reinforced-lib

Maksymilian Wojnar, Wojciech Ciężobka

Apr 13, 2024
# GUIDES

1 Key components ................................. 3
2 JAX Backend .................................. 5
3 Edge Device Export ......................... 7
4 Table of Contents .............................. 9
   4.1 Guides .................................. 9
   4.2 API Documentation ...................... 9
   4.3 Indices and tables ...................... 102
5 Citing Reinforced-lib ....................... 103

Python Module Index .......................... 105
Index .............................................. 107
Introducing Reinforced-lib: a lightweight Python library for rapid development of reinforcement-learning (RL) solutions. It is open-source, prioritizes ease of use, provides comprehensive documentation, and offers both deep reinforcement learning (DRL) and classic non-neural agents. Built on JAX, it facilitates exporting trained models to embedded devices, and makes it great for research and prototyping with RL algorithms. Access to JAX’s JIT functionality ensure high-performance results.
Reinforced-lib facilitates seamless interaction between RL agents and the environment. Here are the key components within the library, represented in the API as different modules.

- **RLib** – The core module which provides a simple and intuitive interface to manage agents, use extensions, and configure the logging system. Even if you’re not an RL expert, RLib makes it easy to implement the agent-environment interaction loop.

- **Agents** – Choose from a variety of RL agents available in the Agents module. These agents are designed to be versatile and work with any environment. If needed, you can even create your own agents using our documented recipes.

- **Extensions** – Enhance agent observations with domain-specific knowledge by adding a suitable extension from the Extensions module. This module enables seamless agent switching and parameter tuning without extensive reconfiguration.

- **Loggers** – This module allows you to monitor agent-environment interactions. Customize and adapt logging to your specific needs, capturing training metrics, internal agent state, or environment observations. The library includes various loggers for creating plots and output files, simplifying visualization and data processing.

The figure below provides a visual representation of Reinforced-lib and the data-flow between its modules.
Chapter 1. Key components
Our library is built on top of JAX, a high-performance numerical computing library. JAX makes it easy to implement RL algorithms efficiently. It provides powerful transformations, including JIT compilation, automatic differentiation, vectorization, and parallelization. Our library is fully compatible with DeepMind’s JAX ecosystem, granting access to state-of-the-art RL models and helper libraries. JIT compilation significantly accelerates execution and ensures portability across different architectures (CPUs, GPUs, TPUs) without requiring code modifications. JAX offers another benefit through its robust pseudorandom number generator system, employed in our library to guarantee result reproducibility. This critical aspect of scientific research is frequently underestimated but remains highly significant.
Reinforced-lib is designed to work seamlessly on wireless, low-powered devices, where resources are limited. It’s the perfect solution for energy-constrained environments that may struggle with other ML frameworks. You can export your trained models to TensorFlow Lite with ease, reducing runtime overhead and optimizing performance. This means you can deploy RL agents on resource-limited devices efficiently.
Chapter Four

Table of Contents

Explore the power of Reinforced-lib with our easy-to-follow guides and practical examples in the documentation. Unleash the potential of RL for wireless networks and discover exciting possibilities for your projects. Happy reading!

4.1 Guides

- Getting started
- Examples
- Custom agents
- Custom extensions
- Custom loggers

4.2 API Documentation

- API
- Agents
- Extensions
- Loggers
- Utils
- Exceptions

4.2.1 Getting started

Installation

With pip

You can install the latest version of Reinforced-lib from PyPI via:

```
pip install reinforced-lib
```
From source

To have an easy access to the example files, you can clone the source code from our repository, and than install it locally with pip:

```
git clone git@github.com:m-wojnar/reinforced-lib.git
cd reinforced-lib
pip install .
```

In the spirit of making Reinforced-lib a lightweight solution, we included only the necessary dependencies in the base requirements. To fully benefit from Reinforced-lib conveniences, like TF Lite export, install with the “full” suffix:

```
pip install ".[full]"
```

Note: Since we tested the Reinforced-lib on Python 3.9, we recommend using Python 3.9+.

Basic usage

The vital benefit of using Reinforced-lib is a simplification of the agent-environment interaction loop. Thanks to the RLib class, the initialization of the agent and passing the environment state to the agent are significantly simplified. Below, we present the basic training loop with the simplifications provided by Reinforced-lib.

```
import gymnasium as gym
import optax
from chex import Array
from flax import linen as nn

from reinforced_lib import RLib
from reinforced_lib.agents.deep import DQN
from reinforced_lib.exts import Gymnasium

class QNetwork(nn.Module):
    @nn.compact
    def __call__(self, x: Array) -> Array:
        x = nn.Dense(256)(x)
        x = nn.relu(x)
        return nn.Dense(2)(x)

if __name__ == '__main__':
    rl = RLib(
        agent_type=DQN,
        agent_params={
            'q_network': QNetwork(),
            'optimizer': optax.rmsprop(3e-4, decay=0.95, eps=1e-2),
        },
        ext_type=Gymnasium,
        ext_params={'env_id': 'CartPole-v1'}
    )
```

(continues on next page)
After the necessary imports, we create an instance of the RLib class. We provide the chosen agent type and the appropriate extension for the problem. This extension will help the agent to infer necessary information from the environment. Next create a gymnasium environment and define the training loop. Inside the loop, we call the \texttt{sample} method which passes the observations to the agent, updates the agent’s internal state and returns an action proposed by the agent’s policy. We apply the returned action in the environment to get its altered state. We encourage you to see the \textit{API} section for more details about the \texttt{RLib} class.

Note that in the example above, we use the deep reinforcement learning agent. Our library provides a wide range of agents, including both deep and classic reinforcement learning agents. To learn more about the available agents, check out the \textit{Agents} section. You can also create your own agent. To learn more about creating custom agents, check out the \textit{Custom agents} section.

The extensions are also a crucial part of the Reinforced-lib. You can use the built-in extensions listed in the \textit{Extensions} section, but we highly encourage you to create your own extensions. To learn more about extensions check out the \textit{Custom extensions} section.

### Training and inference modes

Reinforced-lib provides two modes of operation: training and inference. The training mode is the default one. It enables the agent to learn from the environment. The inference mode is used to evaluate the agent’s performance or to use the agent in the production environment. To use the inference mode, you have to set the \texttt{is\_training} flag to \texttt{False} in the \texttt{sample} method:

\[
\text{action} = \text{rl}\text{.sample(*env\_state, is\_training=False)}
\]

### Interaction with multiple agents

Reinforced-lib allows you to use multiple agent instances in the same environment. This feature is useful when you want to train multiple agents in parallel or use multiple agents to solve the same problem. To achieve this, you need to initialize the instances of the agents by calling the \texttt{init} method of the \texttt{RLib} class a certain number of times:

\[
\text{rl = RLib(..)}
\]

\[
\text{for _ in range(n\_agents):}
\quad \text{rl\text{.init()}}
\]

4.2. API Documentation
Reproducibility

JAX is focused on reproducibility, and it provides a robust pseudo-random number generator (PRNG) that allows you to control the randomness of the computations. PRNG requires setting the random seed to ensure that the results of the computation are reproducible. Reinforced-lib provides an API for setting the random seed for the JAX library. You can set the seed by providing the `seed` parameter when creating the instance of the agent:

```python
rl = RLib(...)  
rl.init(seed=123)
```

The seed is initially configured as 42 and the `init` method is triggered automatically after the first sampling call. It eliminates the need to manually call the `init` method unless you want to provide custom seed, thus ensuring reproducibility.

**Note:** Remember that the reproducibility of the computations is guaranteed only for the agents from Reinforced-lib. You have to ensure that the environment you use is reproducible as well.

Loggers

The loggers module provides a simple yet powerful API for visualizing and analyzing the running algorithm or watching the training process. You can monitor any observations passed to the agent, the agent’s state, and the basic metrics in real time. If you want to learn more about the loggers module, check out the *Custom loggers* section.

**Basic logging**

Below is the simplest example of using one of the built-in loggers:

```python
rl = RLib(...
           
           logger_types=TensorboardLogger,
           logger_sources='cumulative'
         )
```

In the example above, we use `TensorboardLogger` to print the cumulative reward of the agent. The `logger_sources` parameter specifies the predefined source of the logger. The source is a name of the observation, the agent’s state, or the metric. TensorBoard is a powerful visualization toolkit that allows you to monitor the training process in real time, create interactive visualizations, and save the logs for later analysis. You can use the `TensorboardLogger` along with other loggers built into Reinforced-lib. To learn more about available loggers, check out the *Loggers module* section.

**Warning:** Some loggers perform actions upon completion of the training, such as saving the logs, closing the file, or uploading the logs to the cloud. Therefore, it is important to gracefully close the Reinforced-lib instance to ensure that the logs are saved properly. If you create a Reinforced-lib instance in a function, the destructor will be called automatically when the function ends and you do not have to worry about anything. However, if you create an instance in the main script, you have to close it manually by calling the `finish` method:

```python
rl = RLib(...)  
# ...
rl.finish()
```

or by using the `del` statement:
Advanced logging

You can easily change the logger type, add more sources, and customize the parameters of the logger:

```python
rl = RLib(...)
    # ...
    del rl
```

Note that `terminal` is the observation name, `epsilon` is name of the state of the DQN agent, and `action` is the name of the metric. You can mix sources names as long as it does not lead to inconclusiveness. In the example above, it can be seen that `action` is both the name of the observation and the metric. In this case, you have to write the source name as a tuple containing a name and the type of the source (`str`, `SourceType`) as in the code above.

You can also plug multiple loggers to output the logs to different destinations simultaneously:

```python
rl = RLib(...
    ...
    logger_types=[PlotsLogger,
        logger_sources=['terminal', 'epsilon', ('action', SourceType.METRIC)],
        logger_params={ 'plots_smoothing': 0.9}
    )
```

Custom logging

Note that you are not limited to logging only the predefined sources. You can log any value you want. To do this, you can use the `log` method of the `RLib` class. Remember to define a logger that will be used to log the value. You can do this by providing the only logger type (without sources) or by providing the logger type and the source set to `None`:

```python
rl = RLib(...
    ...
    logger_types=TensorboardLogger
)
rl.log('my_value', 42)
```

It is possible to mix predefined sources with custom ones:

```python
rl = RLib(...
    ...
    logger_types=[TensorboardLogger, PlotsLogger, StdoutLogger],
        logger_sources=('reward', SourceType.METRIC)
    )
rl.log('my_value', 42)
```
Of course, you can also create your own logger. To learn more about creating custom loggers, check out the Custom loggers section.

**Saving experiments**

The RLib class provides an API for saving your experiment in a compressed .lz4 format. You can later reconstruct the experiment state and continue from the exact point where you ended or you can alter some training parameters during the reloading process.

**Full reconstruction**

We can imagine a scenario where we set up the experiment, perform a little training, and then we need to take a break. Therefore, we save the experiment at the latest state that we would later want to carry on from. When we are ready to continue with the training, we load the whole experiment to a new RLib instance.

```python
from reinforced_lib import RLib

# Setting up the experiment
rl = RLib(...)

# Do some training
# ...

# Saving experiment state for later
rl.save("<checkpoint-path>")

# Do some other staff, quit the script if you want.

# Load the saved training
rl = RLib.load("<checkpoint-path>")

# Continue the training
# ...
```

Reinforced-lib can even save the architecture of your agent’s neural network. It is possible thanks to the cloudpickle library allowing to serialize the flax modules. However, if you use your own implementation of agents or extensions, you have to ensure that they are available when you restore the experiment as Reinforced-lib does not save the source code of the custom classes.

**Note:** Remember that the RLib class will not save the state of the environment. You have to save the environment state separately if you want to continue the training from the exact point where you ended.

**Warning:** As of today (2024-02-08), cloudpickle does not support the serialization of the custom modules defined outside of the main definition. It means that if you implement part of your model in a separate class, you will not be able to restore the experiment. We are working on a solution to this problem.

The temporary solution is to define the whole model in one class as follows:

```python
class QNetwork(nn.Module):
    @nn.compact
    def __call__(self, x):
```
Dynamic parameter change

Another feature of the saving mechanism is that it allows us to dynamically change training parameters. Let us recall the above example and modify it a little. We now want to modify on-the-fly the learning rate of the optimizer:

```python
from reinforced_lib import RLib
from reinforced_lib.agents.deep import DQN
from reinforced_lib.exts import Gymnasium

# Setting up the experiment
rl = RLib(
    agent_type=DQN,
    agent_params={
        'q_network': QNetwork(),
        'optimizer': optax.adam(1e-3),
    },
    ext_type=Gymnasium,
    ext_params={'env_id': 'CartPole-v1'}
)

# Do some training
```

(continues on next page)
# Saving experiment state for later
rl.save("<checkpoint-path>")

# Load the saved training with altered parameters
rl = RLib.load(
    "<checkpoint-path>",
    agent_params={
        'q_network': QNetwork(),
        'optimizer': optax.adam(1e-4),
    }
)

# Continue the training with new parameters
# ...

You can change as many parameters as you want. The provided example is constrained only to the agent’s parameter alteration, but you can modify the extension’s parameters in the same way. You can even completely modify the loggers behaviour by providing new ones in logger_types, logger_sources and logger_params.

**Automatic checkpointing**

The RLib class provides an API for automatic checkpointing. You can specify the frequency of saving the experiment state and the path to the directory where the checkpoints will be saved. The checkpointing mechanism is based on the save() method, so you can use the same API for reloading the experiment.

```python
rl = RLib(
    ...
    auto_checkpoint=5,
    auto_checkpoint_path="<checkpoint-dir>"
)

# Do some training
# ...

# Load the saved training
rl = RLib.load("<checkpoint-path>")
```

The auto_checkpoint parameter specifies the frequency of saving the experiment state (in this case every 5th update of the agent). The auto_checkpoint_path parameter specifies the path to the directory where the checkpoints will be saved.
Export to TF Lite

Reinforced-lib offers a convenient API to export the agent to the TensorFlow Lite format, allowing seamless integration with embedded devices or deployment to production environments.

Exporting the agent

To export model you can leverage the `to_tflite` method of the RLib class:

```python
rl.to_tflite(<model-path>)
```

By default, the exported model will include the core functionalities of the agent, namely the `init`, `update`, and `sample` methods. It’s important to note that the `init` method will initialize the basic state of the agent. For deep learning agents, this means the neural network weights will be randomly initialized, while for multi-armed bandit agents, the state will be filled with default values.

Exporting with trained state

If you wish to export the agent with the state of a specific trained agent, you can achieve this by providing the `agent_id` parameter:

```python
rl.to_tflite(<model-path>, agent_id=<agent-id>)
```

By specifying the `agent_id` parameter, the exported model will be initialized with the state of the corresponding agent.

Exporting for inference mode

In some cases, you might only need the agent for inference purposes. To export the agent for inference mode, you need to set the `sample_only` flag to `True` and provide the relevant `agent_id` parameter:

```python
rl.to_tflite(<model-path>, agent_id=<agent-id>, sample_only=True)
```

In this scenario, the exported model will only contain the `init` and `sample` methods of the agent, and the `init` method will return the state of the specified agent.

Requirements

**Note:** To export the agent to the TensorFlow Lite format, the `tensorflow` package is required. To install the package, run the following command:

```bash
pip install tensorflow
```

All built-in agents are adapted to the seamless export to the TensorFlow Lite format. If you want to export a custom agent, you need to implement the `update_observation_space` and `sample_observation_space` methods. Although not mandatory, we strongly encourage their implementation as they allow easy sampling of the parameters of the agent’s methods. To learn more about the agent’s methods, check out the **Custom agents** section.
64-bit floating-point precision

By default, JAX uses 32-bit floating-point precision. However, in some cases, you might want to use 64-bit floating-point precision. The easiest way to achieve this is to set the JAX_ENABLE_X64 environment variable to True:

```bash
export JAX_ENABLE_X64=True
```

Alternatively, you can set the environment variable in your Python script:

```python
import os
os.environ['JAX_ENABLE_X64'] = 'True'
```

Real-world examples

To help you get started and learn how to utilize Reinforced-lib in real-world scenarios, we have prepared a comprehensive set of examples. We strongly encourage you to explore them in the dedicated Examples section.

To access the source code of these examples, simply navigate to the examples directory on our GitHub repository.

4.2.2 Examples

Integration with Gymnasium

Gymnasium, formerly known as OpenAI Gym, is a popular toolkit that provides a standardized interface for reinforcement learning environments. Gymnasium offers a variety of environments, from simple classic control tasks like balancing a pole, which is described below in detail, to complex games like Atari and MuJoCo. It even supports creating custom environments, making it a versatile tool for all things reinforcement learning research.

Reinforced-lib on the other hand provides implementations of various reinforcement learning algorithms. It can seamlessly integrate with Gymnasium by using the environments provided by Gymnasium as the learning context for the algorithms implemented in Reinforced-lib. This integration allows developers to easily train and evaluate their reinforcement learning models using a wide variety of pre-defined scenarios.

Cart Pole example

The Cart Pole environment is a classic control task in which the goal is to balance a pole on a cart. The environment is described by a 4-dimensional state space, which consists of the cart’s position, the cart’s velocity, the pole’s angle, and the pole’s angular velocity. The agent can take one of two actions: push the cart to the left or push the cart to the right. The episode ends when the pole falls below a certain angle or the cart moves outside of the environment’s boundaries. The goal is to keep the pole balanced for as long as possible.

The following example demonstrates how to train a reinforcement learning agent using Reinforced-lib and Gymnasium. The agent uses the Deep Q-Learning (DQN) algorithm to learn how to balance the pole. The DQN algorithm is implemented in Reinforced-lib and the Cart Pole environment is provided by Gymnasium.

We start with the necessary imports:

```python
import gymnasium as gym
import optax
from chex import Array
from flax import linen as nn
```

(continues on next page)
from reinforced_lib import RLib
from reinforced_lib.agents.deep import DQN
from reinforced_lib.exs import Gymnasium
from reinforced_lib.logs import StdoutLogger, TensorboardLogger

We then define the QNetwork approximator as a simple multi-layer perceptron with a ReLU activation function:

```python
class QNetwork(nn.Module):
    @nn.compact
    def __call__(self, x: Array) -> Array:
        x = nn.Dense(256)(x)
        x = nn.relu(x)
        return nn.Dense(2)(x)
```

Next, we step into the `run` function, which is responsible for training the agent. We start by instantiating the Reinforced-lib, specifying the agent as a DQN, the extension as `Gymnasium`, and the loggers as `StdoutLogger` and `TensorboardLogger` to log both to the console and to a TensorBoard file. Note that we specify the environment type in the parameters of the extension to allow for automatic inference of environment properties, such as the state and action space sizes.

```python
def run(num_epochs: int) -> None:
    rl = RLib(
        agent_type=DQN,
        agent_params={
            'q_network': QNetwork(),
            'optimizer': optax.rmsprop(3e-4, decay=0.95, eps=1e-2),
            'discount': 0.95,
            'epsilon_decay': 0.9975
        },
        ext_type=Gymnasium,
        ext_params={'env_id': 'CartPole-v1'},
        logger_types=[StdoutLogger, TensorboardLogger]
    )
```

We then start the training loop where we iterate over the number of epochs and for each epoch we let the agent interact with the environment. We start by resetting the environment and sampling the initial action of the agent. Then we run the agent in the environment by performing the action in the environment and sampling the next action. We continue this loop until the environment reaches a terminal state. We log the length of the epoch as the performance metric and move on to the next epoch.

```python
for epoch in range(num_epochs):
    env = gym.make('CartPole-v1', render_mode='no')
    _, _ = env.reset()
    action = env.action_space.sample()
    terminal = False
    epoch_len = 0

    while not terminal:
        env_state = env.step(action.item())
        action = rl.sample(*env_state)
        terminal = env_state[2] or env_state[3]
        epoch_len += 1
```

(continues on next page)
We start the training by calling the `run` function with the number of epochs as an argument:

```python
if __name__ == '__main__':
    run(num_epochs=300)
```

The complete, runnable code can be copy pasted from the following snippet:

```python
import gymnasium as gym
import optax
from chex import Array
from flax import linen as nn

from reinforced_lib import RLib
from reinforced_lib.agents.deep import DQN
from reinforced_lib.exts import Gymnasium
from reinforced_lib.logs import StdoutLogger, TensorboardLogger

class QNetwork(nn.Module):
    @nn.compact
    def __call__(self, x: Array) -> Array:
        x = nn.Dense(256)(x)
        x = nn.relu(x)
        return nn.Dense(2)(x)

def run(num_epochs: int) -> None:
    rl = RLib(
        agent_type=DQN,
        agent_params={
            'q_network': QNetwork(),
            'optimizer': optax.rmsprop(3e-4, decay=0.95, eps=1e-2),
            'discount': 0.95,
            'epsilon_decay': 0.9975
        },
        ext_type=Gymnasium,
        ext_params={'env_id': 'CartPole-v1'},
        logger_types=[StdoutLogger, TensorboardLogger]
    )

    for epoch in range(num_epochs):
        env = gym.make('CartPole-v1', render_mode='no')

        _, _ = env.reset()
        action = env.action_space.sample()
        terminal = False
        epoch_len = 0

        while not terminal:
            env_state = env.step(action.item())

(continues on next page)
action = rl.sample(*env_state)
terminal = env_state[2] or env_state[3]
epoch_len += 1

rl.log('epoch_len', epoch_len)

if __name__ == '__main__':
    run(num_epochs=300)

Other examples

We provide a few more examples of Reinforced-lib and Gymnasium integration in the examples directory of the Reinforced-lib repository. The examples include the training of the DQN agent in the Cart Pole environment (described above) and the training of the DDPG agent in the Pendulum environment. The examples are fully runnable and can be used as a starting point for your own reinforcement learning experiments with Reinforced-lib and Gymnasium.

Connection with ns-3

We will demonstrate the cooperation of Reinforced-lib with an external WiFi simulation software based on an example of an ML-controlled rate adaptation (RA) manager. To simulate the WiFi environment, we will use the popular, research oriented network simulator – ns-3. To learn more about the simulator, we encourage to visit the official website or read the ns-3 tutorial.

Environment setup

To perform experiments with Python-based Reinforced-lib and C++-based ns-3, you need to setup an environment which consists of the following:

- favourite C++ compiler (we assume that you already have one in your dev stack),
- ns-3 (connection tested on the ns-3.37 version),
- ns3-ai (GitHub repository).

Since the ns-3 requires the compilation, we will install all the required modules, transfer ns-3 files required for the communication with Reinforced-lib, and compile everything once at the very end.

Installing ns-3

There are a few ways to install ns-3, all described in the ns-3 wiki, but we recommend to install ns-3 by cloning the git dev repository:

```
git clone https://gitlab.com/nsnam/ns-3-dev.git
```

We recommend setting the simulator to the 3.37 version, since we do not guarantee the compatibility with other versions. To set the ns-3 to the 3.37:

```
cd ns-3-dev  # this directory will be referenced as YOUR_NS3_PATH since now on
git reset --hard 4407a9528eac81476546a50597cc6e016a428f43
```
Installing ns3-ai

The ns3-ai module interconnects ns-3 and Reinforced-lib (or any other python-written software) by transferring data through the shared memory pool. The memory is accessed by both sides thus making the connection. You can read more about the ns3-ai on the ns3-ai official repository.

**Warning:** Unfortunately, ns3-ai (as of 18.07.2023) is not compatible with the ns-3.36 or later. We have forked and modified the official ns3-ai repository to make it compatible with the 3.37 version. To install the compatible, forked version run the following commands

```bash
cd $YOUR_NS3_PATH/contrib/
git clone --single-branch --branch ml4wifi https://github.com/m-wojnar/ns3-ai.git
pip install "$YOUR_NS3_PATH/contrib/ns3-ai/py_interface"
```

Transferring ns3 files

In `$REINFORCED_LIB/examples/ns-3-ra/` there are two directories. The scratch contains an example RA scenario, which will be described in the next section. The contrib directory contains a rlib-wifi-manager module with the specification of a custom rate adaptation manager that communicates with python with the use of ns3-ai. You need to transfer both of these directories in the appropriate locations by running the following commands:

```bash
cp $REINFORCED_LIB/examples/ns-3-ra/scratch/* $YOUR_NS3_PATH/scratch/
cp -r $REINFORCED_LIB/examples/ns-3-ra/contrib/rlib-wifi-manager $YOUR_NS3_PATH/contrib/
```

**Note:** To learn more about adding contrib modules to ns-3, visit the ns-3 manual.

Compiling ns3

To have the simulator working and fully integrated with the Reinforced-lib, we need to compile it. We do this from the $YOUR_NS3_PATH in two steps, by first configuring the compilation and than by building ns-3:

```bash
cd $YOUR_NS3_PATH
./ns3 configure --build-profile=optimized --enable-examples --enable-tests
./ns3 build
```

Once you have built ns-3, you can test the ns-3 and Reinforced-lib integration by executing the script that runs an example rate adaptation scenario controlled by the UCB agent.

```bash
cd $REINFORCED_LIB
./test/test_ns3_integration.sh $YOUR_NS3_PATH
```

On success, in your home directory, there should be a `rlib-ns3-integration-test.csv` file generated filled with some data.
## Simulation scenario

### ns-3 (C++) part

In the `scratch` directory we supply an example scenario `rlib-sim.cc` to test the rate adaptation manager in the 802.11ax environment. The scenario is highly customizable but the key points are that there is one access point (AP) and a variable number (`--nWifi`) of stations (STA); there is an uplink, saturated communication (from stations to AP) and the AP is in line of sight with all the stations; all the stations are at the point of $(0,0)$ m and the AP can either be at $(0,0)$ m as well or in some distance (`--initialPosition`) from the stations. The AP can also be moving with a constant velocity (`--velocity`) to simulate dynamic scenarios. Other assumptions from the simulation are the A-MPDU frame aggregation, 5 Ghz frequency band, and single spatial stream.

By typing `$YOUR_NS3_PATH/build/scratch/ns3.37-ra-sim-optimized --help` you can go over the simulation parameters and learn what is the function of each.

```bash
./build/scratch/ns3.37-ra-sim-optimized --help
[Program Options] [General Arguments]

**Program Options:**

- `--area`: Size of the square in which stations are wandering (m) [RWPM, mobility type] [40]
- `--channelWidth`: Channel width (MHz) [20]
- `--csvPath`: Save an output file in the CSV format
- `--dataRate`: Aggregate traffic generators data rate (Mb/s) [125]
- `--deltaPower`: Power change (dBm) [0]
- `--initialPosition`: Initial position of the AP on X axis (m) [Distance mobility, type] [0]
- `--intervalPower`: Interval between power change (s) [4]
- `--logEvery`: Time interval between successive measurements (s) [1]
- `--lossModel`: Propagation loss model to use [LogDistance, Nakagami, [LogDistance]]
- `--minGI`: Shortest guard interval (ns) [3200]
- `--mobilityModel`: Mobility model [Distance, RWPM, [Distance]]
- `--nodeSpeed`: Maximum station speed (m/s) [RWPM mobility type] [1.4]
- `--nodePause`: Maximum time station waits in newly selected position (s) [RWPM, mobility type] [20]
- `--nWifi`: Number of transmitting stations [1]
- `--pcapPath`: Save a PCAP file from the AP
- `--simulationTime`: Duration of the simulation excluding warmup stage (s) [20]
- `--velocity`: Velocity of the AP on X axis (m/s) [Distance mobility type] [0]
- `--warmupTime`: Duration of the warmup stage (s) [2]
- `--wifiManager`: Rate adaptation manager [ns3::RLibWifiManager]
- `--wifiManagerName`: Name of the Wi-Fi manager in CSV

**General Arguments:**

- `--PrintGlobals`: Print the list of globals.
- `--PrintGroups`: Print the list of groups.
- `--PrintGroup=[group]`: Print all TypeIds of group.
- `--PrintTypeIds`: Print all TypeIds.
- `--PrintAttributes=[typeid]`: Print all attributes of typeid.
- `--PrintVersion`: Print the ns-3 version.
- `--PrintHelp`: Print this help message.
Reinforced-lib (Python) part

The provided rate adaptation manager is implemented in the file `$REINFORCED_LIB/examples/ns-3-ra/main.py`. Here we specify the communication with the ns-3 simulator by defining the environment’s observation space and the action space, we create the RLib agent, we provide the agent-environment interaction loop which reacts to the incoming (aggregated) frames by responding with an appropriate MCS, and cleans up the environment when the simulation is done. Below we include and explain the essential fragments from the `main.py` script.

```python
from ext import IEEE_802_11_ax_RA
from particle_filter import ParticleFilter
from py_interface import *
from reinforced_lib import RLib
from reinforced_lib.agents.mab import *
```

We import the RA extension, agents and the RLib module. Line 6 is responsible for importing the structures from the ns3-ai library.

```python
class Env(Structure):
    _pack_ = 1
    _fields_ = [
        ('power', c_double),
        ('time', c_double),
        ('cw', c_uint32),
        ('n_failed', c_uint32),
        ('n_successful', c_uint32),
        ('n_wifi', c_uint32),
        ('station_id', c_uint32),
        ('type', c_uint8)
    ]
```

```python
class Act(Structure):
    _pack_ = 1
    _fields_ = [
        ('station_id', c_uint32),
        ('mcs', c_uint8)
    ]
```

Next we define the ns3-ai structures that describe the environment space and action space accordingly. The structures must strictly reflect the ones defined in the header file `contrib/rlib-wifi-manager/model/rlib-wifi-manager.h` because it is the interface of the shared memory data bridge between python and C++. You can learn more about the data exchange model here.

```python
rl = RLib(
    agent_type=agent_type,
    agent_params=agent_params,
    ext_type=IEEE_802_11_ax_RA
)
exp = Experiment(mempool_key, memory_size, 'ra-sim', ns3_path)
var = Ns3AIRL(memblock_key, Env, Act)
```

In line 73, we create an instance of RLib by supplying the appropriate, parametrized agent and the 802.11ax environment.
extension. We define the ns3-ai experiment in line 79 by setting the memory key, the memory size, the name of the ns-3 scenario, and the path to the ns3 root directory. In line 80, we create a handler to the shared memory interface by providing an arbitrary key and the previously defined environment and action structures.

```python
try:
    ns3_process = exp.run(ns3_args, show_output=True)

    while not var.isFinish():
        with var as data:
            if data is None:
                break

            if data.env.type == 0:
                data.act.station_id = rl.init(seed)

            elif data.env.type == 1:
                observation = {
                    'time': data.env.time,
                    'n_successful': data.env.n_successful,
                    'n_failed': data.env.n_failed,
                    'n_wifi': data.env.n_wifi,
                    'power': data.env.power,
                    'cw': data.env.cw
                }

                data.act.station_id = data.env.station_id
                data.act.mcs = rl.sample(agent_id=data.env.station_id, **observation)

            ns3_process.wait()

finally:
    del exp
```

The final step to make the example work is to define the agent-environment interaction loop. We loop while the ns3 simulation is running (line 85) and if there is any data to be read (line 86). We differentiate the environment observation by a type attribute which indicates whether it is an initialization frame or not. On initialization (line 90), we have to initialize our RL agent with some seed. In the opposite case we translate the observation to a dictionary (lines 94-102) and override the action structure with the received station ID (line 104) and the appropriate MCS selected by the RL agent (line 105). The last thing is to clean up the shared memory environment when the simulation is finished (lines 107 and 107).

### Example experiments

We supply the `$REINFORCED_LIB/examples/ns-3-ra/main.py` script with the CLI so that you can test the rate adaptation manager in different scenarios. We reflect all the command line arguments listed in `ns3 scenario scratch/ra-sim.cc` with four additional arguments:

- `--agent` – the type of RL agent responsible for the RA, a required argument,
- `--mempoolKey` – shared memory pool key, which is an arbitrary integer, greater than 1000, default is 1234.
- `--ns3Path` – path to the ns3 root directory, a required argument,

You can try running the following commands to test the Reinforced-lib rate adaptation manager in different example scenarios:
a. Static scenario with 1 AP and 1 STA both positioned in the same place, RA handled by the UCB agent

```python
python $REINFORCED_LIB/examples/ns-3-ra/main.py --agent="UCB" --ns3Path="$YOUR_NS3_PATH"
```

b. Static scenario with 1 AP and 1 STA both positioned in the same place, RA handled by the UCB agent. Output saved to the $HOME/ra-results.csv file and .pcap saved to the $HOME/ra-experiment-0-0.pcap.

```python
python $REINFORCED_LIB/examples/ns-3-ra/main.py --agent="UCB" --ns3Path="$YOUR_NS3_PATH" --csvPath="$HOME/ra-results.csv" --pcapPath="$HOME/ra-experiment"
```

c. Static scenario with 1 AP and 16 stations at a 10 m distance, RA handled by the ThompsonSampling agent.

```python
python $REINFORCED_LIB/examples/ns-3-ra/main.py --agent="ThompsonSampling" --ns3_path="$YOUR_NS3_PATH" --nWifi=16 --initialPosition=10
```

d. Dynamic scenario with 1 AP and 1 STA starting at 0 m and moving away from AP with a velocity of 1 m/s, RA handled by the ParticleFilter agent.

```python
python $REINFORCED_LIB/examples/ns-3-ra/main.py --agent="ParticleFilter" --ns3Path="$YOUR_NS3_PATH" --velocity=1
```

**Source code of the example**

The complete, runnable code can be found in the examples/ns-3-ra directory of the Reinforced-lib repository. The example provides many useful scripts for reproducing our experiments and can be used as a starting point for your own reinforcement learning experiments with Reinforced-lib and ns-3. We also encourage you to see another example - implementation of the centralized contention window optimization with DRL (CCOD) in the examples/ns-3-ccod directory which presents a deep reinforcement learning scenario with Reinforced-lib and ns-3.

### 4.2.3 Custom agents

Although our library provides a palette of already implemented agents, you might want to add a personalised one to the collection. This guide is to help you with this task.

**Implementing new agents**

To fully benefit from Reinforced-lib features, including JAX JIT optimization, your agent should inherit from the abstract class `BaseAgent`. We present adding a custom agent on an example of a simple epsilon-greedy agent:

```python
class EGreedy(BaseAgent)
```

Firstly, we need to define the state of our agent, which in our case will hold

- quality values of each arm \(Q\),
- number of each arms’ tries \(N\),
and will inherit from `AgentState`:

```python
@dataclass
class EGreedyState(AgentState):

    Q: Array
    N: Array
```

The `BaseAgent` interface breaks the agent’s behaviour into three methods:

- `init(PRNGKey, ...) -> AgentState` - initializes the agent’s state,
- `update(AgentState, PRNGKey, ...) -> AgentState` - updates the agent’s state after performing some action and receiving a reward,
- `sample(AgentState, PRNGKey, ...) -> Action` - samples new action according to the agent’s and environment’s state.

We define the Epsilon-greedy agent, which will have 3 static methods:

```python
# This method initializes the agent with 'n_arms' arms
@staticmethod
def init(
    key: PRNGKey,
    n_arms: int
) -> EGreedyState:

    return EGreedyState(
        # The initial Q values are set as zeros, due to the lack of prior knowledge
        Q=jnp.zeros(n_arms),
        # The numbers of tries are set as ones, to avoid null division in Q value update
        N=jnp.ones(n_arms, dtype=int)
    )

# This method updates the agents state
@staticmethod
def update(
    state: EGreedyState,
    key: PRNGKey,
    action: int,
    reward: Scalar,
) -> EGreedyState:

    return EGreedyState(
        # Q value update
        Q=state.Q.at[action].add((reward - state.Q[action]) / state.N[action]),
        # Incrementing the number of tries on appropriate arm
        N=state.N.at[action].add(1)
    )

# This method samples new action according to the agents state (experience)
@staticmethod
```

(continues on next page)
def sample(
    state: EGreedyState,
    key: PRNGKey,
    e: Scalar
) -> int:

    # Split PRNGkey to use it twice
    epsilon_key, choice_key = jax.random.split(key)

    # We further want to jax.jit this function, so basic 'if' is not allowed here
    return jax.lax.cond(
        # The agent experiments with probability e
        jax.random.uniform(epsilon_key) < e,
        # On exploration, agent chooses a random arm
        lambda: jax.random.choice(choice_key, state.Q.size),
        # On exploitation, agent chooses the best known arm
        lambda: jnp.argmax(state.Q)
    )

Having defined these static methods, we can implement the class constructor:

def __init__(
    self,
    n_arms: int,
    e: Scalar
) -> None:

    # Make sure that epsilon has correct value
    assert 0 <= e <= 1

    # We specify the features of our agent
    self.n_arms = n_arms

    # Here we can use the jax.jit() functionality with the previously
    # defined behaviour functions, to make the agent super fast.
    # Note that we use partial() to specify the parameters that are
    # constant during the agent's lifetime to avoid passing them
    # every time the function is called.
    self.init = jax.jit(partial(self.init, n_arms=self.n_arms))
    self.update = jax.jit(self.update)
    self.sample = jax.jit(partial(self.sample, e=e))

Now we specify the initialization arguments of our agent (i.e., the parameters that are required by the agent’s constructor). This is done by implementing the static method parameter_space() which returns a dictionary in the format of a Gymnasium (former OpenAI Gym) space. It is not required to implement this method, but it is a good practice to do so. This enables the library to automatically provide initialization arguments specified by extensions.

# Parameters required by the agent constructor in Gymnasium format.
# Type of returned value is required to be gym.spaces.Dict.
Specifying the action space of the agent is accomplished by implementing the action_space property. While not mandatory, adhering to this practice is recommended as it allows users to conveniently inspect the agent’s action space through the action_space method of the RLib class.

```python
# Action returned by the agent in Gymnasium format.
@property
def action_space(self) -> gym.spaces.Space:
    return gym.spaces.Discrete(self.n_arms)
```

Finally, we define the observation spaces for our agent by implementing the properties called update_observation_space and sample_observation_space. Although not mandatory, we strongly encourage their implementation as it allows the library to deduce absent values from raw observations and functions defined in the extensions. Moreover, having these properties implemented facilitates a seamless export of the agent to the TensorFlow Lite format, where the library can automatically generate an example set of parameters during the export procedure.

```python
# Parameters required by the 'update' method in Gymnasium format.
@property
def update_observation_space(self) -> gym.spaces.Dict:
    return gym.spaces.Dict({'
        'action': gym.spaces.Discrete(self.n_arms),
        'reward': gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float)
    })

# Parameters required by the 'sample' method in Gymnasium format.
@property
def sample_observation_space(self) -> gym.spaces.Dict:
    return gym.spaces.Dict({})
```

Now we have a ready to operate epsilon-greedy agent!

**Template agent**

Here is all of the above code in one piece. You can copy-paste it and use as an inspiration to create your own agent.

```python
from functools import partial
import gymnasium as gym
import jax
import jax.numpy as jnp
from chex import dataclass, Array, Scalar, PRNGKey
from reinforced_lib.agents import BaseAgent, AgentState
```

(continues on next page)
@dataclass
class EGreedyState(AgentState):
    Q: Array
    N: Array

class EGreedy(BaseAgent):
    def __init__(self, n_arms: int, e: Scalar) -> None:
        assert 0 <= e <= 1

        self.n_arms = n_arms

        self.init = jax.jit(partial(self.init, n_arms=n_arms))
        self.update = jax.jit(self.update)
        self.sample = jax.jit(partial(self.sample, e=e))

    @staticmethod
def parameter_space() -> gym.spaces.Dict:
        return gym.spaces.Dict({
            'n_arms': gym.spaces.Box(1, jnp.inf, (1,), int),
            'e': gym.spaces.Box(0.0, 1.0, (1,), float)
        })

    @property
def update_observation_space(self) -> gym.spaces.Dict:
        return gym.spaces.Dict({
            'action': gym.spaces.Discrete(self.n_arms),
            'reward': gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float)
        })

    @property
def sample_observation_space(self) -> gym.spaces.Dict:
        return gym.spaces.Dict({})

    @property
def action_space(self) -> gym.spaces.Space:
        return gym.spaces.Discrete(self.n_arms)

    @staticmethod
def init(key: PRNGKey, n_arms: int) -> EGreedyState:
        return EGreedyState(
            Q=jnp.zeros(n_arms),
            N=jnp.ones(n_arms, dtype=int)
        )
Deep learning agents

Although the above example is a simple one, it is not hard to extend it to deep reinforcement learning (DRL) agents. This can be achieved by leveraging the JAX ecosystem, along with the flax library, which provides a convenient way to define neural networks, and optax, which provides a set of optimizers. Below, we provide excerpts of the code for the deep Q-learning agent.

The state of the DRL agent often contains parameters and state of the neural network as well as an experience replay buffer:

```python
@dataclass
class DQNState(AgentState):
    params: dict
    state: dict
    opt_state: optax.OptState
    replay_buffer: ReplayBuffer
    prev_env_state: Array
    epsilon: Scalar
```

The agent’s constructor allows you to specify parameters for the neural network architecture and optimizer, enabling users to have full control over their choice and enhancing the agent’s flexibility:
```python
def __init__(self, q_network: nn.Module, optimizer: optax.GradientTransformation = None, ...
) -> None:
    if optimizer is None:
        optimizer = optax.adam(1e-3)

    self.init = jax.jit(partial(self.init, q_network=q_network, optimizer=optimizer, ...->))

By implementing the constructor in this manner, users gain the flexibility to define their own architecture as follows:

class QNetwork(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Dense(64)(x)
        x = nn.relu(x)
        x = nn.Dense(64)(x)
        x = nn.relu(x)
        return nn.Dense(2)(x)

rl = RLib(
    agent_type=DQN,
    agent_params=
        {'q_network': QNetwork(),
         'optimizer': optax.rmsprop(3e-4, decay=0.95, eps=1e-2)
        },
    ...
)
```

**Note:** In some cases, it is necessary to use a PRNG key in the definition of a neural network to allow the stochastic behavior of the model. The Flax library provides a `make_rng(stream_name)` method that can be used to generate a PRNG key from a given stream. The DRL algorithms implemented in Reinforced-lib offer a stream called `rlib` by default, so you can use it in your custom model as follows: `key = self.make_rng('rlib')`.

During the development of a DRL agent, our library offers a set of utility functions for your convenience. Among these functions is `gradient_step`, designed to streamline parameter updates for the agent using JAX and optax. In the following example code snippet, we showcase the implementation of a step function responsible for performing a single step, taking into account the network, optimizer, and the implemented loss function:

```python
from reinforced_lib.utils.jax_utils import gradient_step

step_fn=partial(
    gradient_step,
    optimizer=optimizer,
    loss_fn=partial(self.loss_fn, q_network=q_network, ...)
)
```
There are also other utility functions that can make it easier to implement DRL agents with flax. These are the init and forward methods which are used to initialize the network and to perform a forward pass through the network. You can find more information about these functions in the documentation.

Our Python library also includes a pre-built experience replay buffer, which is commonly utilized in DRL agents. The following code provides an illustrative example of how to use this utility:

```python
from reinforced_lib.utils.experience_replay import experience_replay, ExperienceReplay,
                                     ReplayBuffer

er = experience_replay(  
    experience_replay_buffer_size,  
    experience_replay_batch_size,  
    obs_space_shape,  
    act_space_shape
)

...  

replay_buffer = er.init()  
...  

replay_buffer = er.append(replay_buffer, prev_env_state, action, reward, terminal, env_  
                                     state)  
perform_update = er.is_ready(replay_buffer)

for _ in range(experience_replay_steps):  
    batch = er.sample(replay_buffer, key)  
...  
```

Developing a DRL agent may pose challenges, so we strongly recommend thoroughly studying an example code of one of our DRL agents prior to building your custom agent.

**Summary**

To sum everything up one more time:

1. All agents inherit from the BaseAgent class.
2. The agent’s state is defined as a dataclass that inherits from the AgentState class.
3. The agent’s behavior is determined by implementing the static methods init, update, and sample.
4. Utilizing jax.jit can significantly increase the agent’s performance.
5. Although not mandatory, it is highly recommended to implement the parameter_space, update_observation_space, and sample_observation_space properties.
6. Implementing a custom DRL agent is possible using the JAX ecosystem and utility functions provided by the library.
4.2.4 Custom extensions

The extensions is a unique functionality that allows a library to infer missing observations that are not originally supported by the environment. You can either choose one of our built-in extensions or implement your own with the help of this short guide.

Key concepts of extensions

There are three main benefits of using extensions:

1. Automatic initialization of agents - an extension can provide default arguments that can be used to initialize an agent. For example, if we would like to train the deep Q-learning agent on a cart-pole environment without using any extension, we would probably do it in the following way:

```python
rl = RLib(
    agent_type=DQN,
    agent_params={
        'q_network': q_network,
        'obs_space_shape': (4,),
        'act_space_size': 2
    },
    no_ext_mode=True
)
```

On the other hand, if we decide to use the Gymnasium extension, some of the parameters can be automatically provided by the extension:

```python
rl = RLib(
    agent_type=DQN,
    agent_params={'q_network': q_network}
    ext_type=Gymnasium,
    ext_params={'env_id': 'CartPole-v1'},
)
```

We can also overwrite all or only part of the default values provided by the extension:

```python
rl = RLib(
    agent_type=DQN,
    agent_params={
        'q_network': q_network,
        'act_space_size': 3
    },
    ext_type=Gymnasium,
    ext_params={'env_id': 'CartPole-v1'},
)
```

2. Simplification of parameter passing - extensions allow automatic matching observations returned by the environment to the appropriate methods of the agent. The code snippet below shows the agent and environment interaction loop without using any extension:

```python
while not terminal:
    env_state, reward, terminal, truncated, info = env.step(action.item())
    action = rl.sample(
```

(continues on next page)
The following code is equivalent to the above but makes use of the properly defined Gymnasium extension:

```python
while not env_state[2]:
    env_state = env.step(action.item())
    action = rl.sample(*env_state)
```

3. Filling missing parameters - some parameters required by the agent can be filled with known values or calculated based on a set of basic observations. For example, a sample method of the Thompson sampling agent requires a context vector. As it is a domain specific knowledge, these values can be found in the appropriate extension. Below is a sample code that could be used to sample the next action in the IEEE 802.11 ax rate adaptation problem without using any extension:

```python
rl = RLib(
    agent_type=ThompsonSampling,
    agent_params={'n_arms': 12},
    no_ext_mode=True
)

observations = {
    'delta_time': 0.18232,
    'n_successful': 10,
    'n_failed': 0,
    'context': jnp.array(
        [7.3, 14.6, 21.9, 29.3, 43.9, 58.5,
         65.8, 73.1, 87.8, 97.5, 109.7, 121.9]
    )
}
action = rl.sample(**observations)
```

If we use the IEEE 802.11ax RA extension, part of these parameters can be provided by the extension:

```python
rl = RLib(
    agent_type=ThompsonSampling,
    ext_type=IEEE_802_11_ax_RA
)

observations = {
    'delta_time': 0.18232,
    'n_successful': 10,
    'n_failed': 0
}
action = rl.sample(**observations)
```
We can also overwrite the values provided by the extension:

```python
observations = {
    'delta_time': 0.18232,
    'n_successful': 10,
    'n_failed': 0,
    'context': jnp.array([1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.])
}
action = rl.sample(**observations)
```

You can define default values as initialization arguments for agents through parameter functions. Additionally, default values or functions to calculate missing observations can be defined using observation functions. To designate these functions correctly, they are decorated with the `@observation` and `@parameter` decorators respectively. A more detailed description of this decorator can be found in the section below.

### Implementing new extensions

To create your own extension, you should inherit from the `abstract class` `BaseExt`. We present adding a custom extension using an example of the extension used in the IEEE 802.11ax rate adaptation problem.

```python
class IEEE_802_11_ax_RA(BaseExt)
```

First, we must specify the observation space of the extension. It is a basic set of environment observations that can be used by the extension to compute missing values. Note that a complete set of all parameters is not necessarily required to use the extension - if an agent does not require a given parameter and it is not used to compute missing values, the extension will ignore it. In the case of the IEEE 802.11ax environment, the observation space can look like this:

```python
observation_space = gym.spaces.Dict({
    'time': gym.spaces.Box(0.0, jnp.inf, (1,), float),
    'n_successful': gym.spaces.Box(0, jnp.inf, (1,), int),
    'n_failed': gym.spaces.Box(0, jnp.inf, (1,), int),
    'n_wifi': gym.spaces.Box(1, jnp.inf, (1,), int),
    'power': gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float),
    'cw': gym.spaces.Discrete(32767)
})
```

Next, we define the `parameter function` that will provide the default power value for agents that require this parameter as a constructor argument. We can do this by creating an appropriate method and decorating it with the `@parameter` decorator. The `parameter functions` are methods of the extension and cannot take any additional arguments:

```python
@parameter()
def default_power(self):
    return 16.0206
```

We can also specify the type of the returned value in Gymnasium (former OpenAI Gym) format. It will help the library to check if a given value type is compatible with the argument required by the agent:

```python
@parameter(parameter_type=gym.spaces.Box(-jnp.inf, jnp.inf, (1,)))
def default_power(self) -> float:
    return 16.0206
```

Note that the name of the function must match the name of the argument required by the agent. If there already exists a function with that name, we can name the function differently and explicitly define the argument name in the decorator:
We define the observation functions by analogy to parameter functions. The differences are that we use the @observation decorator and that the implemented methods can take additional parameters. Below is an example observation function that provides a reward calculated as an approximated throughput in the IEEE 802.11ax environment:

```python
@observation()
def reward(self, mcs, n_successful, n_failed, *args, **kwargs):
    if n_successful + n_failed > 0:
        return self._wifi_modes_rates[mcs] * n_successful / (n_successful + n_failed)
    else:
        return 0.0
```

Note that the observation function can take parameters that are specified in the observation space. BaseExt will automatically pass the given observation to the function to allow dynamic computation of the returned value. What is important, observation methods must take *args and **kwargs as the last parameters (this is required by the internal behavior of the setup_transformations function). As previously, the name of the function should match the name of the filled parameter, but we can specify the parameter name and returned type in the decorator:

```python
@observation(observation_name='reward', observation_type=gym.spaces.Box(-jnp.inf, jnp.inf, (1,)))
def custom_reward(self, mcs: int, n_successful: int, n_failed: int, *args, **kwargs) -> float:
    if n_successful + n_failed > 0:
        return self._wifi_modes_rates[mcs] * n_successful / (n_successful + n_failed)
    else:
        return 0.0
```

Template extension

To simplify the process of creating new extensions, we provide an example extension that can be used as a starting point for creating your own extensions. The IEEE 802.11ax rate adaptation extension can be found here:

```python
import gymnasium as gym
import jax.numpy as jnp
from chex import Array, Scalar
from reinforced_lib.exts import BaseExt, observation, parameter

class IEEE_802_11_ax_RA(BaseExt):
    def __init__(self) -> None:
        super().__init__()
        self.last_time = 0.0

    observation_space = gym.spaces.Dict(
        {'time': gym.spaces.Box(0.0, jnp.inf, (1,), float),
         'n_successful': gym.spaces.Box(0, jnp.inf, (1,), int),
    
    # More code here...
```

(continues on next page)
'n_failed': gym.spaces.Box(0, jnp.inf, (1,), int),
'n_wifi': gym.spaces.Box(1, jnp.inf, (1,), int),
'power': gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float),
'cw': gym.spaces.Discrete(32767)
})

_wifi_modes_rates = jnp.array([ 7.3, 14.6, 21.9, 29.3, 43.9, 58.5, 65.8, 73.1, 87.8, 97.5, 109.7, 121.9 ], dtype=float)

@observation(observation_type=gym.spaces.Box(0.0, jnp.inf, (len(_wifi_modes_rates),), float))
def rates(self, *args, **kwargs) -> Array:
    return self._wifi_modes_rates

@observation(observation_type=gym.spaces.Box(-jnp.inf, jnp.inf, (len(_wifi_modes_rates),), float))
def context(self, *args, **kwargs) -> Array:
    return self.rates()

def reward(self, action: int, n_successful: int, n_failed: int, *args, **kwargs) -> float:
    if n_successful + n_failed > 0:
        return self._wifi_modes_rates[action] * n_successful / (n_successful + n_failed)
    else:
        return 0.0

@observation(observation_type=gym.spaces.Box(0.0, jnp.inf, (1,), float))
def delta_time(self, time: Scalar, *args, **kwargs) -> float:
    delta_time = time - self.last_time
    self.last_time = time
    return delta_time

@observation(observation_type=gym.spaces.Box(-jnp.inf, jnp.inf, (6,), float))
def env_state(self, time: Scalar, n_successful: int, n_failed: int, n_wifi: int, power: Scalar, cw: int, *args, **kwargs) -> Array:
    return jnp.array([self.delta_time(time), n_successful, n_failed, n_wifi, power, cw], dtype=float)

@parameter(parameter_type=gym.spaces.MultiBinary(1))
def terminal(self, *args, **kwargs) -> bool:
    return False

def n_mcs(self) -> int:
    return len(self._wifi_modes_rates)
def n_arms(self) -> int:
    return self.n_mcs()

@parameter(parameter_type=gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float))
def default_power(self) -> Scalar:
    return 16.0206

@parameter(parameter_type=gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float))
def min_reward(self) -> Scalar:
    return 0

@parameter(parameter_type=gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float))
def max_reward(self) -> int:
    return self._wifi_modes_rates.max()

@parameter(parameter_type=gym.spaces.Sequence(gym.spaces.Box(1, jnp.inf, (1,), int)))
def obs_space_shape(self) -> tuple:
    return tuple((6,))

@parameter(parameter_type=gym.spaces.Sequence(gym.spaces.Box(1, jnp.inf, (1,), int)))
def act_space_shape(self) -> tuple:
    return tuple((1,))

@parameter(parameter_type=gym.spaces.Box(1, jnp.inf, (1,), int))
def act_space_size(self) -> int:
    return 12

@parameter(parameter_type=gym.spaces.Sequence(gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float)))
def min_action(self) -> tuple:
    return @

@parameter(parameter_type=gym.spaces.Sequence(gym.spaces.Box(-jnp.inf, jnp.inf, (1,), float)))
def max_action(self) -> tuple:
    return 11

---

Rules and limitations

Extensions are powerful mechanisms that make Reinforced-lib easy to use. The BaseExt methods can handle complex and nested observation spaces, such as these example ones. However, there are some rules and limitations that programmers and users must consider:

- arguments and parameters provided by the user have higher priority than the default or calculated by the extension,
- *parameter functions* cannot take any arguments (except `self`),
- you cannot use an extension with a given agent if the agent requires a parameter that is not listed in the extensions observation space or cannot be provided by an *observation function* - you have to add an observation to the observation space, implement the appropriate *observation function* or use the agent without any extension,
- missing parameter filling is supported only if the type of the extension observation space and the type of agent space can be matched - that means they both must be:
– a dict type - gym.spaces.Dict,

• missing parameter filling is not supported if spaces inherit from gym.spaces.Tuple - values would have to be matched based on the type and this can lead to ambiguities if there are multiple parameters with the same type,

• if spaces do not inherit from gym.spaces.Dict, missing values are matched based on the type of the value, not the name, so the first function that type matches the agent space is chosen,

• if an observation function requires some parameter and it is not provided by a named argument, BaseExt will select the first (possibly nested) positional argument and pass it to the function, but if there are no positional arguments, the library will raise an exception.

How do extensions work?

The main axis of this module is the abstract class BaseExt, which provides the core functionality of extensions. It implements important methods, such as get_agent_params, transform, and setup_transformations. The class internally makes use of these methods to provide a simple and powerful API of Reinforced-lib. You can read more about the BaseExt class here or check out the source code.

4.2.5 Custom loggers

Loggers serve as valuable tools for visualizing and analyzing algorithms in real-time during the training process. They enable users to monitor observations passed to the agent, the agent’s state, and basic metrics.

Key concepts of loggers

A basic concept of loggers is the “source”. Sources are objects that can be logged. There are three types of sources:

• observation - value passed to the agent, such as the current state of the environment,

• state - value that is stored in the agent, for example, the array of Q-values,

• metric - value that is calculated during the training process, such as the cumulative reward.

Sources are represented by a string or a tuple (str, SourceType), where str is a name of the source, and SourceType is an enum that specifies the type of the source. The enum has the following values:

• SourceType.OBSERVATION - the source is an observation passed to the agent,

• SourceType.STATE - the source is the agent’s state,

• SourceType.METRIC - the source is a metric.

When creating an instance of the RLib class, users can specify the sources using the logger_sources parameter. For example:

```python
rl = RLib(
    agent_type=EGreedy,
    agent_params={'e': 0.25},
    ext_type=RecommenderSystemExt,
    logger_types=PlotsLogger,
    logger_sources=[('action', SourceType.METRIC), ('cumulative', SourceType.METRIC)]
)
```
The above example demonstrates a logger of type `PlotsLogger` with two sources: action (representing the last action taken by the agent) and cumulative (representing the cumulative reward). These values will be logged each time the agent takes an action and receives a reward.

Users can use multiple loggers simultaneously by specifying a list of logger types in the `logger_types` parameter. For instance:

```python
rl = RLib(
    ...
    logger_types=[PlotsLogger, CsvLogger, TensorboardLogger],
    logger_sources=[('action', SourceType.METRIC), ('cumulative', SourceType.METRIC)]
)
```

In this example, three loggers (PlotsLogger, CsvLogger, and TensorboardLogger) are used, each logging actions and cumulative rewards.

Users are not restricted to predefined sources and can log any value using the `log` method of the RLib class. The `log` method takes two parameters: name and value. The example below shows how to log a value from a custom source:

```python
rl = RLib(
    ...
    logger_types=[TensorboardLogger, StdoutLogger]
)
for _ in range(epochs):
    ...
    rl.log('Epoch len', epoch_len)
```

Note that the `log` method does not take the `SourceType` parameter. In the provided example, all loggers will log all the custom values passed to the `log` method.

Loggers can be used to log values of different types. The base interface of loggers provides the following methods:

- `log_scalar` - logs a scalar value,
- `log_array` - logs an one-dimensional array,
- `log_dict` - logs a dictionary,
- `log_other` - logs a value of any other type.

The `LogsObserver` class is responsible for selecting the appropriate method of the logger for a given value based on its type.

Loggers can take a variety of parameters. The user can specify these parameters while creating an instance of the RLib class by using the `logger_params` parameter. Note that the parameters are passed to instances of all loggers listed in `logger_type`, so the user should choose appropriate parameter names not to interfere with other loggers.

One of the important concepts of loggers is the synchronization of the logged values. It is important when some values are logged more often than others. Different loggers handle the synchronization in different ways. Below you can find an illustrative example of the synchronization with the `PlotsLogger`:
The first image depicts the plot of epoch length without synchronization, where the value is logged 300 times (once per epoch). In contrast, the second plot shows the same value logged more than 40,000 times (once per step) with synchronization.
Implementing new loggers

To create your own logger, you should inherit from the abstract class `BaseLogger`. We will present creating a custom logger on the example of the `CsvLogger` logger:

```python
class CsvLogger(BaseLogger):

To begin, we need to create a constructor for the loggers. The `__init__` function is capable of accepting various arguments, which can be later provided through the `logger_params` parameter within the `RLib` class constructor. Always ensure to include `**kwargs` in the arguments list to disregard parameters used by other loggers. It is crucial to choose appropriate parameter names, considering they will be passed to instances of all loggers mentioned in `logger_type`. For instance, constructor parameters of `PlotsLogger` should with the prefix `plots_*`, while parameters of `CsvLogger` start with `csv_*`. Below, you'll find an example constructor of `CsvLogger`:

```python
def __init__(self, csv_path: str = None, **kwargs) -> None:
    super().__init__(**kwargs)
    if csv_path is None:
        csv_path = f'rlib-logs-{timestamp()}.csv'
        csv_path = os.path.join(os.path.expanduser("~"), csv_path)
    self._csv_path = csv_path
    self._current_values = set()
    self._step = 0
    self._values = defaultdict(list)
    self._steps = defaultdict(list)
```

The essential logger methods consist of several functions serving the purpose of logging scalar values, arrays, dictionaries, and other objects. Handling the transmission of logged values to the corresponding methods is the responsibility of the `LogsObserver` class. To enable our logger to record values of a specific type, we must override the appropriate methods. For instance, let us examine the `log_scalar` and `log_other` methods of `CsvLogger`:

```python
def log_scalar(self, source: Source, value: Scalar, *_) -> None:
    self._log(source, value)

def log_other(self, source: Source, value: Any, *_) -> None:
    self._log(source, f'"\n"{json.dumps(value)}"')

def _log(self, source: Source, value: any) -> None:
    name = self.source_to_name(source)
    if name in self._current_values:
        self._step += 1
        self._current_values.clear()
    self._current_values.add(name)
    self._values[name].append(value)
    self._steps[name].append(self._step)
```

These are simple methods that log scalars and values of other types. The `log_scalar` function just takes the raw scalar and saves it with a method `_log` of `CsvLogger`. Similarly, the `log_other` function converts a given value to the JSON format and then calls `_log`. The `_log` method saves the value to the `_values` dictionary and controls the `_step` variable. The `_step` variable is used to determine the current step of the logged value.
Note the use of the `source_to_name` method of `BaseLogger` that converts that source to a string. If the source is a string (just a name of an observation, state, or metric), the method returns that string. Otherwise, if the source is a tuple `(str, SourceType)`, the function returns string 

\[ \text{"[name]-[source type name]"} \]

If the logger is not able to log a value of some type (for example, it could be hard to plot a dictionary or a custom object), we do not have to implement the corresponding `log_*` method. If the user tries to log a value of that type with this logger, the library will raise the `UnsupportedLogTypeError` exception.

`BaseLogger` provides the ability to customize the initialization process by overwriting the `init` method, which takes a list of predefined sources for the logger. Additionally, there is another useful method, `finish`, which allows you to perform actions such as saving data, closing files, displaying plots, or carrying out cleanup tasks. This method is automatically triggered when an instance of the `RLib` class is deleted. Alternatively, you can manually trigger the finalization by calling `rl.finish()`. `CsvLogger` uses the `finish` method to save the logged data to a CSV file:

```python
def finish(self) -> None:
    file = open(self._csv_path, 'w')
    file.write(','.join(self._values.keys()) + '\n')

    rows, cols = self._step + 1, len(self._values)
    csv_array = np.full((rows, cols), fill_value='', dtype=object)

    for j, (name, values) in enumerate(self._values.items()):
        for i, v in enumerate(values):
            csv_array[self._steps[name][i], j] = v

    for row in csv_array:
        file.write(','.join(map(str, row)) + '\n')

    file.close()
```

**Template logger**

Here is the above code in one piece. You can copy-paste it and use it as an inspiration to create your own logger. The full source code of the `CsvLogger` can be found [here](#).

```python
import json
import os.path
from collections import defaultdict
import jax.numpy as jnp
import numpy as np
from chex import Array, Scalar
from reinforced_lib.logs import BaseLogger, Source
from reinforced_lib.utils import timestamp

class CsvLogger(BaseLogger):
    def __init__(self, csv_path: str = None, **kwargs) -> None:
        super().__init__(**kwargs)

        if csv_path is None:
            csv_path = f'rlib-logs-{timestamp()}.csv'

(continues on next page)```
```python
csv_path = os.path.join(os.path.expanduser("~"), csv_path)

self._csv_path = csv_path
self._current_values = set()
self._step = 0

self._values = defaultdict(list)
self._steps = defaultdict(list)

def finish(self) -> None:
    file = open(self._csv_path, 'w')
    file.write(','.join(self._values.keys()) + '
')

    rows, cols = self._step + 1, len(self._values)
    csv_array = np.full((rows, cols), fill_value='', dtype=object)

    for j, (name, values) in enumerate(self._values.items()):
        for i, v in enumerate(values):
            csv_array[self._steps[name][i], j] = v

    for row in csv_array:
        file.write(','.join(map(str, row)) + '\n')

    file.close()  
def log_scalar(self, source: Source, value: Scalar, *_) -> None:
    self._log(source, value)

def log_array(self, source: Source, value: Array, *_) -> None:
    if isinstance(value, (np.ndarray, jnp.ndarray)):
        value = value.tolist()
    self._log(source, f"\{{json.dumps(value)}\}")

def log_dict(self, source: Source, value: dict, *_) -> None:
    self._log(source, f"\{{json.dumps(value)}\}")

def log_other(self, source: Source, value: any, *_) -> None:
    self._log(source, f"\{{json.dumps(value)}\}")

def _log(self, source: Source, value: any) -> None:
    name = self.source_to_name(source)

    if name in self._current_values:
        self._step += 1
        self._current_values.clear()

    self._current_values.add(name)
    self._values[name].append(value)
    self._steps[name].append(self._step)
```

(continued from previous page)
4.2.6 API module

RLib class

class RLib(*, agent_type: type | None = None, agent_params: dict[str, any] | None = None, ext_type: type | None = None, ext_params: dict[str, any] | None = None, logger_types: type | list[type] | None = None, logger_sources: tuple[str, SourceType] | str | None | list[tuple[str, SourceType] | str | None] = None, logger_params: dict[str, any] | None = None, no_ext_mode: bool = False, auto_checkpoint: int | None = None, auto_checkpoint_path: str | None = None)

Main class of the library. Exposes a simple and intuitive interface to use the library.

Parameters

- **agent_type** (type, optional) – Type of the selected agent. Must inherit from the BaseAgent class.
- **agent_params** (dict, optional) – Parameters of the selected agent.
- **ext_type** (type, optional) – Type of the selected extension. Must inherit from the BaseExt class.
- **ext_params** (dict, optional) – Parameters of the selected extension.
- **logger_types** (type or list[type], optional) – Types of the selected loggers. Must inherit from the BaseLogger class.
- **logger_sources** (Source or list[Source], optional) – Sources to log.
- **logger_params** (dict, optional) – Parameters of the selected loggers.
- **no_ext_mode** (bool, default=False) – Pass observations directly to the agent (do not use the extensions).
- **auto_checkpoint** (int, optional) – Automatically save the experiment every auto_checkpoint steps. If None, the automatic checkpointing is disabled.
- **auto_checkpoint_path** (str, optional, default=~/) – Path to the directory where the automatic checkpoints will be saved.

**finish()** → None

Used to explicitly finalize the library’s work. In particular, it finishes the logger’s work.

**set_agent**(agent_type: type, agent_params: dict | None = None) → None

Initializes an agent of type agent_type with parameters agent_params. The agent type must inherit from the BaseAgent class. The agent type cannot be changed after the first agent instance has been initialized.

Parameters

- **agent_type** (type) – Type of the selected agent. Must inherit from the BaseAgent class.
- **agent_params** (dict, optional) – Parameters of the selected agent.

**set_ext**(ext_type: type, ext_params: dict | None = None) → None

Initializes an extension of type ext_type with parameters ext_params. The extension type must inherit from the BaseExt class. The extension type cannot be changed after the first agent instance has been initialized.

Parameters

- **ext_type** (type) – Type of selected extension. Must inherit from the BaseExt class.
- **ext_params** (dict, optional) – Parameters of the selected extension.
**set_loggers** (logger_types: type | list[type], logger_sources: tuple[str, SourceType] | str | None | list[tuple[str, SourceType] | str | None] = None, logger_params: dict[str, any] | None = None) → None

Initializes loggers of types `logger_types` with parameters `logger_params`. The logger types must inherit from the `BaseLogger` class. The logger types cannot be changed after the first agent instance has been initialized. `logger_types` and `logger_sources` can be objects or lists of objects, the function broadcasts them so that all loggers are connected to all sources. The `logger_sources` parameter specifies the sources to log. A source can be a name (e.g., "action") or tuple containing the name and the `SourceType` (e.g., ("action", `SourceType.OBSERVATION`)). If the name itself is inconclusive (e.g., it occurs as a metric and as an observation), the behavior depends on the implementation of the logger.

**Parameters**

- `logger_types` *(type or list[type]) – Types of the selected loggers.*
- `logger_sources` *(Source or list[Source], optional) – Sources to log.*
- `logger_params` *(dict, optional) – Parameters of the selected loggers.*

**property observation_space:** *Space*

Returns the observation space of the selected extension (or agent, if `no_ext_mode` is set).

**Returns**

Observation space of the selected extension or agent.

**Return type**

`gym.spaces.Space`

**property action_space:** *Space*

Returns the action space of the selected agent.

**Returns**

Action space of the selected agent.

**Return type**

`gym.spaces.Space`

**init**(seed: int = 42) → int

Initializes a new instance of the agent.

**Parameters**

- `seed` *(int, default=42) – Number used to initialize the JAX pseudo-random number generator.*

**Returns**

Identifier of the created instance.

**Return type**

`int`

**sample**(*args, agent_id: int = 0, is_training: bool = True, update_observations: dict | tuple | any | None = None, sample_observations: dict | tuple | any | None = None, **kwargs) → any

Takes the extension state as an input, updates the agent state, and returns the next action selected by the agent. If `no_ext_mode` is disabled, observations are passed by args and kwargs (the observations must match the extension observation space). If `no_ext_mode` is enabled, observations must be passed by the `update_observations` and `sample_observations` parameters (the observations must match the agent’s `update_observation_space` and `sample_observation_space`). If there are no agent instances initialized, the method automatically initializes the first instance. If the `is_training` flag is set, the `update` and `sample` agent methods will be called. Otherwise, only the `sample` method will be called.

**Parameters**

- `...`
• *args (tuple) – Environment observations.

• agent_id (int, default=0) – The identifier of the agent instance.

• is_training (bool) – Flag indicating whether the agent state should be updated in this step.

• update_observations (dict or tuple or any, optional) – Observations used when no_ext_mode is enabled (must match agent’s update_observation_space).

• sample_observations (dict or tuple or any, optional) – Observations used when no_ext_mode is enabled (must match agent’s sample_observation_space).

• **kwargs (dict) – Environment observations.

Returns
Action selected by the agent.

Return type
any

save(path: str | None = None, *, agent_ids: int | list[int] | None = None) → str

Saves the state of the experiment to a file in lz4 format. For each agent, both the state and the initialization parameters are saved. The extension and loggers settings are saved as well to fully reconstruct the experiment.

Parameters

• path (str, optional) – Path to the checkpoint file. If none specified, saves to the default path. If the .pkl.lz4 suffix is not detected, it will be appended automatically.

• agent_ids (int or Array, optional) – The identifier of the agent instance(s) to save. If none specified, saves the state of all agents.

Returns
Path to the saved checkpoint file.

Return type
str

static load(path: str, *, agent_params: dict[str, any] | None = None, ext_params: dict[str, any] | None = None, logger_types: type | list[type] | None = None, logger_sources: tuple[str, SourceType] | str | None | list[tuple[str, SourceType]] | str | None] | None = None, logger_params: dict[str, any] | None = None) → RLib

Loads the state of the experiment from a file in lz4 format.

Parameters

• path (str) – Path to the checkpoint file.

• agent_params (dict[str, any], optional) – Dictionary of altered agent parameters with their new values, by default None.

• ext_params (dict[str, any], optional) – Dictionary of altered extension parameters with their new values, by default None.

• logger_types (type or list[type], optional) – Types of the selected loggers. Must inherit from the BaseLogger class.

• logger_sources (Source or list[Source], optional) – Sources to log.

• logger_params (dict, optional) – Parameters of the selected loggers.
\texttt{log}(\texttt{name}: \texttt{str}, \texttt{value}: \texttt{any}) \rightarrow \texttt{None}

Logs a custom value.

**Parameters**

- \texttt{name} (\texttt{str}) – The name of the value to log.
- \texttt{value} (\texttt{any}) – The value to log.

\texttt{to_tflite}(\texttt{path}: \texttt{str} | \texttt{None} = \texttt{None}, *, \texttt{agent_id}: \texttt{int} | \texttt{None} = \texttt{None}, \texttt{sample_only}: \texttt{bool} = \texttt{False}) \rightarrow \texttt{None}

Converts the agent to a TensorFlow Lite model and saves it to a file.

**Parameters**

- \texttt{path} (\texttt{str}, \texttt{optional}) – Path to the output file.
- \texttt{agent_id} (\texttt{int}, \texttt{optional}) – The identifier of the agent instance to convert. If specified, state of the selected agent will be saved.
- \texttt{sample_only} (\texttt{bool}) – Flag indicating if the method should save only the sample function.

### 4.2.7 Agents

This module is a set of RL agents. You can either choose one of our built-in agents or implement your agent with the help of the Custom agents guide.

#### BaseAgent

**class AgentState**

Base class for agent state containers.

- \texttt{items}() \rightarrow \texttt{a set-like object providing a view on D's items}
- \texttt{keys}() \rightarrow \texttt{a set-like object providing a view on D's keys}
- \texttt{values}() \rightarrow \texttt{an object providing a view on D's values}

**class BaseAgent**

Base interface of agents.

- \texttt{abstract static init}(\texttt{key}: \texttt{Array}, *\texttt{args}, **\texttt{kwargs}) \rightarrow \texttt{AgentState}
  
  Creates and initializes instance of the agent.

- \texttt{abstract static update}(\texttt{state}: \texttt{AgentState}, \texttt{key}: \texttt{Array}, *\texttt{args}, **\texttt{kwargs}) \rightarrow \texttt{AgentState}
  
  Updates the state of the agent after performing some action and receiving a reward.

- \texttt{abstract static sample}(\texttt{state}: \texttt{AgentState}, \texttt{key}: \texttt{Array}, *\texttt{args}, **\texttt{kwargs}) \rightarrow \texttt{any}
  
  Selects the next action based on the current environment and agent state.

- \texttt{static parameter_space}() \rightarrow \texttt{Dict}
  
  Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be \texttt{gym.spaces.Dict} or \texttt{None}. If \texttt{None}, the user must provide all parameters manually.

- \texttt{property update_observation_space: Space}
  
  Observation space of the \texttt{update} method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If \texttt{None}, the user must provide all parameters manually.
property sample_observation_space: Space
    Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Space
    Action space of the agent in Gymnasium format.

export(init_key: Array, state: AgentState | None = None, sample_only: bool = False) → tuple[any, any, any]
    Exports the agent to TensorFlow Lite format.

    Parameters
    • init_key (PRNGKey) – Key used to initialize the agent.
    • state (AgentState, optional) – State of the agent to be exported. If not specified, the agent is initialized with init_key.
    • sample_only (bool, optional) – If True, the exported agent will only be able to sample actions, but not update its state.

Deep Q-Learning (DQN)

    Bases: AgentState, Mapping
    Container for the state of the deep Q-learning agent.

    params
        Parameters of the Q-network.
        
        Type
dict

    net_state
        State of the Q-network.
        
        Type
dict

    opt_state
        Optimizer state.
        
        Type
        optax.OptState

    replay_buffer
        Experience replay buffer.
        
        Type
        ReplayBuffer

    prev_env_state
        Previous environment state.
        
        Type
        Array
epsilon

*epsilon*-greedy parameter.

**Type**

Scalar

**items**() → a set-like object providing a view on D’s items

**keys**() → a set-like object providing a view on D’s keys

**values**() → an object providing a view on D’s values

class DQN(*q_network*: Module, *obs_space_shape*: Sequence[int | Any], *act_space_size*: int, *optimizer*: GradientTransformation | None = None, *experience_replay_buffer_size*: int = 10000, *experience_replay_batch_size*: int = 64, *experience_replay_steps*: int = 5, *discount*: float | int = 0.99, *epsilon*: float | int = 1.0, *epsilon_decay*: float | int = 0.999, *epsilon_min*: float | int = 0.001)

**Bases**: BaseAgent

Deep Q-learning agent¹ with *epsilon*-greedy exploration and experience replay buffer. The agent uses a deep neural network to approximate the Q-value function. The Q-network is trained to minimize the Bellman error. This agent follows the off-policy learning paradigm and is suitable for environments with discrete action spaces.

**Parameters**

- **q_network** (*nn.Module*) – Architecture of the Q-network.
- **obs_space_shape** (*Shape*) – Shape of the observation space.
- **act_space_size** (int) – Size of the action space.
- **optimizer** (*optax.GradientTransformation, optional*) – Optimizer of the Q-network. If None, the Adam optimizer with learning rate 1e-3 is used.
- **experience_replay_buffer_size** (int, default=10000) – Size of the experience replay buffer.
- **experience_replay_batch_size** (int, default=64) – Batch size of the samples from the experience replay buffer.
- **experience_replay_steps** (int, default=5) – Number of experience replay steps per update.
- **discount** (*Scalar, default=0.99*) – Discount factor. \( \gamma = 0.0 \) means no discount, \( \gamma = 1.0 \) means infinite discount. \( 0 \leq \gamma \leq 1 \)
- **epsilon** (*Scalar, default=1.0*) – Initial *epsilon*-greedy parameter. \( 0 \leq \epsilon \leq 1 \).
- **epsilon_decay** (*Scalar, default=0.999*) – Epsilon decay factor. \( \epsilon_{t+1} = \epsilon_t \times \epsilon_{\text{decay}}. \) \( 0 \leq \epsilon_{\text{decay}} \leq 1 \).
- **epsilon_min** (*Scalar, default=0.01*) – Minimum *epsilon*-greedy parameter. \( 0 \leq \epsilon_{\text{min}} \leq \epsilon \).

static parameter_space() → Dict

Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

property update_observation_space: Dict

Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property sample_observation_space: Dict

Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Discrete

Action space of the agent in Gymnasium format.

static init(key: Array, obs_space_shape: Sequence[int | Any], q_network: Module, optimizer: GradientTransformation, er: ExperienceReplay, epsilon: float | int) → DQNState

Initializes the Q-network, optimizer and experience replay buffer with given parameters. The first state of the environment is assumed to be a tensor of zeros.

Parameters

- key (PRNGKey) – A PRNG key used as the random key.
- obs_space_shape (Shape) – The shape of the observation space.
- q_network (nn.Module) – The Q-network.
- optimizer (optax.GradientTransformation) – The optimizer.
- er (ExperienceReplay) – The experience replay buffer.
- epsilon (Scalar) – The initial \( \epsilon \)-greedy parameter.

Returns

Initial state of the deep Q-learning agent.

Return type

DQNState


Loss is the mean squared Bellman error \( \mathcal{L}(\theta) = \mathbb{E}_{s,a,r,s'} \left[ (r + \gamma \max_{a'} Q(s', a') - Q(s, a))^2 \right] \) where \( s \) is the current state, \( a \) is the current action, \( r \) is the reward, \( s' \) is the next state, \( \gamma \) is the discount factor, \( Q(s, a) \) is the Q-value of the state-action pair. Loss can be calculated on a batch of transitions.

Parameters

- params (dict) – The parameters of the Q-network.
- key (PRNGKey) – A PRNG key used as the random key.
- net_state (dict) – The state of the Q-network.
- params_target (dict) – The parameters of the target Q-network.
- net_state_target (dict) – The state of the target Q-network.
• **batch** (*tuple*) – A batch of transitions from the experience replay buffer.

• **q_network** (*nn.Module*) – The Q-network.

• **discount** (*Scalar*) – The discount factor.

**Returns**
The loss and the new state of the Q-network.

**Return type**
`Tuple[Scalar, dict]`

```python
```

Appends the transition to the experience replay buffer and performs `experience_replay_steps` steps. Each step consists of sampling a batch of transitions from the experience replay buffer, calculating the loss using the `loss_fn` function and performing a gradient step on the Q-network. The $\epsilon$-greedy parameter is decayed by `epsilon_decay`.

**Parameters**

• **state** (*DQNState*) – The current state of the deep Q-learning agent.

• **key** (*PRNGKey*) – A PRNG key used as the random key.

• **env_state** (*Array*) – The current state of the environment.

• **action** (*Array*) – The action taken by the agent.

• **reward** (*Scalar*) – The reward received by the agent.

• **terminal** (*bool*) – Whether the episode has terminated.

• **step_fn** (*Callable*) – The function that performs a single gradient step on the Q-network.

• **er** (*ExperienceReplay*) – The experience replay buffer.

• **experience_replay_steps** (*int*) – The number of experience replay steps.

• **epsilon_decay** (*Scalar*) – The decay rate of the $\epsilon$-greedy parameter.

• **epsilon_min** (*Scalar*) – The minimum value of the $\epsilon$-greedy parameter.

**Returns**
The updated state of the deep Q-learning agent.

**Return type**
`DQNState`

```python
static sample(state: DQNState, key: Array, env_state: Array | ndarray | bool_ | number, q_network: nn.Module, act_space_size: int) → int
```

Samples random action with probability $\epsilon$ and the greedy action with probability $1 - \epsilon$. The greedy action is the action with the highest Q-value.

**Parameters**

• **state** (*DQNState*) – The state of the deep Q-learning agent.

• **key** (*PRNGKey*) – A PRNG key used as the random key.

• **env_state** (*Array*) – The current state of the environment.

• **q_network** (*nn.Module*) – The Q-network.
• **act_space_size** (int) – The size of the action space.

  **Returns**
  
  | Selected action.

  **Return type**
  
  int

**Double Deep Q-Learning (DDQN)**

class DDQNState(

**params**: dict,
  **net_state**: dict,
  **params_target**: dict,
  **net_state_target**: dict,
  **opt_state**: Array | ndarray | bool_ | number | Iterable[ArrayTree] | Mapping[dict, ArrayTree],
  **replay_buffer**: ReplayBuffer,
  **prev_env_state**: Array | ndarray | bool_ | number,
  **epsilon**: float | int)

**Bases**: AgentState, Mapping

Container for the state of the double deep Q-learning agent.

  **params**
  
  Parameters of the main Q-network.

  **Type**
  
  dict

  **net_state**
  
  State of the main Q-network.

  **Type**
  
  dict

  **params_target**
  
  Parameters of the target Q-network.

  **Type**
  
  dict

  **net_state_target**
  
  State of the target Q-network.

  **Type**
  
  dict

  **opt_state**
  
  Optimizer state of the main Q-network.

  **Type**
  
  optax.OptState

  **replay_buffer**
  
  Experience replay buffer.

  **Type**
  
  ReplayBuffer

  **prev_env_state**
  
  Previous environment state.

  **Type**
  
  Array
epsilon
greedy parameter.

Type
Scalar

items() \rightarrow \text{a set-like object providing a view on D's items}

keys() \rightarrow \text{a set-like object providing a view on D's keys}

values() \rightarrow \text{an object providing a view on D's values}

class DDQN(q_network: Module, obs_space_shape: Sequence[int | Any], act_space_size: int, optimizer:
  GradientTransformation | None = None, experience_replay_buffer_size: int = 10000,
  experience_replay_batch_size: int = 64, experience_replay_steps: int = 5, discount: float | int = 0.99,
  epsilon: float | int = 1.0, epsilon_decay: float | int = 0.999, epsilon_min: float | int = 0.001, tau: float | int = 0.01)

Bases: BaseAgent

Double deep Q-learning agent\(^2\) with \(\epsilon\)-greedy exploration and experience replay buffer. The agent uses two Q-networks to stabilize the learning process and avoid overestimation of the Q-values. The main Q-network is trained to minimize the Bellman error. The target Q-network is updated with a soft update. This agent follows the off-policy learning paradigm and is suitable for environments with discrete action spaces.

Parameters

- **q_network (nn.Module)** – Architecture of the Q-networks.
- **obs_space_shape (Shape)** – Shape of the observation space.
- **act_space_size (int)** – Size of the action space.
- **optimizer (optax.GradientTransformation, optional)** – Optimizer of the Q-networks. If None, the Adam optimizer with learning rate 1e-3 is used.
- **experience_replay_buffer_size (int, default=10000)** – Size of the experience replay buffer.
- **experience_replay_batch_size (int, default=64)** – Batch size of the samples from the experience replay buffer.
- **experience_replay_steps (int, default=5)** – Number of experience replay steps per update.
- **discount (Scalar, default=0.99)** – Discount factor. \(\gamma = 0.0\) means no discount, \(\gamma = 1.0\) means infinite discount. \(0 \leq \gamma \leq 1\)
- **epsilon (Scalar, default=1.0)** – Initial \(\epsilon\)-greedy parameter. \(0 \leq \epsilon \leq 1\).
- **epsilon_decay (Scalar, default=0.999)** – Epsilon decay factor. \(\epsilon_{t+1} = \epsilon_t \times \epsilon_{\text{decay}}.0 \leq \epsilon_{\text{decay}} \leq 1\).
- **epsilon_min (Scalar, default=0.01)** – Minimum \(\epsilon\)-greedy parameter. \(0 \leq \epsilon_{\text{min}} \leq \epsilon\).
- **tau (Scalar, default=0.01)** – Soft update factor. \(\tau = 0.0\) means no soft update, \(\tau = 1.0\) means hard update. \(0 \leq \tau \leq 1\).

static parameter_space() → Dict
Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

property update_observation_space: Dict
Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property sample_observation_space: Dict
Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Discrete
Action space of the agent in Gymnasium format.

static init(key: Array, obs_space_shape: Sequence[int | Any], q_network: Module, optimizer: GradientTransformation, er: ExperienceReplay, epsilon: float | int) → DDQNState
Initializes the Q-networks, optimizer and experience replay buffer with given parameters. The first state of the environment is assumed to be a tensor of zeros.

Parameters
• key (PRNGKey) – A PRNG key used as the random key.
• obs_space_shape (Shape) – The shape of the observation space.
• q_network (nn.Module) – The Q-network.
• optimizer (optax.GradientTransformation) – The optimizer.
• er (ExperienceReplay) – The experience replay buffer.
• epsilon (Scalar) – The initial \( \epsilon \)-greedy parameter.

Returns
Initial state of the double Q-learning agent.

Return type
DDQNState

static loss_fn(params: dict, key: Array, state: DDQNState, batch: tuple, q_network: Module, discount: float | int) → tuple[float | int, dict]
Loss is the mean squared Bellman error \( \mathcal{L}(\theta) = \mathbb{E}_{s,a,r,s'} \left[ (r + \gamma \max_{a'} Q'(s', a') - Q(s, a))^2 \right] \) where \( s \) is the current state, \( a \) is the current action, \( r \) is the reward, \( s' \) is the next state, \( \gamma \) is the discount factor, \( Q(s, a) \) is the Q-value of the main Q-network, \( Q'(s', a') \) is the Q-value of the target Q-network. Loss can be calculated on a batch of transitions.

Parameters
• params (dict) – The parameters of the Q-network.
• key (PRNGKey) – A PRNG key used as the random key.
• state (DDQNState) – The state of the double deep Q-learning agent.
• batch (tuple) – A batch of transitions from the experience replay buffer.
• q_network (nn.Module) – The Q-network.
• **discount** (*Scalar*) – The discount factor.

**Returns**
The loss and the new state of the Q-network.

**Return type**
tuple[Scalar, dict]

```python
```

Appends the transition to the experience replay buffer and performs `experience_replay_steps` steps. Each step consists of sampling a batch of transitions from the experience replay buffer, calculating the loss using the `loss_fn` function, performing a gradient step on the main Q-network, and soft updating the target Q-network. Soft update of the parameters is defined as $\theta_{\text{target}} = \tau \theta + (1 - \tau)\theta_{\text{target}}$. The $\epsilon$-greedy parameter is decayed by `epsilon_decay`.

**Parameters**
- **state** (*DDQNState*) – The current state of the double Q-learning agent.
- **key** (*PRNGKey*) – A PRNG key used as the random key.
- **env_state** (*Array*) – The current state of the environment.
- **action** (*Array*) – The action taken by the agent.
- **reward** (*Scalar*) – The reward received by the agent.
- **terminal** (*bool*) – Whether the episode has terminated.
- **step_fn** (*Callable*) – The function that performs a single gradient step on the Q-network.
- **er** (*ExperienceReplay*) – The experience replay buffer.
- **experience_replay_steps** (*int*) – The number of experience replay steps.
- **epsilon_decay** (*Scalar*) – The decay rate of the $\epsilon$-greedy parameter.
- **epsilon_min** (*Scalar*) – The minimum value of the $\epsilon$-greedy parameter.
- **tau** (*Scalar*) – The soft update parameter.

**Returns**
The updated state of the double Q-learning agent.

**Return type**
*DDQNState*

```python
def static sample(state: DDQNState, key: Array, env_state: Array | ndarray | bool_ | number, q_network: nn.Module, act_space_size: int) -> int
```

Samples random action with probability $\epsilon$ and the greedy action with probability $1 - \epsilon$ using the main Q-network. The greedy action is the action with the highest Q-value.

**Parameters**
- **state** (*DDQNState*) – The state of the double Q-learning agent.
- **key** (*PRNGKey*) – A PRNG key used as the random key.
- **env_state** (*Array*) – The current state of the environment.
- **q_network** (*nn.Module*) – The Q-network.
- **act_space_size** (*int*) – The size of the action space.
Returns
Selected action.

Return type
int

Deep Expected SARSA

class ExpectedSarsaState

Bases: AgentState, Mapping

Container for the state of the deep expected SARSA agent.

params
Parameters of the Q-network.

Type
dict

net_state
State of the Q-network.

Type
dict

opt_state
Optimizer state.

Type
optax.OptState

replay_buffer
Experience replay buffer.

Type
ReplayBuffer

prev_env_state
Previous environment state.

Type
Array

items() → a set-like object providing a view on D’s items

keys() → a set-like object providing a view on D’s keys

values() → an object providing a view on D’s values

class ExpectedSara

Bases: BaseAgent

Deep expected SARSA agent with temperature parameter $\tau$ and experience replay buffer. The agent uses a deep neural network to approximate the Q-value function. The Q-network is trained to minimize the Bellman error. This agent follows the on-policy learning paradigm and is suitable for environments with discrete action spaces.
Parameters

• **q_network** *(nn.Module)* – Architecture of the Q-network.

• **obs_space_shape** *(Shape)* – Shape of the observation space.

• **act_space_size** *(int)* – Size of the action space.

• **optimizer** *(optax.GradientTransformation, optional)* – Optimizer of the Q-network. If None, the Adam optimizer with learning rate 1e-3 is used.

• **experience_replay_buffer_size** *(int, default=10000)* – Size of the experience replay buffer.

• **experience_replay_batch_size** *(int, default=64)* – Batch size of the samples from the experience replay buffer.

• **experience_replay_steps** *(int, default=5)* – Number of experience replay steps per update.

• **discount** *(Scalar, default=0.99)* – Discount factor. $\gamma = 0$ means no discount, $\gamma = 1.0$ means infinite discount. $0 \leq \gamma \leq 1$

• **tau** *(Scalar, default=1.0)* – Temperature parameter. $\tau = 0.0$ means no exploration, $\tau = \infty$ means infinite exploration. $\tau > 0$

**static parameter_space() → Dict**

Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

**property update_observation_space: Dict**

Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extension and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

**property sample_observation_space: Dict**

Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extension and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

**property action_space: Discrete**

Action space of the agent in Gymnasium format.

**static init(key: Array, obs_space_shape: Sequence[int | Any], q_network: Module, optimizer: GradientTransformation, er: ExperienceReplay) → ExpectedSarsaState**

Initializes the Q-network, optimizer and experience replay buffer with given parameters. The first state of the environment is assumed to be a tensor of zeros.

Parameters

• **key** *(PRNGKey)* – A PRNG key used as the random key.

• **obs_space_shape** *(Shape)* – The shape of the observation space.

• **q_network** *(nn.Module)* – The Q-network.

• **optimizer** *(optax.GradientTransformation)* – The optimizer.

• **er** *(ExperienceReplay)* – The experience replay buffer.

Returns

Initial state of the deep expected SARSA agent.
Return type

*ExpectedSarsaState*

**static loss_fn**

```python
```

Loss is the mean squared Bellman error $L(\theta) = \mathbb{E}_{s,a,r,s'} (r + \gamma \sum_{a'} \pi(a'|s')Q(s', a') - Q(s, a))^2$ where $s$ is the current state, $a$ is the current action, $r$ is the reward, $s'$ is the next state, $\gamma$ is the discount factor, $Q(s, a)$ is the Q-value of the state-action pair. Loss can be calculated on a batch of transitions.

**Parameters**

- **params** *(dict)* – The parameters of the Q-network.
- **key** *(PRNGKey)* – A PRNG key used as the random key.
- **net_state** *(dict)* – The state of the Q-network.
- **params_target** *(dict)* – The parameters of the target Q-network.
- **net_state_target** *(dict)* – The state of the target Q-network.
- **batch** *(tuple)* – A batch of transitions from the experience replay buffer.
- **q_network** *(nn.Module)* – The Q-network.
- **discount** *(Scalar)* – The discount factor.
- **tau** *(Scalar)* – The temperature parameter.

**Returns**

The loss and the new state of the Q-network.

**Return type**

Tuple[Scalar, dict]

**static update**

```python
```

Appends the transition to the experience replay buffer and performs `experience_replay_steps` steps. Each step consists of sampling a batch of transitions from the experience replay buffer, calculating the loss using the `loss_fn` function and performing a gradient step on the Q-network.

**Parameters**

- **state** *(ExpectedSarsaState)* – The current state of the deep expected SARSA agent.
- **key** *(PRNGKey)* – A PRNG key used as the random key.
- **env_state** *(Array)* – The current state of the environment.
- **action** *(Array)* – The action taken by the agent.
- **reward** *(Scalar)* – The reward received by the agent.
- **terminal** *(bool)* – Whether the episode has terminated.
- **q_network** *(nn.Module)* – The Q-network.
- **step_fn** *(Callable)* – The function that performs a single gradient step on the Q-network.
- **er** *(ExperienceReplay)* – The experience replay buffer.
- **experience_replay_steps** *(int)* – The number of experience replay steps.
Returns
The updated state of the deep expected SARSA agent.

Return type
ExpectedSarsaState

static sample(state: ExpectedSarsaState, key: Array, env_state: Array | ndarray | bool_ | number, 
q_network: Module, act_space_size: int, tau: float | int) → int

Selects an action using the softmax policy with the temperature parameter $\tau$:

$$
\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}
$$

Parameters
- **state** (ExpectedSarsaState) – The state of the deep expected SARSA agent.
- **key** (PRNGKey) – A PRNG key used as the random key.
- **env_state** (Array) – The current state of the environment.
- **q_network** (nn.Module) – The Q-network.
- **act_space_size** (int) – The size of the action space.
- **tau** (Scalar) – The temperature parameter.

Returns
Selected action.

Return type
int

Deep Deterministic Policy Gradient (DDPG)

class DDPGState(q_params: dict, q_net_state: dict, q_params_target: dict, q_net_state_target: dict, q_opt_state: 
Array | ndarray | bool_ | number | Iterable[ArrayTree] | Mapping[Any, ArrayTree], a_params: 
dict, a_net_state: dict, a_params_target: dict, a_net_state_target: dict, a_opt_state: Array | 
ndarray | bool_ | number | Iterable[ArrayTree] | Mapping[Any, ArrayTree], replay_buffer: 
ReplayBuffer, prev_env_state: Array | ndarray | bool_ | number, noise: float | int)

Bases: AgentState, Mapping

Container for the state of the deep deterministic policy gradient agent.

**q_params**
Parameters of the Q-network.

Type
dict

**q_net_state**
State of the Q-network.

Type
dict

**q_params_target**
Parameters of the target Q-network.

Type
dict
**q_net_state_target**
State of the target Q-network.

**Type**
dict

**q_opt_state**
Optimizer state of the Q-network.

**Type**
optax.OptState

**a_params**
Parameters of the policy network.

**Type**
dict

**a_net_state**
State of the policy network.

**Type**
dict

**a_params_target**
Parameters of the target policy network.

**Type**
dict

**a_net_state_target**
State of the target policy network.

**Type**
dict

**a_opt_state**
Optimizer state of the policy network.

**Type**
optax.OptState

**replay_buffer**
Experience replay buffer.

**Type**
*ReplayBuffer*

**prev_env_state**
Previous environment state.

**Type**
Array

**noise**
Current noise level.

**Type**
Scalar

**items()** → a set-like object providing a view on D's items
**keys()** → a set-like object providing a view on D's keys

**values()** → an object providing a view on D’s values

```python
class DDPG(q_network: Module, a_network: Module, obs_space_shape: Sequence[int | Any], act_space_shape: Sequence[int | Any], min_action: Array | ndarray | bool_ | number | float | int, max_action: Array | ndarray | bool_ | number | float | int, q_optimizer: GradientTransformation | None = None, a_optimizer: GradientTransformation | None = None, experience_replay_buffer_size: int = 10000, experience_replay_batch_size: int = 64, experience_replay_steps: int = 5, discount: float | int = 0.99, noise: float | int | None = None, noise_decay: float | int = 0.99, noise_min: float | int = 0.01, tau: float | int = 0.01)
```

Bases: `BaseAgent`

Deep deterministic policy gradient agent with white Gaussian noise exploration and experience replay buffer. The agent simultaneously learns a Q-function and a policy. The Q-function is updated using the Bellman equation. The policy is learned using the gradient of the Q-function with respect to the policy parameters to maximize the Q-value. The agent uses two Q-networks and two policy networks to stabilize the learning process and avoid overestimation. The target networks are updated with a soft update. This agent follows the off-policy learning paradigm and is suitable for environments with continuous action spaces.

**Parameters**

- **q_network** (nn.Module) – Architecture of the Q-networks. The input to the network should be two tensors of observations and actions respectively.
- **a_network** (nn.Module) – Architecture of the policy networks.
- **obs_space_shape** (Shape) – Shape of the observation space.
- **act_space_shape** (Shape) – Shape of the action space.
- **min_action** (Scalar or Array) – Minimum action value.
- **max_action** (Scalar or Array) – Maximum action value.
- **q_optimizer** (optax.GradientTransformation, optional) – Optimizer of the Q-networks. If None, the Adam optimizer with learning rate 1e-3 is used.
- **a_optimizer** (optax.GradientTransformation, optional) – Optimizer of the policy networks. If None, the Adam optimizer with learning rate 1e-3 is used.
- **experience_replay_buffer_size** (int, default=10000) – Size of the experience replay buffer.
- **experience_replay_batch_size** (int, default=64) – Batch size of the samples from the experience replay buffer.
- **experience_replay_steps** (int, default=5) – Number of experience replay steps per update.
- **discount** (Scalar, default=0.99) – Discount factor. \( \gamma = 0.0 \) means no discount, \( \gamma = 1.0 \) means infinite discount. \( 0 \leq \gamma \leq 1 \)
- **noise** (Scalar, default=(max_action - min_action) / 2) – Initial Gaussian noise level. \( 0 \leq \sigma \)
- **noise_decay** (Scalar, default=0.99) – Gaussian noise decay factor. \( \sigma_{t+1} = \sigma_t \times \sigma_{decay} \), \( 0 \leq \sigma_{decay} \leq 1 \)

---


4.2. API Documentation
• **noise_min** *(Scalar, default=0.01)* – Minimum Gaussian noise level. $0 \leq \sigma_{\text{min}} \leq \sigma$.

• **tau** *(Scalar, default=0.01)* – Soft update factor. $\tau = 0.0$ means no soft update, $\tau = 1.0$ means hard update. $0 \leq \tau \leq 1$.

**References**

static parameter_space() → Dict

Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

property update_observation_space: Dict

Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property sample_observation_space: Dict

Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Box

Action space of the agent in Gymnasium format.

static init(key: Array, obs_space_shape: Sequence[| Any], act_space_shape: Sequence[| Any], q_network: Module, a_network: Module, q_optimizer: GradientTransformation, a_optimizer: GradientTransformation, er: ExperienceReplay, noise: float | int) → DDPGState

Initializes the Q-networks and the policy networks, optimizers, and experience replay buffer. The first state of the environment is assumed to be a tensor of zeros.

Parameters

• **key** *(PRNGKey)* – A PRNG key used as the random key.

• **obs_space_shape** *(Shape)* – The shape of the observation space.

• **act_space_shape** *(Shape)* – The shape of the action space.

• **q_network** *(nn.Module)* – The Q-network.

• **a_network** *(nn.Module)* – The policy network.

• **q_optimizer** *(optax.GradientTransformation)* – The Q-network optimizer.

• **a_optimizer** *(optax.GradientTransformation)* – The policy network optimizer.

• **er** *(ExperienceReplay)* – The experience replay buffer.

• **noise** *(Scalar)* – The initial noise value.

Returns

Initial state of the deep deterministic policy gradient agent.

Return type

**DDPGState**

static q_loss_fn(q_params: dict, key: Array, state: DDPGState, batch: tuple, q_network: Module, a_network: Module, discount: float | int) → tuple[float | int, dict]

Loss is the mean squared Bellman error $\mathcal{L}(\theta) = \mathbb{E}_{s,a,r,s'} \left[ (r + \gamma \max_a Q'(s', \pi'(s')) - Q(s, a))^2 \right]$ where $s$ is the current state, $a$ is the current action, $r$ is the reward, $s'$ is the next state, $\gamma$ is the discount factor,
$Q(s, a)$ is the Q-value of the main Q-network, $Q'(s, a)$ is the Q-value of the target Q-network, and $\pi'(s)$ is the action of the target policy network. The policy network parameters are considered as fixed. Loss can be calculated on a batch of transitions.

Parameters
- **q_params** (dict) – The parameters of the Q-network.
- **key** (PRNGKey) – A PRNG key used as the random key.
- **state** (DDPGState) – The state of the deep deterministic policy gradient agent.
- **batch** (tuple) – A batch of transitions from the experience replay buffer.
- **q_network** (nn.Module) – The Q-network.
- **a_network** (nn.Module) – The policy network.
- **discount** (Scalar) – The discount factor.

Returns
The loss and the new state of the Q-network.

Return type
tuple[Scalar, dict]

static a_loss_fn(a_params: dict, key: Array, state: DDPGState, batch: tuple, q_network: Module, a_network: Module) \rightarrow tuple[float | int, dict]

The policy network is updated using the gradient of the Q-network to maximize the Q-value of the current state and action $\max_{\theta} E_{s,a} [Q(s, \pi_\theta(s))]$. Q-network parameters are considered as fixed. The policy network can be updated on a batch of transitions.

Parameters
- **a_params** (dict) – The parameters of the policy network.
- **key** (PRNGKey) – A PRNG key used as the random key.
- **state** (DDPGState) – The state of the deep deterministic policy gradient agent.
- **batch** (tuple) – A batch of transitions from the experience replay buffer.
- **q_network** (nn.Module) – The Q-network.
- **a_network** (nn.Module) – The policy network.

Returns
The loss and the new state of the policy network.

Return type
tuple[Scalar, dict]


Appends the transition to the experience replay buffer and performs experience_replay_steps steps.

Each step consists of sampling a batch of transitions from the experience replay buffer, calculating the Q-network loss and the policy network loss using q_loss_fn and a_loss_fn respectively, performing a gradient step on both networks, and soft updating the target networks. Soft update of the parameters is defined as $\theta_{target} = \tau \theta + (1 - \tau) \theta_{target}$. The noise parameter is decayed by noise_decay.

Parameters
- **state** (DDPGState) – The current state of the double Q-learning agent.
• **key** (*PRNGKey*) – A PRNG key used as the random key.
• **env_state** (*Array*) – The current state of the environment.
• **action** (*Array*) – The action taken by the agent.
• **reward** (*Scalar*) – The reward received by the agent.
• **terminal** (*bool*) – Whether the episode has terminated.
• **q_step_fn** (*Callable*) – The function that performs a single gradient step on the Q-network.
• **a_step_fn** (*Callable*) – The function that performs a single gradient step on the policy network.
• **er** (*ExperienceReplay*) – The experience replay buffer.
• **experience_replay_steps** (*int*) – The number of experience replay steps.
• **noise_decay** (*Scalar*) – The decay rate of the noise parameter.
• **noise_min** (*Scalar*) – The minimum value of the noise parameter.
• **tau** (*Scalar*) – The soft update parameter.

**Returns**
The updated state of the deep deterministic policy gradient agent.

**Return type**
*DDPGState*

```python
def static sample(state: DDPGState, key: Array, env_state: Array | ndarray | bool_ | number, a_network: Module, min_action: float | int, max_action: float | int) -> Array | ndarray | bool_ | number | float | int
```

Calculates deterministic action using the policy network. Then adds white Gaussian noise with standard deviation `state.noise` to the action and clips it to the range \([min\_action, max\_action]\).

**Parameters**
• **state** (*DDPGState*) – The state of the double Q-learning agent.
• **key** (*PRNGKey*) – A PRNG key used as the random key.
• **env_state** (*Array*) – The current state of the environment.
• **a_network** (*nn.Module*) – The policy network.
• **min_action** (*Scalar or Array*) – The minimum value of the action.
• **max_action** (*Scalar or Array*) – The maximum value of the action.

**Returns**
Selected action.

**Return type**
*Scalar or Array*
**Epsilon-greedy**

```python
class EGreedyState(Q: Array | ndarray | bool_ | number, N: Array | ndarray | bool_ | number)
    Bases: AgentState, Mapping
    Container for the state of the $\epsilon$-greedy agent.
    Q
        Action-value function estimates for each arm.
        Type Array
    N
        Number of tries for each arm.
        Type Array

items() \rightarrow a set-like object providing a view on D's items
keys() \rightarrow a set-like object providing a view on D's keys
values() \rightarrow an object providing a view on D's values
```

```python
class EGreedy(n_arms: int, e: float | int, optimistic_start: float | int = 0.0, alpha: float | int = 0.0)
    Bases: BaseAgent
    Epsilon-greedy agent with an optimistic start behavior and optional exponential recency-weighted average update. It selects a random action from a set of all actions $A$ with probability $\epsilon$ (exploration), otherwise it chooses the currently best action (exploitation).
    Parameters
    - n_arms (int) – Number of bandit arms. $N \in \mathbb{N}_+$.
    - e (float) – Experiment rate (epsilon). $\epsilon \in [0, 1]$.
    - optimistic_start (float, default=0.0) – Interpreted as the optimistic start to encourage exploration in the early stages.
    - alpha (float, default=0.0) – If non-zero, exponential recency-weighted average is used to update $Q$ values. $\alpha \in [0, 1]$.
```

**References**

- static parameter_space() \rightarrow Dict
  Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

- property update_observation_space: Dict
  Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

---

property sample_observation_space: Dict
Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Discrete
Action space of the agent in Gymnasium format.

static init(key: Array, n_arms: int, optimistic_start: float | int) → EGreedyState
Creates and initializes instance of the $\epsilon$-greedy agent for n_arms arms. Action-value function estimates are set to optimistic_start value and the number of tries is one for each arm.

Parameters
- key (PRNGKey) – A PRNG key used as the random key.
- n_arms (int) – Number of bandit arms.
- optimistic_start (float) – Interpreted as the optimistic start to encourage exploration in the early stages.

Returns
Initial state of the $\epsilon$-greedy agent.

Return type
EGreedyState

static update(state: EGreedyState, key: Array, action: int, reward: float | int, alpha: float | int) → EGreedyState
In the stationary case, the action-value estimate for a given arm is updated as $Q_{t+1} = Q_t + \frac{1}{t}[R_t - Q_t]$ after receiving reward $R_t$ at step $t$ and the number of tries for the corresponding arm is incremented. In the non-stationary case, the update follows the equation $Q_{t+1} = Q_t + \alpha[R_t - Q_t]$.

Parameters
- state (EGreedyState) – Current state of the agent.
- key (PRNGKey) – A PRNG key used as the random key.
- action (int) – Previously selected action.
- reward (float) – Reward collected by the agent after taking the previous action.
- alpha (float) – Exponential recency-weighted average factor (used when $\alpha > 0$).

Returns
Updated agent state.

Return type
EGreedyState

static sample(state: EGreedyState, key: Array, e: float | int) → int
Epsilon-greedy agent follows the policy:
$$A = \begin{cases} \arg\max_{a \in A} Q(a) & \text{with probability } 1 - \epsilon, \\ \text{random action} & \text{with probability } \epsilon. \end{cases}$$

Parameters
- state (EGreedyState) – Current state of the agent.
- key (PRNGKey) – A PRNG key used as the random key.
• \( \epsilon (\text{float}) \) – Experiment rate (epsilon).

**Returns**
Selected action.

**Return type**
int

## Exp3

class **Exp3State** (omega: Array | ndarray | bool_ | number)

Bases: AgentState, Mapping

Container for the state of the Exp3 agent.

**omega**
Preference for each arm.

Type
Array

**items()** → a set-like object providing a view on D’s items

**keys()** → a set-like object providing a view on D’s keys

**values()** → an object providing a view on D’s values

class **Exp3** (n_arms: int, gamma: float | int, min_reward: float | int, max_reward: float | int)

Bases: BaseAgent

Basic Exp3 agent for stationary multi-armed bandit problems with exploration factor \( \gamma \). The higher the value, the more the agent explores. The implementation is inspired by the work of Auer et al.\(^6\). There are many variants of the Exp3 algorithm, you can find more information in the original paper.

**Parameters**

• \( \text{n_arms} (\text{int}) \) – Number of bandit arms. \( N \in \mathbb{N}_+ \).

• \( \text{gamma} (\text{float}) \) – Exploration factor. \( \gamma \in (0, 1] \).

• \( \text{min_reward} (\text{float}) \) – Minimum possible reward.

• \( \text{max_reward} (\text{float}) \) – Maximum possible reward.

**References**

static **parameter_space** () → Dict

Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

**property update_observation_space: Dict**

Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

---

property sample_observation_space: Dict
Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extension and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Discrete
Action space of the agent in Gymnasium format.

static init(key: Array, n_arms: int) → Exp3State
Initializes the Exp3 agent state with uniform preference for each arm.

Parameters
• key (PRNGKey) – A PRNG key used as the random key.
• n_arms (int) – Number of bandit arms.

Returns
Initial state of the Exp3 agent.

Return type
Exp3State

Agent updates its preference for the selected arm $a$ according to the following formula:

$$\omega_{t+1}(a) = \omega_t(a) \exp\left(\frac{\gamma r}{\pi(a)K}\right)$$

where $\omega_{t+1}(a)$ is the preference of arm $a$ at time $t + 1$, $\pi(a)$ is the probability of selecting arm $a$, and $K$ is the number of arms. The reward $r$ is normalized to the range $[0, 1]$. The exponential growth significantly increases the weight of good arms, so in the long use of the agent it is important to ensure that the values of $\omega$ do not exceed the maximum value of the floating point type!

Parameters
• state (Exp3State) – Current state of the agent.
• key (PRNGKey) – A PRNG key used as the random key.
• action (int) – Previously selected action.
• reward (float) – Reward collected by the agent after taking the previous action.
• gamma (float) – Exploration factor.
• min_reward (float) – Minimum possible reward.
• max_reward (float) – Maximum possible reward.

Returns
Updated agent state.

Return type
Exp3State

static sample(state: Exp3State, key: Array, gamma: float | int) → int
The Exp3 policy is stochastic. Algorithm chooses a random arm with probability $\gamma$, otherwise it draws arm $a$ with probability $\omega(a)/\sum_{b=1}^{N} \omega(b)$.

Parameters
• state (Exp3State) – Current state of the agent.
• **key** (*PRNGKey*) – A PRNG key used as the random key.

• **gamma** (*float*) – Exploration factor.

**Returns**

Selected action.

**Return type**

`int`

---

**Softmax**

```python
class SoftmaxState(H: Array | ndarray | bool_ | number, r: float | int, n: int)
    Bases: AgentState, Mapping
    Container for the state of the Softmax agent.

    H
        Preference for each arm.
        Type
            Array

    r
        Average of all obtained rewards $\tilde{R}$.
        Type
            float

    n
        Step number.
        Type
            int

    items() → a set-like object providing a view on D's items

    keys() → a set-like object providing a view on D's keys

    values() → an object providing a view on D's values
```

```python
class Softmax(n_arms: int, lr: float | int, alpha: float | int = 0.0, tau: float | int = 1.0, multiplier: float | int = 1.0)
    Bases: BaseAgent
    Softmax agent with baseline and optional exponential recency-weighted average update. It learns a preference
    function $H$, which indicates a preference of selecting one arm over others. Algorithm policy can be controlled
    by the temperature parameter $\tau$. The implementation is inspired by the work of Sutton and Barto. Note: For
    this agent, some environments find it very beneficial to use 64-bit JAX mode!

    **Parameters**

    • **n_arms** (*int*) – Number of bandit arms. $N \in \mathbb{N}_+$.
    • **lr** (*float*) – Step size. $lr > 0$.
    • **alpha** (*float*, *default=0.0*) – If non-zero, exponential recency-weighted average is used to update $\tilde{R}$. $\alpha \in [0, 1]$.
    • **tau** (*float*, *default=1.0*) – Temperature parameter. $\tau > 0$.
    • **multiplier** (*float*, *default=1.0*) – Multiplier for the reward. $multiplier > 0$.
```
static parameter_space() → Dict
Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

property update_observation_space: Dict
Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property sample_observation_space: Dict
Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Discrete
Action space of the agent in Gymnasium format.

static init(key: Array, n_arms: int) → SoftmaxState
Creates and initializes instance of the Softmax agent for n_arms arms. Preferences $H$ for each arm are set to zero, as well as the average of all rewards $\bar{R}$. The step number $n$ is initialized to one.

Parameters
- **key** (PRNGKey) – A PRNG key used as the random key.
- **n_arms** (int) – Number of bandit arms.

Returns
Initial state of the Softmax agent.

Return type
SoftmaxState

Preferences $H$ can be learned by stochastic gradient ascent. The softmax algorithm searches for such a set of preferences that maximizes the expected reward $E[R]$. The updates of $H$ for each action $a$ are calculated as:

$$H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(\nabla A_t = a - \pi_t(a)),$$

where $\bar{R}_t$ is the average of all rewards up to but not including step $t$ (by definition $\bar{R}_1 = R_1$). The derivation of given formula can be found in Page 67.5.

In the stationary case, $\bar{R}_t$ can be calculated as $\bar{R}_{t+1} = \bar{R}_t + \frac{1}{t}[R_t - \bar{R}_t]$. To improve the algorithm’s performance in the non-stationary case, we apply $\bar{R}_{t+1} = \bar{R}_t + \alpha[R_t - \bar{R}_t]$ with a constant step size $\alpha$.

Reward $R_t$ is multiplied by multiplier before updating preferences to allow for more flexible reward scaling while keeping the algorithm’s properties.

Parameters
- **state** (SoftmaxState) – Current state of the agent.
- **key** (PRNGKey) – A PRNG key used as the random key.
- **action** (int) – Previously selected action.
- **reward** (float) – Reward collected by the agent after taking the previous action.
- **lr** (float) – Step size.
• **alpha** (*float*) – Exponential recency-weighted average factor (used when $\alpha > 0$).

• **tau** (*float*) – Temperature parameter.

• **multiplier** (*float*) – Multiplier for the reward.

Returns

Updated agent state.

**Return type**

`SoftmaxState`

**static sample**(*state: SoftmaxState, key: Array, tau: float | int) → int*

The policy of the Softmax algorithm is stochastic. The algorithm draws the next action from the softmax distribution. The probability of selecting action $i$ is calculated as:

$$softmax(H)_i = \frac{\exp(H_i/\tau)}{\sum_{h \in H} \exp(h/\tau)}.$$

**Parameters**

• **state** (*SoftmaxState*) – Current state of the agent.

• **key** (*PRNGKey*) – A PRNG key used as the random key.

• **tau** (*float*) – Temperature parameter.

Returns

Selected action.

**Return type**

`int`

**Thompson sampling**

**class ThompsonSamplingState**(*alpha: Array | ndarray | bool_ | number, beta: Array | ndarray | bool_ | number)*

Bases: `AgentState`, `Mapping`

Container for the state of the Thompson sampling agent.

**alpha**

Number of successful tries for each arm.

Type

`Array`

**beta**

Number of failed tries for each arm.

Type

`Array`

**items**() → a set-like object providing a view on D’s items

**keys**() → a set-like object providing a view on D’s keys

**values**() → an object providing a view on D’s values
class ThompsonSampling(n_arms: int, decay: float | int = 0.0)
Bases: BaseAgent

Contextual Bernoulli Thompson sampling agent with the exponential smoothing. The implementation is inspired by the work of Krotov et al.\(^7\). Thompson sampling is based on a beta distribution with parameters related to the number of successful and failed attempts. Higher values of the parameters decrease the entropy of the distribution while changing the ratio of the parameters shifts the expected value.

Parameters

- **n_arms** (int) – Number of bandit arms. \(N \in \mathbb{N}_+\).
- **decay** (float, default=0.0) – Decay rate. If equal to zero, smoothing is not applied. \(w \geq 0\).

References

**static parameter_space()** → Dict

Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

**property update_observation_space:** Dict

Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

**property sample_observation_space:** Dict

Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

**property action_space:** Discrete

Action space of the agent in Gymnasium format.

**static init(key: Array, n_arms: int) → ThompsonSamplingState**

Creates and initializes an instance of the Thompson sampling agent for n_arms arms. The \(\alpha\) and \(\beta\) vectors are set to zero to create a non-informative prior distribution. The last_decay is also set to zero.

Parameters

- **key** (PRNGKey) – A PRNG key used as the random key.
- **n_arms** (int) – Number of bandit arms.

Returns

Initial state of the Thompson sampling agent.

Return type

*ThompsonSamplingState*


Thompson sampling can be adjusted to non-stationary environments by exponential smoothing of values of vectors \(\alpha\) and \(\beta\) which increases the entropy of a distribution over time. Given a result of trial \(s\), we apply the following equations for each action \(a\):

\[\begin{align*}
\alpha^{(a)}_{t+1} &= (1 - \text{decay}) \alpha^{(a)}_t + \text{decay} \cdot \text{trial}\_outcome^{(a)}_t \\
\beta^{(a)}_{t+1} &= (1 - \text{decay}) \beta^{(a)}_t + \text{decay} \cdot \text{trial}\_outcome^{(a)}_t \\
\end{align*}\]

\(^7\) Alexander Krotov, Anton Kiryanov and Evgeny Khorov. 2020. Rate Control With Spatial Reuse for Wi-Fi 6 Dense Deployments. IEEE Access. 8. 168898-168909.
where $\Delta t$ is the time elapsed since the last action selection and $w$ is the decay rate.

**Parameters**

- `state (ThompsonSamplingState)` – Current state of the agent.
- `key (PRNGKey)` – A PRNG key used as the random key.
- `action (int)` – Previously selected action.
- `n_successful (int)` – Number of successful tries.
- `n_failed (int)` – Number of failed tries.
- `delta_time (float)` – Time elapsed since the last action selection.
- `decay (float)` – Decay rate.

**Returns**

Updated agent state.

**Return type**

`ThompsonSamplingState`

**static sample**

```python
static sample(state: ThompsonSamplingState, key: Array, context: Array | ndarray | bool | number) -> int
```

The Thompson sampling policy is stochastic. The algorithm draws $q_a$ from the distribution $\text{Beta}(1 + \alpha(a), 1 + \beta(a))$ for each arm $a$. The next action is selected as

$$A = \text{argmax}_{a \in A} q_a r_a,$$

where $r_a$ is contextual information for the arm $a$, and $A$ is a set of all actions.

**Parameters**

- `state (ThompsonSamplingState)` – Current state of the agent.
- `key (PRNGKey)` – A PRNG key used as the random key.
- `context (Array)` – One-dimensional array of features for each arm.

**Returns**

Selected action.

**Return type**

`int`

**Normal Thompson sampling**

**class NormalThompsonSamplingState**

```python
class NormalThompsonSamplingState(alpha: Array | ndarray | bool | number, beta: Array | ndarray | bool | number, lam: Array | ndarray | bool | number, mu: Array | ndarray | bool | number)
```

Bases: `AgentState`, `Mapping`

Container for the state of the normal Thompson sampling agent.

**alpha**

The concentration parameter of the inverse-gamma distribution.

**Type**

Array
beta
The scale parameter of the inverse-gamma distribution.
Type
Array

lam
The number of observations.
Type
Array

mu
The mean of the normal distribution.
Type
Array

items() → a set-like object providing a view on D's items
keys() → a set-like object providing a view on D's keys
values() → an object providing a view on D's values

class NormalThompsonSampling(n_arms: int, alpha: float | int, beta: float | int, lam: float | int, mu: float | int)

Bases: BaseAgent

Normal Thompson sampling agent[10]. The normal-inverse-gamma distribution is a conjugate prior for the normal distribution with unknown mean and variance. The parameters of the distribution are updated after each observation. The mean of the normal distribution is sampled from the normal-inverse-gamma distribution and the action with the highest expected value is selected.

Parameters
• n_arms (int) – Number of bandit arms. \( N \in \mathbb{N}_+ \).
• alpha (float) – See also NormalThompsonSamplingState for interpretation. \( \alpha > 0 \).
• beta (float) – See also NormalThompsonSamplingState for interpretation. \( \beta > 0 \).
• lam (float) – See also NormalThompsonSamplingState for interpretation. \( \lambda > 0 \).
• mu (float) – See also NormalThompsonSamplingState for interpretation. \( \mu \in \mathbb{R} \).

References

static parameter_space() → Dict
Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be gym.spaces.Dict or None. If None, the user must provide all parameters manually.

property update_observation_space: Dict
Observation space of the update method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property sample_observation_space: Dict
  Observation space of the sample method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If None, the user must provide all parameters manually.

property action_space: Space
  Action space of the agent in Gymnasium format.

  Creates and initializes an instance of the normal Thompson sampling agent for n_arms arms and the given initial parameters for the prior distribution.

Parameters
• key (PRNGKey) – A PRNG key used as the random key.
• n_arms (int) – Number of bandit arms.
• alpha (float) – See also NormalThompsonSamplingState for interpretation.
• beta (float) – See also NormalThompsonSamplingState for interpretation.
• lam (float) – See also NormalThompsonSamplingState for interpretation.
• mu (float) – See also NormalThompsonSamplingState for interpretation.

Returns
Initial state of the normal Thompson sampling agent.

Return type
NormalThompsonSamplingState

  Normal Thompson sampling update according to \[ Page 76, 10. \]

\[
\alpha_{t+1}(a) = \alpha_t(a) + \frac{1}{2}
\]
\[
\beta_{t+1}(a) = \beta_t(a) + \frac{\lambda_t(a)(r_t(a) - \mu_t(a))^2}{2(\lambda_t(a) + 1)}
\]
\[
\lambda_{t+1}(a) = \lambda_t(a) + \lambda
\]
\[
\mu_{t+1}(a) = \frac{\mu_t(a)\lambda_t(a) + r_t(a)}{\lambda_t(a) + 1}
\]

Parameters
• state (NormalThompsonSamplingState) – Current state of the agent.
• key (PRNGKey) – A PRNG key used as the random key.
• action (int) – Previously selected action.
• reward (Float) – Reward obtained upon execution of action.

Returns
Updated agent state.

Return type
NormalThompsonSamplingState
**static inverse_gamma**

**(key: Array, concentration: Array | ndarray | bool_ | number, scale: Array | ndarray | bool_ | number) \rightarrow Array | ndarray | bool_ | number**

Samples from the inverse gamma distribution. Implementation is based on the gamma distribution and the following dependence:

**Parameters**

- **key** (*PRNGKey*) – A PRNG key used as the random key.
- **concentration** (*Array*) – The concentration parameter of the inverse-gamma distribution.
- **scale** (*Array*) – The scale parameter of the inverse-gamma distribution.

**Returns**

Sampled values from the inverse gamma distribution.

**Return type**

`Array` | `ndarray` | `bool_` | `number`

**static sample**

**(state: NormalThompsonSamplingState, key: Array) \rightarrow int**

The normal Thompson sampling policy is stochastic. The algorithm draws $q_a$ from the distribution `Normal(\mu(a), \text{scale}(a)/\sqrt{X(a)})` for each arm $a$ where $\text{scale}(a)$ is sampled from the inverse gamma distribution with parameters $\alpha(a)$ and $\beta(a)$. The next action is selected as $A = \arg\max_{a \in A} q_a$, where $A$ is a set of all actions.

**Parameters**

- **state** (*NormalThompsonSamplingState*) – Current state of the agent.
- **key** (*PRNGKey*) – A PRNG key used as the random key.

**Returns**

Selected action.

**Return type**

`int`

**Log-normal Thompson sampling**

**class LogNormalThompsonSampling**(n_arms: int, alpha: float | int, beta: float | int, lam: float | int, mu: float | int)

**Bases:** `NormalThompsonSampling`

Log-normal Thompson sampling agent. This algorithm is designed to handle positive rewards by transforming them into the log-space. For more details, refer to the documentation on `NormalThompsonSampling`.

**static update**

**(state: NormalThompsonSamplingState, key: Array, action: int, reward: float | int) \rightarrow NormalThompsonSamplingState**

Log-normal Thompson sampling update. The update is analogous to the one in `NormalThompsonSampling` except that the reward is transformed into the log-space.

**Parameters**

- **state** (*NormalThompsonSamplingState*) – Current state of the agent.
- **key** (*PRNGKey*) – A PRNG key used as the random key.
- **action** (*int*) – Previously selected action.
- **reward** *(Float)* – Reward obtained upon execution of action.

**Returns**
Updated agent state.

**Return type**
*NormalThompsonSamplingState*

**static sample** *(state: NormalThompsonSamplingState, key: Array) → int*
Sampling actions is analogous to the one in *NormalThompsonSampling* except that the expected value of the log-normal distribution is computed instead of the expected value of the normal distribution.

**Parameters**
- **state** *(NormalThompsonSamplingState)* – Current state of the agent.
- **key** *(PRNGKey)* – A PRNG key used as the random key.

**Returns**
Selected action.

**Return type**
*int*

### Upper confidence bound (UCB)

**class** UCBState *(R: Array | ndarray | bool_ | number, N: Array | ndarray | bool_ | number)*

**Bases:** *AgentState, Mapping*

Container for the state of the UCB agent.

**R**
Sum of the rewards obtained for each arm.

**Type**
*Array*

**N**
Number of tries for each arm.

**Type**
*Array*

**items** *()* → a set-like object providing a view on D’s items

**keys** *()* → a set-like object providing a view on D’s keys

**values** *()* → an object providing a view on D’s values

**class** UCB *(n_arms: int, c: float | int, gamma: float | int = 1.0)*

**Bases:** *BaseAgent*

UCB agent with optional discounting. The main idea behind this algorithm is to introduce a preference factor in its policy, so that the selection of the next action is based on both the current estimation of the action-value function and the uncertainty of this estimation.

**Parameters**
- **n_arms** *(int)* – Number of bandit arms. \( N \in \mathbb{N}_+ \).
- **c** *(float)* – Degree of exploration. \( c \geq 0 \).
• \( \texttt{gamma(float, default=1.0)} \) – If less than one, a discounted UCB algorithm\(^8\) is used. \( \gamma \in (0, 1] \).

References

\[ \text{static parameter_space()} \to \text{Dict} \]

Parameters of the agent constructor in Gymnasium format. Type of returned value is required to be \texttt{gym.spaces.Dict} or \texttt{None}. If \texttt{None}, the user must provide all parameters manually.

\[ \text{property update_observation_space: Dict} \]

Observation space of the \texttt{update} method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If \texttt{None}, the user must provide all parameters manually.

\[ \text{property sample_observation_space: Dict} \]

Observation space of the \texttt{sample} method in Gymnasium format. Allows to infer missing observations using an extensions and easily export the agent to TensorFlow Lite format. If \texttt{None}, the user must provide all parameters manually.

\[ \text{property action_space: Space} \]

Action space of the agent in Gymnasium format.

\[ \text{static init(key: Array, n_arms: int)} \to \text{UCBState} \]

Creates and initializes instance of the UCB agent for \texttt{n_arms} arms. The sum of the rewards is set to zero and the number of tries is set to one for each arm.

**Parameters**

- \texttt{key(PRNGKey)} – A PRNG key used as the random key.
- \texttt{n_arms(int)} – Number of bandit arms.

**Returns**

Initial state of the UCB agent.

**Return type**

\texttt{UCBState}

\[ \text{static update(state: UCBState, key: Array, action: int, reward: float|int, gamma: float|int)} \to \text{UCBState} \]

In the stationary case, the sum of the rewards for a given arm is increased by reward \( r \) obtained after step \( t \) and the number of tries for the corresponding arm is incremented. In the non-stationary case, the update follows the equations

**Parameters**

- \texttt{state(UCBState)} – Current state of agent.
- \texttt{key(PRNGKey)} – A PRNG key used as the random key.
- \texttt{action(int)} –Previously selected action.
- \texttt{reward(float)} – Reward collected by the agent after taking the previous action.
- \texttt{gamma(float)} – Discount factor.

Returns
Updated agent state.

Return type
UCBState

```python
static sample(state: UCBState, key: Array, c: float | int) -> int
```

UCB agent follows the policy

\[
A = \arg\max_{a \in A} \left[ Q(a) + c \sqrt{\frac{\ln \left( \sum_{a' \in A} N(a') \right)}{N(a)}} \right].
\]

where \( A \) is a set of all actions and \( Q \) is calculated as \( Q(a) = R(a) / N(a) \). The second component of the sum represents a sort of upper bound on the value of \( Q \), where \( c \) behaves like a confidence interval and the square root - like an approximation of the \( Q \) function estimation uncertainty. Note that the UCB policy is deterministic (apart from choosing between several optimal actions).

Parameters
- **state** (UCBState) – Current state of the agent.
- **key** (PRNGKey) – A PRNG key used as the random key.
- **c** (float) – Degree of exploration.

Returns
Selected action.

Return type
int

### 4.2.8 Extensions

This module is a set of extensions. You can either choose one of our built-in extensions or implement your extension with the help of the [Custom extensions guide](#).

**BaseExt**

class BaseExt

Container for domain-specific knowledge and functions for a given environment. Provides the transformation from the raw observations to the agent update and sample spaces. Stores the default argument values for agent initialization.

**abstract property observation_space**: Space

Basic observations of the environment in Gymnasium format.

```python
def get_agent_params(agent_type: type | None = None, agent_parameter_space: Dict | None = None, user_parameters: dict[str, any] | None = None) -> dict[str, any]
```

Composes agent initialization arguments from values passed by the user and default values stored in the parameter functions. Returns a dictionary with the parameters matching the agent parameters space.

Parameters
- **agent_type** (type, optional) – Type of the selected agent.
- **agent_parameter_space** (gym.spaces.Dict, optional) – Parameters required by the agents’ constructor in Gymnasium format.
• **user_parameters** *(dict, optional)* – Parameters provided by the user.

**Returns**
Dictionary with the initialization parameters for the agent.

**Return type**
dict

**setup_transformations** *(agent_update_space: Space | None = None, agent_sample_space: Space | None = None) → None*

Creates functions that transform raw observations and values provided by the observation functions to the agent update and sample spaces.

**Parameters**

- **agent_update_space** *(gym.spaces.Space, optional)* – Observations required by the agent update function in Gymnasium format.
- **agent_sample_space** *(gym.spaces.Space, optional)* – Observations required by the agent sample function in Gymnasium format.

**transform** *(*args, action: any | None = None, **kwargs) → tuple[any, any]*

Transforms raw observations and values provided by the observation functions to the agent observation and sample spaces. Provides the last action selected by the agent if it is required by the agent.

**Parameters**

- ***args** *(tuple)* – Environment observations.
- **action** *(any)* – The last action selected by the agent.
- ****kwargs** *(dict)* – Environment observations.

**Returns**
Agent update and sample observations.

**Return type**
tuple[any, any]

**BasicMab**

**class BasicMab** *(n_arms: int)*

**Bases:** BaseExt

Basic multi-armed bandit (MAB) extension for Reinforced-lib. This extension can be used with MAB algorithms which do not require any additional information about the environment apart from the number of arms.

**Gymnasium**

**class Gymnasium** *(env_id: str)*

**Bases:** BaseExt

Gymnasium extension. Simplifies interaction of RL agents with the Gymnasium environments by providing the environment state, reward, terminal flag, and shapes of the observation and action spaces.

**Parameters**

- **env_id** *(str)* – Name of the Gymnasium environment.

---

1 Gymnasium [https://gymnasium.farama.org](https://gymnasium.farama.org)
References

Extension utils

class ObservationInfo(name: str, type: Space)
    Description of the observation function that provides one of the values from the agent observation space.
    name
        Name of the provided observation.
        Type
            str
    type
        Type of the provided value in Gymnasium format.
        Type
            gym.spaces.Space
    name: str
        Alias for field number 0
    type: Space
        Alias for field number 1

class ParameterInfo(name: str, type: Space)
    Description of the parameter function that provides one of the parameters of the agent constructor.
    name
        Name of the provided parameter.
        Type
            str
    type
        Type of the provided parameter in Gymnasium format.
        Type
            gym.spaces.Space
    name: str
        Alias for field number 0
    type: Space
        Alias for field number 1

observation(observation_name: str | None = None, observation_type: Space | None = None) → Callable
    Decorator used to annotate the observation functions.

    Parameters
    • observation_name (str, optional) – Name of the provided observation.
    • observation_type (gym.spaces.Space, optional) – Type of the provided value in Gymnasium format.

    Returns
    Function that returns the appropriate observation.

    Return type
    Callable
parameter\( (\text{parameter\_name: str | None = None, parameter\_type: Space | None = None} \to \text{Callable}) \)

Decorator used to annotate the parameter functions.

Parameters

- **parameter\_name** \( (\text{str, optional}) \) – Name of the provided parameter.
- **parameter\_type** \( (\text{gym.spaces.Space, optional}) \) – Type of the provided parameter in Gymnasium format.

Returns

Function that returns the appropriate parameter.

Return type

Callable

test\_box\( (a: \text{Space}, b: \text{Box}) \to \text{bool} \)

Tests if the space \( a \) is identical to the gym.space.Box space \( b \).

Parameters

- \( a \) \( (\text{gym.spaces.Space}) \) – Space \( a \).
- \( b \) \( (\text{gym.spaces.Box}) \) – Box space \( b \).

Returns

Result of the comparison.

Return type

bool

test\_discrete\( (a: \text{Space}, b: \text{Discrete}) \to \text{bool} \)

Tests if the space \( a \) is identical to the gym.space.Discrete space \( b \).

Parameters

- \( a \) \( (\text{gym.spaces.Space}) \) – Space \( a \).
- \( b \) \( (\text{gym.spaces.Discrete}) \) – Discrete space \( b \).

Returns

Result of the comparison.

Return type

bool

test\_multi\_binary\( (a: \text{Space}, b: \text{MultiBinary}) \to \text{bool} \)

Tests if the space \( a \) is identical to the gym.space.MultiBinary space \( b \).

Parameters

- \( a \) \( (\text{gym.spaces.Space}) \) – Space \( a \).
- \( b \) \( (\text{gym.spaces.MultiBinary}) \) – MultiBinary space \( b \).

Returns

Result of the comparison.

Return type

bool

test\_multi\_discrete\( (a: \text{Space}, b: \text{MultiDiscrete}) \to \text{bool} \)

Tests if the space \( a \) is identical to the gym.space.MultiDiscrete space \( b \).

Parameters
• a (`gym.spaces.Space`) – Space a.
• b (`gym.spaces.MultiDiscrete`) – MultiDiscrete space b.

**Returns**
Result of the comparison.

**Return type**
bool

**test_sequence**(*a: Space, b: Sequence*) → bool
Tests if the space a is identical to the gym.space.Sequence space b.

**Parameters**

• a (`gym.spaces.Space`) – Space a.
• b (`gym.spaces.Sequence`) – Sequence space b.

**Returns**
Result of the comparison.

**Return type**
bool

**test_space**(*a: Space, b: Space*) → bool
Tests if the space a is identical to the space b.

**Parameters**

• a (`gym.spaces.Space`) – Space a.
• b (`gym.spaces.Space`) – Space b.

**Returns**
Result of the comparison.

**Return type**
bool

### 4.2.9 Loggers module

This module is a set of loggers. You can either choose one of our built-in loggers or implement your logger with the help of the Custom loggers guide.

**LogsObserver**

**class LogsObserver**

Class responsible for managing singleton instances of the loggers, initialization and finalization of the loggers, and passing the logged values to the appropriate loggers and their methods.

**add_logger**(*source: tuple[[str, SourceType] | str] | None, logger_type: type, logger_params: dict[str, any]*) → None

Initializes a singleton instance of the logger and connects a given source with that logger.

**Parameters**

• `source` (Source) – Source to connect.
• `logger_type` (type) – Type of the selected logger.
• **logger_params** *(dict)* – Parameters of the selected logger.

**init_loggers()**
Initializes all loggers by calling their *init* method.

**finish_loggers()**
Finalizes the work of all loggers by calling their *finish* method.

**update_observations**(observations: *any*) → *None*
Passes new observations to the loggers.

  Parameters
  
  observations *(dict or any)* – Observations received by the agent.

**update_agent_state**(agent_state: *BaseAgent*) → *None*
Passes the agent state to the loggers.

  Parameters
  
  agent_state *(BaseAgent)* – Current agent state.

**update_metrics**(metric: *any*, metric_name: *str*) → *None*
Passes metrics to loggers.

  Parameters
  
  • metric *(any)* – Metric value.
  
  • metric_name *(str)* – Name of the metric.

**update_custom**(value: *any*, name: *str*) → *None*
Passes values provided by the user to the loggers.

  Parameters
  
  • value *(any)* – Value to log.
  
  • name *(str)* – Name of the value.

**BaseLogger**

class **BaseLogger**(**kwargs**)
Base interface for loggers.

**init**(sources: *list[Source]|None]*) → *None*
Initializes the logger given the list of all sources defined by the user.

  Parameters
  
  sources *(list[Source])* – List containing the sources to log.

**finish()** → *None*
Finalizes the loggers work (e.g., closes file or shows plots).

**log_scalar**(source: *tuple|str*, SourceType|str|None] | *int*, custom: *bool*) → *None*
Method of the logger interface used for logging scalar values.

  Parameters
  
  • source *(Source)* – Source of the logged value.
  
  • value *(float)* – Scalar to log.
  
  • custom *(bool)* – Whether the source is a custom source.
log_array(source: tuple[str, SourceType] | str | None, value: Array | ndarray | bool_ | number, custom: bool) → None

Method of the logger interface used for logging one-dimensional arrays.

Parameters

- **source** ([Source]) – Source of the logged value.
- **value** (Array) – Array to log.
- **custom** (bool) – Whether the source is a custom source.

log_dict(source: tuple[str, SourceType] | str | None, value: dict, custom: bool) → None

Method of the logger interface used for logging dictionaries.

Parameters

- **source** ([Source]) – Source of the logged value.
- **value** (dict) – Dictionary to log.
- **custom** (bool) – Whether the source is a custom source.

log_other(source: tuple[str, SourceType] | str | None, value: any, custom: bool) → None

Method of the logger interface used for logging other values.

Parameters

- **source** ([Source]) – Source of the logged value.
- **value** (any) – Value of any type to log.
- **custom** (bool) – Whether the source is a custom source.

static source_to_name(source: tuple[str, SourceType] | str | None) → str

Returns a full name of the source. If source is a string itself, returns that string. Otherwise, it returns a string in the format “name-sourcetype” (e.g., “action-metric”).

Parameters

- **source** ([Source]) – Source of the logged value.

Returns

Name of the source.

Return type

str

CsvLogger

class CsvLogger(csv_path: str | None = None, **kwargs)

Bases: BaseLogger

Logger that saves values in CSV format. It saves the logged values to the CSV file when the experiment is finished. CsvLogger synchronizes the logged values in time. It means that if the same source is logged twice in a row, the step number will be incremented for all columns and the logger will move to the next row.

Parameters

- **csv_path** (str, default=“~/rlib-logs-[date]-[time].csv”) – Path to the output file.

finish() → None

Saves the logged values to the CSV file.
log_scalar(source: tuple[str, SourceType] | str | None, value: float | int, *_) → None

Logs a scalar as a standard value in a column.

Parameters

- source (Source) – Source of the logged value.
- value (float) – Scalar to log.

log_array(source: tuple[str, SourceType] | str | None, value: Array | ndarray | bool_ | number, *_) → None

Logs an array as a JSON string.

Parameters

- source (Source) – Source of the logged value.
- value (Array) – Array to log.

log_dict(source: tuple[str, SourceType] | str | None, value: dict, *_) → None

Logs a dictionary as a JSON string.

Parameters

- source (Source) – Source of the logged value.
- value (dict) – Dictionary to log.

log_other(source: tuple[str, SourceType] | str | None, value: any, *_) → None

Logs an object as a JSON string.

Parameters

- source (Source) – Source of the logged value.
- value (any) – Value of any type to log.

StdoutLogger

class StdoutLogger(**kwargs)

Bases: BaseLogger

Logger that writes values to the standard output.

finish() → None

Prints the last row if there are any unprinted values left.

log_scalar(source: tuple[str, SourceType] | str | None, value: float | int, custom: bool) → None

Logs a scalar as the standard value.

Parameters

- source (Source) – Source of the logged value.
- value (float) – Scalar to log.
- custom (bool) – Whether the source is a custom source.

log_array(source: tuple[str, SourceType] | str | None, value: Array | ndarray | bool_ | number, custom: bool) → None

Logs an array as a JSON string.

Parameters

- source (Source) – Source of the logged value.
- **value** (*Array*) – Array to log.
- **custom** (*bool*) – Whether the source is a custom source.

`log_dict(source: tuple[str, SourceType] | str | None, value: dict, custom: bool) → None`

Logs a dictionary as a JSON string.

**Parameters**
- **source** (*Source*) – Source of the logged value.
- **value** (*dict*) – Dictionary to log.
- **custom** (*bool*) – Whether the source is a custom source.

`log_other(source: tuple[str, SourceType] | str | None, value: any, custom: bool) → None`

Logs an object as a JSON string.

**Parameters**
- **source** (*Source*) – Source of the logged value.
- **value** (*any*) – Value of any type to log.
- **custom** (*bool*) – Whether the source is a custom source.

---

### PlotsLogger

**class** **PlotsLogger**

```
class PlotsLogger(plots_dir: str | None = None, plots_ext: str = 'svg', plots_smoothing: float | int = 0.6, plots_scatter: bool = False, plots_sync_steps: bool = False, **kwargs)
```

**Bases:** `BaseLogger`

Logger that presents and saves values as matplotlib plots. Offers smoothing of the curve, scatter plots, and multiple curves in a single chart (while logging arrays). **PlotsLogger** is able to synchronize the logged values in time. This means that if the same source is logged less often than other sources, the step will be increased accordingly to maintain the appropriate spacing between the values on the x-axis.

**Parameters**
- **plots_dir** (*str*, `default=``~``") – Output directory for the plots.
- **plots_ext** (*str*, `default=``svg``") – Extension of the saved plots.
- **plots_smoothing** (*float*, `default=0.6`) – Weight of the exponential moving average (EMA/EWMA)¹ used for smoothing. \( \alpha \in [0, 1) \).
- **plots_scatter** (*bool*, `default=False`) – Set to True if you want to generate a scatter plot instead of a line plot. `plots_smoothing` parameter does not apply to the scatter plots.
- **plots_sync_steps** (*bool*, `default=False`) – Set to True if you want to synchronize the logged values in time.

¹ [https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average](https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average)
References

**finish() → None**
Shows the generated plots and saves them to the output directory with the specified extension (the names of the files follow the pattern “rlib-plot-[source]-[date]-[time].[ext]”).

**log_scalar**(source: tuple[str, SourceType] | str | None, value: float | int, *) → None
Adds a given scalar to the plot values.

**Parameters**
- **source** (Source) – Source of the logged value.
- **value** (float) – Scalar to log.

**log_array**(source: tuple[str, SourceType] | str | None, value: Array | ndarray | bool_ | number, *) → None
Log values from an array to the same plot. Creates multiple line plots for each value in the array.

**Parameters**
- **source** (Source) – Source of the logged value.
- **value** (Array) – Array to log.

**TensorboardLogger**

**class TensorboardLogger**(tb_log_dir: str | None = None, tb_comet_config: dict[str, any] | None = None, tb_sync_steps: bool = False, **)**

Bases: **BaseLogger**

Logger that saves values in TensorBoard\(^2\) format. Offers a possibility to log to Comet\(^3\). **TensorboardLogger** synchronizes the logged values in time. This means that if the same source is logged less often than other sources, the step will be increased accordingly to maintain the appropriate spacing between the values on the x-axis.

**Parameters**
- **tb_log_dir** (str, optional) – Path to the output directory. If None, the default directory is used.
- **tb_comet_config** (dict, optional) – Configuration for the Comet logger. If None, the logger is disabled.
- **tb_sync_steps** (bool, default=False) – Set to True if you want to synchronize the logged values in time.

References

**finish() → None**
Closes the summary writer.

**log_scalar**(source: tuple[str, SourceType] | str | None, value: float | int, *) → None
Adds a given scalar to the summary writer.

**Parameters**
- **source** (Source) – Source of the logged value.

\(^2\) TensorBoard. https://www.tensorflow.org/tensorboard

\(^3\) Comet. https://www.comet.ml
• value (float) – Scalar to log.

log_array(source: tuple[str, SourceType] | str | None, value: Array | ndarray | bool_ | number, *) → None
Log values from an array to the same plot. Creates multiple line plots for each value in the array.

Parameters
• source (Source) – Source of the logged value.
• value (Array) – Array to log.

log_dict(source: tuple[str, SourceType] | str | None, value: dict, *) → None
Logs a dictionary as a JSON string.

Parameters
• source (Source) – Source of the logged value.
• value (dict) – Dictionary to log.

log_other(source: tuple[str, SourceType] | str | None, value: any, *) → None
Logs an object as a JSON string.

Parameters
• source (Source) – Source of the logged value.
• value (any) – Value of any type to log.

WeightsAndBiasesLogger

class WeightsAndBiasesLogger(wandb_sync_steps: bool = False, wandb_kwargs: dict | None = None, **kwargs):

Bases: BaseLogger

Logger that saves values to Weights & Biases platform. WeightsAndBiasesLogger synchronizes the logged values in time. This means that if the same source is logged less often than other sources, the step will be increased accordingly to maintain the appropriate spacing between the values on the x-axis.

Note: to use this logger, you need to log into W&B before running the script. The necessary steps are described in the official documentation.

Parameters
• wandb_sync_steps (bool, default=False) – Set to True if you want to synchronize the logged values in time.
• wandb_kwargs (dict, optional) – Additional keyword arguments passed to wandb.init function.

---

4 Weights & Biases. https://docs.wandb.ai/
References

finish() → None
Finishes the W&B run.

log_scalar(source: tuple[str, SourceType] | str | None, value: float | int, _) → None
Logs a scalar value to the W&B logger.

Parameters

• source (Source) – Source of the logged value.
• value (float) – Scalar to log.

log_array(source: tuple[str, SourceType] | str | None, value: Array | ndarray | bool_ | number, _) → None
Logs an array to the W&B logger.

Parameters

• source (Source) – Source of the logged value.
• value (Array) – Array to log.

log_dict(source: tuple[str, SourceType] | str | None, value: dict, _) → None
Logs a dictionary to the W&B logger.

Parameters

• source (Source) – Source of the logged value.
• value (dict) – Dictionary to log.

log_other(source: tuple[str, SourceType] | str | None, value: any, _) → None
Logs an object to the W&B logger.

Parameters

• source (Source) – Source of the logged value.
• value (any) – Value of any type to log.

4.2.10 Utils

This module contains a collection of utility functions that are used throughout the library.

JAX

Performs a gradient step on the objective with respect to grad_loss_fn function. grad_loss_fn should return tuple of (loss, aux) where loss is the value to be minimized and aux is auxiliary value to be returned (can be None).

Parameters

• objective (any) – Objective to be optimized.
• loss_params (tuple) – Parameters to pass to loss_fn.
• **opt_state** (*optax.OptState*) – Optimizer state.

• **optimizer** (*optax.GradientTransformation*) – Optimizer to use for gradient step.

• **loss_fn** (*Callable*) – Function that returns the loss to be minimized. Can return additional values as well.

**Returns**
- **out** – Tuple containing the updated objective and optimizer state, as well as the loss value.

**Return type**
tuple[any, any, optax.OptState, Scalar]

**init** *(model: Module, key: PRNGKey, *x: Any) → tuple[dict, dict]*)

Initializes the flax model.

**Parameters**
- **model** (*nn.Module*) – Model to be initialized.
- **key** (*PRNGKey*) – A PRNG key used as the random key.
- **x** (any) – Input to the model.

**Returns**
Tuple containing the parameters and the state of the model.

**Return type**
tuple[dict, dict]

**forward** *(model: Module, params: dict, state: dict, key: PRNGKey, *x: Any) → tuple[Array | ndarray | bool_ | number, dict]*)

Forward pass through the flax model. **Note:** by default, the model is provided with two random key streams: one for the dropout layers and one for the user. This is done to ensure that the dropout is always initialized with the same random key, and that the user can use the custom key for any other purpose. The custom key is available in the model by calling `self.make_rng('rlib')`.

**Parameters**
- **model** (*nn.Module*) – Model to be used for forward pass.
- **params** (dict) – Parameters of the model.
- **state** (dict) – State of the network.
- **key** (*PRNGKey*) – A PRNG key used as the random key.
- **x** (any) – Input to the model.

**Returns**
Tuple containing the output of the model and the updated state.

**Return type**
tuple[Array, dict]
Experience replay

class ExperienceReplay(init: Callable, append: Callable, sample: Callable, is_ready: Callable)
Container for experience replay buffer functions.

    init
    Function that initializes the replay buffer.
        Type
        Callable

    append
    Function that appends a new values to the replay buffer.
        Type
        Callable

    sample
    Function that samples a batch from the replay buffer.
        Type
        Callable

    is_ready
    Function that checks if the replay buffer is ready to be sampled.
        Type
        Callable

    items() → a set-like object providing a view on D's items
    keys() → a set-like object providing a view on D's keys
    values() → an object providing a view on D's values

class ReplayBuffer(states: Array | ndarray | bool_ | number, actions: Array | ndarray | bool_ | number, rewards: Array | ndarray | bool_ | number, terminals: Array | ndarray | bool_ | number, next_states: Array | ndarray | bool_ | number, size: int, ptr: int)
Dataclass containing the replay buffer values. The replay buffer is implemented as a circular buffer.

    states
    Array containing the states.
        Type
        Array

    actions
    Array containing the actions.
        Type
        Array

    rewards
    Array containing the rewards.
        Type
        Array
**terminals**
Array containing the terminal flags.

    Type
    Array

**next_states**
Array containing the next states.

    Type
    Array

**size**
Current size of the replay buffer.

    Type
    int

**ptr**
Current pointer of the replay buffer.

    Type
    int

**items()** → a set-like object providing a view on D's items

**keys()** → a set-like object providing a view on D's keys

**values()** → an object providing a view on D's values

### experience_replay(buffer_size: int, batch_size: int, obs_space_shape: Sequence[int | Any], act_space_shape: Sequence[int | Any]) → ExperienceReplay

Experience replay buffer used for off-policy learning. Improves the stability of the learning process by reducing the correlation between the samples and enables an agent to learn from past experiences.

**Parameters**

- **buffer_size** (int) – Maximum size of the replay buffer.
- **batch_size** (int) – Size of the batch to be sampled from the replay buffer.
- **obs_space_shape** (Shape) – Shape of the observation space.
- **act_space_shape** (Shape) – Shape of the action space.

**Returns**

- **out** – Container for experience replay buffer functions.

**Return type**

*ExperienceReplay*
Particle filter

class ParticleFilterState(positions: Array | ndarray | bool_ | number, logit_weights: Array | ndarray | bool_ | number)

Bases: AgentState, Mapping

Container for the state of the particle filter agent.

positions
Positions of the particles.
  Type
  Array

logit_weights
Unnormalized log weights of the particles.
  Type
  Array

items() → a set-like object providing a view on D's items
keys() → a set-like object providing a view on D's keys
values() → an object providing a view on D's values

simple_resample(operands: tuple[ParticleFilterState, Array]) → ParticleFilterState
Samples new particle positions from a categorical distribution with particle weights, then sets all weights equal.

Parameters
operands (tuple[ParticleFilterState, PRNGKey]) – Tuple containing the filter state and a PRNG key.

Returns
Updated filter state.

Return type
ParticleFilterState

effective_sample_size(state: ParticleFilterState, threshold: float | int = 0.5) → bool
Calculates the effective sample size⁹ (ESS). If ESS is smaller than the number of sample times threshold, then a resampling is necessary.

Parameters
• state (ParticleFilterState) – Current state of the filter.
• threshold (float, default=0.5) – Threshold value used to decide if a resampling is necessary. \( thr \in (0, 1) \).

Returns
Information whether a resampling should be performed.

Return type
bool

⁹ https://en.wikipedia.org/wiki/Effective_sample_size#Weighted_samples
**References**

**simple_transition** *(state: ParticleFilterState, key: Array, scale: float | int, *args) → ParticleFilterState*

Performs simple movement of the particle positions according to a normal distribution with $\mu = 0$ and $\sigma = scale$.

**Parameters**

- **state** *(ParticleFilterState)* – Current state of the filter.
- **key** *(PRNGKey)* – A PRNG key used as the random key.
- **scale** *(float)* – Scale of the random movement of particles. $scale > 0$.

**Returns**

Updated filter state.

**Return type**

*ParticleFilterState*

**linear_transition** *(state: ParticleFilterState, key: Array, scale: float | int, delta_time: float | int) → ParticleFilterState*

Performs movement of the particle positions according to a normal distribution with $\mu = 0$ and $\sigma = scale \cdot \Delta t$, where $\Delta t$ is the time elapsed since the last update.

**Parameters**

- **state** *(ParticleFilterState)* – Current state of the filter.
- **key** *(PRNGKey)* – A PRNG key used as the random key.
- **scale** *(float)* – Scale of the random movement of particles. $scale > 0$.
- **delta_time** *(float)* – Time elapsed since the last update.

**Returns**

Updated filter state.

**Return type**

*ParticleFilterState*

**affine_transition** *(state: ParticleFilterState, key: Array, scale: Array | ndarray | bool_ | number, delta_time: float | int) → ParticleFilterState*

Performs movement of the particle positions according to a normal distribution with $\mu = 0$ and $\sigma = scale_0 \cdot \Delta t + scale_1$, where $\Delta t$ is the time elapsed since the last update.

**Parameters**

- **state** *(ParticleFilterState)* – Current state of the filter.
- **key** *(PRNGKey)* – A PRNG key used as the random key.
- **scale** *(Array)* – Scale of the random movement of particles. $scale_0, scale_1 > 0$.
- **delta_time** *(float)* – Time elapsed since the last update.

**Returns**

Updated filter state.

**Return type**

*ParticleFilterState*

Bases: object

Particle filter (sequential Monte Carlo) algorithm estimating the internal environment state given noisy or partial observations.

Parameters

- **initial_distribution_fn** (callable) – Function that samples the initial particle positions; takes two positional arguments:
  - key: a PRNG key used as a random key (PRNGKey).
  - shape: shape of the sample (Shape).

Returns the initial particle positions (Array).

- **positions_shape** (Array) – Shape of the particle positions array.

- **weights_shape** (Array) – Shape of the particle weights array.

- **scale** (Array) – Scale of the random movement of the particles.

- **observation_fn** (callable) – Function that updates particles based on an observation from the environment; takes two positional arguments:
  - state: the state of the filter (ParticleFilterState).
  - observation: an observation from the environment (any).

Returns the updated state of the filter (ParticleFilterState).

- **resample_fn** (callable, default=particle_filter.simple_resample) – Function that performs resampling of the particles; takes one positional argument:
  - operands: a tuple containing the filter state and a PRNG key (tuple[ParticleFilterState, PRNGKey]).

Returns the updated state of the filter (ParticleFilterState).

- **resample_criterion_fn** (callable, default=particle_filter.effective_sample_size) – Function that checks if a resampling is necessary; takes one positional argument:
  - state: the state of the filter (ParticleFilterState).

Returns information whether a resampling should be performed (bool).

- **transition_fn** (callable, default=particle_filter.simple_transition) –
Function that updates the particle positions; takes four positional arguments:

- state: the state of the filter (ParticleFilterState).
- key: a PRNG key used as a random key (PRNGKey).
- scale: scale of the random movement of the particles (Array).
- time: the current time (float).

Returns the updated state of the filter (ParticleFilterState).

static init(key: Array, initial_distribution_fn: Callable, positions_shape: Sequence[int | Any], weights_shape: Sequence[int | Any]) → ParticleFilterState

Creates and initializes an instance of the particle filter.

Parameters

- **key** (PRNGKey) – A PRNG key used as the random key.
- **initial_distribution_fn** (callable) – Function that samples the initial particle positions.
  - key: PRNG key used as a random key (PRNGKey).
  - shape: shape of the sample (Shape).

Returns

Initial state of the Particle Filter.

Return type

ParticleFilterState

static update(state: ParticleFilterState, key: Array, observation_fn: Callable[[ParticleFilterState, any], ParticleFilterState], observation: any, resample_fn: Callable[[tuple[ParticleFilterState, Array], ParticleFilterState], resampleCriterion_fn: Callable[[ParticleFilterState], bool], transition_fn: Callable[[ParticleFilterState, Array, Array | ndarray | bool_ | number | float | int, float | int], ParticleFilterState], delta_time: float | int, scale: Array | ndarray | bool_ | number | float | int) → ParticleFilterState

Updates the state of the filter based on an observation from the environment, then performs resampling (if necessary) and transition of the particles.

Parameters

- **state** (ParticleFilterState) – Current state of the filter.
- **key** (PRNGKey) – A PRNG key used as the random key.
- **observation_fn** (callable) – Function that updates particles based on an observation from the environment; takes two positional arguments:
  - state: the state of the filter (ParticleFilterState).
  - observation: an observation from the environment (any).

Returns the updated state of the filter (ParticleFilterState).

- **observation** (any) – An observation from the environment.
• **resample_fn** *(callable, default=particle_filter.simple_resample)* –
Function that performs resampling of the particles; takes one positional argument:
– operands: a tuple containing the filter state and a PRNG key *(tuple[ParticleFilterState, PRNGKey])*.
Returns the updated state of the filter *(ParticleFilterState)*.

• **resample_criterion_fn** *(callable, default=particle_filter.effective_sample_size)* –
Function that checks if a resampling is necessary; takes one positional argument:
– state: the state of the filter *(ParticleFilterState)*.
Returns information whether a resampling should be performed *(bool)*.

• **transition_fn** *(callable, default=particle_filter.simple_transition)* –
Function that updates the particle positions; takes four positional arguments:
– state: the state of the filter *(ParticleFilterState)*.
– key: a PRNG key used as a random key *(PRNGKey)*.
– scale: scale of the random movement of the particles *(Array)*.
– time: the current time *(float)*.

• **delta_time** *(float)* – Time difference between the current and the previous observation.

• **scale** *(Array)* – Scale of the random movement of the particles.

**Returns**
Updated filter state.

**Return type**
*ParticleFilterState*

**static sample** *(state: ParticleFilterState, key: Array) → Array | ndarray | bool_ | number | float | int*
Samples the estimated environment state from a categorical distribution with particle weights.

**Parameters**

• **state** *(ParticleFilterState)* – Current state of the filter.

• **key** *(PRNGKey)* – A PRNG key used as the random key.

**Returns**
Estimated environment state.

**Return type**
Array
4.2.11 Exceptions

**exception NoAgentError**
Raised when no agent is specified.

**exception NoExtensionError**
Raised when no extension is specified.

**exception IncorrectTypeError**

\[
\text{provided}\_\text{type}: \text{type} | \text{None} = \text{None}, \text{expected}\_\text{module}: \text{str} | \text{None} = \text{None}
\]
Raised when provided class type is incorrect.

Parameters

- **provided_type** (type, optional) – Type provided by the user.
- **expected_module** (str, optional) – Name of the module that `provided_type` should match.

**exception IncorrectAgentTypeError**

\[
\text{provided}\_\text{type}: \text{type}
\]
Raised when provided agent does not inherit from the BaseAgent class.

Parameters

- **provided_type** (type) – Type provided by the user.

**exception IncorrectExtensionTypeError**

\[
\text{provided}\_\text{type}: \text{type}
\]
Raised when provided extension does not inherit from the BaseExt class.

Parameters

- **provided_type** (type) – Type provided by the user.

**exception IncorrectLoggerTypeError**

\[
\text{provided}\_\text{type}: \text{type}
\]
Raised when provided logger does not inherit from the BaseLogger class.

Parameters

- **provided_type** (type) – Type provided by the user.

**exception ForbiddenOperationError**
Raised when the user performs a forbidden operation.

**exception ForbiddenAgentChangeError**
Raised when the user changes the agent type after the first agent instance has been initialized.

**exception ForbiddenExtensionChangeError**
Raised when the user changes the extension type after the first agent instance has been initialized.

**exception ForbiddenExtensionSetError**
Raised when the user sets the extension type when `no_ext_mode` is enabled.

**exception ForbiddenLoggerSetError**
Raised when the user adds a new logger after the first step has been made.

**exception IncorrectSpaceError**
Raised when an unknown space is provided, for example a custom Gymnasium space.

**exception UnimplementedSpaceError**
Raised when an observation space is required but not implemented.
exception IncompatibleSpacesError(ext_space: Space, agent_space: Space)
Raised when the observation spaces of two different modules are not compatible.

Parameters
- ext_space (gym.spaces.Space) – Observation space of the extension.
- agent_space (gym.spaces.Space) – Observation space of the agent.

exception NoDefaultParameterError(extension_type: type, parameter_name: str, parameter_type: Space)
Raised when the extension does not define a default parameter value for the agent.

Parameters
- extension_type (type) – Type of the used extension.
- parameter_name (str) – Name of the missing parameter.
- parameter_type (gym.spaces.Space) – Type of the missing parameter.

exception UnsupportedLogTypeError(logger_type: type, log_type: type)
Raised when the user logs values that are not supported by the logger.

Parameters
- logger_type (type) – Type of the used logger.
- log_type (type) – Type of the logged value.

exception IncorrectSourceTypeError(provided_type: type)
Raised when the provided source is not a correct source type (i.e., Union[Tuple[str, SourceType], str]).

Parameters
- provided_type (type) – Type provided by the user.

exception UnsupportedCustomLogsError(logger_type: type)
Raised when the user tries to log custom values with a logger that does not support custom logging.

Parameters
- logger_type (type) – Type of the used logger.

4.3 Indices and tables

- genindex
- search
CITING REINFORCED-LIB

To cite this repository, please use the following BibTeX entry for the Reinforced-lib paper:

```bibtex
@article{reinforcedlib2022,  
  author = {Maksymilian Wojnar and Szymon Szott and Krzysztof Rusek and Wojciech Ciezobka},  
  title = {{R}einforced-lib: {R}apid prototyping of reinforcement learning solutions},  
  journal = {SoftwareX},  
  volume = {26},  
  pages = {101706},  
  year = {2024},  
  issn = {2352-7110},  
  doi = {https://doi.org/10.1016/j.softx.2024.101706},  
  url = {https://www.sciencedirect.com/science/article/pii/S2352711024000773}
}
```
reinforced_lib.utils.exceptions, 101
reinforced_lib.utils.experience_replay, 94
reinforced_lib.utils.jax_utils, 92
reinforced_lib.utils.particle_filter, 96
loss_fn() (ExpectedSarsa static method), 60

module
reinforced_lib.utils.exceptions, 101
reinforced_lib.utils.experience_replay, 94
reinforced_lib.utils.jax_utils, 92
reinforced_lib.utils.particle_filter, 96
mu (NormalThompsonSamplingState attribute), 76

N
N (EGreedyState attribute), 67
n (SoftmaxState attribute), 71
N (UCBSate attribute), 79
net_state (DDQNState attribute), 54
net_state (DDQNState attribute), 50
net_state (ExpectedSarsaState attribute), 58
net_state_target (DDQNState attribute), 54
next_states (ReplayBuffer attribute), 95
NoAgentError, 101
NoDefaultParameterError, 102
NoExtensionError, 101
noise (DDPGState attribute), 62
NormalThompsonSampling (class in reinforced_lib.agents.mab.normal_thompson_sampling), 76
NormalThompsonSamplingState (class in reinforced_lib.agents.mab.normal_thompson_sampling), 75

O
observation_space (BaseExt property), 81
observation_space (RLib property), 47
omega (Exp3State attribute), 69
opt_state (DDQNState attribute), 54
opt_state (DDQNState attribute), 50
opt_state (ExpectedSarsaState attribute), 58

P
parameter_space() (BaseAgent static method), 49
parameter_space() (DDPG static method), 64
parameter_space() (DDQN static method), 56
parameter_space() (DDQN static method), 52
parameter_space() (EGreedy static method), 67
parameter_space() (Exp3 static method), 69
parameter_space() (ExpectedSarsa static method), 59
parameter_space() (NormalThompsonSampling static method), 76
parameter_space() (Softmax static method), 71
parameter_space() (ThompsonSampling static method), 74
parameter_space() (UCB static method), 80
params (DDQNState attribute), 54
params (DDQNState attribute), 50
params (ExpectedSarsaState attribute), 58
params_target (DDQNState attribute), 54
ParticleFilter (class in reinforced_lib.utils.particle_filter), 97
ParticleFilterState (class in reinforced_lib.utils.particle_filter), 96
PlotsLogger (class in reinforced_lib.logs.plots_logger), 89
positions (ParticleFilterState attribute), 96
prev_env_state (DDPGState attribute), 62
prev_env_state (DDQNState attribute), 54
prev_env_state (DDQNState attribute), 50
prev_env_state (ExpectedSarsaState attribute), 58
ptr (ReplayBuffer attribute), 95
Q
Q (EGreedyState attribute), 67
q_loss_fn() (DDPG static method), 64
q_net_state (DDPGState attribute), 61
q_net_state_target (DDPGState attribute), 61
q_opt_state (DDPGState attribute), 62
q_params (DDPGState attribute), 61
q_params_target (DDPGState attribute), 61
r (SoftmaxState attribute), 71
R
r (UCBSate attribute), 79
reinforced_lib.utils.exceptions module, 101
reinforced_lib.utils.experience_replay module, 94
reinforced_lib.utils.jax_utils module, 92
reinforced_lib.utils.particle_filter module, 96
replay_buffer (DDPGState attribute), 62
replay_buffer (DDQNState attribute), 54
replay_buffer (DDQNState attribute), 50
replay_buffer (ExpectedSarsaState attribute), 58
ReplayBuffer (class in reinforced_lib.utils.experience_replay), 94
rewards (ReplayBuffer attribute), 94
RLib (class in reinforced_lib.rlib), 46
S
sample (ExperienceReplay attribute), 94
sample() (BaseAgent static method), 49
sample() (DDPG static method), 66
sample() (DDQN static method), 57
sample() (DDQN static method), 53
sample() (EGreedy static method), 68
sample() (Exp3 static method), 70
sample() (ExpectedSarsa static method), 61
sample() (LogNormalThompsonSampling static method), 79
sample() (NormalThompsonSampling static method), 78
sample() (ParticleFilter static method), 100
sample() (RLib method), 47
sample() (Softmax static method), 73
sample() (ThompsonSampling static method), 75
sample() (UCB static method), 81
sample_observation_space (BaseAgent property), 49
sample_observation_space (DDQN property), 64
sample_observation_space (NormalThompsonSampling property), 76
sample_observation_space (Softmax property), 72
sample_observation_space (ThompsonSampling property), 74
sample_observation_space (UCB property), 80
save() (RLib method), 48
set_agent() (RLib method), 46
set_ext() (RLib method), 46
set_loggers() (RLib method), 46
setup_transformations() (BaseExt method), 82
simple_resample() (in module reinforced_lib.utils.particle_filter), 96
simple_transition() (in module reinforced_lib.utils.particle_filter), 97
size (ReplayBuffer attribute), 95
Softmax (class in reinforced_lib.agents.mab.softmax), 71
SoftmaxState (class in reinforced_lib.agents.mab.softmax), 71
source_to_name() (BaseLogger static method), 87
states (ReplayBuffer attribute), 94
StdoutLogger (class in reinforced_lib.logs.stdout_logger), 88

TensorboardLogger (class in reinforced_lib.logs.tensorboard_logger), 90
terminals (ReplayBuffer attribute), 94
ThompsonSampling (class in reinforced_lib.agents.mab.thompson_sampling), 73
ThompsonSamplingState (class in reinforced_lib.agents.mab.thompson_sampling), 73
to_tflite() (RLib method), 49
transform() (BaseExt method), 82

UCB (class in reinforced_lib.agents.mab.ucb), 79
UCBState (class in reinforced_lib.agents.mab.ucb), 79
UnimplementedSpaceError, 101
UnsupportedCustomLogsError, 102
UnsupportedLogTypeError, 102
update() (BaseAgent static method), 49
update() (DDPG static method), 65
update() (DDQN static method), 57
update() (DQN static method), 53
update() (EGreedy static method), 68
update() (Exp3 static method), 70
update() (ExpectedSarsa static method), 60
update() (LogNormalThompsonSampling static method), 78
update() (NormalThompsonSampling static method), 77
update() (ParticleFilter static method), 99
update() (Softmax static method), 72
update() (ThompsonSampling static method), 74
update() (UCB static method), 80
update_agent_state() (LogsObserver method), 86
update_custom() (LogsObserver method), 86
update_metrics() (LogsObserver method), 86
update_observation_space (BaseAgent property), 49
update_observation_space (DDPG property), 64
update_observation_space (DDQN property), 56
update_observation_space (DQN property), 52
update_observation_space (EGreedy property), 67
update_observation_space (Exp3 property), 69
update_observation_space (ExpectedSarsa property), 59
update_observation_space (NormalThompsonSampling property), 76
update_observation_space (Softmax property), 72
update_observation_space (ThompsonSampling property), 74
update_observation_space (UCB property), 80
update_observations() (LogsObserver method), 86

values() (AgentState method), 49
values() (DDPGState method), 49
values() (DDQNSate method), 55
values() (DQNState method), 51
values() (EGreedyState method), 67
values() (Exp3State method), 69
values() (ExpectedSarsaState method), 58
values() (ExperienceReplay method), 94
values() (NormalThompsonSamplingState method), 76
values() (ParticleFilterState method), 96
values() (ReplayBuffer method), 95
values() (SoftmaxState method), 71
values() (ThompsonSamplingState method), 73
values() (UCBState method), 79
WeightsAndBiasesLogger (class in reinforced_lib.logs.wandb_logger), 91