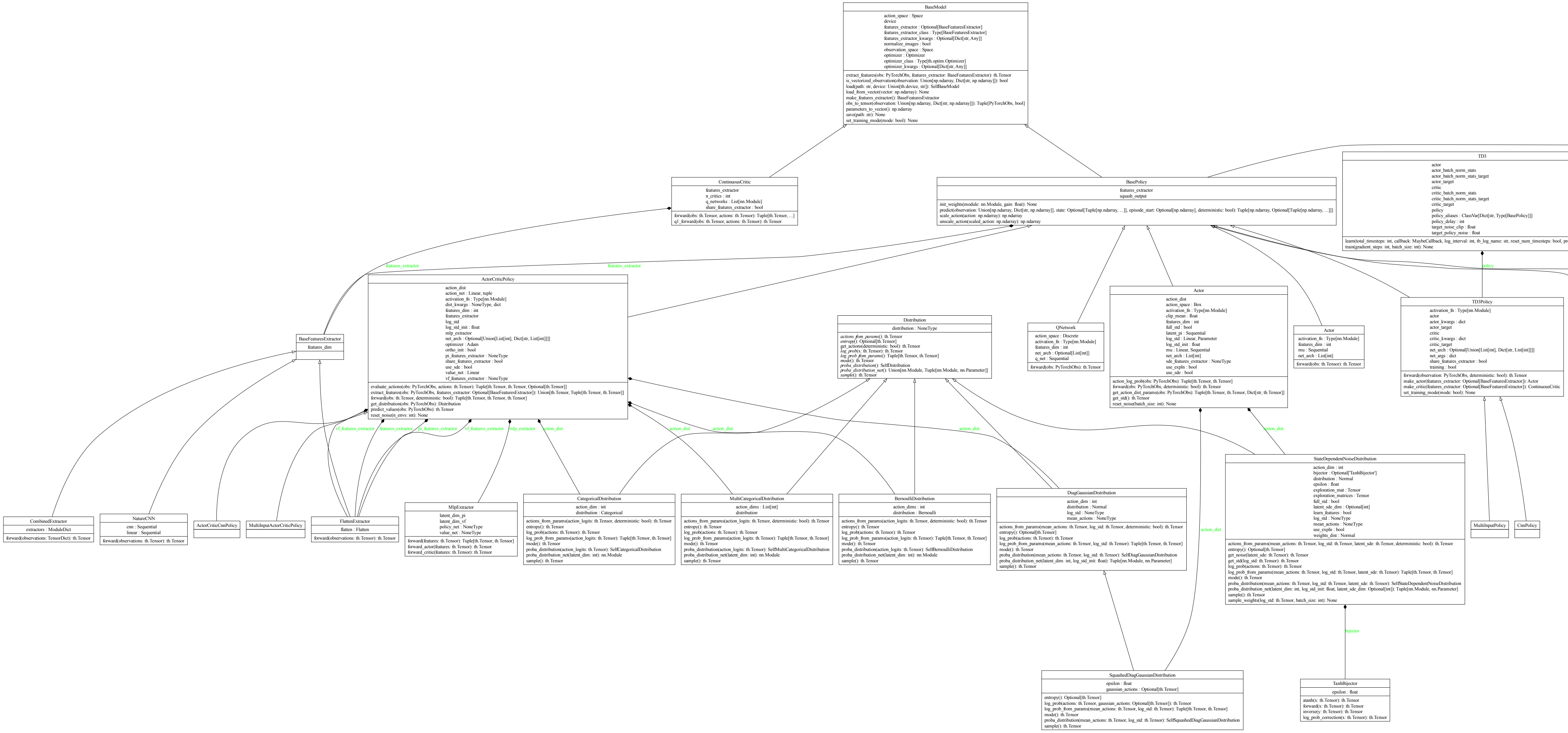
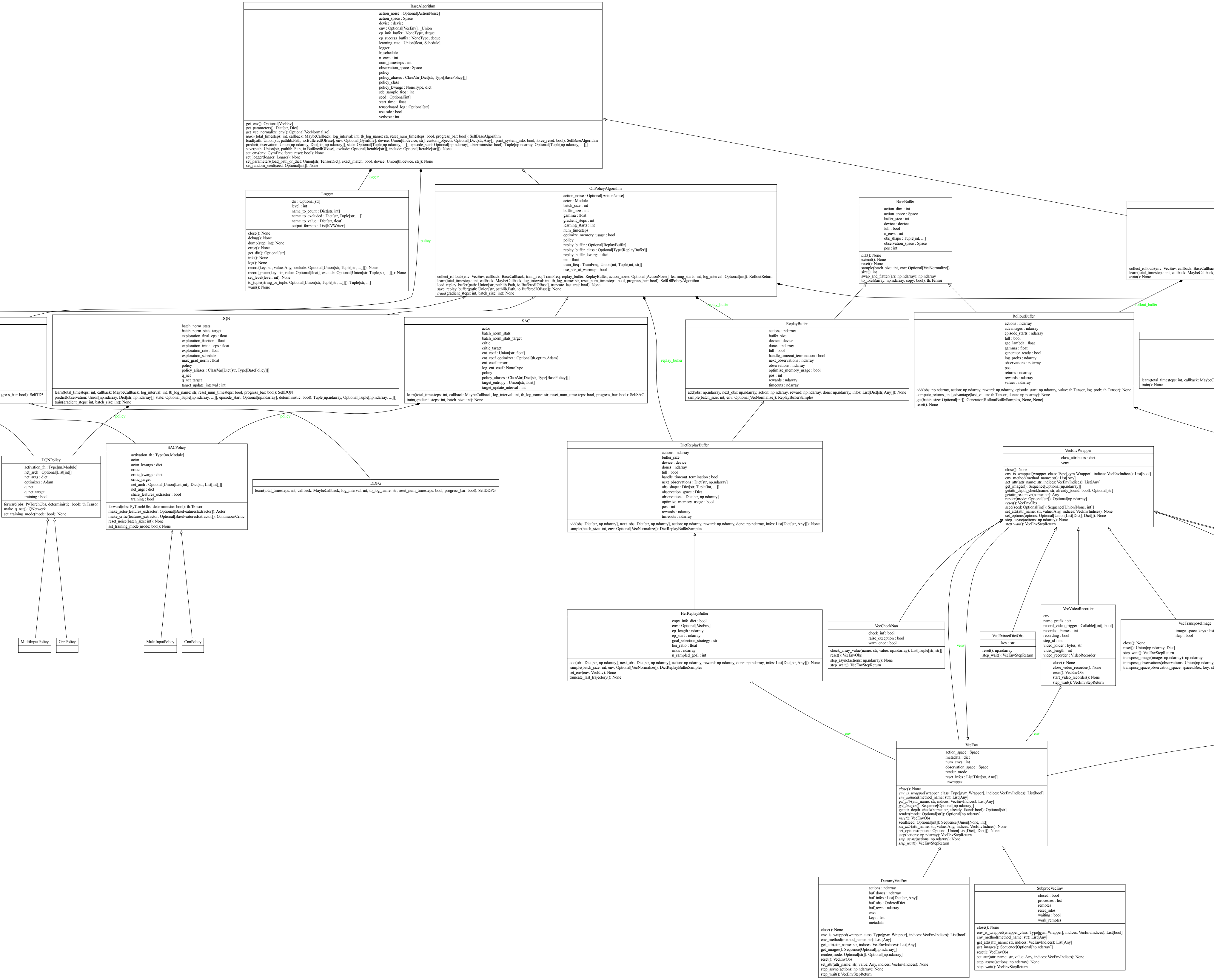
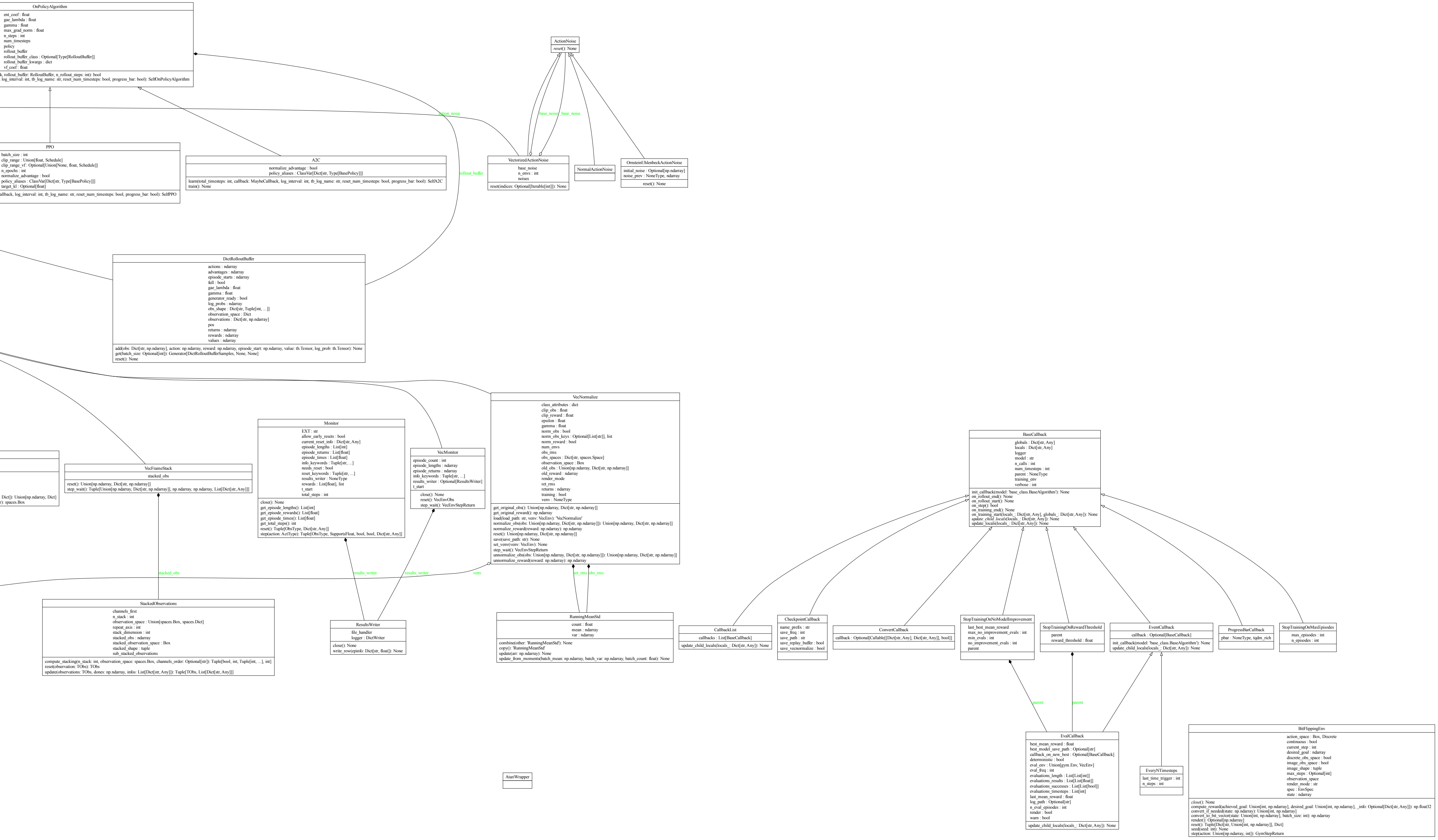
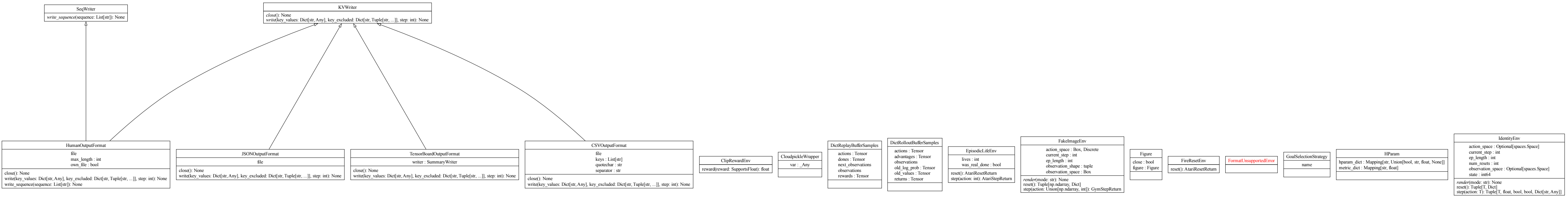
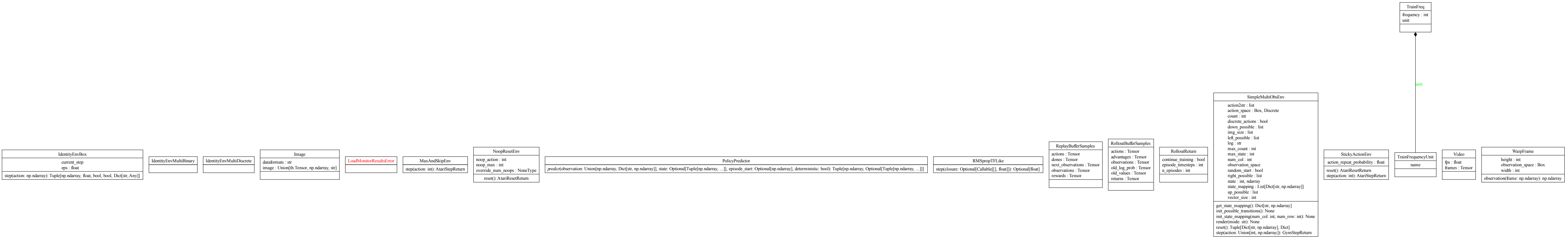

BaseModel
action_space : Space
device
features_extractor : Optional[BaseFeaturesExtractor]
features_extractor_class : Type[BaseFeaturesExtractor]
features_extractor_kwargs : Optional[Dict[str, Any]]
normalize_images : bool
observation_space : Space
optimizer : Optimizer
optimizer_class : Type[th.optim.Optimizer]
optimizer_kwargs : Optional[Dict[str, Any]]
extract_features(obs: PyTorchObs, features_extractor: BaseFeaturesExtractor): th.Tensor
is_vectorized_observation(observation: Union[np.ndarray, Dict[str, np.ndarray]]): bool
load(path: str, device: Union[th.device, str]): SelfBaseModel
load_from_vector(vector: np.ndarray): None
make_features_extractor(): BaseFeaturesExtractor
obs_to_tensor(observation: Union[np.ndarray, Dict[str, np.ndarray]]): Tuple[PyTorchObs, bool]
parameters_to_vector(): np.ndarray
save(path: str): None
set_training_mode(mode: bool): None
ContinuousCritic
features_extractor
n_critics : int
q_networks : List[nn.Module]
share_features_extractor : bool
forward(obs: th.Tensor, actions: th.Tensor): Tuple[th.Tensor, ...]
q1_forward(obs: th.Tensor, actions: th.Tensor): th.Tensor
BasePolicy
features_extractor
squash_output
init_weights(module: nn.Module, gain: float): None
predict(observation: Union[np.ndarray, Dict[str, np.ndarray]], state: Optional[Tuple[np.ndarray, ...]], episode_start: Optional[np.ndarray], deterministic: bool): Tuple[np.ndarray, Optional[Tuple[np.ndarray, ...]]]
scale_action(action: np.ndarray): np.ndarray
unscale_action(scaled_action: np.ndarray): np.ndarray
TD3
actor
actor_batch_norm_stats
actor_batch_norm_stats_target
actor_target
critic
critic_batch_norm_stats
critic_batch_norm_stats_target
critic_target
policy
policy_aliases : ClassVar[Dict[str, Type[BasePolicy]]]
policy_delay : int
target_noise_clip : float
target_policy_noise : float
learn(total_timesteps: int, callback: MaybeCallback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, pr
train(gradient_steps: int, batch_size: int): None
policy
features_extractor
features_extractor
ActorCriticPolicy
action_dist
action_net : Linear, tuple
activation_fn : Type[nn.Module]
dist_kwargs : NoneType, dict
features_dim : int
features_extractor
log_std
log_std_init : float
mlp_extractor
net_arch : Optional[Union[List[int], Dict[str, List[int]]]]
optimizer : Adam
ortho_init : bool
pi_features_extractor : NoneType
share_features_extractor : bool
use_sde : bool
value_net : Linear
vf_features_extractor : NoneType
evaluate_actions(obs: PyTorchObs, actions: th.Tensor): Tuple[th.Tensor, th.Tensor, Optional[th.Tensor]]
extract_features(obs: PyTorchObs, features_extractor: Optional[BaseFeaturesExtractor]): Union[th.Tensor, Tuple[th.Tensor, th.Tensor]]
forward(obs: th.Tensor, deterministic: bool): Tuple[th.Tensor, th.Tensor, th.Tensor]
get_distribution(obs: PyTorchObs): Distribution
predict_values(obs: PyTorchObs): th.Tensor
reset_noise(n_envs: int): None
BaseFeaturesExtractor
features_dim
Distribution
distribution : NoneType
actions_from_params(): th.Tensor
entropy(): Optional[th.Tensor]
get_actions(deterministic: bool): th.Tensor
log_prob(x: th.Tensor): th.Tensor
log_prob_from_params(): Tuple[th.Tensor, th.Tensor]
mode(): th.Tensor
proba_distribution(): SelfDistribution
proba_distribution_net(): Union[nn.Module, Tuple[nn.Module, nn.Parameter]]
sample(): th.Tensor
QNetwork
action_space : Discrete
activation_fn : Type[nn.Module]
features_dim : int
net_arch : Optional[List[int]]
q_net : Sequential
forward(obs: PyTorchObs): th.Tensor
Actor
action_dist
action_space : Box
activation_fn : Type[nn.Module]
clip_mean : float
features_dim : int
full_std : bool
latent_pi : Sequential
log_std : Linear, Parameter
log_std_init : float
mu : Linear, Sequential
net_arch : List[int]
sde_features_extractor : NoneType
use_expln : bool
use_sde : bool
action_log_prob(obs: PyTorchObs): Tuple[th.Tensor, th.Tensor]
forward(obs: PyTorchObs, deterministic: bool): th.Tensor
get_action_dist_params(obs: PyTorchObs): Tuple[th.Tensor, th.Tensor, Dict[str, th.Tensor]]
get_std(): th.Tensor
reset_noise(batch_size: int): None
Actor
activation_fn : Type[nn.Module]
features_dim : int
mu : Sequential
net_arch : List[int]
forward(obs: th.Tensor): th.Tensor
TD3Policy
activation_fn : Type[nn.Module]
actor
actor_kwargs : dict
actor_target
critic
critic_kwargs : dict
critic_target
net_arch : Optional[Union[List[int], Dict[str, List[int]]]]
net_args : dict
share_features_extractor : bool
training : bool
forward(observation: PyTorchObs, deterministic: bool): th.Tensor
make_actor(features_extractor: Optional[BaseFeaturesExtractor]): Actor
make_critic(features_extractor: Optional[BaseFeaturesExtractor]): ContinuousCritic
set_training_mode(mode: bool): None
vf_features_extractor
features_extractor
pi_features_extractor
vf_features_extractor
mlp_extractor
action_dist
action_dist
action_dist
action_dist
action_dist
action_dist
CombinedExtractor
extractors : ModuleDict
forward(observations: TensorDict): th.Tensor
NatureCNN
cnn : Sequential
linear : Sequential
forward(observations: th.Tensor): th.Tensor
ActorCriticCnnPolicy
MultiInputActorCriticPolicy
FlattenExtractor
flatten : Flatten
forward(observations: th.Tensor): th.Tensor
MlpExtractor
latent_dim_pi
latent_dim_vf
policy_net : NoneType
value_net : NoneType
forward(features: th.Tensor): Tuple[th.Tensor, th.Tensor]
forward_actor(features: th.Tensor): th.Tensor
forward_critic(features: th.Tensor): th.Tensor
CategoricalDistribution
action_dim : int
distribution : Categorical
actions_from_params(action_logits: th.Tensor, deterministic: bool): th.Tensor
entropy(): th.Tensor
log_prob(actions: th.Tensor): th.Tensor
log_prob_from_params(action_logits: th.Tensor): Tuple[th.Tensor, th.Tensor]
mode(): th.Tensor
proba_distribution(action_logits: th.Tensor): SelfCategoricalDistribution
proba_distribution_net(latent_dim: int): nn.Module
sample(): th.Tensor
MultiCategoricalDistribution
action_dims : List[int]
distribution
actions_from_params(action_logits: th.Tensor, deterministic: bool): th.Tensor
entropy(): th.Tensor
log_prob(actions: th.Tensor): th.Tensor
log_prob_from_params(action_logits: th.Tensor): Tuple[th.Tensor, th.Tensor]
mode(): th.Tensor
proba_distribution(action_logits: th.Tensor): SelfMultiCategoricalDistribution
proba_distribution_net(latent_dim: int): nn.Module
sample(): th.Tensor
BernoulliDistribution
action_dims : int
distribution : Bernoulli
actions_from_params(action_logits: th.Tensor, deterministic: bool): th.Tensor
entropy(): th.Tensor
log_prob(actions: th.Tensor): th.Tensor
log_prob_from_params(action_logits: th.Tensor): Tuple[th.Tensor, th.Tensor]
mode(): th.Tensor
proba_distribution(action_logits: th.Tensor): SelfBernoulliDistribution
proba_distribution_net(latent_dim: int): nn.Module
sample(): th.Tensor
DiagGaussianDistribution
action_dim : int
distribution : Normal
log_std : NoneType
mean_actions : NoneType
actions_from_params(mean_actions: th.Tensor, log_std: th.Tensor, deterministic: bool): th.Tensor
entropy(): Optional[th.Tensor]
log_prob(actions: th.Tensor): th.Tensor
log_prob_from_params(mean_actions: th.Tensor, log_std: th.Tensor): Tuple[th.Tensor, th.Tensor]
mode(): th.Tensor
proba_distribution(mean_actions: th.Tensor, log_std: th.Tensor): SelfDiagGaussianDistribution
proba_distribution_net(latent_dim: int, log_std_init: float): Tuple[nn.Module, nn.Parameter]
sample(): th.Tensor
action_dist
StateDependentNoiseDistribution
action_dim : int
bijector : Optional['TanhBijector']
distribution : Normal
epsilon : float
exploration_mat : Tensor
exploration_matrices : Tensor
full_std : bool
latent_sde_dim : Optional[int]
learn_features : bool
log_std : NoneType
mean_actions : NoneType
use_expln : bool
weights_dist : Normal
actions_from_params(mean_actions: th.Tensor, log_std: th.Tensor, latent_sde: th.Tensor, deterministic: bool): th.Tensor
entropy(): Optional[th.Tensor]
get_noise(latent_sde: th.Tensor): th.Tensor
get_std(log_std: th.Tensor): th.Tensor
log_prob(actions: th.Tensor): th.Tensor
log_prob_from_params(mean_actions: th.Tensor, log_std: th.Tensor, latent_sde: th.Tensor): Tuple[th.Tensor, th.Tensor]
mode(): th.Tensor
proba_distribution(mean_actions: th.Tensor, log_std: th.Tensor, latent_sde: th.Tensor): SelfStateDependentNoiseDistribution
proba_distribution_net(latent_dim: int, log_std_init: float, latent_sde_dim: Optional[int]): Tuple[nn.Module, nn.Parameter]
sample(): th.Tensor
sample_weights(log_std: th.Tensor, batch_size: int): None
MultiInputPolicy
CnnPolicy
bijector
SquashedDiagGaussianDistribution
epsilon : float
gaussian_actions : Optional[th.Tensor]
entropy(): Optional[th.Tensor]
log_prob(actions: th.Tensor, gaussian_actions: Optional[th.Tensor]): th.Tensor
log_prob_from_params(mean_actions: th.Tensor, log_std: th.Tensor): Tuple[th.Tensor, th.Tensor]
mode(): th.Tensor
proba_distribution(mean_actions: th.Tensor, log_std: th.Tensor): SelfSquashedDiagGaussianDistribution
sample(): th.Tensor
TanhBijector
epsilon : float
atanh(x: th.Tensor): th.Tensor
forward(x: th.Tensor): th.Tensor
inverse(y: th.Tensor): th.Tensor
log_prob_correction(x: th.Tensor): th.Tensor


BaseAlgorithm
action_noise : Optional[ActionNoise]
action_space : Space
device : device
env : Optional[VecEnv], _Union
ep_info_buffer : NoneType, deque
ep_success_buffer : NoneType, deque
learning_rate : Union[float, Schedule]
logger
lr_schedule
n_envs : int
num_timesteps : int
observation_space : Space
policy
policy_aliases : ClassVar[Dict[str, Type[BasePolicy]]]
policy_class
policy_kwargs : NoneType, dict
sde_sample_freq : int
seed : Optional[int]
start_time : float
tensorboard_log : Optional[str]
use_sde : bool
verbose : int
get_env(): Optional[VecEnv]
get_parameters(): Dict[str, Dict]
get_vec_normalize_env(): Optional[VecNormalize]
learn(total_timesteps: int, callback: MaybeCallback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfBaseAlgorithm
load(path: Union[str, pathlib.Path, io.BufferedIOBase], env: Optional[GymEnv], device: Union[th.device, str], custom_objects: Optional[Dict[str, Any]], print_system_info: bool, force_reset: bool): SelfBaseAlgorithm
predict(observation: Union[np.ndarray, Dict[str, np.ndarray]], state: Optional[Tuple[np.ndarray, ...]], episode_start: Optional[np.ndarray], deterministic: bool): Tuple[np.ndarray, Optional[Tuple[np.ndarray, ...]]]
save(path: Union[str, pathlib.Path, io.BufferedIOBase], exclude: Optional[Iterable[str]], include: Optional[Iterable[str]]): None
set_env(env: GymEnv, force_reset: bool): None
set_logger(logger: Logger): None
set_parameters(load_path_or_dict: Union[str, TensorDict], exact_match: bool, device: Union[th.device, str]): None
set_random_seed(seed: Optional[int]): None
_logger
Logger
dir : Optional[str]
level : int
name_to_count : Dict[str, int]
name_to_excluded : Dict[str, Tuple[str, ...]]
name_to_value : Dict[str, float]
output_formats : List[KVWriter]
close(): None
debug(): None
dump(step: int): None
error(): None
get_dir(): Optional[str]
info(): None
log(): None
record(key: str, value: Any, exclude: Optional[Union[str, Tuple[str, ...]]]): None
record_mean(key: str, value: Optional[float], exclude: Optional[Union[str, Tuple[str, ...]]]): None
set_level(level: int): None
to_tuple(string_or_tuple: Optional[Union[str, Tuple[str, ...]]]): Tuple[str, ...]
warn(): None
policy
OffPolicyAlgorithm
action_noise : Optional[ActionNoise]
actor : Module
batch_size : int
buffer_size : int
gamma : float
gradient_steps : int
learning_starts : int
num_timesteps
optimize_memory_usage : bool
policy
replay_buffer : Optional[ReplayBuffer]
replay_buffer_class : Optional[Type[ReplayBuffer]]
replay_buffer_kwargs : dict
tau : float
train_freq : TrainFreq, Union[int, Tuple[int, str]]
use_sde_at_warmup : bool
collect_rollouts(env: VecEnv, callback: BaseCallback, train_freq: TrainFreq, replay_buffer: ReplayBuffer, action_noise: Optional[ActionNoise], learning_starts: int, log_interval: Optional[int]): RolloutReturn
learn(total_timesteps: int, callback: MaybeCallback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfOffPolicyAlgorithm
load_replay_buffer(path: Union[str, pathlib.Path, io.BufferedIOBase], truncate_last_traj: bool): None
save_replay_buffer(path: Union[str, pathlib.Path, io.BufferedIOBase]): None
train(gradient_steps: int, batch_size: int): None
BaseBuffer
action_dim : int
action_space : Space
buffer_size : int
device : device
full : bool
n_envs : int
obs_shape : Tuple[int, ...]
observation_space : Space
pos : int
add(): None
extend(): None
reset(): None
sample(batch_size: int, env: Optional[VecNormalize])
size(): int
swap_and_flatten(arr: np.ndarray): np.ndarray
to_torch(array: np.ndarray, copy: bool): th.Tensor
collect_rollouts(env: VecEnv, callback: BaseCallbac
learn(total_timesteps: int, callback: MaybeCallback,
train(): None
rollout_buffer
ogress_bar: bool): SelfTD3
DQN
batch_norm_stats
batch_norm_stats_target
exploration_final_eps : float
exploration_fraction : float
exploration_initial_eps : float
exploration_rate : float
exploration_schedule
max_grad_norm : float
policy
policy_aliases : ClassVar[Dict[str, Type[BasePolicy]]]
q_net
q_net_target
target_update_interval : int
learn(total_timesteps: int, callback: MaybeCallback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfDQN
predict(observation: Union[np.ndarray, Dict[str, np.ndarray]], state: Optional[Tuple[np.ndarray, ...]], episode_start: Optional[np.ndarray], deterministic: bool): Tuple[np.ndarray, Optional[Tuple[np.ndarray, ...]]]
train(gradient_steps: int, batch_size: int): None
SAC
actor
batch_norm_stats
batch_norm_stats_target
critic
critic_target
ent_coef : Union[str, float]
ent_coef_optimizer : Optional[th.optim.Adam]
ent_coef_tensor
log_ent_coef : NoneType
policy
policy_aliases : ClassVar[Dict[str, Type[BasePolicy]]]
target_entropy : Union[str, float]
target_update_interval : int
learn(total_timesteps: int, callback: MaybeCallback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfSAC
train(gradient_steps: int, batch_size: int): None
replay_buffer
replay_buffer
ReplayBuffer
actions : ndarray
buffer_size
device : device
dones : ndarray
full : bool
handle_timeout_termination : bool
next_observations : ndarray
observations : ndarray
optimize_memory_usage : bool
pos : int
rewards : ndarray
timeouts : ndarray
add(obs: np.ndarray, next_obs: np.ndarray, action: np.ndarray, reward: np.ndarray, done: np.ndarray, infos: List[Dict[str, Any]]): None
sample(batch_size: int, env: Optional[VecNormalize]): ReplayBufferSamples
RolloutBuffer
actions : ndarray
advantages : ndarray
episode_starts : ndarray
full : bool
gae_lambda : float
gamma : float
generator_ready : bool
log_probs : ndarray
observations : ndarray
pos
returns : ndarray
rewards : ndarray
values : ndarray
add(obs: np.ndarray, action: np.ndarray, reward: np.ndarray, episode_start: np.ndarray, value: th.Tensor, log_prob: th.Tensor): None
compute_returns_and_advantage(last_values: th.Tensor, dones: np.ndarray): None
get(batch_size: Optional[int]): Generator[RolloutBufferSamples, None, None]
reset(): None
learn(total_timesteps: int, callback: MaybeC
train(): None
policy
policy
DQNPolicy
activation_fn : Type[nn.Module]
net_arch : Optional[List[int]]
net_args : dict
optimizer : Adam
q_net
q_net_target
training : bool
forward(obs: PyTorchObs, deterministic: bool): th.Tensor
make_q_net(): QNetwork
set_training_mode(mode: bool): None
SACPolicy
activation_fn : Type[nn.Module]
actor
actor_kwargs : dict
critic
critic_kwargs : dict
critic_target
net_arch : Optional[Union[List[int], Dict[str, List[int]]]]
net_args : dict
share_features_extractor : bool
training : bool
forward(obs: PyTorchObs, deterministic: bool): th.Tensor
make_actor(features_extractor: Optional[BaseFeaturesExtractor]): Actor
make_critic(features_extractor: Optional[BaseFeaturesExtractor]): ContinuousCritic
reset_noise(batch_size: int): None
set_training_mode(mode: bool): None
DDPG
learn(total_timesteps: int, callback: MaybeCallback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfDDPG
DictReplayBuffer
actions : ndarray
buffer_size
device : device
dones : ndarray
full : bool
handle_timeout_termination : bool
next_observations : Dict[str, np.ndarray]
obs_shape : Dict[str, Tuple[int, ...]]
observation_space : Dict
observations : Dict[str, np.ndarray]
optimize_memory_usage : bool
pos : int
rewards : ndarray
timeouts : ndarray
add(obs: Dict[str, np.ndarray], next_obs: Dict[str, np.ndarray], action: np.ndarray, reward: np.ndarray, done: np.ndarray, infos: List[Dict[str, Any]]): None
sample(batch_size: int, env: Optional[VecNormalize]): DictReplayBufferSamples
VecEnvWrapper
class_attributes : dict
venv
close(): None
env_is_wrapped(wrapper_class: Type[gym.Wrapper], indices: VecEnvIndices): List[bool]
env_method(method_name: str): List[Any]
get_attr(attr_name: str, indices: VecEnvIndices): List[Any]
get_images(): Sequence[Optional[np.ndarray]]
getattr_depth_check(name: str, already_found: bool): Optional[str]
getattr_recursive(name: str): Any
render(mode: Optional[str]): Optional[np.ndarray]
reset(): VecEnvObs
seed(seed: Optional[int]): Sequence[Union[None, int]]
set_attr(attr_name: str, value: Any, indices: VecEnvIndices): None
set_options(options: Optional[Union[List[Dict], Dict]]): None
step_async(actions: np.ndarray): None
step_wait(): VecEnvStepReturn
MultiInputPolicy
CnnPolicy
MultiInputPolicy
CnnPolicy
HerReplayBuffer
copy_info_dict : bool
env : Optional[VecEnv]
ep_length : ndarray
ep_start : ndarray
goal_selection_strategy : str
her_ratio : float
infos : ndarray
n_sampled_goal : int
add(obs: Dict[str, np.ndarray], next_obs: Dict[str, np.ndarray], action: np.ndarray, reward: np.ndarray, done: np.ndarray, infos: List[Dict[str, Any]]): None
sample(batch_size: int, env: Optional[VecNormalize]): DictReplayBufferSamples
set_env(env: VecEnv): None
truncate_last_trajectory(): None
VecCheckNan
check_inf : bool
raise_exception : bool
warn_once : bool
check_array_value(name: str, value: np.ndarray): List[Tuple[str, str]]
reset(): VecEnvObs
step_async(actions: np.ndarray): None
step_wait(): VecEnvStepReturn
venv
VecExtractDictObs
key : str
reset(): np.ndarray
step_wait(): VecEnvStepReturn
VecVideoRecorder
env
name_prefix : str
record_video_trigger : Callable[[int], bool]
recorded_frames : int
recording : bool
step_id : int
video_folder : bytes, str
video_length : int
video_recorder : VideoRecorder
close(): None
close_video_recorder(): None
reset(): VecEnvObs
start_video_recorder(): None
step_wait(): VecEnvStepReturn
VecTransposeImage
image_space_keys : list
skip : bool
close(): None
reset(): Union[np.ndarray, Dict]
step_wait(): VecEnvStepReturn
transpose_image(image: np.ndarray): np.ndarray
transpose_observations(observations: Union[np.ndarray,
transpose_space(observation_space: spaces.Box, key: st
env
env
VecEnv
action_space : Space
metadata : dict
num_envs : int
observation_space : Space
render_mode
reset_infos : List[Dict[str, Any]]
unwrapped
close(): None
env_is_wrapped(wrapper_class: Type[gym.Wrapper], indices: VecEnvIndices): List[bool]
env_method(method_name: str): List[Any]
get_attr(attr_name: str, indices: VecEnvIndices): List[Any]
get_images(): Sequence[Optional[np.ndarray]]
getattr_depth_check(name: str, already_found: bool): Optional[str]
render(mode: Optional[str]): Optional[np.ndarray]
reset(): VecEnvObs
seed(seed: Optional[int]): Sequence[Union[None, int]]
set_attr(attr_name: str, value: Any, indices: VecEnvIndices): None
set_options(options: Optional[Union[List[Dict], Dict]]): None
step(actions: np.ndarray): VecEnvStepReturn
step_async(actions: np.ndarray): None
step_wait(): VecEnvStepReturn
DummyVecEnv
actions : ndarray
buf_dones : ndarray
buf_infos : List[Dict[str, Any]]
buf_obs : OrderedDict
buf_rews : ndarray
envs
keys : list
metadata
close(): None
env_is_wrapped(wrapper_class: Type[gym.Wrapper], indices: VecEnvIndices): List[bool]
env_method(method_name: str): List[Any]
get_attr(attr_name: str, indices: VecEnvIndices): List[Any]
get_images(): Sequence[Optional[np.ndarray]]
render(mode: Optional[str]): Optional[np.ndarray]
reset(): VecEnvObs
set_attr(attr_name: str, value: Any, indices: VecEnvIndices): None
step_async(actions: np.ndarray): None
step_wait(): VecEnvStepReturn
SubprocVecEnv
closed : bool
processes : list
remotes
reset_infos
waiting : bool
work_remotes
close(): None
env_is_wrapped(wrapper_class: Type[gym.Wrapper], indices: VecEnvIndices): List[bool]
env_method(method_name: str): List[Any]
get_attr(attr_name: str, indices: VecEnvIndices): List[Any]
get_images(): Sequence[Optional[np.ndarray]]
reset(): VecEnvObs
set_attr(attr_name: str, value: Any, indices: VecEnvIndices): None
step_async(actions: np.ndarray): None
step_wait(): VecEnvStepReturn


OnPolicyAlgorithm
ent_coef : float
gae_lambda : float
gamma : float
max_grad_norm : float
n_steps : int
num_timesteps
policy
rollout_buffer
rollout_buffer_class : Optional[Type[RolloutBuffer]]
rollout_buffer_kwargs : dict
vf_coef : float
k, rollout_buffer: RolloutBuffer, n_rollout_steps: int): bool
, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfOnPolicyAlgorithm
ActionNoise
reset(): None
action_noise
base_noise
_base_noise
rollout_buffer
PPO
batch_size : int
clip_range : Union[float, Schedule]
clip_range_vf : Optional[Union[None, float, Schedule]]
n_epochs : int
normalize_advantage : bool
policy_aliases : ClassVar[Dict[str, Type[BasePolicy]]]
target_kl : Optional[float]
allback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfPPO
A2C
normalize_advantage : bool
policy_aliases : ClassVar[Dict[str, Type[BasePolicy]]]
learn(total_timesteps: int, callback: MaybeCallback, log_interval: int, tb_log_name: str, reset_num_timesteps: bool, progress_bar: bool): SelfA2C
train(): None
VectorizedActionNoise
base_noise
n_envs : int
noises
reset(indices: Optional[Iterable[int]]): None
NormalActionNoise
OrnsteinUhlenbeckActionNoise
initial_noise : Optional[np.ndarray]
noise_prev : NoneType, ndarray
reset(): None
DictRolloutBuffer
actions : ndarray
advantages : ndarray
episode_starts : ndarray
full : bool
gae_lambda : float
gamma : float
generator_ready : bool
log_probs : ndarray
obs_shape : Dict[str, Tuple[int, ...]]
observation_space : Dict
observations : Dict[str, np.ndarray]
pos
returns : ndarray
rewards : ndarray
values : ndarray
add(obs: Dict[str, np.ndarray], action: np.ndarray, reward: np.ndarray, episode_start: np.ndarray, value: th.Tensor, log_prob: th.Tensor): None
get(batch_size: Optional[int]): Generator[DictRolloutBufferSamples, None, None]
reset(): None
VecNormalize
class_attributes : dict
clip_obs : float
clip_reward : float
epsilon : float
gamma : float
norm_obs : bool
norm_obs_keys : Optional[List[str]], list
norm_reward : bool
num_envs
obs_rms
obs_spaces : Dict[str, spaces.Space]
observation_space : Box
old_obs : Union[np.ndarray, Dict[str, np.ndarray]]
old_reward : ndarray
render_mode
ret_rms
returns : ndarray
training : bool
venv : NoneType
get_original_obs(): Union[np.ndarray, Dict[str, np.ndarray]]
get_original_reward(): np.ndarray
load(load_path: str, venv: VecEnv): 'VecNormalize'
normalize_obs(obs: Union[np.ndarray, Dict[str, np.ndarray]]): Union[np.ndarray, Dict[str, np.ndarray]]
normalize_reward(reward: np.ndarray): np.ndarray
reset(): Union[np.ndarray, Dict[str, np.ndarray]]
save(save_path: str): None
set_venv(venv: VecEnv): None
step_wait(): VecEnvStepReturn
unnormalize_obs(obs: Union[np.ndarray, Dict[str, np.ndarray]]): Union[np.ndarray, Dict[str, np.ndarray]]
unnormalize_reward(reward: np.ndarray): np.ndarray
Monitor
EXT : str
allow_early_resets : bool
current_reset_info : Dict[str, Any]
episode_lengths : List[int]
episode_returns : List[float]
episode_times : List[float]
info_keywords : Tuple[str, ...]
needs_reset : bool
reset_keywords : Tuple[str, ...]
results_writer : NoneType
rewards : List[float], list
t_start
total_steps : int
close(): None
get_episode_lengths(): List[int]
get_episode_rewards(): List[float]
get_episode_times(): List[float]
get_total_steps(): int
reset(): Tuple[ObsType, Dict[str, Any]]
step(action: ActType): Tuple[ObsType, SupportsFloat, bool, bool, Dict[str, Any]]
VecMonitor
episode_count : int
episode_lengths : ndarray
episode_returns : ndarray
info_keywords : Tuple[str, ...]
results_writer : Optional[ResultsWriter]
t_start
close(): None
reset(): VecEnvObs
step_wait(): VecEnvStepReturn
Dict]): Union[np.ndarray, Dict]
r): spaces.Box
VecFrameStack
stacked_obs
reset(): Union[np.ndarray, Dict[str, np.ndarray]]
step_wait(): Tuple[Union[np.ndarray, Dict[str, np.ndarray]], np.ndarray, np.ndarray, List[Dict[str, Any]]]
BaseCallback
globals : Dict[str, Any]
locals : Dict[str, Any]
logger
model : str
n_calls : int
num_timesteps : int
parent : NoneType
training_env
verbose : int
init_callback(model: 'base_class.BaseAlgorithm'): None
on_rollout_end(): None
on_rollout_start(): None
on_step(): bool
on_training_end(): None
on_training_start(locals_: Dict[str, Any], globals_: Dict[str, Any]): None
update_child_locals(locals_: Dict[str, Any]): None
update_locals(locals_: Dict[str, Any]): None
stacked_obs
results_writer
results_writer
venv
ret_rms
obs_rms
StackedObservations
channels_first
n_stack : int
observation_space : Union[spaces.Box, spaces.Dict]
repeat_axis : int
stack_dimension : int
stacked_obs : ndarray
stacked_observation_space : Box
stacked_shape : tuple
sub_stacked_observations
compute_stacking(n_stack: int, observation_space: spaces.Box, channels_order: Optional[str]): Tuple[bool, int, Tuple[int, ...], int]
reset(observation: TObs): TObs
update(observations: TObs, dones: np.ndarray, infos: List[Dict[str, Any]]): Tuple[TObs, List[Dict[str, Any]]]
ResultsWriter
file_handler
logger : DictWriter
close(): None
write_row(epinfo: Dict[str, float]): None
RunningMeanStd
count : float
mean : ndarray
var : ndarray
combine(other: 'RunningMeanStd'): None
copy(): 'RunningMeanStd'
update(arr: np.ndarray): None
update_from_moments(batch_mean: np.ndarray, batch_var: np.ndarray, batch_count: float): None
CallbackList
callbacks : List[BaseCallback]
update_child_locals(locals_: Dict[str, Any]): None
CheckpointCallback
name_prefix : str
save_freq : int
save_path : str
save_replay_buffer : bool
save_vecnormalize : bool
ConvertCallback
callback : Optional[Callable[[Dict[str, Any], Dict[str, Any]], bool]]
StopTrainingOnNoModelImprovement
last_best_mean_reward
max_no_improvement_evals : int
min_evals : int
no_improvement_evals : int
parent
StopTrainingOnRewardThreshold
parent
reward_threshold : float
EventCallback
callback : Optional[BaseCallback]
init_callback(model: 'base_class.BaseAlgorithm'): None
update_child_locals(locals_: Dict[str, Any]): None
ProgressBarCallback
pbar : NoneType, tqdm_rich
StopTrainingOnMaxEpisodes
max_episodes : int
n_episodes : int
parent
parent
EvalCallback
best_mean_reward : float
best_model_save_path : Optional[str]
callback_on_new_best : Optional[BaseCallback]
deterministic : bool
eval_env : Union[gym.Env, VecEnv]
eval_freq : int
evaluations_length : List[List[int]]
evaluations_results : List[List[float]]
evaluations_successes : List[List[bool]]
evaluations_timesteps : List[int]
last_mean_reward : float
log_path : Optional[str]
n_eval_episodes : int
render : bool
warn : bool
update_child_locals(locals_: Dict[str, Any]): None
AtariWrapper
EveryNTimesteps
last_time_trigger : int
n_steps : int
BitFlippingEnv
action_space : Box, Discrete
continuous : bool
current_step : int
desired_goal : ndarray
discrete_obs_space : bool
image_obs_space : bool
image_shape : tuple
max_steps : Optional[int]
observation_space
render_mode : str
spec : EnvSpec
state : ndarray
close(): None
compute_reward(achieved_goal: Union[int, np.ndarray], desired_goal: Union[int, np.ndarray], _info: Optional[Dict[str, Any]]): np.float32
convert_if_needed(state: np.ndarray): Union[int, np.ndarray]
convert_to_bit_vector(state: Union[int, np.ndarray], batch_size: int): np.ndarray
render(): Optional[np.ndarray]
reset(): Tuple[Dict[str, Union[int, np.ndarray]], Dict]
seed(seed: int): None
step(action: Union[np.ndarray, int]): GymStepReturn


SeqWriter
write_sequence(sequence: List[str]): None
KVWriter
close(): None
write(key_values: Dict[str, Any], key_excluded: Dict[str, Tuple[str, ...]], step: int): None
HumanOutputFormat
file
max_length : int
own_file : bool
close(): None
write(key_values: Dict[str, Any], key_excluded: Dict[str, Tuple[str, ...]], step: int): None
write_sequence(sequence: List[str]): None
JSONOutputFormat
file
close(): None
write(key_values: Dict[str, Any], key_excluded: Dict[str, Tuple[str, ...]], step: int): None
TensorBoardOutputFormat
writer : SummaryWriter
close(): None
write(key_values: Dict[str, Any], key_excluded: Dict[str, Tuple[str, ...]], step: int): None
CSVOutputFormat
file
keys : List[str]
quotechar : str
separator : str
close(): None
write(key_values: Dict[str, Any], key_excluded: Dict[str, Tuple[str, ...]], step: int): None
ClipRewardEnv
reward(reward: SupportsFloat): float
CloudpickleWrapper
var : _Any
DictReplayBufferSamples
actions : Tensor
dones : Tensor
next_observations
observations
rewards : Tensor
DictRolloutBufferSamples
actions : Tensor
advantages : Tensor
observations
old_log_prob : Tensor
old_values : Tensor
returns : Tensor
EpisodicLifeEnv
lives : int
was_real_done : bool
reset(): AtariResetReturn
step(action: int): AtariStepReturn
FakeImageEnv
action_space : Box, Discrete
current_step : int
ep_length : int
observation_shape : tuple
observation_space : Box
render(mode: str): None
reset(): Tuple[np.ndarray, Dict]
step(action: Union[np.ndarray, int]): GymStepReturn
Figure
close : bool
figure : Figure
FireResetEnv
reset(): AtariResetReturn
FormatUnsupportedError
GoalSelectionStrategy
name
HParam
hparam_dict : Mapping[str, Union[bool, str, float, None]]
metric_dict : Mapping[str, float]
IdentityEnv
action_space : Optional[spaces.Space]
current_step : int
ep_length : int
num_resets : int
observation_space : Optional[spaces.Space]
state : int64
render(mode: str): None
reset(): Tuple[T, Dict]
step(action: T): Tuple[T, float, bool, bool, Dict[str, Any]]


IdentityEnvBox
current_step
eps : float
step(action: np.ndarray): Tuple[np.ndarray, float, bool, bool, Dict[str, Any]]
IdentityEnvMultiBinary
IdentityEnvMultiDiscrete
Image
dataformats : str
image : Union[th.Tensor, np.ndarray, str]
LoadMonitorResultsError
MaxAndSkipEnv
step(action: int): AtariStepReturn
NoopResetEnv
noop_action : int
noop_max : int
override_num_noops : NoneType
reset(): AtariResetReturn
PolicyPredictor
predict(observation: Union[np.ndarray, Dict[str, np.ndarray]], state: Optional[Tuple[np.ndarray, ...]], episode_start: Optional[np.ndarray], deterministic: bool): Tuple[np.ndarray, Optional[Tuple[np.ndarray, ...]]]
RMSpropTFLike
step(closure: Optional[Callable[[], float]]): Optional[float]
ReplayBufferSamples
actions : Tensor
dones : Tensor
next_observations : Tensor
observations : Tensor
rewards : Tensor
RolloutBufferSamples
actions : Tensor
advantages : Tensor
observations : Tensor
old_log_prob : Tensor
old_values : Tensor
returns : Tensor
RolloutReturn
continue_training : bool
episode_timesteps : int
n_episodes : int
SimpleMultiObsEnv
action2str : list
action_space : Box, Discrete
count : int
discrete_actions : bool
down_possible : list
img_size : list
left_possible : list
log : str
max_count : int
max_state : int
num_col : int
observation_space
random_start : bool
right_possible : list
state : int, ndarray
state_mapping : List[Dict[str, np.ndarray]]
up_possible : list
vector_size : int
get_state_mapping(): Dict[str, np.ndarray]
init_possible_transitions(): None
init_state_mapping(num_col: int, num_row: int): None
render(mode: str): None
reset(): Tuple[Dict[str, np.ndarray], Dict]
step(action: Union[int, np.ndarray]): GymStepReturn
StickyActionEnv
action_repeat_probability : float
reset(): AtariResetReturn
step(action: int): AtariStepReturn
TrainFreq
frequency : int
unit
unit
TrainFrequencyUnit
name
Video
fps : float
frames : Tensor
WarpFrame
height : int
observation_space : Box
width : int
observation(frame: np.ndarray): np.ndarray