imitation

Center for Human-Compatible AI

Nov 18, 2022
## GETTING STARTED

1 Main Features

1.1 Installation ................................................. 3
1.2 What is imitation? ........................................ 4
1.3 Limitations on Horizon Length ......................... 4
1.4 First Steps ................................................ 5
1.5 Behavioral Cloning (BC) ................................. 8
1.6 Generative Adversarial Imitation Learning (GAIL) .. 11
1.7 Adversarial Inverse Reinforcement Learning (AIRL) ........................................................................ 19
1.8 DAgger ........................................................ 27
1.9 Density-based reward modeling ........................... 36
1.10 Maximum Causal Entropy Inverse Reinforcement Learning (MCE IRL) ........................................... 39
1.11 Preference comparisons ................................... 43
1.12 Train an Agent using Behavior Cloning .......... 48
1.13 Train an Agent using the DAgger Algorithm ...... 50
1.14 Train an Agent using Generative Adversarial Imitation Learning ..................................................... 53
1.15 Train an Agent using Adversarial Inverse Reinforcement Learning .................................................. 57
1.16 Learning a Reward Function using Preference Comparisons ........................................................... 62
1.17 Learning a Reward Function using Preference Comparisons on Atari ............................................. 71
1.18 Generating rollouts .......................................... 77
1.19 Learn a Reward Function using Maximum Conditional Entropy Inverse Reinforcement Learning .... 78
1.20 Learning a Reward Function using Kernel Density ................................................................. 84
1.21 Loading Experts ............................................. 87

2 API Reference .......................... 91

2.1 imitation ..................................................... 91
2.2 Developer Guide ........................................... 220
2.3 Contributing ............................................... 223
2.4 Release Notes .............................................. 226
2.5 License ....................................................... 227
2.6 Index .......................................................... 227

Python Module Index .......................... 229

Index ........................................ 231
Imitation provides clean implementations of imitation and reward learning algorithms, under a unified and user-friendly API. Currently, we have implementations of Behavioral Cloning, DAgger (with synthetic examples), density-based reward modeling, Maximum Causal Entropy Inverse Reinforcement Learning, Adversarial Inverse Reinforcement Learning, Generative Adversarial Imitation Learning, and Deep RL from Human Preferences.

You can find us on GitHub at http://github.com/HumanCompatibleAI/imitation.
CHAPTER ONE

MAIN FEATURES

• Built on and compatible with Stable Baselines 3 (SB3).
• Modular Pytorch implementations of Behavioral Cloning, DAgger, GAIL, and AIRL that can train arbitrary SB3 policies.
• GAIL and AIRL have customizable reward and discriminator networks.
• Scripts to train policies using SB3 and save rollouts from these policies as synthetic “expert” demonstrations.
• Data structures and scripts for loading and storing expert demonstrations.

1.1 Installation

1.1.1 Prerequisites

• Python 3.8+
• (Optional) OpenGL (to render gym environments)
• (Optional) FFmpeg (to encode videos of renders)
• (Optional) MuJoCo (follow instructions to install mujoco_py v1.5 here)

1.1.2 Installation from PyPI

To install the latest PyPI release, simply run:

```bash
pip install imitation
```

1.1.3 Installation from source

Installation from source is useful if you wish to contribute to the development of imitation, or if you need features that have not yet been made available in a stable release:

```bash
git clone http://github.com/HumanCompatibleAI/imitation
cd imitation
pip install -e
```

There are also a number of dependencies used for running tests and building the documentation, which can be installed with:
1.2 What is imitation?

Imitation is an open-source library providing high-quality, reliable and modular implementations of seven reward and imitation learning algorithms, built on modern backends like PyTorch and Stable Baselines. It includes implementations of Behavioral Cloning (BC), DAgger, Generative Adversarial Imitation Learning (GAIL), Adversarial Inverse Reinforcement Learning (AIRL), Reward Learning through Preference Comparisons, Maximum Causal Entropy Inverse Reinforcement Learning (MCE IRL), and Density-based reward modeling. The algorithms follow a consistent interface, making it simple to train and compare a range of algorithms.

A key use case of imitation is as an experimental baseline. Small implementation details in imitation learning algorithms can have significant impacts on performance, which can lead to spurious positive results if a weak experimental baseline is used. To address this challenge, imitation’s algorithms have been carefully benchmarked and compared to prior implementations. The codebase is statically type-checked and over 90% of it is covered by automated tests.

In addition to providing reliable baselines, imitation aims to simplify the process of developing novel reward and imitation learning algorithms. Its implementations are modular: users can freely change the reward or policy network architecture, RL algorithm and optimizer without touching the codebase itself. Algorithms can be extended by subclassing and overriding relevant methods. Imitation also provides utility methods to handle common tasks to support the development of entirely novel algorithms.

Our goal for imitation is to make it easier to use, develop, and compare imitation and reward learning algorithms. The library is in active development, and we welcome contributions and feedback.

Check out our recommended First Steps for an overview of how to use the library. We also have tutorials, such as Train an Agent using Behavior Cloning, that provide detailed examples of specific algorithms. If you are interested in helping develop imitation then we suggest you refer to the Developer Guide as well as more specific guidelines for Contributing.

1.3 Limitations on Horizon Length

1.3.1 Variable Horizon Environments Considered Harmful

Reinforcement learning (RL) algorithms are commonly trained and evaluated in variable horizon environments. In these environments, the episode ends when some termination condition is reached (rather than after a fixed number of steps). This typically corresponds to success, such as reaching the top of the mountain in MountainCar, or to failure, such as the pole falling down in CartPole. A variable horizon will tend to speed up RL training, by increasing the proportion of samples where the agent's actions still have a meaningful impact on the reward, pruning out states that are already a foregone conclusion.

However, termination conditions must be carefully hand-designed for each environment. Their inclusion therefore provides a significant source of information about the reward. Evaluating reward and imitation learning algorithms in variable-horizon environments can therefore be deeply misleading. In fact, reward learning in commonly used variable horizon environments such as MountainCar and CartPole can be solved by learning a single bit: the sign of the reward. Of course, an algorithm being able to learn a single bit predicts very little about its performance in real-world tasks, that do not usually come with such an informative termination condition.

To make matters worse, some algorithms have a strong inductive bias towards a particular sign. Indeed, Figure 5 of Kostrikov et al (2021) shows that GAIL is able to reach a third of expert performance even without seeing any expert demonstrations. Consequently, algorithms that happen to have an inductive bias aligned with the task (e.g. positive reward bias for environments where longer episodes are better) may outperform unbiased algorithms on certain
environments. Conversely, algorithms with a misaligned inductive bias will perform worse than an unbiased algorithm. This may lead to illusory discrepancies between algorithms, or even different implementations of the same algorithm.

Kostrikov et al (2021) introduces a way to correct for this bias. However, this does not solve the problem of information leakage. Rather, it merely ensures that different algorithms are all able to equally exploit the information leak provided by the termination condition.

In light of this issue, we would strongly recommend users evaluate imitation and other reward or imitation learning algorithms only in fixed-horizon environments. This is a common, though unfortunately not ubiquitous, practice in reward learning papers. For example, Christiano et al (2017) use fixed horizon environments because:

Removing variable length episodes leaves the agent with only the information encoded in the environment itself; human feedback provides its only guidance about what it ought to do.

Many environments, like HalfCheetah, are naturally fixed-horizon. Moreover, most variable-horizon tasks can be easily converted into fixed-horizon tasks. Our sister project seals provides fixed-horizon versions of many commonly used MuJoCo continuous control tasks, as well as mitigating other potential pitfalls in reward learning evaluation.

Given the serious issues with evaluation and training in variable horizon tasks, imitation will by default throw an error if training AIRM, GAIL, or DRLHP in variable horizon tasks. If you have read this document and understand the problems that variable horizon tasks can cause but still want to train in variable horizon settings, you can override this safety check by setting allow_variable_horizon=True. Note this check is not applied for BC or DAgger, which operate on individual transitions (not episodes) and so cannot exploit the information leak.

Usage with allow_variable_horizon=True is not officially supported, and we will not optimize imitation algorithms to perform well in this situation, as it would not represent real progress. Examples of situations where setting this flag may nonetheless be appropriate include:

1. Investigating the bias introduced by variable horizon tasks – e.g. comparing variable to fixed horizon tasks.
2. For unit tests to verify algorithms continue to run on variable horizon environments.
3. Where the termination condition is trivial (e.g. has the robot fallen over?) and the target behaviour is complex (e.g. solve a Rubik’s cube). In this case, while the termination condition still helps reward and imitation learning, the problem remains highly non-trivial even with this information side-channel. However, the existence of this side-channel should of course be prominently disclosed.

See this GitHub issue for further discussion.

### 1.3.2 Non-Support for Infinite Length Horizons

At the moment, we do not support infinite-length horizons. Many of the imitation algorithms, especially those relying on RL, do not easily port over to infinite-horizon setups. Similarly, much of the logging and reward calculation logic assumes the existence of a finite horizon. Although we may explore workarounds in the future, this is not a feature that we can currently support.

### 1.4 First Steps

Imitation can be used in two main ways: through its command-line interface (CLI) or Python API. The CLI allows you to quickly train and test algorithms and policies directly from the command line. The Python API provides greater flexibility and extensibility, and allows you to inter-operate with your existing Python environment.
1.4.1 CLI Quickstart

We provide several CLI scripts as front-ends to the algorithms implemented in imitation. These use Sacred for configuration and replicability.

For information on how to configure Sacred CLI options, see the Sacred docs.

```
#!/usr/bin/env bash

# Train PPO agent on pendulum and collect expert demonstrations. Tensorboard logs saved in quickstart/rl/
python -m imitation.scripts.train_rl with pendulum common.fast train.fast rl.fast fast
   --common.log_dir=quickstart/rl/

# Train GAIL from demonstrations. Tensorboard logs saved in output/ (default log directory).
python -m imitation.scripts.train_adversarial gail with pendulum common.fast
   --demonstrations.fast train.fast rl.fast fast demonstrations.rollout_path=quickstart/rl/
   --rollouts/final.npz

# Train AIRL from demonstrations. Tensorboard logs saved in output/ (default log directory).
python -m imitation.scripts.train_adversarial airl with pendulum common.fast
   --demonstrations.fast train.fast rl.fast fast demonstrations.rollout_path=quickstart/rl/
   --rollouts/final.npz
```

**Note:** Remove the `fast` options from the commands above to allow training run to completion.

**Tip:** python -m imitation.scripts.train_rl print_config will list Sacred script options. These configuration options are also documented in each script's docstrings.

1.4.2 Python Interface Quickstart

Here's an example script that loads CartPole demonstrations and trains BC, GAIL, and AIRL models on that data. You will need to pip install seals or pip install imitation[test] to run this.

```python
"""This is a simple example demonstrating how to clone the behavior of an expert.

Refer to the jupyter notebooks for more detailed examples of how to use the algorithms.
"""

```python
import gym
import numpy as np
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.ppo import MlpPolicy

from imitation.algorithms import bc
from imitation.data import rollout
```

(continues on next page)
from imitation.data.wrappers import RolloutInfoWrapper

env = gym.make("CartPole-v1")
rng = np.random.default_rng(0)

def train_expert():
    print("Training a expert.")
    expert = PPO(
        policy=MlpPolicy,
        env=env,
        seed=0,
        batch_size=64,
        ent_coef=0.0,
        learning_rate=0.0003,
        n_epochs=10,
        n_steps=64,
    )
    expert.learn(100)  # Note: change this to 100000 to train a decent expert.
    return expert

def sample_expert_transitions():
    expert = train_expert()

    print("Sampling expert transitions.")
    rollouts = rollout.rollout(
        expert,
        DummyVecEnv([lambda: RolloutInfoWrapper(env)]),
        rollout.make_sample_until(min_timesteps=None, min_episodes=50),
        rng=rng,
    )
    return rollout.flatten_trajectories(rollouts)

transitions = sample_expert_transitions()
brc_trainer = bc.BC(
    observation_space=env.observation_space,
    action_space=env.action_space,
    demonstrations=transitions,
    rng=rng,
)

reward, _ = evaluate_policy(
    bc_trainer.policy,  # type: ignore[arg-type]
    env,
    n_eval_episodes=3,
    render=True,
)

print(f"Reward before training: {reward}")

print("Training a policy using Behavior Cloning")
1.5 Behavioral Cloning (BC)

Behavioral cloning directly learns a policy by using supervised learning on observation-action pairs from expert demonstrations. It is a simple approach to learning a policy, but the policy often generalizes poorly and does not recover well from errors.

Alternatives to behavioral cloning include DAgger (similar but gathers on-policy demonstrations) and GAIL/AIL (more robust approaches to learning from demonstrations).

1.5.1 Example

Detailed example notebook: Train an Agent using Behavior Cloning

```python
import numpy as np
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.ppo import MlpPolicy
from imitation.algorithms import bc
from imitation.data import rollout
from imitation.data.wrappers import RolloutInfoWrapper

rng = np.random.default_rng(0)
env = gym.make("CartPole-v1")
expert = PPO(policy=MlpPolicy, env=env)
expert.learn(1000)

rollouts = rollout.rollout(
    expert,
    DummyVecEnv([lambda: RolloutInfoWrapper(env)]),
    rollout.make_sample_until(min_timesteps=None, min_episodes=50),
    rng=rng,
)
transitions = rollout.flatten_trajectories(rollouts)

bc_trainer = bc.BC(
    observation_space=env.observation_space,
    action_space=env.action_space,
)  
bc_trainer.train(n_epochs=1)

reward, _ = evaluate_policy(
    bc_trainer.policy,  # type: ignore[arg-type]
    env,
    n_eval_episodes=3,
    render=True,
)
print(f"Reward after training: {reward}\n")
```
demonstrations=transitions,
    rng=rng,
})
bc_trainer.train(n_epochs=1)
reward, _ = evaluate_policy(bc_trainer.policy, env, 10)
print("Reward: ", reward)

1.5.2 API

class imitation.algorithms.bc.BC(*, observation_space, action_space, rng, policy=None,
    demonstrations=None, batch_size=32, minibatch_size=None,
    optimizer_cls=<class 'torch.optim.adam.Adam'>, optimizer_kwargs=None, ent_weight=0.001, l2_weight=0.0,
    device='auto', custom_logger=None)

Bases: DemonstrationAlgorithm

Behavioral cloning (BC).

Recovers a policy via supervised learning from observation-action pairs.

__init__(*, observation_space, action_space, rng, policy=None, demonstrations=None, batch_size=32,
    minibatch_size=None, optimizer_cls=<class 'torch.optim.adam.Adam'>, optimizer_kwargs=None, ent_weight=0.001, l2_weight=0.0, device='auto', custom_logger=None)

Builds BC.

Parameters

- **observation_space** (Space) – the observation space of the environment.
- **action_space** (Space) – the action space of the environment.
- **rng** (Generator) – the random state to use for the random number generator.
- **policy** (Optional[ActorCriticPolicy]) – a Stable Baselines3 policy; if unspecified, defaults to FeedForward32Policy.
- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal, None]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).
- **batch_size** (int) – The number of samples in each batch of expert data.
- **minibatch_size** (Optional[int]) – size of minibatch to calculate gradients over. The gradients are accumulated until batch_size examples are processed before making an optimization step. This is useful in GPU training to reduce memory usage, since fewer examples are loaded into memory at once, facilitating training with larger batch sizes, but is generally slower. Must be a factor of batch_size. Optional, defaults to batch_size.
- **optimizer_cls** (Type[Optimizer]) – optimiser to use for supervised training.
- **optimizer_kwargs** (Optional[Mapping[str, Any]]) – keyword arguments, excluding learning rate and weight decay, for optimiser construction.
- **ent_weight** (float) – scaling applied to the policy’s entropy regularization.
- **l2_weight** (float) – scaling applied to the policy’s L2 regularization.

1.5. Behavioral Cloning (BC)
- **device** (Union[Union[str, device]]) – name/identity of device to place policy on.
- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

**Raises**

- **ValueError** – If `weight_decay` is specified in `optimizer_kwargs` (use the parameter `l2_weight` instead), or if the batch size is not a multiple of the minibatch size.

**allow_variable_horizon**: `bool`

- If True, allow variable horizon trajectories; otherwise error if detected.

**property policy**: **ActorCriticPolicy**

- Returns a policy imitating the demonstration data.

**Return type**

- ActorCriticPolicy

**save_policy**(policy_path)

- Save policy to a path. Can be reloaded by `.reconstruct_policy()`.

**Parameters**

- **policy_path** (Union[str, bytes, PathLike]) – path to save policy to.

**Return type**

- None

**set_demonstrations**(demonstrations)

- Sets the demonstration data.

- Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

**Parameters**

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[array, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

**Return type**

- None

**train**(*, n_epochs=None, n_batches=None, on_epoch_end=None, on_batch_end=None, log_interval=500, log_rollouts_venv=None, log_rollouts_n_episodes=5, progress_bar=True, reset_tensorboard=False)

- Train with supervised learning for some number of epochs.

- Here an ‘epoch’ is just a complete pass through the expert data loader, as set by `self.set_expert_data_loader()`. Note, that when you specify `n_batches` smaller than the number of batches in an epoch, the `on_epoch_end` callback will never be called.

**Parameters**

- **n_epochs** (Optional[int]) – Number of complete passes made through expert data before ending training. Provide exactly one of `n_epochs` and `n_batches`.

- **n_batches** (Optional[int]) – Number of batches loaded from dataset before ending training. Provide exactly one of `n_epochs` and `n_batches`.

- **on_epoch_end** (Optional[Callable[[], None]]) – Optional callback with no parameters to run at the end of each epoch.

- **on_batch_end** (Optional[Callable[[], None]]) – Optional callback with no parameters to run at the end of each batch.
• \textbf{log\_interval} (int) – Log stats after every \texttt{log\_interval} batches.

• \textbf{log\_rollouts\_venv} (Optional[VecEnv]) – If not None, then this VecEnv (whose observation and actions spaces must match \texttt{self.observation\_space} and \texttt{self.action\_space}) is used to generate rollout stats, including average return and average episode length. If None, then no rollouts are generated.

• \textbf{log\_rollouts\_n\_episodes} (int) – Number of rollouts to generate when calculating rollout stats. Non-positive number disables rollouts.

• \textbf{progress\_bar} (bool) – If True, then show a progress bar during training.

• \textbf{reset\_tensorboard} (bool) – If True, then start plotting to Tensorboard from $x=0$ even if \texttt{.train()} logged to Tensorboard previously. Has no practical effect if \texttt{.train()} is being called for the first time.

1.6 Generative Adversarial Imitation Learning (GAIL)

GAIL learns a policy by simultaneously training it with a discriminator that aims to distinguish expert trajectories against trajectories from the learned policy.

1.6.1 Notes

• GAIL paper: Generative Adversarial Imitation Learning

Example

Detailed example notebook: \textit{Train an Agent using Generative Adversarial Imitation Learning}

```python
import numpy as np
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.ppo import MlpPolicy
from imitation.algorithms.adversarial.gail import GAIL
from imitation.data import rollout
from imitation.data.wrappers import RolloutInfoWrapper
from imitation.rewards.reward_nets import BasicRewardNet
from imitation.util.networks import RunningNorm
from imitation.util.util import make_vec_env

rng = np.random.default_rng(0)

env = gym.make("seals/CartPole-v0")
expert = PPO(policy=MlpPolicy, env=env, n_steps=64)
expert.learn(1000)

rollouts = rollout.rollout(
    expert,
    make_vec_env(
```

(continues on next page)
imitation

"seals/CartPole-v0",
  n_envs=5,
  post_wrappers=[lambda env, _: RolloutInfoWrapper(env)],
  rng=rng,
),
rollout.make_sample_until(min_timesteps=None, min_episodes=60),
  rng=rng,
)

venv = make_vec_env("seals/CartPole-v0", n_envs=5, rng=rng)
learner = PPO(env=venv, policy=MlpPolicy)
reward_net = BasicRewardNet(
  venv.observation_space,
  venv.action_space,
  normalize_input_layer=RunningNorm,
)
gail_trainer = GAIL(
  demonstrations=rollouts,
  demo_batch_size=1024,
  gen_replay_buffer_capacity=2048,
  n_disc_updates_per_round=4,
  venv=venv,
  gen_algo=learner,
  reward_net=reward_net,
)
gail_trainer.train(20000)
rewards, _ = evaluate_policy(learner, venv, 100, return_episode_rewards=True)
print("Rewards:", rewards)

API

class imitation.algorithms.adversarial.gail.GAIL(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)

Bases: AdversarialTrainer

Generative Adversarial Imitation Learning (GAIL).

__init__(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)

Generative Adversarial Imitation Learning.

Parameters

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

- **demo_batch_size** (int) – The number of samples in each batch of expert data. The discriminator batch size is twice this number because each discriminator batch contains a generator sample for every expert sample.

- **venv** (VecEnv) – The vectorized environment to train in.
• **gen_algo** (*BaseAlgorithm*) – The generator RL algorithm that is trained to maximize discriminator confusion. Environment and logger will be set to *venv* and *custom_logger*.

• **reward_net** (*RewardNet*) – a Torch module that takes an observation, action and next observation tensor as input, then computes the logits. Used as the GAIL discriminator.

• **kwargs** – Passed through to *AdversarialTrainer.*

allow_variable_horizon: *bool*

If True, allow variable horizon trajectories; otherwise error if detected.

**property logger:** *HierarchicalLogger*

Return type

*HierarchicalLogger*

logits_expert_is_high(*state, action, next_state, done, log_policy_act_prob=None*)

Compute the discriminator's logits for each state-action sample.

Parameters

• **state** (*Tensor*) – The state of the environment at the time of the action.

• **action** (*Tensor*) – The action taken by the expert or generator.

• **next_state** (*Tensor*) – The state of the environment after the action.

• **done** (*Tensor*) – whether a *terminal state* (as defined under the MDP of the task) has been reached.

• **log_policy_act_prob** (*Optional*[Tensor]) – The log probability of the action taken by the generator, log *P(a|s).*

Return type

*Tensor*

Returns

The logits of the discriminator for each state-action sample.

**property policy:** *BasePolicy*

Returns a policy imitating the demonstration data.

Return type

*BasePolicy*

**property reward_test:** *RewardNet*

Reward used to train policy at “test” time after adversarial training.

Return type

*RewardNet*

**property reward_train:** *RewardNet*

Reward used to train generator policy.

Return type

*RewardNet*

**set_demonstrations**(*demonstrations*)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.
Parameters

`demonstrations` (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[np.ndarray, torch.Tensor]]], TransitionsMinimal]) – Either a Torch `DataLoader`, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, `TransitionKind` instance, or a Sequence of Trajectory objects.

Return type

None

`train(total_timesteps, callback=None)`

Alternates between training the generator and discriminator.

Every “round” consists of a call to `train_gen(self.gen_train_timesteps)`, a call to `train_disc`, and finally a call to `callback(round)`.

Training ends once an additional “round” would cause the number of transitions sampled from the environment to exceed `total_timesteps`.

Parameters

- `total_timesteps` (int) – An upper bound on the number of transitions to sample from the environment during training.
- `callback` (Optional[Callable[[int, None]]]) – A function called at the end of every round which takes in a single argument, the round number. Round numbers are in range(`total_timesteps // self.gen_train_timesteps`).

Return type

None

`train_disc(*, expert_samples=None, gen_samples=None)`

Perform a single discriminator update, optionally using provided samples.

Parameters

- `expert_samples` (Optional[Mapping]) – Transition samples from the expert in dictionary form. If provided, must contain keys corresponding to every field of the `Transitions` dataclass except “infos”. All corresponding values can be either NumPy arrays or Tensors. Extra keys are ignored. Must contain `self.demo_batch_size` samples. If this argument is not provided, then `self.demo_batch_size` expert samples from `self.demo_data_loader` are used by default.
- `gen_samples` (Optional[Mapping]) – Transition samples from the generator policy in same dictionary form as `expert_samples`. If provided, must contain exactly `self.demo_batch_size` samples. If not provided, then take `len(expert_samples)` samples from the generator replay buffer.

Return type

Mapping[str, float]

Returns

Statistics for discriminator (e.g. loss, accuracy).

`train_gen(total_timesteps=None, learn_kwargs=None)`

Trains the generator to maximize the discriminator loss.

After the end of training populates the generator replay buffer (used in discriminator training) with `self.disc_batch_size` transitions.

Parameters
• **total_timesteps** (Optional[int]) – The number of transitions to sample from `self.venv_train` during training. By default, `self.gen_train_timesteps`.

• **learn_kwargs** (Optional[Mapping]) – kwargs for the Stable Baselines `RLModel.learn()` method.

**Return type**

None

venv: VecEnv

The original vectorized environment.

venv_train: VecEnv

Like `self.venv`, but wrapped with train reward unless in debug mode.

If `debug_use_ground_truth=True` was passed into the initializer then `self.venv_train` is the same as `self.venv`.

venw_wrapped: VecEnvWrapper

---

### `imitation.algorithms.adversarial.common.AdversarialTrainer`(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, demo_minibatch_size=None, n_disc_updates_per_round=2, log_dir='output', disc_opt_cls=<class 'torch.optim.adam.Adam'>, disc_opt_kwargs=None, gen_train_timesteps=None, gen_replay_buffer_capacity=None, custom_logger=None, init_tensorboard=False, init_tensorboard_graph=False, debug_use_ground_truth=False, allow_variable_horizon=False)

Bases: `DemonstrationAlgorithm[Transitions]`

Base class for adversarial imitation learning algorithms like GAIL and AIRL.

__init__(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, demo_minibatch_size=None, n_disc_updates_per_round=2, log_dir='output', disc_opt_cls=<class 'torch.optim.adam.Adam'>, disc_opt_kwargs=None, gen_train_timesteps=None, gen_replay_buffer_capacity=None, custom_logger=None, init_tensorboard=False, init_tensorboard_graph=False, debug_use_ground_truth=False, allow_variable_horizon=False)

Builds AdversarialTrainer.

**Parameters**

• **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

• **demo_batch_size** (int) – The number of samples in each batch of expert data. The discriminator batch size is twice this number because each discriminator batch contains a generator sample for every expert sample.
• **venv** (VecEnv) – The vectorized environment to train in.

• **gen_algo** (BaseAlgorithm) – The generator RL algorithm that is trained to maximize discriminator confusion. Environment and logger will be set to venv and custom_logger.

• **reward_net** (RewardNet) – a Torch module that takes an observation, action and next observation tensors as input and computes a reward signal.

• **demo_minibatch_size** (Optional[int]) – size of minibatch to calculate gradients over. The gradients are accumulated until the entire batch is processed before making an optimization step. This is useful in GPU training to reduce memory usage, since fewer examples are loaded into memory at once, facilitating training with larger batch sizes, but is generally slower. Must be a factor of demo_batch_size. Optional, defaults to demo_batch_size.

• **n_disc_updates_per_round** (int) – The number of discriminator updates after each round of generator updates in AdversarialTrainer.learn().

• **log_dir** (Union[str, bytes, PathLike]) – Directory to store TensorBoard logs, plots, etc. in.

• **disc_opt_cls** (Type[Optimizer]) – The optimizer for discriminator training.

• **disc_opt_kwargs** (Optional[Mapping]) – Parameters for discriminator training.

• **gen_train_timesteps** (Optional[int]) – The number of steps to train the generator policy for each iteration. If None, then defaults to the batch size (for on-policy) or number of environments (for off-policy).

• **gen_replay_buffer_capacity** (Optional[int]) – The capacity of the generator replay buffer (the number of obs-action-obs samples from the generator that can be stored). By default this is equal to gen_train_timesteps, meaning that we sample only from the most recent batch of generator samples.

• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

• **init_tensorboard** (bool) – If True, makes various discriminator TensorBoard summaries.

• **init_tensorboard_graph** (bool) – If both this and init_tensorboard are True, then write a Tensorboard graph summary to disk.

• **debug_use_ground_truth** (bool) – If True, use the ground truth reward for self.train_env. This disables the reward wrapping that would normally replace the environment reward with the learned reward. This is useful for sanity checking that the policy training is functional.

• **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

Raises

ValueError – if the batch size is not a multiple of the minibatch size.

allow_variable_horizon: bool

If True, allow variable horizon trajectories; otherwise error if detected.

property logger: HierarchicalLogger

Return type

HierarchicalLogger
abstract logits_expert_is_high(state, action, next_state, done, log_policy_act_prob=None)

Compute the discriminator’s logits for each state-action sample.
A high value corresponds to predicting expert, and a low value corresponds to predicting generator.

Parameters

• state (Tensor) – state at time t, of shape (batch_size,) + state_shape.
• action (Tensor) – action taken at time t, of shape (batch_size,) + action_shape.
• next_state (Tensor) – state at time t+1, of shape (batch_size,) + state_shape.
• done (Tensor) – binary episode completion flag after action at time t, of shape (batch_size,).
• log_policy_act_prob (Optional[Tensor]) – log probability of generator policy taking action at time t.

Return type
Tensor

Returns
Discriminator logits of shape (batch_size,). A high output indicates an expert-like transition.

property policy: BasePolicy
Returns a policy imitating the demonstration data.

Return type
BasePolicy

abstract property reward_test: RewardNet
Reward used to train policy at “test” time after adversarial training.

Return type
RewardNet

abstract property reward_train: RewardNet
Reward used to train generator policy.

Return type
RewardNet

set_demonstrations(demonstrations)
Sets the demonstration data.
Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

Parameters
demonstrations (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

Return type
None

train(total_timesteps, callback=None)
Alternates between training the generator and discriminator.
Every “round” consists of a call to train_gen(self, gen_train_timesteps), a call to train_disc, and finally a call to callback(round).
Training ends once an additional “round” would cause the number of transitions sampled from the environment to exceed \textit{total_timesteps}.

Parameters

\begin{itemize}
\item \textbf{total_timesteps} (int) – An upper bound on the number of transitions to sample from the environment during training.
\item \textbf{callback} (Optional[Callable[[int, None]]]) – A function called at the end of every round which takes in a single argument, the round number. Round numbers are in range(total_timesteps // self.gen_train_timesteps).
\end{itemize}

Return type

None

\textbf{train_disc}(*, expert_samples=None, gen_samples=None)

Perform a single discriminator update, optionally using provided samples.

Parameters

\begin{itemize}
\item \textbf{expert_samples} (Optional[Mapping]) – Transition samples from the expert in dictionary form. If provided, must contain keys corresponding to every field of the \textit{Transitions} dataclass except “infos”. All corresponding values can be either NumPy arrays or Tensors. Extra keys are ignored. Must contain \textit{self.demo_batch_size} samples. If this argument is not provided, then \textit{self.demo_batch_size} expert samples from \textit{self.demo_data_loader} are used by default.
\item \textbf{gen_samples} (Optional[Mapping]) – Transition samples from the generator policy in same dictionary form as \textit{expert_samples}. If provided, must contain exactly \textit{self.demo_batch_size} samples. If not provided, then take \textit{len(expert_samples)} samples from the generator replay buffer.
\end{itemize}

Return type

Mapping[str, float]

Returns

Statistics for discriminator (e.g. loss, accuracy).

\textbf{train_gen}(total_timesteps=None, learn_kwags=None)

Trains the generator to maximize the discriminator loss.

After the end of training populates the generator replay buffer (used in discriminator training) with \textit{self.disc_batch_size} transitions.

Parameters

\begin{itemize}
\item \textbf{total_timesteps} (Optional[int]) – The number of transitions to sample from \textit{self.venv_train} during training. By default, \textit{self.gen_train_timesteps}.
\item \textbf{learn_kwags} (Optional[Mapping]) – kwargs for the Stable Baselines \textit{RModel.learn()} method.
\end{itemize}

Return type

None

\textbf{venv}: \textbf{VecEnv}

The original vectorized environment.

\textbf{venv_train}: \textbf{VecEnv}

Like \textit{self.venv}, but wrapped with train reward unless in debug mode.

If \textit{debug_use_ground_truth=True} was passed into the initializer then \textit{self.venv_train} is the same as \textit{self.venv}.
venv_wrapped: VecEnvWrapper

1.7 Adversarial Inverse Reinforcement Learning (AIRL)

AIRL, similar to GAIL, adversarially trains a policy against a discriminator that aims to distinguish the expert demonstrations from the learned policy. Unlike GAIL, AIRL recovers a reward function that is more generalizable to changes in environment dynamics.

The expert policy must be stochastic.

1.7.1 Notes

- AIRL paper: Learning Robust Rewards with Adversarial Inverse Reinforcement Learning

Example

Detailed example notebook: Train an Agent using Adversarial Inverse Reinforcement Learning

```python
import numpy as np
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.ppo import MlpPolicy
from imitation.algorithms.adversarial.airl import AIRL
from imitation.data import rollout
from imitation.data.wrappers import RolloutInfoWrapper
from imitation.rewards.reward_nets import BasicShapedRewardNet
from imitation.util.networks import RunningNorm
from imitation.util.util import make_vec_env

rng = np.random.default_rng(0)

env = gym.make("seals/CartPole-v0")
expert = PPO(policy=MlpPolicy, env=env)
expert.learn(1000)

rollouts = rollout.rollout(
    expert,
    make_vec_env(
        "seals/CartPole-v0",
        rng=rng,
        n_envs=5,
        post_wrappers=[lambda env, _: RolloutInfoWrapper(env)],
    ),
    rollout.make_sample_until(min_timesteps=None, min_episodes=60),
    rng=rng,
)
```

(continues on next page)
venv = make_vec_env("seals/CartPole-v0", rng=rng, n_envs=8)
learner = PPO(env=venv, policy=MlpPolicy)
reward_net = BasicShapedRewardNet(
    venv.observation_space,
    venv.action_space,
    normalize_input_layer=RunningNorm,
)
airl_trainer = AIRL(
    demonstrations=rollouts,
    demo_batch_size=1024,
    gen_replay_buffer_capacity=2048,
    n_disc_updates_per_round=4,
    venv=venv,
    gen_algo=learner,
    reward_net=reward_net,
)
airl_trainer.train(20000)
rewards, _ = evaluate_policy(learner, venv, 100, return_episode_rewards=True)
print("Rewards: ", rewards)

API

class imitation.algorithms.adversarial.airl.AIRL(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)
Bases: AdversarialTrainer
Adversarial Inverse Reinforcement Learning (AIRL).

__init__(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)
Builds an AIRL trainer.

Parameters

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).
- **demo_batch_size** (int) – The number of samples in each batch of expert data. The discriminator batch size is twice this number because each discriminator batch contains a generator sample for every expert sample.
- **venv** (VecEnv) – The vectorized environment to train in.
- **gen_algo** (BaseAlgorithm) – The generator RL algorithm that is trained to maximize discriminator confusion. Environment and logger will be set to venv and custom_logger.
- **reward_net** (RewardNet) – Reward network; used as part of AIRL discriminator.
- **kwargs** – Passed through to AdversarialTrainer.__init__.

Raises

TypeError – If gen_algo.policy does not have an evaluate_actions attribute (present in ActorCriticPolicy), needed to compute log-probability of actions.
allow_variable_horizon: bool
If True, allow variable horizon trajectories; otherwise error if detected.

property logger: HierarchicalLogger
Return type
HierarchicalLogger

logits_expert_is_high(state, action, next_state, done, log_policy_act_prob=None)
Compute the discriminator’s logits for each state-action sample.

In Fu’s AIRL paper (https://arxiv.org/pdf/1710.11248.pdf), the discriminator output was given as

\[ D_\theta(s, a) = \frac{\exp r_\theta(s, a)}{\exp r_\theta(s, a) + \pi(a|s)} \]

with a high value corresponding to the expert and a low value corresponding to the generator.

In other words, the discriminator output is the probability that the action is taken by the expert rather than the generator.

The logit of the above is given as

\[ \logit(D_\theta(s, a)) = r_\theta(s, a) - \log \pi(a|s) \]

which is what is returned by this function.

Parameters
- state (Tensor) – The state of the environment at the time of the action.
- action (Tensor) – The action taken by the expert or generator.
- next_state (Tensor) – The state of the environment after the action.
- done (Tensor) – whether a terminal state (as defined under the MDP of the task) has been reached.
- log_policy_act_prob (Optional[Tensor]) – The log probability of the action taken by the generator, \(\log \pi(a|s)\).

Return type
Tensor

Returns
The logits of the discriminator for each state-action sample.

Raises
TypeError – If log_policy_act_prob is None.

property policy: BasePolicy
Returns a policy imitating the demonstration data.

Return type
BasePolicy

property reward_test: RewardNet
Returns the unshaped version of reward network used for testing.

Return type
RewardNet
property reward_train: RewardNet

Reward used to train generator policy.

Return type

RewardNet

set_demonstrations(demonstrations)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

Parameters

demonstrations (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

Return type

None

train(total_timesteps, callback=None)

Alternates between training the generator and discriminator.

Every “round” consists of a call to train_gen(self.gen_train_timesteps), a call to train_disc, and finally a call to callback(round).

Training ends once an additional “round” would cause the number of transitions sampled from the environment to exceed total_timesteps.

Parameters

• total_timesteps (int) – An upper bound on the number of transitions to sample from the environment during training.

• callback (Optional[Callable[[int], None]]) – A function called at the end of every round which takes in a single argument, the round number. Round numbers are in range(total_timesteps // self.gen_train_timesteps).

Return type

None

train_disc(*, expert_samples=None, gen_samples=None)

Perform a single discriminator update, optionally using provided samples.

Parameters

• expert_samples (Optional[Mapping]) – Transition samples from the expert in dictionary form. If provided, must contain keys corresponding to every field of the Transitions dataclass except “infos”. All corresponding values can be either NumPy arrays or Tensors. Extra keys are ignored. Must contain self.demo_batch_size samples. If this argument is not provided, then self.demo_batch_size expert samples from self.demo_data_loader are used by default.

• gen_samples (Optional[Mapping]) – Transition samples from the generator policy in same dictionary form as expert_samples. If provided, must contain exactly self.demo_batch_size samples. If not provided, then take len(expert_samples) samples from the generator replay buffer.

Return type

Mapping[str, float]
Returns
Statistics for discriminator (e.g. loss, accuracy).

train_gen(total_timesteps=None, learn_kwargs=None)
Trains the generator to maximize the discriminator loss.

After the end of training populates the generator replay buffer (used in discriminator training) with
self.disc_batch_size transitions.

Parameters
- **total_timesteps** (Optional[int]) – The number of transitions to sample from
  self.venv_train during training. By default, self.gen_train_timesteps.
- **learn_kwargs** (Optional[Mapping]) – kwargs for the Stable Baselines RLMModel.learn()
  method.

Return type
None

venv: VecEnv
The original vectorized environment.

venv_train: VecEnv
Like self.venv, but wrapped with train reward unless in debug mode.

venv_wrapped: VecEnvWrapper

class imitation.algorithms.adversarial.common.AdversarialTrainer(*, demonstrations,
  demo_batch_size, venv,
  gen_algo, reward_net,
  demo_minibatch_size=None,
  n_disc_updates_per_round=2,
  log_dir='output/',
  disc_opt_cls=<class 'torch.optim.adam.Adam'>,
  disc_opt_kwargs=None,
  gen_train_timesteps=None,
  gen_replay_buffer_capacity=None,
  custom_logger=None,
  init_tensorboard=False,
  init_tensorboard_graph=False,
  debug_use_ground_truth=False,
  allow_variable_horizon=False)

Bases: DemonstrationAlgorithm[Transitions]

Base class for adversarial imitation learning algorithms like GAIL and AIRL.

__init__(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, demo_minibatch_size=None,
  n_disc_updates_per_round=2, log_dir='output/', disc_opt_cls=<class 'torch.optim.adam.Adam'>,
  disc_opt_kwargs=None, gen_train_timesteps=None, gen_replay_buffer_capacity=None,
  custom_logger=None, init_tensorboard=False, init_tensorboard_graph=False,
  debug_use_ground_truth=False, allow_variable_horizon=False)

Builds AdversarialTrainer.
Parameters

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

- **demo_batch_size** (int) – The number of samples in each batch of expert data. The discriminator batch size is twice this number because each discriminator batch contains a generator sample for every expert sample.

- **venv** (VecEnv) – The vectorized environment to train in.

- **gen_algo** (BaseAlgorithm) – The generator RL algorithm that is trained to maximize discriminator confusion. Environment and logger will be set to venv and custom_logger.

- **reward_net** (RewardNet) – a Torch module that takes an observation, action and next observation tensors as input and computes a reward signal.

- **demo_minibatch_size** (Optional[int]) – size of minibatch to calculate gradients over. The gradients are accumulated until the entire batch is processed before making an optimization step. This is useful in GPU training to reduce memory usage, since fewer examples are loaded into memory at once, facilitating training with larger batch sizes, but is generally slower. Must be a factor of demo_batch_size. Optional, defaults to demo_batch_size.

- **n_disc_updates_per_round** (int) – The number of discriminator updates after each round of generator updates in AdversarialTrainer.learn().

- **log_dir** (Union[str, bytes, PathLike]) – Directory to store TensorBoard logs, plots, etc. in.

- **disc_opt_cls** (Type[Optimizer]) – The optimizer for discriminator training.

- **disc_opt_kwargs** (Optional[Mapping]) – Parameters for discriminator training.

- **gen_train_timesteps** (Optional[int]) – The number of steps to train the generator policy for each iteration. If None, then defaults to the batch size (for on-policy) or number of environments (for off-policy).

- **gen_replay_buffer_capacity** (Optional[int]) – The capacity of the generator replay buffer (the number of obs-action-obs samples from the generator that can be stored). By default this is equal to gen_train_timesteps, meaning that we sample only from the most recent batch of generator samples.

- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

- **init_tensorboard** (bool) – If True, makes various discriminator TensorBoard summaries.

- **init_tensorboard_graph** (bool) – If both this and init_tensorboard are True, then write a Tensorboard graph summary to disk.

- **debug_use_ground_truth** (bool) – If True, use the ground truth reward for self.train_env. This disables the reward wrapping that would normally replace the environment reward with the learned reward. This is useful for sanity checking that the policy training is functional.

- **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination

Raises

* ValueError – if the batch size is not a multiple of the minibatch size.

**allow_variable_horizon**: bool

If True, allow variable horizon trajectories; otherwise error if detected.

**abstract** **logits_expert_is_high**(state, action, next_state, done, log_policy_act_prob=None)

Compute the discriminator’s logits for each state-action sample.

A high value corresponds to predicting expert, and a low value corresponds to predicting generator.

**Parameters**

- **state** (Tensor) – state at time t, of shape (batch_size,) + state_shape.
- **action** (Tensor) – action taken at time t, of shape (batch_size,) + action_shape.
- **next_state** (Tensor) – state at time t+1, of shape (batch_size,) + state_shape.
- **done** (Tensor) – binary episode completion flag after action at time t, of shape (batch_size,).
- **log_policy_act_prob** (Optional[Tensor]) – log probability of generator policy taking action at time t.

**Return type**

Tensor

**Returns**

Discriminator logits of shape (batch_size,). A high output indicates an expert-like transition.

**property** **policy**: BasePolicy

Returns a policy imitating the demonstration data.

**Return type**

BasePolicy

**abstract** **property** **reward_test**: RewardNet

Reward used to train policy at “test” time after adversarial training.

**Return type**

RewardNet

**abstract** **property** **reward_train**: RewardNet

Reward used to train generator policy.

**Return type**

RewardNet

**set_demonstrations**(demonstrations)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

**Parameters**

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.
Return type
None

\texttt{train}(total\_timesteps, \textit{callback}=None)

Alternates between training the generator and discriminator.

Every “round” consists of a call to \texttt{train\_gen(self.gen\_train\_timesteps)}, a call to \texttt{train\_disc}, and finally a call to \texttt{callback(round)}.

Training ends once an additional “round” would cause the number of transitions sampled from the environment to exceed \texttt{total\_timesteps}.

Parameters
- \texttt{total\_timesteps} (int) – An upper bound on the number of transitions to sample from the environment during training.
- \texttt{callback} (Optional[Callable[int, None]]) – A function called at the end of every round which takes in a single argument, the round number. Round numbers are in \texttt{range(total\_timesteps // self.gen\_train\_timesteps)}.

Return type
None

\texttt{train\_disc}(*, expert\_samples=None, gen\_samples=None)

Perform a single discriminator update, optionally using provided samples.

Parameters
- \texttt{expert\_samples} (Optional[Mapping]) – Transition samples from the expert in dictionary form. If provided, must contain keys corresponding to every field of the \texttt{Transitions} dataclass except “infos”. All corresponding values can be either NumPy arrays or Tensors. Extra keys are ignored. Must contain \texttt{self.demo\_batch\_size} samples. If this argument is not provided, then \texttt{self.demo\_batch\_size} expert samples from \texttt{self.demo\_data\_loader} are used by default.
- \texttt{gen\_samples} (Optional[Mapping]) – Transition samples from the generator policy in same dictionary form as \texttt{expert\_samples}. If provided, must contain exactly \texttt{self.demo\_batch\_size} samples. If not provided, then take \texttt{len(expert\_samples)} samples from the generator replay buffer.

Return type
Mapping[str, float]

Returns
Statistics for discriminator (e.g. loss, accuracy).

\texttt{train\_gen}(total\_timesteps=None, learn\_kwargs=None)

Trains the generator to maximize the discriminator loss.

After the end of training populates the generator replay buffer (used in discriminator training) with \texttt{self.disc\_batch\_size} transitions.

Parameters
- \texttt{total\_timesteps} (Optional[int]) – The number of transitions to sample from \texttt{self.venv\_train} during training. By default, \texttt{self.gen\_train\_timesteps}.
- \texttt{learn\_kwargs} (Optional[Mapping]) – kwargs for the Stable Baselines \texttt{RLModel.learn()} method.

Return type
None
venv: VecEnv
The original vectorized environment.

venv_train: VecEnv
Like self.venv, but wrapped with train reward unless in debug mode.

If debug_use_ground_truth=True was passed into the initializer then self.venv_train is the same as self.venv.

venv_wrapped: VecEnvWrapper

1.8 DAgger

DAgger (Dataset Aggregation) iteratively trains a policy using supervised learning on a dataset of observation-action pairs from expert demonstrations (like behavioral cloning), runs the policy to gather observations, queries the expert for good actions on those observations, and adds the newly labeled observations to the dataset. DAgger improves on behavioral cloning by training on a dataset that better resembles the observations the trained policy is likely to encounter, but it requires querying the expert online.

1.8.1 Notes

- DAgger paper: A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning

Example

Detailed example notebook: Train an Agent using the DAgger Algorithm

```python
import tempfile
import numpy as np
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.ppo import MlpPolicy

from imitation.algorithms import bc
from imitation.algorithms.dagger import SimpleDAggerTrainer

rng = np.random.default_rng(0)
env = gym.make("CartPole-v1")
expert = PPO(policy=MlpPolicy, env=env)
expert.learn(1000)
env = DummyVecEnv([lambda: gym.make("CartPole-v1")])

bc_trainer = bc.BC(
    observation_space=env.observation_space,
    action_space=env.action_space,
    rng=rng,
)

with tempfile.TemporaryDirectory(prefix="dagger_example_") as tmpdir:
    print(tmpdir)
    dagger_trainer = SimpleDAggerTrainer(
        bc_trainer,
        expert,  # DAgger evaluator
        env,  # DAgger environment
        expert_path=tmpdir,  # Optional policy to start from
        expert_env=env,  # Optional environment
        # Additional arguments
    )
```

(continues on next page)
imitation

venv=venv,
scratch_dir=tmpdir,
expert_policy=expert,
bc_trainer=bc_trainer,
rng=rng,
)
dagger_trainer.train(2000)

reward, _ = evaluate_policy(dagger_trainer.policy, env, 10)
print("Reward:", reward)

API

class imitation.algorithms.dagger.InteractiveTrajectoryCollector(venv, get_robot_acts, beta, save_dir, rng)

Bases: VecEnvWrapper

DAgger VecEnvWrapper for querying and saving expert actions.

Every call to .step(actions) accepts and saves expert actions to self.save_dir, but only forwards expert actions to the wrapped VecEnv with probability self.beta. With probability 1 - self.beta, a “robot” action (i.e an action from the imitation policy) is forwarded instead.

Demonstrations are saved as TrajectoryWithRew to self.save_dir at the end of every episode.

__init__(venv, get_robot_acts, beta, save_dir, rng)

Builds InteractiveTrajectoryCollector.

Parameters

- **venv** (VecEnv) – vectorized environment to sample trajectories from.
- **get_robot_acts** (Callable[[ndarray, ndarray]]) – get robot actions that can be substituted for human actions. Takes a vector of observations as input & returns a vector of actions.
- **beta** (float) – fraction of the time to use action given to .step() instead of robot action. The choice of robot or human action is independently randomized for each individual Env at every timestep.
- **save_dir** (Union[str, bytes, PathLike]) – directory to save collected trajectories in.
- **rng** (Generator) – random state for random number generation.

close()

Clean up the environment’s resources.

Return type

None

env_is_wrapped(wrapper_class, indices=None)

Check if environments are wrapped with a given wrapper.

Parameters

- **method_name** – The name of the environment method to invoke.
- **indices** (Union[None, int, Iterable[int]]) – Indices of envs whose method to call
**env_method** *(method_name, *method_args, indices=None, **method_kwargs)*

Call instance methods of vectorized environments.

**Parameters**

- **method_name** *(str)* – The name of the environment method to invoke.
- **indices** *(Union[None, int, Iterable[int]])* – Indices of envs whose method to call
- **method_args** – Any positional arguments to provide in the call
- **method_kwargs** – Any keyword arguments to provide in the call

**Return type**

List[bool]

**Returns**

True if the env is wrapped, False otherwise, for each env queried.

**get_attr** *(attr_name, indices=None)*

Return attribute from vectorized environment.

**Parameters**

- **attr_name** *(str)* – The name of the attribute whose value to return
- **indices** *(Union[None, int, Iterable[int]])* – Indices of envs to get attribute from

**Return type**

List[Any]

**Returns**

List of values of ‘attr_name’ in all environments

**get_images** *

Return RGB images from each environment

**Return type**

Sequence[ndarray]

**getattr_depth_check** *(name, already_found)*

See base class.

**Return type**

str

**Returns**

name of module whose attribute is being shadowed, if any.

**getattr_recursive** *(name)*

Recursively check wrappers to find attribute.

**Parameters**

- **name** *(str)* – name of attribute to look for
imitation

Return type

Any

Returns

attribute

metadata = {'render.modes': ['human', 'rgb_array']}

render(mode='human')

Gym environment rendering

Parameters

mode (str) – the rendering type

Return type

Optional[ndarray]

reset()

Resets the environment.

Returns

first observation of a new trajectory.

Return type

obs

seed(seed=None)

Set the seed for the DAgger random number generator and wrapped VecEnv.

The DAgger RNG is used along with self.beta to determine whether the expert or robot action is forwarded to the wrapped VecEnv.

Parameters

seed (Optional[int]) – The random seed. May be None for completely random seeding.

Return type

List[Optional[int]]

Returns

A list containing the seeds for each individual env. Note that all list elements may be None, if the env does not return anything when seeded.

set_attr(attr_name, value, indices=None)

Set attribute inside vectorized environments.

Parameters

• attr_name (str) – The name of attribute to assign new value
• value (Any) – Value to assign to attr_name
• indices (Union[None, int, Iterable[int]]) – Indices of envs to assign value

Return type

None

Returns

step(actions)

Step the environments with the given action

Parameters

actions (ndarray) – the action
Return type
Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]], ndarray, ndarray, List[Dict]]

Returns
observation, reward, done, information

**step_async(actions)**
Steps with a \(1 - \beta\) chance of using self.get_robot_acts instead.

DAgger needs to be able to inject imitation policy actions randomly at some subset of time steps. This method has a self.beta chance of keeping the actions passed in as an argument, and a \(1 - \text{self.beta}\) chance of forwarding actions generated by self.get_robot_acts instead. “robot” (i.e. imitation policy) action if necessary.

At the end of every episode, a TrajectoryWithRew is saved to self.save_dir, where every saved action is the expert action, regardless of whether the robot action was used during that timestep.

Parameters
actions (ndarray) – the _intended_ demonstrator/expert actions for the current state. This will be executed with probability self.beta. Otherwise, a “robot” (typically a BC policy) action will be sampled and executed instead via self.get_robot_act.

Return type
None

**step_wait()**
Returns observation, reward, etc after previous step_async() call.

Stores the transition, and saves trajectory as demo once complete.

Return type
Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]], ndarray, ndarray, List[Dict]]

Returns
Observation, reward, dones (is terminal?) and info dict.

**traj_accum**: Optional[TrajectoryAccumulator]

property unwrapped: VecEnv

Return type
VecEnv

**class imitation.algorithms.dagger.DaggerTrainer(**, venv, scratch_dir, rng, beta_schedule=None, bc_trainer, custom_logger=None)

Bases: BaseImitationAlgorithm

DAgger training class with low-level API suitable for interactive human feedback.

In essence, this is just BC with some helpers for incrementally resuming training and interpolating between demonstrator/learnt policies. Interaction proceeds in “rounds” in which the demonstrator first provides a fresh set of demonstrations, and then an underlying BC is invoked to fine-tune the policy on the entire set of demonstrations collected in all rounds so far. Demonstrations and policy/trainer checkpoints are stored in a directory with the following structure:

```plaintext
scratch-dir-name/
    checkpoint-001.pt
    checkpoint-002.pt
```

(continues on next page)
DEFAULT_N_EPOCHS: int = 4

The default number of BC training epochs in `extend_and_update`.

`__init__`(*, venv, scratch_dir, rng, beta_schedule=None, bc_trainer, custom_logger=None)

Builds DAggerTrainer.

Parameters

• **venv** (VecEnv) – Vectorized training environment.

• **scratch_dir** (Union[str, bytes, PathLike]) – Directory to use to store intermediate training information (e.g. for resuming training).

• **rng** (Generator) – random state for random number generation.

• **beta_schedule** (Optional[Callable[[int], float]]) – Provides a value of beta (the probability of taking expert action in any given state) at each round of training. If None, then linear_beta_schedule will be used instead.

• **bc_trainer** (BC) – A BC instance used to train the underlying policy.

• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

allow_variable_horizon: bool

If True, allow variable horizon trajectories; otherwise error if detected.

property batch_size: int

Return type

int

create_trajectory_collector()

Create trajectory collector to extend current round’s demonstration set.

Return type

`InteractiveTrajectoryCollector`

Returns

A collector configured with the appropriate beta, imitator policy, etc. for the current round. Refer to the documentation for `InteractiveTrajectoryCollector` to see how to use this.
extend_and_update(bc_train_kwargs=None)

Extend internal batch of data and train BC.

Specifically, this method will load new transitions (if necessary), train the model for a while, and advance
the round counter. If there are no fresh demonstrations in the demonstration directory for the current round,
then this will raise a NeedsDemosException instead of training or advancing the round counter. In that case,
the user should call .create_trajectory_collector() and use the returned InteractiveTrajectoryCollector
to produce a new set of demonstrations for the current interaction round.

Parameters
bc_train_kwargs (Optional[Mapping[str, Any]]) – Keyword arguments for calling
BC.train(). If the log_rollouts_venv key is not provided, then it is set to self.venv by de-
default. If neither of the n_epochs and n_batches keys are provided, then n_epochs is set to
self.DEFAULT_N_EPOCHS.

Return type
int

Returns
New round number after advancing the round counter.

property logger: HierarchicalLogger

Returns logger for this object.

Return type
HierarchicalLogger

property policy: BasePolicy

Return type
BasePolicy

save_policy(policy_path)

Save the current policy only (and not the rest of the trainer).

Parameters
policy_path (Union[str, bytes, PathLike]) – path to save policy to.

Return type
None

save_trainer()

Create a snapshot of trainer in the scratch/working directory.

The created snapshot can be reloaded with reconstruct_trainer(). In addition to saving one copy of the
policy in the trainer snapshot, this method saves a second copy of the policy in its own file. Having a second
copy of the policy is convenient because it can be loaded on its own and passed to evaluation routines for
other algorithms.

Returns
a path to one of the created DAggerTrainer checkpoints. policy_path: a path to one of the
created DAggerTrainer policies.

Return type
checkpoint_path

class imitation.algorithms.dagger.SimpleDAggerTrainer(*, venv, scratch_dir, expert_policy, rng,
expert_trajs=None,
**dagger_trainer_kwargs)

Bases: DAggerTrainer
Simpler subclass of DAggerTrainer for training with synthetic feedback.

**DEFAULT_N_EPOCHS**: int = 4

The default number of BC training epochs in `extend_and_update`.

```python
__init__(*, venv, scratch_dir, expert_policy, rng, expert_trajs=None, **dagger_trainer_kwargs)
```

Builds SimpleDAggerTrainer.

**Parameters**

- **venv** (`VecEnv`) – Vectorized training environment. Note that when the robot action is randomly injected (in accordance with `beta_schedule` argument), every individual environment will get a robot action simultaneously for that timestep.
- **scratch_dir** (`Union[Optional[Union[str, bytes, PathLike]]]`) – Directory to use to store intermediate training information (e.g. for resuming training).
- **expert_policy** (`BasePolicy`) – The expert policy used to generate synthetic demonstrations.
- **rng** (`Generator`) – Random state to use for the random number generator.
- **expert_trajs** (`Optional[Sequence[Trajectory]]`) – Optional starting dataset that is inserted into the round 0 dataset.
- **dagger_trainer_kwargs** – Other keyword arguments passed to the superclass initializer `DAggerTrainer.__init__`.

**Raises**

- **ValueError** – The observation or action space does not match between `venv` and `expert_policy`.

**allow_variable_horizon**: bool

If True, allow variable horizon trajectories; otherwise error if detected.

**property batch_size**: int

**Return type**

- int

```python
create_trajectory_collector()
```

Create trajectory collector to extend current round’s demonstration set.

**Return type**

- `InteractiveTrajectoryCollector`

**Returns**

A collector configured with the appropriate beta, imitator policy, etc. for the current round. Refer to the documentation for `InteractiveTrajectoryCollector` to see how to use this.

```python
extend_and_update(bc_train_kwargs=None)
```

Extend internal batch of data and train BC.

Specifically, this method will load new transitions (if necessary), train the model for a while, and advance the round counter. If there are no fresh demonstrations in the demonstration directory for the current round, then this will raise a `NeedsDemosException` instead of training or advancing the round counter. In that case, the user should call `.create_trajectory_collector()` and use the returned `InteractiveTrajectoryCollector` to produce a new set of demonstrations for the current interaction round.

**Parameters**

- **bc_train_kwargs** (`Optional[Mapping[str, Any]]`) – Keyword arguments for calling
BC.train(). If the log_rollouts_venv key is not provided, then it is set to self.venv by default. If neither of the n_epochs and n_batches keys are provided, then n_epochs is set to self.DEFAULT_N_EPOCHS.

Return type
int

Returns
New round number after advancing the round counter.

property logger: HierarchicalLogger
Returns logger for this object.

property policy: BasePolicy
Return type
BasePolicy

save_policy(policy_path)
Save the current policy only (and not the rest of the trainer).

Parameters
policy_path (Union[str, bytes, PathLike]) – path to save policy to.

Return type
None

save_trainer()
Create a snapshot of trainer in the scratch/working directory.

The created snapshot can be reloaded with reconstruct_trainer(). In addition to saving one copy of the policy in the trainer snapshot, this method saves a second copy of the policy in its own file. Having a second copy of the policy is convenient because it can be loaded on its own and passed to evaluation routines for other algorithms.

Returns
a path to one of the created DAggerTrainer checkpoints. policy_path: a path to one of the created DAggerTrainer policies.

Return type
checkpoint_path

train(total_timesteps, *, rollout_round_min_episodes=3, rollout_round_min_timesteps=500, bc_train_kwargs=None)
Train the DAgger agent.

The agent is trained in “rounds” where each round consists of a dataset aggregation step followed by BC update step.

During a dataset aggregation step, self.expert_policy is used to perform rollouts in the environment but there is a 1 - beta chance (beta is determined from the round number and self.beta_schedule) that the DAgger agent’s action is used instead. Regardless of whether the DAgger agent’s action is used during the rollout, the expert action and corresponding observation are always appended to the dataset. The number of environment steps in the dataset aggregation stage is determined by the rollout_round_min* arguments.

During a BC update step, BC.train() is called to update the DAgger agent on all data collected so far.

Parameters
• **total_timesteps** (int) – The number of timesteps to train inside the environment. In practice this is a lower bound, because the number of timesteps is rounded up to finish the minimum number of episodes or timesteps in the last DAgger training round, and the environment timesteps are executed in multiples of `self.venv.num_envs`.

• **rollout_round_min_episodes** (int) – The number of episodes the must be completed before a dataset aggregation step ends.

• **rollout_round_min_timesteps** (int) – The number of environment timesteps that must be completed before a dataset aggregation step ends. Also, that any round will always train for at least `self.batch_size` timesteps, because otherwise BC could fail to receive any batches.

• **bc_train_kwargs** (Optional[dict]) – Keyword arguments for calling `BC.train()`. If the `log_rollouts_venv` key is not provided, then it is set to `self.venv` by default. If neither of the `n_epochs` and `n_batches` keys are provided, then `n_epochs` is set to `self.DEFAULT_N_EPOCHS`.

Return type
None

1.9 Density-based reward modeling

1.9.1 Example

Detailed example notebook: *Learning a Reward Function using Kernel Density*

```python
import pprint
import numpy as np
from stable_baselines3 import PPO
from stable_baselines3.common.policies import ActorCriticPolicy
from imitation.algorithms import density as db
from imitation.data import types
from imitation.util import util
rng = np.random.default_rng(0)
env = util.make_vec_env("Pendulum-v1", rng=rng, n_envs=2)
rollouts = types.load("../tests/testdata/expert_models/pendulum_0/rollouts/final.npz")
imitation_trainer = PPO(ActorCriticPolicy, env)
density_trainer = db.DensityAlgorithm(
venv=env,
demonstrations=rollouts,
rl_algo=imitation_trainer,
rng=rng,
)
density_trainer.train()

def print_stats(density_trainer, n_trajectories):
stats = density_trainer.test_policy(n_trajectories=n_trajectories)
```

(continues on next page)
print("True reward function stats:")
pprint.pprint(stats)
stats_im = density_trainer.test_policy(true_reward=False, n_trajectories=n_→trajectories)
print("Imitation reward function stats:")
pprint.pprint(stats_im)

print("Stats before training:")
print_stats(density_trainer, 1)
density_trainer.train_policy(100)

print("Stats after training:")
print_stats(density_trainer, 1)

1.9.2 API

class imitation.algorithms.density.DensityAlgorithm(*, demonstrations, venv, rng, den-
sity_type=DensityType.STATE_ACTION_DENSITY, kernel='gaussian', kernel_bandwidth=0.5,
rl_algo=None, is_stationary=True, standardise_inputs=True, custom_logger=None,
allow_variable_horizon=False)

Bases: DemonstrationAlgorithm

Learns a reward function based on density modeling.

Specifically, it constructs a non-parametric estimate of \(p(s), p(s,a), p(s,s')\) and then computes a reward using the log of these probabilities.

__init__(*, demonstrations, venv, rng, density_type=DensityType.STATE_ACTION_DENSITY,
kernel='gaussian', kernel_bandwidth=0.5, rl_algo=None, is_stationary=True,
standardise_inputs=True, custom_logger=None, allow_variable_horizon=False)

Builds DensityAlgorithm.

Parameters

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal, None]) – expert demonstration trajectories.
- **density_type** (DensityType) – type of density to train on: single state, state-action pairs, or state-state pairs.
- **kernel** (str) – kernel to use for density estimation with sklearn.KernelDensity.
- **kernel_bandwidth** (float) – bandwidth of kernel. If standardise_inputs is true and you are using a Gaussian kernel, then it probably makes sense to set this somewhere between 0.1 and 1.
- **venv** (VecEnv) – The environment to learn a reward model in. We don’t actually need any environment interaction to fit the reward model, but we use this to extract the observation and action space, and to train the RL algorithm rl_algo (if specified).
- **rng** (Generator) – random state for sampling from demonstrations.
• **rl_algo** (Optional[BaseAlgorithm]) – An RL algorithm to train on the resulting reward model (optional).

• **is_stationary** (bool) – if True, share same density models for all timesteps; if False, use a different density model for each timestep. A non-stationary model is particularly likely to be useful when using STATE_DENSITY, to encourage agent to imitate entire trajectories, not just a few states that have high frequency in the demonstration dataset. If non-stationary, demonstrations must be trajectories, not transitions (which do not contain timesteps).

• **standardise_inputs** (bool) – if True, then the inputs to the reward model will be standardised to have zero mean and unit variance over the demonstration trajectories. Otherwise, inputs will be passed to the reward model with their ordinary scale.

• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

• **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

allow_variable_horizon: bool

If True, allow variable horizon trajectories; otherwise error if detected.

**buffering_wrapper:** BufferingWrapper

**density_type:** DensityType

**is_stationary:** bool

**kernel:** str

**kernel_bandwidth:** float

**property logger:** HierarchicalLogger

Return type
HierarchicalLogger

**property policy:** BasePolicy

Returns a policy imitating the demonstration data.

Return type
BasePolicy

**rl_algo:** Optional[BaseAlgorithm]

**set_demonstrations**(demonstrations)

Sets the demonstration data.

Return type
None

**standardise:** bool

**test_policy**(n_trajectories=10, true_reward=True)

Test current imitation policy on environment & give some rollout stats.

Parameters
• **n_trjectories** (int) – number of rolled-out trajectories.

• **true_reward** (bool) – should this use ground truth reward from underlying environment (True), or imitation reward (False)?

**Returns**

`rollout statistics collected by`  
`imitation.utils.rollout.rollout_stats()`.

**Return type**

`dict`

`train()`

Fits the density model to demonstration data `self.transitions`.

**Return type**

`None`

`train_policy(n_timesteps=1000000, **kwargs)`

Train the imitation policy for a given number of timesteps.

**Parameters**

• **n_timesteps** (int) – number of timesteps to train the policy for.

• **kwargs** (`dict`) – extra arguments that will be passed to the `learn()` method of the imitation RL model. Refer to Stable Baselines docs for details.

**Return type**

`None`

`transitions`:

`Dict[Optional[int], ndarray]`

`venv`:

`VecEnv`

`venv_wrapped`:

`RewardVecEnvWrapper`

`wrapper_callback`:

`WrappedRewardCallback`

### 1.10 Maximum Causal Entropy Inverse Reinforcement Learning (MCE IRL)

Implements Modeling Interaction via the Principle of Maximum Causal Entropy.

#### 1.10.1 Example

Detailed example notebook: *Learn a Reward Function using Maximum Conditional Entropy Inverse Reinforcement Learning*

```python
from functools import partial
from seals import base_envs
from seals.diagnostics.cliff_world import CliffWorldEnv
import numpy as np
```

(continues on next page)
from stable_baselines3.common.vec_env import DummyVecEnv

from imitation.algorithms.mce_irl import (MCEIRL,
mce_occupancy_measures,
mce_partition_fh,
)
from imitation.data import rollout
from imitation.rewards import reward_nets

rng = np.random.default_rng(0)

env_creator = partial(CliffWorldEnv, height=4, horizon=8, width=7, use_xy_obs=True)
env_single = env_creator()

state_env_creator = lambda: base_envs.ExposePOMDPStateWrapper(env_creator())

# This is just a vectorized environment because `generate_trajectories` expects one
state_venv = DummyVecEnv([state_env_creator] * 4)

_, _, pi = mce_partition_fh(env_single)

_, om = mce_occupancy_measures(env_single, pi=pi)

reward_net = reward_nets.BasicRewardNet(
    env_single.observation_space,
    env_single.action_space,
    hid_sizes=[256],
    use_action=False,
    use_done=False,
    use_next_state=False,
)

# training on analytically computed occupancy measures
mce_irl = MCEIRL(
    om,
    env_single,
    reward_net,
    log_interval=250,
    optimizer_kwargs={"lr": 0.01},
    rng=rng,
)

occ_measure = mce_irl.train()

imitation_trajs = rollout.generate_trajectories(
    policy=mce_irl.policy,
    venv=state_venv,
    sample_until=rollout.make_min_timesteps(5000),
    rng=rng,
)

print("Imitation stats: ", rollout.rollout_stats(imitation_trajs))
1.10.2 API

class imitation.algorithms.mce_irl.MCEIRL(demonstrations, env, reward_net, rng, optimizer_cls=<class 'torch.optim.adam.Adam'>, optimizer_kwars=None, discount=1.0, linf_eps=0.001, grad_l2_eps=0.0001, log_interval=100, *, custom_logger=None)

Bases: DemonstrationAlgorithm[TransitionsMinimal]

Tabular MCE IRL.

Reward is a function of observations, but policy is a function of states.

The “observations” effectively exist just to let MCE IRL learn a reward in a reasonable feature space, giving a helpful inductive bias, e.g. that similar states have similar reward.

Since we are performing planning to compute the policy, there is no need for function approximation in the policy.

__init__(demonstrations, env, reward_net, rng, optimizer_cls=<class 'torch.optim.adam.Adam'>, optimizer_kwars=None, discount=1.0, linf_eps=0.001, grad_l2_eps=0.0001, log_interval=100, *, custom_logger=None)

Creates MCE IRL.

Parameters

- **demonstrations** (Union[ndarray, Iterable[Mapping[str, Union[ndarray, Tensor]]]], TransitionsMinimal, None) – Demonstrations from an expert (optional). Can be a sequence of trajectories, or transitions, an iterable over mappings that represent a batch of transitions, or a state occupancy measure. The demonstrations must have observations one-hot coded unless demonstrations is a state-occupancy measure.
- **env** (TabularModelPOMDP) – a tabular MDP.
- **rng** (Generator) – random state used for sampling from policy.
- **reward_net** (RewardNet) – a neural network that computes rewards for the supplied observations.
- **optimizer_cls** (Type[Optimizer]) – optimizer to use for supervised training.
- **optimizer_kwars** (Optional[Mapping[str, Any]]) – keyword arguments for optimizer construction.
- **discount** (float) – the discount factor to use when computing occupancy measure. If not 1.0 (undiscounted), then demonstrations must either be a (discounted) state-occupancy measure, or trajectories. Transitions are not allowed as we cannot discount them appropriately without knowing the timestep they were drawn from.
- **linf_eps** (float) – optimisation terminates if the $l_{\infty}$ distance between the demonstrator’s state occupancy measure and the state occupancy measure for the current reward falls below this value.
- **grad_l2_eps** (float) – optimisation also terminates if the $\ell_2$ norm of the MCE IRL gradient falls below this value.
- **log_interval** (Optional[int]) – how often to log current loss stats (using logging). None to disable.
- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.
Raises

ValueError – if the env horizon is not finite (or an integer).

allow_variable_horizon: bool

If True, allow variable horizon trajectories; otherwise error if detected.

demo_state_om: Optional[ndarray]

property logger: HierarchicalLogger

    Return type
    HierarchicalLogger

property policy: BasePolicy

    Returns a policy imitating the demonstration data.
    
    Return type
    BasePolicy

set_demonstrations(demonstrations)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

Parameters

demonstrations (Union[ndarray, Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

    Return type
    None

train(max_iter=1000)

Runs MCE IRL.

Parameters

max_iter (int) – The maximum number of iterations to train for. May terminate earlier if self.linf_eps or self.grad_l2_eps thresholds are reached.

    Return type
    ndarray

Returns

State occupancy measure for the final reward function. self.reward_net and self.optimizer will be updated in-place during optimisation.

class imitation.algorithms.baseDemonstrationAlgorithm(*, demonstrations, custom_logger=None, allow_variable_horizon=False)

Bases: BaseImitationAlgorithm, Generic[TransitionKind]

An algorithm that learns from demonstration: BC, IRL, etc.

__init__(*, demonstrations, custom_logger=None, allow_variable_horizon=False)

Creates an algorithm that learns from demonstrations.

Parameters

  * demonstrations (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal, None]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a
sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.
- **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

allow_variable_horizon:  bool

If True, allow variable horizon trajectories; otherwise error if detected.

abstract property policy:  BasePolicy

Returns a policy imitating the demonstration data.

Return type

BasePolicy

abstract set_demonstrations(demonstrations)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

Parameters

demonstrations  (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

Return type

None

1.11 Preference comparisons

The preference comparison algorithm learns a reward function from preferences between pairs of trajectories. The comparisons are modeled as being generated from a Bradley-Terry (or Boltzmann rational) model, where the probability of preferring trajectory A over B is proportional to the exponential of the difference between the return of trajectory A minus B. In other words, the difference in returns forms a logit for a binary classification problem, and accordingly the reward function is trained using a cross-entropy loss to predict the preference comparison.

1.11.1 Notes

- Our implementation is based on the Deep Reinforcement Learning from Human Preferences algorithm.
- An ensemble of reward networks can also be trained instead of a single network. The uncertainty in the preference between the member networks can be used to actively select preference queries.
Example

Detailed example notebook: Learning a Reward Function using Preference Comparisons

```python
import numpy as np

from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.ppo import MlpPolicy

from imitation.algorithms import preference_comparisons
from imitation.policies.base import FeedForward32Policy, NormalizeFeaturesExtractor
from imitation.rewards.reward_nets import BasicRewardNet
from imitation.rewards.reward_wrapper import RewardVecEnvWrapper
from imitation.util.networks import RunningNorm
from imitation.util.util import make_vec_env

rng = np.random.default_rng(0)

venv = make_vec_env("Pendulum-v1", rng=rng)

reward_net = BasicRewardNet(
    venv.observation_space, venv.action_space, normalize_input_layer=RunningNorm,
)

fragmenter = preference_comparisons.RandomFragmenter(warning_threshold=0, rng=rng)
gatherer = preference_comparisons.SyntheticGatherer(rng=rng)
preference_model = preference_comparisons.PreferenceModel(reward_net)
reward_trainer = preference_comparisons.BasicRewardTrainer(
    preference_model=preference_model,
    loss=preference_comparisons.CrossEntropyRewardLoss(),
    epochs=3,
    rng=rng,
)

agent = PPO(
    policy=FeedForward32Policy,
    policy_kwargs=dict(
        features_extractor_class=NormalizeFeaturesExtractor,
        features_extractor_kwargs=dict(normalize_class=RunningNorm),
    ),
    env=venv,
    n_steps=2048 // venv.num_envs,
)

trajectory_generator = preference_comparisons.AgentTrainer(
    algorithm=agent,
    reward_fn=reward_net,
    venv=venv,
    exploration_frac=0.0,
    rng=rng,
)
```

(continues on next page)
pref_comparisons = preference_comparisons.PreferenceComparisons("Continued from previous page"
    trajectory_generator,
    reward_net,
    num_iterations=5,
    fragmenter=fragmenter,
    preference_gatherer=gatherer,
    reward_trainer=reward_trainer,
    initial_epoch_multiplier=1,
)

pref_comparisons.train(total_timesteps=5_000, total_comparisons=200)

reward, _ = evaluate_policy(agent.policy, venv, 10)
print("Reward:", reward)

API

class imitation.algorithms.preference_comparisons.PreferenceComparisons(trajectory_generator, reward_model, num_iterations, fragmenter=None, preference_gatherer=None, reward_trainer=None, comparison_queue_size=None, fragment_length=100, transition_oversampling=1, initial_comparison_frac=0.1, initial_epoch_multiplier=200.0, custom_logger=None, allow_variable_horizon=False, rng=None, query_schedule="hyperbolic")

Bases: BaseImitationAlgorithm

Main interface for reward learning using preference comparisons.

__init__(trajectory_generator, reward_model, num_iterations, fragmenter=None, preference_gatherer=None, reward_trainer=None, comparison_queue_size=None, fragment_length=100, transition_oversampling=1, initial_comparison_frac=0.1, initial_epoch_multiplier=200.0, custom_logger=None, allow_variable_horizon=False, rng=None, query_schedule="hyperbolic")

Initialize the preference comparison trainer.

The loggers of all subcomponents are overridden with the logger used by this class.

Parameters
• **trajectory_generator** (*TrajectoryGenerator*) – generates trajectories while optionally training an RL agent on the learned reward function (can also be a sampler from a static dataset of trajectories though).

• **reward_model** (*RewardNet*) – a RewardNet instance to be used for learning the reward

• **num_iterations** (int) – number of times to train the agent against the reward model and then train the reward model against newly gathered preferences.

• **fragmenter** (Optional[*Fragmenter]*) – takes in a set of trajectories and returns pairs of fragments for which preferences will be gathered. These fragments could be random, or they could be selected more deliberately (active learning). Default is a random fragmenter.

• **preference_gatherer** (Optional[*PreferenceGatherer]*) – how to get preferences between trajectory fragments. Default (and currently the only option) is to use synthetic preferences based on ground-truth rewards. Human preferences could be implemented here in the future.

• **reward_trainer** (Optional[*RewardTrainer]*) – trains the reward model based on pairs of fragments and associated preferences. Default is to use the preference model and loss function from DRLHP.

• **comparison_queue_size** (Optional[int]) – the maximum number of comparisons to keep in the queue for training the reward model. If None, the queue will grow without bound as new comparisons are added.

• **fragment_length** (int) – number of timesteps per fragment that is used to elicit preferences

• **transition_oversampling** (float) – factor by which to oversample transitions before creating fragments. Since fragments are sampled with replacement, this is usually chosen > 1 to avoid having the same transition in too many fragments.

• **initial_comparison_frac** (float) – fraction of the total_comparisons argument to train() that will be sampled before the rest of training begins (using a randomly initialized agent). This can be used to pretrain the reward model before the agent is trained on the learned reward, to help avoid irreversibly learning a bad policy from an untrained reward. Note that there will often be some additional pretraining comparisons since comparisons_per_iteration won’t exactly divide the total number of comparisons. How many such comparisons there are depends discontinuously on total_comparisons and comparisons_per_iteration.

• **initial_epoch_multiplier** (float) – before agent training begins, train the reward model for this many more epochs than usual (on fragments sampled from a random agent).

• **custom_logger** (Optional[*HierarchicalLogger]*) – Where to log to; if None (default), creates a new logger.

• **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

• **rng** (Optional[*Generator]*) – random number generator to use for initializing subcomponents such as fragmenter. Only used when default components are used; if you instantiate your own fragmenter, preference gatherer, etc., you are responsible for seeding them!

• **query_schedule** (Union[str, Callable[[float], float]]) – one of (“constant”, “hyperbolic”, “inverse_quadratic”), or a function that takes in a float between 0 and 1 inclusive,
representing a fraction of the total number of timesteps elapsed up to some time T, and returns a potentially unnormalized probability indicating the fraction of total_comparisons that should be queried at that iteration. This function will be called num_iterations times in __init__() with values from np.linspace(0, 1, num_iterations) as input. The outputs will be normalized to sum to 1 and then used to apportion the comparisons among the num_iterations iterations.

Raises
  ValueError – if query_schedule is not a valid string or callable.

allow_variable_horizon: bool
  If True, allow variable horizon trajectories; otherwise error if detected.

property logger: HierarchicalLogger

Return type
  HierarchicalLogger

train(total_timesteps, total_comparisons, callback=None)
  Train the reward model and the policy if applicable.

Parameters
  • total_timesteps (int) – number of environment interaction steps
  • total_comparisons (int) – number of preferences to gather in total
  • callback (Optional[Callable[[int], None]]) – callback functions called at the end of each iteration

Return type
  Mapping[str, Any]

Returns
  A dictionary with final metrics such as loss and accuracy of the reward model.

class imitation.algorithms.base.BaseImitationAlgorithm(*, custom_logger=None, allow_variable_horizon=False)

Bases: ABC

Base class for all imitation learning algorithms.

__init__(*, custom_logger=None, allow_variable_horizon=False)
  Creates an imitation learning algorithm.

Parameters
  • custom_logger (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.
  • allow_variable_horizon (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/getting-started/variable-horizon.html before overriding this.

allow_variable_horizon: bool
  If True, allow variable horizon trajectories; otherwise error if detected.
1.12 Train an Agent using Behavior Cloning

Behavior cloning is the most naive approach to imitation learning. We take the transitions of trajectories taken by some expert and use them as training samples to train a new policy. The method has many drawbacks and often does not work. However in this example, where we train an agent for the CartPole-v1 environment, it is feasible.

First we need some kind of expert in CartPole-v1 so we can sample some expert trajectories. For convenience we just train one using the stable-baselines3 library.

```python
import gym
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy

env = gym.make("CartPole-v1")
expert = PPO(
    policy=MlpPolicy,
    env=env,
    seed=0,
    batch_size=64,
    ent_coef=0.0,
    learning_rate=0.0003,
    n_epochs=10,
    n_steps=64,
)
expert.learn(1000)  # Note: set to 100000 to train a proficient expert
```

Let's quickly check if the expert is any good. We usually should be able to reach a reward of 500, which is the maximum achievable value.

```python
from stable_baselines3.common.evaluation import evaluate_policy

reward, _ = evaluate_policy(expert, env, 10)
print(reward)
```

62.8

Now we can use the expert to sample some trajectories. We flatten them right away since we are only interested in the individual transitions for behavior cloning. imitation comes with a number of helper functions that makes collecting those transitions really easy. First we collect 50 episode rollouts, then we flatten them to just the transitions that we need for training. Note that the rollout function requires a vectorized environment and needs the RolloutInfoWrapper around each of the environments.

```python
from imitation.data import rollout
from imitation.data.wrappers import RolloutInfoWrapper
```
from stable_baselines3.common.vec_env import DummyVecEnv
import numpy as np
rng = np.random.default_rng()
rollouts = rollout.rollout(expert,
    DummyVecEnv([lambda: RolloutInfoWrapper(env)]),
    rollout.make_sample_until(min_timesteps=None, min_episodes=50),
    rng=rng,
)
transitions = rollout.flatten_trajectories(rollouts)

Let's have a quick look at what we just generated using those library functions:

print(f"The `rollout` function generated a list of {len(rollouts)} {type(rollouts[0])}.
After flattening, this list is turned into a {type(transitions)} object containing
→{len(transitions)} transitions.
The transitions object contains arrays for: {', '.join(transitions.__dict__.keys())}."
")

The `rollout` function generated a list of 50 <class 'imitation.data.types.TrajectoryWithRew'>.
After flattening, this list is turned into a <class 'imitation.data.types.Transitions'> object containing 1723 transitions.
The transitions object contains arrays for: obs, acts, infos, next_obs, dones.

After we collected our transitions, it's time to set up our behavior cloning algorithm.

from imitation.algorithms import bc
bc_trainer = bc.BC(
    observation_space=env.observation_space,
    action_space=env.action_space,
    demonstrations=transitions,
    rng=rng,
)

As you can see the untrained policy only gets poor rewards:

reward_before_training, _ = evaluate_policy(bc_trainer.policy, env, 10)
print(f"Reward before training: {reward_before_training}"
)

Reward before training: 67.5

After training, we can match the rewards of the expert (500):

bc_trainer.train(n_epochs=1)
reward_after_training, _ = evaluate_policy(bc_trainer.policy, env, 10)
print(f"Reward after training: {reward_after_training}"
)
1.13 Train an Agent using the DAgger Algorithm

The DAgger algorithm is an extension of behavior cloning. In behavior cloning, the training trajectories are recorded directly from an expert. In DAgger, the learner generates the trajectories but an expert corrects the actions with the optimal actions in each of the visited states. This ensures that the state distribution of the training data matches that of the learner’s current policy.

First we need an expert to learn from:

```python
import gym
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy

env = gym.make("CartPole-v1")
expert = PPO(
    policy=MlpPolicy,
    env=env,
    seed=0,
    batch_size=64,
    ent_coef=0.0,
    learning_rate=0.0003,
    n_epochs=10,
    n_steps=64,
)
expert.learn(1000)  # Note: set to 100000 to train a proficient expert
```

Then we can construct a DAgger trainer and use it to train the policy on the cartpole environment.

```python
import tempfile
import gym
import numpy as np
from stable_baselines3.common.vec_env import DummyVecEnv
```

(continues on next page)
from imitation.algorithms import bc
from imitation.algorithms.dagger import SimpleDAGgerTrainer

venv = DummyVecEnv([lambda: gym.make("CartPole-v1")])

bc_trainer = bc.BC(
    observation_space=env.observation_space,
    action_space=env.action_space,
    rng=np.random.default_rng(),
)

with tempfile.TemporaryDirectory(prefix="dagger_example_") as tmpdir:
    print(tmpdir)
    dagger_trainer = SimpleDAGgerTrainer(
        venv=venv,
        scratch_dir=tmpdir,
        expert_policy=expert,
        bc_trainer=bc_trainer,
        rng=np.random.default_rng(),
    )

dagger_trainer.train(2000)

```
/tmp/dagger_example_yimeuguz
-----------------------------------
| batch_size | 32  |
| bc/        |     |
|            |     |
| batch      | 0   |
| ent_loss   | -0.000693 |
| entropy    | 0.693 |
| epoch      | 0   |
| l2_loss    | 0   |
| l2_norm    | 36.5 |
| loss       | 0.693 |
| neglogp    | 0.693 |
| prob_true_act | 0.5 |
| samples_so_far | 32  |
| rollout/   |     |
|            |     |
| return_max | 18  |
| return_mean| 16  |
| return_min | 14  |
| return_std | 1.41 |
-----------------------------------
```

(continues on next page)
Finally, the evaluation shows, that we actually trained a policy that solves the environment (500 is the max reward).
1.14 Train an Agent using Generative Adversarial Imitation Learning

The idea of generative adversarial imitation learning is to train a discriminator network to distinguish between expert trajectories and learner trajectories. The learner is trained using a traditional reinforcement learning algorithm such as PPO and is rewarded for trajectories that make the discriminator think that it was an expert trajectory.

As usual, we first need an expert. Note that we now use a variant of the CartPole environment from the seals package, which has fixed episode durations. Read more about why we do this here.

```python
import gym
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
import seals  # needed to load environments

env = gym.make("seals/CartPole-v0")
expert = PPO(
    policy=MlpPolicy,
    env=env,
    seed=0,
    batch_size=64,
    ent_coef=0.0,
    learning_rate=0.0003,
    n_epochs=10,
    n_steps=64,
)
expert.learn(1000)  # Note: set to 100000 to train a proficient expert
```

We generate some expert trajectories, that the discriminator needs to distinguish from the learner's trajectories.

```python
from imitation.data import rollout
from imitation.data.wrappers import RolloutInfoWrapper
from imitation.util import make_vec_env
from stable_baselines3.common.vec_env import DummyVecEnv
import numpy as np

rng = np.random.default_rng()
rollouts = rollout.rollout(
    expert,
    make_vec_env("seals/CartPole-v0"),
)  # (continues on next page)
```
Now we are ready to set up our GAIL trainer. Note, that the reward_net is actually the network of the discriminator. We evaluate the learner before and after training so we can see if it made any progress.

```python
from imitation.algorithms.adversarial.gail import GAIL
from imitation.rewards.reward_nets import BasicRewardNet
from imitation.util.networks import RunningNorm
from imitation.util.util import make_vec_env
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.vec_env import DummyVecEnv
import gym

venv = make_vec_env("seals/CartPole-v0", n_envs=8, rng=rng)
learner = PPO(
    env=venv,
    policy=MlpPolicy,
    batch_size=64,
    ent_coef=0.0,
    learning_rate=0.0003,
    n_epochs=10,
)
reward_net = BasicRewardNet(
    venv.observation_space, venv.action_space, normalize_input_layer=RunningNorm
)
gail_trainer = GAIL(
    demonstrations=rollouts,
    demo_batch_size=1024,
    gen_replay_buffer_capacity=2048,
    n_disc_updates_per_round=4,
    venv=venv,
    gen_algo=learner,
    reward_net=reward_net,
)

learner_rewards_before_training, _ = evaluate_policy(
    learner, venv, 100, return_episode_rewards=True
)
gail_trainer.train(20000)  # Note: set to 300000 for better results
learner_rewards_after_training, _ = evaluate_policy(
    learner, venv, 100, return_episode_rewards=True
)
```
| raw/ | |
| gen/rollout/ep_len_mean | 500 |
| gen/rollout/ep_rew_mean | 32.1 |
| gen/time/fps | 4856 |
| gen/time/iterations | 1 |
| gen/time/time_elapsed | 3 |
| gen/time/total_timesteps | 16384 |
| raw/ | |
| disc/disc_acc | 0.452 |
| disc/disc_acc_expert | 0.904 |
| disc/disc_acc_gen | 0 |
| disc/disc_entropy | 0.691 |
| disc/disc_loss | 0.701 |
| disc/disc_proportion_expert_pred | 0.952 |
| disc/disc_proportion_expert_true | 0.5 |
| disc/global_step | 1 |
| disc/n_expert | 1.02e+03 |
| disc/n_generated | 1.02e+03 |
| raw/ | |
| disc/disc_acc | 0.471 |
| disc/disc_acc_expert | 0.941 |
| disc/disc_acc_gen | 0 |
| disc/disc_entropy | 0.691 |
| disc/disc_loss | 0.699 |
| disc/disc_proportion_expert_pred | 0.971 |
| disc/disc_proportion_expert_true | 0.5 |
| disc/global_step | 1 |
| disc/n_expert | 1.02e+03 |
| disc/n_generated | 1.02e+03 |
| raw/ | |
| disc/disc_acc | 0.474 |
| disc/disc_acc_expert | 0.947 |
| disc/disc_acc_gen | 0 |
| disc/disc_entropy | 0.691 |
| disc/disc_loss | 0.696 |
| disc/disc_proportion_expert_pred | 0.974 |
| disc/disc_proportion_expert_true | 0.5 |
| disc/global_step | 1 |
| disc/n_expert | 1.02e+03 |
| disc/n_generated | 1.02e+03 |
| raw/ | |
| disc/disc_acc | 0.484 |
| disc/disc_acc_expert | 0.969 |
| disc/disc_acc_gen | 0 |

(continues on next page)
When we look at the histograms of rewards before and after learning, we can see that the learner is not perfect yet, but it made some progress at least. If not, just re-run the above cell.

```python
import matplotlib.pyplot as plt
import numpy as np

print(np.mean(learner_rewards_after_training))
print(np.mean(learner_rewards_before_training))

plt.hist(
    [learner_rewards_before_training, learner_rewards_after_training],
    label=['untrained', 'trained'],
)
plt.legend()
```
1.15 Train an Agent using Adversarial Inverse Reinforcement Learning

As usual, we first need an expert. Note that we now use a variant of the CartPole environment from the seals package, which has fixed episode durations. Read more about why we do this here.

```python
import gym
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
import seals  # needed to load environments

env = gym.make("seals/CartPole-v0")
expert = PPO(
    policy=MlpPolicy,
    env=env,
    seed=0,
    batch_size=64,
)
```

(continues on next page)
ent_coef=0.0,
learning_rate=0.0003,
n_epochs=10,
n_steps=64,
)
expert.learn(1000)  # Note: set to 100000 to train a proficient expert

We generate some expert trajectories, that the discriminator needs to distinguish from the learner’s trajectories.

```python
from imitation.data import rollout
from imitation.data.wrappers import RolloutInfoWrapper
from imitation.util.util import make_vec_env
from stable_baselines3.common.vec_env import DummyVecEnv
import numpy as np
rng = np.random.default_rng()
rollouts = rollout.rollout(
    expert,
    make_vec_env(
        "seals/CartPole-v0",
        n_envs=5,
        post_wrappers=[lambda env, _: RolloutInfoWrapper(env)],
        rng=rng,
    ),
    rollout.make_sample_until(min_timesteps=None, min_episodes=60),
    rng=rng,
)
```

Now we are ready to set up our AIRL trainer. Note, that the reward_net is actually the network of the discriminator. We evaluate the learner before and after training so we can see if it made any progress.

```python
from imitation.algorithms.adversarial.airl import AIRL
from imitation.rewards.reward_nets import BasicShapedRewardNet
from imitation.util.networks import RunningNorm
from imitation.util.util import make_vec_env
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
import gym
import seals

venv = make_vec_env("seals/CartPole-v0", n_envs=8, rng=rng)
learner = PPO(
    env=venv,
    policy=MlpPolicy,
    batch_size=64,
    ent_coef=0.0,
    learning_rate=0.0003,
    n_epochs=10,
)`
reward_net = BasicShapedRewardNet(
    venv.observation_space, venv.action_space, normalize_input_layer=RunningNorm
)

airl_trainer = AIRL(
    demonstrations=rollouts,
    demo_batch_size=1024,
    gen_replay_buffer_capacity=2048,
    n_disc_updates_per_round=4,
    venv=venv,
    gen_algo=learner,
    reward_net=reward_net,
)

learner_rewards_before_training, _ = evaluate_policy(
    learner, venv, 100, return_episode_rewards=True
)

airl_trainer.train(20000)  # Note: set to 300000 for better results

learner_rewards_after_training, _ = evaluate_policy(
    learner, venv, 100, return_episode_rewards=True
)
| disc/disc_proportion_expert_true  | 0.5 |
| disc/global_step                 | 1   |
| disc/n_expert                    | 1.02e+03 |
| disc/n_generated                 | 1.02e+03 |

<table>
<thead>
<tr>
<th>raw/</th>
</tr>
</thead>
<tbody>
<tr>
<td>disc/disc_acc</td>
</tr>
<tr>
<td>disc/disc_acc_expert</td>
</tr>
<tr>
<td>disc/disc_acc_gen</td>
</tr>
<tr>
<td>disc/disc_entropy</td>
</tr>
<tr>
<td>disc/disc_loss</td>
</tr>
<tr>
<td>disc/disc_proportion_expert_pred</td>
</tr>
<tr>
<td>disc/disc_proportion_expert_true</td>
</tr>
<tr>
<td>disc/global_step</td>
</tr>
<tr>
<td>disc/n_expert</td>
</tr>
<tr>
<td>disc/n_generated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>raw/</th>
</tr>
</thead>
<tbody>
<tr>
<td>disc/disc_acc</td>
</tr>
<tr>
<td>disc/disc_acc_expert</td>
</tr>
<tr>
<td>disc/disc_acc_gen</td>
</tr>
<tr>
<td>disc/disc_entropy</td>
</tr>
<tr>
<td>disc/disc_loss</td>
</tr>
<tr>
<td>disc/disc_proportion_expert_pred</td>
</tr>
<tr>
<td>disc/disc_proportion_expert_true</td>
</tr>
<tr>
<td>disc/global_step</td>
</tr>
<tr>
<td>disc/n_expert</td>
</tr>
<tr>
<td>disc/n_generated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mean/</th>
</tr>
</thead>
<tbody>
<tr>
<td>disc/disc_acc</td>
</tr>
<tr>
<td>disc/disc_acc_expert</td>
</tr>
<tr>
<td>disc/disc_acc_gen</td>
</tr>
<tr>
<td>disc/disc_entropy</td>
</tr>
<tr>
<td>disc/disc_loss</td>
</tr>
<tr>
<td>disc/disc_proportion_expert_pred</td>
</tr>
<tr>
<td>disc/disc_proportion_expert_true</td>
</tr>
<tr>
<td>disc/global_step</td>
</tr>
<tr>
<td>disc/n_expert</td>
</tr>
<tr>
<td>disc/n_generated</td>
</tr>
<tr>
<td>gen/rollout/ep_len_mean</td>
</tr>
<tr>
<td>gen/rollout/ep_rew_mean</td>
</tr>
<tr>
<td>gen/time/fps</td>
</tr>
<tr>
<td>gen/time/iterations</td>
</tr>
<tr>
<td>gen/time/time_elapsed</td>
</tr>
<tr>
<td>gen/time/total_timesteps</td>
</tr>
<tr>
<td>gen/train/approx_kl</td>
</tr>
<tr>
<td>gen/train/clip_fraction</td>
</tr>
<tr>
<td>gen/train/clip_range</td>
</tr>
</tbody>
</table>
### Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen/train/entropy_loss</td>
<td>-0.688</td>
</tr>
<tr>
<td>gen/train/explained_variance</td>
<td>0.0767</td>
</tr>
<tr>
<td>gen/train/learning_rate</td>
<td>0.0003</td>
</tr>
<tr>
<td>gen/train/loss</td>
<td>11.1</td>
</tr>
<tr>
<td>gen/train/n_updates</td>
<td>10</td>
</tr>
<tr>
<td>gen/train/policy_gradient_loss</td>
<td>-0.00191</td>
</tr>
<tr>
<td>gen/train/value_loss</td>
<td>45.2</td>
</tr>
</tbody>
</table>

When we look at the histograms of rewards before and after learning, we can see that the learner is not perfect yet, but it made some progress at least. If not, just re-run the above cell.

```python
import matplotlib.pyplot as plt
import numpy as np

print(np.mean(learner_rewards_after_training))
print(np.mean(learner_rewards_before_training))

plt.hist([learner_rewards_before_training, learner_rewards_after_training],
         label=['untrained', 'trained'],
         )
plt.legend()
plt.show()
```

8.32
8.46

![Histogram of rewards](image)
1.16 Learning a Reward Function using Preference Comparisons

The preference comparisons algorithm learns a reward function by comparing trajectory segments to each other. To set up the preference comparisons algorithm, we first need to set up a lot of its internals beforehand:

```python
import random
from imitation.algorithms import preference_comparisons
from imitation.rewards.reward_nets import BasicRewardNet
from imitation.util.networks import RunningNorm
from imitation.util.util import make_vec_env
from imitation.policies.base import FeedForward32Policy, NormalizeFeaturesExtractor
import gym
from stable_baselines3 import PPO
import numpy as np

rng = np.random.default_rng(0)
venv = make_vec_env("Pendulum-v1", rng=rng)

reward_net = BasicRewardNet(
    venv.observation_space, venv.action_space, normalize_input_layer=RunningNorm
)

fragmenter = preference_comparisons.RandomFragmenter(
    warning_threshold=0,
    rng=rng,
)

gatherer = preference_comparisons.SyntheticGatherer(rng=rng)
preference_model = preference_comparisons.PreferenceModel(reward_net)
reward_trainer = preference_comparisons.BasicRewardTrainer(
    preference_model=preference_model,
    loss=preference_comparisons.CrossEntropyRewardLoss(),
    epochs=3,
    rng=rng,
)

agent = PPO(
    policy=FeedForward32Policy,
    policy_kwargs=dict(
        features_extractor_class=NormalizeFeaturesExtractor,
        features_extractor_kwargs=dict(normalize_class=RunningNorm),
    ),
    env=venv,
    seed=0,
    n_steps=2048 // venv.num_envs,
    batch_size=64,
    ent_coef=0.0,
    learning_rate=0.0003,
    n_epochs=10,
)
```

(continues on next page)
trajectory_generator = preference_comparisons.AgentTrainer(
    algorithm=agent,
    reward_fn=reward_net,
    venv=venv,
    exploration_frac=0.0,
    rng=rng,
)

pref_comparisons = preference_comparisons.PreferenceComparisons(
    trajectory_generator,
    reward_net,
    num_iterations=5,
    fragmenter=fragmenter,
    preference_gatherer=gatherer,
    reward_trainer=reward_trainer,
    fragment_length=100,
    transition_oversampling=1,
    initial_comparison_frac=0.1,
    allow_variable_horizon=False,
    initial_epoch_multiplier=1,
)

Then we can start training the reward model. Note that we need to specify the total timesteps that the agent should be
trained and how many fragment comparisons should be made.

pref_comparisons.train(
    total_timesteps=5_000,  # For good performance this should be 1_000_000
    total_comparisons=200,  # For good performance this should be 5_000
)

Query schedule: [20, 51, 41, 34, 29, 25]
Collecting 40 fragments (4000 transitions)
Requested 4000 transitions but only 0 in buffer. Sampling 4000 additional transitions.
Creating fragment pairs
Gathering preferences
Dataset now contains 20 comparisons
Training agent for 1000 timesteps

<table>
<thead>
<tr>
<th>raw/</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>agent/rollout/ep_len_mean</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>agent/rollout/ep_rew_mean</td>
<td>-1.31e+03</td>
<td></td>
</tr>
<tr>
<td>agent/rollout/ep_rew_wrapped_mean</td>
<td>-25.9</td>
<td></td>
</tr>
<tr>
<td>agent/time/fps</td>
<td>4403</td>
<td></td>
</tr>
<tr>
<td>agent/time/iterations</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>agent/time/time_elapsed</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>agent/time/total_timesteps</td>
<td>2048</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mean/</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>agent/rollout/ep_len_mean</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>
Collecting 102 fragments (10200 transitions)
Requested 10200 transitions but only 1600 in buffer. Sampling 8600 additional transitions.
Creating fragment pairs
Gathering preferences
Dataset now contains 71 comparisons
Training agent for 1000 timesteps

| raw/ | | |
| agent/rollout/ep_len_mean | 200 | |
| agent/rollout/ep_rew_mean | -1.33e+03 | |
| agent/rollout/ep_rew_wrapped_mean | -28.7 | |
| agent/time/fps | 4424 | |
| agent/time/iterations | 1 | |
| agent/time/time_elapsed | 0 | |
| agent/time/total_timesteps | 4096 | |
| agent/train/approx_kl | 0.0027488796 | |
| agent/train/clip_fraction | 0.019 | |
| agent/train/clip_range | 0.2 | |
| agent/train/entropy_loss | -1.42 | |
| agent/train/explained_variance | 0.0125 | |
| agent/train/learning_rate | 0.0003 | |
| agent/train/loss | 0.131 | |
| agent/train/n_updates | 10 | |
| agent/train/policy_gradient_loss | -0.00279 | |
| agent/std | 0.998 | |
| agent/train/value_loss | 2.42 | |
| preferences/entropy | 0.038 | |
| reward/epoch-0/train/accuracy | 0.15 | |
| reward/epoch-0/train/gt_reward_loss | 0.0553 | |
| reward/epoch-0/train/loss | 1.06 | |
| reward/epoch-1/train/accuracy | 0.45 | |
| reward/epoch-1/train/gt_reward_loss | 0.0553 | |
| reward/epoch-1/train/loss | 0.654 | |
| reward/epoch-2/train/accuracy | 0.8 | |
| reward/epoch-2/train/gt_reward_loss | 0.0553 | |
| reward/epoch-2/train/loss | 0.452 | |

| reward/ | | |
| final/train/accuracy | 0.8 | |
| final/train/gt_reward_loss | 0.0553 | |
| final/train/loss | 0.452 | |
| agent/train/explained_variance | 0.0125 |
| agent/train/learning_rate     | 0.0003 |
| agent/train/loss              | 0.131 |
| agent/train/n_updates         | 10    |
| agent/train/policy_gradient_loss | -0.00279 |
| agent/train/std               | 0.998 |
| agent/train/value_loss        | 2.42  |

```
-------------------------------------------------------
| mean/ |
| agent/rollout/ep_len_mean  | 200   |
| agent/rollout/ep_rew_mean  | -1.33e+03 |
| agent/rollout/ep_rew_wrapped_mean | -28.7 |
| agent/time/fps             | 4.42e+03 |
| agent/time/iterations      | 1     |
| agent/time/time_elapsed    | 0     |
| agent/time/total_timesteps | 4.1e+03 |
| agent/train/approx_kl      | 0.00258 |
| agent/train/clip_fraction  | 0.0167 |
| agent/train/clip_range     | 0.2   |
| agent/train/entropy_loss   | -1.42 |
| agent/train/explained_variance | 0.362 |
| agent/train/learning_rate  | 0.0003 |
| agent/train/loss           | 0.277 |
| agent/train/n_updates      | 20    |
| agent/train/policy_gradient_loss | -0.00205 |
| agent/train/std            | 1.01  |
| agent/train/value_loss     | 1.12  |
| preferences/entropy        | 0.000287 |
| reward/epoch-0/train/accuracy | 0.917 |
| reward/epoch-0/train/gt_reward_loss | 0.0115 |
| reward/epoch-0/train/loss  | 0.236 |
| reward/epoch-1/train/accuracy | 0.969 |
| reward/epoch-1/train/gt_reward_loss | 0.0115 |
| reward/epoch-1/train/loss  | 0.164 |
| reward/epoch-2/train/accuracy | 0.969 |
| reward/epoch-2/train/gt_reward_loss | 0.0115 |
| reward/epoch-2/train/loss  | 0.134 |
| reward/ final/train/accuracy | 0.969 |
| final/train/gt_reward_loss | 0.0115 |
| final/train/loss           | 0.134 |
```

Collecting 82 fragments (8200 transitions)
Requested 8200 transitions but only 1600 in buffer. Sampling 6600 additional transitions.
Creating fragment pairs
Gathering preferences
Dataset now contains 112 comparisons
Training agent for 1000 timesteps

```
(continues on next page)
```
<table>
<thead>
<tr>
<th></th>
<th>Mean/</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent/rollout/ep_len_mean</td>
<td>200</td>
</tr>
<tr>
<td>agent/rollout/ep_rew_mean</td>
<td>-1.28e+03</td>
</tr>
<tr>
<td>agent/rollout/ep_rew_wrapped_mean</td>
<td>-28.7</td>
</tr>
<tr>
<td>agent/time/fps</td>
<td>4.39e+03</td>
</tr>
<tr>
<td>agent/time/iterations</td>
<td>1</td>
</tr>
<tr>
<td>agent/time/time_elapsed</td>
<td>0</td>
</tr>
<tr>
<td>agent/time/total_timesteps</td>
<td>6.14e+03</td>
</tr>
<tr>
<td>agent/train/approx_kl</td>
<td>0.0033</td>
</tr>
<tr>
<td>agent/train/clip_fraction</td>
<td>0.0284</td>
</tr>
<tr>
<td>agent/train/clip_range</td>
<td>0.2</td>
</tr>
<tr>
<td>agent/train/entropy_loss</td>
<td>-1.43</td>
</tr>
<tr>
<td>agent/train/explained_variance</td>
<td>0.675</td>
</tr>
<tr>
<td>agent/train/learning_rate</td>
<td>0.0003</td>
</tr>
<tr>
<td>agent/train/loss</td>
<td>0.216</td>
</tr>
<tr>
<td>agent/train/n_updates</td>
<td>30</td>
</tr>
<tr>
<td>agent/train/policy_gradient_loss</td>
<td>-0.00386</td>
</tr>
<tr>
<td>agent/train/std</td>
<td>1.01</td>
</tr>
<tr>
<td>agent/train/value_loss</td>
<td>1.09</td>
</tr>
<tr>
<td>preferences/entropy</td>
<td>2.72e-09</td>
</tr>
<tr>
<td>reward/epoch-0/train/accuracy</td>
<td>0.961</td>
</tr>
<tr>
<td>reward/epoch-0/train/gt_reward_loss</td>
<td>0.00866</td>
</tr>
<tr>
<td>reward/epoch-0/train/loss</td>
<td>0.117</td>
</tr>
<tr>
<td>reward/epoch-1/train/accuracy</td>
<td>0.969</td>
</tr>
<tr>
<td>reward/epoch-1/train/gt_reward_loss</td>
<td>0.00866</td>
</tr>
<tr>
<td>reward/epoch-1/train/loss</td>
<td>0.1</td>
</tr>
<tr>
<td>reward/epoch-2/train/accuracy</td>
<td>0.969</td>
</tr>
<tr>
<td>reward/epoch-2/train/gt_reward_loss</td>
<td>0.00866</td>
</tr>
<tr>
<td>reward/epoch-2/train/loss</td>
<td>0.0887</td>
</tr>
<tr>
<td>reward/final/train/accuracy</td>
<td>0.969</td>
</tr>
<tr>
<td>reward/final/train/gt_reward_loss</td>
<td>0.00866</td>
</tr>
<tr>
<td>reward/final/train/loss</td>
<td>0.0887</td>
</tr>
</tbody>
</table>
Collecting 68 fragments (6800 transitions)
Requested 6800 transitions but only 1600 in buffer. Sampling 5200 additional transitions.
Creating fragment pairs
Gathering preferences
Dataset now contains 146 comparisons
Training agent for 1000 timesteps

| raw/ | |
|------------------------------------------------------|
| agent/rollout/ep_len_mean | 200 |
| agent/rollout/ep_rew_mean | -1.3e+03 |
| agent/rollout/ep_rew_wrapped_mean | -27.8 |
| agent/time/fps | 4416 |
| agent/time/iterations | 1 |
| agent/time/time_elapsed | 0 |
| agent/time/total_timesteps | 8192 |
| agent/train/approx_kl | 0.003300739 |
| agent/train/clip_fraction | 0.0284 |
| agent/train/clip_range | 0.2 |
| agent/train/entropy_loss | -1.43 |
| agent/train/explained_variance | 0.675 |
| agent/train/learning_rate | 0.0003 |
| agent/train/loss | 0.216 |
| agent/train/n_updates | 30 |
| agent/train/policy_gradient_loss | -0.00386 |
| agent/train/std | 1.01 |
| agent/train/value_loss | 1.09 |

| mean/ | |
|------------------------------------------------------|
| agent/rollout/ep_len_mean | 200 |
| agent/rollout/ep_rew_mean | -1.3e+03 |
| agent/rollout/ep_rew_wrapped_mean | -27.8 |
| agent/time/fps | 4.42e+03 |
| agent/time/iterations | 1 |
| agent/time/time_elapsed | 0 |
| agent/time/total_timesteps | 8.19e+03 |
| agent/train/approx_kl | 0.00418 |
| agent/train/clip_fraction | 0.0245 |
| agent/train/clip_range | 0.2 |
| agent/train/entropy_loss | -1.42 |
| agent/train/explained_variance | 0.698 |
| agent/train/learning_rate | 0.0003 |
| agent/train/loss | 0.795 |
| agent/train/n_updates | 40 |
| agent/train/policy_gradient_loss | -0.00298 |
| agent/train/std | 0.988 |
| agent/train/value_loss | 1.77 |
| preferences/entropy | 0.00173 |
| reward/epoch-0/train/accuracy | 0.964 |
| reward/epoch-0/train/gt_reward_loss | 0.00699 |
| reward/epoch-0/train/loss | 0.0962 |
| reward/epoch-1/train/accuracy | 0.969 |
| reward/epoch-1/train/gt_reward_loss | 0.00699 |
| reward/epoch-1/train/loss | 0.0816 |
| reward/epoch-2/train/accuracy | 0.969 |
| reward/epoch-2/train/gt_reward_loss | 0.007 |
| reward/epoch-2/train/loss | 0.0763 |
| reward/ |
| final/train/accuracy | 0.969 |
| final/train/gt_reward_loss | 0.007 |
| final/train/loss | 0.0763 |

Collecting 58 fragments (5800 transitions)

Requested 5800 transitions but only 1600 in buffer. Sampling 4200 additional transitions.

Creating fragment pairs

Gathering preferences

Dataset now contains 175 comparisons

Training agent for 1000 timesteps

| raw/ |
| agent/rollout/ep_len_mean | 200 |
| agent/rollout/ep_rew_mean | -1.28e+03 |
| agent/rollout/ep_rew_wrapped_mean | -25.7 |
| agent/time/fps | 4420 |
| agent/time/iterations | 1 |
| agent/time/time_elapsed | 0 |
| agent/time/total_timesteps | 10240 |
| agent/train/approx_kl | 0.004183973 |
| agent/train/clip_fraction | 0.0245 |
| agent/train/clip_range | 0.2 |
| agent/train/entropy_loss | -1.42 |
| agent/train/explained_variance | 0.698 |
| agent/train/learning_rate | 0.0003 |
| agent/train/loss | 0.795 |
| agent/train/n_updates | 40 |
| agent/train/policy_gradient_loss | -0.00298 |
| agent/train/std | 0.988 |
| agent/train/value_loss | 1.77 |

| mean/ |
| agent/rollout/ep_len_mean | 200 |
| agent/rollout/ep_rew_mean | -1.28e+03 |
| agent/rollout/ep_rew_wrapped_mean | -25.7 |
| agent/time/fps | 4.42e+03 |
| agent/time/iterations | 1 |
| agent/time/time_elapsed | 0 |
| agent/time/total_timesteps | 1.02e+04 |
| agent/train/approx_kl | 0.00785 |
| agent/train/clip_fraction | 0.0557 |
| agent/train/clip_range | 0.2 |
| agent/train/entropy_loss | -1.39 |
| agent/train/explained_variance | 0.849 |
### 1.16. Learning a Reward Function using Preference Comparisons

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent/train/learning_rate</td>
<td>0.0003</td>
</tr>
<tr>
<td>agent/train/loss</td>
<td>0.493</td>
</tr>
<tr>
<td>agent/train/n_updates</td>
<td>50</td>
</tr>
<tr>
<td>agent/train/policy_gradient_loss</td>
<td>-0.00619</td>
</tr>
<tr>
<td>agent/train/std</td>
<td>0.945</td>
</tr>
<tr>
<td>agent/train/value_loss</td>
<td>1.24</td>
</tr>
<tr>
<td>preferences/entropy</td>
<td>0.0064</td>
</tr>
<tr>
<td>reward/epoch-0/train/accuracy</td>
<td>0.947</td>
</tr>
<tr>
<td>reward/epoch-0/train/gt_reward_loss</td>
<td>0.00607</td>
</tr>
<tr>
<td>reward/epoch-0/train/loss</td>
<td>0.1</td>
</tr>
<tr>
<td>reward/epoch-1/train/accuracy</td>
<td>0.969</td>
</tr>
<tr>
<td>reward/epoch-1/train/gt_reward_loss</td>
<td>0.00607</td>
</tr>
<tr>
<td>reward/epoch-1/train/loss</td>
<td>0.0897</td>
</tr>
<tr>
<td>reward/epoch-2/train/accuracy</td>
<td>0.968</td>
</tr>
<tr>
<td>reward/epoch-2/train/gt_reward_loss</td>
<td>0.0124</td>
</tr>
<tr>
<td>reward/epoch-2/train/loss</td>
<td>0.0939</td>
</tr>
<tr>
<td>final/train/accuracy</td>
<td>0.968</td>
</tr>
<tr>
<td>final/train/gt_reward_loss</td>
<td>0.0124</td>
</tr>
<tr>
<td>final/train/loss</td>
<td>0.0939</td>
</tr>
</tbody>
</table>

Collecting 50 fragments (5000 transitions)

Requested 5000 transitions but only 1600 in buffer. Sampling 3400 additional transitions.

Creating fragment pairs
Gathering preferences
Dataset now contains 200 comparisons
Training agent for 1000 timesteps

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent/rollout/ep_len_mean</td>
<td>200</td>
</tr>
<tr>
<td>agent/rollout/ep_rew_mean</td>
<td>-1.27e+03</td>
</tr>
<tr>
<td>agent/rollout/ep_rew_wrapped_mean</td>
<td>-24</td>
</tr>
<tr>
<td>agent/time/fps</td>
<td>4428</td>
</tr>
<tr>
<td>agent/time/iterations</td>
<td>1</td>
</tr>
<tr>
<td>agent/time/time_elapsed</td>
<td>0</td>
</tr>
<tr>
<td>agent/time/total_timesteps</td>
<td>12288</td>
</tr>
<tr>
<td>agent/train/approx_kl</td>
<td>0.00785001</td>
</tr>
<tr>
<td>agent/train/clip_fraction</td>
<td>0.0557</td>
</tr>
<tr>
<td>agent/train/clip_range</td>
<td>0.2</td>
</tr>
<tr>
<td>agent/train/entropy_loss</td>
<td>-1.39</td>
</tr>
<tr>
<td>agent/train/explained_variance</td>
<td>0.849</td>
</tr>
<tr>
<td>agent/train/learning_rate</td>
<td>0.0003</td>
</tr>
<tr>
<td>agent/train/loss</td>
<td>0.493</td>
</tr>
<tr>
<td>agent/train/n_updates</td>
<td>50</td>
</tr>
<tr>
<td>agent/train/policy_gradient_loss</td>
<td>-0.00619</td>
</tr>
<tr>
<td>agent/train/std</td>
<td>0.945</td>
</tr>
<tr>
<td>agent/train/value_loss</td>
<td>1.24</td>
</tr>
</tbody>
</table>
After we trained the reward network using the preference comparisons algorithm, we can wrap our environment with that learned reward.

```python
from imitation.rewards.reward_wrapper import RewardVecEnvWrapper

learned_reward_venv = RewardVecEnvWrapper(venv, reward_net.predict)
```

Now we can train an agent, that only sees those learned reward.

```python
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy

learner = PPO(
    policy=MlpPolicy,
    env=learned_reward_venv,
    seed=0,
    batch_size=64,
)```
ent_coef=0.0,
learning_rate=0.0003,
n_epochs=10,
n_steps=64,
)
learner.learn(1000)  # Note: set to 100000 to train a proficient expert

<stable_baselines3.ppo.ppo.PPO at 0x7f581773a9a0>

Then we can evaluate it using the original reward.

from stable_baselines3.common.evaluation import evaluate_policy

reward, _ = evaluate_policy(learner.policy, venv, 10)
print(reward)

-1064.3757024000001

download this notebook here

### 1.17 Learning a Reward Function using Preference Comparisons on Atari

In this case, we will use a convolutional neural network for our policy and reward model. We will also shape the learned reward model with the policy’s learned value function, since these shaped rewards will be more informative for training - incentivizing agents to move to high-value states. In the interests of execution time, we will only do a little bit of training - much less than in the previous preference comparison notebook. To run this notebook, be sure to install the atari extras, for example by running `pip install imitation[atari]`.

First, we will set up the environment, reward network, et cetera.

```python
import torch as th
import gym
from gym.wrappers import TimeLimit
import numpy as np

from seals.util import AutoResetWrapper

from stable_baselines3 import PPO
from stable_baselines3.common.atari_wrappers import AtariWrapper
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.vec_env import VecFrameStack
from stable_baselines3.ppo import CnnPolicy

from imitation.algorithms import preference_comparisons
from imitation.data.wrappers import RolloutInfoWrapper
from imitation.policies.base import NormalizeFeaturesExtractor
from imitation.rewards.reward_nets import CnnRewardNet
```

(continues on next page)
device = th.device("cuda" if th.cuda.is_available() else "cpu")

rng = np.random.default_rng()

# Here we ensure that our environment has constant-length episodes by resetting it when done, and running until 100 timesteps have elapsed.
# For real training, you will want a much longer time limit.
def constant_length_asteroids(num_steps):
    atari_env = gym.make("AsteroidsNoFrameskip-v4")
    preprocessed_env = AtariWrapper(atari_env)
    endless_env = AutoResetWrapper(preprocessed_env)
    limited_env = TimeLimit(endless_env, max_episode_steps=num_steps)
    return RolloutInfoWrapper(limited_env)

# For real training, you will want a vectorized environment with 8 environments in parallel.
# This can be done by passing in n_envs=8 as an argument to make_vec_env.
venv = make_vec_env(constant_length_asteroids, env_kwargs={"num_steps": 100})
venv = VecFrameStack(venv, n_stack=4)

reward_net = CnnRewardNet(
    venv.observation_space,
    venv.action_space,
).to(device)

fragmenter = preference_comparisons.RandomFragmenter(warning_threshold=0, rng=rng)
gatherer = preference_comparisons.SyntheticGatherer(rng=rng)
preference_model = preference_comparisons.PreferenceModel(reward_net)
reward_trainer = preference_comparisons.BasicRewardTrainer(
    preference_model=preference_model,
    loss=preference_comparisons.CrossEntropyRewardLoss(),
    epochs=3,
    rng=rng,
)

agent = PPO(
    policy=CnnPolicy,
    env=venv,
    seed=0,
    n_steps=16,  # To train on atari well, set this to 128
    batch_size=16,  # To train on atari well, set this to 256
    ent_coef=0.01,
    learning_rate=0.00025,
    n_epochs=4,
)

trajectory_generator = preference_comparisons.AgentTrainer(
    algorithm=agent,
    reward_fn=reward_net,
    venv=venv,
    exploration_frac=0.0,
)
rng=rng,
)

pref_comparisons = preference_comparisons.PreferenceComparisons(
    trajectory_generator,
    reward_net,
    num_iterations=2,
    fragmenter=fragmenter,
    preference_gatherer=gatherer,
    reward_trainer=reward_trainer,
    fragment_length=10,
    transition_oversampling=1,
    initial_comparison_frac=0.1,
    allow_variable_horizon=False,
    initial_epoch_multiplier=1,
)

We are now ready to train the reward model.

pref_comparisons.train(
    total_timesteps=16,
    total_comparisons=15,
)

<table>
<thead>
<tr>
<th>raw/</th>
<th></th>
<th>mean/</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>agent/rollout/ep_rew_wrapped_mean</td>
<td>-0.886</td>
<td>agent/rollout/ep_rew_wrapped_mean</td>
<td>-0.886</td>
</tr>
<tr>
<td>agent/time/fps</td>
<td>165</td>
<td>agent/time/fps</td>
<td>165</td>
</tr>
<tr>
<td>agent/time/iterations</td>
<td>1</td>
<td>agent/time/iterations</td>
<td>1</td>
</tr>
<tr>
<td>agent/time/time_elapsed</td>
<td>0</td>
<td>agent/time/time_elapsed</td>
<td>0</td>
</tr>
<tr>
<td>agent/time/total_timesteps</td>
<td>16</td>
<td>agent/time/total_timesteps</td>
<td>16</td>
</tr>
<tr>
<td>agent/train/approx_kl</td>
<td>0.000151</td>
<td>agent/train/approx_kl</td>
<td>0.000151</td>
</tr>
<tr>
<td>agent/train/clip_fraction</td>
<td>0</td>
<td>agent/train/clip_fraction</td>
<td>0</td>
</tr>
<tr>
<td>agent/train/clip_range</td>
<td>0.2</td>
<td>agent/train/clip_range</td>
<td>0.2</td>
</tr>
<tr>
<td>agent/train/entropy_loss</td>
<td>-2.64</td>
<td>agent/train/entropy_loss</td>
<td>-2.64</td>
</tr>
<tr>
<td>agent/train/explained_variance</td>
<td>-0.0054</td>
<td>agent/train/explained_variance</td>
<td>-0.0054</td>
</tr>
<tr>
<td>agent/train/learning_rate</td>
<td>0.00025</td>
<td>agent/train/learning_rate</td>
<td>0.00025</td>
</tr>
</tbody>
</table>

| agent/train/loss | -0.0378 |
| agent/train/_n_updates | 4 |
| agent/train/policy_gradient_loss | -0.00735 |
| agent/train/value_loss | 0.00892 |
| preferences/entropy | 0.582 |
| reward/epoch-0/train/accuracy | 0 |
| reward/epoch-0/train/gt_reward_loss | 0.313 |
| reward/epoch-0/train/loss | 0.862 |
| reward/epoch-1/train/accuracy | 0 |
| reward/epoch-1/train/gt_reward_loss | 0.313 |
| reward/epoch-1/train/loss | 0.806 |
| reward/epoch-2/train/accuracy | 0 |
| reward/epoch-2/train/gt_reward_loss | 0.313 |
| reward/epoch-2/train/loss | 0.753 |
| reward/ |
| final/train/accuracy | 0 |
| final/train/gt_reward_loss | 0.313 |
| final/train/loss | 0.753 |

Collecting 18 fragments (180 transitions)
Requested 180 transitions but only 0 in buffer. Sampling 180 additional transitions.
Creating fragment pairs
Gathering preferences
Dataset now contains 10 comparisons
Training agent for 8 timesteps

| raw/ |
| agent/rollout/ep_rew_wrapped_mean | -0.572 |
| agent/time/fps | 150 |
| agent/time/iterations | 1 |
| agent/time/time_elapsed | 0 |
| agent/time/total_timesteps | 32 |
| agent/train/approx_kl | 0.000125076622 |
| agent/train/clip_fraction | 0 |
| agent/train/clip_range | 0.2 |
| agent/train/entropy_loss | -2.64 |
| agent/train/explained_variance | -0.0054 |
| agent/train/learning_rate | 0.00025 |
| agent/train/loss | -0.0378 |
| agent/train/n_updates | 4 |
| agent/train/policy_gradient_loss | -0.00735 |
| agent/train/value_loss | 0.00892 |

| mean/ |
| agent/rollout/ep_rew_wrapped_mean | -0.572 |
| agent/time/fps | 150 |
| agent/time/iterations | 1 |
| agent/time/time_elapsed | 0 |
| agent/time/total_timesteps | 32 |
| agent/train/approx_kl | 0.000125076622 |
| agent/train/clip_fraction | 0 |

(continues on next page)
| agent/train/clip_range | 0.2 |
| agent/train/entropy_loss | -2.64 |
| agent/train/explained_variance | -0.617 |
| agent/train/learning_rate | 0.00025 |
| agent/train/loss | -0.035 |
| agent/train/n_updates | 8 |
| agent/train/policy_gradient_loss | -0.0052 |
| agent/train/value_loss | 0.014 |
| preferences/entropy | 0.644 |
| reward/epoch-0/train/accuracy | 0.3 |
| reward/epoch-0/train/gt_reward_loss | 0.503 |
| reward/epoch-0/train/loss | 0.733 |
| reward/epoch-1/train/accuracy | 0.4 |
| reward/epoch-1/train/gt_reward_loss | 0.503 |
| reward/epoch-1/train/loss | 0.724 |
| reward/epoch-2/train/accuracy | 0.5 |
| reward/epoch-2/train/gt_reward_loss | 0.503 |
| reward/epoch-2/train/loss | 0.715 |
| reward/final/train/accuracy | 0.5 |
| reward/final/train/gt_reward_loss | 0.503 |
| reward/final/train/loss | 0.715 |

Collecting 10 fragments (100 transitions)
Requested 100 transitions but only 0 in buffer. Sampling 100 additional transitions.
Creating fragment pairs
Gathering preferences
Dataset now contains 15 comparisons
Training agent for 8 timesteps

| raw/ |
| agent/rollout/ep_rew_wrapped_mean | -0.183 |
| agent/time/fps | 167 |
| agent/time/iterations | 1 |
| agent/time/time_elapsed | 0 |
| agent/time/total_timesteps | 48 |
| agent/train/approx_kl | 0.000119797885 |
| agent/train/clip_fraction | 0 |
| agent/train/clip_range | 0.2 |
| agent/train/entropy_loss | -2.64 |
| agent/train/explained_variance | -0.617 |
| agent/train/learning_rate | 0.00025 |
| agent/train/loss | -0.035 |
| agent/train/n_updates | 8 |
| agent/train/policy_gradient_loss | -0.0052 |
| agent/train/value_loss | 0.014 |

| mean/ |
| agent/rollout/ep_rew_wrapped_mean | -0.183 |
| agent/time/fps | 167 |
| agent/time/iterations | 1 |

1.17. Learning a Reward Function using Preference Comparisons on Atari
We can now wrap the environment with the learned reward model, shaped by the policy’s learned value function. Note that if we were training this for real, we would want to normalize the output of the reward net as well as the value function, to ensure their values are on the same scale. To do this, use the `NormalizedRewardNet` class from `src/imitation/rewards/reward_nets.py` on `reward_net`, and modify the potential to add a `RunningNorm` module from `src/imitation/util/networks.py`.

```python
from imitation.rewards.reward_nets import ShapedRewardNet, cnn_transpose
from imitation.rewards.reward_wrapper import RewardVecEnvWrapper

def value_potential(state):
    state_ = cnn_transpose(state)
    return agent.policy.predict_values(state_)

shaped_reward_net = ShapedRewardNet(
    base=reward_net,
    potential=value_potential,
    discount_factor=0.99,
)
learned_reward_venv = RewardVecEnvWrapper(venv, shaped_reward_net.predict)
```

Next, we train an agent that sees only the shaped, learned reward.
learner = PPO(
    policy=CnnPolicy,
    env=learned_reward_venv,
    seed=0,
    batch_size=64,
    ent_coef=0.0,
    learning_rate=0.0003,
    n_epochs=10,
    n_steps=64,
)
learner.learn(1000)

We now evaluate the learner using the original reward.

```python
from stable_baselines3.common.evaluation import evaluate_policy

reward, _ = evaluate_policy(learner.policy, venv, 10)
print(reward)
```

```none
0.3
```

### 1.18 Generating rollouts

When generating rollouts in image environments, be sure to use the agent’s `get_env()` function rather than using the original environment.

The learner re-arranges the observations space to put the channel environment in the first dimension, and `get_env()` will correctly provide a wrapped environment doing this.

```python
from imitation.data import rollout

rollouts = rollout.rollout(
    learner,
    # Note that passing venv instead of agent.get_env()
    # here would fail.
    learner.get_env(),
    rollout.make_sample_until(min_timesteps=None, min_episodes=3),
    rng=rng,
)
```

download this notebook here
1.19 Learn a Reward Function using Maximum Conditional Entropy Inverse Reinforcement Learning

Here, we’re going to take a tabular environment with a pre-defined reward function, Cliffworld, and solve for the optimal policy. We then generate demonstrations from this policy, and use them to learn an approximation to the true reward function with MCE IRL. Finally, we directly compare the learned reward to the ground-truth reward (which we have access to in this example).

Cliffworld is a POMDP, and its “observations” consist of the (partial) observations proper and the (full) hidden environment state. We use `DictExtractWrapper` to extract only the hidden states from the environment, turning it into a fully observable MDP to make computing the optimal policy easy.

```python
from functools import partial
from seals import base_envs
from seals.diagnostics.cliff_world import CliffWorldEnv
from stable_baselines3.common.vec_env import DummyVecEnv
import numpy as np
from imitation.algorithms.mce_irl import (MCEIRL,
                                           mce_occupancy_measures,
                                           mce_partition_fh,
                                           TabularPolicy,
                                       )
from imitation.data import rollout
from imitation.rewards import reward_nets

env_creator = partial(CliffWorldEnv, height=4, horizon=8, width=7, use_xy_obs=True)
env_single = env_creator()

state_env_creator = lambda: base_envs.ExposePOMDPStateWrapper(env_creator())

# This is just a vectorized environment because `generate_trajectories` expects one
state_venv = DummyVecEnv([[state_env_creator] * 4])
```

Then we derive an expert policy using Bellman backups. We analytically compute the occupancy measures, and also sample some expert trajectories.

```python
_, _, pi = mce_partition_fh(env_single)
_, om = mce_occupancy_measures(env_single, pi=pi)

rng = np.random.default_rng()
expert = TabularPolicy(
    state_space=env_single.state_space,
    action_space=env_single.action_space,
    pi=pi,
    rng=rng,
)

expert_trajs = rollout.generate_trajectories(
```
(continued from previous page)

```python
policy=expert,
venv=state_venv,
sample_until=rollout.make_min_timesteps(5000),
rng=rng,
)

print("Expert stats: ", rollout.rollout_stats(expert_trajs))

Expert stats: {'n_traj': 628, 'return_min': -31.0, 'return_mean': 5.714968152866242, 'return_std': 8.43917722649147, 'return_max': 14.0, 'len_min': 8, 'len_mean': 8.0, 'len_std': 0.0, 'len_max': 8}
```

1.19.1 Training the reward function

The true reward here is not linear in the reduced feature space (i.e $(x,y)$ coordinates). Finding an appropriate linear reward is impossible, but an MLP should Just Work™.

```python
import matplotlib.pyplot as plt
import torch as th

def train_mce_irl(demos, hidden_sizes, lr=0.01, **kwargs):
    reward_net = reward_nets.BasicRewardNet(
        env_single.observation_space,
        env_single.action_space,
        hid_sizes=hidden_sizes,
        use_action=False,
        use_done=False,
        use_next_state=False,
    )
    mce_irl = MCEIRL(
        demos,
        env_single,
        reward_net,
        log_interval=250,
        optimizer_kwarg=dict(lr=lr),
        rng=rng,
    )
    occ_measure = mce_irl.train(**kwargs)

    imitation_trajs = rollout.generate_trajectories(
        policy=mce_irl.policy,
        venv=state_venv,
        sample_until=rollout.make_min_timesteps(5000),
        rng=rng,
    )
    print("Imitation stats: ", rollout.rollout_stats(imitation_trajs))

    plt.figure(figsize=(10, 5))
```

(continues on next page)
As you can see, a linear reward model cannot fit the data. Even though we’re training the model on analytically computed occupancy measures for the optimal policy, the resulting reward and occupancy frequencies diverge sharply.
Now, let's try using a very simple nonlinear reward model: an MLP with a single hidden layer. We first train it on the analytically computed occupancy measures. This should give a very precise result.

```python
train_mce_irl(om, hidden_sizes=[256])
```

Now, let's try using a very simple nonlinear reward model: an MLP with a single hidden layer. We first train it on the analytically computed occupancy measures. This should give a very precise result.

```python
train_mce_irl(om, hidden_sizes=[256])
```
| weight_norm | 11.5 |
|--------------------------
| grad_norm | 0.062 |
| iteration | 250 |
| linf_delta | 0.049 |
| weight_norm | 18 |
|--------------------------
| grad_norm | 0.0524 |
| iteration | 500 |
| linf_delta | 0.0175 |
| weight_norm | 20.1 |
|--------------------------
| grad_norm | 0.0406 |
| iteration | 750 |
| linf_delta | 0.0088 |
| weight_norm | 21.8 |

Imitation stats: ```
{ 'n_traj': 628, 'return_min': -31.0, 'return_mean': 5.777070063694268, 'return_std': 9.154267896169321, 'return_max': 14.0, 'len_min': 8, 'len_mean': 8.0, 'len_std': 0.0, 'len_max': 8}```
Then we train it on trajectories sampled from the expert. This gives a stochastic approximation to occupancy measure, so performance is a little worse. Using more expert trajectories should improve performance – try it!

```python
mce_irl_from_trajs = train_mce_irl(expert_trajs[0:10], hidden_sizes=[256])
```

Imitation stats: {
    'n_traj': 628, 'return_min': -31.0, 'return_mean': 5.5, 'return_std': -8.35488720015947, 'return_max': 14.0, 'len_min': 8, 'len_mean': 8.0, 'len_std': 0.0, 'len_max': 8}
While the learned reward function is quite different from the true reward function, it induces a virtually identical occupancy measure over the states. In particular, states below the top row get almost the same reward as top-row states. This is because in Cliff World, there is an upward-blowing wind which will push the agent toward the top row with probability 0.3 at every timestep.

Even though the agent only gets reward in the top row squares, and maximum reward in the top righthand square, the reward model considers it to be almost as good to end up in one of the squares below the top rightmost square, since the wind will eventually blow the agent to the goal square.

download this notebook here

1.20 Learning a Reward Function using Kernel Density

This demo shows how to train a Pendulum agent (exciting!) with our simple density-based imitation learning baselines. DensityTrainer has a few interesting parameters, but the key ones are:

1. density_type: this governs whether density is measured on $(s,s')$ pairs (db.STATE_STATE_DENSITY), $(s,a)$ pairs (db.STATE_ACTION_DENSITY), or single states (db.STATE_DENSITY).

2. is_stationary: determines whether a separate density model is used for each time step $t$ (False), or the same model is used for transitions at all times (True).

3. standardise_inputs: if True, each dimension of the agent state vectors will be normalised to have zero mean and unit variance over the training dataset. This can be useful when not all elements of the demonstration vector are on the same scale, or when some elements have too wide a variation to be captured by the fixed kernel width (1 for Gaussian kernel).

4. kernel: changes the kernel used for non-parametric density estimation. gaussian and exponential are the best bets; see the sklearn docs for the rest.

```python
import pprint
from imitation.algorithms import density as db
from imitation.data import types
from imitation.util import util

# Set FAST = False for longer training. Use True for testing and CI.
FAST = True
if FAST:

    # (continues on next page)
```
N_VEC = 1
N_TRAJECTORIES = 1
N_ITERATIONS = 1
N_RL_TRAIN_STEPS = 100

else:
    N_VEC = 8
    N_TRAJECTORIES = 10
    N_ITERATIONS = 100
    N_RL_TRAIN_STEPS = int(1e5)

from stable_baselines3.common.policies import ActorCriticPolicy
from stable_baselines3 import PPO
from huggingface_sb3 import load_from_hub
from imitation.data import rollout
from stable_baselines3.common.vec_env import DummyVecEnv
import gym
import numpy as np
rng = np.random.default_rng()
env_name = "Pendulum-v1"
expert = PPO.load(
    load_from_hub("HumanCompatibleAI/ppo-Pendulum-v1", "ppo-Pendulum-v1.zip"))
).policy
rollout_env = DummyVecEnv(
    [lambda: RolloutInfoWrapper(gym.make(env_name)) for _ in range(N_VEC)]
)
rollouts = rollout.rollout(
    expert,
    rollout_env,
    rollout.make_sample_until(min_timesteps=2000, min_episodes=57),
    rng=rng,
)

env = util.make_vec_env(env_name, n_envs=N_VEC, rng=rng)

imitation_trainer = PPO(ActorCriticPolicy, env, learning_rate=3e-4, n_steps=2048)
density_trainer = db.DensityAlgorithm(
    env=env,
    rng=rng,
    demonstrations=rollouts,
    rl_algo=imitation_trainer,
    density_type=db.DensityType.STATE_ACTION_DENSITY,
    is_stationary=True,
    kernel="gaussian",
    kernel_bandwidth=0.2,  # found using divination & some palm reading
    standardise_inputs=True,
)
density_trainer.train()
def print_stats(density_trainer, n_trajectories, epoch=""):
    stats = density_trainer.test_policy(n_trajectories=n_trajectories)
    print("True reward function stats:")
    pprint.pprint(stats)
    stats_im = density_trainer.test_policy(
        true_reward=False,
        n_trajectories=n_trajectories,
    )
    print(f"Imitation reward function stats, epoch {epoch}:")
    pprint.pprint(stats_im)

novice_stats = density_trainer.test_policy(n_trajectories=N_TRAJECTORIES)
print("Stats before training:")
print_stats(density_trainer, 1)

print("Stats after training:")
for i in range(N_ITERATIONS):
    density_trainer.train_policy(N_RL_TRAIN_STEPS)
    print_stats(density_trainer, 1, epoch=str(i))

Stats before training:
True reward function stats:
{'len_max': 200,
 'len_mean': 200.0,
 'len_min': 200,
 'len_std': 0.0,
 'monitor_return_len': 1,
 'monitor_return_max': -822.98752,
 'monitor_return_mean': -822.98752,
 'monitor_return_min': -822.98752,
 'monitor_return_std': 0.0,
 'n_traj': 1,
 'return_max': -822.9875172190368,
 'return_mean': -822.9875172190368,
 'return_min': -822.9875172190368,
 'return_std': 0.0}

Imitation reward function stats, epoch :
{'len_max': 200,
 'len_mean': 200.0,
 'len_min': 200,
 'len_std': 0.0,
 'monitor_return_len': 1,
 'monitor_return_max': -1178.229315,
 'monitor_return_mean': -1178.229315,
 'monitor_return_min': -1178.229315,
 'monitor_return_std': 0.0,
 'n_traj': 1,
 'return_max': -6794.506880283356,
 'return_mean': -6794.506880283356,
 'return_min': -6794.506880283356,
 'return_std': 0.0}
Stats after training:
True reward function stats:

```{len_max': 200,
'len_mean': 200.0,
'len_min': 200,
'len_std': 0.0,
'monitor_return_len': 1,
'monitor_return_max': -1204.345543,
'monitor_return_mean': -1204.345543,
'monitor_return_min': -1204.345543,
'monitor_return_std': 0.0,
'n_traj': 1,
'return_max': -1204.3455424308777,
'return_mean': -1204.3455424308777,
'return_min': -1204.3455424308777,
'return_std': 0.0}
```

Imitation reward function stats, epoch 0:

```{len_max': 200,
'len_mean': 200.0,
'len_min': 200,
'len_std': 0.0,
'monitor_return_len': 1,
'monitor_return_max': -993.922645,
'monitor_return_mean': -993.922645,
'monitor_return_min': -993.922645,
'monitor_return_std': 0.0,
'n_traj': 1,
'return_max': -6345.576963186264,
'return_mean': -6345.576963186264,
'return_min': -6345.576963186264,
'return_std': 0.0}
```

### 1.21 Loading Experts

The algorithms in the imitation library are all about learning from some kind of expert. In many cases this expert is a piece of software itself. The imitation library natively supports experts trained using the stable-baselines3 reinforcement learning library.

For example, BC and DAgger can learn from an expert policy and the command line interface of AIRL/GAIL allows one to specify an expert to sample demonstrations from.

In the First Steps tutorial, we first train an expert policy using the stable-baselines3 library and then imitate it’s behavior using Behavioral Cloning (BC). In practice, you may want to load a pre-trained policy for performance reasons.
1.21.1 Loading a policy from a file

The Python interface provides a \texttt{load\_policy()} function to which you pass a \texttt{policy\_type}, a VecEnv and any extra kwargs to pass to the corresponding policy loader.

```python
from imitation.policies.serialize import load_policy
from imitation.util import util

venv = util.make_vec_env("your-env", n_envs=4)
local_policy = load_policy("ppo", venv, loader_kwargs={"path": "path/to/model.zip"})
```

To load a policy from disk, use either \texttt{ppo} or \texttt{sac} as the policy type. The path is specified by \texttt{path} in the \texttt{loader\_kwargs} and it should either point to a zip file containing the policy or a directory containing a \texttt{model.zip} file that was created by stable-baselines3.

In the command line interface the \texttt{expert.policy\_type} and \texttt{expert.loader\_kwargs} parameters define the expert policy to load. For example, to train AIRL on a PPO expert, you would use the following command:

```bash
python -m imitation.scripts.train_adversarial airl \
    with expert.policy\_type=ppo expert.loader\_kwargs.path="path/to/model.zip"
```

1.21.2 Loading a policy from HuggingFace

HuggingFace is a popular repository for pre-trained models.

To load a stable-baselines3 policy from HuggingFace, use either \texttt{ppo-huggingface} or \texttt{sac-huggingface} as the policy type. By default, the policies are loaded from the HumanCompatibleAI organization, but you can override this by setting the \texttt{organization} parameter in the \texttt{loader\_kwargs}. When using the Python API, you also have to specify the environment name as \texttt{env\_name}.

```python
from imitation.policies.serialize import load_policy
from imitation.util import util

venv = util.make_vec_env("your-env", n_envs=4)
remote_policy = load_policy("ppo-huggingface",
    loader_kwargs=dict(
        organization="your-org",
        env_name="your-env"
    )
)
```

In the command line interface, the \texttt{env-name} is automatically injected into the \texttt{loader\_kwargs} and does not need to be defined explicitly. In this example, to train AIRL on a PPO expert that was loaded from \texttt{your-org} on HuggingFace:

```bash
python -m imitation.scripts.train_adversarial airl \
    with expert.policy\_type=ppo-huggingface expert.loader\_kwargs.organization=your-org
```
1.21.3 Uploading a policy to HuggingFace

The huggingface-sb3 package provides utilities to push your models to HuggingFace and load them from there. Make sure to use the naming scheme helpers as described in the readme. Otherwise, the loader will not be able to find your model in the repository.

For a convenient high-level interface to train RL models and upload them to HuggingFace, we recommend using the rl-baselines3-zoo.

1.21.4 Custom expert types

If you want to use a custom expert type, you can write a corresponding factory function according to PolicyLoaderFn() and then register it at the policy_registry. For example:

```python
from imitation.policies.serialize import policy_registry
from stable_baselines3.common import policies

def my_policy_loader(venv, some_param: int) -> policies.BasePolicy:
    # load your policy here
    return policy

policy_registry.register("my-policy", my_policy_loader)
```

Then, you can use my-policy as the policy_type in the command line interface or the Python API:

```bash
python -m imitation.scripts.train_adversarial airl \
    with expert.policy_type=my-policy expert.loader_kwargs.some_param=42
```
2.1 imitation

 imitation: implementations of imitation and reward learning algorithms.

**Modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation.algorithms</td>
<td>Implementations of imitation and reward learning algorithms.</td>
</tr>
<tr>
<td>imitation.data</td>
<td>Modules handling environment data.</td>
</tr>
<tr>
<td>imitation.policies</td>
<td>Classes defining policies and methods to manipulate them (e.g.</td>
</tr>
<tr>
<td>imitation.regularization</td>
<td>Implements a variety of regularization techniques for NN weights.</td>
</tr>
<tr>
<td>imitation.rewards</td>
<td>Reward models: neural network modules, serialization, preprocessing, etc.</td>
</tr>
<tr>
<td>imitation.scripts</td>
<td>Command-line scripts.</td>
</tr>
<tr>
<td>imitation.testing</td>
<td>Helper methods for unit tests.</td>
</tr>
<tr>
<td>imitation.util</td>
<td>General utility functions: e.g.</td>
</tr>
</tbody>
</table>

2.1.1 imitation.algorithms

Implementations of imitation and reward learning algorithms.
**imitation**

**Modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation.algorithms.adversarial</td>
<td>Adversarial imitation learning algorithms, AIRL and GAIL.</td>
</tr>
<tr>
<td>imitation.algorithms.base</td>
<td>Module of base classes and helper methods for imitation learning algorithms.</td>
</tr>
<tr>
<td>imitation.algorithms.bc</td>
<td>Behavioural Cloning (BC).</td>
</tr>
<tr>
<td>imitation.algorithms.density</td>
<td>Density-based baselines for imitation learning.</td>
</tr>
<tr>
<td>imitation.algorithms.mce_irl</td>
<td>Finite-horizon tabular Maximum Causal Entropy IRL.</td>
</tr>
<tr>
<td>imitation.algorithms.preference_comparisons</td>
<td>Learning reward models using preference comparisons.</td>
</tr>
</tbody>
</table>

**imitation.algorithms.adversarial**

Adversarial imitation learning algorithms, AIRL and GAIL.

**Modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation.algorithms.adversarial.airl</td>
<td>Adversarial Inverse Reinforcement Learning (AIRL).</td>
</tr>
<tr>
<td>imitation.algorithms.adversarial.common</td>
<td>Core code for adversarial imitation learning, shared between GAIL and AIRL.</td>
</tr>
<tr>
<td>imitation.algorithms.adversarial.gail</td>
<td>Generative Adversarial Imitation Learning (GAIL).</td>
</tr>
</tbody>
</table>

**imitation.algorithms.adversarial.airl**

Adversarial Inverse Reinforcement Learning (AIRL).

**Classes**

**AIRL(*, demonstrations, demo_batch_size, ...)** Adversarial Inverse Reinforcement Learning (AIRL).

```python
class imitation.algorithms.adversarial.airl.AIRL(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)
    Bases: AdversarialTrainer
    Adversarial Inverse Reinforcement Learning (AIRL).

    __init__(*demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)
        Builds an AIRL trainer.
        Parameters
        - **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).```

92 Chapter 2. API Reference
• **demo_batch_size** *(int)* – The number of samples in each batch of expert data. The discriminator batch size is twice this number because each discriminator batch contains a generator sample for every expert sample.

• **venv** *(VecEnv)* – The vectorized environment to train in.

• **gen_algo** *(BaseAlgorithm)* – The generator RL algorithm that is trained to maximize discriminator confusion. Environment and logger will be set to **venv** and **custom_logger**.

• **reward_net** *(RewardNet)* – Reward network; used as part of AIRL discriminator.

• ****kwargs** – Passed through to **AdversarialTrainer.__init__**.

 Raises

**TypeError** – If **gen_algo.policy** does not have an *evaluate_actions* attribute (present in *ActorCriticPolicy*), needed to compute log-probability of actions.

**logits_expert_is_high** *(state, action, next_state, done, log_policy_act_prob=None)*

Compute the discriminator’s logits for each state-action sample.

In Fu’s AIRL paper (https://arxiv.org/pdf/1710.11248.pdf), the discriminator output was given as

\[ D_\theta(s,a) = \frac{\exp r_\theta(s,a)}{\exp r_\theta(s,a) + \pi(a|s)} \]

with a high value corresponding to the expert and a low value corresponding to the generator.

In other words, the discriminator output is the probability that the action is taken by the expert rather than the generator.

The logit of the above is given as

\[ \text{logit}(D_\theta(s,a)) = r_\theta(s,a) - \log \pi(a|s) \]

which is what is returned by this function.

 Parameters

• **state** *(Tensor)* – The state of the environment at the time of the action.

• **action** *(Tensor)* – The action taken by the expert or generator.

• **next_state** *(Tensor)* – The state of the environment after the action.

• **done** *(Tensor)* – Whether a *terminal state* (as defined under the MDP of the task) has been reached.

• **log_policy_act_prob** *(Optional[Tensor])* – The log probability of the action taken by the generator, \( \log \pi(a|s) \).

 Returns

**Tensor**

The logits of the discriminator for each state-action sample.

 Raises

**TypeError** – If **log_policy_act_prob** is None.

**property reward_test:** *(RewardNet)*

Returns the unshaped version of reward network used for testing.

 Returns

**RewardNet**

2.1. imitation
property reward_train: RewardNet
    Reward used to train generator policy.

    Return type
    RewardNet

venv: VecEnv
    The original vectorized environment.

venv_train: VecEnv
    Like self.venv, but wrapped with train reward unless in debug mode.

    If debug_use_ground_truth=True was passed into the initializer then self.venv_train is the same as self.venv.

venv_wrapped: VecEnvWrapper

imitation.algorithms.adversarial.common

Core code for adversarial imitation learning, shared between GAIL and AIRL.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```
compute_train_stats(...) Train statistics for GAIL/AIRL discriminator.
```

Classes

```
AdversarialTrainer(*, demonstrations, ...[, ...]) Base class for adversarial imitation learning algorithms like GAIL and AIRL.
```

class imitation.algorithms.adversarial.common.AdversarialTrainer(*, demonstrations,
    demo_batch_size, venv,
    gen_algo, reward_net,
    demo_minibatch_size=None,
    n_disc_updates_per_round=2,
    log_dir='output/',
    disc_opt_cls=<class 'torch.optim.ADAM.Adam'>,
    disc_opt_kwargs=None,
    gen_train_timesteps=None,
    gen_replay_buffer_capacity=None,
    custom_logger=None,
    init_tensorboard=False,
    init_tensorboard_graph=False,
    debug_use_ground_truth=False,
    al-
    low_variable_horizon=False)
Bases: `DemonstrationAlgorithm[Transitions]`

Base class for adversarial imitation learning algorithms like GAIL and AIRL.

```python
__init__(*demonstrations, demo_batch_size, venv, gen_algo, reward_net, demo_minibatch_size=None, n_disc_updates_per_round=2, log_dir=output_path, disc_opt_cls=torch.optim.Adam, disc_opt_kwargs=None, gen_train_timesteps=None, gen_replay_buffer_capacity=None, custom_logger=None, init_tensorboard=False, init_tensorboard_graph=False, debug_use_ground_truth=False, allow_variable_horizon=False)
```

Builds `AdversarialTrainer`.

**Parameters**

- `demonstrations` (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Demonstrations from an expert (optional). Transitions expressed directly as a `types.TransitionsMinimal` object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

- `demo_batch_size` (int) – The number of samples in each batch of expert data. The discriminator batch size is twice this number because each discriminator batch contains a generator sample for every expert sample.

- `venv` (VecEnv) – The vectorized environment to train in.

- `gen_algo` (BaseAlgorithm) – The generator RL algorithm that is trained to maximize discriminator confusion. Environment and logger will be set to `venv` and `custom_logger`.

- `reward_net` (RewardNet) – a Torch module that takes an observation, action and next observation tensors as input and computes a reward signal.

- `demo_minibatch_size` (Optional[int]) – size of minibatch to calculate gradients over. The gradients are accumulated until the entire batch is processed before making an optimization step. This is useful in GPU training to reduce memory usage, since fewer examples are loaded into memory at once, facilitating training with larger batch sizes, but is generally slower. Must be a factor of `demo_batch_size`. Optional, defaults to `demo_batch_size`.

- `n_disc_updates_per_round` (int) – The number of discriminator updates after each round of generator updates in `AdversarialTrainer.learn()`.

- `log_dir` (Union[str, bytes, PathLike]) – Directory to store TensorBoard logs, plots, etc. in.

- `disc_opt_cls` (Type[Optimizer]) – The optimizer for discriminator training.

- `disc_opt_kwargs` (Optional[Mapping]) – Parameters for discriminator training.

- `gen_train_timesteps` (Optional[int]) – The number of steps to train the generator policy for each iteration. If None, then defaults to the batch size (for on-policy) or number of environments (for off-policy).

- `gen_replay_buffer_capacity` (Optional[int]) – The capacity of the generator replay buffer (the number of obs-action-obs samples from the generator that can be stored). By default this is equal to `gen_train_timesteps`, meaning that we sample only from the most recent batch of generator samples.

- `custom_logger` (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

- `init_tensorboard` (bool) – If True, makes various discriminator TensorBoard summaries.
• `init_tensorboard_graph` (bool) – If both this and `init_tensorboard` are True, then write a Tensorboard graph summary to disk.

• `debug_use_ground_truth` (bool) – If True, use the ground truth reward for `self.train_env`. This disables the reward wrapping that would normally replace the environment reward with the learned reward. This is useful for sanity checking that the policy training is functional.

• `allow_variable_horizon` (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

Raises

`ValueError` – if the batch size is not a multiple of the minibatch size.

abstract `logits_expert_is_high` (state, action, next_state, done, log_policy_act_prob=None)

Compute the discriminator’s logits for each state-action sample.

A high value corresponds to predicting expert, and a low value corresponds to predicting generator.

Parameters

• `state` (Tensor) – state at time t, of shape `(batch_size,) + state_shape`.

• `action` (Tensor) – action taken at time t, of shape `(batch_size,) + action_shape`.

• `next_state` (Tensor) – state at time t+1, of shape `(batch_size,) + state_shape`.

• `done` (Tensor) – binary episode completion flag after action at time t, of shape `(batch_size,)`.

• `log_policy_act_prob` (Optional[Tensor]) – log probability of generator policy taking `action` at time t.

Return type

Tensor

Returns

Discriminator logits of shape `(batch_size,)`. A high output indicates an expert-like transition.

property `policy`: BasePolicy

Returns a policy imitating the demonstration data.

Return type

BasePolicy

abstract property `reward_test`: RewardNet

Reward used to train policy at “test” time after adversarial training.

Return type

RewardNet

abstract property `reward_train`: RewardNet

Reward used to train generator policy.

Return type

RewardNet
set_demonstrations(demonstrations)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

Parameters

demonstrations (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

Return type
None

train(total_timesteps, callback=None)

Alternates between training the generator and discriminator.

Every “round” consists of a call to train_gen(self.gen_train_timesteps), a call to train_disc, and finally a call to callback(round).

Training ends once an additional “round” would cause the number of transitions sampled from the environment to exceed total_timesteps.

Parameters

• total_timesteps (int) – An upper bound on the number of transitions to sample from the environment during training.

• callback (Optional[Callable[[int], None]]) – A function called at the end of every round which takes in a single argument, the round number. Round numbers are in range(total_timesteps // self.gen_train_timesteps).

Return type
None

train_disc(*, expert_samples=None, gen_samples=None)

Perform a single discriminator update, optionally using provided samples.

Parameters

• expert_samples (Optional[Mapping]) – Transition samples from the expert in dictionary form. If provided, must contain keys corresponding to every field of the Transitions dataclass except “infos”. All corresponding values can be either NumPy arrays or Tensors. Extra keys are ignored. Must contain self.demo_batch_size samples. If this argument is not provided, then self.demo_batch_size expert samples from self.demo_data_loader are used by default.

• gen_samples (Optional[Mapping]) – Transition samples from the generator policy in same dictionary form as expert_samples. If provided, must contain exactly self.demo_batch_size samples. If not provided, then take len(expert_samples) samples from the generator replay buffer.

Return type
Mapping[str, float]

Returns
Statistics for discriminator (e.g. loss, accuracy).

train_gen(total_timesteps=None, learn_kwargs=None)

Trains the generator to maximize the discriminator loss.
After the end of training populates the generator replay buffer (used in discriminator training) with 
self.disc_batch_size transitions.

**Parameters**

- **total_timesteps** (Optional[int]) – The number of transitions to sample from 
  self.venv_train during training. By default, self.gen_train_timesteps.
- **learn_kwargs** (Optional[Mapping]) – kwargs for the Stable Baselines RLModel.learn() 
  method.

**Return type**

None

**venv**: VecEnv

The original vectorized environment.

**venv_train**: VecEnv

Like self.venv, but wrapped with train reward unless in debug mode.
If debug_use_ground_truth=True was passed into the initializer then self.venv_train is the same as self.venv.

**venv_wrapped**: VecEnvWrapper

imitation.algorithms.adversarial.common.compute_train_stats(disc_logits_expert_is_high, 
labels_expert_is_one, disc_loss)

Train statistics for GAIL/AIRL discriminator.

**Parameters**

- **disc_logits_expert_is_high** (Tensor) – discriminator logits produced by AdversarialTrainer.logits_expert_is_high.
- **labels_expert_is_one** (Tensor) – integer labels describing whether logit was for an ex-
  pert (0) or generator (1) sample.
- **disc_loss** (Tensor) – final discriminator loss.

**Return type**

Mapping[str, float]

**Returns**

A mapping from statistic names to float values.

imitation.algorithms.adversarial.gail

Generative Adversarial Imitation Learning (GAIL).

**Classes**

<table>
<thead>
<tr>
<th>GAIL(*, demonstrations, demo_batch_size, ...)</th>
<th>Generative Adversarial Imitation Learning (GAIL).</th>
</tr>
</thead>
<tbody>
<tr>
<td>RewardNetFromDiscriminatorLogit(base)</td>
<td>Converts the discriminator logits raw value to a reward signal.</td>
</tr>
</tbody>
</table>
class imitation.algorithms.adversarial.gail.GAIL(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)

Bases: AdversarialTrainer

Generative Adversarial Imitation Learning (GAIL).

__init__(*, demonstrations, demo_batch_size, venv, gen_algo, reward_net, **kwargs)

Generative Adversarial Imitation Learning.

Parameters

- demonstrations (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

- demo_batch_size (int) – The number of samples in each batch of expert data. The discriminator batch size is twice this number because each discriminator batch contains a generator sample for every expert sample.

- venv (VecEnv) – The vectorized environment to train in.

- gen_algo (BaseAlgorithm) – The generator RL algorithm that is trained to maximize discriminator confusion. Environment and logger will be set to venv and custom_logger.

- reward_net (RewardNet) – a Torch module that takes an observation, action and next observation tensor as input, then computes the logits. Used as the GAIL discriminator.

- **kwargs – Passed through to AdversarialTrainer.__init__.

allow_variable_horizon: bool

If True, allow variable horizon trajectories; otherwise error if detected.

logits_expert_is_high(state, action, next_state, done, log_policy_act_prob=None)

Compute the discriminator’s logits for each state-action sample.

Parameters

- state (Tensor) – The state of the environment at the time of the action.

- action (Tensor) – The action taken by the expert or generator.

- next_state (Tensor) – The state of the environment after the action.

- done (Tensor) – whether a terminal state (as defined under the MDP of the task) has been reached.

- log_policy_act_prob (Optional[Tensor]) – The log probability of the action taken by the generator, \(\log P(a|s)\).

Return type

Tensor

Returns

The logits of the discriminator for each state-action sample.

property reward_test: RewardNet

Reward used to train policy at “test” time after adversarial training.

Return type

RewardNet
**property reward_train:**  *RewardNet*

Reward used to train generator policy.

**Return type**

*RewardNet*

**venv:**  *VecEnv*

The original vectorized environment.

**venv_train:**  *VecEnv*

Like *self.venv*, but wrapped with train reward unless in debug mode.

If *debug_use_ground_truth=True* was passed into the initializer then *self.venv_train* is the same as *self.venv*.

**venv_wrapped:**  *VecEnvWrapper*

---

**class imitation.algorithms.adversarial.gail.RewardNetFromDiscriminatorLogit(base)**

**Bases:** *RewardNet*

Converts the discriminator logits raw value to a reward signal.

Wrapper for reward network that takes in the logits of the discriminator probability distribution and outputs the corresponding reward for the GAIL algorithm.

Below is the derivation of the transformation that needs to be applied.

The GAIL paper defines the cost function of the generator as:

\[ \log D \]

as shown on line 5 of Algorithm 1. In the paper, \( D \) is the probability distribution learned by the discriminator, where \( D(X) = 1 \) if the trajectory comes from the generator, and \( D(X) = 0 \) if it comes from the expert. In this implementation, we have decided to use the opposite convention that \( D(X) = 0 \) if the trajectory comes from the generator, and \( D(X) = 1 \) if it comes from the expert. Therefore, the resulting cost function is:

\[ \log (1 - D) \]

Since our algorithm trains using a reward function instead of a loss function, we need to invert the sign to get:

\[ R = -\log (1 - D) = \log \frac{1}{1 - D} \]

Now, let \( L \) be the output of our reward net, which gives us the logits of \( D \) (\( L = \text{logit} \ D \)). We can write:

\[ D = \text{sigmoid} \ L = \frac{1}{1 + e^{-L}} \]

Since \( 1 - \text{sigmoid} \ (L) \) is the same as \( \text{sigmoid} \ (-L) \), we can write:

\[ R = -\log \text{sigmoid} \ (-L) \]

which is a non-decreasing map from the logits of \( D \) to the reward.

**__init__(base)**

Builds LogSigmoidRewardNet to wrap *reward_net*.

**forward(state, action, next_state, done)**

Compute rewards for a batch of transitions and keep gradients.

**Return type**

*Tensor*

**training:**  *bool*
imitation.algorithms.base

Module of base classes and helper methods for imitation learning algorithms.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```python
make_data_loader(transitions, batch_size[, ...]) Converts demonstration data to Torch data loader.
```

Classes

```python
BaseImitationAlgorithm(*[, custom_logger, ...]) Base class for all imitation learning algorithms.
```

```python
DemonstrationAlgorithm(*, demonstrations[, ...]) An algorithm that learns from demonstration: BC, IRL, etc.
```

class imitation.algorithms.base.BaseImitationAlgorithm(*, custom_logger=None, allow_variable_horizon=False)

Bases: ABC

Base class for all imitation learning algorithms.

```python
__init__(*, custom_logger=None, allow_variable_horizon=False)
```

Creates an imitation learning algorithm.

Parameters

- custom_logger (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

- allow_variable_horizon (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/getting-started/variable-horizon.html before overriding this.

```python
allow_variable_horizon: bool
```

If True, allow variable horizon trajectories; otherwise error if detected.

property logger:  HierarchicalLogger

Return type

HierarchicalLogger

class imitation.algorithms.base.DemonstrationAlgorithm(*, demonstrations, custom_logger=None, allow_variable_horizon=False)

Bases: BaseImitationAlgorithm, Generic[TransitionKind]

An algorithm that learns from demonstration: BC, IRL, etc.
__init__(*, demonstrations, custom_logger=None, allow_variable_horizon=False)

Creates an algorithm that learns from demonstrations.

**Parameters**

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal, None]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

- **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

**allow_variable_horizon**: bool

If True, allow variable horizon trajectories; otherwise error if detected.

**abstract property policy**: BasePolicy

Returns a policy imitating the demonstration data.

**Return type**

BasePolicy

**abstract set_demonstrations**(demonstrations)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

**Parameters**

- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

**Return type**

None

**imitation.algorithms.base.make_data_loader**(transitions, batch_size, data_loader_kwargs=None)

Converts demonstration data to Torch data loader.

**Parameters**

- **transitions** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).

- **batch_size** (int) – The size of the batch to create. Does not change the batch size if transitions is already an iterable of transition batches.

- **data_loader_kwargs** (Optional[Mapping[str, Any]]) – Arguments to pass to th.data.DataLoader.

**Return type**

Iterable[Mapping[str, Union[ndarray, Tensor]]]
imitation

Returns
An iterable of transition batches.

Raises

- **ValueError** – if `transitions` is an iterable over transition batches with batch size not equal to `batch_size`; or if `transitions` is transitions or a sequence of trajectories with total timesteps less than `batch_size`.

- **TypeError** – if `transitions` is an unsupported type.

**imitation.algorithms.bc**

Behavioural Cloning (BC).

Trains policy by applying supervised learning to a fixed dataset of (observation, action) pairs generated by some expert demonstrator.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

- `enumerate_batches(batch_it)` Prepends batch stats before the batches of a batch iterator.

- `reconstruct_policy(policy_path[, device])` Reconstruct a saved policy.

**Classes**

```python
class imitation.algorithms.bc.BC(*, observation_space, action_space, rng, policy=None, demonstrations=None, batch_size=32, minibatch_size=None, optimizer_cls=<class 'torch.optim.adam.Adam'>, optimizer_kwargs=None, ent_weight=0.001, l2_weight=0.0, device='auto', custom_logger=None)
```

Bases: `DemonstrationAlgorithm`

Behavioral cloning (BC).

Recovers a policy via supervised learning from observation-action pairs.
__init__(*, observation_space, action_space, rng, policy=None, demonstrations=None, batch_size=32, minibatch_size=None, optimizer_cls=<class 'torch.optim.Adam'>, optimizer_kwargs=None, ent_weight=0.001, l2_weight=0.0, device='auto', custom_logger=None)

Builds BC.

Parameters

- **observation_space** (Space) – the observation space of the environment.
- **action_space** (Space) – the action space of the environment.
- **rng** (Generator) – the random state to use for the random number generator.
- **policy** (Optional[ActorCriticPolicy]) – a Stable Baselines3 policy; if unspecified, defaults to FeedForward32Policy.
- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal, None]) – Demonstrations from an expert (optional). Transitions expressed directly as a types.TransitionsMinimal object, a sequence of trajectories, or an iterable of transition batches (mappings from keywords to arrays containing observations, etc).
- **batch_size** (int) – The number of samples in each batch of expert data.
- **minibatch_size** (Optional[int]) – size of minibatch to calculate gradients over. The gradients are accumulated until batch_size examples are processed before making an optimization step. This is useful in GPU training to reduce memory usage, since fewer examples are loaded into memory at once, facilitating training with larger batch sizes, but is generally slower. Must be a factor of batch_size. Optional, defaults to batch_size.
- **optimizer_cls** (Type[Optimizer]) – optimiser to use for supervised training.
- **optimizer_kwargs** (Optional[Mapping[str, Any]]) – keyword arguments, excluding learning rate and weight decay, for optimiser construction.
- **ent_weight** (float) – scaling applied to the policy’s entropy regularization.
- **l2_weight** (float) – scaling applied to the policy’s L2 regularization.
- **device** (Union[str, device]) – name/identity of device to place policy on.
- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

Raises

- **ValueError** – If weight_decay is specified in optimizer_kwargs (use the parameter l2_weight instead), or if the batch size is not a multiple of the minibatch size.

allow_variable_horizon:  bool

If True, allow variable horizon trajectories; otherwise error if detected.

property policy:  ActorCriticPolicy

Returns a policy imitating the demonstration data.

Return type

ActorCriticPolicy

save_policy(policy_path)

Save policy to a path. Can be reloaded by .reconstruct_policy().

Parameters

- **policy_path** (Union[str, bytes, PathLike]) – path to save policy to.
Return type
None

**set_demonstrations**(demonstrations)
Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

Parameters
- **demonstrations** (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[np.ndarray, torch.Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

Return type
None

**train**(*, n_epochs=None, n_batches=None, on_epoch_end=None, on_batch_end=None, log_interval=500, log_rollouts_venv=None, log_rollouts_n_episodes=5, progress_bar=True, reset_tensorboard=False)
Train with supervised learning for some number of epochs.

Here an ‘epoch’ is just a complete pass through the expert data loader, as set by self.set_expert_data_loader(). Note, that when you specify n_batches smaller than the number of batches in an epoch, the on_epoch_end callback will never be called.

Parameters
- **n_epochs** (Optional[int]) – Number of complete passes made through expert data before ending training. Provide exactly one of n_epochs and n_batches.
- **n_batches** (Optional[int]) – Number of batches loaded from dataset before ending training. Provide exactly one of n_epochs and n_batches.
- **on_epoch_end** (Optional[Callable[[], None]]) – Optional callback with no parameters to run at the end of each epoch.
- **on_batch_end** (Optional[Callable[[], None]]) – Optional callback with no parameters to run at the end of each batch.
- **log_interval** (int) – Log stats after every log_interval batches.
- **log_rollouts_venv** (Optional[VecEnv]) – If not None, then this VecEnv (whose observation and actions spaces must match self.observation_space and self.action_space) is used to generate rollout stats, including average return and average episode length. If None, then no rollouts are generated.
- **log_rollouts_n_episodes** (int) – Number of rollouts to generate when calculating rollout stats. Non-positive number disables rollouts.
- **progress_bar** (bool) – If True, then show a progress bar during training.
- **reset_tensorboard** (bool) – If True, then start plotting to Tensorboard from x=0 even if .train() logged to Tensorboard previously. Has no practical effect if .train() is being called for the first time.

```python
class imitation.algorithms.bc.BCLogger(logger)
    Bases: object

    Utility class to help logging information relevant to Behavior Cloning.

    __init__(logger)
    Create new BC logger.
```

2.1. imitation
Parameters

logger (HierarchicalLogger) – The logger to feed all the information to.

log_batch(batch_num, batch_size, num_samples_so_far, training_metrics, rollout_stats)

log_epoch(epoch_number)

reset_tensorboard_steps()

class imitation.algorithms.bc.BCTrainingMetrics(neglogp, entropy, ent_loss, prob_true_act, l2_norm, l2_loss, loss)

Bases: object

Container for the different components of behavior cloning loss.

__init__(neglogp, entropy, ent_loss, prob_true_act, l2_norm, l2_loss, loss)

ent_loss: Tensor

entropy: Tensor

l2_loss: Tensor

l2_norm: Tensor

loss: Tensor

neglogp: Tensor

prob_true_act: Tensor

class imitation.algorithms.bc.BatchIteratorWithEpochEndCallback(batch_loader, n_epochs, n_batches, on_epoch_end)

Bases: object

Loops through batches from a batch loader and calls a callback after every epoch.

Will throw an exception when an epoch contains no batches.

__init__(batch_loader, n_epochs, n_batches, on_epoch_end)

batch_loader: Iterable[Mapping[str, Union[ndarray, Tensor]]]

n_batches: Optional[int]

n_epochs: Optional[int]

on_epoch_end: Optional[Callable[[int], None]]

class imitation.algorithms.bc.BehaviorCloningLossCalculator(ent_weight, l2_weight)

Bases: object

Functor to compute the loss used in Behavior Cloning.

__init__(ent_weight, l2_weight)

ent_weight: float

l2_weight: float
class imitation.algorithms.bc.RolloutStatsComputer(venv, n_episodes)

Bases: object

Computes statistics about rollouts.

Parameters

• venv (Optional[VecEnv]) – The vectorized environment in which to compute the rollouts.

• n_episodes (int) – The number of episodes to base the statistics on.

__init__(venv, n_episodes)
	n_episodes: int

venv: Optional[VecEnv]

imitation.algorithms.bc.enumerate_batches(batch_it)

Prepends batch stats before the batches of a batch iterator.

Return type

Iterable[Tuple[Tuple[int, int, int], Mapping[str, Union[ndarray, Tensor]]]]

imitation.algorithms.bc.reconstruct_policy(policy_path, device='auto')

Reconstruct a saved policy.

Parameters

• policy_path (str) – path where .save_policy() has been run.

• device (Union[device, str]) – device on which to load the policy.

Returns

policy with reloaded weights.

Return type

policy

imitation.algorithms.dagger


Interactively trains policy by collecting some demonstrations, doing BC, collecting more demonstrations, doing BC again, etc. Initially the demonstrations just come from the expert’s policy; over time, they shift to be drawn more and more from the imitator’s policy.

Functions

# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

reconstruct_trainer(scratch_dir, venv[, ...])

Reconstruct trainer from the latest snapshot in some working directory.
Classes

- **BetaSchedule()**
  Computes beta (\% of time demonstration action used) from training round.

- **DAggerTrainer(*, venv, scratch_dir, rng[,...])**
  DAgger training class with low-level API suitable for interactive human feedback.

- **InteractiveTrajectoryCollector(venv,...)**
  DAgger VecEnvWrapper for querying and saving expert actions.

- **LinearBetaSchedule(rampdown_rounds)**
  Linearly-decreasing schedule for beta.

- **SimpleDAggerTrainer(*, venv, scratch_dir,...)**
  Simpler subclass of DAggerTrainer for training with synthetic feedback.

Exceptions

- **NeedsDemosException**
  Signals demos need to be collected for current round before continuing.

```python
class imitation.algorithms.dagger.BetaSchedule
    Bases: ABC
    Computes beta (\% of time demonstration action used) from training round.

class imitation.algorithms.dagger.DAggerTrainer(*, venv, scratch_dir, rng, beta_schedule=None, bc_trainer, custom_logger=None)
    Bases: BaseImitationAlgorithm
    DAgger training class with low-level API suitable for interactive human feedback.
    In essence, this is just BC with some helpers for incrementally resuming training and interpolating between demonstrator/learner policies. Interaction proceeds in “rounds” in which the demonstrator first provides a fresh set of demonstrations, and then an underlying BC is invoked to fine-tune the policy on the entire set of demonstrations collected in all rounds so far. Demonstrations and policy/trainer checkpoints are stored in a directory with the following structure:

    scratch-dir-name/
        checkpoint-001.pt
        checkpoint-002.pt
        ...
        checkpoint-XZY.pt
        checkpoint-latest.pt
        demos/
            round-000/
                demos_round_000_000.npz
                demos_round_000_001.npz
                ...
            round-001/
                demos_round_001_000.npz
                ...
            ...
            round-XZY/
            ...
```

```
DEFAULT_N_EPOCHS: int = 4
The default number of BC training epochs in `extend_and_update`.

```
__init__(*, venv, scratch_dir, rng, beta_schedule=None, bc_trainer, custom_logger=None)
```
Builds DAggerTrainer.

**Parameters**

- **scratch_dir** ([Union](https://docs.python.org/3/library/typing.html#typing.Union)
  `[str, bytes, PathLike]`) – Directory to use to store intermediate training information (e.g. for resuming training).
- **rng** ([Generator](https://docs.python.org/3/library/typing.html#typing.Generator)) – random state for random number generation.
- **beta_schedule** (Optional[Callable[[int, float]]]) – Provides a value of beta (the probability of taking expert action in any given state) at each round of training. If `None`, then `linear_beta_schedule` will be used instead.
- **bc_trainer** ([BC](https://stable-baselines3.readthedocs.io/en/master/api/dagger.html)) – A BC instance used to train the underlying policy.
- **custom_logger** (Optional[[HierarchicalLogger]])) – Where to log to; if `None` (default), creates a new logger.

```
property batch_size: int
```

**Return type**

`int`

```
create_trajectory_collector()
```
Create trajectory collector to extend current round’s demonstration set.

**Return type**

`InteractiveTrajectoryCollector`

**Returns**

A collector configured with the appropriate beta, imitator policy, etc. for the current round. Refer to the documentation for `InteractiveTrajectoryCollector` to see how to use this.

```
extend_and_update(bc_train_kwargs=None)
```
Extend internal batch of data and train BC.

Specifically, this method will load new transitions (if necessary), train the model for a while, and advance the round counter. If there are no fresh demonstrations in the demonstration directory for the current round, then this will raise a `NeedsDemosException` instead of training or advancing the round counter. In that case, the user should call `create_trajectory_collector()` and use the returned `InteractiveTrajectoryCollector` to produce a new set of demonstrations for the current interaction round.

**Parameters**

- **bc_train_kwargs** (Optional[Mapping[str, Any]]) – Keyword arguments for calling `BC.train()`. If the `log_rollouts_venv` key is not provided, then it is set to `self.venv` by default. If neither of the `n_epochs` and `n_batches` keys are provided, then `n_epochs` is set to `self.DEFAULT_N_EPOCHS`.

**Return type**

`int`

**Returns**

New round number after advancing the round counter.
property logger:  *HierarchicalLogger*

Returns logger for this object.

**Return type**

*HierarchicalLogger*

property policy:  *BasePolicy*

**Return type**

*BasePolicy*

**save_policy**(policy_path)

Save the current policy only (and not the rest of the trainer).

**Parameters**

- **policy_path** (Union[str, bytes, PathLike]) – path to save policy to.

**Return type**

None

**save_trainer**()

Create a snapshot of trainer in the scratch/working directory.

The created snapshot can be reloaded with `reconstruct_trainer()`. In addition to saving one copy of the policy in the trainer snapshot, this method saves a second copy of the policy in its own file. Having a second copy of the policy is convenient because it can be loaded on its own and passed to evaluation routines for other algorithms.

**Returns**

- a path to one of the created *DAggerTrainer* checkpoints. policy_path: a path to one of the created *DAggerTrainer* policies.

**Return type**

checkpoint_path

**class** imitation.algorithms.dagger.InteractiveTrajectoryCollector(venv, get_robot_acts, beta, save_dir, rng)

**Bases:** VecEnvWrapper

*DAgger* VecEnvWrapper for querying and saving expert actions.

Every call to `.step(actions)` accepts and saves expert actions to `self.save_dir`, but only forwards expert actions to the wrapped VecEnv with probability `self.beta`. With probability `1 - self.beta`, a “robot” action (i.e an action from the imitation policy) is forwarded instead.

Demonstrations are saved as *TrajectoryWithRew* to `self.save_dir` at the end of every episode.

**__init__**(venv, get_robot_acts, beta, save_dir, rng)

Builds InteractiveTrajectoryCollector.

**Parameters**

- **venv** (VecEnv) – vectorized environment to sample trajectories from.

- **get_robot_acts** (Callable[[ndarray, ndarray]]) – get robot actions that can be substituted for human actions. Takes a vector of observations as input & returns a vector of actions.

- **beta** (float) – fraction of the time to use action given to `.step()` instead of robot action. The choice of robot or human action is independently randomized for each individual *Env* at every timestep.
- **save_dir** (Union[str, bytes, PathLike]) – directory to save collected trajectories in.
- **rng** (Generator) – random state for random number generation.

**reset()**

Resets the environment.

**Returns**

first observation of a new trajectory.

**Return type**

obs

**seed**(seed=\text{None})

Set the seed for the DAgger random number generator and wrapped VecEnv.

The DAgger RNG is used along with self.beta to determine whether the expert or robot action is forwarded to the wrapped VecEnv.

**Parameters**

- **seed** (Optional[int]) – The random seed. May be None for completely random seeding.

**Return type**

List[Optional[int]]

**Returns**

A list containing the seeds for each individual env. Note that all list elements may be None, if the env does not return anything when seeded.

**step_async**(actions)

Steps with a 1 - beta chance of using self.get_robot_acts instead.

DAgger needs to be able to inject imitation policy actions randomly at some subset of time steps. This method has a self.beta chance of keeping the actions passed in as an argument, and a 1 - self.beta chance of forwarding actions generated by self.get_robot_acts instead. ‘robot’ (i.e. imitation policy) action if necessary.

At the end of every episode, a TrajectoryWithRew is saved to self.save_dir, where every saved action is the expert action, regardless of whether the robot action was used during that timestep.

**Parameters**

- **actions** (ndarray) – the _intended_ demonstrator/expert actions for the current state. This will be executed with probability self.beta. Otherwise, a “robot” (typically a BC policy) action will be sampled and executed instead via self.get_robot_act.

**Return type**

None

**step_wait()**

Returns observation, reward, etc after previous step_async() call.

Stores the transition, and saves trajectory as demo once complete.

**Return type**

Tuple[Union[ndarray, Dict[str, ndarray], Tuple[ndarray, ...]], ndarray, ndarray, List[Dict]]

**Returns**

Observation, reward, dones (is terminal?) and info dict.

**traj_accum:** Optional[TrajectoryAccumulator]
class imitation.algorithms.dagger.LinearBetaSchedule(rampdown_rounds)
    Bases: BetaSchedule
    Linearly-decreasing schedule for beta.
    __init__(rampdown_rounds)
        Builds LinearBetaSchedule.

Parameters
rampdown_rounds (int) – number of rounds over which to anneal beta.

exception imitation.algorithms.dagger.NeedsDemosException
    Bases: Exception
    Signals demos need to be collected for current round before continuing.

class imitation.algorithms.dagger.SimpleDAggerTrainer(*, venv, scratch_dir, expert_policy, rng, 
               expert_trajs=None, **dagger_trainer_kwargs)
    Bases: DAggerTrainer
    Simpler subclass of DAggerTrainer for training with synthetic feedback.
    __init__(*, venv, scratch_dir, expert_policy, rng, expert_trajs=None, **dagger_trainer_kwargs)
        Builds SimpleDAggerTrainer.

Parameters
• venv (VecEnv) – Vectorized training environment. Note that when the robot action is randomly injected (in accordance with beta_schedule argument), every individual environment will get a robot action simultaneously for that timestep.
• scratch_dir (Union[str, bytes, PathLike]) – Directory to use to store intermediate training information (e.g. for resuming training).
• expert_policy (BasePolicy) – The expert policy used to generate synthetic demonstrations.
• rng (Generator) – Random state to use for the random number generator.
• expert_trajs (Optional[Sequence[Trajectory]]) – Optional starting dataset that is inserted into the round 0 dataset.
• dagger_trainer_kwargs – Other keyword arguments passed to the superclass initializer DAggerTrainer.__init__.

Raises
ValueError – The observation or action space does not match between venv and expert_policy.

allow_variable_horizon: bool
    If True, allow variable horizon trajectories; otherwise error if detected.

train(total_timesteps, *, rollout_round_min_episodes=3, rollout_round_min_timesteps=500, 
      bc_train_kwargs=None)
    Train the DAgger agent.
    The agent is trained in “rounds” where each round consists of a dataset aggregation step followed by BC update step.
    During a dataset aggregation step, self.expert_policy is used to perform rollouts in the environment but there is a 1 - beta chance (beta is determined from the round number and self.beta_schedule) that the
DAgger agent's action is used instead. Regardless of whether the DAgger agent's action is used during the rollout, the expert action and corresponding observation are always appended to the dataset. The number of environment steps in the dataset aggregation stage is determined by the `rollout_round_min` arguments.

During a BC update step, `BC.train()` is called to update the DAgger agent on all data collected so far.

**Parameters**

- `total_timesteps` (int) – The number of timesteps to train inside the environment. In practice this is a lower bound, because the number of timesteps is rounded up to finish the minimum number of episodes or timesteps in the last DAgger training round, and the environment timesteps are executed in multiples of `self.venv.num_envs`.

- `rollout_round_min_episodes` (int) – The number of episodes the must be completed before a dataset aggregation step ends.

- `rollout_round_min_timesteps` (int) – The number of environment timesteps that must be completed before a dataset aggregation step ends. Also, that any round will always train for at least `self.batch_size` timesteps, because otherwise BC could fail to receive any batches.

- `bc_train_kwargs` (Optional[dict]) – Keyword arguments for calling `BC.train()`. If the `log_rollouts_venv` key is not provided, then it is set to `self.venv` by default. If neither of the `n_epochs` and `n_batches` keys are provided, then `n_epochs` is set to `self.DEFAULT_N_EPOCHS`.

**Return type**

`None`

### imitation.algorithms.dagger.reconstruct_trainer(scratch_dir, venv, custom_logger=None, device='auto')

Reconstruct trainer from the latest snapshot in some working directory.

Requires vectorized environment and (optionally) a logger, as these objects cannot be serialized.

**Parameters**

- `scratch_dir` (Union[str, bytes, PathLike]) – path to the working directory created by a previous run of this algorithm. The directory should contain `checkpoint-latest.pt` and `policy-latest.pt` files.

- `venv` (VecEnv) – Vectorized training environment.

- `custom_logger` (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

- `device` (Union[device, str]) – device on which to load the trainer.

**Return type**

`DAggerTrainer`

**Returns**

A deserialized `DAggerTrainer`. 
imitation

imitation.algorithms.density

Density-based baselines for imitation learning.

Each of these algorithms learns a density estimate on some aspect of the demonstrations, then rewards the agent for following that estimate.

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DensityAlgorithm(*, demonstrations, venv, rng)</td>
<td>Learns a reward function based on density modeling.</td>
</tr>
<tr>
<td>DensityType(value)</td>
<td>Input type the density model should use.</td>
</tr>
</tbody>
</table>

class imitation.algorithms.density.DensityAlgorithm(*, demonstrations, venv, rng, density_type=DensityType.STATE_ACTION_DENSITY, kernel='gaussian', kernel_bandwidth=0.5, rl_algo=None, is_stationary=True, standardise_inputs=True, custom_logger=None, allow_variable_horizon=False)

Bases: DemonstrationAlgorithm

Learns a reward function based on density modeling.

Specifically, it constructs a non-parametric estimate of $p(s)$, $p(s,a)$, $p(s,s')$ and then computes a reward using the log of these probabilities.

__init__(*, demonstrations, venv, rng, density_type=DensityType.STATE_ACTION_DENSITY, kernel='gaussian', kernel_bandwidth=0.5, rl_algo=None, is_stationary=True, standardise_inputs=True, custom_logger=None, allow_variable_horizon=False)

Builds DensityAlgorithm.

Parameters

- demonstrations (Union[Iterable[Trajectory], Iterable[Mapping[str, Union[n.ndarray, Tensor]]], TransitionsMinimal, None]) – expert demonstration trajectories.
- density_type (DensityType) – type of density to train on: single state, state-action pairs, or state-state pairs.
- kernel_bandwidth (float) – bandwidth of kernel. If standardise_inputs is true and you are using a Gaussian kernel, then it probably makes sense to set this somewhere between 0.1 and 1.
- venv (VecEnv) – The environment to learn a reward model in. We don’t actually need any environment interaction to fit the reward model, but we use this to extract the observation and action space, and to train the RL algorithm rl_algo (if specified).
- rng (Generator) – random state for sampling from demonstrations.
- rl_algo (Optional[BaseAlgorithm]) – An RL algorithm to train on the resulting reward model (optional).
- is_stationary (bool) – if True, share same density models for all timesteps; if False, use a different density model for each timestep. A non-stationary model is particularly likely to be useful when using STATE_DENSITY, to encourage agent to imitate entire trajectories, not just a few states that have high frequency in the demonstration dataset. If
non-stationary, demonstrations must be trajectories, not transitions (which do not contain timesteps).

- **standardise_inputs** (bool) – if True, then the inputs to the reward model will be standardised to have zero mean and unit variance over the demonstration trajectories. Otherwise, inputs will be passed to the reward model with their ordinary scale.

- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

- **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

```python
buffering_wrapper: BufferingWrapper
density_type: DensityType
is_stationary: bool
kernel: str
kernel_bandwidth: float

property policy: BasePolicy

Returns a policy imitating the demonstration data.

**Return type**
BasePolicy

rl_algo: Optional[BaseAlgorithm]

set_demonstrations(demonstrations)

Sets the demonstration data.

**Return type**
None

standardise: bool
test_policy(*, n_trajectories=10, true_reward=True)

Test current imitation policy on environment & give some rollout stats.

**Parameters**

- **n_trajectories** (int) – number of rolled-out trajectories.

- **true_reward** (bool) – should this use ground truth reward from underlying environment (True), or imitation reward (False)?

**Returns**

rollout statistics collected by
imitation.utils.rollout.rollout_stats()

**Return type**
dict

2.1. imitation
train()
   Fits the density model to demonstration data self.transitions.

   Return type
   None

train_policy(n_timesteps=1000000, **kwargs)
   Train the imitation policy for a given number of timesteps.

   Parameters
   • n_timesteps (int) – number of timesteps to train the policy for.
   • kwargs (dict) – extra arguments that will be passed to the learn() method of the imitation RL model. Refer to Stable Baselines docs for details.

   Return type
   None

transitions: Dict[Optional[int], ndarray]

venv: VecEnv

venv_wrapped: RewardVecEnvWrapper

wrapper_callback: WrappedRewardCallback

class imitation.algorithms.density.DensityType(value)
   Bases: Enum
   Input type the density model should use.

   STATE_ACTION_DENSITY = 2
      Density on (s,a) pairs.

   STATE_DENSITY = 1
      Density on state s.

   STATE_STATE_DENSITY = 3
      Density on (s,s') pairs.

imitation.algorithms.mce_irl

Finite-horizon tabular Maximum Causal Entropy IRL.

Follows the description in chapters 9 and 10 of Brian Ziebart’s PhD thesis.

Functions

# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
**mce_occupancy_measures** (env, *(reward, pi, ...)) Calculate state visitation frequency $D_s$ for each state $s$ under a given policy $\pi$.

**mce_partition_fh** (env, *(reward, discount)) Performs the soft Bellman backup for a finite-horizon MDP.

**squeeze_r** (r_output) Squeeze a reward output tensor down to one dimension, if necessary.

### Classes

**MCEIRL** (demonstrations, env, reward_net, rng) Tabular MCE IRL.

**TabularPolicy** (state_space, action_space, pi, rng) A tabular policy.

class imitation.algorithms.mce_irl.MCEIRL(demonstrations, env, reward_net, rng, optimizer_cls=<class 'torch.optim.adam.Adam'>, optimizer_kwargs=None, discount=1.0, l1_eps=0.001, l2_eps=0.0001, log_interval=100, *, custom_logger=None)

Bases: DemonstrationAlgorithm[TransitionsMinimal]

Tabular MCE IRL.

Reward is a function of observations, but policy is a function of states.

The “observations” effectively exist just to let MCE IRL learn a reward in a reasonable feature space, giving a helpful inductive bias, e.g. that similar states have similar reward.

Since we are performing planning to compute the policy, there is no need for function approximation in the policy.

__init__ (demonstrations, env, reward_net, rng, optimizer_cls=<class 'torch.optim.adam.Adam'>, optimizer_kwargs=None, discount=1.0, l1_eps=0.001, l2_eps=0.0001, log_interval=100, *, custom_logger=None)

Creates MCE IRL.

**Parameters**

- **demonstrations** (Union[ndarray, Iterable[Trajectory], Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal, None) – Demonstrations from an expert (optional). Can be a sequence of trajectories, or transitions, an iterable over mappings that represent a batch of transitions, or a state occupancy measure. The demonstrations must have observations one-hot coded unless demonstrations is a state-occupancy measure.

- **env** (TabularModelPOMDP) – a tabular MDP.

- **rng** (Generator) – random state used for sampling from policy.

- **reward_net** (RewardNet) – a neural network that computes rewards for the supplied observations.

- **optimizer_cls** (Type[Optimizer]) – optimizer to use for supervised training.

- **optimizer_kwargs** (Optional[Mapping[str, Any]]) – keyword arguments for optimizer construction.

- **discount** (float) – the discount factor to use when computing occupancy measure. If not 1.0 (undiscounted), then demonstrations must either be a (discounted) state-occupancy

---

2.1. imitation
measure, or trajectories. Transitions are not allowed as we cannot discount them appropriately without knowing the timestep they were drawn from.

- **\texttt{linf}\_eps** (float) – optimisation terminates if the $l_{\infty}$ distance between the demonstrator’s state occupancy measure and the state occupancy measure for the current reward falls below this value.
- **\texttt{grad}\_l2\_eps** (float) – optimisation also terminates if the $ell_2$ norm of the MCE IRL gradient falls below this value.
- **\texttt{log}\_interval** (Optional[int]) – how often to log current loss stats (using logging). None to disable.
- **\texttt{custom}\_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

Raises

- **ValueError** – if the env horizon is not finite (or an integer).

demo_state_om: Optional[ndarray]

property policy: BasePolicy

Returns a policy imitating the demonstration data.

Return type

BasePolicy

set_demonstrations(demonstrations)

Sets the demonstration data.

Changing the demonstration data on-demand can be useful for interactive algorithms like DAgger.

Parameters

- **demonstrations** (Union[ndarray, Iterable[Trajectory], Iterable[Mapping[str, Union[ndarray, Tensor]]], TransitionsMinimal]) – Either a Torch DataLoader, any other iterator that yields dictionaries containing “obs” and “acts” Tensors or NumPy arrays, TransitionKind instance, or a Sequence of Trajectory objects.

Return type

None

train(max_iter=1000)

Runs MCE IRL.

Parameters

- **max\_iter** (int) – The maximum number of iterations to train for. May terminate earlier if self.linf\_eps or self.grad\_l2\_eps thresholds are reached.

Return type

ndarray

Returns

State occupancy measure for the final reward function. self.reward\_net and self.optimizer will be updated in-place during optimisation.

class imitation.algorithms.mce_irl.TabularPolicy(state_space, action_space, pi, rng)

Bases: BasePolicy

A tabular policy. Cannot be trained – prediction only.
__init__(state_space, action_space, pi, rng)

Builds TabularPolicy.

Parameters

- **state_space** (Space) – The state space of the environment.
- **action_space** (Space) – The action space of the environment.
- **pi** (ndarray) – A tabular policy. Three-dimensional array, where \( \pi[t,s,a] \) is the probability of taking action \( a \) at state \( s \) at timestep \( t \).
- **rng** (Generator) – Random state, used for sampling when predict is called with \( \text{deterministic}=\text{False} \).

forward(observation, deterministic=False)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

Return type

NoReturn

pi: ndarray

predict(observation, state=None, episode_start=None, deterministic=False)

Predict action to take in given state.

Arguments follow SB3 naming convention as this is an SB3 policy. In this convention, observations are returned by the environment, and state is a hidden state used by the policy (used by us to keep track of timesteps).

What is **observation** here is a state in the underlying MDP, and would be called **state** elsewhere in this file.

Parameters

- **observation** (Union[ndarray, Mapping[str, ndarray]]) – States in the underlying MDP.
- **state** (Optional[Tuple[ndarray, ...]]) – Hidden states of the policy – used to represent timesteps by us.
- **episode_start** (Optional[ndarray]) – Has episode completed?
- **deterministic** (bool) – If true, pick action with highest probability; otherwise, sample.

Return type

Tuple[ndarray, Optional[Tuple[ndarray, ...]]]

Returns

Tuple of the actions and new hidden states.

rng: Generator
imitation

**set_pi** *(pi)*
Sets tabular policy to *pi*.

**Return type**
None

**mce_occupancy_measures** *(env, *, reward=None, pi=None, discount=1.0)*
Calculate state visitation frequency *D* for each state *s* under a given policy *pi*.
You can get *pi* from mce_partition_fh.

**Parameters**

- **env** *(TabularModelPOMDP)* – a tabular MDP.
- **reward** *(Optional[ndarray]*) – reward matrix. Defaults is env.reward_matrix.
- **pi** *(Optional[ndarray]*) – policy to simulate. Defaults to soft-optimal policy w.r.t reward matrix.
- **discount** *(float)* – rate to discount the cumulative occupancy measure *D*.

**Return type**
Tuple[ndarray, ndarray]

**Returns**

Tuple of *D* (ndarray) and *Dcum* (ndarray). *D* is of shape (env.horizon, env.n_states) and records the probability of being in a given state at a given timestep. *Dcum* is of shape (env.n_states,) and records the expected discounted number of times each state is visited.

**Raises**
ValueError – if env.horizon is None (infinite horizon).

**mce_partition_fh** *(env, *, reward=None, discount=1.0)*
Performs the soft Bellman backup for a finite-horizon MDP.
Calculates *V*\^\{soft\}, *Q*\^\{soft\}, and *pi* using recurrences (9.1), (9.2), and (9.3) from Ziebart (2010).

**Parameters**

- **env** *(TabularModelPOMDP)* – a tabular, known-dynamics MDP.
- **reward** *(Optional[ndarray]*) – a reward matrix. Defaults to env.reward_matrix.
- **discount** *(float)* – discount rate.

**Return type**
Tuple[ndarray, ndarray, ndarray]

**Returns**

(*V*, *Q*, *pi*) corresponding to the soft values, Q-values and MCE policy. *V* is a 2d array, indexed *V*[t,s]. *Q* is a 3d array, indexed *Q*[t,s,a]. *pi* is a 3d array, indexed *pi*[t,s,a].

**Raises**
ValueError – if env.horizon is None (infinite horizon).

**squeeze_r** *(r_output)*
Squeeze a reward output tensor down to one dimension, if necessary.

**Parameters**

- **r_output** *(th.Tensor)* – output of reward model. Can be either 1D ([n_states]) or 2D ([n_states, 1]).

**Return type**
Tensor
Returns
squeezed reward of shape `[n_states]`.

**imitation.algorithms.preference_comparisons**

Learning reward models using preference comparisons.
Trains a reward model and optionally a policy based on preferences between trajectory fragments.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```python
get_base_model(reward_model)
    rtype
    RewardNet

preference_collate_fn(batch)
    rtype
    Tuple[Sequence[Tuple[TrajectoryWithRew, TrajectoryWithRew]], ndarray]
```
Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ActiveSelectionFragmenter</strong> (preference_model, ...)</td>
<td>Sample fragments of trajectories based on active selection.</td>
</tr>
<tr>
<td><strong>AgentTrainer</strong> (algorithm, reward_fn, venv, rng)</td>
<td>Wrapper for training an SB3 algorithm on an arbitrary reward function.</td>
</tr>
<tr>
<td><strong>BasicRewardTrainer</strong> (preference_model, loss, rng)</td>
<td>Train a basic reward model.</td>
</tr>
<tr>
<td><strong>CrossEntropyRewardLoss()</strong></td>
<td>Compute the cross entropy reward loss.</td>
</tr>
<tr>
<td><strong>EnsembleTrainer</strong> (preference_model, loss, rng)</td>
<td>Train a reward ensemble.</td>
</tr>
<tr>
<td><strong>Fragmenter</strong> ([custom_logger])</td>
<td>Class for creating pairs of trajectory fragments from a set of trajectories.</td>
</tr>
<tr>
<td><strong>LossAndMetrics</strong> (loss, metrics)</td>
<td>Loss and auxiliary metrics for reward network training.</td>
</tr>
<tr>
<td><strong>PreferenceComparisons</strong> (trajectory_generator, ...)</td>
<td>Main interface for reward learning using preference comparisons.</td>
</tr>
<tr>
<td><strong>PreferenceDataset</strong> ([max_size])</td>
<td>A PyTorch Dataset for preference comparisons.</td>
</tr>
<tr>
<td><strong>PreferenceGatherer</strong> ([rng, custom_logger])</td>
<td>Base class for gathering preference comparisons between trajectory fragments.</td>
</tr>
<tr>
<td><strong>PreferenceModel</strong> ([model[, noise_prob, ...]])</td>
<td>Class to convert two fragments' rewards into preference probability.</td>
</tr>
<tr>
<td><strong>RandomFragmenter</strong> ([rng[, warning_threshold, ...]])</td>
<td>Sample fragments of trajectories uniformly at random with replacement.</td>
</tr>
<tr>
<td><strong>RewardLoss()</strong></td>
<td>A loss function over preferences.</td>
</tr>
<tr>
<td><strong>RewardTrainer</strong> ([preference_model[, custom_logger]])</td>
<td>Abstract base class for training reward models using preference comparisons.</td>
</tr>
<tr>
<td><strong>SyntheticGatherer</strong> ([temperature, ...])</td>
<td>Computes synthetic preferences using ground-truth environment rewards.</td>
</tr>
<tr>
<td><strong>TrajectoryDataset</strong> ([trajectories, rng[, ...]])</td>
<td>A fixed dataset of trajectories.</td>
</tr>
<tr>
<td><strong>TrajectoryGenerator</strong> ([custom_logger])</td>
<td>Generator of trajectories with optional training logic.</td>
</tr>
</tbody>
</table>

```python
class imitation.algorithms.preference_comparisons.ActiveSelectionFragmenter(preference_model, base_fragmenter, fragment_sample_factor, uncertainty_on='logit', custom_logger=None)
```

Bases: **Fragmenter**

Sample fragments of trajectories based on active selection.

Actively picks the fragment pairs with the highest uncertainty (variance) of rewards/probabilities/predictions from ensemble model.

```python
__init__(preference_model, base_fragmenter, fragment_sample_factor, uncertainty_on='logit', custom_logger=None)
```

Initialize the active selection fragmenter.

**Parameters**

- **preference_model** (*PreferenceModel*) – an ensemble model that predicts the preference of the first fragment over the other.

- **base_fragmenter** (*Fragmenter*) – fragmenter instance to get fragment pairs from trajectories
• **fragment_sample_factor** (float) – the factor of the number of fragment pairs to sample from the base fragmenter

• **uncertainty_on** (str) – the variable to calculate the variance on. Can be logit|probability|label.

• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

Raises

**ValueError** – Preference model not wrapped over an ensemble of networks.

**raise_uncertainty_on_not_supported()**

Return type

NoReturn

**property uncertainty_on: str**

Return type

str

**variance_estimate**(rews1, rews2)

Gets the variance estimate from the rewards of a fragment pair.

Parameters

• **rews1** (Tensor) – rewards obtained by all the ensemble models for the first fragment.
  Shape - (fragment_length, num_ensemble_members)

• **rews2** (Tensor) – rewards obtained by all the ensemble models for the second fragment.
  Shape - (fragment_length, num_ensemble_members)

Return type

float

Returns

the variance estimate based on the **uncertainty_on** flag.

class imitation.algorithms.preference_comparisons.AgentTrainer(algorithm, reward_fn, venv, rng, 
exploration_frac=0.0,  
switch_prob=0.5, 
random_prob=0.5,  
custom_logger=None)

Bases: TrajectoryGenerator

Wrapper for training an SB3 algorithm on an arbitrary reward function.

**__init__**(algorithm, reward_fn, venv, rng, exploration_frac=0.0, switch_prob=0.5, random_prob=0.5, 
custom_logger=None)

Initialize the agent trainer.

Parameters

• **algorithm** (BaseAlgorithm) – the stable-baselines algorithm to use for training.

• **reward_fn** (Union[RewardFn, RewardNet]) – either a RewardFn or a RewardNet instance that will supply the rewards used for training the agent.

• **venv** (VecEnv) – vectorized environment to train in.

• **rng** (Generator) – random number generator used for exploration and for sampling.
• **exploration_frac** (float) – fraction of the trajectories that will be generated partially randomly rather than only by the agent when sampling.

• **switch_prob** (float) – the probability of switching the current policy at each step for the exploratory samples.

• **random_prob** (float) – the probability of picking the random policy when switching during exploration.

• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

**property logger:**  
*HierarchicalLogger*

**Return type**  
*HierarchicalLogger*

**sample(steps)**  
Sample a batch of trajectories.

**Parameters**  
steps (int) – All trajectories taken together should have at least this many steps.

**Return type**  
*Sequence[TrajectoryWithRew]*

**Returns**  
A list of sampled trajectories with rewards (which should be the environment rewards, not ones from a reward model).

**train(steps, **kwargs)**  
Train the agent using the reward function specified during instantiation.

**Parameters**  

• steps (int) – number of environment timesteps to train for

• **kwargs** – other keyword arguments to pass to BaseAlgorithm.train()

**Raises**  
*RuntimeError* – Transitions left in self.buffering_wrapper; call self.sample first to clear them.

**Return type**  
*None*

**class imitation.algorithms.preference_comparisons.BasicRewardTrainer**(preference_model, loss, rng, batch_size=32, minibatch_size=None, epochs=1, lr=0.001, custom_logger=None, regularizer_factory=None)

**Bases:** RewardTrainer

Train a basic reward model.

**__init__**(preference_model, loss, rng, batch_size=32, minibatch_size=None, epochs=1, lr=0.001, custom_logger=None, regularizer_factory=None)

Initialize the reward model trainer.

**Parameters**
• **preference_model** (PreferenceModel) – the preference model to train the reward network.

• **loss** (RewardLoss) – the loss to use

• **rng** (Generator) – the random number generator to use for splitting the dataset into training and validation.

• **batch_size** (int) – number of fragment pairs per batch

• **minibatch_size** (Optional[int]) – size of minibatch to calculate gradients over. The gradients are accumulated until batch_size examples are processed before making an optimization step. This is useful in GPU training to reduce memory usage, since fewer examples are loaded into memory at once, facilitating training with larger batch sizes, but is generally slower. Must be a factor of batch_size. Optional, defaults to batch_size.

• **epochs** (int) – number of epochs in each training iteration (can be adjusted on the fly by specifying an epoch_multiplier in self.train() if longer training is desired in specific cases).

• **lr** (float) – the learning rate

• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

• **regularizer_factory** (Optional[RegularizerFactory]) – if you would like to apply regularization during training, specify a regularizer factory here. The factory will be used to construct a regularizer. See imitation.regularization. RegularizerFactory for more details.

Raises

**ValueError** – if the batch size is not a multiple of the minibatch size.

```python
class imitation.algorithms.preference_comparisons.CrossEntropyRewardLoss
Bases: RewardLoss

Compute the cross entropy reward loss.

_init_(self)
    Create cross entropy reward loss.

forward(self, fragment_pairs, preferences, preference_model)
    Computes the loss.
```

Parameters

• **fragment_pairs** (Sequence[Tuple[Trajectory, Trajectory]]) – Batch consisting of pairs of trajectory fragments.

• **preferences** (ndarray) – The probability that the first fragment is preferred over the second. Typically 0, 1 or 0.5 (tie).
- **preference_model** (*PreferenceModel*) – model to predict the preferred fragment from a pair.

**Return type**

*LossAndMetrics*

**Returns**

The cross-entropy loss between the probability predicted by the reward model and the target probabilities in `preferences`. Metrics are accuracy, and `gt_reward_loss`, if the ground truth reward is available.

**training**: `bool`

```
class imitation.algorithms.preference_comparisons.EnsembleTrainer(preference_model, loss, rng, 
batch_size=32, minibatch_size=None, 
epochs=1, lr=0.001, 
custom_logger=None, 
regularizer_factory=None)
```

Bases: `BasicRewardTrainer`

Train a reward ensemble.

```
__init__(preference_model, loss, rng, batch_size=32, minibatch_size=None, epochs=1, lr=0.001, 
custom_logger=None, regularizer_factory=None)
```

Initialize the reward model trainer.

**Parameters**

- **preference_model** (*PreferenceModel*) – the preference model to train the reward network.

- **loss** (*RewardLoss*) – the loss to use

- **rng** (*Generator*) – random state for the internal RNG used in bagging

- **batch_size** (*int*) – number of fragment pairs per batch

- **minibatch_size** (*Optional[int]*) – size of minibatch to calculate gradients over. The gradients are accumulated until `batch_size` examples are processed before making an optimization step. This is useful in GPU training to reduce memory usage, since fewer examples are loaded into memory at once, facilitating training with larger batch sizes, but is generally slower. Must be a factor of `batch_size`. Optional, defaults to `batch_size`.

- **epochs** (*int*) – number of epochs in each training iteration (can be adjusted on the fly by specifying an `epoch_multiplier` in `self.train()` if longer training is desired in specific cases).

- **lr** (*float*) – the learning rate

- **custom_logger** (*Optional[HierarchicalLogger]*) – Where to log to; if None (default), creates a new logger.

- **regularizer_factory** (*Optional[RegularizerFactory]*) – A factory for creating a regularizer. If None, no regularization is used.

**Raises**

- **TypeError** – if model is not a RewardEnsemble.

**property logger**: `HierarchicalLogger`

**Return type**

*HierarchicalLogger*

---

126 Chapter 2. API Reference
**regularizer**: `Optional[Regularizer]`

**class** imitation.algorithms.preference_comparisons.Fragmenter(*custom_logger=None*)

Bases: ABC

Class for creating pairs of trajectory fragments from a set of trajectories.

**__init__**(*custom_logger=None*)

Initialize the fragmenter.

**Parameters**

`custom_logger` (`Optional[HierarchicalLogger]`) – Where to log to; if None (default), creates a new logger.

**class** imitation.algorithms.preference_comparisons.LossAndMetrics(*loss: Tensor, metrics: Mapping[str, Tensor]*)

Bases: tuple

Loss and auxiliary metrics for reward network training.

**loss**: `Tensor`

**metrics**: `Mapping[str, Tensor]`

**class** imitation.algorithms.preference_comparisons.PreferenceComparisons(*trajectory_generator, reward_model, num_iterations, fragmenter=None, preference_gatherer=None, reward_trainer=None, comparison_queue_size=None, fragment_length=100, transition_oversampling=1, initial_comparison_frac=0.1, initial_epoch_multiplier=200.0, custom_logger=None, allow_variable_horizon=False, rng=None, query_schedule='hyperbolic')

Bases: BaseImitationAlgorithm

Main interface for reward learning using preference comparisons.

**__init__**(*trajectory_generator, reward_model, num_iterations, fragmenter=None, preference_gatherer=None, reward_trainer=None, comparison_queue_size=None, fragment_length=100, transition_oversampling=1, initial_comparison_frac=0.1, initial_epoch_multiplier=200.0, custom_logger=None, allow_variable_horizon=False, rng=None, query_schedule='hyperbolic')

Initialize the preference comparison trainer.

The loggers of all subcomponents are overridden with the logger used by this class.
Parameters

- **trajectory_generator** *(TrajectoryGenerator)* – generates trajectories while optionally training an RL agent on the learned reward function (can also be a sampler from a static dataset of trajectories though).

- **reward_model** *(RewardNet)* – a RewardNet instance to be used for learning the reward

- **num_iterations** *(int)* – number of times to train the agent against the reward model and then train the reward model against newly gathered preferences.

- **fragmenter** *(Optional*[Fragmenter]*) – takes in a set of trajectories and returns pairs of fragments for which preferences will be gathered. These fragments could be random, or they could be selected more deliberately (active learning). Default is a random fragmenter.

- **preference_gatherer** *(Optional*[PreferenceGatherer]*) – how to get preferences between trajectory fragments. Default (and currently the only option) is to use synthetic preferences based on ground-truth rewards. Human preferences could be implemented here in the future.

- **reward_trainer** *(Optional*[RewardTrainer]*) – trains the reward model based on pairs of fragments and associated preferences. Default is to use the preference model and loss function from DRLHP.

- **comparison_queue_size** *(Optional*[int]*) – the maximum number of comparisons to keep in the queue for training the reward model. If None, the queue will grow without bound as new comparisons are added.

- **fragment_length** *(int)* – number of timesteps per fragment that is used to elicit preferences

- **transition_oversampling** *(float)* – factor by which to oversample transitions before creating fragments. Since fragments are sampled with replacement, this is usually chosen > 1 to avoid having the same transition in too many fragments.

- **initial_comparison_frac** *(float)* – fraction of the total comparisons argument to train() that will be sampled before the rest of training begins (using a randomly initialized agent). This can be used to pretrain the reward model before the agent is trained on the learned reward, to help avoid irreversibly learning a bad policy from an untrained reward. Note that there will often be some additional pretraining comparisons since comparisons_per_iteration won’t exactly divide the total number of comparisons. How many such comparisons there are depends discontinuously on total_comparisons and comparisons_per_iteration.

- **initial_epoch_multiplier** *(float)* – before agent training begins, train the reward model for this many more epochs than usual (on fragments sampled from a random agent).

- **custom_logger** *(Optional*[HierarchicalLogger]*) – Where to log to; if None (default), creates a new logger.

- **allow_variable_horizon** *(bool)* – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

- **rng** *(Optional*[Generator]*) – random number generator to use for initializing subcomponents such as fragmenter. Only used when default components are used; if you instantiate your own fragmenter, preference gatherer, etc., you are responsible for seeding them!
• **query_schedule** (Union[str, Callable[[float, float]]) – one of (“constant”, “hyperbolic”, “inverse_quadratic”), or a function that takes in a float between 0 and 1 inclusive, representing a fraction of the total number of timesteps elapsed up to some time T, and returns a potentially unnormalized probability indicating the fraction of total_comparisons that should be queried at that iteration. This function will be called num_iterations times in __init__() with values from np.linspace(0, 1, num_iterations) as input. The outputs will be normalized to sum to 1 and then used to apportion the comparisons among the num_iterations iterations.

*Raises*

- **ValueError** – if query_schedule is not a valid string or callable.

**allow_variable_horizon**: bool

If True, allow variable horizon trajectories; otherwise error if detected.

**train**(total_timesteps, total_comparisons, callback=None)

Train the reward model and the policy if applicable.

**Parameters**

- **total_timesteps** (int) – number of environment interaction steps
- **total_comparisons** (int) – number of preferences to gather in total
- **callback** (Optional[Callable[[int], None]]) – callback functions called at the end of each iteration

**Return type**

Mapping[str, Any]

**Returns**

A dictionary with final metrics such as loss and accuracy of the reward model.

class imitation.algorithms.preference_comparisons.PreferenceDataset(max_size=None)

Bases: Dataset

A PyTorch Dataset for preference comparisons.

Each item is a tuple consisting of two trajectory fragments and a probability that fragment 1 is preferred over fragment 2.

This dataset is meant to be generated piece by piece during the training process, which is why data can be added via the .push() method.

**__init__**(max_size=None)

Builds an empty PreferenceDataset.

**Parameters**

- **max_size** (Optional[int]) – Maximum number of preference comparisons to store in the dataset. If None (default), the dataset can grow indefinitely. Otherwise, the dataset acts as a FIFO queue, and the oldest comparisons are evicted when push() is called and the dataset is at max capacity.

**static load**(path)

**Return type**

PreferenceDataset

**push**(fragments, preferences)

Add more samples to the dataset.

**Parameters**
• **fragments** (Sequence[Tuple[TrajectoryWithRew, TrajectoryWithRew]]) – list of pairs of trajectory fragments to add

• **preferences** (ndarray) – corresponding preference probabilities (probability that fragment 1 is preferred over fragment 2)

Raises

**ValueError** – *preferences* shape does not match *fragments* or has non-float32 dtype.

Return type

None

```
save(path)
```

Return type

None

```python
class imitation.algorithms.preference_comparisons.PreferenceGatherer(rng=None, custom_logger=None)
```

Bases: ABC

Base class for gathering preference comparisons between trajectory fragments.

```python
__init__(rng=None, custom_logger=None)
```

Initializes the preference gatherer.

Parameters

• **rng** (Optional[Generator]) – random number generator, if applicable.

• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

```python
class imitation.algorithms.preference_comparisons.PreferenceModel(model, noise_prob=0.0, discount_factor=1.0, threshold=50)
```

Bases: Module

Class to convert two fragments’ rewards into preference probability.

```python
__init__(model, noise_prob=0.0, discount_factor=1.0, threshold=50)
```

Create Preference Prediction Model.

Parameters

• **model** (*RewardNet*) – base model to compute reward.

• **noise_prob** (*float*) – assumed probability with which the preference is uniformly random (used for the model of preference generation that is used for the loss).

• **discount_factor** (*float*) – the model of preference generation uses a softmax of returns as the probability that a fragment is preferred. This is the discount factor used to calculate those returns. Default is 1, i.e. undiscounted sums of rewards (which is what the DRLHP paper uses).

• **threshold** (*float*) – the preference model used to compute the loss contains a softmax of returns. To avoid overflows, we clip differences in returns that are above this threshold. This threshold is therefore in logspace. The default value of 50 means that probabilities below 2e-22 are rounded up to 2e-22.

Raises

**ValueError** – if *RewardEnsemble* is wrapped around a class other than *AddSTDRewardWrapper*. 130 Chapter 2. API Reference
forward(fragment_pairs)

Computes the preference probability of the first fragment for all pairs.

Note: This function passes the gradient through for non-ensemble models.
For an ensemble model, this function should not be used for loss calculation. It can be used in case where passing the gradient is not required such as during active selection or inference time. Therefore, the EnsembleTrainer passes each member network through this function instead of passing the EnsembleNetwork object with the use of ensemble_member_index.

Parameters

fragment_pairs (Sequence[Tuple[Trajectory, Trajectory]]) – batch of pair of fragments.

Return type

tuple[Tensor, Optional[Tensor]]

Returns

A tuple with the first element as the preference probabilities for the first fragment for all fragment pairs given by the network(s). If the ground truth rewards are available, it also returns gt preference probabilities in the second element of the tuple (else None). Reward probability shape - (num_fragment_pairs,) for non-ensemble reward network and (num_fragment_pairs, num_networks) for an ensemble of networks.

probability(rews1, rews2)

Computes the Boltzmann rational probability the first trajectory is best.

Parameters

rews1 (Tensor) – array/matrix of rewards for the first trajectory fragment. matrix for ensemble models and array for non-ensemble models.
rews2 (Tensor) – array/matrix of rewards for the second trajectory fragment. matrix for ensemble models and array for non-ensemble models.

Return type

Tensor

Returns

The softmax of the difference between the (discounted) return of the first and second trajectory. Shape - (num_ensemble_members,) for ensemble model and () for non-ensemble model which is a torch scalar.

rewards(transitions)

Computes the reward for all transitions.

Parameters

transitions (Transitions) – batch of obs-act-obs-done for a fragment of a trajectory.

Return type

Tensor

Returns

The reward given by the network(s) for all the transitions. Shape - (num_transitions,) for Single reward network and (num_transitions, num_networks) for ensemble of networks.

class imitation.algorithms.preference_comparisons.RandomFragmenter(rng, warning_threshold=10, custom_logger=None)
Sample fragments of trajectories uniformly at random with replacement.

Note that each fragment is part of a single episode and has a fixed length. This leads to a bias: transitions at the beginning and at the end of episodes are less likely to occur as part of fragments (this affects the first and last fragment_length transitions).

An additional bias is that trajectories shorter than the desired fragment length are never used.

```python
__init__(rng, warning_threshold=10, custom_logger=None)
```

Initialize the fragmenter.

- **rng** (Generator) – the random number generator
- **warning_threshold** (int) – give a warning if the number of available transitions is less than this many times the number of required samples. Set to 0 to disable this warning.
- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

class imitation.algorithms.preference_comparisons.RewardLoss

A loss function over preferences.

```python
abstract forward(fragment_pairs, preferences, preference_model)
```

Computes the loss.

- **fragment_pairs** (Sequence[Tuple[Trajectory, Trajectory]]) – Batch consisting of pairs of trajectory fragments.
- **preferences** (ndarray) – The probability that the first fragment is preferred over the second. Typically 0, 1 or 0.5 (tie).
- **preference_model** (PreferenceModel) – model to predict the preferred fragment from a pair.

Returns: # noqa: DAR202

- **loss**: the loss metrics: a dictionary of metrics that can be logged

Return type

* LossAndMetrics

training: bool

class imitation.algorithms.preference_comparisons.RewardTrainer(preference_model, custom_logger=None)

Abstract base class for training reward models using preference comparisons.

This class contains only the actual reward model training code, it is not responsible for gathering trajectories and preferences or for agent training (see :class: PreferenceComparisons for that).
__init__(preference_model, custom_logger=None)

Initialize the reward trainer.

Parameters

- **preference_model** (PreferenceModel) – the preference model to train the reward network.
- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

property logger:  

Return type  
HierarchicalLogger

train(dataset, epoch_multiplier=1.0)

Train the reward model on a batch of fragment pairs and preferences.

Parameters

- **dataset** (PreferenceDataset) – the dataset of preference comparisons to train on.
- **epoch_multiplier** (float) – how much longer to train for than usual (measured relatively).

Return type  
None

class imitation.algorithms.preference_comparisons.SyntheticGatherer(temperature=1,  
discount_factor=1,  
sample=True, rng=None,  
threshold=50,  
custom_logger=None)

Computes synthetic preferences using ground-truth environment rewards.

__init__(temperature=1, discount_factor=1, sample=True, rng=None, threshold=50,  
custom_logger=None)

Initialize the synthetic preference gatherer.

Parameters

- **temperature** (float) – the preferences are sampled from a softmax, this is the temperature used for sampling. temperature=0 leads to deterministic results (for equal rewards, 0.5 will be returned).
- **discount_factor** (float) – discount factor that is used to compute how good a fragment is. Default is to use undiscounted sums of rewards (as in the DRLHLP paper).
- **sample** (bool) – if True (default), the preferences are 0 or 1, sampled from a Bernoulli distribution (or 0.5 in the case of ties with zero temperature). If False, then the underlying Bernoulli probabilities are returned instead.
- **rng** (Optional[Generator]) – random number generator, only used if temperature > 0 and sample=True
- **threshold** (float) – preferences are sampled from a softmax of returns. To avoid overflows, we clip differences in returns that are above this threshold (after multiplying with temperature). This threshold is therefore in logspace. The default value of 50 means that probabilities below 2e-22 are rounded up to 2e-22.
• **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

Raises
- **ValueError** – if `sample` is true and no random state is provided.

```python
class imitation.algorithms.preference_comparisons.TrajectoryDataset(trajectories, rng, custom_logger=None)
```

Bases: `TrajectoryGenerator`

A fixed dataset of trajectories.

```python
__init__(trajectories, rng, custom_logger=None)
```

Creates a dataset loaded from `path`.

Parameters
- **trajectories** (Sequence[TrajectoryWithRew]) – the dataset of rollouts.
- **rng** (Generator) – RNG used for shuffling dataset.
- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

```python
sample(steps)
```

Sample a batch of trajectories.

Parameters
- **steps** (int) – All trajectories taken together should have at least this many steps.

Return type
- Sequence[TrajectoryWithRew]

Returns
- A list of sampled trajectories with rewards (which should be the environment rewards, not ones from a reward model).

```python
class imitation.algorithms.preference_comparisons.TrajectoryGenerator(custom_logger=None)
```

Bases: `ABC`

Generator of trajectories with optional training logic.

```python
__init__(custom_logger=None)
```

Builds TrajectoryGenerator.

Parameters
- **custom_logger** (Optional[HierarchicalLogger]) – Where to log to; if None (default), creates a new logger.

```
property logger: HierarchicalLogger
```

Return type
- HierarchicalLogger

```python
abstract sample(steps)
```

Sample a batch of trajectories.

Parameters
- **steps** (int) – All trajectories taken together should have at least this many steps.

Return type
- Sequence[TrajectoryWithRew]
### Returns
A list of sampled trajectories with rewards (which should be the environment rewards, not ones from a reward model).

#### `train(steps, **kwargs)`
Train an agent if the trajectory generator uses one.

By default, this method does nothing and doesn’t need to be overridden in subclasses that don’t require training.

**Parameters**
- `steps` (int) – number of environment steps to train for.
- `**kwargs` – additional keyword arguments to pass on to the training procedure.

**Return type**
None

#### `get_base_model(reward_model)`

**Return type**
`RewardNet`

#### `preference_collate_fn(batch)`

**Return type**
`Tuple[Sequence[Tuple[TrajectoryWithRew, TrajectoryWithRew]], ndarray]`

## 2.1.2 imitation.data

Modules handling environment data.

For example: types for transitions/trajectories; methods to compute rollouts; buffers to store transitions; helpers for these modules.

### Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imitation.data.buffer</code></td>
<td>Buffers to store NumPy arrays and transitions in.</td>
</tr>
<tr>
<td><code>imitation.data.rollout</code></td>
<td>Methods to collect, analyze and manipulate transition and trajectory rollouts.</td>
</tr>
<tr>
<td><code>imitation.data.types</code></td>
<td>Types and helper methods for transitions and trajectories.</td>
</tr>
<tr>
<td><code>imitation.data.wrappers</code></td>
<td>Environment wrappers for collecting rollouts.</td>
</tr>
</tbody>
</table>

#### imitation.data.buffer

Buffers to store NumPy arrays and transitions in.
**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```python
num_samples(data) Computes the number of samples contained in data.
```

**Classes**

```python
Buffer(capacity, sample_shapes, dtypes) A FIFO ring buffer for NumPy arrays of a fixed shape and dtype.
```

```python
ReplayBuffer(capacity[, venv, obs_shape, ...]) Buffer for Transitions.
```

class imitation.data.buffer.Buffer(capacity, sample_shapes, dtypes)

Bases: object

A FIFO ring buffer for NumPy arrays of a fixed shape and dtype.
Supports random sampling with replacement.

```python
__init__(capacity, sample_shapes, dtypes)

Constructs a Buffer.
```

**Parameters**

- **capacity** (int) – The number of samples that can be stored.
- **sample_shapes** (Mapping[str, Tuple[int, ...]]) – A dictionary mapping string keys to the shape of samples associated with that key.
- **dtypes** (np.dtype-like) – A dictionary mapping string keys to the dtype of samples associated with that key.

**Raises**

- **KeyError** – sample_shapes and dtypes have different keys.

```python
capacity: int

The number of data samples that can be stored in this buffer.
```

classmethod from_data(data, capacity=None, truncate_ok=False)

Constructs and return a Buffer containing the provided data.
Shapes and dtypes are automatically inferred.

**Parameters**

- **data** (Mapping[str, ndarray]) – A dictionary mapping keys to data arrays. The arrays may differ in their shape, but should agree in the first axis.
- **capacity** (Optional[int]) – The Buffer capacity. If not provided, then this is automatically set to the size of the data, so that the returned Buffer is at full capacity.
- **truncate_ok** (bool) – Whether to error if capacity < the number of samples in data. If False, then only store the last capacity samples from data when overcapacity.
**Examples**

In the following examples, suppose the arrays in `data` are length-1000.

*Buffer* with same capacity as arrays in `data`:

```python
Buffer.from_data(data)
```

*Buffer* with larger capacity than arrays in `data`:

```python
Buffer.from_data(data, 10000)
```

*Buffer* with smaller capacity than arrays in `data`. Without `truncate_ok=True`, `from_data` will error:

```python
Buffer.from_data(data, 5, truncate_ok=True)
```

**Return type**

*Buffer*

**Returns**

Buffer of specified capacity containing provided `data`.

**Raises**

- `ValueError` – `data` is empty.
- `ValueError` – `data` has items mapping to arrays differing in the length of their first axis.

**sample**(*n_samples*)

Uniformly sample `n_samples` samples from the buffer with replacement.

**Parameters**

- `n_samples` (int) – The number of samples to randomly sample.

**Returns**

An array with shape

```
(n_samples) + self.sample_shape.
```

**Return type**

samples (np.ndarray)

**Raises**

- `ValueError` – The buffer is empty.

**sample_shapes**: Mapping[str, Tuple[int, ...]]

The shapes of each data sample stored in this buffer.

**size**()

Returns the number of samples stored in the buffer.

**Return type**

- `int`

**store**(*data, truncate_ok=False*)

Stores new data samples, replacing old samples with FIFO priority.

**Parameters**
• **data** (Mapping[str, ndarray]) – A dictionary mapping keys $k$ to arrays with shape $(n_{samples}) + self.sample_shapes[k]$, where $n_{samples}$ is less than or equal to `self.capacity`.

• **truncate_ok** (bool) – If False, then error if the length of `transitions` is greater than `self.capacity`. Otherwise, store only the final `self.capacity` transitions.

Raises

• **ValueError** – `data` is empty.

• **ValueError** – If $n_{samples}$ is greater than `self.capacity`.

• **ValueError** – `data` is the wrong shape.

Return type

None

class imitation.data.buffer.ReplayBuffer(capacity, venv=None, *, obs_shape=None, act_shape=None, obs_dtype=None, act_dtype=None)

Bases: object

Buffer for Transitions.

__init__((capacity, venv=None, *, obs_shape=None, act_shape=None, obs_dtype=None, act_dtype=None))

Constructs a ReplayBuffer.

Parameters

• **capacity** (int) – The number of samples that can be stored.

• **venv** (Optional[VecEnv]) – The environment whose action and observation spaces can be used to determine the data shapes of the underlying buffers. Mutually exclusive with shape and dtype arguments.

• **obs_shape** (Optional[Tuple[int, ...]]) – The shape of the observation space.

• **act_shape** (Optional[Tuple[int, ...]]) – The shape of the action space.

• **obs_dtype** (Optional[dtype]) – The dtype of the observation space.

• **act_dtype** (Optional[dtype]) – The dtype of the action space.

Raises

• **ValueError** – Couldn’t infer the observation and action shapes and dtypes from the arguments.

• **ValueError** – Specified both venv and shapes/dtypes.

capacity: int

The number of data samples that can be stored in this buffer.

classmethod from_data(transitions, capacity=None, truncate_ok=False)

Construct and return a ReplayBuffer containing the provided data.

Shapes and dtypes are automatically inferred, and the returned ReplayBuffer is ready for sampling.

Parameters

• **transitions** (Transitions) – Transitions to store.

• **capacity** (Optional[int]) – The ReplayBuffer capacity. If not provided, then this is automatically set to the size of the data, so that the returned Buffer is at full capacity.
• **truncate_ok** (bool) – Whether to error if capacity < the number of samples in data. If False, then only store the last capacity samples from data when overcapacity.

## Examples

*ReplayBuffer* with same capacity as arrays in *data*:

```python
ReplayBuffer.from_data(data)
```

*ReplayBuffer* with larger capacity than arrays in *data*:

```python
ReplayBuffer.from_data(data, 10000)
```

*ReplayBuffer* with smaller capacity than arrays in *data*. Without **truncate_ok**=True, *from_data* will error:

```python
ReplayBuffer.from_data(data, 5, truncate_ok=True)
```

### Return type

*ReplayBuffer*

### Returns

A new ReplayBuffer.

### sample(*n_samples*)

Sample obs-act-obs triples.

#### Parameters

- **n_samples** (int) – The number of samples.

#### Return type

*Transitions*

#### Returns

A Transitions named tuple containing *n_samples* transitions.

### size()

Returns the number of samples stored in the buffer.

#### Return type

*Optional*[int]

### store(*transitions, truncate_ok=True*)

Store obs-act-obs triples.

#### Parameters

- **transitions** (*Transitions*) – Transitions to store.
- **truncate_ok** (bool) – If False, then error if the length of *transitions* is greater than *self.capacity*. Otherwise, store only the final *self.capacity* transitions.

#### Raises

*ValueError* – The arguments didn’t have the same length.

#### Return type

None
**imitation.data.buffer.num_samples(data)**

Computes the number of samples contained in `data`.

**Parameters**

- `data` (Mapping[Any, ndarray]) – A Mapping from keys to NumPy arrays.

**Return type**

- int

**Returns**

The unique length of the first dimension of arrays contained in `data`.

**Raises**

- ValueError – The length is not unique.

---

**imitation.data.rollout**

Methods to collect, analyze and manipulate transition and trajectory rollouts.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

discounted_sum(arr, gamma) Calculate the discounted sum of `arr`.

flatten_trajectories(trajectories) Flatten a series of trajectory dictionaries into arrays.

flatten_trajectories_with_rew(trajectories) `rtype` TransitionsWithRew

generate_trajectories(policy, venv, [...]) Generate trajectory dictionaries from a policy and an environment.

generate_transitions(policy, venv, [...]) Generate obs-action-next_obs-reward tuples.

make_min_episodes(n) Terminate after collecting n episodes of data.

make_min_timesteps(n) Terminate at the first episode after collecting n timesteps of data.

make_sample_until([min_timesteps, min_episodes]) Returns a termination condition sampling for a number of timesteps and episodes.

policy_to_callable(policy, venv[, ...]) Converts any policy-like object into a function from observations to actions.

rollout(policy, venv, sample_until, rng, *) Generate policy rollouts.

rollout_stats(trajectories) Calculates various stats for a sequence of trajectories.

unwrap_traj(traj) Uses RolloutInfoWrapper-captured obs and rews to replace fields.
```
**Classes**

### TrajectoryAccumulator

#### Class

```python
class imitation.data.rollout.TrajectoryAccumulator

Bases: object
```

Accumulates trajectories step-by-step.

Useful for collecting completed trajectories while ignoring partially-completed trajectories (e.g. when rolling out a VecEnv to collect a set number of transitions). Each in-progress trajectory is identified by a ‘key’, which enables several independent trajectories to be collected at once. They key can also be left at its default value of None if you only wish to collect one trajectory.

**__init__()**

Initialise the trajectory accumulator.

**add_step(step_dict, key=None)**

Add a single step to the partial trajectory identified by key.

Generally a single step could correspond to, e.g., one environment managed by a VecEnv.

**Parameters**

- **step_dict** (`Mapping[str, Union[numpy.ndarray, Mapping[str, Any]]]`) – dictionary containing information for the current step. Its keys could include any (or all) attributes of a `TrajectoryWithRew` (e.g. “obs”, “acts”, etc.).

- **key** (`Optional[Hashable]`) – key to uniquely identify the trajectory to append to, if working with multiple partial trajectories.

**Return type**

none

**add_steps_and_auto_finish(acts, obs, rews, dones, infos)**

Calls `add_step` repeatedly using acts and the returns from `venv.step`.

Also automatically calls `finish_trajectory()` for each `done == True`. Before calling this method, each environment index key needs to be initialized with the initial observation (usually from `venv.reset()`).

See the body of `util.rollout.generate_trajectory` for an example.

**Parameters**

- **acts** (`ndarray`) – Actions passed into `VecEnv.step()`.

- **obs** (`ndarray`) – Return value from `VecEnv.step(acts)`.

- **rews** (`ndarray`) – Return value from `VecEnv.step(acts)`.

- **dones** (`ndarray`) – Return value from `VecEnv.step(acts)`.

- **infos** (`List[dict]`) – Return value from `VecEnv.step(acts)`.

**Return type**

`List[TrajectoryWithRew]`

**Returns**

A list of completed trajectories. There should be one trajectory for each `True` in the `dones` argument.
**finish_trajectory** *(key, terminal)*  
Complete the trajectory labelled with *key*.

**Parameters**

- **key** *(Hashable)* – key uniquely identifying which in-progress trajectory to remove.
- **terminal** *(bool)* – trajectory has naturally finished (i.e. includes terminal state).

**Returns**

list of completed trajectories popped from  
`self.partial_trajectories`.

**Return type**

`traj`

`imitation.data.rollout.discounted_sum***(arr, gamma)**`

Calculate the discounted sum of *arr*.

If *arr* is an array of rewards, then this computes the return; however, it can also be used to e.g. compute discounted state occupancy measures.

**Parameters**

- **arr** *(ndarray)* – 1 or 2-dimensional array to compute discounted sum over. Last axis is timestep, from current time step (first) to last timestep (last). First axis (if present) is batch dimension.
- **gamma** *(float)* – the discount factor used.

**Return type**

Union[ndarray, float]

**Returns**

The discounted sum over the timestep axis. The first timestep is undiscounted, i.e. we start at $\gamma^0$.

`imitation.data.rollout.flatten_trajectories***(trajectories)**`

Flatten a series of trajectory dictionaries into arrays.

**Parameters**

- **trajectories** *(Sequence[Trajectory])* – list of trajectories.

**Return type**

`Transitions`

**Returns**

The trajectories flattened into a single batch of Transitions.

`imitation.data.rollout.flatten_trajectories_with_rew***(trajectories)**`

**Return type**

`TransitionsWithRew`

`imitation.data.rollout.generate_trajectories***(policy, venv, sample_until, rng, *, deterministic_policy=False)**`

Generate trajectory dictionaries from a policy and an environment.

**Parameters**

- **policy** *(Union[BaseAlgorithm, BasePolicy, Callable[[ndarray, ndarray], None])* – Can be any of the following: 1) A stable_baselines3 policy or algorithm trained on the gym
environment. 2) A Callable that takes an ndarray of observations and returns an ndarray of corresponding actions. 3) None, in which case actions will be sampled randomly.

- **venv** *(VecEnv)* – The vectorized environments to interact with.
- **sample_until** *(Callable[[Sequence[TrajectoryWithRew]], bool])* – A function determining the termination condition. It takes a sequence of trajectories, and returns a bool. Most users will want to use one of *min_episodes* or *min_timesteps*.
- **deterministic_policy** *(bool)* – If True, asks policy to deterministically return action. Note the trajectories might still be non-deterministic if the environment has non-determinism!
- **rng** *(Generator)* – used for shuffling trajectories.

Return type

Sequence[TrajectoryWithRew]

Returns

Sequence of trajectories, satisfying *sample_until*. Additional trajectories may be collected to avoid biasing process towards short episodes; the user should truncate if required.

```
imitation.data.rollout.generate_transitions(policy, venv, n_timesteps, rng, *, truncate=True, **kwargs)
```

Generate obs-action-next_obs-reward tuples.

Parameters

- **policy** *(Union[BaseAlgorithm, BasePolicy, Callable[[ndarray, ndarray], None]])*
  - Can be any of the following: - A stable_baselines3 policy or algorithm trained on the gym environment - A Callable that takes an ndarray of observations and returns an ndarray of corresponding actions - None, in which case actions will be sampled randomly
- **venv** *(VecEnv)* – The vectorized environments to interact with.
- **n_timesteps** *(int)* – The minimum number of timesteps to sample.
- **rng** *(Generator)* – The random state to use for sampling trajectories.
- **truncate** *(bool)* – If True, then drop any additional samples to ensure that exactly *n_timesteps* samples are returned.
- ****kwargs** – Passed-through to *generate_trajectories*.

Return type

TransitionsWithRew

Returns

A batch of Transitions. The length of the constituent arrays is guaranteed to be at least *n_timesteps* (if specified), but may be greater unless *truncate* is provided as we collect data until the end of each episode.

```
imitation.data.rollout.make_min_episodes(n)
```

Terminate after collecting *n* episodes of data.

Parameters

- **n** *(int)* – Minimum number of episodes of data to collect. May overshoot if two episodes complete simultaneously (unlikely).

Return type

Callable[[Sequence[TrajectoryWithRew]], bool]
Returns
A function implementing this termination condition.

imitation.data.rollout.make_min_timesteps(n)
Terminate at the first episode after collecting n timesteps of data.

Parameters
- n (int) – Minimum number of timesteps of data to collect. May overshoot to nearest episode boundary.

Return type
Callable[[Sequence[TrajectoryWithRew]], bool]

Returns
A function implementing this termination condition.

imitation.data.rollout.make_sample_until(min_timesteps=None, min_episodes=None)
Returns a termination condition sampling for a number of timesteps and episodes.

Parameters
- min_timesteps (Optional[int]) – Sampling will not stop until there are at least this many timesteps.
- min_episodes (Optional[int]) – Sampling will not stop until there are at least this many episodes.

Return type
Callable[[Sequence[TrajectoryWithRew]], bool]

Returns
A termination condition.

Raises
ValueError – Neither of n_timesteps and n_episodes are set, or either are non-positive.

imitation.data.rollout.policy_to_callable(policy, venv, deterministic_policy=False)
Converts any policy-like object into a function from observations to actions.

Return type
Callable[[ndarray, ndarray], ndarray]

imitation.data.rollout.rollout(policy, venv, sample_until, rng, *, unwrap=True, exclude_infos=True, verbose=True, **kwargs)
Generate policy rollouts.
The .infos field of each Trajectory is set to None to save space.

Parameters
- policy (Union[BaseAlgorithm, BasePolicy, Callable[[ndarray, ndarray], None], None]) – Can be any of the following: 1) A stable_baselines3 policy or algorithm trained on the gym environment. 2) A Callable that takes an ndarray of observations and returns an ndarray of corresponding actions. 3) None, in which case actions will be sampled randomly.
- venv (VecEnv) – The vectorized environments.
- sample_until (Callable[[Sequence[TrajectoryWithRew]], bool]) – End condition for rollout sampling.
- rng (Generator) – Random state to use for sampling.
• **unwrap** (bool) – If True, then save original observations and rewards (instead of potentially wrapped observations and rewards) by calling **unwrap_traj**.

• **exclude_infos** (bool) – If True, then exclude **infos** from pickle by setting this field to None. Excluding **infos** can save a lot of space during pickles.

• **verbose** (bool) – If True, then print out rollout stats before saving.

• **kwargs** – Passed through to **generate_trajectories**.

**Return type**
Sequence[**TrajectoryWithRew**]

**Returns**
Sequence of trajectories, satisfying **sample_until**. Additional trajectories may be collected to avoid biasing process towards short episodes; the user should truncate if required.

**imitation.data.rollout.rollout_stats**(trajectories)
Calculates various stats for a sequence of trajectories.

**Parameters**
trajectories (Sequence[**TrajectoryWithRew**]) – Sequence of trajectories.

**Return type**
Mapping[str,float]

**Returns**
Dictionary containing n_traj collected (int), along with episode return statistics (keys: monitor_, return_{min,mean,std,max}, float values) and trajectory length statistics (keys: len_{min,mean,std,max}, float values).

return_* values are calculated from environment rewards. monitor_* values are calculated from Monitor-captured rewards, and are only included if the trajectories contain Monitor infos.

**imitation.data.rollout.unwrap_traj**(traj)
Uses RolloutInfoWrapper-captured obs and rews to replace fields.

This can be useful for bypassing other wrappers to retrieve the original obs and rews.

Fails if **infos** is None or if the trajectory was generated from an environment without imitation.data.wrappers.RolloutInfoWrapper

**Parameters**
traj (**TrajectoryWithRew**) – A trajectory generated from RolloutInfoWrapper-wrapped Environments.

**Return type**
**TrajectoryWithRew**

**Returns**
A copy of traj with replaced obs and rews fields.

**Raises**
**ValueError** – If traj.infos is None
imitation

imitation.data.types

Types and helper methods for transitions and trajectories.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dataclass_quick_asdict(obj)</code></td>
<td>Extract dataclass to items using <code>dataclasses.fields</code> + dict comprehension.</td>
</tr>
<tr>
<td><code>load(path)</code></td>
<td>Loads a sequence of trajectories saved by <code>save()</code> from <code>path</code>.</td>
</tr>
<tr>
<td><code>load_with_rewards(path)</code></td>
<td>Loads a sequence of trajectories with rewards from a file.</td>
</tr>
<tr>
<td><code>parse_optional_path(path[, allow_relative, ...])</code></td>
<td>Parse an optional path to a <code>pathlib.Path</code> object.</td>
</tr>
<tr>
<td><code>parse_path(path[, allow_relative, ...])</code></td>
<td>Parse a path to a <code>pathlib.Path</code> object.</td>
</tr>
<tr>
<td><code>save(path, trajectories)</code></td>
<td>Save a sequence of Trajectories to disk using a NumPy-based format.</td>
</tr>
<tr>
<td><code>transitions_collate_fn(batch)</code></td>
<td>Custom <code>torch.utils.data.DataLoader</code> <code>collate_fn</code> for <code>TransitionsMinimal</code>.</td>
</tr>
</tbody>
</table>

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Trajectory(obs, acts, infos, terminal)</code></td>
<td>A trajectory, e.g. a one episode rollout from an expert policy.</td>
</tr>
<tr>
<td><code>TrajectoryWithRew(obs, acts, infos, ...)</code></td>
<td>A <code>Trajectory</code> that additionally includes reward information.</td>
</tr>
<tr>
<td><code>Transitions(obs, acts, infos, next_obs, dones)</code></td>
<td>A batch of obs-act-obs-done transitions.</td>
</tr>
<tr>
<td><code>TransitionsMinimal(obs, acts, infos)</code></td>
<td>A Torch-compatible <code>Dataset</code> of obs-act transitions.</td>
</tr>
<tr>
<td><code>TransitionsWithRew(obs, acts, infos, ...)</code></td>
<td>A batch of obs-act-obs-reward-done transitions.</td>
</tr>
</tbody>
</table>

```python
class imitation.data.types.Trajectory(obs, acts, infos, terminal)
    Bases: object

    A trajectory, e.g. a one episode rollout from an expert policy.

    __init__(obs, acts, infos, terminal)

    acts: ndarray
        Actions, shape (trajectory_len, ) + action_shape.

    infos: Optional[ndarray]
        An array of info dicts, length trajectory_len.

    obs: ndarray
        Observations, shape (trajectory_len + 1, ) + observation_shape.

    terminal: bool
        Does this trajectory (fragment) end in a terminal state?

        Episodes are always terminal. Trajectory fragments are also terminal when they contain the final state of an episode (even if missing the start of the episode).
```
class imitation.data.types.TrajectoryWithRew(\textit{obs}, \textit{acts}, \textit{infos}, \textit{terminal}, \textit{rews})

Bases: \textit{Trajectory}

A \textit{Trajectory} that additionally includes reward information.

\_\_\_init\_\_(\textit{obs}, \textit{acts}, \textit{infos}, \textit{terminal}, \textit{rews})

\textbf{rews}: \textbf{ndarray}

Reward, shape (trajectory\_len, ). dtype float.

class imitation.data.types.Transitions(\textit{obs}, \textit{acts}, \textit{infos}, \textit{next\_obs}, \textit{dones})

Bases: \textit{TransitionsMinimal}

A batch of obs-act-obs-done transitions.

\_\_\_init\_\_(\textit{obs}, \textit{acts}, \textit{infos}, \textit{next\_obs}, \textit{dones})

\textbf{dones}: \textbf{ndarray}

(batch\_size, ).

\textit{done}[\textit{i}] \text{ is true iff } \textit{next\_obs}[\textit{i}] \text{ the last observation of an episode.}

\textbf{Type}

Boolean array indicating episode termination. Shape

\textbf{next\_obs}: \textbf{ndarray}

(batch\_size, ) + observation\_shape.

The \textit{i}'th observation \textit{next\_obs}[\textit{i}] in this array is the observation after the agent has taken action \textit{acts}[\textit{i}].

\textbf{Invariants}:

\begin{itemize}
    \item \textit{next\_obs}.dtype == \textit{obs}.dtype
    \item len(\textit{next\_obs}) == len(\textit{obs})
\end{itemize}

\textbf{Type}

New observation. Shape

class imitation.data.types.TransitionsMinimal(\textit{obs}, \textit{acts}, \textit{infos})

Bases: Dataset, Sequence[Mapping[str, ndarray]]

A Torch-compatible Dataset of obs-act transitions.

This class and its subclasses are usually instantiated via \textit{imitation.data.rollout.flatten\_trajectories}.

Indexing an instance \textit{trans} of TransitionsMinimal with an integer \textit{i} returns the \textit{i}'th \textit{Dict[str, np.ndarray]} sample, whose keys are the field names of each dataclass field and whose values are the \textit{i}th elements of each field value.

Slicing returns a possibly empty instance of TransitionsMinimal where each field has been sliced.

\_\_\_init\_\_(\textit{obs}, \textit{acts}, \textit{infos})

\textbf{acts}: \textbf{ndarray}

(batch\_size, ) + action\_shape.

\textbf{Type}

Actions. Shape
**imitation**

**infos**: `ndarray`

(batch_size,).

**Type**

Array of info dicts. Shape

**obs**: `ndarray`

(batch_size,) + observation_shape.

The i’th observation `obs[i]` in this array is the observation seen by the agent when choosing action `acts[i]`. `obs[i]` is not required to be from the timestep preceding `obs[i+1]`.

**Type**

Previous observations. Shape

**class** `imitation.data.types.TransitionsWithRew(obs, acts, infos, next_obs, dones, rews)`

**Bases**: `Transitions`

A batch of obs-act-obs-rew-done transitions.

**def __init__(self, obs, acts,infos,next_obs,dones,rews)**

**rews**: `ndarray`

(batch_size, ). dtype float.

The reward `rew[i]` at the i’th timestep is received after the agent has taken action `acts[i]`.

**Type**

Reward. Shape

**imitation.data.types.dataclass_quick_asdict(obj)**

Extract dataclass to items using `dataclasses.fields` + dict comprehension.

This is a quick alternative to `dataclasses.asdict`, which expensively and undocumentedly deep-copies every numpy array value. See https://stackoverflow.com/a/52229565/1091722.

**Parameters**

- **obj** – A dataclass instance.

**Return type**

Dict[str, Any]

**Returns**

A dictionary mapping from `obj` field names to values.

**imitation.data.types.load(path)**

Loads a sequence of trajectories saved by `save()` from `path`.

**Return type**

Sequence[Trajectory]

**imitation.data.types.load_with_rewards(path)**

Loads a sequence of trajectories with rewards from a file.

**Return type**

Sequence[TrajectoryWithRew]

**imitation.data.types.parse_optional_path(path, allow_relative=True, base_directory=None)**

Parse an optional path to a `pathlib.Path` object.

All resulting paths are resolved, absolute paths. If `allow_relative` is True, then relative paths are allowed as input, and are resolved relative to the current working directory, or relative to `base_directory` if it is specified.
Parameters

- **path** (Union[str, bytes, PathLike, None]) – The path to parse. Can be a string, bytes, or os.PathLike.
- **allow_relative** (bool) – If True, then relative paths are allowed as input, and are resolved relative to the current working directory. If False, an error is raised if the path is not absolute.
- **base_directory** (Optional[Path]) – If specified, then relative paths are resolved relative to this directory, instead of the current working directory.

Return type
Optional[Path]

Returns
A pathlib.Path object, or None if path is None.

**imitation.data.types.parse_path**(path, allow_relative=True, base_directory=None)

Parse a path to a pathlib.Path object.

All resulting paths are resolved, absolute paths. If allow_relative is True, then relative paths are allowed as input, and are resolved relative to the current working directory, or relative to base_directory if it is specified.

Parameters

- **path** (Union[str, bytes, PathLike]) – The path to parse. Can be a string, bytes, or os.PathLike.
- **allow_relative** (bool) – If True, then relative paths are allowed as input, and are resolved relative to the current working directory. If False, an error is raised if the path is not absolute.
- **base_directory** (Optional[Path]) – If specified, then relative paths are resolved relative to this directory, instead of the current working directory.

Return type
Path

Returns
A pathlib.Path object.

Raises

- **ValueError** – If allow_relative is False and the path is not absolute.
- **ValueError** – If base_directory is specified and allow_relative is False.

**imitation.data.types.save**(path, trajectories)

Save a sequence of Trajectories to disk using a NumPy-based format.

We create an .npyz dictionary with the following keys:

- **obs** flattened observations from all trajectories. Note that the leading dimension of this array will be len(trajectories) longer than the acts and infos arrays, because we always have one more observation than we have actions in any trajectory.
- **acts** flattened actions from all trajectories.
- **infos** flattened info dicts from all trajectories. Any trajectories with no info dict will have their entry in this array set to the empty dictionary.
- **terminal** boolean array indicating whether each trajectory is done.
- **indices** indices indicating where to split the flattened action and infos arrays, in order to recover the original trajectories. Will be a 1D array of length len(trajectories).

Parameters

- **path** (Union[str, bytes, PathLike]) – Trajectories are saved to this path.
- **trajectories** (Sequence[Trajectory]) – The trajectories to save.
Raises

*ValueError* – If not all trajectories have the same type, i.e. some are *Trajectory* and others are *TrajectoryWithRew*.

**imitation.data.types.transitions_collate_fn**(*batch*)

Custom `torch.utils.data.DataLoader` `collate_fn` for *TransitionsMinimal*.

Use this as the `collate_fn` argument to `DataLoader` if using an instance of *TransitionsMinimal* as the `dataset` argument.

**Parameters**

- **batch** (`Sequence[Mapping[str, ndarray]]`) – The batch to collate.

**Return type**

`Mapping[str, Union[ndarray, Tensor]]`

**Returns**

A collated batch. Uses Torch’s default collate function for everything except the “infos” key. For “infos”, we join all the info dicts into a list of dicts. (The default behavior would recursively collate every info dict into a single dict, which is incorrect.)

**imitation.data.wrappers**

Environment wrappers for collecting rollouts.

**Classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BufferingWrapper</code> (<code>venv[...].</code>)</td>
<td>Saves transitions of underlying VecEnv.</td>
</tr>
<tr>
<td><code>RolloutInfoWrapper</code> (<code>env</code>)</td>
<td>Add the entire episode’s rewards and observations to <code>info</code> at episode end.</td>
</tr>
</tbody>
</table>

**class** `imitation.data.wrappers.BufferingWrapper` (`venv, error_on_premature_reset=True`)  

**Bases:** `VecEnvWrapper`

Saves transitions of underlying VecEnv.

Retrieve saved transitions using `pop_transitions()`.

**__init__** (`venv, error_on_premature_reset=True`)  

Builds BufferingWrapper.

**Parameters**

- **venv** (`VecEnv`) – The wrapped VecEnv.
- **error_on_premature_reset** (`bool`) – Error if `reset()` is called on this wrapper and there are saved samples that haven’t yet been accessed.

**error_on_premature_event**: `bool`

**n_transitions**: `Optional[int]`

**pop_finished_trajectories**()  

Pops recorded complete trajectories `trajs` and episode lengths `ep_lens`.

**Return type**

`Tuple[Sequence[TrajectoryWithRew], Sequence[int]]`
Returns
A tuple \((\text{trajs, ep_lens})\) where \(\text{trajs}\) is a sequence of trajectories including the terminal state (but possibly missing initial states, if \(\text{pop\_trajectories}\) was previously called) and \(\text{ep\_lens}\) is a sequence of episode lengths. Note the episode length will be longer than the trajectory length when the trajectory misses initial states.

\[\text{pop\_trajectories}()\]

Pops recorded trajectories \(\text{trajs}\) and episode lengths \(\text{ep\_lens}\).

Return type
Tuple[Sequence[\text{TrajectoryWithRew}], Sequence[int]]

Returns
A tuple \((\text{trajs, ep\_lens})\). \(\text{trajs}\) is a sequence of trajectory fragments, consisting of data collected after the last call to \(\text{pop\_trajectories}\). They may miss initial states (if \(\text{pop\_trajectories}\) previously returned a fragment for that episode), and terminal states (if the episode has yet to complete). \(\text{ep\_lens}\) is the total length of completed episodes.

\[\text{pop\_transitions}()\]

Pops recorded transitions, returning them as an instance of Transitions.

Return type
\text{TransitionsWithRew}

Returns
All transitions recorded since the last call.

Raises
\text{RuntimeError} – empty (no transitions recorded since last pop).

\[\text{reset}(**\text{kwargs})\]

Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns
observation

\[\text{step\_async}(\text{actions})\]

Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

\[\text{step\_wait}()\]

Wait for the step taken with step_async().

Returns
observation, reward, done, information

\text{class imitation.data.wrappers.RolloutInfoWrapper(env)}

Bases: Wrapper

Add the entire episode’s rewards and observations to \(\text{info}\) at episode end.

Whenever \(\text{done} = \text{True}\), \(\text{info}[\text{“rollouts”}]\) is a dict with keys \(\text{“obs”}\) and \(\text{“rews”}\), whose corresponding values hold the NumPy arrays containing the raw observations and rewards seen during this episode.
__init__(env)
Builds RolloutInfoWrapper.

Parameters
env (Env) – Environment to wrap.

reset(**kwargs)
Resets the environment to an initial state and returns an initial observation.

Note that this function should not reset the environment’s random number generator(s); random variables in the environment’s state should be sampled independently between multiple calls to reset(). In other words, each call of reset() should yield an environment suitable for a new episode, independent of previous episodes.

Returns
the initial observation.

Return type
observation (object)

step(action)
Run one timestep of the environment’s dynamics. When end of episode is reached, you are responsible for calling reset() to reset this environment’s state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters
action (object) – an action provided by the agent

Returns
agent’s observation of the current environment reward (float) : amount of reward returned after previous action done (bool) : whether the episode has ended, in which case further step() calls will return undefined results info (dict) : contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type
observation (object)

2.1.3 imitation.policies

Classes defining policies and methods to manipulate them (e.g. serialization).

Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation.policies.base</td>
<td>Custom policy classes and convenience methods.</td>
</tr>
<tr>
<td>imitation.policies.exploration_wrapper</td>
<td>Wrapper to turn a policy into a more exploratory version.</td>
</tr>
<tr>
<td>imitation.policies.replay_buffer_wrapper</td>
<td>Wrapper for reward labeling for transitions sampled from a replay buffer.</td>
</tr>
<tr>
<td>imitation.policies.serialize</td>
<td>Load serialized policies of different types.</td>
</tr>
</tbody>
</table>
**imitation.policies.base**

Custom policy classes and convenience methods.

**Classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FeedForward32Policy</strong>(*args, <strong>kwargs)</strong></td>
<td>A feed forward policy network with two hidden layers of 32 units.</td>
</tr>
<tr>
<td><strong>HardCodedPolicy</strong>(<em>observation_space, action_space)</em>*</td>
<td>Abstract class for hard-coded (non-trainable) policies.</td>
</tr>
<tr>
<td><strong>NormalizeFeaturesExtractor</strong>(<em>observation_space)</em>*</td>
<td>Feature extractor that flattens then normalizes input.</td>
</tr>
<tr>
<td><strong>RandomPolicy</strong>(<em>observation_space, action_space)</em>*</td>
<td>Returns random actions.</td>
</tr>
<tr>
<td><strong>SAC1024Policy</strong>(*args, <strong>kwargs)</strong></td>
<td>Actor and value networks with two hidden layers of 1024 units respectively.</td>
</tr>
<tr>
<td><strong>ZeroPolicy</strong>(<em>observation_space, action_space)</em>*</td>
<td>Returns constant zero action.</td>
</tr>
</tbody>
</table>

**class** imitation.policies.base.**FeedForward32Policy**(*args, **kwargs*)

Bases: ActorCriticPolicy

A feed forward policy network with two hidden layers of 32 units.

This matches the IRL policies in the original AIRL paper.

Note: This differs from stable_baselines3 ActorCriticPolicy in two ways: by having 32 rather than 64 units, and by having policy and value networks share weights except at the final layer, where there are different linear heads.

**__init__**( *args, **kwargs*)

Builds FeedForward32Policy; arguments passed to ActorCriticPolicy.

**training**: bool

**class** imitation.policies.base.**HardCodedPolicy**(*observation_space, action_space*)

Bases: BasePolicy, ABC

Abstract class for hard-coded (non-trainable) policies.

**__init__**( *observation_space, action_space*)

Builds HardcodedPolicy with specified observation and action space.

**forward**( *args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

**training**: bool

**class** imitation.policies.base.**NormalizeFeaturesExtractor**(*observation_space, normalize_class=<class 'imitation.util.networks.RunningNorm'>)

**training**: bool
Bases: `FlattenExtractor`

Feature extractor that flattens then normalizes input.

```python
__init__(observation_space, normalize_class=<class 'imitation.util.networks.RunningNorm'>)
```

Builds NormalizeFeaturesExtractor.

**Parameters**

- **observation_space** (`Space`): The space observations lie in.
- **normalize_class** (`Type[Module]`): The class to use to normalize observations (after being flattened). This can be any Module that preserves the shape; e.g. `nn.BatchNorm*` or `nn.LayerNorm`.

```python
forward(observations)
```

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

**Return type**

Tensor

```python
class imitation.policies.base.RandomPolicy(observation_space, action_space)
```

Bases: `HardCodedPolicy`

Returns random actions.

```python
optimizer: Optional[th.optim.Optimizer]
```

```python
training: bool
```

```python
class imitation.policies.base.SAC1024Policy(*args, **kwargs)
```

Bases: `SACPolicy`

Actor and value networks with two hidden layers of 1024 units respectively.

This matches the implementation of SAC policies in the PEBBLE paper. See: https://arxiv.org/pdf/2106.05091.pdf https://github.com/denisyarats/pytorch_sac/blob/master/config/agent/sac.yaml

Note: This differs from stable_baselines3 SACPolicy by having 1024 hidden units in each layer instead of the default value of 256.

```python
__init__(*args, **kwargs)
```

Builds SAC1024Policy; arguments passed to `SACPolicy`.

```python
training: bool
```

```python
class imitation.policies.base.ZeroPolicy(observation_space, action_space)
```

Bases: `HardCodedPolicy`

Returns constant zero action.
optimizer: Optional[th.optim.Optimizer]
training: bool

**imitation.policies.exploration_wrapper**

Wrapper to turn a policy into a more exploratory version.

**Classes**

<table>
<thead>
<tr>
<th>ExplorationWrapper(policy, venv, ...[, ...])</th>
<th>Wraps a PolicyCallable to create a partially randomized version.</th>
</tr>
</thead>
</table>

```python
class imitation.policies.exploration_wrapper.ExplorationWrapper(policy, venv, random_prob, switch_prob, rng, deterministic_policy=False)
```

**Bases:** object

Wraps a PolicyCallable to create a partially randomized version.

This wrapper randomly switches between two policies: the wrapped policy, and a random one. After each action, the current policy is kept with a certain probability. Otherwise, one of these two policies is chosen at random (without any dependence on what the current policy is).

The random policy uses the `action_space.sample()` method.

```python
__init__(policy, venv, random_prob, switch_prob, rng, deterministic_policy=False)
```

Initializes the ExplorationWrapper.

**Parameters**

- **policy** (Union[BaseAlgorithm, BasePolicy, Callable[[ndarray], ndarray], None]) – The policy to randomize.
- **venv** (VecEnv) – The environment to use (needed for sampling random actions).
- **random_prob** (float) – The probability of picking the random policy when switching.
- **switch_prob** (float) – The probability of switching away from the current policy.
- **rng** (Generator) – The random state to use for seeding the environment and for switching policies.
- **deterministic_policy** (bool) – Whether to make the policy deterministic when not exploring. This must be False when policy is a PolicyCallable.

**imitation.policies.replay_buffer_wrapper**

Wrapper for reward labeling for transitions sampled from a replay buffer.
**ReplayBufferRewardWrapper** *(buffer_size, ...)*  
Relabel the rewards in transitions sampled from a ReplayBuffer.

```python
class imitation.policies.replay_buffer_wrapper.ReplayBufferRewardWrapper(buffer_size, observation_space, action_space, replay_buffer_class, reward_fn, **kwargs)
```

Bases: ReplayBuffer

Relabel the rewards in transitions sampled from a ReplayBuffer.

```python
__init__(buffer_size, observation_space, action_space, replay_buffer_class, reward_fn, **kwargs)
```

Builds ReplayBufferRewardWrapper.

**Parameters**

- **buffer_size** (int) – Max number of elements in the buffer
- **observation_space** (Space) – Observation space
- **action_space** (Space) – Action space
- **replay_buffer_class** (Type[ReplayBuffer]) – Class of the replay buffer.
- **reward_fn** (RewardFn) – Reward function for reward relabeling.
- **kwargs** – keyword arguments for ReplayBuffer.

```python
add(*args, **kwargs)
```

Add elements to the buffer.

```
property full: bool
```

Return type

bool

```
property pos: int
```

Return type

int

```python
sample(*args, **kwargs)
```

Sample elements from the replay buffer. Custom sampling when using memory efficient variant, as we should not sample the element with index `self.pos` See https://github.com/DLR-RM/stable-baselines3/pull/28#issuecomment-637559274

**Parameters**

- **batch_size** – Number of element to sample
- **env** – associated gym VecEnv to normalize the observations/rewards when sampling

**Returns**
imitation.policies.serialize

Load serialized policies of different types.

Module Attributes

| PolicyLoaderFn | A policy loader function that takes a VecEnv before any other custom arguments and returns a stable_baselines3 base policy policy. |
| policy_registry | Registry of policy loading functions. |

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

| load_policy(policy_type, venv, **kwargs) | Load serialized policy. |
| load_stable_baselines_model(cls, path, venv, ...) | Helper method to load RL models from Stable Baselines. |
| save_stable_model(output_dir, model[, filename]) | Serialize Stable Baselines model. |

Classes

```python
SavePolicyCallback(policy_dir, *args, **kwargs)
```

Bases: EventCallback

Saves the policy using save_stable_model each time it is called.

Should be used in conjunction with callbacks.EveryNTimesteps or another event-based trigger.

```python
__init__(policy_dir, *args, **kwargs)
```

Builds SavePolicyCallback.

Parameters

- **policy_dir** (Path) – Directory to save checkpoints.
- **args** – Passed through to callbacks.EventCallback.
- **kwargs** – Passed through to callbacks.EventCallback.

2.1. imitation
imitation.policies.serialize.load_policy(policy_type, venv, **kwargs)

Load serialized policy.

Note on the kwargs:
- zero and random policy take no kwargs
- ppo and sac policies take a path argument with a path to a zip file or to a folder containing a model.zip file.
- ppo-huggingface and sac-huggingface policies take an env_name and optional organization argument.

Parameters

- **policy_type** (str) – A key in policy_registry, e.g. ppo.
- **venv** (VecEnv) – An environment that the policy is to be used with.
- **kwargs** – Additional arguments to pass to the policy loader.

Return type

BasePolicy

Returns

The deserialized policy.

imitation.policies.serialize.load_stable_baselines_model(cls, path, venv, **kwargs)

Helper method to load RL models from Stable Baselines.

Parameters

- **cls** (Type[TypeVar[Algorithm], bound= BaseAlgorithm]]) – Stable Baselines RL algorithm.
- **path** (str) – Path to zip file containing saved model data or to a folder containing a model.zip file.
- **venv** (VecEnv) – Environment to train on.
- **kwargs** – Passed through to cls.load.

Raises

- FileNotFoundError – If path is not a directory containing a model.zip file.
- FileExistsError – If path contains a vec_normalize.pkl file (unsupported).

Return type

TypeVar[Algorithm, bound=BaseAlgorithm]

Returns

The deserialized RL algorithm.

imitation.policies.serialize.policy_registry: Registry[Callable[[], BasePolicy]] =  
<imitation.util.registry.Registry object>

Registry of policy loading functions. Add your own here if desired.

imitation.policies.serialize.save_stable_model(output_dir, model, filename='model.zip')

Serialize Stable Baselines model.

Load later with load_policy(...., policy_path=output_dir).

Parameters

- **output_dir** (Path) – Path to the save directory.
• **model** (BaseAlgorithm) – The stable baselines model.

• **filename** (str) – The filename of the model.

Return type
None

## 2.1.4 imitation.regularization

Implements a variety of regularization techniques for NN weights.

### Modules

- **imitation.regularization.regularizers** Implements the regularizer base class and some standard regularizers.

- **imitation.regularization.updaters** Implements parameter scaling algorithms to update the parameters of a regularizer.

### imitation.regularization.regularizers

Implements the regularizer base class and some standard regularizers.

### Classes

- **LossRegularizer**

  Abstract base class for regularizers that add a loss term to the loss function.

- **LpRegularizer**

  Applies Lp regularization to a loss function.

- **Regularizer**

  Abstract class for creating regularizers with a common interface.

- **RegularizerFactory**

  Protocol for functions that create regularizers.

- **WeightDecayRegularizer**

  Applies weight decay to a loss function.

- **WeightRegularizer**

  Abstract base class for regularizers that regularize the weights of a network.

```python
class imitation.regularization.regularizers.LossRegularizer(optimizer, initial_lambda, ...):
    Bases: Regularizer[Union[Tensor, float]]
    Abstract base class for regularizers that add a loss term to the loss function.
    Requires the user to implement the _loss_penalty method.
    lambda_: float
    lambda_updater: Optional[LambdaUpdater]
    logger: HierarchicalLogger
    optimizer: Optimizer
```

2.1. imitation
**regularize_and_backward**(loss)

Add the regularization term to the loss and compute gradients.

**Parameters**

- **loss** (Tensor) – The loss to regularize.

**Return type**

Union[Tensor, float]

**Returns**

The regularized loss.

**val_split**: Optional[float]

### class imitation.regularization.regularizers.LpRegularizer(optimizer, initial_lambda, lambda_updater, logger, p, val_split=None)

**Bases**: LossRegularizer

Applies Lp regularization to a loss function.

**__init__**(optimizer, initial_lambda, lambda_updater, logger, p, val_split=None)

Initialize the regularizer.

**p**: int

### class imitation.regularization.regularizers.Regularizer(optimizer, initial_lambda, lambda_updater, logger, val_split=None)

**Bases**: ABC, Generic[R]

Abstract class for creating regularizers with a common interface.

**__init__**(optimizer, initial_lambda, lambda_updater, logger, val_split=None)

Initialize the regularizer.

**Parameters**

- **optimizer** (Optimizer) – The optimizer to which the regularizer is attached.
- **initial_lambda** (float) – The initial value of the regularization parameter.
- **lambda_updater** (Optional[ LambdaUpdater ]) – A callable object that takes in the current lambda and the train and val loss, and returns the new lambda.
- **logger** (HierarchicalLogger) – The logger to which the regularizer will log its parameters.
- **val_split** (Optional[float]) – The fraction of the training data to use as validation data for the lambda updater. Can be none if no lambda updater is provided.

**Raises**

- **ValueError** – if no lambda updater (lambda_updater) is provided and the initial regularization strength (initial_lambda) is zero.
- **ValueError** – if a validation split (val_split) is provided but it's not a float in the (0, 1) interval.
- **ValueError** – if a lambda updater is provided but no validation split is provided.
- **ValueError** – if a validation split is set, but no lambda updater is provided.
classmethod create(initial_lambda, lambda_updater=None, val_split=0.0, **kwargs)

Create a regularizer.

    Return type
    RegularizerFactory[TypeVar(Self, bound= Regularizer)]

lambda_: float

lambda_updater: Optional[LambdaUpdater]

logger: HierarchicalLogger

optimizer: Optimizer

abstract regularize_and_backward(loss)

Abstract method for performing the regularization step.

The return type is a generic and the specific implementation must describe the meaning of the return type.

This step will also call loss.backward() for the user. This is because the regularizer may require the loss to be called before or after the regularization step. Leaving this to the user would force them to make their implementation dependent on the regularizer algorithm used, which is prone to errors.

    Parameters
    loss (Tensor) – The loss to regularize.

    Return type
    TypeVar(R)

update_params(train_loss, val_loss)

Update the regularization parameter.

This method calls the lambda_updater to update the regularization parameter, and assigns the new value to self.lambda_. Then logs the new value using the provided logger.

    Parameters

    • train_loss (Union[Tensor, float]) – The loss on the training set.

    • val_loss (Union[Tensor, float]) – The loss on the validation set.

    Return type
    None

val_split: Optional[float]

class imitation.regularization.regularizers.RegularizerFactory(*args, **kwargs)

Bases: Protocol[T_Regularizer_co]

Protocol for functions that create regularizers.

The regularizer factory is meant to be used as a way to create a regularizer in two steps. First, the end-user creates a regularizer factory by calling the .create() method of a regularizer class. This allows specifying all the relevant configuration to the regularization algorithm. Then, the network algorithm finishes setting up the optimizer and logger, and calls the regularizer factory to create the regularizer.

This two-step process separates the configuration of the regularization algorithm from additional “operational” parameters. This is useful because it solves two problems:

1. The end-user does not have access to the optimizer and logger when configuring the regularization algorithm.

2. Validation of the configuration is done outside the network constructor.
It also allows re-using the same regularizer factory for multiple networks.

```python
__init__(*args, **kwargs)
```

class imitation.regularization.regularizers.WeightDecayRegularizer(optimizer, initial_lambda, lambda_updater, logger, val_split=None)

Bases: WeightRegularizer

Applies weight decay to a loss function.

lambda_: float

lambda_updater: Optional[LambdaUpdater]

logger: HierarchicalLogger

optimizer: Optimizer

val_split: Optional[float]

class imitation.regularization.regularizers.WeightRegularizer(optimizer, initial_lambda, lambda_updater, logger, val_split=None)

Bases: Regularizer

Abstract base class for regularizers that regularize the weights of a network.

Requires the user to implement the _weight_penalty method.

lambda_: float

lambda_updater: Optional[LambdaUpdater]

logger: HierarchicalLogger

optimizer: Optimizer

regularize_and_backward(loss)

Regularize the weights of the network, and call loss.backward().

Return type

None

val_split: Optional[float]

imitation.regularization.updaters

Implements parameter scaling algorithms to update the parameters of a regularizer.
Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IntervalParamScaler</strong>(scaling_factor, ...)</td>
<td>Scales the lambda of the regularizer by some constant factor.</td>
</tr>
<tr>
<td><strong>LambdaUpdater</strong>(args, **kwargs)</td>
<td>Protocol type for functions that update the regularizer parameter.</td>
</tr>
</tbody>
</table>

```python
class imitation.regularization.updaters.IntervalParamScaler(scaling_factor, tolerable_interval):
    Bases: LambdaUpdater
    Scales the lambda of the regularizer by some constant factor.
    Lambda is scaled up if the ratio of the validation loss to the training loss is above the tolerable interval, and scaled down if the ratio is below the tolerable interval. Nothing happens if the ratio is within the tolerable interval.
    __init__(scaling_factor, tolerable_interval)
        Initialize the interval parameter scaler.
    Parameters
        • scaling_factor (float) – The factor by which to scale the lambda, a value in (0, 1).
        • tolerable_interval (Tuple[float, float]) – The interval within which the ratio of the validation loss to the training loss is considered acceptable. A tuple whose first element is at least 0 and the second element is greater than the first.
    Raises
        • ValueError – If the tolerable interval is not a tuple of length 2.
        • ValueError – if the scaling factor is not in (0, 1).
        • ValueError – if the tolerable interval is negative or not a proper interval.

class imitation.regularization.updaters.LambdaUpdater(*args, **kwargs):
    Bases: Protocol
    Protocol type for functions that update the regularizer parameter.
    A callable object that takes in the current lambda and the train and val loss, and returns the new lambda. This has been implemented as a protocol and not an ABC because a user might wish to provide their own implementation without having to inherit from the base class, e.g. by defining a function instead of a class.
    Note: if you implement LambdaUpdater, your implementation MUST be purely functional, i.e. side-effect free. The class structure should only be used to store constant hyperparameters. (Alternatively, closures can be used for that).
    __init__(*args, **kwargs)
```

2.1.5 imitation.rewards

Reward models: neural network modules, serialization, preprocessing, etc.
**imitation**

## Modules

<table>
<thead>
<tr>
<th>Modules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation.rewards.reward_function</td>
<td>Type alias shared by reward-related code.</td>
</tr>
<tr>
<td>imitation.rewards.reward_nets</td>
<td>Constructs deep network reward models.</td>
</tr>
<tr>
<td>imitation.rewards.reward_wrapper</td>
<td>Common wrapper for adding custom reward values to an environment.</td>
</tr>
<tr>
<td>imitation.rewards.serialize</td>
<td>Load serialized reward functions of different types.</td>
</tr>
</tbody>
</table>

### imitation.rewards.reward_function

Type alias shared by reward-related code.

### Classes

```python
RewardFn(*args, **kwargs) Abstract class for reward function.
```

```python
class imitation.rewards.reward_function.RewardFn(*args, **kwargs):
    Bases: Protocol
    Abstract class for reward function.
    Requires implementation of __call__() to compute the reward given a batch of states, actions, next states and dones.
    __init__(*args, **kwargs)
```

### imitation.rewards.reward_nets

Constructs deep network reward models.

### Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```python
cnnTranspose(tens) Transpose a (b,h,w,c)-formatted tensor to (b,c,h,w) format.
```
### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AddSTDRewardWrapper</strong>(\text{base}[\text{default_alpha}])</td>
<td>Adds a multiple of the estimated standard deviation to mean reward.</td>
</tr>
<tr>
<td><strong>BasicPotentialCNN</strong>(\text{observation_space, hid_sizes})</td>
<td>Simple implementation of a potential using a CNN.</td>
</tr>
<tr>
<td><strong>BasicPotentialMLP</strong>(\text{observation_space, ...})</td>
<td>Simple implementation of a potential using an MLP.</td>
</tr>
<tr>
<td><strong>BasicRewardNet</strong>(\text{observation_space, action_space})</td>
<td>MLP that takes as input the state, action, next state and done flag.</td>
</tr>
<tr>
<td><strong>BasicShapedRewardNet</strong>(\text{observation_space, ...})</td>
<td>Shaped reward net based on MLPs.</td>
</tr>
<tr>
<td><strong>CnnRewardNet</strong>(\text{observation_space, ...})</td>
<td>CNN that takes as input the state, action, next state and done flag.</td>
</tr>
<tr>
<td><strong>ForwardWrapper</strong>(\text{base})</td>
<td>An abstract RewardNetWrapper that changes the behavior of forward.</td>
</tr>
<tr>
<td><strong>NormalizedRewardNet</strong>(\text{base, normalize_output_layer})</td>
<td>A reward net that normalizes the output of its base network.</td>
</tr>
<tr>
<td><strong>PredictProcessedWrapper</strong>(\text{base})</td>
<td>An abstract RewardNetWrapper that changes the behavior of predict_processed.</td>
</tr>
<tr>
<td><strong>RewardEnsemble</strong>(\text{observation_space, ...})</td>
<td>A mean ensemble of reward networks.</td>
</tr>
<tr>
<td><strong>RewardNet</strong>(\text{observation_space, action_space[\text{...}]}</td>
<td>Minimal abstract reward network.</td>
</tr>
<tr>
<td><strong>RewardNetWithVariance</strong>(\text{observation_space, ...})</td>
<td>A reward net that keeps track of its epistemic uncertainty through variance.</td>
</tr>
<tr>
<td><strong>RewardNetWrapper</strong>(\text{base})</td>
<td>Abstract class representing a wrapper modifying a RewardNet's functionality.</td>
</tr>
<tr>
<td><strong>ShapedRewardNet</strong>(\text{base, potential, discount_factor})</td>
<td>A RewardNet consisting of a base network and a potential shaping.</td>
</tr>
</tbody>
</table>

```python
class imitation.rewards.reward_nets.AddSTDRewardWrapper\(\text{base, default\_alpha=0.0}\)
    Bases: PredictProcessedWrapper

    Adds a multiple of the estimated standard deviation to mean reward.

    __init__\(\text{base, default\_alpha=0.0}\)
        Create a reward network that adds a multiple of the standard deviation.

    Parameters
        - **base** \((\text{RewardNetWithVariance})\) – A reward network that keeps track of its epistemic variance. This is used to compute the standard deviation.
        - **default\_alpha** \((\text{float})\) – multiple of standard deviation to add to the reward mean. Defaults to 0.0.

    Raises
        - TypeError – if base is not an instance of RewardNetWithVariance

    predict_processed\(\text{state, action, next\_state, done, alpha=None, **kwargs}\)
        Compute a lower/upper confidence bound on the reward without gradients.

    Parameters
        - **state** \((\text{ndarray})\) – Current states of shape \((batch\_size,)+state\_shape\).
        - **action** \((\text{ndarray})\) – Actions of shape \((batch\_size,)+action\_shape\).
        - **next\_state** \((\text{ndarray})\) – Successor states of shape \((batch\_size,)+state\_shape\).
        - **done** \((\text{ndarray})\) – End-of-episode (terminal state) indicator of shape \((batch\_size,)\).
```
• **alpha** (Optional[float]) – multiple of standard deviation to add to the reward mean. Defaults to the value provided at initialization.

• **kwargs** – are not used

Return type

ndarray

Returns

Estimated lower confidence bounds on rewards of shape (batch_size,).

class imitation.rewards.reward_nets.BasicPotentialCNN(\(observation\_space, \text{hid}\_sizes, hwc\_format=	ext{True}, **\text{kwargs}\))

Bases: Module

Simple implementation of a potential using a CNN.

__init__(\(observation\_space, \text{hid}\_sizes, hwc\_format=	ext{True}, **\text{kwargs}\))

Initialize the potential.

Parameters

• **observation_space** (Space) – observation space of the environment.

• **hid_sizes** (Iterable[int]) – number of channels in hidden layers of the CNN.

• **hwc_format** (bool) – format of the observation. True if channel dimension is last, False if channel dimension is first.

• **kwargs** – passed straight through to build_cnn.

Raises

ValueError – if observations are not images.

forward(\(state\))

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

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

Return type

Tensor

training: bool

class imitation.rewards.reward_nets.BasicPotentialMLP(\(observation\_space, \text{hid}\_sizes, **\text{kwargs}\))

Bases: Module

Simple implementation of a potential using an MLP.

__init__(\(observation\_space, \text{hid}\_sizes, **\text{kwargs}\))

Initialize the potential.

Parameters

• **observation_space** (Space) – observation space of the environment.

• **hid_sizes** (Iterable[int]) – widths of the hidden layers of the MLP.
• **kwargs –** passed straight through to `build_mlp`.

**forward**(*state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

**Return type**

`Tensor`

**class** `imitation.rewards.reward_nets.BasicRewardNet`

*observation_space*, *action_space*, *use_state=True*, *use_action=True*, *use_next_state=False*, *use_done=False*, **kwargs

Bases: `RewardNet`

MLP that takes as input the state, action, next state and done flag.

These inputs are flattened and then concatenated to one another. Each input can enabled or disabled by the `use_*` constructor keyword arguments.

**__init__**(*observation_space*, *action_space*, *use_state=True*, *use_action=True*, *use_next_state=False*, *use_done=False*, **kwargs)

Builds reward MLP.

**Parameters**

• **observation_space** (Space) – The observation space.

• **action_space** (Space) – The action space.

• **use_state** (bool) – should the current state be included as an input to the MLP?

• **use_action** (bool) – should the current action be included as an input to the MLP?

• **use_next_state** (bool) – should the next state be included as an input to the MLP?

• **use_done** (bool) – should the “done” flag be included as an input to the MLP?

• **kwargs** – passed straight through to `build_mlp`.

**forward**(*state, action, next_state, done*)

Compute rewards for a batch of transitions and keep gradients.

**class** `imitation.rewards.reward_nets.BasicShapedRewardNet`

*observation_space*, *action_space*, *, reward_hid_sizes=(32,), potential_hid_sizes=(32, 32), use_state=True, use_action=True, use_next_state=False, use_done=False*, **kwargs

Bases: `ShapedRewardNet`

Shaped reward net based on MLPs.
This is just a very simple convenience class for instantiating a BasicRewardNet and a BasicPotentialMLP and wrapping them inside a ShapedRewardNet. Mainly exists for backwards compatibility after https://github.com/HumanCompatibleAI/imitation/pull/311 to keep the scripts working.

**TODO(ejnnr):** if we ever modify AIRL so that it takes in a RewardNet instance directly (instead of a class and kwargs) and instead instantiate the RewardNet inside the scripts, then it probably makes sense to get rid of this class.

```python
__init__(observation_space, action_space, *, reward_hid_sizes=(32,), potential_hid_sizes=(32, 32), use_state=True, use_action=True, use_next_state=False, use_done=False, discount_factor=0.99, **kwargs)
```

Builds a simple shaped reward network.

**Parameters**

- `observation_space` (Space) – The observation space.
- `action_space` (Space) – The action space.
- `reward_hid_sizes` (Sequence[int]) – sequence of widths for the hidden layers of the base reward MLP.
- `potential_hid_sizes` (Sequence[int]) – sequence of widths for the hidden layers of the potential MLP.
- `use_state` (bool) – should the current state be included as an input to the reward MLP?
- `use_action` (bool) – should the current action be included as an input to the reward MLP?
- `use_next_state` (bool) – should the next state be included as an input to the reward MLP?
- `use_done` (bool) – should the “done” flag be included as an input to the reward MLP?
- `discount_factor` (float) – discount factor for the potential shaping.
- `**kwargs` – passed straight through to `BasicRewardNet` and `BasicPotentialMLP`.

```
class imitation.rewards.reward_nets.CnnRewardNet
```

Bases: `RewardNet`

CNN that takes as input the state, action, next state and done flag.

Inputs are boosted to tensors with channel, height, and width dimensions, and then concatenated. Image inputs are assumed to be in (h,w,c) format, unless the argument hwc_format=False is passed in. Each input can be enabled or disabled by the `use_*` constructor keyword arguments, but either `use_state` or `use_next_state` must be True.

```python
__init__(observation_space, action_space, use_state=True, use_action=True, use_next_state=False, use_done=False, hwc_format=True, **kwargs)
```

Builds reward CNN.

**Parameters**

- `observation_space` (Space) – The observation space.
- `action_space` (Space) – The action space.
- `use_state` (bool) – Should the current state be included as an input to the CNN?
• **use_action** (bool) – Should the current action be included as an input to the CNN?
• **use_next_state** (bool) – Should the next state be included as an input to the CNN?
• **use_done** (bool) – Should the “done” flag be included as an input to the CNN?
• **hwc_format** (bool) – Are image inputs in (h,w,c) format (True), or (c,h,w) (False)? If hwc_format is False, image inputs are not transposed.
• **kwargs** – Passed straight through to `build_cnn`.

**Raises**

**ValueError** – if observation or action space is not easily massaged into a CNN input.

**forward**(state, action, next_state, done)
Computes rewardNet value on input state, action, next_state, and done flag.

Takes inputs that will be used, transposes image states to (c,h,w) format if needed, reshares inputs to have compatible dimensions, concatenates them, and inputs them into the CNN.

**Parameters**

• **state** (Tensor) – current state.
• **action** (Tensor) – current action.
• **next_state** (Tensor) – next state.
• **done** (Tensor) – flag for whether the episode is over.

**Returns**

reward of the transition.

**Return type**

th.Tensor

**get_num_channels_obs**(space)
Gets number of channels for the observation.

**Return type**

int

**training**: bool

---

**class** imitation.rewards.reward_nets.ForwardWrapper(base)

**Bases**: RewardNetWrapper

An abstract RewardNetWrapper that changes the behavior of forward.

Note that all forward wrappers must be placed before all predict processed wrappers.

**__init__**(base)
Create a forward wrapper.

**Parameters**

• **base** (RewardNet) – The base reward network

**Raises**

**ValueError** – if the base network is a **PredictProcessedWrapper**.

**training**: bool
class imitation.rewards.reward_nets.NormalizedRewardNet(base, normalize_output_layer)

    Bases: PredictProcessedWrapper
    
    A reward net that normalizes the output of its base network.

    __init__(base, normalize_output_layer)
    
    Initialize the NormalizedRewardNet.

    Parameters
    
    • base (RewardNet) – a base RewardNet
    • normalize_output_layer (Type[BaseNorm]) – The class to use to normalize rewards.
        This can be any nn.Module that preserves the shape; e.g. nn.Identity, nn.LayerNorm, or
        networks.RunningNorm.

    predict_processed(state, action, next_state, done, update_stats=True, **kwargs)
    
    Compute normalized rewards for a batch of transitions without gradients.

    Parameters
    
    • state (ndarray) – Current states of shape (batch_size,) + state_shape.
    • action (ndarray) – Actions of shape (batch_size,) + action_shape.
    • next_state (ndarray) – Successor states of shape (batch_size,) + state_shape.
    • done (ndarray) – End-of-episode (terminal state) indicator of shape (batch_size,).
    • update_stats (bool) – Whether to update the running stats of the normalization layer.
    • **kwargs – kwargs passed to base predict_processed call.

    Return type
    
    ndarray

    Returns
    
    Computed normalized rewards of shape (batch_size,).

    training: bool

class imitation.rewards.reward_nets.PredictProcessedWrapper(base)

    Bases: RewardNetWrapper
    
    An abstract RewardNetWrapper that changes the behavior of predict_processed.

    Subclasses should override predict_processed. Implementations should pass along kwargs to the base reward
    net’s predict_processed method.

    Note: The wrapper will default to forwarding calls to device, forward,
        preprocess and predict to the base reward net unless explicitly overridden in a subclass.

    forward(state, action, next_state, done)
    
    Compute rewards for a batch of transitions and keep gradients.

    Return type
    
    Tensor

    predict(state, action, next_state, done)
    
    Compute rewards for a batch of transitions without gradients.

    Converting th.Tensor rewards from predict_th to NumPy arrays.

    Parameters
• **state** (ndarray) – Current states of shape \((batch_size,) + state\_shape\).
• **action** (ndarray) – Actions of shape \((batch_size,) + action\_shape\).
• **next_state** (ndarray) – Successor states of shape \((batch_size,) + state\_shape\).
• **done** (ndarray) – End-of-episode (terminal state) indicator of shape \((batch_size,)\).

**Return type**
ndarray

**Returns**
Computed rewards of shape \((batch_size,)\).

**abstract predict_processed**(state, action, next_state, done, **kwargs)
Predict processed must be overridden in subclasses.

**Return type**
ndarray

**predict_th**(state, action, next_state, done)
Compute th.Tensor rewards for a batch of transitions without gradients.
Preprocesses the inputs, output th.Tensor reward arrays.

**Parameters**
• **state** (ndarray) – Current states of shape \((batch_size,) + state\_shape\).
• **action** (ndarray) – Actions of shape \((batch_size,) + action\_shape\).
• **next_state** (ndarray) – Successor states of shape \((batch_size,) + state\_shape\).
• **done** (ndarray) – End-of-episode (terminal state) indicator of shape \((batch_size,)\).

**Return type**
Tensor

**Returns**
Computed th.Tensor rewards of shape \((batch_size,)\).

**training**: bool

**class** imitation.rewards.reward_nets.RewardEnsemble(\(\text{observation\_space, action\_space, members}\))
**Bases**: RewardNetWithVariance

A mean ensemble of reward networks.

A reward ensemble is made up of individual reward networks. To maintain consistency the “output” of a reward network will be defined as the results of its `predict_processed`. Thus for example the mean of the ensemble is the mean of the results of its members predict processed classes.

**__init__**(\(\text{observation\_space, action\_space, members}\))
Initialize the RewardEnsemble.

**Parameters**
• **observation_space** (Space) – the observation space of the environment
• **action_space** (Space) – the action space of the environment
• **members** (Iterable[RewardNet]) – the member networks that will make up the ensemble.

**Raises**
ValueError – if num_members is less than 1
forward(*args)

    The forward method of the ensemble should in general not be used directly.

    Return type
    Tensor

members: ModuleList

property num_members

    The number of members in the ensemble.

predict(state, action, next_state, done, **kwargs)

    Return the mean of the ensemble members.

predict_processed(state, action, next_state, done, **kwargs)

    Return the mean of the ensemble members.

    Return type
    ndarray

predict_processed_all(state, action, next_state, done, **kwargs)

    Get the results of predict processed on all of the members.

    Parameters
    • state (ndarray) – Current states of shape \((batch\_size,) + state\_shape\).
    • action (ndarray) – Actions of shape \((batch\_size,) + action\_shape\).
    • next_state (ndarray) – Successor states of shape \((batch\_size,) + state\_shape\).
    • done (ndarray) – End-of-episode (terminal state) indicator of shape \((batch\_size,)\).
    • **kwargs – passed along to ensemble members.

    Return type
    ndarray

    Returns
    The result of predict processed for each member in the ensemble of
    shape \((batch\_size, num\_members)\).

predict_reward_moments(state, action, next_state, done, **kwargs)

    Compute the standard deviation of the reward distribution for a batch.

    Parameters
    • state (ndarray) – Current states of shape \((batch\_size,) + state\_shape\).
    • action (ndarray) – Actions of shape \((batch\_size,) + action\_shape\).
    • next_state (ndarray) – Successor states of shape \((batch\_size,) + state\_shape\).
    • done (ndarray) – End-of-episode (terminal state) indicator of shape \((batch\_size,)\).
    • **kwargs – passed along to predict processed.

    Return type
    Tuple[ndarray, ndarray]

    Returns
    • Reward mean of shape \((batch\_size,)\).
    • Reward variance of shape \((batch\_size,)\).
class imitation.rewards.reward_nets.RewardNet(\texttt{observation\_space, action\_space, normalize\_images=\texttt{True}})

Bases: Module, ABC

Minimal abstract reward network.

Only requires the implementation of a forward pass (calculating rewards given a batch of states, actions, next states and dones).

\_\_\_init\_\_(\texttt{observation\_space, action\_space, normalize\_images=\texttt{True}})

Initialize the RewardNet.

Parameters

- \texttt{observation\_space} (Space) – the observation space of the environment
- \texttt{action\_space} (Space) – the action space of the environment
- \texttt{normalize\_images} (bool) – whether to automatically normalize image observations to [0, 1] (from 0 to 255). Defaults to \texttt{True}.

property \texttt{device}: device

Heuristic to determine which device this module is on.

Return type

device

property \texttt{dtype}: dtype

Heuristic to determine dtype of module.

Return type

dtype

abstract \texttt{forward}(\texttt{state, action, next\_state, done})

Compute rewards for a batch of transitions and keep gradients.

Return type

Tensor

\texttt{predict}(\texttt{state, action, next\_state, done})

Compute rewards for a batch of transitions without gradients.

Converting \texttt{th.Tensor} rewards from \texttt{predict\_th} to NumPy arrays.

Parameters

- \texttt{state} (ndarray) – Current states of shape (\texttt{batch\_size,}) + \texttt{state\_shape}.
- \texttt{action} (ndarray) – Actions of shape (\texttt{batch\_size,}) + \texttt{action\_shape}.
- \texttt{next\_state} (ndarray) – Successor states of shape (\texttt{batch\_size,}) + \texttt{state\_shape}.
- \texttt{done} (ndarray) – End-of-episode (terminal state) indicator of shape (\texttt{batch\_size,}).

Return type

ndarray

Returns

Computed rewards of shape (\texttt{batch\_size,}).
**predict_processed** *(state, action, next_state, done, **kwargs)*

Compute the processed rewards for a batch of transitions without gradients.

Defaults to calling `predict`. Subclasses can override this to normalize or otherwise modify the rewards in ways that may help RL training or other applications of the reward function.

**Parameters**

- **state** (ndarray) – Current states of shape `(batch_size,) + state_shape`.
- **action** (ndarray) – Actions of shape `(batch_size,) + action_shape`.
- **next_state** (ndarray) – Successor states of shape `(batch_size,) + state_shape`.
- **done** (ndarray) – End-of-episode (terminal state) indicator of shape `(batch_size,)`.
- **kwargs** – additional kwargs may be passed to change the functionality of subclasses.

**Return type**

ndarray

**Returns**

Computed processed rewards of shape `(batch_size,)`.

**predict_th** *(state, action, next_state, done)*

Compute th.Tensor rewards for a batch of transitions without gradients.

Preprocesses the inputs, output th.Tensor reward arrays.

**Parameters**

- **state** (ndarray) – Current states of shape `(batch_size,) + observation_space.shape`.
- **action** (ndarray) – Actions of shape `(batch_size,) + action_space.shape`.
- **next_state** (ndarray) – Successor states of shape `(batch_size,) + observation_space.shape`.
- **done** (ndarray) – End-of-episode (terminal state) indicator of shape `(batch_size,)`.

**Return type**

Tensor

**Returns**

Computed th.Tensor rewards of shape `(batch_size,)`.

**preprocess** *(state, action, next_state, done)*

Preprocess a batch of input transitions and convert it to PyTorch tensors.

The output of this function is suitable for its forward pass, so a typical usage would be `model(*model.preprocess(transitions))`.

**Parameters**

- **state** (ndarray) – The observation input. Its shape is `(batch_size,) + observation_space.shape`.
- **action** (ndarray) – The action input. Its shape is `(batch_size,) + action_space.shape`. The None dimension is expected to be the same as None dimension from `obs_input`.
- **next_state** (ndarray) – The observation input. Its shape is `(batch_size,) + observation_space.shape`.
- **done** (ndarray) – Whether the episode has terminated. Its shape is `(batch_size,)`.

**Returns**

a Tuple of tensors containing observations, actions, next observations and dones.
Return type
Preprocessed transitions

training: bool
class imitation.rewards.reward_nets.RewardNetWithVariance(observation_space, action_space,
        normalize_images=True)

Bases: RewardNet
A reward net that keeps track of its epistemic uncertainty through variance.

abstract predict_reward_moments(state, action, next_state, done, **kwargs)
    Compute the mean and variance of the reward distribution.

Parameters

• state (ndarray) – Current states of shape (batch_size,) + state_shape.
• action (ndarray) – Actions of shape (batch_size,) + action_shape.
• next_state (ndarray) – Successor states of shape (batch_size,) + state_shape.
• done (ndarray) – End-of-episode (terminal state) indicator of shape (batch_size,).
• **kwargs – may modify the behavior of subclasses

Return type
Tuple[ndarray, ndarray]

Returns

• Estimated reward mean of shape (batch_size,).
• Estimated reward variance of shape (batch_size,). # noqa: DAR202

training: bool
class imitation.rewards.reward_nets.RewardNetWrapper(base)

Bases: RewardNet
Abstract class representing a wrapper modifying a RewardNet's functionality.

In general RewardNetWrapper's should either subclass ForwardWrapper or PredictProcessedWrapper.

__init__(base)
    Initialize a RewardNet wrapper.

Parameters

base (RewardNet) – the base RewardNet to wrap.

property base: RewardNet

Return type

RewardNet

property device: device
    Heuristic to determine which device this module is on.

Return type
device
property dtype:  dtype
    Heuristic to determine dtype of module.
    
    Return type
dtype
preprocess(state, action, next_state, done)
    Preprocess a batch of input transitions and convert it to PyTorch tensors.
    
    The output of this function is suitable for its forward pass, so a typical usage would be model(*model.
    preprocess(transitions)).

    Parameters
    • state (ndarray) – The observation input. Its shape is (batch_size,) + observation_space.shape.
    • action (ndarray) – The action input. Its shape is (batch_size,) + action_space.shape. The None dimension is expected to be the same as None dimension from obs_input.
    • next_state (ndarray) – The observation input. Its shape is (batch_size,) + observation_space.shape.
    • done (ndarray) – Whether the episode has terminated. Its shape is (batch_size,).

    Returns
    a Tuple of tensors containing observations, actions, next observations and dones.

    Return type
    Preprocessed transitions
training:  bool
class imitation.rewards.reward_nets.ShapedRewardNet(base, potential, discount_factor)
    Bases: ForwardWrapper
    A RewardNet consisting of a base network and a potential shaping.

    __init__(base, potential, discount_factor)
    Setup a ShapedRewardNet instance.

    Parameters
    • base (RewardNet) – the base reward net to which the potential shaping will be added. Shaping must be applied directly to the raw reward net. See error below.
    • potential (Callable[[Tensor], Tensor]) – A callable which takes a batch of states (as a PyTorch tensor) and returns a batch of potentials for these states. If this is a PyTorch Module, it becomes a submodule of the ShapedRewardNet instance.
    • discount_factor (float) – discount factor to use for the potential shaping.

    forward(state, action, next_state, done)
    Compute rewards for a batch of transitions and keep gradients.

    training:  bool
imitation.rewards.reward_nets.cnn_transpose(tens)
    Transpose a (b,h,w,c)-formatted tensor to (b,c,h,w) format.

    Return type
    Tensor
**imitation**

**imitation.rewards.reward_wrapper**

Common wrapper for adding custom reward values to an environment.

**Classes**

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RewardVecEnvWrapper(venv, reward_fn[, ...])</code></td>
<td>Uses a provided reward_fn to replace the reward function returned by <code>step()</code>.</td>
</tr>
<tr>
<td><code>WrappedRewardCallback(episode_rewards, ...)</code></td>
<td>Logs mean wrapped reward as part of RL (or other) training.</td>
</tr>
</tbody>
</table>

```python
class imitation.rewards.reward_wrapper.RewardVecEnvWrapper(venv, reward_fn, ep_history=100):
    Bases: VecEnvWrapper
    Uses a provided reward_fn to replace the reward function returned by `step()`.
    Automatically resets the inner VecEnv upon initialization. A tricky part about this class is keeping track of the most recent observation from each environment.
    Will also include the previous reward given by the inner VecEnv in the returned info dict under the `original_env_rew` key.

    __init__(venv, reward_fn, ep_history=100)
    Builds RewardVecEnvWrapper.

    Parameters
    • `venv` (VecEnv) – The VecEnv to wrap.
    • `reward_fn` (RewardFn) – A function that wraps takes in vectorized transitions (obs, act, next_obs) a vector of episode timesteps, and returns a vector of rewards.
    • `ep_history` (int) – The number of episode rewards to retain for computing mean reward.

    property envs

    make_log_callback()
    Creates WrappedRewardCallback connected to this RewardVecEnvWrapper.

    Return type
    WrappedRewardCallback

    reset()
    Reset all the environments and return an array of observations, or a tuple of observation arrays.
    If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

    Returns
    observation

    step_async(actions)
    Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.
    You should not call this if a step_async run is already pending.
```

2.1. imitation
`step_wait()`

Wait for the step taken with `step_async()`.

**Returns**
observation, reward, done, information

class imitation.rewards.reward_wrapper.WrappedRewardCallback(episode_rewards, *args, **kwargs)

Bases: BaseCallback

Logs mean wrapped reward as part of RL (or other) training.

__init__(episode_rewards, *args, **kwargs)

Builds WrappedRewardCallback.

**Parameters**

- **episode_rewards** (Deque[float]) – A queue that episode rewards will be placed into.
- ***args** – Passed through to `callbacks.BaseCallback`.
- ****kwargs** – Passed through to `callbacks.BaseCallback`.

**imitation.rewards.serialize**

Load serialized reward functions of different types.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

- `load_reward(reward_type, reward_path, venv, ...)` Load serialized reward.
- `load_zero(path, venv)`

**rtype**

`RewardFn`

**Classes**

`ValidateRewardFn(reward_fn)` Wrap reward function to add sanity check.

class imitation.rewards.serialize.ValidateRewardFn(reward_fn)

Bases: RewardFn

Wrap reward function to add sanity check.

Checks that the length of the reward vector is equal to the batch size of the input.

__init__(reward_fn)

Builds the reward validator.

**Parameters**

- **reward_fn** (`RewardFn`) – base reward function
\texttt{imitation.rewards.serialize.load_reward}(\texttt{reward_type}, \texttt{reward_path}, \texttt{venv}, **\texttt{kwargs})

Load serialized reward.

Parameters

- \texttt{reward_type} (str) – A key in \texttt{reward_registry}. Valid types include \texttt{RewardNet_normalized}, \texttt{RewardNet_shaped}, \texttt{RewardNet_unnormalized}, \texttt{zero}, \texttt{RewardNet_std_added}, \texttt{RewardNet_unshaped}.
- \texttt{reward_path} (str) – A path specifying the reward.
- \texttt{venv} (\texttt{VecEnv}) – An environment that the policy is to be used with.
- **\texttt{kwargs} – kwargs to pass to reward fn

Return type

\texttt{RewardFn}

Returns

The deserialized reward.

\texttt{imitation.rewards.serialize.load_zero}(\texttt{path}, \texttt{venv})

Return type

\texttt{RewardFn}

\section*{2.1.6 \texttt{imitation.scripts}}

Command-line scripts.

\textbf{Modules}

\begin{tabular}{ll}
\texttt{imitation.scripts.analyze} & Commands to analyze experimental results. \\
\texttt{imitation.scripts.common} & Common configuration elements for scripts. \\
\texttt{imitation.scripts.config} & Configuration settings for scripts. \\
\texttt{imitation.scripts.convert_trajs} & Converts old-style pickle trajectories to new-style NPZ trajectories. \\
\texttt{imitation.scripts.eval_policy} & Evaluate policies: render policy interactively, save videos, log episode return. \\
\texttt{imitation.scripts.train_adversarial} & Train GAIL or AIRL. \\
\texttt{imitation.scripts.train_imitation} & Trains DAgger on synthetic demonstrations generated from an expert policy. \\
\texttt{imitation.scripts.train_preference_comparisons} & Train a reward model using preference comparisons. \\
\texttt{imitation.scripts.train_rl} & Uses RL to train a policy from scratch, saving rollouts and policy.
\end{tabular}
imitation

imitation.scripts.analyze

Commands to analyze experimental results.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```
analyze_imitation(csv_output_path, ...) Parse Sacred logs and generate a DataFrame for imitation learning results.
gather_tb_directories() Gather Tensorboard directories from a parallel_ex run.
main_console()
```

```python
imitation.scripts.analyze.analyze_imitation(csv_output_path, tex_output_path, print_table, table_verbosity)
```

Parse Sacred logs and generate a DataFrame for imitation learning results.

This function calls the helper `_gather_sacred_dicts`, which captures its arguments automatically via Sacred. Provide those arguments to select which Sacred results to parse.

**Parameters**

- `csv_output_path` (Optional[str]) – If provided, then save a CSV output file to this path.
- `tex_output_path` (Optional[str]) – If provided, then save a LaTeX-format table to this path.
- `print_table` (bool) – If True, then print the dataframe to stdout.
- `table_verbosity` (int) – Increasing levels of verbosity, from 0 to 2, increase the number of columns in the table.

**Return type**

DataFrame

**Returns**

The DataFrame generated from the Sacred logs.

```python
imitation.scripts.analyze.gather_tb_directories()
```

Gather Tensorboard directories from a parallel_ex run.

The directories are copied to a unique directory in `/tmp/analysis_tb/` under subdirectories matching the Tensorboard events’ Ray Tune trial names.

This function calls the helper `_gather_sacred_dicts`, which captures its arguments automatically via Sacred. Provide those arguments to select which Sacred results to parse.

**Return type**

dict

**Returns**

A dict with two keys. “gather_dir” (str) is a path to a /tmp/ directory containing all the TensorBoard runs filtered from `source_dir`. “n_tb_dirs” (int) is the number of TensorBoard directories that were filtered.
Raises

`OSError` – If the symlink cannot be created.

```
imitation.scripts.analyze.main_console()
```

### imitation.scripts.common

Common configuration elements for scripts.

#### Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imitation.scripts.common.common</code></td>
<td>Common configuration elements for scripts.</td>
</tr>
<tr>
<td><code>imitation.scripts.common.demonstrations</code></td>
<td>Common configuration element for scripts learning from demonstrations.</td>
</tr>
<tr>
<td><code>imitation.scripts.common.expert</code></td>
<td>Common configuration elements for loading of expert policies.</td>
</tr>
<tr>
<td><code>imitation.scripts.common.reward</code></td>
<td>Common configuration elements for reward network training.</td>
</tr>
<tr>
<td><code>imitation.scripts.common.rl</code></td>
<td>Common configuration elements for reinforcement learning.</td>
</tr>
<tr>
<td><code>imitation.scripts.common.train</code></td>
<td>Common configuration elements for training imitation algorithms.</td>
</tr>
<tr>
<td><code>imitation.scripts.common.wb</code></td>
<td>Weights &amp; Biases configuration elements for scripts.</td>
</tr>
</tbody>
</table>

#### Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```
hook(config, command_name, logger)
```

- `make_log_dir(_run, log_dir, log_level)` Creates log directory and sets up symlink to Sacred logs.
- `make_rng(seed)` Creates a `np.random.Generator` with the given seed.
- `make_venv(env_name, num_vec, parallel, ...)` Builds the vector environment.
- `setup_logging(_run, log_format_strs)` Builds the imitation logger.

```
imitation.scripts.common.common.hook(config, command_name, logger)
```

```
imitation.scripts.common.common.make_log_dir(_run, log_dir, log_level)
```

Creates log directory and sets up symlink to Sacred logs.

**Parameters**

- `log_dir (str)` – The directory to log to.

---

2.1. imitation
• **log_level** ([Union[int, str]]) – The threshold of the logger. Either an integer level (10, 20, ...), a string of digits (‘10’, ‘20’), or a string of the designated level (‘DEBUG’, ‘INFO’, ...).

**Return type**

Path

**Returns**

The `log_dir`. This avoids the caller needing to capture this argument.

```python
imitation.scripts.common.common.make_rng(_seed)
```

Creates a `np.random.Generator` with the given seed.

**Return type**

Generator

```python
imitation.scripts.common.common.make_venv(env_name, num_vec, parallel, log_dir, max_episode_steps, env_make_kwargs, **kwargs)
```

Builds the vector environment.

**Parameters**

- **env_name** (str) – The environment to train in.
- **num_vec** (int) – Number of `gym.Env` instances to combine into a vector environment.
- **parallel** (bool) – Whether to use “true” parallelism. If True, then use `SubProcVecEnv`. Otherwise, use `DummyVecEnv` which steps through environments serially.
- **max_episode_steps** (int) – If not None, then a TimeLimit wrapper is applied to each environment to artificially limit the maximum number of timesteps in an episode.
- **log_dir** (str) – Logs episode return statistics to a subdirectory `monitor`.
- **env_make_kwargs** (Mapping[str, Any]) – The kwargs passed to `spec.make` of a gym environment.
- **kwargs** – Passed through to `util.make_vec_env`.

**Yields**

The constructed vector environment.

**Return type**

Generator[VecEnv, None, None]

```python
imitation.scripts.common.common.setup_logging(_run, log_format_strs)
```

Builds the imitation logger.

**Parameters**

- **log_format_strs** (Sequence[str]) – The types of formats to log to.

**Return type**

Tuple[HierarchicalLogger, Path]

**Returns**

The configured imitation logger and `log_dir`. Returning `log_dir` avoids the caller needing to capture this value.
Common configuration element for scripts learning from demonstrations.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

- `generate_expert_trajs(n_expert_demos)` Generates expert demonstrations.
  - **Parameters**
    - `n_expert_demos` (Optional[int]) – The number of trajectories to load. Dataset is truncated to this length if specified.
  - **Return type**
    - Optional[Sequence[Trajectory]]
  - **Returns**
    - The expert trajectories.
  - **Raises**
    - `ValueError` – If `n_expert_demos` is None.

- `get_expert_trajectories(rollout_path)`
  - **Return type**
    - Sequence[Trajectory]

- `load_expert_trajs(rollout_path, n_expert_demos)` Loads expert demonstrations.
  - **Parameters**
    - `rollout_path` (str) – A path containing a pickled sequence of types.Trajectory.
    - `n_expert_demos` (Optional[int]) – The number of trajectories to load. Dataset is truncated to this length if specified.
  - **Return type**
    - Sequence[Trajectory]
  - **Returns**
    - The expert trajectories.
  - **Raises**
    - `ValueError` – There are fewer trajectories than `n_expert_demos`. 
imitation

imitation.scripts.common.expert

Common configuration elements for loading of expert policies.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

config_hook(config, command_name, logger)

get_expert_policy(venv, policy_type, ...)
```

imitation.scripts.common.expert.config_hook(config, command_name, logger)

imitation.scripts.common.expert.get_expert_policy(venv, policy_type, loader_kwargs, common)

imitation.scripts.common.reward

Common configuration elements for reward network training.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

config_hook(config, command_name, logger)

make_reward_net(venv, net_cls, net_kwargs, ...)
```

imitation.scripts.common.reward.config_hook(config, command_name, logger)

Sets default values for net_cls and net_kwargs.

imitation.scripts.common.reward.make_reward_net(venv, net_cls, net_kwargs, normalize_output_layer, add_std_alpha, ensemble_size, ensemble_member_config)

Builds a reward network.

Parameters

- **venv** (VecEnv) – Vectorized environment reward network will predict reward for.
- **net_cls** (Type[RewardNet]) – Class of reward network to construct.
- **net_kwargs** (Mapping[str, Any]) – Keyword arguments passed to reward network constructor.
- **normalize_output_layer** (Optional[Type[BaseNorm]]) – Wrapping the reward_net with NormalizedRewardNet to normalize the reward output.
• **add_std_alpha** *(Optional*[float]*) – multiple of reward function standard deviation to add to the reward in `predict_processed`. Must be None when using a reward function that does not keep track of variance. Defaults to None.

• **ensemble_size** *(Optional*[int]*) – The number of ensemble members to create. Must set if using `net_cls = :class: reward_nets.RewardEnsemble`.


**Return type**

*RewardNet*

**Returns**

A, possibly wrapped, instance of `net_cls`.

**Raises**

*ValueError* – Using a reward ensemble but failed to provide configuration.

---

**imitation.scripts.common.rl**

Common configuration elements for reinforcement learning.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>config_hook(config, command_name, logger)</code></td>
<td>Sets defaults equivalent to sb3.PPO default hyperparameters.</td>
</tr>
<tr>
<td><code>load_rl_algo_from_path(seed, agent_path, ...)</code></td>
<td>rtype <code>BaseAlgorithm</code></td>
</tr>
<tr>
<td><code>make_rl_algo(venv, rl_cls, batch_size, ...)</code></td>
<td>Instantiates a Stable Baselines3 RL algorithm.</td>
</tr>
</tbody>
</table>

**imitation.scripts.common.rl.config_hook(config, command_name, logger)**

Sets defaults equivalent to sb3.PPO default hyperparameters.

**imitation.scripts.common.rl.load_rl_algo_from_path(seed, agent_path, ...)**

rtpe `BaseAlgorithm`

**imitation.scripts.common.rl.make_rl_algo(venv, rl_cls, batch_size, ...)**

Instantiates a Stable Baselines3 RL algorithm.

**Parameters**

• **venv** *(VecEnv)* – The vectorized environment to train on.
imitation

- **rl_cls** (Type[BaseAlgorithm]) – Type of a Stable Baselines3 RL algorithm.
- **batch_size** (int) – The batch size of the RL algorithm.
- **rl_kwargs** (Mapping[str, Any]) – Keyword arguments for RL algorithm constructor.
- **train** (Mapping[str, Any]) – Configuration for the train ingredient. We need the policy_cls and policy_kwargs component.
- **relabel_reward_fn** (Optional[RewardFn]) – Reward function used for reward relabeling in replay or rollout buffers of RL algorithms.

Return type
 BaseAlgorithm

Returns
 The RL algorithm.

Raises
- **ValueError** – gen_batch_size not divisible by venv.num_envs.
- **TypeError** – rl_cls is neither OnPolicyAlgorithm nor OffPolicyAlgorithm.

imitation.scripts.common.train

Common configuration elements for training imitation algorithms.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eval_policy(rl_algo, venv, n_episodes_eval)</code></td>
<td>Evaluation of imitation learned policy. Has the side effect of setting rl_algo’s environment to venv if it is a BaseAlgorithm.</td>
</tr>
<tr>
<td><code>suppress_sacred_error(policy_kwargs)</code></td>
<td>No-op so Sacred recognizes policy_kwargs is used (in rl and elsewhere).</td>
</tr>
</tbody>
</table>

imitation.scripts.common.train.eval_policy(rl_algo, venv, n_episodes_eval)

Evaluation of imitation learned policy.

Parameters

- **rl_algo** (Union[BaseAlgorithm, BasePolicy]) – Algorithm to evaluate.
- **venv** (VecEnv) – Environment to evaluate on.
- **n_episodes_eval** (int) – The number of episodes to average over when calculating the average episode reward of the imitation policy for return.

Return type
 Mapping[str, float]

Returns
 A dictionary with two keys. “imit_stats” gives the return value of rollout_stats() on rollouts test-reward-wrapped environment, using the final policy (remember that the ground-truth reward
can be recovered from the “monitor_return” key). “expert_stats” gives the return value of `roll-out_stats()` on the expert demonstrations loaded from `rollout_path`.

`imitation.scripts.common.train.suppress_sacred_error(policy_kwargs)`

No-op so Sacred recognizes `policy_kwargs` is used (in rl and elsewhere).

```
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```
wandb_init(_run, wandb_name_prefix, ...) Putting everything together to get the W&B kwags for wandb.init().
```

```
imitation.scripts.common.wb.wandb_init(_run, wandb_name_prefix, wandb_tag, wandb_kwags, wandb_additional_info, log_dir)
Putting everything together to get the W&B kwags for wandb.init().
```

**Parameters**

- `wandb_name_prefix` (str) – User-specified prefix for wandb run name.
- `wandb_tag` (Optional[str]) – User-specified tag for this run.
- `wandb_kwags` (Mapping[str, Any]) – User-specified kwags for wandb.init().
- `wandb_additional_info` (Mapping[str, Any]) – User-specific additional info to add to wandb experiment config.
- `log_dir` (str) – W&B logs will be stored in directory `{log_dir}/wandb/`.

**Raises**

- `ModuleNotFoundError` – wandb is not installed.

**Return type**

- None

```
imitation.scripts.config
```

Configuration settings for scripts.
### Modules

<table>
<thead>
<tr>
<th><code>imitation.scripts.config.analyze</code></th>
<th>Configuration settings for analyze, inspecting results from completed experiments.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imitation.scripts.config.eval_policy</code></td>
<td>Configuration settings for eval_policy, evaluating pre-trained policies.</td>
</tr>
<tr>
<td><code>imitation.scripts.config.train_adversarial</code></td>
<td>Configuration for imitation.scripts.train_adversarial.</td>
</tr>
<tr>
<td><code>imitation.scripts.config.train_imitation</code></td>
<td>Configuration settings for train_dagger, training DAgger from synthetic demos.</td>
</tr>
<tr>
<td><code>imitation.scripts.config.train_preference_comparisons</code></td>
<td>Configuration for imitation.scripts.train_preference_comparisons.</td>
</tr>
<tr>
<td><code>imitation.scripts.config.train_rl</code></td>
<td>Configuration settings for train_rl, training a policy with RL.</td>
</tr>
</tbody>
</table>

**imitation.scripts.config.analyze**

Configuration settings for analyze, inspecting results from completed experiments.

**imitation.scripts.config.eval_policy**

Configuration settings for eval_policy, evaluating pre-trained policies.

**imitation.scripts.config.train_adversarial**

Configuration for imitation.scripts.train_adversarial.

**imitation.scripts.config.train_imitation**

Configuration settings for train_dagger, training DAgger from synthetic demos.

**imitation.scripts.config.train_preference_comparisons**

Configuration for imitation.scripts.train_preference_comparisons.

**imitation.scripts.config.train_rl**

Configuration settings for train_rl, training a policy with RL.
imitation.scripts.convert_trajs

Converts old-style pickle trajectories to new-style NPZ trajectories.

See https://github.com/HumanCompatibleAI/imitation/pull/448 for a description of the new trajectory format.

This script takes as command-line input multiple paths to saved trajectories, in the old .pkl or new .npz format. It then saves each sequence in the new .npz format. The path is the same as the original but with an “.npz” extension (i.e. “A.pkl” -> “A.npz”, “A.npz” -> “A.npz”, “A” -> “A.npz”, “A.foo” -> “A.foo.npz”).

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

main()

update_traj_file_in_place(path_str, /) Modifies trajectories pickle file in-place to update data
to new format.
```

imitation.scripts.convert_trajs.main()

imitation.scripts.convert_trajs.update_traj_file_in_place(path_str, /) Modifies trajectories pickle file in-place to update data to new format.

The new data is saved as `Sequence[imitation.types.TrajectoryWithRew]`.

Parameters

- `path_str` (str) – Path to a pickle file containing `Sequence[imitation.types.Trajectory]` or `Sequence[imitation.old_types.TrajectoryWithRew]`.

Return type

None

imitation.scripts.eval_policy

Evaluate policies: render policy interactively, save videos, log episode return.

Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

eval_policy(_run, eval_n_timesteps, ...[, ...]) Rolls a policy out in an environment, collecting statistics.

main_console()

video_wrapper_factory(log_dir, **kwargs) Returns a function that wraps the environment in a video recorder.
```
Classes

**InteractiveRender**(venv, fps)

Render the wrapped environment(s) on screen.

```python
class imitation.scripts.eval_policy.InteractiveRender(venv, fps)
    Bases: VecEnvWrapper
    Render the wrapped environment(s) on screen.
    __init__(venv, fps)
        Builds renderer for venv running at fps frames per second.
    reset()
        Reset all the environments and return an array of observations, or a tuple of observation arrays.
        If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.
            Returns
                observation
    step_wait()
        Wait for the step taken with step_async().
            Returns
                observation, reward, done, information
```

**eval_policy**(run, eval_n_timesteps, eval_n_episodes, render, render_fps, videos, video_kwargs, reward_type=None, reward_path=None, rollout_save_path=None, explore_kwargs=None)

Rolls a policy out in an environment, collecting statistics.

**Parameters**

- **eval_n_timesteps** (Optional[int]) – Minimum number of timesteps to evaluate for. Set exactly one of `eval_n_episodes` and `eval_n_timesteps`.
- **eval_n_episodes** (Optional[int]) – Minimum number of episodes to evaluate for. Set exactly one of `eval_n_episodes` and `eval_n_timesteps`.
- **render** (bool) – If True, renders interactively to the screen.
- **render_fps** (int) – The target number of frames per second to render on screen.
- **videos** (bool) – If True, saves videos to `log_dir`.
- **video_kwargs** (Mapping[str, Any]) – Keyword arguments passed through to `video_wrapper.VideoWrapper`.
- **reward_type** (Optional[str]) – If specified, overrides the environment reward with a reward of this.
- **reward_path** (Optional[str]) – If `reward_type` is specified, the path to a serialized reward of `reward_type` to override the environment reward with.
- **rollout_save_path** (Optional[str]) – where to save rollouts used for computing stats to disk; if None, then do not save.
• **explore_kwargs** (Optional[Mapping[str, Any]]) – keyword arguments to an exploration wrapper to apply before rolling out, not including policy_callable, venv, and rng; if None, then do not wrap.

**Returns**

Return value of `imitation.util.rollout.rollout_stats()`.

`imitation.scripts.eval_policy.main_console()`

`imitation.scripts.eval_policy.video_wrapper_factory(log_dir, **kwargs)`

Returns a function that wraps the environment in a video recorder.

**imitation.scripts.train_adversarial**

Train GAIL or AIRL.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

`airl()`

`gail()`

`main_console()`

`save(trainer, save_path)`

Save discriminator and generator.

`train_adversarial(_run, show_config, ...)`

Train an adversarial-network-based imitation learning algorithm.

`imitation.scripts.train_adversarial.airl()`

`imitation.scripts.train_adversarial.gail()`

`imitation.scripts.train_adversarial.main_console()`

`imitation.scripts.train_adversarial.save(trainer, save_path)`

Save discriminator and generator.

`imitation.scripts.train_adversarial.train_adversarial(_run, show_config, algo_cls, algorithm_kwargs, total_timesteps, checkpoint_interval, agent_path)`

Train an adversarial-network-based imitation learning algorithm.

**Checkpoints:**

• AdversarialTrainer train and test RewardNets are saved to

  \[ f"{log_dir}/checkpoints/{step}/reward_{train,test}.pt" \]

  where step is either the training round or “final”.

• Generator policies are saved to \[ f"{log_dir}/checkpoints/{step}/gen_policy/" \].
Parameters

- **show_config** (bool) – Print the merged config before starting training. This is analogous to the print_config command, but will show config after rather than before merging algorithm_specific arguments.

- **algo_cls** (Type[AdversarialTrainer]) – The adversarial imitation learning algorithm to use.

- **algorithm_kwargs** (Mapping[str, Any]) – Keyword arguments for the GAIL or AIRL constructor.

- **total_timesteps** (int) – The number of transitions to sample from the environment during training.

- **checkpoint_interval** (int) – Save the discriminator and generator models every checkpoint_interval rounds and after training is complete. If 0, then only save weights after training is complete. If <0, then don’t save weights at all.

- **agent_path** (Optional[str]) – Path to a directory containing a pre-trained agent. If provided, then the agent will be initialized using this stored policy (warm start). If not provided, then the agent will be initialized using a random policy.

Return type

Mapping[str, Mapping[str, float]]

Returns

A dictionary with two keys. “imit_stats” gives the return value of rollout_stats() on rollouts test-reward-wrapped environment, using the final policy (remember that the ground-truth reward can be recovered from the “monitor_return” key). “expert_stats” gives the return value of rollout_stats() on the expert demonstrations.

**imitation.scripts.train_imitation**

Trains DAgger on synthetic demonstrations generated from an expert policy.

Functions

```
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bc()</strong></td>
<td>Run BC experiment using a Sacred interface to BC.</td>
</tr>
<tr>
<td><strong>dagger()</strong></td>
<td>Run synthetic DAgger experiment using a Sacred interface to SimpleDAggerTrainer.</td>
</tr>
<tr>
<td><strong>main_console()</strong></td>
<td></td>
</tr>
<tr>
<td><strong>make_policy(venv, policy_cl, policy_kw, ...)</strong></td>
<td>Makes policy.</td>
</tr>
<tr>
<td><strong>train_imitation(_run, bc_kw, ...)</strong></td>
<td>Runs DAgger (if use_dagger) or BC (otherwise) training.</td>
</tr>
</tbody>
</table>

**imitation.scripts.train_imitation.bc()**

Run BC experiment using a Sacred interface to BC.

Return type

Mapping[str, Mapping[str, float]]
Returns
Statistics for rollouts from the trained policy and expert data.

```python
 imitation.scripts.train_imitation.dagger()
```

Run synthetic DAgger experiment using a Sacred interface to SimpleDAggerTrainer.

**Return type**
```python
 Mapping[str, Mapping[str, float]]
```

**Returns**
Statistics for rollouts from the trained policy and expert data.

```python
 imitation.scripts.train_imitation.main_console()
```

```python
 imitation.scripts.train_imitation.make_policy(venv, policy_cls, policy_kwars, agent_path)
```

Makes policy.

**Parameters**
- **venv** (VecEnv) – Vectorized environment we will be imitating demos from.
- **policy_cls** (Type[BasePolicy]) – Type of a Stable Baselines3 policy architecture. Specify only if policy_path is not specified.
- **policy_kwars** (Mapping[str, Any]) – Keyword arguments for policy constructor. Specify only if policy_path is not specified.
- **agent_path** (Optional[str]) – Path to serialized policy. If provided, then load the policy from this path. Otherwise, make a new policy. Specify only if policy_cls and policy_kwargs are not specified.

**Return type**
```python
 BasePolicy
```

**Returns**
A Stable Baselines3 policy.

```python
 imitation.scripts.train_imitation.train_imitation(_run, bc_kwars, bc_train_kwars, dagger, use_dagger, agent_path)
```

Runs DAgger (if use_dagger) or BC (otherwise) training.

**Parameters**
- **bc_kwars** (Mapping[str, Any]) – Keyword arguments passed through to bc. BC constructor.
- **bc_train_kwars** (Mapping[str, Any]) – Keyword arguments passed through to BC.train() method.
- **dagger** (Mapping[str, Any]) – Arguments for DAgger training.
- **use_dagger** (bool) – If True, train using DAgger; otherwise, use BC.
- **agent_path** (Optional[str]) – Path to serialized policy. If provided, then load the policy from this path. Otherwise, make a new policy. Specify only if policy_cls and policy_kwargs are not specified.

**Return type**
```python
 Mapping[str, Mapping[str, float]]
```

**Returns**
Statistics for rollouts from the trained policy and demonstration data.
**imitation**

**imitation.scripts.train_preference_comparisons**

Train a reward model using preference comparisons. Can be used as a CLI script, or the `train_preference_comparisons` function can be called directly.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

main_console()

save_checkpoint(trainer, save_path, ...)  # Save reward model and optionally policy.
save_model(agent_trainer, save_path)    # Save the model as model.zip.
train_preference_comparisons(...)       # Train a reward model using preference comparisons.
```

**imitation.scripts.train_preference_comparisons**. `main_console()`

**imitation.scripts.train_preference_comparisons**. `save_checkpoint`(*trainer, save_path, ...*)

Save reward model and optionally policy.

**imitation.scripts.train_preference_comparisons**. `save_model`(*agent_trainer, save_path*)

Save the model as model.zip.
Train a reward model using preference comparisons.

**Parameters**

- `total_timesteps` (int) – number of environment interaction steps
- `total_comparisons` (int) – number of preferences to gather in total
- `num_iterations` (int) – number of times to train the agent against the reward model and then train the reward model against newly gathered preferences.
- `comparison_queue_size` (Optional[int]) – the maximum number of comparisons to keep in the queue for training the reward model. If None, the queue will grow without bound as new comparisons are added.
- `fragment_length` (int) – number of timesteps per fragment that is used to elicit preferences
- `transition_oversampling` (float) – factor by which to oversample transitions before creating fragments. Since fragments are sampled with replacement, this is usually chosen > 1 to avoid having the same transition in too many fragments.
- `initial_comparison_frac` (float) – fraction of `total_comparisons` that will be sampled before the rest of training begins (using the randomly initialized agent). This can be used to
pretrain the reward model before the agent is trained on the learned reward.

- **exploration_frac** (float) – fraction of trajectory samples that will be created using partially random actions, rather than the current policy. Might be helpful if the learned policy explores too little and gets stuck with a wrong reward.

- **trajectory_path** (Optional[str]) – either None, in which case an agent will be trained and used to sample trajectories on the fly, or a path to a pickled sequence of TrajectoryWithRew to be trained on.

- **trajectory_generator_kwargs** (Mapping[str, Any]) – kwargs to pass to the trajectory generator.

- **save_preferences** (bool) – if True, store the final dataset of preferences to disk.

- **agent_path** (Optional[str]) – if given, initialize the agent using this stored policy rather than randomly.

- **preference_model_kwargs** (Mapping[str, Any]) – passed to PreferenceModel

- **reward_trainer_kwargs** (Mapping[str, Any]) – passed to BasicRewardTrainer or EnsembleRewardTrainer

- **gatherer_cls** (Type[PreferenceGatherer]) – type of PreferenceGatherer to use (defaults to SyntheticGatherer)

- **gatherer_kwargs** (Mapping[str, Any]) – passed to the PreferenceGatherer specified by gatherer_cls

- **active_selection** (bool) – use active selection fragmenter instead of random fragmenter

- **active_selection_oversampling** (int) – factor by which to oversample random fragments from the base fragmenter of active selection. this is usually chosen > 1 to allow the active selection algorithm to pick fragment pairs with highest uncertainty. = 1 implies no active selection.

- **uncertainty_on** (str) – passed to ActiveSelectionFragmenter

- **fragmenter_kwargs** (Mapping[str, Any]) – passed to RandomFragmenter

- **allow_variable_horizon** (bool) – If False (default), algorithm will raise an exception if it detects trajectories of different length during training. If True, overrides this safety check. WARNING: variable horizon episodes leak information about the reward via termination condition, and can seriously confound evaluation. Read https://imitation.readthedocs.io/en/latest/guide/variable_horizon.html before overriding this.

- **checkpoint_interval** (int) – Save the reward model and policy models (if trajectory_generator contains a policy) every checkpoint_interval iterations and after training is complete. If 0, then only save weights after training is complete. If <0, then don’t save weights at all.

- **query_schedule** (Union[str, Callable[[float], float]]) – one of (“constant”, “hyperbolic”, “inverse_quadratic”). A function indicating how the total number of preference queries should be allocated to each iteration. “hyperbolic” and “inverse_quadratic” apportion fewer queries to later iterations when the policy is assumed to be better and more stable.

**Return type**

Mapping[str, Any]

**Returns**

Rollout statistics from trained policy.
Raises

ValueError – Inconsistency between config and deserialized policy normalization.

**imitation.scripts.train_rl**

Uses RL to train a policy from scratch, saving rollouts and policy.

**This can be used:**

1. To train a policy on a ground-truth reward function, as a source of synthetic “expert” demonstrations to train IRL or imitation learning algorithms.
2. To train a policy on a learned reward function, to solve a task or as a way of evaluating the quality of the learned reward function.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```python
main_console()
```

```python
train_rl(*, total_timesteps, ...)
```

Trains an expert policy from scratch and saves the rollouts and policy.

### imitation.scripts.train_rl.main_console()

### imitation.scripts.train_rl.train_rl(*, total_timesteps, normalize_reward, normalize_kwargs, reward_type, reward_path, load_reward_kwargs, rollout_save_final, rollout_save_n_timesteps, rollout_save_n_episodes, policy_save_interval, policy_save_final, agent_path)

Trains an expert policy from scratch and saves the rollouts and policy.

**Checkpoints:**

At applicable training steps `step` (where step is either an integer or “final”):

- Policies are saved to `{log_dir}/policies/{step}/`.
- Rollouts are saved to `{log_dir}/rollouts/{step}.npz`.

**Parameters**

- **total_timesteps** (int) – Number of training timesteps in `model.learn()`.
- **normalize_reward** (bool) – Applies normalization and clipping to the reward function by keeping a running average of training rewards. Note: this is may be redundant if using a learned reward that is already normalized.
- **normalize_kwargs** (dict) – kwargs for `VecNormalize`.
- **reward_type** (Optional[str]) – If provided, then load the serialized reward of this type, wrapping the environment in this reward. This is useful to test whether a reward model transfers. For more information, see `imitation.rewards.serialize.load_reward`.

2.1. imitation
• **reward_path** (Optional[str]) – A specifier, such as a path to a file on disk, used by reward_type to load the reward model. For more information, see `imitation.rewards.serialize.load_reward`.

• **load_reward_kwargs** (Optional[Mapping[str, Any]]) – Additional kwargs to pass to `predict_processed`. Examples are ‘alpha’ for :class: `AddSTDRewardWrapper` and ‘update_stats’ for :class: `NormalizedRewardNet`.

• **rollout_save_final** (bool) – If True, then save rollouts right after training is finished.

• **rollout_save_n_timesteps** (Optional[int]) – The minimum number of timesteps saved in every file. Could be more than `rollout_save_n_timesteps` because trajectories are saved by episode rather than by transition. Must set exactly one of `rollout_save_n_timesteps` and `rollout_save_n_episodes`.

• **rollout_save_n_episodes** (Optional[int]) – The number of episodes saved in every file. Must set exactly one of `rollout_save_n_timesteps` and `rollout_save_n_episodes`.

• **policy_save_interval** (int) – The number of training updates between intermediate rollout saves. If the argument is nonpositive, then don’t save intermediate updates.

• **policy_save_final** (bool) – If True, then save the policy right after training is finished.

• **agent_path** (Optional[str]) – Path to load warm-started agent.

**Return type**
```
Mapping[str, float]
```

**Returns**
The return value of `rollout_stats()` using the final policy.

### 2.1.7 imitation.testing

Helper methods for unit tests.

May also be useful for users of imitation.

**Modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imitation.testing.expert_trajectories</code></td>
<td>Test utilities to conveniently generate expert trajectories.</td>
</tr>
<tr>
<td><code>imitation.testing.reward_improvement</code></td>
<td>Utility functions used to check if rewards improved wrt to previous rewards.</td>
</tr>
<tr>
<td><code>imitation.testing.reward_nets</code></td>
<td>Utility functions for testing reward nets.</td>
</tr>
</tbody>
</table>

**imitation.testing.expert_trajectories**

Test utilities to conveniently generate expert trajectories.
Functions

# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

```
```

generate_expert_trajectories(env_id, ...) Generate expert trajectories for the given environment.
lazy_generate_expert_trajectories(...) Generate or load expert trajectories from cache.
make_expert_transition_loader(cache_dir, ...) Creates different kinds of PyTorch data loaders for expert transitions.

```
```

imitation.testing.expert_trajectories.generate_expert_trajectories(env_id, num_trajectories, rng)

Generate expert trajectories for the given environment.

Note: will just pull a pretrained policy from the Hugging Face model hub.

Parameters

• `env_id` (str) – The environment to generate trajectories for.
• `num_trajectories` (int) – The number of trajectories to generate.
• `rng` (Generator) – The random number generator to use.

Return type

Sequence[TrajectoryWithRew]

Returns

A list of trajectories with rewards.

imitation.testing.expert_trajectories.lazy_generate_expert_trajectories(cache_path, env_id, num_trajectories, rng)

Generate or load expert trajectories from cache.

Parameters

• `cache_path` (PathLike) – A path to the folder to be used as cache for the expert trajectories.
• `env_id` (str) – The environment to generate trajectories for.
• `num_trajectories` (int) – The number of trajectories to generate.
• `rng` (Generator) – The random number generator to use.

Return type

Sequence[TrajectoryWithRew]

Returns

A list of trajectories with rewards.

imitation.testing.expert_trajectories.make_expert_transition_loader(cache_dir, batch_size, expert_data_type, env_name, rng, num_trajectories=1)

Creates different kinds of PyTorch data loaders for expert transitions.

Parameters

• `cache_dir` (Path) – The directory to use for caching the expert trajectories.
imitation

- **batch_size** (int) – The batch size to use for the data loader.
- **expert_data_type** (str) – The type of expert data to use. Can be one of “data_loader”, “ducktyped_data_loader”, “transitions”.
- **env_name** (str) – The environment to generate trajectories for.
- **rng** (Generator) – The random number generator to use.
- **num_trajectories** (int) – The number of trajectories to generate.

**Raises**

ValueError – If `expert_data_type` is not one of the supported types.

**Returns**

A pytorch data loader for expert transitions.

**imitation.testing.reward_improvement**

Utility functions used to check if rewards improved wrt to previous rewards.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

- `is_significant_reward_improvement(...[, p_value])` Checks if the new rewards are really better than the old rewards.
- `mean_reward_improved_by(old_rews, new_rews, ...)` Checks if mean rewards improved wrt.

**imitation.testing.reward_improvement.is_significant_reward_improvement**

Checks if the new rewards are really better than the old rewards.

Ensures that this is not just due to lucky sampling by a permutation test.

**Parameters**

- **old_rewards** (Iterable[float]) – Iterable of “old” trajectory rewards (e.g. before training).
- **new_rewards** (Iterable[float]) – Iterable of “new” trajectory rewards (e.g. after training).
- **p_value** (float) – The maximum probability, that the old rewards are just as good as the new rewards, that we tolerate.

**Return type**

bool

**Returns**

True, if the new rewards are most probably better than the old rewards. For this, the probability, that the old rewards are just as good as the new rewards must be below `p_value`.
```python
>>> is_significant_reward_improvement((5, 6, 7, 4, 4), (7, 5, 9, 9, 12))
True

>>> is_significant_reward_improvement((5, 6, 7, 4, 4), (7, 5, 9, 7, 4))
False

>>> is_significant_reward_improvement((5, 6, 7, 4, 4), (7, 5, 9, 7, 4), p_value=0.3)
True
```

```python
imitation.testing.reward_improvement.mean_reward_improved_by(old_rews, new_rews, min_improvement)
```

Checks if mean rewards improved wrt. to old rewards by a certain amount.

**Parameters**

- `old_rews` ([Iterable[float]]) – Iterable of “old” trajectory rewards (e.g. before training).
- `new_rews` ([Iterable[float]]) – Iterable of “new” trajectory rewards (e.g. after training).
- `min_improvement` (float) – The minimum amount of improvement that we expect.

**Returns**

`True` if the mean of the new rewards is larger than the mean of the old rewards by min_improvement.

```python
>>> mean_reward_improved_by([5, 8, 7], [8, 9, 10], 2)
True

>>> mean_reward_improved_by([5, 8, 7], [8, 9, 10], 5)
False
```

**imitation.testing.reward_nets**

Utility functions for testing reward nets.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```python
make_ensemble(obs_space, action_space[, ...])
```

Create a simple reward ensemble.

2.1. imitation
**MockRewardNet** (observation_space, action_space)  
A mock reward net for testing.

class imitation.testing.reward_nets.MockRewardNet(observation_space, action_space, value=0.0)  
Bases: RewardNet  
A mock reward net for testing.

__init__ (observation_space, action_space, value=0.0)  
Create mock reward.

Parameters
- **observation_space** (Space) – observation space of the env
- **action_space** (Space) – action space of the env
- **value** (float) – The reward to always return. Defaults to 0.0.

forward (state, action, next_state, done)  
Compute rewards for a batch of transitions and keep gradients.

Return type  
Tensor

training: bool

imitation.testing.reward_nets.make_ensemble(obs_space, action_space, num_members=2, **kwargs)  
Create a simple reward ensemble.

### 2.1.8 imitation.util

General utility functions: e.g. logging, configuration, etc.

**Modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation.util.logger</td>
<td>Logging for quantitative metrics and free-form text.</td>
</tr>
<tr>
<td>imitation.util.networks</td>
<td>Helper methods to build and run neural networks.</td>
</tr>
<tr>
<td>imitation.util.registry</td>
<td>Registry mapping IDs to objects, such as environments or policy loaders.</td>
</tr>
<tr>
<td>imitation.util.sacred</td>
<td>Helper methods for the sacred experimental configuration and logging framework.</td>
</tr>
<tr>
<td>imitation.util.util</td>
<td>Miscellaneous utility methods.</td>
</tr>
<tr>
<td>imitation.util.video_wrapper</td>
<td>Wrapper to record rendered video frames from an environment.</td>
</tr>
</tbody>
</table>
**imitation.util.logger**

Logging for quantitative metrics and free-form text.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass

configure([folder, format_strs]) Configure Stable Baselines logger to be `accumulate_means()`-compatible.

make_output_format(_format, log_dir[, ...]) Returns a logger for the requested format.
```

**Classes**

```python
HierarchicalLogger(default_logger[, format_strs]) A logger supporting contexts for accumulating mean values.

WandbOutputFormat() A stable-baseline logger that writes to wandb.
```

class imitation.util.logger.HierarchicalLogger(default_logger[, format_strs])

Bases: Logger

A logger supporting contexts for accumulating mean values.

`self.accumulate_means` creates a context manager. While in this context, values are logged to a sub-logger, with only mean values recorded in the top-level (root) logger.

```python
>>> import tempfile
>>> with tempfile.TemporaryDirectory() as dir:
...     logger = HierarchicalLogger(dir, ('log',))
...     # record the key value pair (loss, 1.0) to path `dir`
...     logger.record("loss", 1.0)
...     logger.dump(step=1)
...     with logger.accumulate_means("dataset"):  # record the key value pair (`raw/dataset/entropy", 5.0) to path
...         logger.record("entropy", 5.0)
...         logger.dump(step=100)
...     logger.record("entropy", 6.0)
...     logger.dump(step=200)
...     # record the key value pair (`mean/dataset/entropy", 5.5) to path
...     logger.dump(step=1)
...     with logger.add_accumulate_prefix("foo"), logger.accumulate_means("bar"):  # record the key value pair ("raw/foo/bar/biz", 42.0) to path
...         logger.dump(step=2000)
```
... logger.record("biz", 42.0)
... logger.dump(step=2000)
... # record the key value pair `("mean/foo/bar/biz", 42.0)` to path
... # `dir` at step 1.
... logger.dump(step=1)
... with open(os.path.join(dir, 'log.txt')) as f:
...     print(f.read())

<table>
<thead>
<tr>
<th>loss</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>mean/</td>
<td></td>
</tr>
<tr>
<td>dataset/entropy</td>
<td>5.5</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>mean/</td>
<td></td>
</tr>
<tr>
<td>foo/bar/biz</td>
<td>42</td>
</tr>
</tbody>
</table>

__init__(default_logger, format_strs=('stdout', 'log', 'csv'))

Builds HierarchicalLogger.

Parameters

- **default_logger** (Logger) – The default logger when not in an accumulate_means context. Also the logger to which mean values are written to after exiting from a context.

- **format_strs** (Sequence[str]) – A list of output format strings that should be used by every Logger initialized by this class during an AccumulatingMeans context. For details on available output formats see stable_baselines3.logger.make_output_format.

accumulate_means(name)

Temporarily modifies this HierarchicalLogger to accumulate means values.

Within this context manager, self.record(key, value) writes the “raw” values in f"{self.default_logger.log_dir}/[{accumulate_prefix}/]{name}" under the key "raw/ [{accumulate_prefix}/]{name}/[{key_prefix}/]{key}". where accumulate_prefix is the concatenation of all prefixes added by add_accumulate_prefix and key_prefix is the concatenation of all prefixes added by add_key_prefix, if any. At the same time, any call to self.record will also accumulate mean values on the default logger by calling:

```python
self.default_logger.record_mean(
    f"mean/[{accumulate_prefix}/]{name}/[{key_prefix}/]{key}",
    value,
)
```

Multiple prefixes may be active at once. In this case the prefix is simply the concatenation of each of the active prefixes in the order they were created e.g. if the active prefixes are ['foo', 'bar'] then the prefix is 'foo/bar'.

After the context exits, calling self.dump() will write the means of all the “raw” values accumulated during this context to self.default_logger under keys of the form mean/{prefix}/name/{key}.

Note that the behavior of other logging methods, log and record_mean are unmodified and will go straight to the default logger.
Parameters

name (str) – A string key which determines the folder where raw data is written and temporary logging prefixes for raw and mean data. Entering an `accumulate_means` context in the future with the same `subdir` will safely append to logs written in this folder rather than overwrite.

Yields

None when the context is entered.

Raises

`RuntimeError` – If this context is entered into while already in an `accumulate_means` context.

Return type

Generator[None, None, None]

add_accumulate_prefix(prefix)

Add a prefix to the subdirectory used to accumulate means.

This prefix only applies when a `accumulate_means` context is active. If there are multiple active prefixes, then they are concatenated.

Parameters

prefix (str) – The prefix to add to the named sub.

Yields

None when the context manager is entered

Raises

`RuntimeError` – if accumulate means context is already active.

Return type

Generator[None, None, None]

add_key_prefix(prefix)

Add a prefix to the keys logged during an accumulate_means context.

This prefix only applies when a `accumulate_means` context is active. If there are multiple active prefixes, then they are concatenated.

Parameters

prefix (str) – The prefix to add to the keys.

Yields

None when the context manager is entered

Raises

`RuntimeError` – if accumulate means context is already active.

Return type

Generator[None, None, None]

close()

closes the file

current_logger: Optional[Logger]

default_logger: Logger

dump(step=0)

Write all of the diagnostics from the current iteration
format_strs: Sequence[str]

get_accumulate_prefixes()

    Return type
    str

get_dir()

    Get directory that log files are being written to. will be None if there is no output directory (i.e., if you
didn’t call start)

    Return type
    str

    Returns
    the logging directory

log(*args, **kwargs)

    Write the sequence of args, with no separators, to the console and output files (if you’ve configured an
output file).

    level: int. (see logger.py docs) If the global logger level is higher than
the level argument here, don’t print to stdout.

Parameters

    • args – log the arguments
    • level – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

record(key, val, exclude=None)

    Log a value of some diagnostic Call this once for each diagnostic quantity, each iteration If called many
times, last value will be used.

Parameters

    • key – save to log this key
    • value – save to log this value
    • exclude – outputs to be excluded

record_mean(key, val, exclude=None)

    The same as record(), but if called many times, values averaged.

Parameters

    • key – save to log this key
    • value – save to log this value
    • exclude – outputs to be excluded

set_level(level)

    Set logging threshold on current logger.

Parameters

    • level (int) – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40,
DISABLED=50)

    Return type
    None
class imitation.util.logger.WandbOutputFormat
    Bases: KVWriter
    A stable-baseline logger that writes to wandb.
    Users need to call wandb.init() before initializing WandbOutputFormat.

__init__()
    Initializes an instance of WandbOutputFormat.

    Raises
    ModuleNotFoundError – wandb is not installed.

close()
    Close owned resources

    Return type
    None

write(key_values, key_excluded, step=0)
    Write a dictionary to file

    Parameters
    • key_values (Dict[str, Any]) –
    • key_excluded (Dict[str, Union[str, Tuple[str, ...]]]) –
    • step (int) –

    Return type
    None

imitation.util.logger.configure(folder=None, format_strs=None)
    Configure Stable Baselines logger to be accumulate_means()-compatible.
    After this function is called, stable_baselines3.logger.{configure,reset}() are replaced with stubs that raise Run-timeError.

    Parameters
    • folder (Union[str, bytes, PathLike, None]) – Argument from stable_baselines3.logger.configure.
    • format_strs (Optional[Sequence[str]]) – An list of output format strings. For details on available output formats see stable_baselines3.logger.make_output_format.

    Return type
    HierarchicalLogger

    Returns
    The configured HierarchicalLogger instance.

imitation.util.logger.make_output_format(_format, log_dir, log_suffix=", max_length=50)
    Returns a logger for the requested format.

    Parameters
    • _format (str) – the requested format to log to (‘stdout’, ‘log’, ‘json’ or ‘csv’ or ‘tensorboard’).
    • log_dir (str) – the logging directory.
    • log_suffix (str) – the suffix for the log file.
• **`max_length`** (int) – the maximum length beyond which the keys get truncated.

**Return type**

`KVWriter`

**Returns**

the logger.

### imitation.util.networks

Helper methods to build and run neural networks.

#### Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>build_cnn</code></td>
<td>Constructs a Torch CNN.</td>
</tr>
<tr>
<td><code>build_mlp</code></td>
<td>Constructs a Torch MLP.</td>
</tr>
<tr>
<td><code>training_mode</code></td>
<td>Temporarily switch module <code>m</code> to specified training mode.</td>
</tr>
</tbody>
</table>

#### Classes

```python
BaseNorm(num_features[, eps])
```

Base class for layers that try to normalize the input to mean 0 and variance 1. Similar to BatchNorm, LayerNorm, etc. but whereas they only use statistics from the current batch at train time, we use statistics from all batches.

**Parameters**

- **num_features** (int) – Number of features; the length of the non-batch dimension.
- **eps** (float) – Small constant for numerical stability. Inputs are rescaled by $1 / \sqrt{\text{estimated\_variance} + \text{eps}}$.

```python
class imitation.util.networks.BaseNorm(num_features, eps=1e-05)
```

Bases: Module, ABC

Base class for layers that try to normalize the input to mean 0 and variance 1.

Similar to BatchNorm, LayerNorm, etc. but whereas they only use statistics from the current batch at train time, we use statistics from all batches.

```python
__init__(num_features, eps=1e-05)
```

Builds RunningNorm.

**Parameters**

- **num_features** (int) – Number of features; the length of the non-batch dimension.
- **eps** (float) – Small constant for numerical stability. Inputs are rescaled by $1 / \sqrt{\text{estimated\_variance} + \text{eps}}$. 

```python
count: Tensor
```
forward(x)

Updates statistics if in training mode. Returns normalized x.

Return type
Tensor

reset_running_stats()

Resets running stats to defaults, yielding the identity transformation.

Return type
None

running_mean: Tensor
running_var: Tensor

abstract update_stats(batch)

Update self.running_mean, self.running_var and self.count.

Return type
None

class imitation.util.networks.EMANorm(num_features, decay=0.99, eps=1e-05)

Bases: BaseNorm

Similar to RunningNorm but uses an exponential weighting.

__init__(num_features, decay=0.99, eps=1e-05)

Builds EMARunningNorm.

Parameters

• num_features (int) – Number of features; the length of the non-batch dim.
• decay (float) – how quickly the weight on past samples decays over time.
• eps (float) – small constant for numerical stability.

Raises
ValueError – if decay is out of range.

inv_learning_rate: Tensor
num_batches: IntTensor

reset_running_stats()

Reset the running stats of the normalization layer.

update_stats(batch)

Update self.running_mean and self.running_var in batch mode.

Reference Algorithm 3 from: https://github.com/HumanCompatibleAI/imitation/files/9456540/Incremental_batch_EMA_and_EMV.pdf

Parameters

• batch (Tensor) – A batch of data to use to update the running mean and variance.

Return type
None
class imitation.util.networks.RunningNorm(num_features, eps=1e-05)
Bases: BaseNorm

Normalizes input to mean 0 and standard deviation 1 using a running average.
Similar to BatchNorm, LayerNorm, etc. but whereas they only use statistics from the current batch at train time, we use statistics from all batches.
This should closely replicate the common practice in RL of normalizing environment observations, such as using VecNormalize in Stable Baselines.

count: Tensor
running_mean: Tensor
running_var: Tensor
training: bool

update_stats(batch)
Update self.running_mean, self.running_var and self.count.

Parameters
batch (Tensor) – A batch of data to use to update the running mean and variance.

Return type
None

class imitation.util.networks.SqueezeLayer
Bases: Module

Torch module that squeezes a B*1 tensor down into a size-B vector.

forward(x)
Defines the computation performed at every call.
Should be overridden by all subclasses.

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

training: bool

imitation.util.networks.build_cnn(in_channels, hid_channels, out_size=1, name=None, activation=<class 'torch.nn.modules.activation.ReLU'>, kernel_size=3, stride=1, padding='same', dropout_prob=0.0, squeeze_output=False)

Constructs a Torch CNN.

Parameters

- in_channels (int) – number of channels of individual inputs; input to the CNN will have shape (batch_size, in_size, in_height, in_width).
- hid_channels (Iterable[int]) – number of channels of hidden layers. If this is an empty iterable, then we build a linear function approximator.
- out_size (int) – size of output vector.
• **name** (Optional[str]) – Name to use as a prefix for the layers ID.

• **activation** (Type[Module]) – activation to apply after hidden layers.

• **kernel_size** (int) – size of convolutional kernels.

• **stride** (int) – stride of convolutional kernels.

• **padding** (Union[int, str]) – padding of convolutional kernels.

• **dropout_prob** (float) – Dropout probability to use after each hidden layer. If 0, no dropout layers are added to the network.

• **squeeze_output** (bool) – if out_size=1, then squeeze_input=True ensures that CNN output is of size \((B,)\) instead of \((B,1)\).

**Returns**

a CNN mapping from inputs of size \((\text{batch}_\text{size}, \text{in}_\text{size}, \text{in}_\text{height}, \text{in}_\text{width})\) to \((\text{batch}_\text{size}, \text{out}_\text{size})\), unless \(\text{out}_\text{size}=1\) and \(\text{squeeze}_\text{output}=\text{True}\), in which case the output is of size \((\text{batch}_\text{size}, \text{out}_\text{size})\).

**Return type**

n.Module

**Raises**

ValueError – if squeeze_output was supplied with \(\text{out}_\text{size}!=1\).

**imitation.util.networks.build_mlp(in_size, hid_sizes, out_size=1, name=None, activation=<class 'torch.nn.modules.activation.ReLU'>, dropout_prob=0.0, squeeze_output=False, flatten_input=False, normalize_input_layer=None)**

Constructs a Torch MLP.

**Parameters**

• **in_size** (int) – size of individual input vectors; input to the MLP will be of shape \((\text{batch}_\text{size}, \text{in}_\text{size})\).

• **hid_sizes** (Iterable[int]) – sizes of hidden layers. If this is an empty iterable, then we build a linear function approximator.

• **out_size** (int) – size of output vector.

• **name** (Optional[str]) – Name to use as a prefix for the layers ID.

• **activation** (Type[Module]) – activation to apply after hidden layers.

• **dropout_prob** (float) – Dropout probability to use after each hidden layer. If 0, no dropout layers are added to the network.

• **squeeze_output** (bool) – if out_size=1, then squeeze_input=True ensures that MLP output is of size \((B,)\) instead of \((B,1)\).

• **flatten_input** (bool) – should input be flattened along axes 1, 2, 3, ...? Useful if you want to, e.g., process small images inputs with an MLP.

• **normalize_input_layer** (Optional[Type[Module]]) – if specified, module to use to normalize inputs; e.g. `nn.BatchNorm` or `RunningNorm`.

**Returns**

an MLP mapping from inputs of size \((\text{batch}_\text{size}, \text{in}_\text{size})\) to \((\text{batch}_\text{size}, \text{out}_\text{size})\), unless \(\text{out}_\text{size}=1\) and \(\text{squeeze}_\text{output}=\text{True}\), in which case the output is of size \((\text{batch}_\text{size}, \text{out}_\text{size})\).
Return type
n.Module

Raises
ValueError – if squeeze_output was supplied with out_size!=1.

imitation.util.networks.evaluating(m: Module, *, mode: bool = False)
Temporarily switch module m to specified training mode.

Parameters
• m – The module to switch the mode of.
• mode – whether to set training mode (True) or evaluation (False).

Yields
The module m.

imitation.util.networks.training(m: Module, *, mode: bool = True)
Temporarily switch module m to specified training mode.

Parameters
• m – The module to switch the mode of.
• mode – whether to set training mode (True) or evaluation (False).

Yields
The module m.

imitation.util.networks.training_mode(m, mode=False)
Temporarily switch module m to specified training mode.

Parameters
• m (Module) – The module to switch the mode of.
• mode (bool) – whether to set training mode (True) or evaluation (False).

Yields
The module m.

imitation.util.registry
Registry mapping IDs to objects, such as environments or policy loaders.

Module Attributes

| LoaderFn | The type stored in Registry is commonly an instance of LoaderFn. |
# Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

- `build_loader_fn_require_env(fn, **kwargs)` Converts a factory taking an environment into a LoaderFn.
- `build_loader_fn_require_space(fn, **kwargs)` Converts a factory taking observation and action space into a LoaderFn.
- `load_attr(name)` Load an attribute in format path.to.module:attribute.

## Classes

**Registry**

A registry mapping IDs to type T objects, with support for lazy loading.

```python
 imitation.util.registry.Registry

    The type stored in Registry is commonly an instance of LoaderFn.
    alias of Callable[[...], T]

class imitation.util.registry.Registry
    Bases: Generic[T]

    A registry mapping IDs to type T objects, with support for lazy loading.
    The registry allows for insertion and retrieval. Modification of existing elements is not allowed.
    If the registered item is a string, it is assumed to be a path to an attribute in the form path.to.module:attribute. In this case, the module is loaded only if and when the registered item is retrieved.
    This is helpful both to reduce overhead from importing unused modules, and when some modules may have additional dependencies that are not installed in all deployments.
    Note: This is a similar idea to gym.EnvRegistry.

    __init__()
        Builds empty Registry.

    get(key)
        Return type
        TypeVar(T)

    keys()
        Return type
        Iterable[str]

    register(key, *, value=None, indirect=None)
```

**imitation.util.registry.build_loader_fn_require_env(fn, **kwargs)**

Converts a factory taking an environment into a LoaderFn.

```python
    Return type
    Callable[[... , TypeVar(T)]
```
imitation.util.registry.build_loader_fn_require_space(fn, **kwargs)

Converts a factory taking observation and action space into a LoaderFn.

**Return type**

Callable[..., TypeVar(T)]

imitation.util.registry.load_attr(name)

Load an attribute in format path.to.module:attribute.

**imitation.util.sacred**

Helper methods for the sacred experimental configuration and logging framework.

**Functions**

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_sacred_symlink(log_dir, run)</td>
<td>Constructs a symlink &quot;.{log_dir}/sacred&quot; \rightarrow &quot;${SA-CRED_PATH}&quot;.</td>
</tr>
<tr>
<td>dict_get_nested(d, nested_key, *, sep, default)</td>
<td>rtype Any</td>
</tr>
<tr>
<td>dir_contains_sacred_jsons(dir_path)</td>
<td>rtype bool</td>
</tr>
<tr>
<td>filter_subdirs(root_dir[, filter_fn, nested_ok])</td>
<td>Walks through a directory tree, returning paths to filtered subdirectories.</td>
</tr>
<tr>
<td>get_sacred_dir_from_run(run)</td>
<td>Returns path to the sacred directory, or None if not found.</td>
</tr>
</tbody>
</table>

**Classes**

```python
class SacredDicts(sacred_dir: Path, config: dict, run: dict)
```

Bases: tuple

Each dict `foo` is loaded from `f"{sacred_dir}/foo.json"`.

- config: dict
- classmethod load_from_dir(sacred_dir)
- run: dict
- sacred_dir: Path
imitation.util.sacred.build_sacred_symlink(
\( \text{log\_dir, run} \))

Constructs a symlink “\( \text{log\_dir}/sacred \) => \( \$\{\text{SACRED\_PATH}\} \).

Return type
None

imitation.util.sacred.dict_get_nested(d, nested\_key, *, sep=’,’, default=None)

Return type
Any

imitation.util.sacred.dir_contains_sacred_jsons(dir\_path)

Return type
bool

imitation.util.sacred.filter_subdirs(root\_dir, filter\_fn=<function dir\_contains\_sacred\_jsons>, *, nested-ok=False)

Walks through a directory tree, returning paths to filtered subdirectories.

Does not follow symlinks.

Parameters

- root\_dir (Path) – The start of the directory tree walk.
- filter\_fn (Callable[[Path], bool]) – A function with takes a directory path and returns True if we should include the directory path in this function’s return value.
- nested-ok (bool) – Allow returning “nested” directories, i.e. a return value where some elements are subdirectories of other elements.

Return type
Sequence[Path]

Returns
A list of all subdirectory paths where filter\_fn(path) == True.

Raises
ValueError – If nested-ok is False and one of the filtered directory paths is a subdirectory of another.

imitation.util.sacred.get_sacred_dir_from_run(run)

Returns path to the sacred directory, or None if not found.

Return type
Optional[Path]

imitation.util.util

Miscellaneous utility methods.
Functions

```python
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

```python
docstring_parameter(*args, **kwargs)
Treats the docstring as a format string, substituting in the arguments.

endless_iter(iterable)
Generator that endlessly yields elements from iterable.

get_first_iter_element(iterable)
Get first element of an iterable and a new fresh iterable.

make_seeds()
Generate n random seeds from a random state.

make_unique_timestamp()
Timestamp, with random uuid added to avoid collisions.

make_vec_env(env_name, *, rng[, n_envs, ...])
Makes a vectorized environment.

oric(x)
Optimal rounding under integer constraints.

safe_to_numpy()
Convert torch tensor to numpy.

safe_to_tensor(array, **kwargs)
Converts a NumPy array to a PyTorch tensor.

tensor_iter_norm(tensor_iter[, ord])
Compute the norm of a big vector that is produced one tensor chunk at a time.
```

**imitation.util.util.docstring_parameter(*args, **kwargs)**

Treats the docstring as a format string, substituting in the arguments.

**imitation.util.util.endless_iter(iterable)**

Generator that endlessly yields elements from iterable.

```python
>>> x = range(2)
>>> it = endless_iter(x)
>>> next(it)
0
>>> next(it)
1
>>> next(it)
0
```

**Parameters**

- **iterable** (Iterable[TypeVar(T)]) – The non-iterator iterable object to endlessly iterate over.

**Return type**

- Iterator[TypeVar(T)]

**Returns**

- An iterator that repeats the elements in iterable forever.

**Raises**

- ValueError – if iterable is an iterator – that will be exhausted, so cannot be iterated over endlessly.

**imitation.util.util.get_first_iter_element(iterable)**

Get first element of an iterable and a new fresh iterable.

The fresh iterable has the first element added back using itertools.chain. If the iterable is not an iterator, this is equivalent to (next(iter(iterable)), iterable).

**Parameters**

- **iterable** (Iterable[TypeVar(T)]) – The iterable to get the first element of.
Return type
   Tuple[TypeVar(T), Iterable[TypeVar(T)]]

Returns
   A tuple containing the first element of the iterable, and a fresh iterable with all the elements.

Raises
   ValueError – iterable is empty – the first call to it returns no elements.

imitation.util.util.make_seeds(rng: Generator) -> int

Generate n random seeds from a random state.

Parameters
   • rng (Generator) – The random state to use to generate seeds.
   • n (Optional[int]) – The number of seeds to generate.

Return type
   Union[Sequence[int], int]

Returns
   A list of n random seeds.

imitation.util.util.make_unique_timestamp()

Timestamp, with random uuid added to avoid collisions.

Return type
   str

imitation.util.util.make_vec_env(env_name, *, rng, n_envs=8, parallel=False, log_dir=None,
     max_episode_steps=None, post_wrappers=None, env_make_kwargs=None)

Makes a vectorized environment.

Parameters
   • env_name (str) – The Env’s string id in Gym.
   • rng (Generator) – The random state to use to seed the environment.
   • n_envs (int) – The number of duplicate environments.
   • parallel (bool) – If True, uses SubprocVecEnv; otherwise, DummyVecEnv.
   • log_dir (Optional[str]) – If specified, saves Monitor output to this directory.
   • max_episode_steps (Optional[int]) – If specified, wraps each env in a TimeLimit wrapper with this episode length. If not specified and max_episode_steps exists for this env_name in the Gym registry, uses the registry max_episode_steps for every TimeLimit wrapper (this automatic wrapper is the default behavior when calling gym.make). Otherwise the environments are passed into the VecEnv unwrapped.
   • post_wrappers (Optional[Sequence[Callable[[Env, int], Env]]]) – If specified, iteratively wraps each environment with each of the wrappers specified in the sequence. The argument should be a Callable accepting two arguments, the Env to be wrapped and the environment index, and returning the wrapped Env.
   • env_make_kwargs (Optional[Mapping[str, Any]]) – The kwargs passed to spec.make.

Return type
   VecEnv
Returns
A VecEnv initialized with $n_{\text{envs}}$ environments.

```python
imitation.util.util.oric(x)
```
Optimal rounding under integer constraints.

Given a vector of real numbers such that the sum is an integer, returns a vector of rounded integers that preserves
the sum and which minimizes the Lp-norm of the difference between the rounded and original vectors for all $p
\geq 1$. Algorithm from https://arxiv.org/abs/1501.00014. Runs in $O(n \log n)$ time.

**Parameters**

- $x$ (ndarray) – A 1D vector of real numbers that sum to an integer.

**Return type**

ndarray

**Returns**

A 1D vector of rounded integers, preserving the sum.

```python
imitation.util.util.safe_to_numpy(obj: Union[ndarray, Tensor], warn: bool = False) → ndarray
```
Convert torch tensor to numpy.

If the object is already a numpy array, return it as is. If the object is none, returns none.

**Parameters**

- $obj$ (Union[ndarray, Tensor]) – torch tensor object to convert to numpy array
- $warn$ (bool) – if True, warn if the object is not already a numpy array. Useful for warning
  the user of a potential performance hit if a torch tensor is not the expected input type.

**Return type**

Optional[ndarray]

**Returns**

Object converted to numpy array

```python
imitation.util.util.safe_to_tensor(array, **kwargs)
```
Converts a NumPy array to a PyTorch tensor.

The data is copied in the case where the array is non-writable. Unfortunately if you just use `th.as_tensor`
for this, an ugly warning is logged and there’s undefined behavior if you try to write to the tensor.

**Parameters**

- $array$ (Union[ndarray, Tensor]) – The array to convert to a PyTorch tensor.
- $kwargs$ – Additional keyword arguments to pass to `th.as_tensor`.

**Return type**

Tensor

**Returns**

A PyTorch tensor with the same content as $array$.

```python
imitation.util.util.tensor_iter_norm(tensor_iter, ord=2)
```
Compute the norm of a big vector that is produced one tensor chunk at a time.

**Parameters**

- $tensor_{\text{iter}}$ (Iterable[Tensor]) – an iterable that yields tensors.
- $ord$ (Union[int, float]) – order of the p-norm (can be any int or float except 0 and NaN).
**Return type**

Tensor

**Returns**

Norm of the concatenated tensors.

**Raises**

**ValueError** – ord is 0 (unsupported).

---

**imitation.util.video_wrapper**

Wrapper to record rendered video frames from an environment.

**Classes**

---

**VideoWrapper**(env, directory[, single_video])

Creates videos from wrapped environment by calling render after each timestep.

**class** imitation.util.video_wrapper.VideoWrapper(env, directory, single_video=True)

**Bases:** Wrapper

Creates videos from wrapped environment by calling render after each timestep.

**__init__**(env, directory, single_video=True)

Builds a VideoWrapper.

**Parameters**

- **env** (Env) – the wrapped environment.
- **directory** (Path) – the output directory.
- **single_video** (bool) – if True, generates a single video file, with episodes concatenated. If False, a new video file is created for each episode. Usually a single video file is what is desired. However, if one is searching for an interesting episode (perhaps by looking at the metadata), then saving to different files can be useful.

**close()**

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

**Return type**

None

**directory**: Path

**episode_id**: int

**reset()**

Resets the environment to an initial state and returns an initial observation.

Note that this function should not reset the environment’s random number generator(s); random variables in the environment’s state should be sampled independently between multiple calls to reset(). In other words, each call of reset() should yield an environment suitable for a new episode, independent of previous episodes.
Returns
the initial observation.

Return type
observation (object)

single_video: bool

step(action)
Run one timestep of the environment’s dynamics. When end of episode is reached, you are responsible for calling `reset()` to reset this environment’s state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters
action (object) – an action provided by the agent

Returns
agent’s observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type
observation (object)

video_recorder: Optional[VideoRecorder]

2.2 Developer Guide

This guide explains the library structure of imitation. The code is organized such that logically similar files are grouped into a subpackage. We maintain the following subpackages in `src/imitation`:

- **algorithms**: the core implementation of imitation and reward learning algorithms.
- **data**: modules to collect, store and manipulate transitions and trajectories from RL environments.
- **envs**: provides test environments.
- **policies**: provides modules that define policies and methods to manipulate them (e.g., serialization).
- **regularization**: implements a variety of regularization techniques for NN weights.
- **rewards**: modules to build, serialize and preprocess neural network based reward functions.
- **scripts**: command-line scripts for running experiments through Sacred.
- **util**: provides utility functions like logging, configurations, etc.

2.2.1 Algorithms

The `imitation.algorithms.base` module defines the following two classes:

- **BaseImitationAlgorithm**: Base class for all imitation algorithms.
- **DemonstrationAlgorithm**: Base class for all demonstration-based algorithms like BC, IRL, etc. This class subclasses `BaseImitationAlgorithm`.

Demonstration algorithms offer the following methods and properties:

- policy property that returns a policy imitating the demonstration data.
set_demonstrations method that sets the demonstrations data for learning.

All of the algorithms provide the train method for training an agent and/or a reward network.

All the available algorithms are present in algorithms/ with each algorithm in a distinct file. Adversarial algorithms like AIRL and GAIL are present in algorithms/adversarial.

2.2.2 Data

Modules handling environment data.

For example: types for transitions/trajectories; methods to compute rollouts; buffers to store transitions; helpers for these modules.

data.wrapper.BufferingWrapper: Wraps a vectorized environment VecEnv to save the trajectories from all the environments in a buffer.

data.wrapper.RolloutInfoWrapper: Wraps a gym.Env environment to log the original observations and rewards received from the environment. The observations and rewards of the entire episode are logged in the info dictionary with the key "rollout", in the final time step of the episode. This wrapper is useful for saving rollout trajectories, especially in cases where you want to bypass the reward and/or observation overrides from other wrappers. See data.rollout.unwrap_traj for details and scripts/train_rl.py for an example use case.

data.rollout.rollout: Generates rollout by taking in any policy as input along with the environment.

2.2.3 Policies

The imitation.policies subpackage contains the following modules:

- policies.base: defines commonly used policies across the library like FeedForward32Policy, SAC1024Policy, NormalizeFeaturesExtractor, etc.
- policies.exploration_wrapper: defines the ExplorationWrapper class that wraps a policy to create a partially randomized policy useful for exploration.
- policies.replay_buffer_wrapper: defines the ReplayBufferRewardWrapper to wrap a replay buffer that returns transitions with rewards specified by a reward function.
- policies.serialize: defines various functions to save and load serialized policies from the disk or the Hugging Face hub.

2.2.4 Regularization

The imitation.regularization subpackage provides an API for creating neural network regularizers. It provides classes such as regularizers.LpRegularizer and regularizers.WeightDecayRegularizer to regularize the loss function and the weights of a network, respectively. The updaters.IntervalParamScaler class also provides support to scale the lambda hyperparameter of a regularizer up when the ratio of validation to training loss is above an upper bound, and scales it down when the ratio drops below a lower bound.
2.2.5 Rewards

The imitation.rewards subpackage contains code related to building, serializing, and loading reward networks. Some of the classes include:

- rewards.reward_nets.RewardNet: is the base reward network class. Reward networks can take state, action, and the next state as input to predict the reward. The forward method is used while training the network, whereas the predict method is used during evaluation.
- rewards.reward_nets.BasicRewardNet: builds a MLP reward network.
- rewards.reward_wrapper.RewardVecEnvWrapper: This class wraps a VecEnv with a custom RewardFn. The default reward function of the environment is overridden with the passed reward function, and the original rewards are stored in the info_dict with the original_env rew key. This class is used to override the original reward function of an environment with a learned reward function from the reward learning algorithms like preference comparisons.

The imitation.rewards.serialize module contains functions to load serialized reward functions.

2.2.6 Scripts

We use Sacred to provide a command-line interface to run the experiments. The scripts to run the end-to-end experiments are available in scripts/. You can take a look at the following doc links to understand how to use Sacred:

- Experiment Overview: Explains how to create and run experiments. Each script, defined in scripts/, has a corresponding experiment object, defined in scripts/config, with the experiment object and Python source files named after the algorithm(s) supported. For example, the train_rl_ex object is defined in scripts.config.train_rl and its main function is in scripts.train_rl.
- Ingredients: Explains how to use ingredients to avoid code duplication across experiments. The ingredients used in our experiments are defined in scripts/common/:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation.scripts.common.common</td>
<td>Common configuration elements for scripts.</td>
</tr>
<tr>
<td>imitation.scripts.common.demonstrations</td>
<td>Common configuration element for scripts learning from demonstrations.</td>
</tr>
<tr>
<td>imitation.scripts.common.reward</td>
<td>Common configuration elements for reward network training.</td>
</tr>
<tr>
<td>imitation.scripts.common.rl</td>
<td>Common configuration elements for reinforcement learning.</td>
</tr>
<tr>
<td>imitation.scripts.common.train</td>
<td>Common configuration elements for training imitation algorithms.</td>
</tr>
<tr>
<td>imitation.scripts.common.wb</td>
<td>Weights &amp; Biases configuration elements for scripts.</td>
</tr>
</tbody>
</table>

- Configurations: Explains how to use configurations to parametrize runs. The configurations for different algorithms are defined in their file in scripts/. Some of the commonly used configs and ingredients used across algorithms are defined in scripts/common/.
- Command-Line Interface: Explains how to run the experiments through the command-line interface. Also, note the section on how to print configs to verify the configurations used for the run.
- Controlling Randomness: Explains how to control randomness by seeding experiments through Sacred.
2.2.7 Util

`imitation.util.logger.HierarchicalLogger`: A logger that supports contexts for accumulating the mean of values of all the logged keys. The logger internally maintains one separate `stable_baselines3.common.logger.Logger` object for logging the mean values, and one `Logger` object for the raw values for each context. The `accumulate_means` context cannot be called inside an already open `accumulate_means` context. The `imitation.util.logger.configure` function can be used to easily construct a `HierarchicalLogger` object.

`imitation.util.networks`: This module provides some additional neural network layers that can be used for imitation like `RunningNorm` and `EMANorm` that normalize their inputs. The module also provides functions like `build_mlp` and `build_cnn` to quickly build neural networks.

`imitation.util.util`: This module provides miscellaneous util functions like `make_vec_env` to easily construct vectorized environments and `safe_to_tensor` that converts a NumPy array to a PyTorch tensor.

`imitation.util.video_wrapper.VideoWrapper`: A wrapper to record rendered videos from an environment.

2.3 Contributing

2.3.1 Code of Conduct

To ensure that the imitation community remains open and inclusive, we have a few ground rules that we ask contributors to adhere to. This isn’t an exhaustive list of things that you can’t do. Rather, take it in the spirit in which it’s intended — a guide to make it easier to enrich all of us and the technical communities in which we participate.

- **Be friendly and patient.**
- **Be welcoming.** We strive to be a community that welcomes and supports people of all backgrounds and identities. This includes, but is not limited to members of any race, ethnicity, culture, national origin, colour, immigration status, social and economic class, educational level, sex, sexual orientation, gender identity and expression, age, size, family status, political belief, religion, and mental and physical ability.
- **Be considerate.** Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and you should take those consequences into account when making decisions. Remember that we’re a world-wide community, so you might not be communicating in someone else’s primary language.
- **Be respectful.** Not all of us will agree all the time, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. Members of the imitation community should be respectful when dealing with other members as well as with people outside the imitation community.
- **Be careful in the words that you choose.** We are a community of professionals, and we conduct ourselves professionally. Be kind to others. Do not insult or put down other participants. Harassment and other exclusionary behavior aren’t acceptable. This includes, but is not limited to:
  - Violent threats or language directed against another person.
  - Discriminatory jokes and language.
  - Posting sexually explicit or violent material.
  - Posting (or threatening to post) other people’s personally identifying information without their consent ("doxing").
  - Personal insults, especially those using racist or sexist terms.
  - Unwelcome sexual attention.
imitation

– Advocating for, or encouraging, any of the above behavior.
– Repeated harassment of others. In general, if someone asks you to stop, then stop.

• When we disagree, try to understand why. It is important that we resolve disagreements and differing views constructively. Focus on helping to resolve issues and learning from mistakes.

Adapted from the original text courtesy of the Django project, licensed under a Creative Commons Attribution 3.0 License.

2.3.2 Ways to contribute

There are four main ways you can contribute to imitation:

• Reporting bugs
• Suggesting new features
• Contributing to the documentation
• Contributing to the codebase

Please note that by contributing to the project, you are agreeing to license your work under imitation’s MIT license, as per GitHub’s terms of service.

Reporting bugs

This section guides you through submitting a new bug report for imitation. Following the guidelines below helps maintainers and the community understand your report and reproduce the issue.

You can submit a new bug report by creating an issue on GitHub and labeling it as a bug. Before you do so, please make sure that:

• You are using the latest stable version of imitation — to check your version, run pip show imitation,
• You have read the relevant section of the documentation that relates to your issue,
• You have checked existing bug reports to make sure that your issue has not already been reported, and
• You have a minimal, reproducible example of the issue.

When submitting a bug report, please include the following information:

• A clear, concise description of the bug,
• A minimal, reproducible example of the bug, with installation instructions, code, and error message,
• Information on your OS name and version, Python version, and other relevant information (e.g. hardware configuration if using the GPU), and
• Whether the problem arose when upgrading to a certain version of imitation, and if so, what version.
Suggesting new features

This section explains how you can submit a new feature request, including completely new features and minor improvements to existing functionality. Following these guidelines helps maintainers and the community understand your request and intended use cases and find related suggestions.

You can submit a new bug report by creating an issue on GitHub and labeling it as an enhancement. Before you do so, please make sure that:

- You have checked the documentation that relates to your request, as it may be that such feature is already available,
- You have checked existing feature requests to make sure that there is no similar request already under discussion, and
- You have a minimal use case that describes the relevance of the feature.

When you submit the feature request:

- Use a clear and descriptive title for the GitHub issue to easily identify the suggestion.
- Describe the current behavior, and explain what behavior you expected to see instead and why.
- If you want to request an API change, provide examples of how the feature would be used.
- If you want to request a new algorithm implementation, please provide a link to the relevant paper or publication.

Contributing to the documentation

One of the simplest ways to start contributing to imitation is through improving the documentation. Currently, our documentation has some gaps, and we would love to have you help us fill them. You can help by adding missing sections of the API docs, editing existing content to make it more readable, clear and accessible, or contributing new content, such as tutorials and FAQs.

If you have struggled to understand something about our codebase and managed to figure it out in the end, please consider improving the relevant documentation section, or adding a tutorial or a FAQ entry, so that other users can learn from your experience.

Before submitting a pull request, please create an issue with the documentation label so that we can track the gap. You can then reference the issue in your pull request by including the issue number.

Contributing to the codebase

You can contribute to the codebase by proposing solutions to issues or feature suggestions you’ve raised yourself, or selecting an existing issue to work on. Please, make sure to create an issue on GitHub before you start working on a pull request, as explained in Reporting bugs and Suggesting new features.

Once you’re ready to start working on your pull request, please make sure to follow our coding style guidelines:

- PEP8, with line width 88.
- Use the black autoformatter.
- Follow the Google Python Style Guide unless it conflicts with the above. Examples of Google-style docstrings can be found here.

Before you submit, please make sure that:

- Your PR includes unit tests for any new features.
- Your PR includes type annotations, except when it would make the code significantly more complex.
• You have run the unit tests and there are no errors. We use pytest for unit testing: run pytest tests/ to run the test suite.

• You should run pre-commit run to run linting and static type checks. We use pytype for static type analysis.

You may wish to configure this as a Git commit hook:

```
pre-commit install
```

These checks are run on CircleCI and are required to pass before merging. Additionally, we track test coverage by CodeCov and require that code coverage should not decrease. This can be overridden by maintainers in exceptional cases. Files in imitation/{examples,scripts}/ have no coverage requirements.

Thank you for your interest in imitation!

As an open-source project, we welcome contributions from all users, and are always open to any feedback or suggestions. This section of the documentation is intended to help you understand the process of contributing to the project.

To keep the community open and inclusive, we have developed a Code of Conduct. If you are not familiar with our Code of Conduct, take a minute to read it before starting your first contribution.

### 2.4 Release Notes

#### 2.4.1 v0.3.1

*Released on 2022-07-29 - GitHub - PyPI*

#### 2.4.2 v0.3.0: Major improvements

*Released on 2022-07-26 - GitHub - PyPI*

#### 2.4.3 v0.2.0: First PyTorch release

*Released on 2020-10-23 - GitHub - PyPI*

#### 2.4.4 v0.1.1: Final TF1 release

*Released on 2020-09-01 - GitHub - PyPI*

#### 2.4.5 v0.1.0: Initial release

*Released on 2020-05-09 - GitHub - PyPI*
2.5 License

This license is also available on the project repository.

MIT License

Copyright (c) 2019-2022 Center for Human-Compatible AI and Google LLC

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2.6 Index

- genindex
- modindex
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>imitation</td>
<td>91</td>
</tr>
<tr>
<td>imitation.algorithms</td>
<td>91</td>
</tr>
<tr>
<td>imitation.algorithms.adversarial</td>
<td>92</td>
</tr>
<tr>
<td>imitation.algorithms.adversarial.airl</td>
<td>92</td>
</tr>
<tr>
<td>imitation.algorithms.adversarial.common</td>
<td>94</td>
</tr>
<tr>
<td>imitation.algorithms.adversarial.gail</td>
<td>98</td>
</tr>
<tr>
<td>imitation.algorithms.base</td>
<td>101</td>
</tr>
<tr>
<td>imitation.algorithms.bc</td>
<td>103</td>
</tr>
<tr>
<td>imitation.algorithms.dagger</td>
<td>107</td>
</tr>
<tr>
<td>imitation.algorithms.density</td>
<td>114</td>
</tr>
<tr>
<td>imitation.algorithms.mce_irl</td>
<td>116</td>
</tr>
<tr>
<td>imitation.algorithms.preference_comparisons</td>
<td>121</td>
</tr>
<tr>
<td>imitation.data</td>
<td>135</td>
</tr>
<tr>
<td>imitation.data.buffer</td>
<td>135</td>
</tr>
<tr>
<td>imitation.data.rollout</td>
<td>140</td>
</tr>
<tr>
<td>imitation.data.types</td>
<td>146</td>
</tr>
<tr>
<td>imitation.data.wrappers</td>
<td>150</td>
</tr>
<tr>
<td>imitation.policies</td>
<td>152</td>
</tr>
<tr>
<td>imitation.policies.base</td>
<td>153</td>
</tr>
<tr>
<td>imitation.policies.exploration_wrapper</td>
<td>155</td>
</tr>
<tr>
<td>imitation.policies.replay_buffer_wrapper</td>
<td>155</td>
</tr>
<tr>
<td>imitation.policies.serialize</td>
<td>157</td>
</tr>
<tr>
<td>imitation.regularization</td>
<td>159</td>
</tr>
<tr>
<td>imitation.regularization.regularizers</td>
<td>159</td>
</tr>
<tr>
<td>imitation.regularization.update</td>
<td>162</td>
</tr>
<tr>
<td>imitation.rewards</td>
<td>163</td>
</tr>
<tr>
<td>imitation.rewards.reward_function</td>
<td>164</td>
</tr>
<tr>
<td>imitation.rewards.reward_nets</td>
<td>164</td>
</tr>
<tr>
<td>imitation.rewards.reward_wrapper</td>
<td>177</td>
</tr>
<tr>
<td>imitation.rewards.serialize</td>
<td>178</td>
</tr>
<tr>
<td>imitation.scripts</td>
<td>179</td>
</tr>
<tr>
<td>imitation.scripts.analyze</td>
<td>180</td>
</tr>
<tr>
<td>imitation.scripts.common</td>
<td>181</td>
</tr>
<tr>
<td>imitation.scripts.common.common</td>
<td>181</td>
</tr>
<tr>
<td>imitation.scripts.common.demonstrations</td>
<td>183</td>
</tr>
<tr>
<td>imitation.scripts.common.expert</td>
<td>184</td>
</tr>
<tr>
<td>imitation.scripts.common.reward</td>
<td>184</td>
</tr>
<tr>
<td>imitation.scripts.common.rl</td>
<td>185</td>
</tr>
<tr>
<td>imitation.scripts.common.train</td>
<td>186</td>
</tr>
<tr>
<td>imitation.scripts.common.wb</td>
<td>187</td>
</tr>
<tr>
<td>imitation.scripts.config</td>
<td>187</td>
</tr>
<tr>
<td>imitation.scripts.config.analyze</td>
<td>188</td>
</tr>
<tr>
<td>imitation.scripts.config.eval_policy</td>
<td>188</td>
</tr>
<tr>
<td>imitation.scripts.config.train_adversarial</td>
<td>188</td>
</tr>
<tr>
<td>imitation.scripts.config.train_imitation</td>
<td>188</td>
</tr>
<tr>
<td>imitation.scripts.config.train_preference_comparisons</td>
<td>188</td>
</tr>
<tr>
<td>imitation.scripts.config.train_imitation</td>
<td>188</td>
</tr>
<tr>
<td>imitation.scripts.config.train_irl</td>
<td>188</td>
</tr>
<tr>
<td>imitation.scripts.convert_trajs</td>
<td>189</td>
</tr>
<tr>
<td>imitation.scripts.eval_policy</td>
<td>189</td>
</tr>
<tr>
<td>imitation.scripts.train_adversarial</td>
<td>191</td>
</tr>
<tr>
<td>imitation.scripts.train_imitation</td>
<td>192</td>
</tr>
<tr>
<td>imitation.scripts.train_preference_comparisons</td>
<td>194</td>
</tr>
<tr>
<td>imitation.scripts.train_irl</td>
<td>197</td>
</tr>
<tr>
<td>imitation.testing</td>
<td>198</td>
</tr>
<tr>
<td>imitation.testing.expertTrajectories</td>
<td>198</td>
</tr>
<tr>
<td>imitation.testing.reward_improvement</td>
<td>200</td>
</tr>
<tr>
<td>imitation.testing.reward_nets</td>
<td>201</td>
</tr>
<tr>
<td>imitation.util</td>
<td>202</td>
</tr>
<tr>
<td>imitation.util.logger</td>
<td>203</td>
</tr>
<tr>
<td>imitation.util.networks</td>
<td>208</td>
</tr>
<tr>
<td>imitation.util.registry</td>
<td>212</td>
</tr>
<tr>
<td>imitation.util.sacred</td>
<td>214</td>
</tr>
<tr>
<td>imitation.util.util</td>
<td>215</td>
</tr>
<tr>
<td>imitation.util.video_wrapper</td>
<td>219</td>
</tr>
</tbody>
</table>
## Symbols

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__()</code> (imitation.algorithms.adversarial.airl.AIRL)</td>
<td>92</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.adversarial.common.AdversarialTrainer)</td>
<td>95</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.adversarial.gail.GAIL)</td>
<td>99</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.adversarial.gail.RewardNetFromDiscriminatorLogit)</td>
<td>100</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.base.BaseImitationAlgorithm)</td>
<td>101</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.base.DemonstrationAlgorithm)</td>
<td>101</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.bc.BC)</td>
<td>103</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.bc.BCLogger)</td>
<td>105</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.bc.BCTrainingMetrics)</td>
<td>106</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.bc.BatchIteratorWithEpochEndCallback)</td>
<td>106</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.bc.BehaviorCloningLossCalculator)</td>
<td>106</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.dagger.DAggerTrainer)</td>
<td>109</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.dagger.InteractiveTrajectoryCollector)</td>
<td>110</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.dagger.LinearBetaSchedule)</td>
<td>112</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.dagger.SimpleDAggerTrainer)</td>
<td>112</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.density.DensityAlgorithm)</td>
<td>114</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.mce_irl.MCEIRL)</td>
<td>117</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.BasicRewardTrainer)</td>
<td>124</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.CrossEntropyRewardTrainer)</td>
<td>125</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.EnsembleTrainer)</td>
<td>126</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.Fragmenter)</td>
<td>126</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.PreferenceComparer)</td>
<td>127</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.PreferenceComparator)</td>
<td>127</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.PreferenceDataset)</td>
<td>129</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.PreferenceGatherer)</td>
<td>130</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.PreferenceModel)</td>
<td>130</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.RandomFragmenter)</td>
<td>132</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.RewardTrainer)</td>
<td>132</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.algorithms.preference_comparisons.SyntheticGatherer)</td>
<td>133</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.buffer.Buffer)</td>
<td>136</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.buffer.ReplayBuffer)</td>
<td>138</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.rollout.TrajectoryAccumulator)</td>
<td>141</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.types.Trajectory)</td>
<td>146</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.types.TrajectoryWithRew)</td>
<td>147</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.types.Transitions)</td>
<td>147</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.types.TransitionsMinimal)</td>
<td>147</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.types.TransitionsWithRew)</td>
<td>147</td>
</tr>
<tr>
<td><code>__init__()</code> (imitation.data.wrappers.BufferingWrapper)</td>
<td>231</td>
</tr>
</tbody>
</table>
IMITATION

allow_variable_horizon (imitation.algorithms.adversarial.gail.GAIL attribute), 99
allow_variable_horizon (imitation.algorithms.base.BaseImitationAlgorithm attribute), 101
allow_variable_horizon (imitation.algorithms.base.DemonstrationAlgorithm attribute), 102
allow_variable_horizon (imitation.algorithms.bc.BC attribute), 104
allow_variable_horizon (imitation.algorithms.dagger.SimpleDAggerTrainer attribute), 112
allow_variable_horizon (imitation.algorithms.preference_comparisons.PreferenceComparisons attribute), 129
analyze_imitation() (in module imitation.scripts.analyze), 180

B
base (imitation.rewards.reward_nets.RewardNetWrapper property), 175
BaseImitationAlgorithm (class in imitation.algorithms.base), 101
BaseNorm (class in imitation.algorithms.base), 208
BasicPotentialCNN (class in imitation.algorithms.base), 166
BasicPotentialMLP (class in imitation.algorithms.base), 166
BasicRewardNet (class in imitation.algorithms.base), 167
BasicRewardTrainer (class in imitation.algorithms.base), 124
BasicShapedRewardNet (class in imitation.algorithms.base), 167
batch_loader (imitation.algorithms.bc.BatchIteratorWithEpochEndCallback attribute), 106
batch_size (imitation.algorithms.dagger.DAggerTrainer property), 109
BatchIteratorWithEpochEndCallback (class in imitation.algorithms.base), 106
BC (class in imitation.algorithms.base), 103
bc() (in module imitation.scripts.train_imitation), 192
BCLogger (class in imitation.algorithms.base), 105
BCTrainingMetrics (class in imitation.algorithms.base), 106
BehaviorCloningLossCalculator (class in imitation.algorithms.base), 106
BetaSchedule (class in imitation.algorithms.dagger), 108
Buffer (class in imitation.data.buffer), 136
BufferingWrapper (imitation.algorithms.density.DensityAlgorithm attribute), 115
BufferingWrapper (class in imitation.data.wrappers), 150
build_cnn() (in module imitation.util.networks), 210
build_loader_fn_require_env() (in module imitation.util.registry), 213
build_loader_fn_require_space() (in module imitation.util.registry), 214
build_mlp() (in module imitation.util.networks), 211
build_sacred_symlink() (in module imitation.util.sacred), 214

C
capacity (imitation.data.buffer.Buffer attribute), 136
capacity (imitation.data.buffer.ReplayBuffer attribute), 138
close() (imitation.util.logger.HierarchicalLogger method), 205
close() (imitation.util.logger.WandbOutputFormat method), 207
close() (imitation.util.video_wrapper.VideoWrapper method), 219
cnn_transpose() (in module imitation.util.networks), 176
CnnRewardNet (class in imitation.algorithms.preference_comparisons), 125
cross_train_stats() (in module imitation.algorithms.adversarial.common), 98
config (imitation.util.sacred.SacredDicts attribute), 214
config_hook() (in module imitation.util.sacred.expert), 184
config_hook() (in module imitation.util.sacred.reward), 184
config_hook() (in module imitation.util.sacred.rl), 185
configure() (imitation.util.logger), 207
count() (in module imitation.util.networks.BaseNorm attribute), 208
current_logger (imitation.util.logger.HierarchicalLogger attribute), 205

Index 233
D

dagger() (in module imitation.scripts.train_imitation), 193
DAggerTrainer (class in imitation.algorithms.dagger), 108
dataclass_quick_asdict() (in module imitation.data.types), 148
default_logger (imitation.util.logger.HierarchicalLogger attribute), 205
DEFAULT_N_EPOCHS (imitation.algorithms.dagger.DAggerTrainer attribute), 108
demo_state_om (imitation.algorithms.mce_irl.MCEIRL attribute), 118
DemonstrationAlgorithm (class in imitation.algorithms.base), 101
density_type (imitation.algorithms.density.DensityAlgorithm attribute), 115
DensityAlgorithm (class in imitation.algorithms.density), 114
density_type (class in imitation.algorithms.density), 116
device (imitation.rewards.reward_nets.RewardNet property), 173
device (imitation.rewards.reward_nets.RewardNetWrapper property), 175
dict_get_nested() (in module imitation.util.sacred), 215
dir_contains_sacred_jsons() (in module imitation.util.sacred), 215
directory (imitation.util.video_wrapper.VideoWrapper attribute), 219
discounted_sum() (in module imitation.data.rollout), 142
docstring_parameter() (in module imitation.util.sacred), 216
dones (imitation.data.types.Transitions attribute), 147
dtype (imitation.rewards.reward_nets.RewardNet property), 173
dtype (imitation.rewards.reward_nets.RewardNetWrapper property), 175
dump() (imitation.util.logger.HierarchicalLogger method), 205

E

EMANorm (class in imitation.util.networks), 209
endless_iter() (in module imitation.util.util), 216
EnsembleTrainer (class in imitation.algorithms.preference_comparisons), 126
ent_loss (imitation.algorithms.bc.BCTrainingMetrics attribute), 106
ent_weight (imitation.algorithms.bc.BehaviorCloningLossCalculator attribute), 106
entropy (imitation.algorithms.bc.BCTrainingMetrics attribute), 106
enumerate_batches() (in module imitation.algorithms.bc), 107
envs (imitation.rewards.reward_wrapper.RewardVecEnvWrapper property), 177
episode_id (imitation.util.video_wrapper.VideoWrapper attribute), 219
error_on_premature_event (imitation.data.wrappers.BufferingWrapper attribute), 150
eval_policy() (in module imitation.scripts.common.train), 186
eval_policy() (in module imitation.scripts.eval_policy), 190
evaluating() (in module imitation.util.networks), 212
ExplorationWrapper (class in imitation.policies.exploration_wrapper), 155
extend_and_update() (imitation.algorithms.dagger.DAggerTrainer method), 109

F

FeedForward32Policy (class in imitation.policies.base), 153
filter_subdirs() (in module imitation.util.sacred), 215
finish_trajectory() (imitation.data.rollout.TrajectoryAccumulator method), 141
flatten_trajectories() (in module imitation.data.rollout), 142
flatten_trajectories_with_rew() (in module imitation.data.rollout), 142
format_strs (imitation.util.logger.HierarchicalLogger attribute), 205
forward() (imitation.algorithms.adversarial.gail.RewardNetFromDiscriminatorLogit method), 100
forward() (imitation.algorithms.mce_irl.TabularPolicy method), 119
forward() (imitation.algorithms.preference_comparisons.CrossEntropyRewardLoss method), 125
forward() (imitation.algorithms.preference_comparisons.PreferenceModel method), 131
forward() (imitation.algorithms.preference_comparisons.RewardLoss method), 132
forward() (imitation.policies.base.HardCodedPolicy method), 153
forward() (imitation.policies.base.NormalizeFeaturesExtractor method), 154
forward() (imitation.rewards.reward_nets.BasicPotentialCNN method), 166
forward() (imitation.rewards.reward_nets.BasicPotentialMLP method), 167
forward() (imitation.rewards.reward_nets.BasicRewardNet method), 167
get_expert_policy() (in module imitation.scripts.common.expert), 184
forward() (imitation.rewards.reward_nets.CnnRewardNet method), 169
get_expert_trajectories() (in module imitation.scripts.common.demonstrations), 183
forward() (imitation.rewards.reward_nets.PredictProcessedWrapper method), 170
get_first_iter_element() (in module imitation.util.util), 216
forward() (imitation.rewards.reward_nets.RewardEnsemble method), 171
get_num_channels_obs() (imitation.rewards.reward_nets.CnnRewardNet method), 169
forward() (imitation.rewards.reward_nets.RewardNet method), 173
get_sacred_dir_from_run() (in module imitation.util.sacred), 215
forward() (imitation.rewards.reward_nets.ShapedRewardNet method), 176
forward() (imitation.testing.reward_nets.MockRewardNet method), 202
ForwardWrapper (class in imitation.rewards.reward_nets), 169
from_data() (imitation.data.buffer.Buffer class method), 136
HierarchicalLogger (class in imitation.util.logger), 203
forward() (imitation.util.networks.BaseNorm method), 208
hook() (in module imitation.scripts.common.common), 181
forward() (imitation.util.networks.SqueezeLayer method), 210
I
imitation module, 91
imitation.algorithms module, 91
imitation.algorithms.adversarial module, 92
imitation.algorithms.adversarial.airl module, 92
imitation.algorithms.adversarial.common module, 94
imitation.algorithms.adversarial.gail module, 98
imitation.algorithms.base module, 101
imitation.algorithms.bc module, 103
imitation.algorithms.dagger module, 107
imitation.algorithms.density module, 114
imitation.algorithms.mce_irl module, 116
imitation.algorithms.preference_comparisons module, 121
imitation.data module, 135
imitation.data.buffer module, 135
imitation.data.rollout module, 140
imitation.data.types module, 146
imitation.data.wrappers module, 150
is_stationary (imitation.algorithms.density.DensityAlgorithm attribute), 115

K

kernel (imitation.algorithms.density.DensityAlgorithm attribute), 115

kernel_bandwidth (imitation.algorithms.density.DensityAlgorithm attribute), 115

keys() (imitation.util.registry.Registry method), 213

L

l2_loss (imitation.algorithms.bc.BCTrainingMetrics attribute), 106

l2_norm (imitation.algorithms.bc.BCTrainingMetrics attribute), 106

l2_weight (imitation.algorithms.bc.BehaviorCloningLossCalculator attribute), 106

lambda_ (imitation.regularization.regularizers.LossRegularizer attribute), 159

lambda_ (imitation.regularization.regularizers.Regularizer attribute), 161

lambda_ (imitation.regularization.regularizers.WeightDecayRegularizer attribute), 162

lambda_ (imitation.regularization.regularizers.WeightRegularizer attribute), 162

lambda_updater (imitation.regularization.regularizers.LossRegularizer attribute), 159

lambda_updater (imitation.regularization.regularizers.Regularizer attribute), 161

lambda_updater (imitation.regularization.regularizers.WeightDecayRegularizer attribute), 162

lambda_updater (imitation.regularization.regularizers.WeightRegularizer attribute), 162

LambdaUpdater (class in imitation.algorithms.dagger), 111

lazy_generate_expert_trajectories() (in module imitation.testing.expert_trajectories), 199

LinearBetaSchedule (class in imitation.algorithms.dagger), 111

load() (imitation.algorithms.preference_comparisons.PreferenceDataset static method), 129

load() (in module imitation.data.types), 148

load_attr() (in module imitation.util.registry), 214

load_expert_trajs() (in module imitation.scripts.common.demonstrations), 183

load_from_dir() (imitation.util.sacred.SacredDicts class method), 214

load_policy() (in module imitation.policies.serialize), 157

load_reward() (in module imitation.rewards.serialize), 178

load_rl_algo_from_path() (in module imitation.scripts.common.rl), 185

load_stable_baselines_model() (in module imitation.data.types), 158

load_with_rewards() (in module imitation.data.types), 148

load_zero() (in module imitation.rewards.serialize), 179

LoaderFn (in module imitation.util.registry), 213

log() (imitation.util.logger.HierarchicalLogger method), 206

log_batch() (imitation.algorithms.bc.BCLogger method), 106

log_epoch() (imitation.algorithms.bc.BCLogger method), 106

logger (imitation.algorithms.base.BaseImitationAlgorithm property), 101

logger (imitation.algorithms.dagger.DAggerTrainer property), 109

logger (imitation.algorithms.preference_comparisons.AgentTrainer property), 126

logger (imitation.algorithms.preference_comparisons.RewardTrainer property), 133

logger (imitation.algorithms.preference_comparisons.TrajectoryGenerator property), 134

logger (imitation.algorithms.regularization.regularizers.LossRegularizer attribute), 159

logger (imitation.algorithms.regularization.regularizers.Regularizer attribute), 161

logger (imitation.algorithms.regularization.regularizers.WeightDecayRegularizer attribute), 162

logger (imitation.algorithms.regularization.regularizers.WeightRegularizer attribute), 162

logits_expert_is_high() (imitation.algorithms.adversarial.airl.AIRL method), 93

logits_expert_is_high() (imitation.algorithms.adversarial.gail.GAIL method), 96

logits_expert_is_high() (imitation.algorithms.adversarial.common.AdversarialTrainer method), 99

loss (imitation.algorithms.bc.BCTrainingMetrics attribute), 106

loss (imitation.algorithms.preference_comparisons.LossAndMetrics attribute), 127

LossAndMetrics (class in imitation.algorithms.preference_comparisons),
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imit. scripts.common.demonstrations</td>
<td>183</td>
</tr>
<tr>
<td>imit. scripts.common.expert</td>
<td>184</td>
</tr>
<tr>
<td>imit. scripts.common.reward</td>
<td>184</td>
</tr>
<tr>
<td>imit. scripts.common.rl</td>
<td>185</td>
</tr>
<tr>
<td>imit. scripts.common.train</td>
<td>186</td>
</tr>
<tr>
<td>imit. scripts.common.wb</td>
<td>187</td>
</tr>
<tr>
<td>imit. scripts.config</td>
<td>187</td>
</tr>
<tr>
<td>imit. scripts.config.analyze</td>
<td>188</td>
</tr>
<tr>
<td>imit. scripts.config.eval_policy</td>
<td>188</td>
</tr>
<tr>
<td>imit. scripts.config.train_adversarial</td>
<td>188</td>
</tr>
<tr>
<td>imit. scripts.config.train_imitation</td>
<td>188</td>
</tr>
<tr>
<td>imit. scripts.config.train_preference_comparisons</td>
<td>188</td>
</tr>
<tr>
<td>imit. scripts.train_imitation</td>
<td>189</td>
</tr>
<tr>
<td>imit. scripts.train_preference</td>
<td>189</td>
</tr>
<tr>
<td>imit. scripts.train_rl</td>
<td>189</td>
</tr>
<tr>
<td>imit. scripts.convert_trajs</td>
<td>189</td>
</tr>
<tr>
<td>imit. scripts.eval_policy</td>
<td>189</td>
</tr>
<tr>
<td>imit. scripts.train_adversarial</td>
<td>191</td>
</tr>
<tr>
<td>imit. scripts.train_imitation</td>
<td>192</td>
</tr>
<tr>
<td>imit. scripts.train_preference_comparisons</td>
<td>194</td>
</tr>
<tr>
<td>imit. scripts.train_rl</td>
<td>197</td>
</tr>
<tr>
<td>imit. testing</td>
<td>198</td>
</tr>
<tr>
<td>imit. testing.expert_trjectories</td>
<td>198</td>
</tr>
<tr>
<td>imit. testing.reward_improvement</td>
<td>200</td>
</tr>
<tr>
<td>imit. testing.reward_nets</td>
<td>201</td>
</tr>
<tr>
<td>imit. util</td>
<td>202</td>
</tr>
<tr>
<td>imit. util.logger</td>
<td>203</td>
</tr>
<tr>
<td>imit. util.networks</td>
<td>208</td>
</tr>
<tr>
<td>imit. util.registry</td>
<td>212</td>
</tr>
<tr>
<td>imit. util.sacred</td>
<td>214</td>
</tr>
<tr>
<td>imit. util.util</td>
<td>215</td>
</tr>
<tr>
<td>imit. util.video_wrapper</td>
<td>219</td>
</tr>
<tr>
<td>NormalizeFeaturesExtractor</td>
<td>(class in imitation.policies.base), 153</td>
</tr>
<tr>
<td>num_batches</td>
<td>(imitation.util.networks.EMANorm attribute), 209</td>
</tr>
<tr>
<td>num_members</td>
<td>(imitation.rewards.reward_nets.RewardEnsemble property), 172</td>
</tr>
<tr>
<td>num_samples()</td>
<td>(in module imitation.data.buffer), 139</td>
</tr>
<tr>
<td>obs</td>
<td>(imitation.data.types.Trajectory attribute), 146</td>
</tr>
<tr>
<td>obs</td>
<td>(imitation.data.types.TransitionsMinimal attribute), 148</td>
</tr>
<tr>
<td>on_epoch_end</td>
<td>(imitation.algorithms.bc.BatchIteratorWithEpochEndCallback attribute), 106</td>
</tr>
<tr>
<td>optimizer</td>
<td>(imitation.policies.base.RandomPolicy attribute), 106</td>
</tr>
<tr>
<td>optimizer</td>
<td>(imitation.policies.base.ZeROPolicy attribute), 106</td>
</tr>
<tr>
<td>optimizer</td>
<td>(imitation.regularization.regularizers.LossRegularizer attribute), 159</td>
</tr>
<tr>
<td>optimizer</td>
<td>(imitation.regularization.regularizers.Regularizer attribute), 161</td>
</tr>
<tr>
<td>optimizer</td>
<td>(imitation.regularization.regularizers.WeightDecayRegularizer attribute), 162</td>
</tr>
<tr>
<td>optimizer</td>
<td>(imitation.regularization.regularizers.WeightRegularizer attribute), 162</td>
</tr>
<tr>
<td>oric()</td>
<td>(in module imitation.util.util), 218</td>
</tr>
<tr>
<td>p</td>
<td>(imitation.regularization.regularizers.LpRegularizer attribute), 160</td>
</tr>
<tr>
<td>parse_optional_path()</td>
<td>(in module imitation.data.types), 148</td>
</tr>
<tr>
<td>parse_path()</td>
<td>(in module imitation.data.types), 149</td>
</tr>
<tr>
<td>pi</td>
<td>(imitation.algorithms.mce_irl.TabularPolicy attribute), 119</td>
</tr>
<tr>
<td>policy</td>
<td>(imitation.algorithms.adversarial.common.AdversarialTrainer property), 96</td>
</tr>
<tr>
<td>policy</td>
<td>(imitation.algorithms.base.DemonstrationAlgorithm property), 102</td>
</tr>
<tr>
<td>policy</td>
<td>(imitation.algorithms.bc.BC property), 104</td>
</tr>
<tr>
<td>policy</td>
<td>(imitation.algorithms.dart.DAggerTrainer property), 110</td>
</tr>
<tr>
<td>policy</td>
<td>(imitation.algorithms.density.DensityAlgorithm property), 115</td>
</tr>
<tr>
<td>policy</td>
<td>(imitation.algorithms.mce_irl.MCEIRL property), 118</td>
</tr>
<tr>
<td>policy_registry</td>
<td>(in module imitation.policies.serialize), 158</td>
</tr>
<tr>
<td>policy_to_callable()</td>
<td>(in module imitation.data.rollout), 144</td>
</tr>
<tr>
<td>PolicyLoaderFn</td>
<td>(in module imitation.policies.serialize), 157</td>
</tr>
</tbody>
</table>
popfinished_trjectories() (imitation.data.wrappers.BufferingWrapper method), 150
pop_trjectories() (imitation.data.wrappers.BufferingWrapper method), 151
pop_transitions() (imitation.data.wrappers.BufferingWrapper method), 151
pos (imitation.policies.replay_buffer_wrapper.ReplayBufferRewardWrapper property), 156
predict() (imitation.algorithms.mce_irl.TabularPolicy method), 119
predict() (imitation.rewards.reward_nets.PredictProcessedWrapper method), 170
predict() (imitation.rewards.reward_nets.RewardNetWithVariance method), 175
predict_processed() (imitation.rewards.reward_nets.AddSTDRewardWrapper method), 165
predict_processed() (imitation.rewards.reward_nets.NormalizedRewardNet method), 170
predict_processed() (imitation.rewards.reward_nets.PredictProcessedWrapper method), 171
predict_processed() (imitation.rewards.reward_nets.RewardEnsemble method), 172
predict_processed() (imitation.rewards.reward_nets.RewardNet method), 173
predict_processed_all() (imitation.rewards.reward_nets.RewardEnsemble method), 172
predict_reward_moments() (imitation.rewards.reward_nets.RewardEnsemble method), 172
predict_reward_moments() (imitation.rewards.reward_nets.RewardNetWithVariance method), 175
predict_th() (imitation.rewards.reward_nets.PredictProcessedWrapper method), 171
predict_th() (imitation.rewards.reward_nets.RewardNet method), 174
PredictProcessedWrapper (class in imitation.rewards.reward_nets), 170
preference_collate_fn() (in module imitation.algorithms.preference_comparisons), 135
PreferenceComparisons (class in imitation.algorithms.preference_comparisons),
PreferenceDataset (class in imitation.algorithms.preference_comparisons), 127
PreferenceGatherer (class in imitation.algorithms.preference_comparisons), 129
PreferenceModel (class in imitation.algorithms.preference_comparisons), 130
preprocess() (imitation.rewards.reward_nets.RewardNet method), 174
preprocess() (imitation.rewards.reward_nets.RewardNetWrapper method), 176
prob_true_act (imitation.algorithms.bc.BCTrainingMetrics attribute), 106
probability() (imitation.algorithms.preference_comparisons.PreferenceModel method), 131
push() (imitation.algorithms.preference_comparisons.PreferenceDataset method), 129
raise_uncertainty_on_not_supported() (imitation.algorithms.preference_comparisons.ActiveSelectionFragmenter method), 123
RandomFragmenter (class in imitation.algorithms.preference_comparisons), 131
RandomPolicy (class in imitation.policies.base), 154
reconstruct_policy() (in module imitation.algorithms.bc), 107
reconstruct_trainer() (in module imitation.algorithms.dagger), 113
record() (imitation.util.logger.HierarchicalLogger method), 206
record_mean() (imitation.util.logger.HierarchicalLogger method), 206
register() (imitation.util.registry.Registry method), 213
Registry (class in imitation.util.registry), 213
regularize_and_backward() (imitation.regularization.regularizers.LossRegularizer method), 159
regularize_and_backward() (imitation.regularization.regularizers.Regularizer method), 161
regularize_and_backward() (imitation.regularization.regularizers.WeightRegularizer method), 162
Regularizer (class in imitation.regularization.regularizers), 160
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class/Method</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_shapes</td>
<td>imitation.data.buffer.Buffer</td>
<td>137</td>
</tr>
<tr>
<td>save()</td>
<td>imitation.algorithms.preference_comparisons.PreferenceDataset</td>
<td>130</td>
</tr>
<tr>
<td>save()</td>
<td>imitation.data.types</td>
<td>149</td>
</tr>
<tr>
<td>save()</td>
<td>imitation.scripts.train_adversarial</td>
<td>191</td>
</tr>
<tr>
<td>save_checkpoint()</td>
<td>imitation.scripts.train_preference_comparisons</td>
<td>194</td>
</tr>
<tr>
<td>save_model()</td>
<td>imitation.scripts.train_preference_comparisons</td>
<td>194</td>
</tr>
<tr>
<td>save_policy()</td>
<td>imitation.algorithms.bc.BC</td>
<td>104</td>
</tr>
<tr>
<td>save_policy()</td>
<td>imitation.algorithms.dagger.DAggerTrainer</td>
<td>110</td>
</tr>
<tr>
<td>save_stable_model()</td>
<td>imitation.scripts.train_preference_comparisons</td>
<td>158</td>
</tr>
<tr>
<td>save_trainer()</td>
<td>imitation.algorithms.dagger.DAggerTrainer</td>
<td>110</td>
</tr>
<tr>
<td>SavePolicyCallback</td>
<td>imitation.policies.serialize</td>
<td>157</td>
</tr>
<tr>
<td>seed()</td>
<td>imitation.algorithms.dagger.InteractiveTrajectoryCollector</td>
<td>111</td>
</tr>
<tr>
<td>set_demonstrations()</td>
<td>imitation.algorithms.adversarial.common</td>
<td>96</td>
</tr>
<tr>
<td>set_demonstrations()</td>
<td>imitation.algorithms.base</td>
<td>102</td>
</tr>
<tr>
<td>set_demonstrations()</td>
<td>imitation.algorithms.bc.BC</td>
<td>105</td>
</tr>
<tr>
<td>set_demonstrations()</td>
<td>imitation.algorithms.density</td>
<td>115</td>
</tr>
<tr>
<td>set_demonstrations()</td>
<td>imitation.algorithms.mce_irl</td>
<td>118</td>
</tr>
<tr>
<td>set_level()</td>
<td>imitation.util.logger.HierarchicalLogger</td>
<td>206</td>
</tr>
<tr>
<td>set_pi()</td>
<td>imitation.algorithms.mce_irl.TabularPolicy</td>
<td>119</td>
</tr>
<tr>
<td>setup_logging()</td>
<td>imitation.scripts.common.common</td>
<td>182</td>
</tr>
<tr>
<td>ShapedRewardNet</td>
<td>imitation.reward_nets</td>
<td>176</td>
</tr>
<tr>
<td>SimpleDaggerTrainer</td>
<td>imitation.algorithms.dagger</td>
<td>112</td>
</tr>
<tr>
<td>single_video</td>
<td>imitation.util.video_wrapper</td>
<td>220</td>
</tr>
<tr>
<td>size()</td>
<td>imitation.data.buffer.Buffer</td>
<td>137</td>
</tr>
<tr>
<td>size()</td>
<td>imitation.data.buffer.ReplayBuffer</td>
<td>139</td>
</tr>
<tr>
<td>squeeze_r()</td>
<td>imitation.algorithms.mce_irl</td>
<td>146</td>
</tr>
<tr>
<td>SqueezeLayer</td>
<td>imitation.util.networks</td>
<td>210</td>
</tr>
<tr>
<td>standardise()</td>
<td>imitation.algorithms.density.DensityAlgorithm</td>
<td>115</td>
</tr>
<tr>
<td>STATE_ACTION_DENSITY</td>
<td>imitation.algorithms.density.DensityType</td>
<td>116</td>
</tr>
<tr>
<td>STATE_DENSITY</td>
<td>imitation.algorithms.density.DensityType</td>
<td>116</td>
</tr>
<tr>
<td>STATE_STATE_DENSITY</td>
<td>imitation.algorithms.density.DensityType</td>
<td>116</td>
</tr>
<tr>
<td>step()</td>
<td>imitation.data.wrappers.RolloutInfoWrapper</td>
<td>152</td>
</tr>
<tr>
<td>step()</td>
<td>imitation.util.video_wrapper.VideoWrapper</td>
<td>220</td>
</tr>
<tr>
<td>step_async()</td>
<td>imitation.algorithms.dagger.InteractiveTrajectoryCollector</td>
<td>111</td>
</tr>
<tr>
<td>step_async()</td>
<td>imitation.data.wrappers BufferingWrapper</td>
<td>151</td>
</tr>
<tr>
<td>step_async()</td>
<td>imitation.reward_wrapper.RewardVecEnvWrapper</td>
<td>177</td>
</tr>
<tr>
<td>step_wait()</td>
<td>imitation.data.wrappers BufferingWrapper</td>
<td>151</td>
</tr>
<tr>
<td>step_wait()</td>
<td>imitation.rewards.reward_wrapper.RewardVecEnvWrapper</td>
<td>177</td>
</tr>
<tr>
<td>store()</td>
<td>imitation.data.buffer.Buffer</td>
<td>137</td>
</tr>
<tr>
<td>store()</td>
<td>imitation.data.buffer.ReplayBuffer</td>
<td>139</td>
</tr>
<tr>
<td>suppress_sacred_error()</td>
<td>imitation.scripts.common.common.train</td>
<td>187</td>
</tr>
<tr>
<td>SyntheticGatherer</td>
<td>imitation.algorithms.preference_comparisons</td>
<td>133</td>
</tr>
<tr>
<td>TabularPolicy</td>
<td>imitation.algorithms.mce_irl</td>
<td>118</td>
</tr>
<tr>
<td>tensor_iter_norm()</td>
<td>imitation.util</td>
<td>218</td>
</tr>
<tr>
<td>terminal</td>
<td>imitation.data.types.Trajectory</td>
<td>146</td>
</tr>
<tr>
<td>test_policy()</td>
<td>imitation.algorithms.density.DensityAlgorithm</td>
<td>115</td>
</tr>
<tr>
<td>train()</td>
<td>imitation.algorithms.adversarial.common.AdversarialTrainer</td>
<td>97</td>
</tr>
<tr>
<td>train()</td>
<td>imitation.algorithms.bc.BC</td>
<td>105</td>
</tr>
</tbody>
</table>
train() (imitation.algorithms.dagger.SimpleDAggerTrainer method), 112
training (imitation.rewards.reward_nets.BasicShapedRewardNet attribute), 167
train() (imitation.algorithms.density.DensityAlgorithm method), 115
training (imitation.rewards.reward_nets.BasicPotentialCNN attribute), 168
train() (imitation.algorithms.mce_irl.MCEIRL method), 118
training (imitation.rewards.reward_nets.CnnRewardNet attribute), 169
train() (imitation.algorithms.preference_comparisons.AgentTrainer attribute), 169
training (imitation.rewards.reward_nets.ForwardWrapper attribute), 170
train() (imitation.algorithms.preference_comparisons.PreferenceComparisons method), 124
training (imitation.rewards.reward_nets.NormalizedRewardNet attribute), 170
train() (imitation.algorithms.preference_comparisons.RewardTrainer attribute), 129
training (imitation.rewards.reward_nets.RewardNet attribute), 171
train() (imitation.algorithms.preference_comparisons.TrajectorGenerator method), 133
training (imitation.rewards.reward_nets.RewardNetWithVariance attribute), 175
train() (imitation.algorithms.preference_comparisons.AgentTrainer method), 135
training (imitation.rewards.reward_nets.RewardNetWrapper attribute), 176
train_adversarial() (in module imitation.scripts.train_adversarial), 191
training() (in module imitation.scripts.train_imitation), 193
train_policy() (imitation.algorithms.density.DensityAlgorithm method), 116
training() (in module imitation.scripts.train_imitation), 193
train_preference_comparisons() (in module imitation.scripts.train_preference_comparisons), 194
training() (in module imitation.scripts.train_rl), 197
train_preference_comparisons() (in module imitation.scripts.train_preference_comparisons), 194
trajectories (class in imitation.data.types), 146
Trajectory (class in imitation.data.types), 146
TrajectoryAccumulator (class in imitation.data.rollout), 141
TrajectoryGenerator (class in imitation.algorithms.preference_comparisons), 134
TrajectoryDataset (class in imitation.algorithms.preference_comparisons), 134
TrainPolicy (class in imitation.algorithms.preference_comparisons), 134
TrajectoryWithRew (class in imitation.data.types), 146
Trajectories (class in imitation.data.types), 147
Transitions (class in imitation.data.types), 147
Transitions (class in imitation.data.types), 147
TransitionsCollateFns (class in imitation.data.types), 150
UncertaintyOn (imitation.algorithms.preference_comparisons.ActiveSelectionFragmenter property), 123
unwrap_traj() (in module imitation.data.rollout), 145
update_params() (imitation.regularization.regularizers.Regularizer method), 161
update_stats() (imitation.util.networks.BaseNorm method), 209
update_stats() (imitation.util.networks.EMANorm method), 209
update_stats() (imitation.util.networks.RunningNorm method), 210
update_traj_file_in_place() (in module imitation.scripts.convert_trajs), 189

V
val_split (imitation.regularization.regularizers.LossRegularizer attribute), 160
val_split (imitation.regularization.regularizers.Regularizer attribute), 161
val_split (imitation.regularization.regularizers.WeightRegularizer attribute), 162
val_split (imitation.regularization.regularizers.WeightRegularizer attribute), 162
ValidateRewardFn (class in imitation.rewards.serialize), 178
variance_estimate() (imitation.algorithms.preference_comparisons.ActiveSelectionFragmenter method), 123
venv (imitation.algorithms.adversarial.airl.AIRL attribute), 94
venv (imitation.algorithms.adversarial.common.AdversarialTrainer attribute), 98
venv (imitation.algorithms.adversarial.gail.GAIL attribute), 100
venv (imitation.algorithms.bc.RolloutStatsComputer attribute), 107
venv (imitation.algorithms.density.DensityAlgorithm attribute), 116
venv_train (imitation.algorithms.adversarial.airl.AIRL attribute), 94
venv_train (imitation.algorithms.adversarial.common.AdversarialTrainer attribute), 98
venv_train (imitation.algorithms.adversarial.gail.GAIL attribute), 100
venv_wrapped (imitation.algorithms.adversarial.airl.AIRL attribute), 94
venv_wrapped (imitation.algorithms.adversarial.common.AdversarialTrainer attribute), 98
venv_wrapped (imitation.algorithms.adversarial.gail.GAIL attribute), 100
venv_wrapped (imitation.algorithms.density.DensityAlgorithm attribute), 116
video_recorder (imitation.util.video_wrapper.VideoWrapper attribute), 220

W
wandb_init() (in module imitation.scripts.common.wb), 187
WandbOutputFormat (class in imitation.util.logger), 206
WeightDecayRegularizer (class in imitation.regularization.regularizers), 162
WeightRegularizer (class in imitation.regularization.regularizers), 162
WrappedRewardCallback (class in imitation.rewards.reward_wrapper), 178
wrapper_callback (imitation.algorithms.density.DensityAlgorithm attribute), 116

Z
ZeroPolicy (class in imitation.policies.base), 154