed to the agent.
- **replay_buffer** (BaseReplayBuffer) – The replay buffer that the agent will push observations to and sample from during learning. If None, defaults to *CircularReplayBuffer*.
- **discount_rate** (*float*) – A number between 0 and 1 specifying how much future rewards are discounted by the agent.
- **n_step** (*int*) – The horizon used in n-step returns to compute TD(n) targets.
- **grad_clip** (*float*) – Gradients will be clipped to between [-grad_clip, grad_clip].
- **reward_clip** (*float*) – Rewards will be clipped to between [-reward_clip, reward_clip].
- **soft_update_fraction** (*float*) – The weight given to the target net parameters in a soft (polyak) update. Also known as tau.
- **batch_size** (*int*) – The size of the batch sampled from the replay buffer during learning.
- **logger** (Logger) – Logger used to log agent's metrics.
- **log_frequency** (*int*) – How often to log the agent's metrics.
- **update_frequency** (*int*) – How frequently to update the agent. A value of 1 means the agent will be updated every time update is called.
- **action_noise** (*float*) – The standard deviation for the noise added to the action taken by the agent during training.
- **min_replay_history** (*int*) – How many observations to fill the replay buffer with before starting to learn.
- **device** – Device on which all computations should be run.
- **id** – Agent identifier.

hive.agents.dqn module

class hive.agents.dqn.**DQNAgent**(*observation_space*, *action_space*, *representation_net*, *stack_size=1*, *id=0*, *optimizer_fn=None*, *loss_fn=None*, *init_fn=None*, *replay_buffer=None*, *discount_rate=0.99*, *n_step=1*, *grad_clip=None*, *reward_clip=None*, *update_period_schedule=None*, *target_net_soft_update=False*, *target_net_update_fraction=0.05*, *target_net_update_schedule=None*, *epsilon_schedule=None*, *test_epsilon=0.001*, *min_replay_history=5000*, *batch_size=32*, *device='cpu'*, *logger=None*, *log_frequency=100*)

Bases: *Agent*

An agent implementing the DQN algorithm. Uses an epsilon greedy exploration policy

Parameters

- **observation_space** (*gym.spaces.Box*) – Observation space for the agent.
- **action_space** (*gym.spaces.Discrete*) – Action space for the agent.
- **representation_net** (FunctionApproximator) – A network that outputs the representations that will be used to compute Q-values (e.g. everything except the final layer of the DQN).
- **stack_size** (int) – Number of observations stacked to create the state fed to the DQN.
- **id** – Agent identifier.
- **optimizer_fn** (OptimizerFn) – A function that takes in a list of parameters to optimize and returns the optimizer. If None, defaults to Adam.
- **loss_fn** (LossFn) – Loss function used by the agent. If None, defaults to SmoothL1Loss.
- **init_fn** (InitializationFn) – Initializes the weights of qnet using create_init_weights_fn.
- **replay_buffer** (BaseReplayBuffer) – The replay buffer that the agent will push observations to and sample from during learning. If None, defaults to *CircularReplayBuffer*.
- **discount_rate** (*float*) – A number between 0 and 1 specifying how much future rewards are discounted by the agent.
- **n_step** (*int*) – The horizon used in n-step returns to compute TD(n) targets.
- **grad_clip** (*float*) – Gradients will be clipped to between [-grad_clip, grad_clip].
- **reward_clip** (*float*) – Rewards will be clipped to between [-reward_clip, reward_clip].
- **update_period_schedule** (Schedule) – Schedule determining how frequently the agent's Q-network is updated.
- **target_net_soft_update** (*bool*) – Whether the target net parameters are replaced by the qnet parameters completely or using a weighted average of the target net parameters and the qnet parameters.
- **target_net_update_fraction** (*float*) – The weight given to the target net parameters in a soft update.
- **target_net_update_schedule** (Schedule) – Schedule determining how frequently the target net is updated.
- **epsilon_schedule** (Schedule) – Schedule determining the value of epsilon through the course of training.

- **test_epsilon** (*float*) – epsilon (probability of choosing a random action) to be used during testing phase.
- **min_replay_history** (*int*) – How many observations to fill the replay buffer with before starting to learn.
- **batch_size** (*int*) – The size of the batch sampled from the replay buffer during learning.
- **device** – Device on which all computations should be run.
- **logger** (ScheduledLogger) – Logger used to log agent's metrics.
- **log_frequency** (*int*) – How often to log the agent's metrics.

create_q_networks(*representation_net*)

Creates the Q-network and target Q-network.

Parameters

representation_net – A network that outputs the representations that will be used to compute Q-values (e.g. everything except the final layer of the DQN).

train()

Changes the agent to training mode.

eval()

Changes the agent to evaluation mode.

preprocess_update_info(*update_info*)

Preprocesses the update_info before it goes into the replay buffer. Clips the reward in update_info.

Parameters

update_info – Contains the information from the current timestep that the agent should use to update itself.

preprocess_update_batch(*batch*)

Preprocess the batch sampled from the replay buffer.

Parameters

batch – Batch sampled from the replay buffer for the current update.

Returns

- (tuple) Inputs used to calculate current state values.
- (tuple) Inputs used to calculate next state values
- Preprocessed batch.

Return type

(tuple)

act(*observation*, *agent_traj_state=None*)

Returns the action for the agent. If in training mode, follows an epsilon greedy policy. Otherwise, returns the action with the highest Q-value.

Parameters

- **observation** – The current observation.
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action

- agent trajectory state

update(*update_info*, *agent_traj_state=None*)

Updates the DQN agent.

Parameters

- **update_info** – dictionary containing all the necessary information from the environment to update the agent. Should contain a full transition, with keys for "observation", "action", "reward", "next_observation", "terminated", and "truncated".
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

save(*dname*)

Saves agent checkpointing information to file for future loading.

Parameters

dname (*str*) – directory where agent should save all relevant info.

load(*dname*)

Loads agent information from file.

Parameters

dname (*str*) – directory where agent checkpoint info is stored.

hive.agents.drqn module

class hive.agents.drqn.**DRQNAgent**(*observation_space*, *action_space*, *representation_net*, *sequence_fn*, *id=0*, *optimizer_fn=None*, *loss_fn=None*, *init_fn=None*, *replay_buffer=None*, *max_seq_len=1*, *discount_rate=0.99*, *n_step=1*, *grad_clip=None*, *reward_clip=None*, *update_period_schedule=None*, *target_net_soft_update=False*, *target_net_update_fraction=0.05*, *target_net_update_schedule=None*, *epsilon_schedule=None*, *test_epsilon=0.001*, *min_replay_history=5000*, *batch_size=32*, *device='cpu'*, *logger=None*, *log_frequency=100*, *store_hidden=True*, *burn_frames=0*, ***kwargs*)

Bases: *DQNAgent*

An agent implementing the DRQN algorithm. Uses an epsilon greedy exploration policy

Parameters

- **observation_space** (*gym.spaces.Box*) – Observation space for the agent.
- **action_space** (*gym.spaces.Discrete*) – Action space for the agent.
- **representation_net** (*SequenceFunctionApproximator*) – A network that outputs the representations that will be used to compute Q-values (e.g. everything except the final layer of the DRQN), as well as the hidden states of the recurrent component. The structure should be similar to ConvRNNNetwork, i.e., it should have a current module component placed between the convolutional layers and MLP layers. It should also define a method that initializes the hidden state of the recurrent module if the computation requires hidden states as input/output.

- **id** – Agent identifier.
- **optimizer_fn** (OptimizerFn) – A function that takes in a list of parameters to optimize and returns the optimizer. If None, defaults to Adam.
- **loss_fn** (LossFn) – Loss function used by the agent. If None, defaults to SmoothL1Loss.
- **init_fn** (InitializationFn) – Initializes the weights of qnet using create_init_weights_fn.
- **replay_buffer** (BaseReplayBuffer) – The replay buffer that the agent will push observations to and sample from during learning. If None, defaults to *RecurrentReplayBuffer*.
- **max_seq_len** (*int*) – The number of consecutive transitions in a sequence.
- **discount_rate** (*float*) – A number between 0 and 1 specifying how much future rewards are discounted by the agent.
- **n_step** (*int*) – The horizon used in n-step returns to compute TD(n) targets.
- **grad_clip** (*float*) – Gradients will be clipped to between [-grad_clip, grad_clip].
- **reward_clip** (*float*) – Rewards will be clipped to between [-reward_clip, reward_clip].
- **update_period_schedule** (Schedule) – Schedule determining how frequently the agent's Q-network is updated.
- **target_net_soft_update** (*bool*) – Whether the target net parameters are replaced by the qnet parameters completely or using a weighted average of the target net parameters and the qnet parameters.
- **target_net_update_fraction** (*float*) – The weight given to the target net parameters in a soft update.
- **target_net_update_schedule** (Schedule) – Schedule determining how frequently the target net is updated.
- **epsilon_schedule** (Schedule) – Schedule determining the value of epsilon through the course of training.
- **test_epsilon** (*float*) – epsilon (probability of choosing a random action) to be used during testing phase.
- **min_replay_history** (*int*) – How many observations to fill the replay buffer with before starting to learn.
- **batch_size** (*int*) – The size of the batch sampled from the replay buffer during learning.
- **device** – Device on which all computations should be run.
- **logger** (ScheduledLogger) – Logger used to log agent's metrics.
- **log_frequency** (*int*) – How often to log the agent's metrics.

create_q_networks(*representation_net*, *sequence_fn*)

Creates the Q-network and target Q-network.

Parameters

representation_net – A network that outputs the representations that will be used to compute Q-values (e.g. everything except the final layer of the DRQN).

preprocess_update_info(*update_info*, *hidden_state*)

Preprocesses the `update_info` before it goes into the replay buffer. Clips the reward in update_info. :param update_info: Contains the information from the current timestep that the

agent should use to update itself.

preprocess_update_batch(*batch*)

Preprocess the batch sampled from the replay buffer.

Parameters
batch – Batch sampled from the replay buffer for the current update.

Returns

- (tuple) Inputs used to calculate current state values.
- (tuple) Inputs used to calculate next state values
- Preprocessed batch.

Return type
(tuple)

act(*observation*, *agent_traj_state=None*)

Returns the action for the agent. If in training mode, follows an epsilon greedy policy. Otherwise, returns the action with the highest Q-value.

Parameters

- **observation** – The current observation.
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

update(*update_info*, *agent_traj_state=None*)

Updates the DRQN agent.

Parameters

- **update_info** – dictionary containing all the necessary information from the environment to update the agent. Should contain a full transition, with keys for "observation", "action", "reward", "next_observation", "terminated", and "truncated".
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

hive.agents.legal_moves_rainbow module

class hive.agents.legal_moves_rainbow.**LegalMovesRainbowAgent**(*observation_space*, *action_space*, *representation_net*, *stack_size=1*, *optimizer_fn=None*, *loss_fn=None*, *init_fn=None*, *id=0*, *replay_buffer=None*, *discount_rate=0.99*, *n_step=1*, *grad_clip=None*, *reward_clip=None*, *update_period_schedule=None*, *target_net_soft_update=False*, *target_net_update_fraction=0.05*, *target_net_update_schedule=None*, *epsilon_schedule=None*, *test_epsilon=0.001*, *min_replay_history=5000*, *batch_size=32*, *device='cpu'*, *logger=None*, *log_frequency=100*, *noisy=True*, *std_init=0.5*, *use_eps_greedy=False*, *double=True*, *dueling=True*, *distributional=True*, *v_min=0*, *v_max=200*, *atoms=51*)

Bases: *RainbowDQNAgent*

A Rainbow agent which supports games with legal actions.

Parameters

- **observation_space** (*gym.spaces.Box*) – Observation space for the agent.
- **action_space** (*gym.spaces.Discrete*) – Action space for the agent.
- **representation_net** (FunctionApproximator) – A network that outputs the representations that will be used to compute Q-values (e.g. everything except the final layer of the DQN).
- **stack_size** (int) – Number of observations stacked to create the state fed to the DQN.
- **id** – Agent identifier.
- **optimizer_fn** (OptimizerFn) – A function that takes in a list of parameters to optimize and returns the optimizer. If None, defaults to Adam.
- **loss_fn** (LossFn) – Loss function used by the agent. If None, defaults to SmoothL1Loss.
- **init_fn** (InitializationFn) – Initializes the weights of qnet using create_init_weights_fn.
- **replay_buffer** (BaseReplayBuffer) – The replay buffer that the agent will push observations to and sample from during learning. If None, defaults to *PrioritizedReplayBuffer*.
- **discount_rate** (*float*) – A number between 0 and 1 specifying how much future rewards are discounted by the agent.
- **n_step** (*int*) – The horizon used in n-step returns to compute TD(n) targets.
- **grad_clip** (*float*) – Gradients will be clipped to between [-grad_clip, grad_clip].
- **reward_clip** (*float*) – Rewards will be clipped to between [-reward_clip, reward_clip].

- **update_period_schedule** (Schedule) – Schedule determining how frequently the agent's Q-network is updated.
- **target_net_soft_update** (*bool*) – Whether the target net parameters are replaced by the qnet parameters completely or using a weighted average of the target net parameters and the qnet parameters.
- **target_net_update_fraction** (*float*) – The weight given to the target net parameters in a soft update.
- **target_net_update_schedule** (Schedule) – Schedule determining how frequently the target net is updated.
- **epsilon_schedule** (Schedule) – Schedule determining the value of epsilon through the course of training.
- **test_epsilon** (*float*) – epsilon (probability of choosing a random action) to be used during testing phase.
- **min_replay_history** (*int*) – How many observations to fill the replay buffer with before starting to learn.
- **batch_size** (*int*) – The size of the batch sampled from the replay buffer during learning.
- **device** – Device on which all computations should be run.
- **logger** (ScheduledLogger) – Logger used to log agent's metrics.
- **log_frequency** (*int*) – How often to log the agent's metrics.
- **noisy** (*bool*) – Whether to use noisy linear layers for exploration.
- **std_init** (*float*) – The range for the initialization of the standard deviation of the weights.
- **use_eps_greedy** (*bool*) – Whether to use epsilon greedy exploration.
- **double** (*bool*) – Whether to use double DQN.
- **dueling** (*bool*) – Whether to use a dueling network architecture.
- **distributional** (*bool*) – Whether to use the distributional RL.
- **vmin** (*float*) – The minimum of the support of the categorical value distribution for distributional RL.
- **vmax** (*float*) – The maximum of the support of the categorical value distribution for distributional RL.
- **atoms** (*int*) – Number of atoms discretizing the support range of the categorical value distribution for distributional RL.

create_q_networks(*representation_net*)

Creates the qnet and target qnet.

preprocess_update_info(*update_info*)

Preprocesses the `update_info` before it goes into the replay buffer. Clips the reward in update_info.

Parameters

update_info – Contains the information from the current timestep that the agent should use to update itself.

preprocess_update_batch(*batch*)

Preprocess the batch sampled from the replay buffer.

Parameters
batch – Batch sampled from the replay buffer for the current update.

Returns

- (tuple) Inputs used to calculate current state values.
- (tuple) Inputs used to calculate next state values
- Preprocessed batch.

Return type
(tuple)

act(*observation*, *agent_traj_state=None*)

Returns the action for the agent. If in training mode, follows an epsilon greedy policy. Otherwise, returns the action with the highest Q-value.

Parameters

- **observation** – The current observation.
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

class hive.agents.legal_moves_rainbow.**LegalMovesHead**(*base_network*)

Bases: Module

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *legal_moves*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

dist(*x*, *legal_moves*)

training: bool

hive.agents.legal_moves_rainbow.**action_encoding**(*action_mask*)

hive.agents.rainbow module

class hive.agents.rainbow.**RainbowDQNAgent**(*observation_space*, *action_space*, *representation_net*, *stack_size=1*, *optimizer_fn=None*, *loss_fn=None*, *init_fn=None*, *id=0*, *replay_buffer=None*, *discount_rate=0.99*, *n_step=1*, *grad_clip=None*, *reward_clip=None*, *update_period_schedule=None*, *target_net_soft_update=False*, *target_net_update_fraction=0.05*, *target_net_update_schedule=None*, *epsilon_schedule=None*, *test_epsilon=0.001*, *min_replay_history=5000*, *batch_size=32*, *device='cpu'*, *logger=None*, *log_frequency=100*, *noisy=True*, *std_init=0.5*, *use_eps_greedy=False*, *double=True*, *dueling=True*, *distributional=True*, *v_min=0*, *v_max=200*, *atoms=51*)

Bases: *DQNAgent*

An agent implementing the Rainbow algorithm.

Parameters

- **observation_space** (*gym.spaces.Box*) – Observation space for the agent.
- **action_space** (*gym.spaces.Discrete*) – Action space for the agent.
- **representation_net** (FunctionApproximator) – A network that outputs the representations that will be used to compute Q-values (e.g. everything except the final layer of the DQN).
- **stack_size** (int) – Number of observations stacked to create the state fed to the DQN.
- **id** – Agent identifier.
- **optimizer_fn** (OptimizerFn) – A function that takes in a list of parameters to optimize and returns the optimizer. If None, defaults to Adam.
- **loss_fn** (LossFn) – Loss function used by the agent. If None, defaults to SmoothL1Loss.
- **init_fn** (InitializationFn) – Initializes the weights of qnet using create_init_weights_fn.
- **replay_buffer** (BaseReplayBuffer) – The replay buffer that the agent will push observations to and sample from during learning. If None, defaults to *PrioritizedReplayBuffer*.
- **discount_rate** (*float*) – A number between 0 and 1 specifying how much future rewards are discounted by the agent.
- **n_step** (*int*) – The horizon used in n-step returns to compute TD(n) targets.
- **grad_clip** (*float*) – Gradients will be clipped to between [-grad_clip, grad_clip].
- **reward_clip** (*float*) – Rewards will be clipped to between [-reward_clip, reward_clip].
- **update_period_schedule** (Schedule) – Schedule determining how frequently the agent's Q-network is updated.
- **target_net_soft_update** (*bool*) – Whether the target net parameters are replaced by the qnet parameters completely or using a weighted average of the target net parameters and the qnet parameters.
- **target_net_update_fraction** (*float*) – The weight given to the target net parameters in a soft update.

- **target_net_update_schedule** (Schedule) – Schedule determining how frequently the target net is updated.
- **epsilon_schedule** (Schedule) – Schedule determining the value of epsilon through the course of training.
- **test_epsilon** (*float*) – epsilon (probability of choosing a random action) to be used during testing phase.
- **min_replay_history** (*int*) – How many observations to fill the replay buffer with before starting to learn.
- **batch_size** (*int*) – The size of the batch sampled from the replay buffer during learning.
- **device** – Device on which all computations should be run.
- **logger** (ScheduledLogger) – Logger used to log agent's metrics.
- **log_frequency** (*int*) – How often to log the agent's metrics.
- **noisy** (*bool*) – Whether to use noisy linear layers for exploration.
- **std_init** (*float*) – The range for the initialization of the standard deviation of the weights.
- **use_eps_greedy** (*bool*) – Whether to use epsilon greedy exploration.
- **double** (*bool*) – Whether to use double DQN.
- **dueling** (*bool*) – Whether to use a dueling network architecture.
- **distributional** (*bool*) – Whether to use the distributional RL.
- **vmin** (*float*) – The minimum of the support of the categorical value distribution for distributional RL.
- **vmax** (*float*) – The maximum of the support of the categorical value distribution for distributional RL.
- **atoms** (*int*) – Number of atoms discretizing the support range of the categorical value distribution for distributional RL.

create_q_networks(*representation_net*)

Creates the Q-network and target Q-network. Adds the appropriate heads for DQN, Dueling DQN, Noisy Networks, and Distributional DQN.

Parameters

representation_net – A network that outputs the representations that will be used to compute Q-values (e.g. everything except the final layer of the DQN).

act(*observation*, *agent_traj_state=None*)

Returns the action for the agent. If in training mode, follows an epsilon greedy policy. Otherwise, returns the action with the highest Q-value.

Parameters

- **observation** – The current observation.
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

update(*update_info*, *agent_traj_state=None*)

Updates the DQN agent.

Parameters

- **update_info** – dictionary containing all the necessary information from the environment to update the agent. Should contain a full transition, with keys for "observation", "action", "reward", "next_observation", "terminated", and "truncated".
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

target_projection(*target_net_inputs*, *next_action*, *reward*, *terminated*)

Project distribution of target Q-values.

Parameters

- **target_net_inputs** – Inputs to feed into the target net to compute the projection of the target Q-values. Should be set from *preprocess_update_batch()*.
- **next_action** (*Tensor*) – Tensor containing next actions used to compute target distribution.
- **reward** (*Tensor*) – Tensor containing rewards for the current batch.
- **terminated** (*Tensor*) – Tensor containing whether the states in
- **terminal.** (*the current batch are*) –

hive.agents.random module

class hive.agents.random.**RandomAgent**(*observation_space*, *action_space*, *id=0*, *logger=None*, ***kwargs*)

Bases: *Agent*

An agent that takes random steps at each timestep.

Parameters

- **observation_space** (*gym.Space*) – The shape of the observations.
- **action_space** (*gym.Space*) – The number of actions available to the agent.
- **id** – Agent identifier.
- **logger** (ScheduledLogger) – Logger used to log agent's metrics.

act(*observation*, *agent_traj_state=None*)

Returns a random action for the agent.

update(*update_info*, *agent_traj_state=None*)

Updates the agent.

Parameters

- **update_info** (*dict*) – Contains information from the environment agent needs to update itself.

- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns
Agent trajectory state.

save(*dname*)

Saves agent checkpointing information to file for future loading.

Parameters
dname (*str*) – directory where agent should save all relevant info.

load(*dname*)

Loads agent information from file.

Parameters
dname (*str*) – directory where agent checkpoint info is stored.

hive.agents.td3 module

class hive.agents.td3.**TD3**(*observation_space*, *action_space*, *representation_net=None*, *actor_net=None*, *critic_net=None*, *init_fn=None*, *actor_optimizer_fn=None*, *critic_optimizer_fn=None*, *critic_loss_fn=None*, *n_critics=2*, *stack_size=1*, *replay_buffer=None*, *discount_rate=0.99*, *n_step=1*, *grad_clip=None*, *reward_clip=None*, *soft_update_fraction=0.005*, *batch_size=64*, *logger=None*, *log_frequency=100*, *update_frequency=1*, *policy_update_frequency=2*, *action_noise=0*, *target_noise=0.2*, *target_noise_clip=0.5*, *min_replay_history=1000*, *device='cpu'*, *id=0*)

Bases: *Agent*

An agent implementing the TD3 algorithm.

Parameters

- **observation_space** (*gym.spaces.Box*) – Observation space for the agent.
- **action_space** (*gym.spaces.Box*) – Action space for the agent.
- **representation_net** (FunctionApproximator) – The network that encodes the observations that are then fed into the actor_net and critic_net. If None, defaults to Identity.
- **actor_net** (FunctionApproximator) – The network that takes the encoded observations from representation_net and outputs the representations used to compute the actions (ie everything except the last layer).
- **critic_net** (FunctionApproximator) – The network that takes two inputs: the encoded observations from representation_net and actions. It outputs the representations used to compute the values of the actions (ie everything except the last layer).
- **init_fn** (InitializationFn) – Initializes the weights of agent networks using create_init_weights_fn.
- **actor_optimizer_fn** (OptimizerFn) – A function that takes in the list of parameters of the actor returns the optimizer for the actor. If None, defaults to Adam.
- **critic_optimizer_fn** (OptimizerFn) – A function that takes in the list of parameters of the critic returns the optimizer for the critic. If None, defaults to Adam.
- **critic_loss_fn** (LossFn) – The loss function used to optimize the critic. If None, defaults to MSELoss.

- **n_critics** (*int*) – The number of critics used by the agent to estimate Q-values. The minimum Q-value is used as the value for the next state when calculating target Q-values for the critic. The output of the first critic is used when computing the loss for the actor. For TD3, the default value is 2. For DDPG, this parameter is 1.
- **stack_size** (*int*) – Number of observations stacked to create the state fed to the agent.
- **replay_buffer** (BaseReplayBuffer) – The replay buffer that the agent will push observations to and sample from during learning. If None, defaults to *CircularReplayBuffer*.
- **discount_rate** (*float*) – A number between 0 and 1 specifying how much future rewards are discounted by the agent.
- **n_step** (*int*) – The horizon used in n-step returns to compute TD(n) targets.
- **grad_clip** (*float*) – Gradients will be clipped to between [-grad_clip, grad_clip].
- **reward_clip** (*float*) – Rewards will be clipped to between [-reward_clip, reward_clip].
- **soft_update_fraction** (*float*) – The weight given to the target net parameters in a soft (polyak) update. Also known as tau.
- **batch_size** (*int*) – The size of the batch sampled from the replay buffer during learning.
- **logger** (Logger) – Logger used to log agent's metrics.
- **log_frequency** (*int*) – How often to log the agent's metrics.
- **update_frequency** (*int*) – How frequently to update the agent. A value of 1 means the agent will be updated every time update is called.
- **policy_update_frequency** (*int*) – Relative update frequency of the actor compared to the critic. The actor will be updated every policy_update_frequency times the critic is updated.
- **action_noise** (*float*) – The standard deviation for the noise added to the action taken by the agent during training.
- **target_noise** (*float*) – The standard deviation of the noise added to the target policy for smoothing.
- **target_noise_clip** (*float*) – The sampled target_noise is clipped to [-target_noise_clip, target_noise_clip].
- **min_replay_history** (*int*) – How many observations to fill the replay buffer with before starting to learn.
- **device** – Device on which all computations should be run.
- **id** – Agent identifier.

create_networks(*representation_net*, *actor_net*, *critic_net*)

Creates the actor and critic networks.

Parameters

- **representation_net** – A network that outputs the shared representations that will be used by the actor and critic networks to process observations.
- **actor_net** – The network that will be used to compute actions.
- **critic_net** – The network that will be used to compute values of state action pairs.

train()

Changes the agent to training mode.

`eval()`

Changes the agent to evaluation mode.

`scale_action(actions)`

Scales actions to [-1, 1].

`unscale_actions(actions)`

Unscales actions from [-1, 1] to expected scale.

`preprocess_update_info(update_info)`

Preprocesses the `update_info` before it goes into the replay buffer. Scales the action to [-1, 1].

Parameters

update_info – Contains the information from the current timestep that the agent should use to update itself.

`preprocess_update_batch(batch)`

Preprocess the batch sampled from the replay buffer.

Parameters

batch – Batch sampled from the replay buffer for the current update.

Returns

- (tuple) Inputs used to calculate current state values.
- (tuple) Inputs used to calculate next state values
- Preprocessed batch.

Return type

(tuple)

`act(observation, agent_traj_state=None)`

Returns the action for the agent. If in training mode, adds noise with standard deviation `self._action_noise`.

Parameters

- **observation** – The current observation.
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

`update(update_info, agent_traj_state=None)`

Updates the TD3 agent.

Parameters

- **update_info** – dictionary containing all the necessary information from the environment to update the agent. Should contain a full transition, with keys for "observation", "action", "reward", "next_observation", "terminated", and "truncated
- **agent_traj_state** – Contains necessary state information for the agent to process current trajectory. This should be updated and returned.

Returns

- action
- agent trajectory state

`save`(*dname*)

Saves agent checkpointing information to file for future loading.

Parameters
dname (*str*) – directory where agent should save all relevant info.

`load`(*dname*)

Loads agent information from file.

Parameters
dname (*str*) – directory where agent checkpoint info is stored.

Module contents

hive.envs package

Subpackages

hive.envs.atari package

Submodules

hive.envs.atari.atari module

Module contents

hive.envs.marlgrid package

Subpackages

hive.envs.marlgrid.ma_envs package

Submodules

hive.envs.marlgrid.ma_envs.base module

hive.envs.marlgrid.ma_envs.checkers module

hive.envs.marlgrid.ma_envs.pursuit module

hive.envs.marlgrid.ma_envs.switch module

Module contents

Submodules

hive.envs.marlgrid.marlgrid module

Module contents

hive.envs.minigrid package

Submodules

hive.envs.minigrid.minigrid module

Module contents

hive.envs.pettingzoo package

Submodules

hive.envs.pettingzoo.pettingzoo module

class hive.envs.pettingzoo.pettingzoo.**PettingZooEnv**(*env_name*, *env_family*, *num_players*, ***kwargs*)

Bases: *BaseEnv*

PettingZoo environment from https://github.com/PettingZoo-Team/PettingZoo

For now, we only support environments from PettingZoo with discrete actions.

Parameters

- **env_name** (*str*) – Name of the environment
- **env_family** (*str*) – Family of the environment such as "Atari",
- **"Classic"** –
- **"SISL"** –
- **"Butterfly"** –
- **"MAgent"** –
- **"MPE".** (*and*) –
- **num_players** (*int*) – Number of learning agents

create_env(*env_name*, *num_players*, ***kwargs*)

create_env_spec(*env_name*, ***kwargs*)

Each family of environments have their own type of observations and actions. You can add support for more families here by modifying observation_space and action_space.

reset()

Resets the state of the environment.

Returns

The initial observation of the new episode. turn (int): The index of the agent which should take turn.

Return type

observation

step(*action*)

Run one time-step of the environment using the input action.

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

Returns
Indicates the next state that is an element of environment's observation space. reward: A reward achieved from the transition. done (bool): Indicates whether the episode has ended. turn (int): Indicates which agent should take turn. info (dict): Additional custom information.

Return type
observation

render(*mode='rgb_array'*)

Displays a rendered frame from the environment.

seed(*seed=None*)

Reseeds the environment.

Parameters
seed (*int*) – Seed to use for environment.

close()

Additional clean up operations

Module contents

hive.envs.wrappers package

Submodules

hive.envs.wrappers.gym_wrappers module

Module contents

Submodules

hive.envs.base module

class hive.envs.base.**BaseEnv**(*env_spec*, *num_players*)

Bases: ABC, *Registrable*

Base class for environments.

Parameters

- **env_spec** (EnvSpec) – An object containing information about the environment.
- **num_players** (*int*) – The number of players in the environment.

abstract reset()

Resets the state of the environment.

Returns
The initial observation of the new episode. turn (int): The index of the agent which should take turn.

Return type
observation

abstract step(*action*)

Run one time-step of the environment using the input action.

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

Returns
Indicates the next state that is an element of environment's observation space. reward: A reward achieved from the transition. done (bool): Indicates whether the episode has ended. turn (int): Indicates which agent should take turn. info (dict): Additional custom information.

Return type
observation

render(*mode='rgb_array'*)

Displays a rendered frame from the environment.

abstract seed(*seed=None*)

Reseeds the environment.

Parameters
seed (*int*) – Seed to use for environment.

save(*save_dir*)

Saves the environment.

Parameters
save_dir (*str*) – Location to save environment state.

load(*load_dir*)

Loads the environment.

Parameters
load_dir (*str*) – Location to load environment state from.

close()

Additional clean up operations

property env_spec

classmethod type_name()

Returns: "env"

class hive.envs.base.**ParallelEnv**(*env_name*, *num_players*, ***kwargs*)

Bases: *BaseEnv*

Base class for environments that take make all agents step in parallel.

ParallelEnv takes an environment that expects an array of actions at each step to execute in parallel, and allows you to instead pass it a single action at each step.

This class makes use of Python's multiple inheritance pattern. Specifically, when writing your parallel environment, it should extend both this class and the class that implements the step method that takes in actions for all agents.

If environment class A has the logic for the step function that takes in the array of actions, and environment class B is your parallel step version of that environment, class B should be defined as:

```
class B(ParallelEnv, A):
    ...
```

The order in which you list the classes is important. ParallelEnv **must** come before A in the order.

Parameters

- **env_spec** (EnvSpec) – An object containing information about the environment.
- **num_players** (*int*) – The number of players in the environment.

reset()

Resets the state of the environment.

Returns
The initial observation of the new episode. turn (int): The index of the agent which should take turn.

Return type
observation

step(*action*)

Run one time-step of the environment using the input action.

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

Returns
Indicates the next state that is an element of environment's observation space. reward: A reward achieved from the transition. done (bool): Indicates whether the episode has ended. turn (int): Indicates which agent should take turn. info (dict): Additional custom information.

Return type
observation

hive.envs.env_spec module

class hive.envs.env_spec.**EnvSpec**(*env_name*, *observation_space*, *action_space*, *env_info=None*)

Bases: object

Object used to store information about environment configuration. Every environment should create an EnvSpec object.

Parameters

- **env_name** – Name of the environment
- **observation_space** (Union[Space, List[Space]]) – Spaces of observations from environment. This should be a single instance or list of gym.Space, depending on if the environment is multiagent.
- **action_space** (Union[Space, List[Space]]) – Spaces of actions expected by environment. This should be a single instance or list of gym.Space, depending on if the environment is multiagent.
- **env_info** – Any other info relevant to this environment. This can include items such as random seeds or parameters used to create the environment

property env_name

property observation_space

property action_space

property env_info

hive.envs.gym_env module

Module contents

hive.replays package

Submodules

hive.replays.circular_replay module

class hive.replays.circular_replay.**CircularReplayBuffer**(*capacity=10000*, *stack_size=1*, *n_step=1*, *gamma=0.99*, *observation_shape=()*, *observation_dtype=<class 'numpy.uint8'>*, *action_shape=()*, *action_dtype=<class 'numpy.int8'>*, *reward_shape=()*, *reward_dtype=<class 'numpy.float32'>*, *extra_storage_types=None*, *num_players_sharing_buffer=None*, *optimize_storage=True*)

Bases: *BaseReplayBuffer*

An efficient version of a circular replay buffer that only stores each observation once.

Constructor for CircularReplayBuffer.

Parameters

- **capacity** (*int*) – Total number of observations that can be stored in the buffer. Note, this is not the same as the number of transitions that can be stored in the buffer.
- **stack_size** (*int*) – The number of frames to stack to create an observation.
- **n_step** (*int*) – Horizon used to compute n-step return reward
- **gamma** (*float*) – Discounting factor used to compute n-step return reward
- **observation_shape** – Shape of observations that will be stored in the buffer.
- **observation_dtype** – Type of observations that will be stored in the buffer. This can either be the type itself or string representation of the type. The type can be either a native python type or a numpy type. If a numpy type, a string of the form np.uint8 or numpy.uint8 is acceptable.
- **action_shape** – Shape of actions that will be stored in the buffer.
- **action_dtype** – Type of actions that will be stored in the buffer. Format is described in the description of observation_dtype.
- **action_shape** – Shape of actions that will be stored in the buffer.

- **action_dtype** – Type of actions that will be stored in the buffer. Format is described in the description of observation_dtype.
- **reward_shape** – Shape of rewards that will be stored in the buffer.
- **reward_dtype** – Type of rewards that will be stored in the buffer. Format is described in the description of observation_dtype.
- **extra_storage_types** (*dict*) – A dictionary describing extra items to store in the buffer. The mapping should be from the name of the item to a (type, shape) tuple.
- **num_players_sharing_buffer** (*int*) – Number of agents that share their buffers. It is used for self-play.
- **optimize_storage** (*bool*) – If True, the buffer will only store each observation once. Otherwise, next_observation will be stored for each transition. Note, if optimize_storage is True, the next_observation for a transition where terminated OR truncated is True will not be correct.

size()

Returns the number of transitions stored in the buffer.

add(*observation*, *next_observation*, *action*, *reward*, *terminated*, *truncated*, ***kwargs*)

Adds a transition to the buffer. The required components of a transition are given as positional arguments. The user can pass additional components to store in the buffer as kwargs as long as they were defined in the specification in the constructor.

sample(*batch_size*)

Sample transitions from the buffer. For a given transition, if it's done is True, the next_observation value should not be taken to have any meaning.

Parameters
batch_size (*int*) – Number of transitions to sample.

save(*dname*)

Save the replay buffer.

Parameters
dname (*str*) – directory where to save buffer. Should already have been created.

load(*dname*)

Load the replay buffer.

Parameters
dname (*str*) – directory where to load buffer from.

class hive.replays.circular_replay.**SimpleReplayBuffer**(*capacity=100000.0*, *compress=False*, *seed=42*, ***kwargs*)

Bases: *BaseReplayBuffer*

A simple circular replay buffers.

Parameters

- **capacity** (*int*) – repaly buffer capacity
- **compress** (*bool*) – if False, convert data to float32 otherwise keep it as int8.
- **seed** (*int*) – Seed for a pseudo-random number generator.

add(*observation*, *next_observation*, *action*, *reward*, *terminated*, *truncated*, ***kwargs*)

Adds transition to the buffer

Parameters

- **observation** – The current observation
- **next_observation** – The next observation
- **action** – The action taken on the current observation
- **reward** – The reward from taking action at current observation
- **terminated** – If the trajectory was terminated at the current transition
- **truncated** – If the trajectory was truncated at the current transition

sample(*batch_size=32*)

sample a minibatch

Parameters
batch_size (*int*) – The number of examples to sample.

size()

returns the number of transitions stored in the replay buffer

save(*dname*)

Saves buffer checkpointing information to file for future loading.

Parameters
dname (*str*) – directory name where agent should save all relevant info.

load(*dname*)

Loads buffer from file.

Parameters
dname (*str*) – directory name where buffer checkpoint info is stored.

Returns
True if successfully loaded the buffer. False otherwise.

hive.replays.circular_replay.**str_to_dtype**(*dtype*)

hive.replays.legal_moves_replay module

class hive.replays.legal_moves_replay.**LegalMovesBuffer**(*capacity*, *beta=0.5*, *stack_size=1*, *n_step=1*, *gamma=0.9*, *observation_shape=()*, *observation_dtype=<class 'numpy.uint8'>*, *action_shape=()*, *action_dtype=<class 'numpy.int8'>*, *reward_shape=()*, *reward_dtype=<class 'numpy.float32'>*, *extra_storage_types=None*, *action_dim=None*, *num_players_sharing_buffer=None*)

Bases: *PrioritizedReplayBuffer*

A Prioritized Replay buffer for the games like Hanabi with legal moves which need to add next_action_mask to the batch.

Parameters

- **capacity** (*int*) – Total number of observations that can be stored in the buffer. Note, this is not the same as the number of transitions that can be stored in the buffer.
- **alpha** (*float*) – Parameter controlling level of prioritization.
- **beta** (*float*) – Parameter controlling level of correction for prioritization.
- **stack_size** (*int*) – The number of frames to stack to create an observation.
- **n_step** (*int*) – Horizon used to compute n-step return reward
- **gamma** (*float*) – Discounting factor used to compute n-step return reward
- **observation_shape** (Tuple) – Shape of observations that will be stored in the buffer.
- **observation_dtype** (type) – Type of observations that will be stored in the buffer. This can either be the type itself or string representation of the type. The type can be either a native python type or a numpy type. If a numpy type, a string of the form np.uint8 or numpy.uint8 is acceptable.
- **action_shape** (Tuple) – Shape of actions that will be stored in the buffer.
- **action_dtype** (type) – Type of actions that will be stored in the buffer. Format is described in the description of observation_dtype.
- **action_shape** – Shape of actions that will be stored in the buffer.
- **action_dtype** – Type of actions that will be stored in the buffer. Format is described in the description of observation_dtype.
- **reward_shape** (Tuple) – Shape of rewards that will be stored in the buffer.
- **reward_dtype** (type) – Type of rewards that will be stored in the buffer. Format is described in the description of observation_dtype.
- **extra_storage_types** (*dict*) – A dictionary describing extra items to store in the buffer. The mapping should be from the name of the item to a (type, shape) tuple.
- **num_players_sharing_buffer** (*int*) – Number of agents that share their buffers. It is used for self-play.

sample(*batch_size*)

Sample transitions from the buffer. Adding next_action_mask to the batch for environments with legal moves.

hive.replays.prioritized_replay module

class hive.replays.prioritized_replay.**PrioritizedReplayBuffer**(*capacity*, *alpha=0.5*, *beta=0.5*, *stack_size=1*, *n_step=1*, *gamma=0.9*, *observation_shape=()*, *observation_dtype=<class 'numpy.uint8'>*, *action_shape=()*, *action_dtype=<class 'numpy.int8'>*, *reward_shape=()*, *reward_dtype=<class 'numpy.float32'>*, *extra_storage_types=None*, *num_players_sharing_buffer=None*)

Bases: *CircularReplayBuffer*

Implements a replay with prioritized sampling. See https://arxiv.org/abs/1511.05952

Parameters

- **capacity** (*int*) – Total number of observations that can be stored in the buffer. Note, this is not the same as the number of transitions that can be stored in the buffer.
- **alpha** (*float*) – Parameter controlling level of prioritization.
- **beta** (*float*) – Parameter controlling level of correction for prioritization.
- **stack_size** (*int*) – The number of frames to stack to create an observation.
- **n_step** (*int*) – Horizon used to compute n-step return reward
- **gamma** (*float*) – Discounting factor used to compute n-step return reward
- **observation_shape** (Tuple) – Shape of observations that will be stored in the buffer.
- **observation_dtype** (type) – Type of observations that will be stored in the buffer. This can either be the type itself or string representation of the type. The type can be either a native python type or a numpy type. If a numpy type, a string of the form np.uint8 or numpy.uint8 is acceptable.
- **action_shape** (Tuple) – Shape of actions that will be stored in the buffer.
- **action_dtype** (type) – Type of actions that will be stored in the buffer. Format is described in the description of observation_dtype.
- **action_shape** – Shape of actions that will be stored in the buffer.
- **action_dtype** – Type of actions that will be stored in the buffer. Format is described in the description of observation_dtype.
- **reward_shape** (Tuple) – Shape of rewards that will be stored in the buffer.
- **reward_dtype** (type) – Type of rewards that will be stored in the buffer. Format is described in the description of observation_dtype.
- **extra_storage_types** (*dict*) – A dictionary describing extra items to store in the buffer. The mapping should be from the name of the item to a (type, shape) tuple.
- **num_players_sharing_buffer** (*int*) – Number of agents that share their buffers. It is used for self-play.

set_beta(*beta*)

sample(*batch_size*)

Sample transitions from the buffer. For a given transition, if it's done is True, the next_observation value should not be taken to have any meaning.

Parameters

batch_size (*int*) – Number of transitions to sample.

update_priorities(*indices*, *priorities*)

Update the priorities of the transitions at the specified indices.

Parameters

- **indices** – Which transitions to update priorities for. Can be numpy array or torch tensor.
- **priorities** – What the priorities should be updated to. Can be numpy array or torch tensor.

save(*dname*)

Save the replay buffer.

Parameters
dname (*str*) – directory where to save buffer. Should already have been created.

load(*dname*)

Load the replay buffer.

Parameters
dname (*str*) – directory where to load buffer from.

class hive.replays.prioritized_replay.**SumTree**(*capacity*)

Bases: object

Data structure used to implement prioritized sampling. It is implemented as a tree where the value of each node is the sum of the values of the subtree of the node.

set_priority(*indices*, *priorities*)

Sets the priorities for the given indices.

Parameters

- **indices** (*np.ndarray*) – Which transitions to update priorities for.
- **priorities** (*np.ndarray*) – What the priorities should be updated to.

sample(*batch_size*)

Sample elements from the sum tree with probability proportional to their priority.

Parameters
batch_size (*int*) – The number of elements to sample.

stratified_sample(*batch_size*)

Performs stratified sampling using the sum tree.

Parameters
batch_size (*int*) – The number of elements to sample.

extract(*queries*)

Get the elements in the sum tree that correspond to the query. For each query, the element that is selected is the one with the greatest sum of "previous" elements in the tree, but also such that the sum is not a greater proportion of the total sum of priorities than the query.

Parameters
queries (*np.ndarray*) – Queries to extract. Each element should be between 0 and 1.

get_priorities(*indices*)

Get the priorities of the elements at indicies.

Parameters
indices (*np.ndarray*) – The indices to query.

save(*dname*)

load(*dname*)

hive.replays.recurrent_replay module

class hive.replays.recurrent_replay.**RecurrentReplayBuffer**(*capacity=10000*, *max_seq_len=1*, *n_step=1*, *gamma=0.99*, *observation_shape=()*, *observation_dtype=<class 'numpy.uint8'>*, *action_shape=()*, *action_dtype=<class 'numpy.int8'>*, *reward_shape=()*, *reward_dtype=<class 'numpy.float32'>*, *extra_storage_types=None*, *hidden_spec=None*, *num_players_sharing_buffer=None*)

Bases: *CircularReplayBuffer*

First implementation of recurrent buffer without storing hidden states

Constructor for RecurrentReplayBuffer.

Parameters

- **capacity** (*int*) – Total number of observations that can be stored in the buffer. Note, this is not the same as the number of transitions that can be stored in the buffer.
- **max_seq_len** (*int*) – The number of consecutive transitions in a sequence sampled from an episode.
- **n_step** (*int*) – Horizon used to compute n-step return reward
- **gamma** (*float*) – Discounting factor used to compute n-step return reward
- **observation_shape** – Shape of observations that will be stored in the buffer.
- **observation_dtype** – Type of observations that will be stored in the buffer. This can either be the type itself or string representation of the type. The type can be either a native python type or a numpy type. If a numpy type, a string of the form np.uint8 or numpy.uint8 is acceptable.
- **action_shape** – Shape of actions that will be stored in the buffer.
- **action_dtype** – Type of actions that will be stored in the buffer. Format is described in the description of observation_dtype.
- **reward_shape** – Shape of rewards that will be stored in the buffer.
- **reward_dtype** – Type of rewards that will be stored in the buffer. Format is described in the description of observation_dtype.
- **extra_storage_types** (*dict*) – A dictionary describing extra items to store in the buffer. The mapping should be from the name of the item to a (type, shape) tuple.
- **num_players_sharing_buffer** (*int*) – Number of agents that share their buffers. It is used for self-play.

add(*observation*, *next_observation*, *action*, *reward*, *terminated*, *truncated*, ***kwargs*)

Adds a transition to the buffer. The required components of a transition are given as positional arguments. The user can pass additional components to store in the buffer as kwargs as long as they were defined in the specification in the constructor.

sample(*batch_size*)

Sample transitions from the buffer. For a given transition, if it's done is True, the next_observation value should not be taken to have any meaning.

Parameters
batch_size (*int*) – Number of transitions to sample.

hive.replays.replay_buffer module

class hive.replays.replay_buffer.**BaseReplayBuffer**

Bases: ABC, *Registrable*

Base class for replay buffers. Every implemented buffer should be a subclass of this class.

abstract add(***data*)

Adds data to the buffer

Parameters
data – data to add to the replay buffer. Subclasses can define this class signature based on use case.

abstract sample(*batch_size*)

sample a minibatch

Parameters
batch_size (*int*) – the number of transitions to sample.

abstract size()

Returns the number of transitions stored in the buffer.

abstract save(*dname*)

Saves buffer checkpointing information to file for future loading.

Parameters
dname (*str*) – directory where agent should save all relevant info.

abstract load(*dname*)

Loads buffer from file.

Parameters
dname (*str*) – directory name where buffer checkpoint info is stored.

Returns
True if successfully loaded the buffer. False otherwise.

classmethod type_name()

Returns: "replay"

Module contents

hive.runners package

Submodules

hive.runners.base module

class hive.runners.base.**Runner**(*environment*, *agents*, *logger*, *experiment_manager*, *train_steps*, *eval_environment=None*, *test_frequency=-1*, *test_episodes=1*, *max_steps_per_episode=1000000000.0*)

Bases: ABC, *Registrable*

Base Runner class used to implement a training loop.

Different types of training loops can be created by overriding the relevant functions.

Parameters

- **environment** (BaseEnv) – Environment used in the training loop.
- **agents** (*list[Agent]*) – List of agents that interact with the environment.
- **logger** (ScheduledLogger) – Logger object used to log metrics.
- **experiment_manager** (Experiment) – Experiment object that saves the state of the training.
- **train_steps** (*int*) – How many steps to train for. This is the number of times that agent.update is called. If this is -1, there is no limit for the number of training steps.
- **test_frequency** (*int*) – After how many training steps to run testing episodes. If this is -1, testing is not run.
- **test_episodes** (*int*) – How many episodes to run testing for.

register_config(*config*)

train_mode(*training*)

If training is true, sets all agents to training mode. If training is false, sets all agents to eval mode.

Parameters

training (*bool*) – Whether to be in training mode.

create_episode_metrics()

Create the metrics used during the loop.

update_step()

Update steps for various schedules. Run testing if appropriate.

run_episode(*environment*)

Run a single episode of the environment.

Parameters

environment (BaseEnv) – Environment in which the agent will take a step in.

run_training()

Run the training loop. Note, to ensure that the test phase is run during the individual runners must call *update_step()* in their *run_episode()* methods. See *SingleAgentRunner* and *MultiAgentRunner* for examples.

run_testing()

Run a testing phase.

resume()

Resume a saved experiment.

classmethod type_name()

Returns
"runner"

hive.runners.multi_agent_loop module

class hive.runners.multi_agent_loop.**MultiAgentRunner**(*environment*, *agents*, *loggers*, *experiment_manager*, *train_steps*, *num_agents*, *eval_environment=None*, *test_frequency=-1*, *test_episodes=1*, *stack_size=1*, *self_play=False*, *max_steps_per_episode=1000000000.0*, *seed=None*)

Bases: *Runner*

Runner class used to implement a multiagent training loop.

Initializes the MultiAgentRunner object.

Parameters

- **environment** (BaseEnv) – Environment used in the training loop.
- **agent** (Agent) – Agent that will interact with the environment
- **loggers** (*List[ScheduledLogger]*) – List of loggers used to log metrics.
- **experiment_manager** (Experiment) – Experiment object that saves the state of the training.
- **train_steps** (*int*) – How many steps to train for. This is the number of times that agent.update is called. If this is -1, there is no limit for the number of training steps.
- **num_agents** (*int*) – Number of agents running in this multiagent experiment.
- **eval_environment** (BaseEnv) – Environment used to evaluate the agent. If None, the environment parameter (which is a function) is used to create a second environment.
- **test_frequency** (*int*) – After how many training steps to run testing episodes. If this is -1, testing is not run.
- **test_episodes** (*int*) – How many episodes to run testing for duing each test phase.
- **stack_size** (*int*) – The number of frames in an observation sent to an agent.
- **self_play** (*bool*) – Whether this multiagent experiment is run in self-play mode. In this mode, only the first agent in the list of agents provided in the config is created. This agent performs actions for each player in the multiagent environment.
- **max_steps_per_episode** (*int*) – The maximum number of steps to run an episode for.
- **seed** (*int*) – Seed used to set the global seed for libraries used by Hive and seed the *Seeder*.

run_one_step(*environment*, *observation*, *turn*, *episode_metrics*, *transition_info*, *agent_traj_states*)

Run one step of the training loop.

If it is the agent's first turn during the episode, do not run an update step. Otherwise, run an update step based on the previous action and accumulated reward since then.

Parameters

- **environment** (BaseEnv) – Environment in which the agent will take a step in.
- **observation** – Current observation that the agent should create an action for.
- **turn** (*int*) – Agent whose turn it is.
- **episode_metrics** (Metrics) – Keeps track of metrics for current episode.
- **transition_info** (TransitionInfo) – Used to keep track of the most recent transition for each agent.
- **agent_traj_states** – List of trajectory state objects that will be passed to each agent when act and update are called. The agent returns new trajectory states to replace the state passed in.

run_end_step(*episode_metrics*, *transition_info*, *agent_traj_states*, *terminated=True*, *truncated=False*)

Run the final step of an episode.

After an episode ends, iterate through agents and update then with the final step in the episode.

Parameters

- **episode_metrics** (Metrics) – Keeps track of metrics for current episode.
- **transition_info** (TransitionInfo) – Used to keep track of the most recent transition for each agent.
- **agent_traj_states** – List of trajectory state objects that will be passed to each agent when act and update are called. The agent returns new trajectory states to replace the state passed in.
- **terminated** (*bool*) – Whether this step was terminal.
- **truncated** (*bool*) – Whether this step was terminal.

run_episode(*environment*)

Run a single episode of the environment.

Parameters

environment (BaseEnv) – Environment in which the agent will take a step in.

hive.runners.single_agent_loop module

class hive.runners.single_agent_loop.**SingleAgentRunner**(*environment*, *agent*, *loggers*, *experiment_manager*, *train_steps*, *eval_environment=None*, *test_frequency=-1*, *test_episodes=1*, *stack_size=1*, *max_steps_per_episode=1000000000.0*, *seed=None*)

Bases: *Runner*

Runner class used to implement a sinle-agent training loop.

Initializes the SingleAgentRunner.

Parameters

- **environment** (BaseEnv) – Environment used in the training loop.
- **agent** (Agent) – Agent that will interact with the environment
- **loggers** (*List[ScheduledLogger]*) – List of loggers used to log metrics.
- **experiment_manager** (Experiment) – Experiment object that saves the state of the training.
- **train_steps** (*int*) – How many steps to train for. This is the number of times that agent.update is called. If this is -1, there is no limit for the number of training steps.
- **eval_environment** (BaseEnv) – Environment used to evaluate the agent. If None, the environment parameter (which is a function) is used to create a second environment.
- **test_frequency** (*int*) – After how many training steps to run testing episodes. If this is -1, testing is not run.
- **test_episodes** (*int*) – How many episodes to run testing for duing each test phase.
- **stack_size** (*int*) – The number of frames in an observation sent to an agent.
- **max_steps_per_episode** (*int*) – The maximum number of steps to run an episode for.
- **seed** (*int*) – Seed used to set the global seed for libraries used by Hive and seed the *Seeder*.

run_one_step(*environment*, *observation*, *episode_metrics*, *transition_info*, *agent_traj_state*)

Run one step of the training loop.

Parameters

- **observation** – Current observation that the agent should create an action for.
- **episode_metrics** (Metrics) – Keeps track of metrics for current episode.

run_end_step(*environment*, *observation*, *episode_metrics*, *transition_info*, *agent_traj_state*)

Run the final step of an episode.

After an episode ends, set the truncated value to true.

Parameters

- **environment** (BaseEnv) – Environment in which the agent will take a step in.
- **observation** – Current observation that the agent should create an action for.
- **episode_metrics** (Metrics) – Keeps track of metrics for current episode.
- **transition_info** (TransitionInfo) – Used to keep track of the most recent transition for the agent.
- **agent_traj_state** – Trajectory state object that will be passed to the agent when act and update are called. The agent returns a new trajectory state object to replace the state passed in.

run_episode(*environment*)

Run a single episode of the environment.

Parameters

environment (BaseEnv) – Environment in which the agent will take a step in.

hive.runners.utils module

hive.runners.utils.**load_config**(*config=None*, *preset_config=None*, *agent_config=None*, *env_config=None*, *logger_config=None*)

Used to load config for experiments. Agents, environment, and loggers components in main config file can be overrided based on other log files.

Parameters

- **config** (*str*) – Path to configuration file. Either this or `preset_config` must be passed.
- **preset_config** (*str*) – Path to a preset hive config. This path should be relative to `hive/configs`. For example, the Atari DQN config would be `atari/dqn.yml`.
- **agent_config** (*str*) – Path to agent configuration file. Overrides settings in base config.
- **env_config** (*str*) – Path to environment configuration file. Overrides settings in base config.
- **logger_config** (*str*) – Path to logger configuration file. Overrides settings in base config.

class hive.runners.utils.**Metrics**(*agents*, *agent_metrics*, *episode_metrics*)

Bases: `object`

Class used to keep track of separate metrics for each agent as well general episode metrics.

Parameters

- **agents** (*list[Agent]*) – List of agents for which object will track metrics.
- **agent_metrics** (*list[(str, (callable | obj))]*) – List of metrics to track for each agent. Should be a list of tuples (metric_name, metric_init) where metric_init is either the initial value of the metric or a callable that takes no arguments and creates the initial metric.
- **episode_metrics** (*list[(str, (callable | obj))]*) – List of non agent specific metrics to keep track of. Should be a list of tuples (metric_name, metric_init) where metric_init is either the initial value of the metric or a callable with no arguments that creates the initial metric.

reset_metrics()

Resets all metrics to their initial values.

get_flat_dict()

Get a flat dictionary version of the metrics. Each agent metric will be prefixed by the agent id.

class hive.runners.utils.**TransitionInfo**(*agents*, *stack_size*)

Bases: `object`

Used to keep track of the most recent transition for each agent.

Any info that the agent needs to remember for updating can be stored here. Should be completely reset between episodes. After any info is extracted, it is automatically removed from the object. Also keeps track of which agents have started their episodes.

This object also handles padding and stacking observations for agents.

Parameters

- **agents** (*list[Agent]*) – list of agents that will be kept track of.
- **stack_size** (*int*) – How many observations will be stacked.

`reset()`

Reset the object by clearing all info.

`is_started(agent)`

Check if agent has started its episode.

Parameters
agent (Agent) – Agent to check.

`start_agent(agent)`

Set the agent's start flag to true.

Parameters
agent (Agent) – Agent to start.

`record_info(agent, info)`

Update some information for the agent.

Parameters

- **agent** (Agent) – Agent to update.
- **info** (*dict*) – Info to add to the agent's state.

`update_reward(agent, reward)`

Add a reward to the agent.

Parameters

- **agent** (Agent) – Agent to update.
- **reward** (*float*) – Reward to add to agent.

`update_all_rewards(rewards)`

Update the rewards for all agents. If rewards is list, it updates the rewards according to the order of agents provided in the initializer. If rewards is a dict, the keys should be the agent ids for the agents and the values should be the rewards for those agents. If rewards is a float or int, every agent is updated with that reward.

Parameters
rewards (*float | list | np.ndarray | dict*) – Rewards to update agents with.

`get_info(agent, terminated=False, truncated=False)`

Get all the info for the agent, and reset the info for that agent. Also adds a done value to the info dictionary that is based on the done parameter to the function.

Parameters

- **agent** (Agent) – Agent to get transition update info for.
- **done** (*bool*) – Whether this transition is terminal.

`get_stacked_state(agent, observation)`

Create a stacked state for the agent. The previous observations recorded by this agent are stacked with the current observation. If not enough observations have been recorded, zero arrays are appended.

Parameters

- **agent** (Agent) – Agent to get stacked state for.
- **observation** – Current observation.

hive.runners.utils.**zeros_like**(*x*)

Create a zero state like some state. This handles slightly more complex objects such as lists and dictionaries of numpy arrays and torch Tensors.

Parameters

x (*np.ndarray | torch.Tensor | dict | list*) – State used to define structure/state of zero state.

hive.runners.utils.**concatenate**(*xs*)

Concatenates numpy arrays or dictionaries of numpy arrays.

Parameters

xs (*list*) – List of objects to concatenate.

Module contents

hive.utils package

Submodules

hive.utils.experiment module

Implementation of a simple experiment class.

class hive.utils.experiment.**Experiment**(*name*, *save_dir*, *saving_schedule*)

Bases: *Registrable*

Implementation of a simple experiment class.

Initializes an experiment object.

The experiment state is an exposed property of objects of this class. It can be used to keep track of objects that need to be saved to keep track of the experiment, but don't fit in one of the standard categories. One example of this is the various schedules used in the Runner class.

Parameters

- **name** (*str*) – Name of the experiment.
- **dir_name** (*str*) – Absolute path to the directory to save/load the experiment.
- **saving_schedule** (Schedule) – Schedule that determines when the experiment should be saved.

register_experiment(***kwargs*)

Registers all the components of an experiment.

Parameters

- **logger** (Logger) – a logger object.
- **agents** (Agent | *list*[Agent]) – either an agent object or a list of agents.
- **environment** (BaseEnv) – an environment object.

register_config(*config*)

Registers the experiment config.

Parameters

config (Chomp) – a config dictionary.

update_step()

Updates the step of the saving schedule for the experiment.

should_save()

Returns whether you should save the experiment at the current step.

save(*tag='current'*)

Saves the experiment. :param tag: Tag to prefix the folder. :type tag: str

is_resumable(*tag='current'*)

Returns true if the experiment is resumable.

Parameters
tag (*str*) – Tag for the saved experiment.

resume(*tag='current'*)

Resumes the experiment from a checkpoint.

Parameters
tag (*str*) – Tag for the saved experiment.

classmethod type_name()

Returns
"experiment_manager"

hive.utils.experiment.**save_component**(*component*, *prefix*)

hive.utils.experiment.**load_component**(*component*, *prefix*)

hive.utils.loggers module

class hive.utils.loggers.**Logger**(*timescales=None*)

Bases: ABC, *Registrable*

Abstract class for logging in hive.

Constructor for base Logger class. Every Logger must call this constructor in its own constructor

Parameters
timescales (*str | list(str)*) – The different timescales at which logger needs to log. If only logging at one timescale, it is acceptable to only pass a string.

register_timescale(*timescale*)

Register a new timescale with the logger.

Parameters
timescale (*str*) – Timescale to register.

abstract log_config(*config*)

Log the config.

Parameters
config (*dict*) – Config parameters.

abstract log_scalar(*name*, *value*, *prefix*)

Log a scalar variable.

Parameters

- **name** (*str*) – Name of the metric to be logged.
- **value** (*float*) – Value to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

abstract log_metrics(*metrics*, *prefix*)

Log a dictionary of values.

Parameters

- **metrics** (*dict*) – Dictionary of metrics to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

abstract save(*dir_name*)

Saves the current state of the log files.

Parameters
dir_name (*str*) – Name of the directory to save the log files.

abstract load(*dir_name*)

Loads the log files from given directory.

Parameters
dir_name (*str*) – Name of the directory to load the log file from.

classmethod type_name()

This should represent a string that denotes the which type of class you are creating. For example, "logger", "agent", or "env".

class hive.utils.loggers.**ScheduledLogger**(*timescales=None*, *logger_schedules=None*)

Bases: *Logger*

Abstract class that manages a schedule for logging.

The update_step method should be called for each step in the loop to update the logger's schedule. The should_log method can be used to check whether the logger should log anything.

This schedule is not strictly enforced! It is still possible to log something even if should_log returns false. These functions are just for the purpose of convenience.

Any timescales not assigned schedule from logger_schedules will be assigned a ConstantSchedule(True).

Parameters

- **timescales** (*str*|*list[str]*) – The different timescales at which logger needs to log. If only logging at one timescale, it is acceptable to only pass a string.
- **logger_schedules** (Schedule|*list*|*dict*) – Schedules used to keep track of when to log. If a single schedule, it is copied for each timescale. If a list of schedules, the schedules are matched up in order with the list of timescales provided. If a dictionary, the keys should be the timescale and the values should be the schedule.

register_timescale(*timescale*, *schedule=None*)

Register a new timescale.

Parameters

- **timescale** (*str*) – Timescale to register.
- **schedule** (Schedule) – Schedule to use for this timescale.

update_step(*timescale*)

Update the step and schedule for a given timescale.

Parameters
timescale (*str*) – A registered timescale.

should_log(*timescale*)

Check if you should log for a given timescale.

Parameters
timescale (*str*) – A registered timescale.

save(*dir_name*)

Saves the current state of the log files.

Parameters
dir_name (*str*) – Name of the directory to save the log files.

load(*dir_name*)

Loads the log files from given directory.

Parameters
dir_name (*str*) – Name of the directory to load the log file from.

class hive.utils.loggers.**NullLogger**(*timescales=None*, *logger_schedules=None*)

Bases: *ScheduledLogger*

A null logger that does not log anything.

Used if you don't want to log anything, but still want to use parts of the framework that ask for a logger.

Any timescales not assigned schedule from logger_schedules will be assigned a ConstantSchedule(True).

Parameters

- **timescales** (*str* | *list[str]*) – The different timescales at which logger needs to log. If only logging at one timescale, it is acceptable to only pass a string.
- **logger_schedules** (Schedule | *list* | *dict*) – Schedules used to keep track of when to log. If a single schedule, it is copied for each timescale. If a list of schedules, the schedules are matched up in order with the list of timescales provided. If a dictionary, the keys should be the timescale and the values should be the schedule.

log_config(*config*)

Log the config.

Parameters
config (*dict*) – Config parameters.

log_scalar(*name*, *value*, *timescale*)

Log a scalar variable.

Parameters

- **name** (*str*) – Name of the metric to be logged.
- **value** (*float*) – Value to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

log_metrics(*metrics*, *timescale*)

Log a dictionary of values.

Parameters

- **metrics** (*dict*) – Dictionary of metrics to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

save(*dir_name*)

Saves the current state of the log files.

Parameters
dir_name (*str*) – Name of the directory to save the log files.

load(*dir_name*)

Loads the log files from given directory.

Parameters
dir_name (*str*) – Name of the directory to load the log file from.

class hive.utils.loggers.**WandbLogger**(*timescales=None*, *logger_schedules=None*, *project=None*, *name=None*, *dir=None*, *mode=None*, *id=None*, *resume=None*, *start_method=None*, ***kwargs*)

Bases: *ScheduledLogger*

A Wandb logger.

This logger can be used to log to wandb. It assumes that wandb is configured locally on your system. Multiple timescales/loggers can be implemented by instantiating multiple loggers with different logger_names. These should still have the same project and run names.

Check the wandb documentation for more details on the parameters.

Parameters

- **timescales** (*str|list[str]*) – The different timescales at which logger needs to log. If only logging at one timescale, it is acceptable to only pass a string.
- **logger_schedules** (*Schedule|list|dict*) – Schedules used to keep track of when to log. If a single schedule, it is copied for each timescale. If a list of schedules, the schedules are matched up in order with the list of timescales provided. If a dictionary, the keys should be the timescale and the values should be the schedule.
- **project** (*str*) – Name of the project. Wandb's dash groups all runs with the same project name together.
- **name** (*str*) – Name of the run. Used to identify the run on the wandb dash.
- **dir** (*str*) – Local directory where wandb saves logs.
- **mode** (*str*) – The mode of logging. Can be "online", "offline" or "disabled". In offline mode, writes all data to disk for later syncing to a server, while in disabled mode, it makes all calls to wandb api's noop's, while maintaining core functionality.
- **id** (*str, optional*) – A unique ID for this run, used for resuming. It must be unique in the project, and if you delete a run you can't reuse the ID.
- **resume** (*bool, str, optional*) – Sets the resuming behavior. Options are the same as mentioned in Wandb's doc.
- **start_method** (*str*) – The start method to use for wandb's process. See https://docs.wandb.ai/guides/track/launch#init-start-error.
- ****kwargs** – You can pass any other arguments to wandb's init method as keyword arguments. Note, these arguments can't be overriden from the command line.

`log_config`(*config*)

Log the config.

Parameters
config (*dict*) – Config parameters.

`log_scalar`(*name*, *value*, *prefix*)

Log a scalar variable.

Parameters

- **name** (*str*) – Name of the metric to be logged.
- **value** (*float*) – Value to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

`log_metrics`(*metrics*, *prefix*)

Log a dictionary of values.

Parameters

- **metrics** (*dict*) – Dictionary of metrics to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

`class hive.utils.loggers.ChompLogger`(*timescales=None*, *logger_schedules=None*)

Bases: *ScheduledLogger*

This logger uses the Chomp data structure to store all logged values which are then directly saved to disk.

Any timescales not assigned schedule from logger_schedules will be assigned a ConstantSchedule(True).

Parameters

- **timescales** (*str|list[str]*) – The different timescales at which logger needs to log. If only logging at one timescale, it is acceptable to only pass a string.
- **logger_schedules** (*Schedule|list|dict*) – Schedules used to keep track of when to log. If a single schedule, it is copied for each timescale. If a list of schedules, the schedules are matched up in order with the list of timescales provided. If a dictionary, the keys should be the timescale and the values should be the schedule.

`log_config`(*config*)

Log the config.

Parameters
config (*dict*) – Config parameters.

`log_scalar`(*name*, *value*, *prefix*)

Log a scalar variable.

Parameters

- **name** (*str*) – Name of the metric to be logged.
- **value** (*float*) – Value to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

`log_metrics`(*metrics*, *prefix*)

Log a dictionary of values.

Parameters

- **metrics** (*dict*) – Dictionary of metrics to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

save(*dir_name*)

Saves the current state of the log files.

Parameters
dir_name (*str*) – Name of the directory to save the log files.

load(*dir_name*)

Loads the log files from given directory.

Parameters
dir_name (*str*) – Name of the directory to load the log file from.

class hive.utils.loggers.**CompositeLogger**(*logger_list*)

Bases: *Logger*

This Logger aggregates multiple loggers together.

This logger is for convenience and allows for logging using multiple loggers without having to keep track of several loggers. When timescales are updated, this logger updates the timescale for each one of its component loggers. When logging, logs to each of its component loggers as long as the logger is not a ScheduledLogger that should not be logging for the timescale.

Constructor for base Logger class. Every Logger must call this constructor in its own constructor

Parameters
timescales (*str | list(str)*) – The different timescales at which logger needs to log. If only logging at one timescale, it is acceptable to only pass a string.

register_timescale(*timescale*, *schedule=None*)

Register a new timescale with the logger.

Parameters
timescale (*str*) – Timescale to register.

log_config(*config*)

Log the config.

Parameters
config (*dict*) – Config parameters.

log_scalar(*name*, *value*, *prefix*)

Log a scalar variable.

Parameters

- **name** (*str*) – Name of the metric to be logged.
- **value** (*float*) – Value to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

log_metrics(*metrics*, *prefix*)

Log a dictionary of values.

Parameters

- **metrics** (*dict*) – Dictionary of metrics to be logged.
- **prefix** (*str*) – Prefix to append to metric name.

update_step(*timescale*)

Update the step and schedule for a given timescale for every ScheduledLogger.

Parameters
timescale (*str*) – A registered timescale.

should_log(*timescale*)

Check if you should log for a given timescale. If any logger in the list is scheduled to log, returns True.

Parameters
timescale (*str*) – A registered timescale.

save(*dir_name*)

Saves the current state of the log files.

Parameters
dir_name (*str*) – Name of the directory to save the log files.

load(*dir_name*)

Loads the log files from given directory.

Parameters
dir_name (*str*) – Name of the directory to load the log file from.

hive.utils.registry module

class hive.utils.registry.**Registrable**

Bases: object

Class used to denote which types of objects can be registered in the RLHive Registry. These objects can also be configured directly from the command line, and recursively built from the config, assuming type annotations are present.

classmethod type_name()

This should represent a string that denotes the which type of class you are creating. For example, "logger", "agent", or "env".

class hive.utils.registry.**Registry**

Bases: object

This is the Registry class for RLHive. It allows you to register different types of *Registrable* classes and objects and generates constructors for those classes in the form of *get_{type_name}*.

These constructors allow you to construct objects from dictionary configs. These configs should have two fields: *name*, which corresponds to the name used when registering a class in the registry, and *kwargs*, which corresponds to the keyword arguments that will be passed to the constructor of the object. These constructors can also build objects recursively, i.e. if a config contains the config for another *Registrable* object, this will be automatically created before being passed to the constructor of the original object. These constructors also allow you to directly specify/override arguments for object constructors directly from the command line. These parameters are specified in dot notation. They also are able to handle lists and dictionaries of Registrable objects.

For example, let's consider the following scenario: Your agent class has an argument *arg1* which is annotated to be *List[Class1]*, *Class1* is *Registrable*, and the *Class1* constructor takes an argument *arg2*. In the passed yml config, there are two different Class1 object configs listed. the constructor will check to see if both *–agent.arg1.0.arg2* and *–agent.arg1.1.arg2* have been passed.

The parameters passed in the command line will be parsed according to the type annotation of the corresponding low level constructor. If it is not one of *int*, *float*, *str*, or *bool*, it simply loads the string into python using a yaml loader.

Each constructor returns the object, as well a dictionary config with all the parameters used to create the object and any Registrable objects created in the process of creating this object.

register(*name*, *constructor*, *type*)

Register a Registrable class/object with RLHive.

Parameters

- **name** (*str*) – Name of the class/object being registered.
- **constructor** (*callable*) – Callable that will be passed all kwargs from configs and be analyzed to get type annotations.
- **type** (*type*) – Type of class/object being registered. Should be subclass of Registrable.

register_all(*base_class*, *class_dict*)

Bulk register function.

Parameters

- **base_class** (*type*) – Corresponds to the *type* of the register function
- **class_dict** (*dict[str, callable]*) – A dictionary mapping from name to constructor.

get_SequenceFn(*object_or_config*, *prefix=None*)

get_SequenceModel(*object_or_config*, *prefix=None*)

get_activation_fn(*object_or_config*, *prefix=None*)

get_advantage_computation_fn(*object_or_config*, *prefix=None*)

get_agent(*object_or_config*, *prefix=None*)

get_env(*object_or_config*, *prefix=None*)

get_env_wrapper(*object_or_config*, *prefix=None*)

get_experiment_manager(*object_or_config*, *prefix=None*)

get_function(*object_or_config*, *prefix=None*)

get_init_fn(*object_or_config*, *prefix=None*)

get_logger(*object_or_config*, *prefix=None*)

get_loss_fn(*object_or_config*, *prefix=None*)

get_norm_fn(*object_or_config*, *prefix=None*)

get_optimizer_fn(*object_or_config*, *prefix=None*)

get_replay(*object_or_config*, *prefix=None*)

get_runner(*object_or_config*, *prefix=None*)

get_schedule(*object_or_config*, *prefix=None*)

hive.utils.registry.**construct_objects**(*object_constructor*, *config*, *prefix=None*)

Helper function that constructs any objects specified in the config that are registrable.

Returns the object, as well a dictionary config with all the parameters used to create the object and any Registrable objects created in the process of creating this object.

Parameters

- **object_constructor** (*callable*) – constructor of object that corresponds to config. The signature of this function will be analyzed to see if there are any *Registrable* objects that might be specified in the config.
- **config** (*dict*) – The kwargs for the object being created. May contain configs for other *Registrable* objects that need to be recursively created.
- **prefix** (*str*) – Prefix that is attached to the argument names when looking for command line arguments.

hive.utils.registry.**get_callable_parsed_args**(*callable*, *prefix=None*)

Helper function that extracts the command line arguments for a given function.

Parameters

- **callable** (*callable*) – function whose arguments will be inspected to extract arguments from the command line.
- **prefix** (*str*) – Prefix that is attached to the argument names when looking for command line arguments.

hive.utils.registry.**get_parsed_args**(*arguments*, *prefix=None*)

Helper function that takes a dictionary mapping argument names to types, and extracts command line arguments for those arguments. If the dictionary contains a key-value pair "bar": int, and the prefix passed is "foo", this function will look for a command line argument "–foo.bar". If present, it will cast it to an int.

If the type for a given argument is not one of *int*, *float*, *str*, or *bool*, it simply loads the string into python using a yaml loader.

Parameters

- **arguments** (*dict[str, type]*) – dictionary mapping argument names to types
- **prefix** (*str*) – prefix that is attached to each argument name before searching for command line arguments.

hive.utils.schedule module

class hive.utils.schedule.**Schedule**

Bases: ABC, *Registrable*

abstract get_value()

Returns the current value of the variable we are tracking

abstract update()

Update the value of the variable we are tracking and return the updated value. The first call to update will return the initial value of the schedule.

classmethod type_name()

This should represent a string that denotes the which type of class you are creating. For example, "logger", "agent", or "env".

class hive.utils.schedule.**LinearSchedule**(*init_value*, *end_value*, *steps*)

Bases: *Schedule*

Defines a linear schedule between two values over some number of steps.

If updated more than the defined number of steps, the schedule stays at the end value.

Parameters

- **init_value** (*int* | *float*) – Starting value for schedule.
- **end_value** (*int* | *float*) – End value for schedule.
- **steps** (*int*) – Number of steps for schedule. Should be positive.

get_value()

Returns the current value of the variable we are tracking

update()

Update the value of the variable we are tracking and return the updated value. The first call to update will return the initial value of the schedule.

class hive.utils.schedule.**ConstantSchedule**(*value*)

Bases: *Schedule*

Returns a constant value over the course of the schedule

Parameters

value – The value to be returned.

get_value()

Returns the current value of the variable we are tracking

update()

Update the value of the variable we are tracking and return the updated value. The first call to update will return the initial value of the schedule.

class hive.utils.schedule.**SwitchSchedule**(*off_value*, *on_value*, *steps*)

Bases: *Schedule*

Returns one value for the first part of the schedule. After the defined number of steps is reached, switches to returning a second value.

Parameters

- **off_value** – The value to be returned in the first part of the schedule.
- **on_value** – The value to be returned in the second part of the schedule.
- **steps** (*int*) – The number of steps after which to switch from the off value to the on value.

get_value()

Returns the current value of the variable we are tracking

update()

Update the value of the variable we are tracking and return the updated value. The first call to update will return the initial value of the schedule.

class hive.utils.schedule.**DoublePeriodicSchedule**(*off_value*, *on_value*, *off_period*, *on_period*)

Bases: *Schedule*

Returns off value for off period, then switches to returning on value for on period. Alternates between the two.

Parameters

- **on_value** – The value to be returned for the on period.
- **off_value** – The value to be returned for the off period.
- **on_period** (*int*) – the number of steps in the on period.
- **off_period** (*int*) – the number of steps in the off period.

get_value()

Returns the current value of the variable we are tracking

update()

Update the value of the variable we are tracking and return the updated value. The first call to update will return the initial value of the schedule.

class hive.utils.schedule.**PeriodicSchedule**(*off_value*, *on_value*, *period*)

Bases: *DoublePeriodicSchedule*

Returns one value on the first step of each period of a predefined number of steps. Returns another value otherwise.

Parameters

- **on_value** – The value to be returned on the first step of each period.
- **off_value** – The value to be returned for every other step in the period.
- **period** (*int*) – The number of steps in the period.

hive.utils.torch_utils module

hive.utils.torch_utils.**numpify**(*t*)

Convert object to a numpy array.

Parameters

t (*np.ndarray | torch.Tensor | obj*) – Converts object to `np.ndarray`.

class hive.utils.torch_utils.**RMSpropTF**(*params*, *lr=0.01*, *alpha=0.9*, *eps=1e-10*, *weight_decay=0*, *momentum=0.0*, *centered=False*, *decoupled_decay=False*, *lr_in_momentum=True*)

Bases: `Optimizer`

Direct cut-paste from rwhightman/pytorch-image-models. https://github.com/rwightman/pytorch-image-models/blob/f7d210d759beb00a3d0834a3ce2d93f6e17f3d38/timm/optim/rmsprop_tf.py
Licensed under Apache 2.0, https://github.com/rwightman/pytorch-image-models/blob/master/LICENSE

Implements RMSprop algorithm (TensorFlow style epsilon)

NOTE: This is a direct cut-and-paste of PyTorch RMSprop with eps applied before sqrt and a few other modifications to closer match Tensorflow for matching hyper-params. Noteworthy changes include:

1. Epsilon applied inside square-root
2. square_avg initialized to ones
3. LR scaling of update accumulated in momentum buffer

Proposed by G. Hinton in his course. The centered version first appears in Generating Sequences With Recurrent Neural Networks.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups

- **lr** (*float, optional*) – learning rate (default: 1e-2)
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **alpha** (*float, optional*) – smoothing (decay) constant (default: 0.9)
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-10)
- **centered** (*bool, optional*) – if `True`, compute the centered RMSProp, the gradient is normalized by an estimation of its variance
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **decoupled_decay** (*bool, optional*) – decoupled weight decay as per https://arxiv.org/abs/1711.05101
- **lr_in_momentum** (*bool, optional*) – learning rate scaling is included in the momentum buffer update as per defaults in Tensorflow

step(*closure=None*)

Performs a single optimization step.

Parameters

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

hive.utils.utils module

hive.utils.utils.**create_folder**(*folder*)

Creates a folder.

Parameters

folder (*str*) – Folder to create.

class hive.utils.utils.**Seeder**

Bases: `object`

Class used to manage seeding in RLHive. It sets the seed for all the frameworks that RLHive currently uses. It also deterministically provides new seeds based on the global seed, in case any other objects in RLHive (such as the agents) need their own seed.

set_global_seed(*seed*)

This reduces some sources of randomness in experiments. To get reproducible results, you must run on the same machine and set the environment variable CUBLAS_WORKSPACE_CONFIG to ":4096:8" or ":16:8" before starting the experiment.

Parameters

seed (*int*) – Global seed.

get_new_seed(*group=None*)

Each time it is called, it increments the current_seed for the requested group and returns it. If no group is specified, the default group is selected.

Parameters

group (*str*) – The name of the group to get the seed for.

class hive.utils.utils.**Chomp**

Bases: `dict`

An extension of the dictionary class that allows for accessing through dot notation and easy saving/loading.

save(*filename*)

Saves the object using pickle.

Parameters
filename (*str*) – Filename to save object.

load(*filename*)

Loads the object.

Parameters
filename (*str*) – Where to load object from.

class hive.utils.utils.**OptimizerFn**

Bases: *Registrable*

A wrapper for callables that produce optimizer functions.

These wrapped callables can be partially initialized through configuration files or command line arguments.

classmethod type_name()

Returns
"optimizer_fn"

class hive.utils.utils.**LossFn**

Bases: *Registrable*

A wrapper for callables that produce loss functions.

These wrapped callables can be partially initialized through configuration files or command line arguments.

classmethod type_name()

Returns
"loss_fn"

class hive.utils.utils.**ActivationFn**

Bases: *Registrable*

A wrapper for callables that produce activation functions.

These wrapped callables can be partially initialized through configuration files or command line arguments.

classmethod type_name()

Returns
"activation_fn"

hive.utils.visualization module

Module contents

3.1.2 Submodules

hive.main module

hive.main.**main**()

3.1.3 Module contents

CHAPTER

FOUR

NOTES

4.1 Reproducibility

Achieving reproducibility in deep RL is difficult. Even when the random seed is fixed, libraries such as PyTorch use algorithms and implementations that are nondeterministic. PyTorch has several options that allow the user to turn off some aspects of this nondeterminism, **but behavior is still usually only replicable if the runs are executed on the same hardware**.

We provide a global seeding class *Seeder* that allows the user to set a global seed for all packages currently used by the framework (NumPy, PyTorch, and Python's random package). It also sets the PyTorch options to turn off nondeterminism. When using this seeding functionality, before starting a run, you must set the environment variable `CUBLAS_WORKSPACE_CONFIG` to either `":16:8"` (limits performance) or `":4096:8"` (uses slightly more memory). See this page for more details.

The *Seeder* class also provides a function *get_new_seed()* that provides a new random seed each time it is called, which is useful when in multi-agent setups where you want each agent to be seeded differently.

4.2 Roadmap

We have quite a lot planned for RLHive, so stay tuned! Here is what we are currently planning to add:

- Policy gradient methods (PPO)
- RNN support
- rliable
- Lifelong RL support
- JAX support
- Continuous Actions
- Dreamerv2/MBRL
- Better config system
- RL Debugging tools
- Log videos of trajectories
- Allow changing objects from command line
- Hyperparameter search integration
- Add logger based on Python logging

If you have a feature request that isn't listed here, check out our *Contributing* page!

CHAPTER

FIVE

CONTRIBUTING

5.1 Core RLHive

Did you spot a bug, or is there something that you think should be added to the main RLHive package? Check our Issues on Github or our *roadmap* to see if it's already on our radar. If not, you can either open an issue to let us know, or you can fork our repo and create a pull request with your own feature/bug fix.

5.2 Creating Issues

We'd love to hear from you on how we can improve RLHive! When creating issues, please follow these guidelines:

- Tag your issue with bug, feature request, or question to help us effectively sort through the issues.
- Include the version of RLHive you are running (run `pip list | grep rlhive`)

5.3 Creating PRs

When contributing to RLHive, please follow these guidelines:

- Create a separate PR for each feature you are adding.
- When you are done writing the feature, create a pull request to the dev branch of the main repo.
- Each pull request must pass all unit tests and linting checks before being merged. Please run these checks locally before creating PRs/pushing changes to the PR to minimize unnecessary runs on our CI system. You can run the following commands from the root directory of the project:
  - Unit Tests: `python -m pytest tests/`
  - Linter: `black .`
- Information (such as installation instructions and editor integrations) for the black formatter is available here.
- Make sure your code is documented using Google style docstrings. For examples, see the rest of our repository.

5.4 Contrib RLHive

We want to encourage users to contribute their own custom components to RLHive. This could be things like agents, environments, runners, replays, optimizers, and anything else. This will allow everyone to easily use and build on your work.

To do this, we have a `contrib` directory that will be part of the package. After adding new components and any relevant citation information to this folder, new versions of RLHive will be updated with these components, allowing your work to get a potentially larger audience. Note, we will be actively maintaining only the code in the main package, not the `contrib` package. We can only commit to giving minimal feedback during the review stage. If a contribution becomes widely adopted by the community, we may move it to the main repository to actively maintain.

When submitting the PR for your contributions, you must provide some results with your new components **that were generated with the RLHive package**, to provide us evidence of the correctness of your implementation.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

h

INDEX

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z