Multi-Agent Twin-Delayed Deep Deterministic Policy Gradient (MATD3)

MATD3 (Multi-Agent Twin Delayed Deep Deterministic Policy Gradients) extends the MADDPG algorithm to reduce overestimation bias in multi-agent domains through the use of a second set of critic networks and delayed updates of the policy networks. This enables superior performance when compared to MADDPG.

Compatible Action Spaces

`Discrete`	`Box`	`MultiDiscrete`	`MultiBinary`
✔️	✔️	❌	❌

Gumbel-Softmax

The Gumbel-Softmax activation function is a differentiable approximation that enables gradient-based optimization through continuous relaxation of discrete action spaces in multi-agent reinforcement learning, allowing agents to learn and improve decision-making in complex environments with discrete choices. If you would like to customise the mlp output activation function, you can define it within the network configuration using the key “output_activation”. User definition for the output activation is however, unnecessary, as the algorithm will select the appropriate function given the environments action space.

Agent Masking

If you need to take actions from agents at different timesteps, you can use agent masking to only retrieve new actions for certain agents whilst providing ‘environment defined actions’ for other agents, which act as a nominal action for such “masked” agents to take. These nominal actions should be returned as part of the `info` dictionary. Following the PettingZoo API we recommend the `info` dictionary to be keyed by the agents, with `env_defined_actions` defined as follows:

```
info = {'speaker_0': {'env_defined_actions':  None},
        'listener_0': {'env_defined_actions': np.array([0,0,0,0,0])}
```

latest

For agents that you wish not to be masked, the `env_defined_actions` should be set to `None`. If your environment has discrete action spaces then provide 'env_defined_actions' as a numpy array with a single value. For example, an action space of type `Discrete(5)` may have an `env_defined_action` of `np.array([4])`. For an environment with continuous actions spaces (e.g. `Box(0, 1, (5,))`) then the shape of the array should be the size of the action space (`np.array([0.5, 0.5, 0.5, 0.5, 0.5])`). Agent masking is handled automatically by the AgileRL multi-agent training function, but can be implemented in a custom loop as follows:

```
env_defined_actions = {agent: info[agent]["env_defined_actions"] for agent in env.agent
state, info = env.reset()  # or: next_state, reward, done, truncation, info = env.step
action, _ = agent.get_action(state, env_defined_actions=env_defined_actions)
```

▶ Example Training Loop

Neural Network Configuration

To configure the architecture of the network's encoder / head, pass a kwargs dict to the MATD3 `net_config` field. Full arguments can be found in the documentation of EvolvableMLP, EvolvableCNN, and EvolvableMultiInput.

For discrete / vector observations:

```
NET_CONFIG = {
      "encoder_config": {'hidden_size': [32, 32]},  # Network head hidden size
      "head_config": {'hidden_size': [32]}      # Network head hidden size
  }
```

For image observations:

```
NET_CONFIG = {
    "encoder_config": {
      'channel_size': [32, 32], # CNN channel size
      'kernel_size': [8, 4],   # CNN kernel size
      'stride_size': [4, 2],   # CNN stride size
    },
    "head_config": {'hidden_size': [32]}  # Network head hidden size
  }
```

For dictionary / tuple observations containing any combination of image, discrete, and vector observations:

```
CNN_CONFIG = {
    "channel_size": [32, 32], # CNN channel size
    "kernel_size": [8, 4],   # CNN kernel size
    "stride_size": [4, 2],   # CNN stride size
}

NET_CONFIG = {
    "encoder_config": {
      "latent_dim": 32,
      # Config for nested EvolvableCNN objects
      "cnn_config": CNN_CONFIG,
      # Config for nested EvolvableMLP objects
      "mlp_config": {
          "hidden_size": [32, 32]
      },
      "vector_space_mlp": True # Process vector observations with an MLP
    },
    "head_config": {'hidden_size': [32]}  # Network head hidden size
  }
```

```
# Create MATD3 agent
agent = MATD3(
  observation_spaces=observation_spaces,
  action_spaces=action_spaces,
  agent_ids=agent_ids,
  net_config=NET_CONFIG
  )
```

Evolutionary Hyperparameter Optimization

AgileRL allows for efficient hyperparameter optimization during training to provide state-of-the-art results in a fraction of the time. For more information on how this is done, please refer to the Evolutionary Hyperparameter Optimization documentation.

Saving and Loading Agents

To save an agent, use the `save_checkpoint` method:

```
from agilerl.algorithms.matd3 import MATD3

# Create MATD3 agent
agent = MATD3(
  observation_spaces=observation_spaces,
  action_spaces=action_spaces,
  agent_ids=agent_ids,
  net_config=NET_CONFIG
  )

checkpoint_path = "path/to/checkpoint"
agent.save_checkpoint(checkpoint_path)
```

To load a saved agent, use the `load` method:

```
from agilerl.algorithms.matd3 import MATD3

checkpoint_path = "path/to/checkpoint"
agent = MATD3.load(checkpoint_path)
```

Parameters

class agilerl.algorithms.matd3.**MATD3**(*args, **kwargs)

Multi-Agent Twin Delayed Deep Deterministic Policy Gradient (MATD3) algorithm.

Paper: https://arxiv.org/abs/1910.01465

PARAMETERS:

- **observation_spaces** (*Union[list[spaces.Space], spaces.Dict]*) – Observation space for each agent
- **action_spaces** (*Union[list[spaces.Space], spaces.Dict]*) – Action space for each agent
- **agent_ids** (*Optional[list[str]], optional*) – Agent ID for each agent
- **O_U_noise** (*bool, optional*) – Use Ornstein Uhlenbeck action noise for exploration. If False, uses Gaussian noise. Defaults to True
- **expl_noise** (*float, optional*) – Scale for Ornstein Uhlenbeck action noise, or standard deviation for Gaussian exploration noise
- **vect_noise_dim** (*int, optional*) – Vectorization dimension of environment for action noise, defaults to 1
- **mean_noise** (*float, optional*) – Mean of exploration noise, defaults to 0.0
- **theta** (*float, optional*) – Rate of mean reversion in Ornstein Uhlenbeck action noise, defaults to 0.15
- **dt** (*float, optional*) – Timestep for Ornstein Uhlenbeck action noise update, defaults to 1e-2
- **index** (*int, optional*) – Index to keep track of object instance during tournament selection and mutation, defaults to 0
- **hp_config** (*HyperparameterConfig, optional*) – RL hyperparameter mutation configuration, defaults to None, whereby algorithm mutations are disabled.
- **policy_freq** (*int, optional*) – Policy update frequency, defaults to 2
- **net_config** (*Optional[dict[str, Any]], optional*) – Network configuration, defaults to None
- **batch_size** (*int, optional*) – Size of batched sample from replay buffer for learning, defaults to 64
- **lr_actor** (*float, optional*) – Learning rate for actor optimizer, defaults to 0.001
- **lr_critic** (*float, optional*) – Learning rate for critic optimizer, defaults to 0.01
- **learn_step** (*int, optional*) – Learning frequency, defaults to 5
- **gamma** (*float, optional*) – Discount factor, defaults to 0.95
- **tau** (*float, optional*) – For soft update of target network parameters, defaults to 0.01
- **normalize_images** (*bool, optional*) – Normalize image observations, defaults to True
- **mut** (*Optional[str], optional*) – Most recent mutation to agent, defaults to None
- **actor_networks** (*Optional[ModuleDict], optional*) – List of custom actor networks, defaults to None
- **critic_networks** (*Optional[list[ModuleDict]], optional*) – List containing two lists of custom critic networks, defaults to None
- **device** (*str, optional*) – Device for accelerated computing, ‘cpu’ or ‘cu ‘cpu’

- **accelerator** (*accelerate.Accelerator(), optional*) – Accelerator for distributed computing, defaults to None
- **torch_compiler** (*Optional[str], optional*) – The torch compile mode 'default', 'reduce-overhead' or 'max-autotune', defaults to None
- **wrap** (*bool, optional*) – Wrap models for distributed training upon creation, defaults to True

action_noise(agent_id: str) → Tensor

Create action noise for exploration, either Ornstein Uhlenbeck or from a normal distribution.

PARAMETERS:

agent_id (*str*) – Agent ID for action dims

RETURNS:

Action noise

RETURN TYPE:

torch.Tensor

assemble_grouped_outputs(agent_outputs: dict[str, ndarray], vect_dim: int) → dict[str, ndarray]

Assembles individual agent outputs into batched outputs for shared policies.

PARAMETERS:

- **agent_outputs** (*dict[str, np.ndarray]*) – Dictionary with individual agent outputs, e.g. {'agent_0': 4, 'agent_1': 7, 'agent_2': 8}
- **vect_dim** (*int*) – Vectorization dimension size, i.e. number of vect envs

RETURNS:

Assembled dictionary with the form {'agent': [4, 7, 8]}

RETURN TYPE:

dict[str, np.ndarray]

assemble_shared_inputs(experience: dict[str, ndarray | dict[str, ndarray] | tuple[ndarray, ...] | Tensor | TensorDict | tuple[Tensor, ...] | dict[str, Tensor] | Number | list[ReasoningPrompts]] | tuple[ndarray | dict[str, ndarray] | tuple[ndarray, ...] | Tensor | TensorDict | tuple[Tensor, ...] | dict[str, Tensor] | Number | list[ReasoningPrompts], ...]) → dict[str, ndarray | dict[str, ndarray] | tuple[ndarray, ...] | Tensor | TensorDict | tuple[Tensor, ...] | dict[str, Tensor] | Number | list[ReasoningPrompts]] | tuple[ndarray | dict[str, ndarray] | tuple[ndarray, ...] | Tensor | TensorDict | tuple[Tensor, ...] | dict[str, Tensor] | Number | list[ReasoningPrompts], ...]

Preprocesses inputs by constructing dictionaries by shared agents.

PARAMETERS:

experience (*ExperiencesType*) – experience to reshape from environment

RETURNS:

Preprocessed inputs

RETURN TYPE:

ExperiencesType

```
build_net_config(net_config: dict[str, dict[str, Any] | Any] | None = None,
    flatten: bool = True, return_encoders: bool = False) → dict[str,
    dict[str, Any] | Any] | tuple[dict[str, dict[str, Any] | Any], dict[str,
    dict[str, dict[str, Any] | Any]]]
```

Extract an appropriate net config for each sub-agent from the passed net config dictionary. If grouped_agents is True, the net config will be built for the grouped agents i.e. through their common prefix in their agent_id, whenever the passed net config is None.

> Note
>
> If return_encoders is True, we return the encoder configs for each sub-agent. The only exception is for MLPs, where we only return the deepest architecture found. This is useful for algorithms with shared critics that process the observations of all agents, and therefore use an *EvolvableMultiInput* module to process the observations of all agents (assigning an encoder to each sub-agent and, optionally, a single *EvolvableMLP* to process the concatenated vector observations).

PARAMETERS:

- **net_config** (*Optional[NetConfigType]*) – Net config dictionary
- **flatten** (*bool, optional*) – Whether to return a net config for each possible sub-agent, even in grouped settings.
- **return_encoders** (*bool, optional*) – Whether to return the encoder configs for each sub-agent. Defaults to False.

RETURNS:

Net config dictionary for each sub-agent

RETURN TYPE:

NetConfigType

```
clone(index: int | None = None, wrap: bool = True) → SelfEvolvableAlgorithm
```

Creates a clone of the algorithm.

PARAMETERS:

- **index** (*Optional[int], optional*) – The index of the clone, defaults to None
- **wrap** (*bool, optional*) – If True, wrap the models in the clone with the accelerator, defaults to False

RETURNS:

A clone of the algorithm

RETURN TYPE:

EvolvableAlgorithm

```
static copy_attributes(agent: SelfEvolvableAlgorithm, clone:
    SelfEvolvableAlgorithm) → SelfEvolvableAlgorithm
```

Copies the non-evolvable attributes of the algorithm to a clone.

PARAMETERS:

clone (*SelfEvolvableAlgorithm*) – The clone of the algorithm.

RETURNS:

The clone of the algorithm.

RETURN TYPE:

SelfEvolvableAlgorithm

```
disassemble_grouped_outputs(group_outputs: dict[str, ndarray], vect_dim: int,
    grouped_agents: dict[str, list[str]]) → dict[str, ndarray]
```

Disassembles batched output by shared policies into their grouped agents' outputs.

> Note
>
> This assumes that for any given sub-agent the termination condition is deterministic, i.e. any given agent will always terminate at the same timestep in different vectorized environments.

PARAMETERS:

- **group_outputs** (*dict[str, np.ndarray]*) – Dictionary to be disassembled, has the form {'agent': [4, 7, 8]}
- **vect_dim** (*int*) – Vectorization dimension size, i.e. number of vect envs
- **grouped_agents** (*dict[str, list[str]]*) – Dictionary of grouped agent IDs

RETURNS:

Assembled dictionary, e.g. {'agent_0': 4, 'agent_1': 7, 'agent_2': 8}

RETURN TYPE:

dict[str, np.ndarray]

```
evolvable_attributes(networks_only: bool = False) → dict[str,
    | ModuleDict | Optimizer | dict[str, Optimizer] | OptimizerWrapper]
```

Returns the attributes related to the evolvable networks in the algorithm. Includes attributes that are either EvolvableModule or ModuleDict objects, as well as the optimizers associated with the networks.

PARAMETERS:

networks_only (*bool, optional*) – If True, only include evolvable networks, defaults to False

RETURNS:

A dictionary of network attributes.

RETURN TYPE:

dict[str, Any]

```
extract_action_masks(infos: dict[str, dict[str, Any]]) → dict[str, ndarray]
```

Extract action masks from info dictionary

PARAMETERS:

infos (*dict[str, dict[...]]*) – Info dict

RETURNS:

Action masks

RETURN TYPE:

dict[str, np.ndarray]

```
extract_agent_masks(infos: dict[str, dict[str, Any]] | None = None) →
    tuple[dict[str, ndarray], dict[str, ndarray]]
```

Extract env_defined_actions from info dictionary and determine agent masks

PARAMETERS:

infos (*dict[str, dict[...]]*) – Info dict

RETURNS:

Env defined actions and agent masks

RETURN TYPE:

tuple[ArrayDict, ArrayDict]

```
get_action(obs: dict[str, ndarray | dict[str, ndarray] | tuple[ndarray, ...]
    | Tensor | TensorDict | tuple[Tensor, ...] | dict[str, Tensor] | Number
    | list[ReasoningPrompts]], infos: dict[str, dict[str, Any]] | None =
    None) → tuple[dict[str, ndarray], dict[str, ndarray]]
```

latest

Returns the next action to take in the environment. Epsilon is the probability of taking a random action, used for exploration. For epsilon-greedy behaviour, set epsilon to 0.

PARAMETERS:

- **obs** (*dict[str, numpy.Array]*) – Environment observations: {'agent_0': state_dim_0, …, 'agent_n': state_dim_n}
- **infos** (*dict[str, dict[...]], optional*) – Information dictionary from environment, defaults to None

RETURNS:

Processed actions for each agent, raw actions for each agent

RETURN TYPE:

tuple[dict[str, np.ndarray], dict[str, np.ndarray]]

```
static get_action_dim(action_space: Box | Discrete | MultiDiscrete | Dict |
    Tuple | MultiBinary | list[Box | Discrete | MultiDiscrete | Dict | Tuple
    | MultiBinary]) → tuple[int, ...]
```

Returns the dimension of the action space as it pertains to the underlying networks (i.e. the output size of the networks).

PARAMETERS:

action_space (*spaces.Space or list[spaces.Space].*) – The action space of the environment.

RETURNS:

The dimension of the action space.

RETURN TYPE:

int.

```
get_group_id(agent_id: str) → str
```

Get the group ID for an agent.

PARAMETERS:

agent_id (*str*) – The agent ID

RETURNS:

The group ID

```
get_lr_names() → list[str]
```

Returns the learning rates of the algorithm.

```
get_policy() → EvolvableModule
```

Returns the policy network of the algorithm.

```
get_setup() → MultiAgentSetup
```

Get the type of multi-agent setup, as determined by the observation spaces of the agents. By having the 'same' observation space, we mean that the spaces are analogous, i.e. we can use the same *EvolvableModule* to process their observations.

1. HOMOGENEOUS: All agents have the same observation space.
2. MIXED: Agents can be grouped by their observation spaces.
3. HETEROGENEOUS: All agents have different observation spaces.

RETURNS:

The type of multi-agent setup.

RETURN TYPE:

MultiAgentSetup

```
static get_state_dim(observation_space: Box | Discrete | MultiDiscrete | Dict
    | Tuple | MultiBinary | list[Box | Discrete | MultiDiscrete | Dict |
    Tuple | MultiBinary]) → tuple[int, ...]
```

Returns the dimension of the state space as it pertains to the underlying networks (i.e. the input size of the networks).

PARAMETERS:

observation_space (*spaces.Space or list[spaces.Space].*) – The observation space of the environment.

RETURNS:

The dimension of the state space.

RETURN TYPE:

tuple[int, …].

```
has_grouped_agents() → bool
```

Whether the algorithm contains groups of agents assigned to the same policy for centralized execution.

RETURN TYPE:

bool

```
property index: int
```

Returns the index of the algorithm.

```
static inspect_attributes(agent: SelfEvolvableAlgorithm, input_args_only:
    bool = False) → dict[str, Any]
```

latest

Inspect and retrieve the attributes of the current object, excluding attributes related to the underlying evolvable networks (i.e. *EvolvableModule*, *torch.optim.Optimizer*) and with an option to include only the attributes that are input arguments to the constructor.

PARAMETERS:

input_args_only (*bool*) – If True, only include attributes that are input arguments to the constructor. Defaults to False.

RETURNS:

A dictionary of attribute names and their values.

RETURN TYPE:

dict[str, Any]

```
learn(experiences: tuple[dict[str, Tensor], ...]) → dict[str, float]
```

Updates agent network parameters to learn from experiences.

PARAMETERS:

experience (*tuple[dict[str, torch.Tensor]]*) – Tuple of dictionaries containing batched states, actions, rewards, next_states, dones in that order for each individual agent.

RETURNS:

Losses for each agent

RETURN TYPE:

dict[str, float]

```
learn_individual(agent_id: str, stacked_actions: Tensor,
    stacked_next_actions: Tensor, states: dict[str, Tensor], next_states:
    dict[str, Tensor], actions: dict[str, Tensor], rewards: dict[str,
    Tensor], dones: dict[str, Tensor]) → tuple[float | None, float]
```

Inner call to each agent for the learning/algo training steps, up until the soft updates. Applies all forward/backward props.

PARAMETERS:

- **agent_id** (*str*) – ID of the agent
- **stacked_actions** (*Optional[torch.Tensor]*) – Stacked actions tensor for CNN architecture
- **stacked_next_actions** (*Optional[torch.Tensor]*) – Stacked next actions tensor for CNN architecture
- **states** (*dict[str, torch.Tensor]*) – Dictionary of current states for each agent
- **actions** (*dict[str, torch.Tensor]*) – Dictionary of actions taken by each agent
- **rewards** (*dict[str, torch.Tensor]*) – Dictionary of rewards received by each agent
- **dones** (*dict[str, torch.Tensor]*) – Dictionary of done flags for each agent

RETURNS:

Tuple containing actor loss (if applicable) and critic loss

RETURN TYPE:

tuple[Optional[float], float]

```
classmethod load(path: str, device: str | device = 'cpu', accelerator:
    Accelerator | None = None) → SelfEvolvableAlgorithm
```

Loads an algorithm from a checkpoint.

PARAMETERS:

- **path** (*string*) – Location to load checkpoint from.
- **device** (*str, optional*) – Device to load the algorithm on, defaults to ‘cpu’
- **accelerator** (*Optional[Accelerator], optional*) – Accelerator object for distributed computing, defaults to None

RETURNS:

An instance of the algorithm

RETURN TYPE:

RLAlgorithm

```
load_checkpoint(path: str) → None
```

Loads saved agent properties and network weights from checkpoint.

PARAMETERS:

path (*string*) – Location to load checkpoint from

```
property mut: Any
```

Returns the mutation object of the algorithm.


```
mutation_hook() → None
```

Executes the hooks registered with the algorithm.

classmethod **population**(size: int, observation_space: Box | Discrete | MultiDiscrete | Dict | Tuple | MultiBinary | list[Box | Discrete | MultiDiscrete | Dict | Tuple | MultiBinary], action_space: Box | Discrete | MultiDiscrete | Dict | Tuple | MultiBinary | list[Box | Discrete | MultiDiscrete | Dict | Tuple | MultiBinary], wrapper_cls: type[SelfAgentWrapper] | None = None, wrapper_kwargs: dict[str, Any] = {}, **kwargs) → list[SelfEvolvableAlgorithm | SelfAgentWrapper]

Creates a population of algorithms.

PARAMETERS:

size (*int.*) – The size of the population.

RETURNS:

A list of algorithms.

RETURN TYPE:

list[SelfEvolvableAlgorithm].

preprocess_observation(observation: ndarray | dict[str, ndarray] | tuple[ndarray, ...] | Tensor | TensorDict | tuple[Tensor, ...] | dict[str, Tensor] | Number | list[ReasoningPrompts]) → dict[str, Tensor | TensorDict | tuple[Tensor, ...] | dict[str, Tensor]]

Preprocesses observations for forward pass through neural network.

PARAMETERS:

observations (*numpy.ndarray[float] or dict[str, numpy.ndarray[float]]*) – Observations of environment

RETURNS:

Preprocessed observations

RETURN TYPE:

torch.Tensor[float] or dict[str, torch.Tensor[float]] or tuple[torch.Tensor[float], …]

process_infos(infos: dict[str, dict[str, Any]] | None = None) → tuple[dict[str, ndarray], dict[str, ndarray], dict[str, ndarray]]

Process the information, extract env_defined_actions, action_masks and agent_masks

PARAMETERS:

infos (*dict[str, dict[...]]*) – Info dict

RETURNS:

Action masks, env defined actions, agent masks

RETURN TYPE:

tuple[ArrayDict, ArrayDict, ArrayDict]

recompile() → None

Recompiles the evolvable modules in the algorithm with the specified torch compiler.

register_mutation_hook(hook: Callable) → None

Registers a hook to be executed after a mutation is performed on the algorithm.

PARAMETERS:

hook (*Callable*) – The hook to be executed after mutation.

register_network_group(group: NetworkGroup) → None

Sets the evaluation network for the algorithm.

PARAMETERS:

name (*str*) – The name of the evaluation network.

reinit_optimizers(optimizer: OptimizerConfig | None = None) → None

Reinitialize the optimizers of an algorithm. If no optimizer is passed, all optimizers are reinitialized.

PARAMETERS:

optimizer (*Optional[OptimizerConfig], optional*) – The optimizer to reinitialize, defaults to None, in which case all optimizers are reinitialized.

reset_action_noise(indices: list[int]) → None

Reset action noise.

PARAMETERS:

indices (*list[int]*) – List of indices to reset noise for

save_checkpoint(path: str) → None

Saves a checkpoint of agent properties and network weights to path.

PARAMETERS:

path (*string*) – Location to save checkpoint at

set_training_mode(training: bool) → None

Sets the training mode of the algorithm.

PARAMETERS:

training (*bool*) – If True, set the algorithm to training mode.

soft_update(net: Module, target: Module) → None

Soft updates target network.

PARAMETERS:

- **net** (*nn.Module*) – Network to be updated
- **target** (*nn.Module*) – Target network

sum_shared_rewards(rewards: dict[str, ndarray]) → dict[str, n

Sums the rewards for grouped agents

PARAMETERS:

rewards (*dict[str, np.ndarray]*) – Reward dictionary from environment

RETURNS:

Summed rewards dictionary

RETURN TYPE:

dict[str, np.ndarray]

```
test(env: str | ParallelEnv, swap_channels: bool = False, max_steps: int |
    None = None, loop: int = 3, sum_scores: bool = True) → float
```

Returns mean test score of agent in environment with epsilon-greedy policy.

PARAMETERS:

- **env** (*Gym-style environment*) – The environment to be tested in
- **swap_channels** (*bool, optional*) – Swap image channels dimension from last to first [H, W, C] -> [C, H, W], defaults to False
- **max_steps** (*int, optional*) – Maximum number of testing steps, defaults to None
- **loop** (*int, optional*) – Number of testing loops/episodes to complete. The returned score is the mean. Defaults to 3
- **sum_scores** (*book, optional*) – Boolean flag to indicate whether to sum sub-agent scores, defaults to True

```
to_device(*experiences: Tensor | TensorDict | tuple[Tensor, ...] | dict[str,
    Tensor]) → tuple[Tensor | TensorDict | tuple[Tensor, ...] | dict[str,
    Tensor], ...]
```

Moves experiences to the device.

PARAMETERS:

experiences (*tuple[torch.Tensor[float], ...]*) – Experiences to move to device

RETURNS:

Experiences on the device

RETURN TYPE:

tuple[torch.Tensor[float], …]

```
unwrap_models() → None
```

Unwraps the models in the algorithm from the accelerator.

```
wrap_models() → None
```

Wraps the models in the algorithm with the accelerator.