mtrl

Release 1.0.0

Shagun Sodhani, Amy Zhang

Dec 31, 2021

GETTING STARTED

CHAPTER

ONE

MTRL

Multi Task RL Algorithms

1.1 Introduction

MTRL is a library of multi-task reinforcement learning algorithms. It has two main components:

- Components and agents that implement the multi-task RL algorithms.
- Experiment setups that enable training/evaluation on different setups.

Together, these two components enable use of MTRL across different environments and setups.

1.1.1 List of publications & submissions using MTRL (please create a pull request to add the missing entries):

- Learning Robust State Abstractions for Hidden-Parameter Block MDPs

1.1.2 License

- MTRL uses MIT License.
- Terms of Use
- Privacy Policy

1.1.3 Citing MTRL

If you use MTRL in your research, please use the following BibTeX entry:

```
@Misc{Sodhani2021MTRL,
  author =       {Shagun Sodhani, Amy Zhang},
  title =        {MTRL - Multi Task RL Algorithms},
  howpublished = {Github},
  year =         {2021},
  url =          {https://github.com/facebookresearch/mtrl}
}
```

1.2 Setup

- Clone the repository: `git clone git@github.com:facebookresearch/mtrl.git`.
- Install dependencies: `pip install -r requirements/dev.txt`

1.3 Usage

- MTRL supports many different multi-task RL algorithms as described here.
- MTRL supports multi-task environments using MTEnv. These environments include MetaWorld and multi-task variants of DMControl Suite
- Refer the **`tutorial]<https://mtrl.readthedocs.io/en/latest/pages/tutorials/overview.html>`_** to get started with MTRL.

1.4 Documentation

https://mtrl.readthedocs.io

1.5 Contributing to MTRL

There are several ways to contribute to MTRL.

1. Use MTRL in your research.
2. Contribute a new algorithm. The currently supported algorithms are listed here and are looking forward to adding more algorithms.
3. Check out the beginner-friendly issues on GitHub and contribute to fixing those issues.
4. Check out additional details here.

1.6 Community

Ask questions in the chat or github issues:

- Chat
- Issues

CHAPTER

TWO

SUPPORTED ALGORITHMS

Following algorithms are supported:

- Multi-task SAC
- Multi-task SAC with Task Encoder
- Multi-headed SAC
- Distral from *Distral: Robust multitask reinforcement learning* [TBC+17]
- PCGrad from *Gradient surgery for multi-task learning* [YKG+20]
- GradNorm from *Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks* [CBLR18]
- DeepMDP from *DeepMDP: Learning Continuous Latent Space Models for Representation Learning* [GKB+19]
- HiPBMDP from *Multi-Task Reinforcement Learning as a Hidden-Parameter Block MDP* [ZSKP20]
- Soft Modularization from *Multi-Task Reinforcement Learning with Soft Modularization* [YXWW20]
- CARE

Along with the standard SAC components (actor, critic, etc), following components are supported and can be used with the base algorithms in plug-and-play fashion:

- Task Encoder
- State Encoders
  - Attention weighted Mixture of Encoders
  - Gated Mixture of Encoders
  - Ensemble of Encoders
  - FiLM Encoders [PSDV+18]
- Multi-headed actor, critic, value-fuctions
- Modularized actor, critic, value-fuctions based on [YXWW20]

For example, we can train a Multi-task SAC with FiLM encoders or train Multi-headed SAC with gated mixture of encoders with or without using task encoders.

Refer to the **`tutorial <>`_** for more details.

2.1 References

CHAPTER

THREE

SUPPORTED ENVIRONMENTS

MTRL supports multi-task environments using MTEnv. These environments include MetaWorld and multi-task variants of DMControl Suite

CHAPTER

FOUR

OVERVIEW

- Setup the code as described in the setup.
- The entry point for the code is *main.py*.
- Run the code using *PYTHONPATH=. python main.py*. This uses the config in *config/config.yaml*.
- The code uses Hydra framework for composing configs.

CHAPTER

FIVE

RUNNING CODE

The code uses Hydra framework for composing configs and running the code. Let us say that we want to train multi-task SAC on MT10 (from MetaWorld). The command for that will look like:

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
agent=state_sac \
env=metaworld-mt10 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True
```

Let us break this command piece by piece.

- `setup=metaworld` says that we want to use a specific setup called `metaworld`. In multitask RL, different works/environments care about different setups. For example, commonly multitask RL environments use episodic rewards as the metric to optimize while MetaWorld [YQH+20] use `success` as the key metric. Some multitask RL setups evaluate the agent on the same set of environments that it was trained on while HiPBMDP [ZSKP20] evalutes on three sets of unseen environments. We abstract away these details via `setup` parameter. Supported values are listed here. When a setup is selected, the corresponding metrics config is also loaded. By default, we also load optimizers and agent components based on the `setup` value but this can be easily overided (as described in the next step). We can easily add a new setup, by defining a new config or updating the existing configs. For example, to add the `hipbdmp` setup, we added a metrics config and new optimizer configs assuming the values should be different for the new setup. But we do not change the agent configs as the agent implementation does not have to change with the setup, though the user is free to update the agent configs as well.
- `agent=state_sac` says that we want to train SAC using state observations.
  - Other supported agents are listed as top-level yaml files here.
  - Update the config files to change the agent's hyper-parameters.
  - Add a new config file to support a new agent.
  - You would note that we are using the `setup` value in the name of component configs and optimizer configs. This is completely optional and the same effect can be achieved by command line overrides. We opt for using multiple config to reduce the overhead of remembering what values to override when running the code.
- `env=metaworld-mt10` says that we want to train on MT10 environment from MetaWorld.
  - Other supported environments are listed here.
  - New environments can be added by creating a new config file in the directory above.
- `agent.multitask.num_envs=10` sets the number of tasks to be 10.

- `agent.multitask.should_use_disentangled_alpha=True` says that we want to learn a different entropy coefficient for each task.

We can update the previous command to train multi-task multi-headed SAC agent by adding an additional argument `agent.multitask.should_use_multi_head_policy=True` as follows:

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
agent=state_sac \
env=metaworld-mt10 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True \
agent.multitask.should_use_multi_head_policy=True
```

We can control more aspetts of training (like seed, number of training steps, batch size etc) by adding ad the previous command to train multi-task multi-headed SAC agent by adding additional arguments as follows:

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
env=metaworld-mt10 \
agent=state_sac \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True \
agent.multitask.should_use_multi_head_policy=True \
experiment.num_train_steps=2000000 \
setup.seed=1 \
replay_buffer.batch_size=1280
```

- `experiment.num_train_steps=2000000` says that we should train the agent for 2 million steps.
- `setup.seed=1` sets the seed to 1.
- `replay_buffer.batch_size=1280` says that batches, sampled from replay buffer, will contain 1280 transitions.

CHAPTER

SIX

BASELINES

6.1 DMControl

6.1.1 Distral

```
MUJOCO_GL="osmesa" LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/nvidia-opengl/:$LD_
↪LIBRARY_PATH PYTHONPATH=. python3 -u main.py \
setup=hipbmdp \
env=dmcontrol-finger-spin-distribution-v0 \
agent=distral \
setup.seed=1 \
agent.distral_alpha=1.0 \
agent.distral_beta=1.0 \
replay_buffer.batch_size=256
```

6.1.2 DeepMDP

```
MUJOCO_GL="osmesa" LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/nvidia-opengl/:$LD_
↪LIBRARY_PATH PYTHONPATH=. python3 -u main.py \
setup=hipbmdp \
env=dmcontrol-finger-spin-distribution-v0 \
agent=deepmdp \
setup.seed=1 \
replay_buffer.batch_size=256
```

6.1.3 GradNorm

```
MUJOCO_GL="osmesa" LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/nvidia-opengl/:$LD_
↪LIBRARY_PATH PYTHONPATH=. python3 -u main.py \
setup=hipbmdp \
env=dmcontrol-finger-spin-distribution-v0 \
agent=deepmdp \
setup.seed=1 \
replay_buffer.batch_size=256
```

6.1.4 PCGrad

```
MUJOCO_GL="osmesa" LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/nvidia-opengl/:$LD_
↪LIBRARY_PATH PYTHONPATH=. python3 -u main.py \
setup=hipbmdp \
env=dmcontrol-finger-spin-distribution-v0 \
agent=pcgrad_sac \
setup.seed=1 \
replay_buffer.batch_size=256
```

6.1.5 HiP-BMDP

```
MUJOCO_GL="osmesa" LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/nvidia-opengl/:$LD_
↪LIBRARY_PATH PYTHONPATH=. python3 -u main.py \
setup=hipbmdp \
env=dmcontrol-finger-spin-distribution-v0 \
agent=hipbmdp \
agent.multitask.should_use_task_encoder=True \
agent.multitask.should_condition_encoder_on_task_info=True \
agent.multitask.should_concatenate_task_info_with_encoder=True \
setup.seed=1 \
replay_buffer.batch_size=256
```

6.2 Metaworld

6.2.1 Multi-task SAC

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
env=metaworld-mt10 \
agent=state_sac \
experiment.num_eval_episodes=1 \
experiment.num_train_steps=2000000 \
setup.seed=1 \
replay_buffer.batch_size=1280 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True \
agent.encoder.type_to_select=identity \
agent.multitask.should_use_multi_head_policy=False \
agent.multitask.actor_cfg.should_condition_model_on_task_info=False \
agent.multitask.actor_cfg.should_condition_encoder_on_task_info=True \
agent.multitask.actor_cfg.should_concatenate_task_info_with_encoder=True
```

6.2.2 Multi-task Multi-headed SAC

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
env=metaworld-mt10 \
agent=state_sac \
experiment.num_eval_episodes=1 \
experiment.num_train_steps=2000000 \
setup.seed=1 \
replay_buffer.batch_size=1280 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True \
agent.encoder.type_to_select=identity \
agent.multitask.should_use_multi_head_policy=True \
agent.multitask.actor_cfg.should_condition_model_on_task_info=False \
agent.multitask.actor_cfg.should_condition_encoder_on_task_info=False \
agent.multitask.actor_cfg.should_concatenate_task_info_with_encoder=False
```

6.2.3 PCGrad

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
env=metaworld-mt10 \
agent=pcgrad_state_sac \
experiment.num_eval_episodes=1 \
experiment.num_train_steps=2000000 \
setup.seed=1 \
replay_buffer.batch_size=1280 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=False \
agent.multitask.should_use_task_encoder=False \
agent.multitask.actor_cfg.should_condition_encoder_on_task_info=False \
agent.multitask.actor_cfg.should_concatenate_task_info_with_encoder=False \
agent.encoder.type_to_select=identity
```

6.2.4 SoftModularization

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
env=metaworld-mt10 \
agent=state_sac \
experiment.num_eval_episodes=1 \
experiment.num_train_steps=2000000 \
setup.seed=1 \
replay_buffer.batch_size=1280 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True \
agent.multitask.should_use_task_encoder=True \
agent.encoder.type_to_select=feedforward \
agent.multitask.actor_cfg.should_condition_model_on_task_info=True \
agent.multitask.actor_cfg.should_condition_encoder_on_task_info=False \
agent.multitask.actor_cfg.should_concatenate_task_info_with_encoder=False \
agent.multitask.actor_cfg.moe_cfg.should_use=True \
```

(continues on next page)

(continued from previous page)

```
agent.multitask.actor_cfg.moe_cfg.mode=soft_modularization \
agent.multitask.should_use_multi_head_policy=False \
agent.encoder.feedforward.hidden_dim=50 \
agent.encoder.feedforward.num_layers=2 \
agent.encoder.feedforward.feature_dim=50 \
agent.actor.num_layers=4 \
agent.multitask.task_encoder_cfg.model_cfg.pretrained_embedding_cfg.should_use=False
```

6.2.5 SAC + FiLM Encoder

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
env=metaworld-mt10 \
agent=state_sac \
experiment.num_eval_episodes=1 \
experiment.num_train_steps=2000000 \
setup.seed=1 \
replay_buffer.batch_size=1280 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True \
agent.multitask.should_use_task_encoder=True \
agent.encoder.type_to_select=film \
agent.multitask.should_use_multi_head_policy=False \
agent.multitask.task_encoder_cfg.model_cfg.pretrained_embedding_cfg.should_use=True \
agent.multitask.task_encoder_cfg.model_cfg.output_dim=6
```

6.2.6 CARE

```
PYTHONPATH=. python3 -u main.py \
setup=metaworld \
env=metaworld-mt10 \
agent=state_sac \
experiment.num_eval_episodes=1 \
experiment.num_train_steps=2000000 \
setup.seed=1 \
replay_buffer.batch_size=1280 \
agent.multitask.num_envs=10 \
agent.multitask.should_use_disentangled_alpha=True \
agent.multitask.should_use_task_encoder=True \
agent.encoder.type_to_select=moe \
agent.multitask.should_use_multi_head_policy=False \
agent.encoder.moe.task_id_to_encoder_id_cfg.mode=attention \
agent.encoder.moe.num_experts=4 \
agent.multitask.task_encoder_cfg.model_cfg.pretrained_embedding_cfg.should_use=True
```

CHAPTER

SEVEN

MTRL PACKAGE

7.1 Subpackages

7.1.1 mtrl.agent package

Subpackages

mtrl.agent.components package

Submodules

mtrl.agent.components.actor module

Actor component for the agent.

class mtrl.agent.components.actor.**Actor**(*env_obs_shape: List[int]*, *action_shape: List[int]*, *hidden_dim: int*, *num_layers: int*, *log_std_bounds: Tuple[float, float]*, *encoder_cfg: omegaconf.dictconfig.DictConfig*, *multitask_cfg: omegaconf.dictconfig.DictConfig*)

Bases: *mtrl.agent.components.actor.BaseActor*

Actor component for the agent.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **hidden_dim** (*int*) – hidden dimensionality of the actor.
- **num_layers** (*int*) – number of layers in the actor.
- **log_std_bounds** (*Tuple[float, float]*) – bounds to clip log of standard deviation.
- **encoder_cfg** (*ConfigType*) – config for the encoder.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

encode(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*) → torch.Tensor

Encode the input observation.

Parameters

- **mtobs** (MTObs) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach_encoder: bool = False*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Compute the predictions from the actor.

Parameters

- **mtobs** (MTObs) – multi-task observation.
- **detach_encoder** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns

tuple of (mean of the gaussian, sample from the gaussian,

log-probability of the sample, log of standard deviation of the gaussian).

Return type Tuple[TensorType, TensorType, TensorType, TensorType]

get_last_shared_layers() → List[torch.nn.modules.module.Module]

Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

make_model(*action_shape: List[int], hidden_dim: int, num_layers: int, encoder_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig*) → torch.nn.modules.module.Module

Make the model for the actor.

Parameters

- **action_shape** (*List[int]*) –
- **hidden_dim** (*int*) –
- **num_layers** (*int*) –
- **encoder_cfg** (*ConfigType*) –
- **multitask_cfg** (*ConfigType*) –

Returns model for the actor.

Return type ModelType

training: bool

class mtrl.agent.components.actor.**BaseActor**(*env_obs_shape: List[int], action_shape: List[int], encoder_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig, *args, **kwargs*)

Bases: *mtrl.agent.components.base.Component*

Interface for the actor component for the agent.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **encoder_cfg** (*ConfigType*) – config for the encoder.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

encode(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*) → torch.Tensor

Encode the input observation.

Parameters

- **mtobs** (*MTObs*) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach_encoder: bool = False*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Compute the predictions from the actor.

Parameters

- **mtobs** (*MTObs*) – multi-task observation.
- **detach_encoder** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns

tuple of (mean of the gaussian, sample from the gaussian,

log-probability of the sample, log of standard deviation of the gaussian).

Return type Tuple[TensorType, TensorType, TensorType, TensorType]

training: bool

mtrl.agent.components.actor.**check_if_should_use_multi_head_policy**(*multitask_cfg: omegaconf.dictconfig.DictConfig*) → bool

mtrl.agent.components.actor.**check_if_should_use_task_encoder**(*multitask_cfg: omegaconf.dictconfig.DictConfig*) → bool

mtrl.agent.components.base module

Interface for the agent components.

class mtrl.agent.components.base.**Component**
Bases: torch.nn.modules.module.Module

Basic component (for building the agent) that every other component should extend.

It inherits *torch.nn.Module*.

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

get_last_shared_layers() → List[torch.nn.modules.module.Module]
Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

training: bool

mtrl.agent.components.critic module

Critic component for the agent.

class mtrl.agent.components.critic.**Critic**(*env_obs_shape: List[int]*, *action_shape: List[int]*, *hidden_dim: int*, *num_layers: int*, *encoder_cfg: omegaconf.dictconfig.DictConfig*, *multitask_cfg: omegaconf.dictconfig.DictConfig*)
Bases: *mtrl.agent.components.base.Component*

Critic component for the agent.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **hidden_dim** (*int*) – hidden dimensionality of the actor.
- **num_layers** (*int*) – number of layers in the actor.
- **encoder_cfg** (*ConfigType*) – config for the encoder.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

encode(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*) → torch.Tensor
Encode the input observation.

Parameters

- **mtobs** (*MTObs*) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Returns encoding of the observation.

Return type TensorType

forward(*mtobs: mtrl.agent.ds.mt_obs.MTObs*, *action: torch.Tensor*, *detach_encoder: bool = False*) → Tuple[torch.Tensor, torch.Tensor]

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_last_shared_layers() → List[torch.nn.modules.module.Module]

Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

training: bool

class mtrl.agent.components.critic.**QFunction**(*obs_dim: int*, *action_dim: int*, *hidden_dim: int*, *num_layers: int*, *multitask_cfg: omegaconf.dictconfig.DictConfig*)

Bases: *mtrl.agent.components.base.Component*

Q-function implemented as a MLP.

Parameters

- **obs_dim** (*int*) – size of the observation.
- **action_dim** (*int*) – size of the action vector.
- **hidden_dim** (*int*) – size of the hidden layer of the model.
- **num_layers** (*int*) – number of layers in the model.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

build_model(*obs_dim: int*, *action_dim: int*, *hidden_dim: int*, *num_layers: int*, *multitask_cfg: omegaconf.dictconfig.DictConfig*) → torch.nn.modules.module.Module

Build the Q-Function.

Parameters

- **obs_dim** (*int*) – size of the observation.
- **action_dim** (*int*) – size of the action vector.
- **hidden_dim** (*int*) – size of the hidden layer of the trunk.
- **num_layers** (*int*) – number of layers in the model.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

Returns

Return type ModelType

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_last_shared_layers() → List[torch.nn.modules.module.Module]
Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

training: bool

mtrl.agent.components.decoder module

Decoder component for the agent.

class mtrl.agent.components.decoder.**PixelDecoder**(*env_obs_shape: List[int], multitask_cfg: omegaconf.dictconfig.DictConfig, feature_dim: int, num_layers: int = 2, num_filters: int = 32*)
Bases: *mtrl.agent.components.base.Component*

Convolutional decoder for pixels observations.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the observation that the actor gets.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **feature_dim** (*int*) – feature dimension.
- **num_layers** (*int, optional*) – number of layers. Defaults to 2.
- **num_filters** (*int, optional*) – number of conv filters per layer. Defaults to 32.

forward(*h: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_last_shared_layers() → List[torch.nn.modules.module.Module]
Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

training: bool

mtrl.agent.components.decoder.**make_decoder**(*env_obs_shape: List[int]*, *decoder_cfg: omegaconf.dictconfig.DictConfig*, *multi-task_cfg: omegaconf.dictconfig.DictConfig*)

mtrl.agent.components.encoder module

Encoder component for the agent.

class mtrl.agent.components.encoder.**Encoder**(*env_obs_shape: List[int]*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, **args*, ***kwargs*)

Bases: *mtrl.agent.components.base.Component*

Interface for the encoder component of the agent.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the observation that the actor gets.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

copy_conv_weights_from(*source:* mtrl.agent.components.encoder.Encoder) → None
Copy convolutional weights from the *source* encoder.

The no-op implementation should be overridden only by encoders that take convnets.

Parameters **source** (Encoder) – encoder to copy weights from.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*) → torch.Tensor
Encode the input observation.

Parameters

- **mtobs** (MTObs) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

training: bool

class mtrl.agent.components.encoder.**FeedForwardEncoder**(*env_obs_shape: List[int]*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *feature_dim: int*, *num_layers: int*, *hidden_dim: int*, *should_tie_encoders: bool*)

Bases: *mtrl.agent.components.encoder.Encoder*

Feedforward encoder for state observations.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the observation that the actor gets.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **feature_dim** (*int*) – feature dimension.
- **num_layers** (*int, optional*) – number of layers. Defaults to 2.
- **hidden_dim** (*int, optional*) – number of conv filters per layer. Defaults to 32.
- **should_tie_encoders** (*bool*) – should the feed-forward layers be tied.

copy_conv_weights_from(*source:* mtrl.agent.components.encoder.Encoder)
Copy convolutional weights from the *source* encoder.

The no-op implementation should be overridden only by encoders that take convnets.

Parameters **source** (Encoder) – encoder to copy weights from.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*)
Encode the input observation.

Parameters

- **mtobs** (MTObs) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

training: bool

class mtrl.agent.components.encoder.**FiLM**(*env_obs_shape: List[int], multitask_cfg: omegaconf.dictconfig.DictConfig, feature_dim: int, num_layers: int, hidden_dim: int, should_tie_encoders: bool*)
Bases: *mtrl.agent.components.encoder.FeedForwardEncoder*

Feedforward encoder for state observations.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the observation that the actor gets.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **feature_dim** (*int*) – feature dimension.
- **num_layers** (*int, optional*) – number of layers. Defaults to 2.
- **hidden_dim** (*int, optional*) – number of conv filters per layer. Defaults to 32.
- **should_tie_encoders** (*bool*) – should the feed-forward layers be tied.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*)
Encode the input observation.

Parameters

- **mtobs** (MTObs) – multi-task observation.

- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

training: bool

class mtrl.agent.components.encoder.**IdentityEncoder**(*env_obs_shape: List[int], multitask_cfg: omegaconf.dictconfig.DictConfig, feature_dim: int*)

Bases: *mtrl.agent.components.encoder.Encoder*

Identity encoder that does not perform any operations.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the observation that the actor gets.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **feature_dim** (*#*) – feature dimension.
- **num_layers** (*#*) – number of layers. Defaults to 2.
- **num_filters** (*#*) – number of conv filters per layer. Defaults to 32.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*)

Encode the input observation.

Parameters

- **mtobs** (*MTObs*) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

training: bool

class mtrl.agent.components.encoder.**MixtureofExpertsEncoder**(*env_obs_shape: List[int], multitask_cfg: omegaconf.dictconfig.DictConfig, encoder_cfg: omegaconf.dictconfig.DictConfig, task_id_to_encoder_id_cfg: omegaconf.dictconfig.DictConfig, num_experts: int*)

Bases: *mtrl.agent.components.encoder.Encoder*

Mixture of Experts based encoder.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the observation that the actor gets.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **encoder_cfg** (*ConfigType*) – config for the experts in the mixture.
- **task_id_to_encoder_id_cfg** (*ConfigType*) – mapping between the tasks and the encoders.
- **num_experts** (*int*) – number of experts.

copy_conv_weights_from(*source*)
Copy convolutional weights from the *source* encoder.

The no-op implementation should be overridden only by encoders that take convnets.

Parameters **source** (Encoder) – encoder to copy weights from.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*)
Encode the input observation.

Parameters

- **mtobs** (MTObs) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

training: bool

class mtrl.agent.components.encoder.**PixelEncoder**(*env_obs_shape: List[int], multitask_cfg: omegaconf.dictconfig.DictConfig, feature_dim: int, num_layers: int = 2, num_filters: int = 32*)

Bases: *mtrl.agent.components.encoder.Encoder*

Convolutional encoder for pixels observations.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the observation that the actor gets.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **feature_dim** (*int*) – feature dimension.
- **num_layers** (*int, optional*) – number of layers. Defaults to 2.
- **num_filters** (*int, optional*) – number of conv filters per layer. Defaults to 32.

copy_conv_weights_from(*source:* mtrl.agent.components.encoder.Encoder)
Copy convolutional weights from the *source* encoder.

The no-op implementation should be overridden only by encoders that take convnets.

Parameters **source** (Encoder) – encoder to copy weights from.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs, *detach: bool = False*)
Encode the input observation.

Parameters

- **mtobs** (MTObs) – multi-task observation.
- **detach** (*bool, optional*) – should detach the observation encoding from the computation graph. Defaults to False.

Raises **NotImplementedError** –

Returns encoding of the observation.

Return type TensorType

forward_conv(*env_obs: torch.Tensor*) → torch.Tensor
Encode the environment observation using the convolutional layers.

Parameters **env_obs** (*TensorType*) – observation from the environment.

Returns encoding of the observation.

Return type TensorType

reparameterize(*mu: torch.Tensor*, *logstd: torch.Tensor*) → torch.Tensor
Reparameterization Trick

Parameters

- **mu** (*TensorType*) – mean of the gaussian.
- **logstd** (*TensorType*) – log of standard deviation of the gaussian.

Returns sample from the gaussian.

Return type TensorType

training: bool

mtrl.agent.components.encoder.**make_encoder**(*env_obs_shape: List[int]*, *encoder_cfg: omegaconf.dictconfig.DictConfig*, *multi-task_cfg: omegaconf.dictconfig.DictConfig*)

mtrl.agent.components.encoder.**tie_weights**(*src*, *trg*)

mtrl.agent.components.hipbmdp_theta module

Implementation of the theta component described in "Multi-Task Reinforcement Learning as a Hidden-Parameter Block MDP" Link: https://arxiv.org/abs/2007.07206

class mtrl.agent.components.hipbmdp_theta.**ThetaModel**(*dim: int*, *output_dim: int*, *num_envs: int*, *train_env_id: List[str]*)

Bases: *mtrl.agent.components.base.Component*

Implementation of the theta component described in "Multi-Task Reinforcement Learning as a Hidden-Parameter Block MDP" Link: https://arxiv.org/abs/2007.07206

Parameters

- **dim** (*int*) – input dimension.
- **output_dim** (*int*) – output dimension.
- **num_envs** (*int*) – number of environments.
- **train_env_id** (*List[str]*) – index of environments corresponding to training tasks. Some strategies (for sampling theta) need this information.

forward(*env_index: torch.Tensor*, *theta_sampling_strategy: str*, *modes: List[str]*) → torch.Tensor
Sample theta.

Following strategies are supported:

- ***embedding* - use an embedding layer and index into it using** task index. This is the default strategy and used during training and testing on in-distribution environments.
- *zero* - set theta as tensor of zeros.
- ***mean* - use an embedding layer and set theta as the mean of** all the embeddings.
- ***mean_train* - use an embedding layer and set theta as the mean of** all the embeddings that were trained.

Parameters

- **env_index** (*TensorType*) –
- **theta_sampling_strategy** (*str*) – strategy to sample theta.
- **modes** (*List[str]*) – List of train/eval/... modes.

Returns sampled theta.

Return type TensorType

training: bool

class mtrl.agent.components.hipbmdp_theta.**ThetaSamplingStrategy**(*value*)
Bases: enum.Enum

Different strategies for sampling theta values.

- ***embedding* - use an embedding layer and index into it using** task index.
- *zero* - set theta as tensor of zeros.
- ***mean* - use an embedding layer and set theta as the mean of** all the embeddings.
- ***mean_train* - use an embedding layer and set theta as the mean of** all the embeddings that were trained.

EMBEDDING = 'embedding'

MEAN = 'mean'

MEAN_TRAIN = 'mean_train'

ZERO = 'zero'

mtrl.agent.components.moe_layer module

Layers for parallelizing computation with mixture of experts.

A mixture of experts(models) can be easily simulated by maintaining a list of models and iterating over them. However, this can be slow in practice. We provide some additional modules which makes it easier to create mixture of experts without slowing down training/inference.

class mtrl.agent.components.moe_layer.**AttentionBasedExperts**(*num_tasks: int*, *num_experts: int*, *embedding_dim: int*, *hidden_dim: int*, *num_layers: int*, *temperature: bool*, *should_use_soft_attention: bool*, *task_encoder_cfg: omegaconf.dictconfig.DictConfig*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *topk: Optional[int] = None*)

Bases: *mtrl.agent.components.moe_layer.MixtureOfExperts*

Class for interfacing with a mixture of experts.

Parameters **multitask_cfg** (*ConfigType*) – config for multitask training.

forward(*task_info:* mtrl.agent.ds.task_info.TaskInfo) → torch.Tensor
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 mtrl.agent.components.moe_layer.**ClusterOfExperts**(*num_tasks: int*, *num_experts: int*, *num_eval_episodes: int*, *batch_size: int*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *env_name: str*, *task_description: Dict[str, str]*, *ordered_task_list: List[str]*, *mapping_cfg: omegaconf.dictconfig.DictConfig*)

Bases: *mtrl.agent.components.moe_layer.MixtureOfExperts*

Map the ith task to a subset (cluster) of experts.

Parameters

- **num_tasks** (*int*) – number of tasks.
- **num_experts** (*int*) – number of experts in the mixture of experts.
- **num_eval_episodes** (*int*) – number of episodes run during evaluation.
- **batch_size** (*int*) – batch size for update.

- **multitask_cfg** (*ConfigType*) – config for multitask training.
- **env_name** (*str*) – name of the environment. This is used with the mapping configuration.
- **task_description** (*Dict[str, str]*) – dictionary mapping task names to descriptions.
- **ordered_task_list** (*List[str]*) – ordered list of tasks. This is needed because the task description is not always ordered.
- **mapping_cfg** (*ConfigType*) – config for mapping the tasks to subset of experts.

training: bool

class mtrl.agent.components.moe_layer.**EnsembleOfExperts**(*num_tasks: int*, *num_experts: int*, *num_eval_episodes: int*, *batch_size: int*, *multitask_cfg: omegaconf.dictconfig.DictConfig*)

Bases: *mtrl.agent.components.moe_layer.MixtureOfExperts*

Ensemble of all the experts.

Parameters

- **num_tasks** (*int*) – number of tasks.
- **num_experts** (*int*) – number of experts in the mixture of experts.
- **num_eval_episodes** (*int*) – number of episodes run during evaluation.
- **batch_size** (*int*) – batch size for update.
- **multitask_cfg** (*ConfigType*) – config for multitask training.

training: bool

class mtrl.agent.components.moe_layer.**FeedForward**(*num_experts: int*, *in_features: int*, *out_features: int*, *num_layers: int*, *hidden_features: int*, *bias: bool = True*)

Bases: torch.nn.modules.module.Module

A feedforward model of mixture of experts layers.

Parameters

- **num_experts** (*int*) – number of experts in the mixture.
- **in_features** (*int*) – size of each input sample for one expert.
- **out_features** (*int*) – size of each output sample for one expert.
- **num_layers** (*int*) – number of layers in the feedforward network.
- **hidden_features** (*int*) – dimensionality of hidden layer in the feedforward network.
- **bias** (*bool, optional*) – if set to False, the layer will not learn an additive bias. Defaults to True.

forward(*x: torch.Tensor*) → torch.Tensor
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 mtrl.agent.components.moe_layer.**Linear**(*num_experts: int*, *in_features: int*, *out_features: int*, *bias: bool = True*)

Bases: `torch.nn.modules.module.Module`

torch.nn.Linear layer extended for use as a mixture of experts.

Parameters

- **num_experts** (*int*) – number of experts in the mixture.
- **in_features** (*int*) – size of each input sample for one expert.
- **out_features** (*int*) – size of each output sample for one expert.
- **bias** (*bool, optional*) – if set to `False`, the layer will not learn an additive bias. Defaults to True.

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

forward(*x: torch.Tensor*) → torch.Tensor

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 mtrl.agent.components.moe_layer.**MaskCache**(*num_tasks: int*, *num_eval_episodes: int*, *batch_size: int*, *task_index_to_mask: torch.Tensor*)

Bases: `object`

In multitask learning, using a mixture of models, different tasks can be mapped to different combination of models. This utility class caches these mappings so that they do not have to be revaluated.

For example, when the model is training over 10 tasks, and the tasks are always ordered, the mapping of task index to encoder indices will be the same and need not be recomputed. We take a very simple approach here: cache using the number of tasks, since in our case, the task ordering during training and evaluation does not change. In more complex cases, a mode (train/eval..) based key could be used.

This gets a little trickier during evaluation. We assume that we are running multiple evaluation episodes (per task) at once. So during evaluation, the agent is inferring over num_tasks*num_eval_episodes at once.

We have to be careful about not caching the mapping during update because neither the task distribution, nor the task ordering, is pre-determined during update. So we explicitly exclude the *batch_size* from the list of keys being cached.

Parameters

- **num_tasks** (*int*) – number of tasks.
- **num_eval_episodes** (*int*) – number of episodes run during evaluation.
- **batch_size** (*int*) – batch size for update.
- **task_index_to_mask** (*TensorType*) – mapping of task index to mask.

get_mask(*task_info:* mtrl.agent.ds.task_info.TaskInfo) → torch.Tensor
Get the mask corresponding to a given task info.

Parameters **task_info** (TaskInfo) –

Returns encoder mask.

Return type TensorType

class mtrl.agent.components.moe_layer.**MixtureOfExperts**(*multitask_cfg: omegaconf.dictconfig.DictConfig*)

Bases: torch.nn.modules.module.Module

Class for interfacing with a mixture of experts.

Parameters **multitask_cfg** (*ConfigType*) – config for multitask training.

forward(*task_info:* mtrl.agent.ds.task_info.TaskInfo) → torch.Tensor
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 mtrl.agent.components.moe_layer.**OneToOneExperts**(*num_tasks: int*, *num_experts: int*, *num_eval_episodes: int*, *batch_size: int*, *multitask_cfg: omegaconf.dictconfig.DictConfig*)

Bases: *mtrl.agent.components.moe_layer.MixtureOfExperts*

Map the output of ith expert with the ith task.

Parameters

- **num_tasks** (*int*) – number of tasks.
- **num_experts** (*int*) – number of experts in the mixture of experts.
- **num_eval_episodes** (*int*) – number of episodes run during evaluation.
- **batch_size** (*int*) – batch size for update.
- **multitask_cfg** (*ConfigType*) – config for multitask training.

mask_cache: ***mtrl.agent.components.moe_layer.MaskCache***

training: **bool**

mtrl.agent.components.reward_decoder module

Reward decoder component for the agent.

class mtrl.agent.components.reward_decoder.**RewardDecoder**(*feature_dim: int*)
Bases: *mtrl.agent.components.base.Component*

Predict reward using the observations.

Parameters **feature_dim** (*int*) – dimension of the feature used to predict the reward.

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_last_shared_layers() → List[torch.nn.modules.module.Module]
Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

training: bool

mtrl.agent.components.scripted_soft_modularization module

mtrl.agent.components.soft_modularization module

Implementation of the soft routing network and MLP described in "Multi-Task Reinforcement Learning with Soft Modularization" Link: https://arxiv.org/abs/2003.13661

class mtrl.agent.components.soft_modularization.**RoutingNetwork**(*in_features: int*, *hidden_features: int*, *num_experts_per_layer: int*, *num_layers: int*)
Bases: *mtrl.agent.components.base.Component*

Class to implement the routing network in 'Multi-Task Reinforcement Learning with Soft Modularization' paper.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs) → torch.Tensor
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 mtrl.agent.components.soft_modularization.**SoftModularizedMLP**(*num_experts: int*, *in_features: int*, *out_features: int*, *num_layers: int*, *hidden_features: int*, *bias: bool = True*)

Bases: *mtrl.agent.components.base.Component*

Class to implement the actor/critic in 'Multi-Task Reinforcement Learning with Soft Modularization' paper. It is similar to layers.FeedForward but allows selection of expert at each layer.

forward(*mtobs:* mtrl.agent.ds.mt_obs.MTObs) → torch.Tensor
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

mtrl.agent.components.task_encoder module

Component to encode the task.

class mtrl.agent.components.task_encoder.**TaskEncoder**(*pretrained_embedding_cfg: omegaconf.dictconfig.DictConfig*, *num_embeddings: int*, *embedding_dim: int*, *hidden_dim: int*, *num_layers: int*, *output_dim: int*)

Bases: *mtrl.agent.components.base.Component*

Encode the task into a vector.

Parameters

- **pretrained_embedding_cfg** (*ConfigType*) – config for using pretrained embeddings.
- **num_embeddings** (*int*) – number of elements in the embedding table. This is used if pretrained embedding is not used.

- **embedding_dim** (*int*) – dimension for the embedding. This is used if pretrained embedding is not used.
- **hidden_dim** (*int*) – dimension of the hidden layer of the trunk.
- **num_layers** (*int*) – number of layers in the trunk.
- **output_dim** (*int*) – output dimension of the task encoder.

forward(*env_index: torch.Tensor*) → torch.Tensor
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

mtrl.agent.components.transition_model module

Transition dynamics for the agent.

class mtrl.agent.components.transition_model.**DeterministicTransitionModel**(*encoder_feature_dim: int*, *action_shape: List[int]*, *layer_width: int*, *multi-task_cfg: omegaconf.dictconfig.DictConf*

Bases: *mtrl.agent.components.transition_model.TransitionModel*

Determinisitc model for predicting the transition dynamics.

Parameters

- **encoder_feature_dim** (*int*) – size of the input feature.
- **action_shape** (*List[int]*) – size of the action vector.
- **layer_width** (*int*) – width for each layer.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, Optional[torch.Tensor]]

Return the mean and standard deviation of the gaussian distribution that the model predicts for the next state.

Parameters **x** (*TensorType*) – input.

Returns [mean of gaussian distribution, sigma of gaussian distribution]

Return type Tuple[TensorType, TensorType]

get_last_shared_layers() → List[torch.nn.modules.module.Module]
Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

sample_prediction(*x: torch.Tensor*) → torch.Tensor
Sample a possible value of next state from the model.

Parameters **x** (*TensorType*) – input.

Returns predicted next state.

Return type TensorType

training: bool

class mtrl.agent.components.transition_model.**ProbabilisticTransitionModel**(*encoder_feature_dim: int*, *action_shape: List[int]*, *layer_width: int*, *multi-task_cfg: omegaconf.dictconfig.DictConf*, *max_sigma: float = 10.0*, *min_sigma: float = 0.0001*)

Bases: *mtrl.agent.components.transition_model.TransitionModel*

Probabilistic model for predicting the transition dynamics.

Parameters

- **encoder_feature_dim** (*int*) – size of the input feature.
- **action_shape** (*List[int]*) – size of the action vector.
- **layer_width** (*int*) – width for each layer.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **max_sigma** (*float, optional*) – maximum value of sigma (of the learned gaussian distribution). Larger values are clipped to this value. Defaults to 1e1.
- **min_sigma** (*float, optional*) – minimum value of sigma (of the learned gaussian distribution). Smaller values are clipped to this value. Defaults to 1e-4.

forward(*x*)

Return the mean and standard deviation of the gaussian distribution that the model predicts for the next state.

Parameters **x** (*TensorType*) – input.

Returns [mean of gaussian distribution, sigma of gaussian distribution]

Return type Tuple[TensorType, TensorType]

get_last_shared_layers() → List[torch.nn.modules.module.Module]
Get the list of last layers (for different sub-components) that are shared across tasks.

This method should be implemented by the subclasses if the component is to be trained with gradnorm algorithm.

Returns list of layers.

Return type List[ModelType]

sample_prediction(*x*)
Sample a possible value of next state from the model.

Parameters **x** (*TensorType*) – input.

Returns predicted next state.

Return type TensorType

training: bool

class mtrl.agent.components.transition_model.**TransitionModel**(*encoder_feature_dim: int, action_shape: List[int], layer_width: int, multi-task_cfg: omega-conf.dictconfig.DictConfig*)

Bases: *mtrl.agent.components.base.Component*

Model for predicting the transition dynamics.

Parameters

- **encoder_feature_dim** (*int*) – size of the input feature.
- **action_shape** (*List[int]*) – size of the action vector.
- **layer_width** (*int*) – width for each layer.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Return the mean and standard deviation of the gaussian distribution that the model predicts for the next state.

Parameters **x** (*TensorType*) – input.

Returns [mean of gaussian distribution, sigma of gaussian distribution]

Return type Tuple[TensorType, TensorType]

sample_prediction(*x: torch.Tensor*) → torch.Tensor
Sample a possible value of next state from the model.

Parameters **x** (*TensorType*) – input.

Returns predicted next state.

Return type TensorType

training: bool

mtrl.agent.components.transition_model.**make_transition_model**(*action_shape: List[int], transition_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig*)

Module contents

mtrl.agent.ds package

Submodules

mtrl.agent.ds.mt_obs module

Datastructure to wrap environment observation, task observation and other task-related information.

class mtrl.agent.ds.mt_obs.**MTObs**(*env_obs: torch.Tensor, task_obs: Optional[torch.Tensor], task_info: Optional[*mtrl.agent.ds.task_info.TaskInfo*]*)

Bases: object

Class to wrap environment observation, task observation and other task-related information.

env_obs

task_info

task_obs

mtrl.agent.ds.task_info module

Datastructure to encapsulate the task-related information.

class mtrl.agent.ds.task_info.**TaskInfo**(*encoding: Union[torch.Tensor, NoneType], compute_grad: bool, env_index: torch.Tensor*)

Bases: object

compute_grad

encoding

env_index

Module contents

Submodules

mtrl.agent.abstract module

Interface for the agent.

class mtrl.agent.abstract.**Agent**(*env_obs_shape: List[int]*, *action_shape: List[int]*, *action_range: Tuple[int, int]*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *device: torch.device*)

Bases: abc.ABC

Abstract agent class that every other agent should extend.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **action_range** (*Tuple[int, int]*) – min and max values for the action vector.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **device** (*torch.device*) – device for the agent.

abstract complete_init(*cfg_to_load_model: omegaconf.dictconfig.DictConfig*) → None

Complete the init process.

The derived classes should implement this to perform different post-processing steps.

Parameters **cfg_to_load_model** (*ConfigType*) – config to load the model.

get_component_name_list_for_checkpointing() → List[Tuple[torch.nn.modules.module.Module, str]]

Get the list of tuples of (model, name) from the agent to checkpoint.

Returns list of tuples of (model, name).

Return type List[Tuple[ModelType, str]]

get_last_shared_layers(*component_name: str*) → Optional[List[torch.nn.modules.module.Module]]

Get the last shared layer for any given component.

Parameters **component_name** (*str*) – given component.

Returns list of layers.

Return type List[ModelType]

get_optimizer_name_list_for_checkpointing() → List[Tuple[torch.optim.optimizer.Optimizer, str]]

Get the list of tuples of (optimizer, name) from the agent to checkpoint.

Returns list of tuples of (optimizer, name).

Return type List[Tuple[OptimizerType, str]]

load(*model_dir: Optional[str]*, *step: Optional[int]*) → None

Load the agent.

Parameters

- **model_dir** (*Optional[str]*) – directory to load the model from.
- **step** (*Optional[int]*) – step for tracking the training of the agent.

load_latest_step(*model_dir: str*) → int
Load the agent using the latest training step.

Parameters **model_dir** (*Optional[str]*) – directory to load the model from.

Returns step for tracking the training of the agent.

Return type int

load_metadata(*model_dir: str*) → Optional[Dict[Any, Any]]
Load the metadata of the agent.

Parameters **model_dir** (*str*) – directory to load the model from.

Returns metadata.

Return type Optional[Dict[Any, Any]]

abstract sample_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*) → numpy.ndarray
Sample the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for sampling the action.

Returns sampled action.

Return type np.ndarray

save(*model_dir: str*, *step: int*, *retain_last_n: int*, *should_save_metadata: bool = True*) → None
Save the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.
- **should_save_metadata** (*bool, optional*) – should training metadata be saved. Defaults to True.

save_components(*model_dir: str*, *step: int*, *retain_last_n: int*) → None
Save the different components of the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.

save_components_or_optimizers(*component_or_optimizer_list: Union[List[Tuple[torch.nn.modules.module.Module, str]], List[Tuple[torch.optim.optimizer.Optimizer, str]]]*, *model_dir: str*, *step: int*, *retain_last_n: int*, *suffix: str = ''*) → None
Save the components and optimizers from the given list.

Parameters

- **component_or_optimizer_list** – (Union[List[Tuple[ComponentType, str]], List[Tuple[OptimizerType, str]]]): list of components and optimizers to save.
- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.
- **suffix** (*str, optional*) – suffix to add at the name of the model before checkpointing. Defaults to "".

save_metadata (*model_dir: str*, *step: int*) → None
Save the metadata.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.

save_optimizers (*model_dir: str*, *step: int*, *retain_last_n: int*) → None
Save the different optimizers of the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.

abstract select_action (*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*) → numpy.ndarray
Select the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for selecting the action.

Returns selected action.

Return type np.ndarray

abstract train (*training: bool = True*) → None
Set the agent in training/evaluation mode

Parameters **training** (*bool, optional*) – should set in training mode. Defaults to True.

abstract update (*replay_buffer: mtrl.replay_buffer.ReplayBuffer*, *logger: mtrl.logger.Logger*, *step: int*, *kwargs_to_compute_gradient: Optional[Dict[str, Any]] = None*, *buffer_index_to_sample: Optional[numpy.ndarray] = None*) → numpy.ndarray
Update the agent.

Parameters

- **replay_buffer** (ReplayBuffer) – replay buffer to sample the data.
- **logger** (Logger) – logger for logging.
- **step** (*int*) – step for tracking the training progress.
- **kwargs_to_compute_gradient** (*Optional[Dict[str, Any]], optional*) – Defaults to None.

- **buffer_index_to_sample** (*Optional[np.ndarray], optional*) – if this parameter is specified, use these indices instead of sampling from the replay buffer. If this is set to *None*, sample from the replay buffer. buffer_index_to_sample Defaults to None.

Returns

index sampled (from the replay buffer) to train the model. If buffer_index_to_sample is not set to None, return buffer_index_to_sample.

Return type np.ndarray

mtrl.agent.deepmdp module

class mtrl.agent.deepmdp.**Agent**(*env_obs_shape: List[int], action_shape: List[int], action_range: Tuple[int, int], device: torch.device, actor_cfg: omegaconf.dictconfig.DictConfig, critic_cfg: omegaconf.dictconfig.DictConfig, decoder_cfg: omegaconf.dictconfig.DictConfig, reward_decoder_cfg: omegaconf.dictconfig.DictConfig, transition_model_cfg: omegaconf.dictconfig.DictConfig, alpha_optimizer_cfg: omegaconf.dictconfig.DictConfig, actor_optimizer_cfg: omegaconf.dictconfig.DictConfig, critic_optimizer_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig, decoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, encoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, reward_decoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, transition_model_optimizer_cfg: omegaconf.dictconfig.DictConfig, discount: float = 0.99, init_temperature: float = 0.01, actor_update_freq: int = 2, critic_tau: float = 0.005, critic_target_update_freq: int = 2, encoder_tau: float = 0.005, loss_reduction: str = 'mean', decoder_update_freq: int = 1, decoder_latent_lambda: float = 0.0, cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None, should_complete_init: bool = True*)

Bases: *mtrl.agent.sac_ae.Agent*

DeepMDP Agent

Abstract agent class that every other agent should extend.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **action_range** (*Tuple[int, int]*) – min and max values for the action vector.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **device** (*torch.device*) – device for the agent.

update_decoder(*batch: mtrl.replay_buffer.ReplayBufferSample, task_info: mtrl.agent.ds.task_info.TaskInfo, logger: mtrl.logger.Logger, step: int, kwargs_to_compute_gradient: Dict[str, Any]*)

Update the decoder component.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** ([Logger]) – logger object.
- **step** (int) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (Dict[str, Any]) –

update_transition_reward_model(*batch: mtrl.replay_buffer.ReplayBufferSample, task_info: mtrl.agent.ds.task_info.TaskInfo, logger: mtrl.logger.Logger, step: int, kwargs_to_compute_gradient: Dict[str, Any]*)

Update the transition model and reward decoder.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** ([Logger]) – logger object.
- **step** (int) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (Dict[str, Any]) –

mtrl.agent.distral module

class mtrl.agent.distral.**Agent**(*env_obs_shape: List[int], action_shape: List[int], action_range: Tuple[int, int], multitask_cfg: omegaconf.dictconfig.DictConfig, device: torch.device, distral_alpha: float, distral_beta: float, agent_index_to_task_index: List[str], distilled_agent_cfg: omegaconf.dictconfig.DictConfig, task_agent_cfg: omegaconf.dictconfig.DictConfig, cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None, should_complete_init: bool = True*)

Bases: mtrl.agent.abstract.Agent

Distral algorithm.

complete_init(*cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig]*) → None

Complete the init process.

The derived classes should implement this to perform different post-processing steps.

Parameters **cfg_to_load_model** (ConfigType) – config to load the model.

load(*model_dir: Optional[str], step: Optional[int]*) → None

Load the agent.

Parameters

- **model_dir** (Optional[str]) – directory to load the model from.
- **step** (Optional[int]) – step for tracking the training of the agent.

load_latest_step(*model_dir: str*) → int

Load the agent using the latest training step.

Parameters **model_dir** (Optional[str]) – directory to load the model from.

Returns step for tracking the training of the agent.

Return type int

sample_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*) → numpy.ndarray
Used during training

save(*model_dir: str*, *step: int*, *retain_last_n: int*, *should_save_metadata: bool = True*) → None
Save the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.
- **should_save_metadata** (*bool, optional*) – should training metadata be saved. Defaults to True.

select_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*) → numpy.ndarray
Used during testing

train(*training: bool = True*) → None
Set the agent in training/evaluation mode

Parameters **training** (*bool, optional*) – should set in training mode. Defaults to True.

update(*replay_buffer: mtrl.replay_buffer.ReplayBuffer*, *logger: mtrl.logger.Logger*, *step: int*, *kwargs_to_compute_gradient: Optional[Dict[str, Any]] = None*, *buffer_index_to_sample: Optional[numpy.ndarray] = None*) → numpy.ndarray
Update the agent.

Parameters

- **replay_buffer** (ReplayBuffer) – replay buffer to sample the data.
- **logger** (Logger) – logger for logging.
- **step** (*int*) – step for tracking the training progress.
- **kwargs_to_compute_gradient** (*Optional[Dict[str, Any]], optional*) – Defaults to None.
- **buffer_index_to_sample** (*Optional[np.ndarray], optional*) – if this parameter is specified, use these indices instead of sampling from the replay buffer. If this is set to *None*, sample from the replay buffer. buffer_index_to_sample Defaults to None.

Returns

index sampled (from the replay buffer) to train the model. If buffer_index_to_sample is not set to None, return buffer_index_to_sample.

Return type np.ndarray

class mtrl.agent.distral.**DistilledAgent**(*env_obs_shape: List[int]*, *action_shape: List[int]*, *action_range: Tuple[int, int]*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *device: torch.device*, *actor_cfg: omegaconf.dictconfig.DictConfig*, *actor_optimizer_cfg: omegaconf.dictconfig.DictConfig*, *cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None*, *should_complete_init: bool = True*)

Bases: *mtrl.agent.abstract.Agent*

Centroid policy for distral

complete_init(*cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig]*) → None
Complete the init process.

The derived classes should implement this to perform different post-processing steps.

Parameters **cfg_to_load_model** (*ConfigType*) – config to load the model.

load(*model_dir: Optional[str]*, *step: Optional[int]*) → None
Load the agent.

Parameters

- **model_dir** (*Optional[str]*) – directory to load the model from.
- **step** (*Optional[int]*) – step for tracking the training of the agent.

load_latest_step(*model_dir: str*) → int
Load the agent using the latest training step.

Parameters **model_dir** (*Optional[str]*) – directory to load the model from.

Returns step for tracking the training of the agent.

Return type int

sample_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*)
Sample the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for sampling the action.

Returns sampled action.

Return type np.ndarray

save(*model_dir: str*, *step: int*, *retain_last_n: int*, *should_save_metadata: bool = True*) → None
Save the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.
- **should_save_metadata** (*bool, optional*) – should training metadata be saved. Defaults to True.

select_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*)
Select the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for selecting the action.

Returns selected action.

Return type np.ndarray

train(*training=True*) → None
Set the agent in training/evaluation mode

Parameters **training** (*bool, optional*) – should set in training mode. Defaults to True.

update(*replay_buffer: mtrl.replay_buffer.ReplayBuffer, logger: mtrl.logger.Logger, step: int, kwargs_to_compute_gradient: Optional[Dict[str, Any]] = None, buffer_index_to_sample: Optional[numpy.ndarray] = None*)
Update the agent.

Parameters

- **replay_buffer** (ReplayBuffer) – replay buffer to sample the data.
- **logger** (Logger) – logger for logging.
- **step** (*int*) – step for tracking the training progress.
- **kwargs_to_compute_gradient** (*Optional[Dict[str, Any]], optional*) – Defaults to None.
- **buffer_index_to_sample** (*Optional[np.ndarray], optional*) – if this parameter is specified, use these indices instead of sampling from the replay buffer. If this is set to *None*, sample from the replay buffer. buffer_index_to_sample Defaults to None.

Returns

index sampled (from the replay buffer) to train the model. If buffer_index_to_sample is not set to None, return buffer_index_to_sample.

Return type np.ndarray

class mtrl.agent.distral.**TaskAgent**(*env_obs_shape: List[int], action_shape: List[int], action_range: Tuple[int, int], multitask_cfg: omegaconf.dictconfig.DictConfig, device: torch.device, agent_cfg: omegaconf.dictconfig.DictConfig, index: int, env_index: int, distral_alpha: float, distral_beta: float, distilled_agent: mtrl.agent.distral.DistilledAgent, cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None, should_complete_init: bool = True*)
Bases: *mtrl.agent.wrapper.Agent*

Wrapper class for the task specific agent

load(*model_dir: Optional[str], step: Optional[int]*) → None
Load the agent.

Parameters

- **model_dir** (*Optional[str]*) – directory to load the model from.
- **step** (*Optional[int]*) – step for tracking the training of the agent.

load_latest_step(*model_dir: str*) → int
Load the agent using the latest training step.

Parameters **model_dir** (*Optional[str]*) – directory to load the model from.

Returns step for tracking the training of the agent.

Return type int

patch_agent() → None
Change some function definitions at runtime.

save(*model_dir: str*, *step: int*, *retain_last_n: int*, *should_save_metadata: bool = True*) → None
Save the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.
- **should_save_metadata** (*bool, optional*) – should training metadata be saved. Defaults to True.

update_actor_and_alpha(*batch: mtrl.replay_buffer.ReplayBufferSample*, *task_info: mtrl.agent.ds.task_info.TaskInfo*, *logger: mtrl.logger.Logger*, *step: int*, *kwargs_to_compute_gradient: Dict[str, Any]*) → None
Update the actor and alpha component.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

mtrl.agent.distral.**gaussian_kld**(*mean1: torch.Tensor*, *logvar1: torch.Tensor*, *mean2: torch.Tensor*, *logvar2: torch.Tensor*) → torch.Tensor
Compute KL divergence between a bunch of univariate Gaussian distributions with the given means and log-variances. ie *KL(N(mean1, logvar1) || N(mean2, logvar2))*

Parameters

- **mean1** (*TensorType*) –
- **logvar1** (*TensorType*) –
- **mean2** (*TensorType*) –
- **logvar2** (*TensorType*) –

Returns [description]

Return type TensorType

mtrl.agent.grad_manipulation module

class mtrl.agent.grad_manipulation.**Agent**(*env_obs_shape: List[int]*, *action_shape: List[int]*, *action_range: Tuple[int, int]*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *agent_cfg: omegaconf.dictconfig.DictConfig*, *device: torch.device*, *cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None*, *should_complete_init: bool = True*)
Bases: *mtrl.agent.wrapper.Agent*

Base Class for Gradient Manipulation Algorithms.

update(*replay_buffer:* mtrl.replay_buffer.ReplayBuffer, *logger:* mtrl.logger.Logger, *step: int*, *kwargs_to_compute_gradient: Optional[Dict[str, Any]] = None*, *buffer_index_to_sample: Optional[numpy.ndarray] = None*) → numpy.ndarray

Update the agent.

Parameters

- **replay_buffer** (ReplayBuffer) – replay buffer to sample the data.
- **logger** (Logger) – logger for logging.
- **step** (*int*) – step for tracking the training progress.
- **kwargs_to_compute_gradient** (*Optional[Dict[str, Any]], optional*) – Defaults to None.
- **buffer_index_to_sample** (*Optional[np.ndarray], optional*) – if this parameter is specified, use these indices instead of sampling from the replay buffer. If this is set to *None*, sample from the replay buffer. buffer_index_to_sample Defaults to None.

Returns

index sampled (from the replay buffer) to train the model. If buffer_index_to_sample is not set to None, return buffer_index_to_sample.

Return type np.ndarray

class mtrl.agent.grad_manipulation.**EnvMetadata**(*env_index: torch.Tensor*, *unique_env_index: torch.Tensor*, *env_index_count: torch.Tensor*)

Bases: object

env_index: torch.Tensor

env_index_count: torch.Tensor

unique_env_index: torch.Tensor

mtrl.agent.gradnorm module

class mtrl.agent.gradnorm.**Agent**(*env_obs_shape: List[int]*, *action_shape: List[int]*, *action_range: Tuple[int, int]*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *agent_cfg: omegaconf.dictconfig.DictConfig*, *device: torch.device*, *cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None*, *should_complete_init: bool = True*)

Bases: *mtrl.agent.grad_manipulation.Agent*

GradNorm algorithm.

mtrl.agent.hipbmdp module

class mtrl.agent.hipbmdp.**Agent**(*env_obs_shape: List[int], action_shape: List[int], action_range: Tuple[int, int], device: torch.device, actor_cfg: omegaconf.dictconfig.DictConfig, critic_cfg: omegaconf.dictconfig.DictConfig, decoder_cfg: omegaconf.dictconfig.DictConfig, reward_decoder_cfg: omegaconf.dictconfig.DictConfig, transition_model_cfg: omegaconf.dictconfig.DictConfig, alpha_optimizer_cfg: omegaconf.dictconfig.DictConfig, actor_optimizer_cfg: omegaconf.dictconfig.DictConfig, critic_optimizer_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig, decoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, encoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, reward_decoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, transition_model_optimizer_cfg: omegaconf.dictconfig.DictConfig, discount: float = 0.99, init_temperature: float = 0.01, actor_update_freq: int = 2, critic_tau: float = 0.005, critic_target_update_freq: int = 2, encoder_tau: float = 0.005, loss_reduction: str = 'mean', decoder_update_freq: int = 1, decoder_latent_lambda: float = 0.0, cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None, should_complete_init: bool = True*)

Bases: *mtrl.agent.deepmdp.Agent*

HiPBMDP Agent

Abstract agent class that every other agent should extend.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **action_range** (*Tuple[int, int]*) – min and max values for the action vector.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **device** (*torch.device*) – device for the agent.

get_task_encoding(*env_index: torch.Tensor, modes: List[str], disable_grad: bool*)

Get the task encoding for the different environments.

Parameters

- **env_index** (*TensorType*) – environment index.
- **modes** (*List[str]*) –
- **disable_grad** (*bool*) – should disable tracking gradient.

Returns task encodings.

Return type TensorType

update_task_encoder(*batch: mtrl.replay_buffer.ReplayBufferSample, task_info: mtrl.agent.ds.task_info.TaskInfo, logger, step, kwargs_to_compute_gradient: Dict[str, Any]*)

Update the task encoder component.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

mtrl.agent.pcgrad module

class mtrl.agent.pcgrad.**Agent**(*env_obs_shape: List[int], action_shape: List[int], action_range: Tuple[int, int], device: torch.device, agent_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig, cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None, should_complete_init: bool = True*)

Bases: *mtrl.agent.grad_manipulation.Agent*

PCGrad algorithm.

get_shuffled_task_indices(*num_tasks: int*) → numpy.ndarray

mtrl.agent.pcgrad.**apply_vector_grad_to_parameters**(*vec: torch.Tensor, parameters: Iterable[torch.Tensor], accumulate: bool = False*)

Apply vector gradients to the parameters

Parameters

- **vec** (*TensorType*) – a single vector represents the gradients of a model.
- **parameters** (*Iterable[TensorType]*) – an iterator of Tensors that are the parameters of a model.

mtrl.agent.sac module

class mtrl.agent.sac.**Agent**(*env_obs_shape: List[int], action_shape: List[int], action_range: Tuple[int, int], device: torch.device, actor_cfg: omegaconf.dictconfig.DictConfig, critic_cfg: omegaconf.dictconfig.DictConfig, alpha_optimizer_cfg: omegaconf.dictconfig.DictConfig, actor_optimizer_cfg: omegaconf.dictconfig.DictConfig, critic_optimizer_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig, discount: float, init_temperature: float, actor_update_freq: int, critic_tau: float, critic_target_update_freq: int, encoder_tau: float, loss_reduction: str = 'mean', cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None, should_complete_init: bool = True*)

Bases: *mtrl.agent.abstract.Agent*

SAC algorithm.

Abstract agent class that every other agent should extend.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **action_range** (*Tuple[int, int]*) – min and max values for the action vector.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **device** (*torch.device*) – device for the agent.

act (*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*, *sample: bool*) → numpy.ndarray
Select/sample the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **mode** (*List[str]*) – mode in which to select the action.
- **sample** (*bool*) – sample (if *True*) or select (if *False*) an action.

Returns selected/sample action.

Return type np.ndarray

complete_init (*cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig]*)
Complete the init process.

The derived classes should implement this to perform different post-processing steps.

Parameters **cfg_to_load_model** (*ConfigType*) – config to load the model.

get_alpha (*env_index: torch.Tensor*) → torch.Tensor
Get the alpha value for the given environments.

Parameters **env_index** (*TensorType*) – environment index.

Returns alpha values.

Return type TensorType

get_last_shared_layers (*component_name: str*) → Optional[List[torch.nn.modules.module.Module]]
Get the last shared layer for any given component.

Parameters **component_name** (*str*) – given component.

Returns list of layers.

Return type List[ModelType]

get_parameters (*name: str*) → List[torch.nn.parameter.Parameter]
Get parameters corresponding to a given component.

Parameters **name** (*str*) – name of the component.

Returns list of parameters.

Return type List[torch.nn.parameter.Parameter]

get_task_encoding (*env_index: torch.Tensor*, *modes: List[str]*, *disable_grad: bool*) → torch.Tensor
Get the task encoding for the different environments.

Parameters

- **env_index** (*TensorType*) – environment index.

- **modes** (*List[str]*) –
- **disable_grad** (*bool*) – should disable tracking gradient.

Returns task encodings.

Return type TensorType

get_task_info(*task_encoding: torch.Tensor*, *component_name: str*, *env_index: torch.Tensor*) → *mtrl.agent.ds.task_info.TaskInfo*

Encode task encoding into task info.

Parameters

- **task_encoding** (*TensorType*) – encoding of the task.
- **component_name** (*str*) – name of the component.
- **env_index** (*TensorType*) – index of the environment.

Returns TaskInfo object.

Return type *TaskInfo*

sample_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*) → numpy.ndarray

Sample the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for sampling the action.

Returns sampled action.

Return type np.ndarray

select_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*) → numpy.ndarray

Select the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for selecting the action.

Returns selected action.

Return type np.ndarray

train(*training: bool = True*) → None

Set the agent in training/evaluation mode

Parameters **training** (*bool, optional*) – should set in training mode. Defaults to True.

update(*replay_buffer:* mtrl.replay_buffer.ReplayBuffer, *logger:* mtrl.logger.Logger, *step: int*, *kwargs_to_compute_gradient: Optional[Dict[str, Any]] = None*, *buffer_index_to_sample: Optional[numpy.ndarray] = None*) → numpy.ndarray

Update the agent.

Parameters

- **replay_buffer** (ReplayBuffer) – replay buffer to sample the data.
- **logger** (Logger) – logger for logging.
- **step** (*int*) – step for tracking the training progress.

- **kwargs_to_compute_gradient** (*Optional[Dict[str, Any]], optional*) – Defaults to None.
- **buffer_index_to_sample** (*Optional[np.ndarray], optional*) – if this parameter is specified, use these indices instead of sampling from the replay buffer. If this is set to *None*, sample from the replay buffer. buffer_index_to_sample Defaults to None.

Returns

index sampled (from the replay buffer) to train the model. If buffer_index_to_sample is not set to None, return buffer_index_to_sample.

Return type np.ndarray

update_actor_and_alpha(*batch:* mtrl.replay_buffer.ReplayBufferSample, *task_info:* mtrl.agent.ds.task_info.TaskInfo, *logger:* mtrl.logger.Logger, *step: int*, *kwargs_to_compute_gradient: Dict[str, Any]*) → None

Update the actor and alpha component.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

update_critic(*batch:* mtrl.replay_buffer.ReplayBufferSample, *task_info:* mtrl.agent.ds.task_info.TaskInfo, *logger:* mtrl.logger.Logger, *step: int*, *kwargs_to_compute_gradient: Dict[str, Any]*) → None

Update the critic component.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

update_decoder(*batch:* mtrl.replay_buffer.ReplayBufferSample, *task_info:* mtrl.agent.ds.task_info.TaskInfo, *logger:* mtrl.logger.Logger, *step: int*, *kwargs_to_compute_gradient: Dict[str, Any]*) → None

Update the decoder component.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

update_task_encoder(*batch: mtrl.replay_buffer.ReplayBufferSample, task_info: mtrl.agent.ds.task_info.TaskInfo, logger: mtrl.logger.Logger, step: int, kwargs_to_compute_gradient: Dict[str, Any]*) → None

Update the task encoder component.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

update_transition_reward_model(*batch: mtrl.replay_buffer.ReplayBufferSample, task_info: mtrl.agent.ds.task_info.TaskInfo, logger: mtrl.logger.Logger, step: int, kwargs_to_compute_gradient: Dict[str, Any]*) → None

Update the transition model and reward decoder.

Parameters

- **batch** (ReplayBufferSample) – batch from the replay buffer.
- **task_info** (TaskInfo) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

mtrl.agent.sac_ae module

class mtrl.agent.sac_ae.**Agent**(*env_obs_shape: List[int], action_shape: List[int], action_range: Tuple[int, int], device: torch.device, actor_cfg: omegaconf.dictconfig.DictConfig, critic_cfg: omegaconf.dictconfig.DictConfig, decoder_cfg: omegaconf.dictconfig.DictConfig, alpha_optimizer_cfg: omegaconf.dictconfig.DictConfig, actor_optimizer_cfg: omegaconf.dictconfig.DictConfig, critic_optimizer_cfg: omegaconf.dictconfig.DictConfig, multitask_cfg: omegaconf.dictconfig.DictConfig, decoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, encoder_optimizer_cfg: omegaconf.dictconfig.DictConfig, discount: float, init_temperature: float, actor_update_freq: int, critic_tau: float, critic_target_update_freq: int, encoder_tau: float, loss_reduction: str = 'mean', decoder_update_freq: int = 1, decoder_latent_lambda: float = 0.0, cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None, should_complete_init: bool = True*)

Bases: *mtrl.agent.sac.Agent*

SAC+AE algorithm.

Abstract agent class that every other agent should extend.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **action_range** (*Tuple[int, int]*) – min and max values for the action vector.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **device** (*torch.device*) – device for the agent.

update_decoder(*batch: mtrl.replay_buffer.ReplayBufferSample, task_info: mtrl.agent.ds.task_info.TaskInfo, logger: mtrl.logger.Logger, step: int, kwargs_to_compute_gradient: Dict[str, Any]*) → None

Update the decoder component.

Parameters

- **batch** (*ReplayBufferSample*) – batch from the replay buffer.
- **task_info** (*TaskInfo*) – task_info object.
- **logger** (*[Logger]*) – logger object.
- **step** (*int*) – step for tracking the training of the agent.
- **kwargs_to_compute_gradient** (*Dict[str, Any]*) –

mtrl.agent.utils module

mtrl.agent.utils.**build_mlp**(*input_dim: int*, *hidden_dim: int*, *output_dim: int*, *num_layers: int*) → torch.nn.modules.module.Module

Utility function to build a mlp model. This assumes all the hidden layers are using the same dimensionality.

Parameters

- **input_dim** (*int*) – input dimension.
- **hidden_dim** (*int*) – dimension of the hidden layers.
- **output_dim** (*int*) – dimension of the output layer.
- **num_layers** (*int*) – number of layers in the mlp.

Returns [description]

Return type ModelType

mtrl.agent.utils.**build_mlp_as_module_list**(*input_dim: int*, *hidden_dim: int*, *output_dim: int*, *num_layers: int*) → torch.nn.modules.module.Module

Utility function to build a module list of layers. This assumes all the hidden layers are using the same dimensionality.

Parameters

- **input_dim** (*int*) – input dimension.
- **hidden_dim** (*int*) – dimension of the hidden layers.
- **output_dim** (*int*) – dimension of the output layer.
- **num_layers** (*int*) – number of layers in the mlp.

Returns [description]

Return type ModelType

class mtrl.agent.utils.**eval_mode**(**models*)
Bases: object

Put the agent in the eval mode

mtrl.agent.utils.**preprocess_obs**(*obs: torch.Tensor*, *bits=5*) → torch.Tensor
Preprocessing image, see https://arxiv.org/abs/1807.03039.

mtrl.agent.utils.**set_seed_everywhere**(*seed: int*) → None
Set seed for reproducibility.

Parameters **seed** (*int*) – seed.

mtrl.agent.utils.**soft_update_params**(*net: torch.nn.modules.module.Module*, *target_net: torch.nn.modules.module.Module*, *tau: float*) → None
Perform soft udpate on the net using target net.

Parameters

- **net** (*[ModelType]*) – model to update.
- **target_net** (*ModelType*) – model to update with.
- **tau** (*float*) – control the extent of update.

mtrl.agent.utils.**weight_init**(*m: torch.nn.modules.module.Module*)
Custom weight init for Conv2D and Linear layers.

mtrl.agent.utils.**weight_init_conv**(*m: torch.nn.modules.module.Module*)

mtrl.agent.utils.**weight_init_linear**(*m: torch.nn.modules.module.Module*)

mtrl.agent.utils.**weight_init_moe_layer**(*m: torch.nn.modules.module.Module*)

mtrl.agent.wrapper module

class mtrl.agent.wrapper.**Agent**(*env_obs_shape: List[int]*, *action_shape: List[int]*, *action_range: Tuple[int, int]*, *multitask_cfg: omegaconf.dictconfig.DictConfig*, *agent_cfg: omegaconf.dictconfig.DictConfig*, *device: torch.device*, *cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig] = None*, *should_complete_init: bool = True*)
Bases: *mtrl.agent.abstract.Agent*

This wrapper agent wraps over the other agents. It is useful for alogorithms like PCGrad and GradNorm that can be used with may policies.

Parameters

- **env_obs_shape** (*List[int]*) – shape of the environment observation that the actor gets.
- **action_shape** (*List[int]*) – shape of the action vector that the actor produces.
- **action_range** (*Tuple[int, int]*) – min and max values for the action vector.
- **multitask_cfg** (*ConfigType*) – config for encoding the multitask knowledge.
- **agent_cfg** (*ConfigType*) – config for the agents that are wrapper over.
- **device** (*torch.device*) – device for the agent.
- **cfg_to_load_model** (*Optional[ConfigType], optional*) – config to load the model from filesystem. Defaults to None.

- **should_complete_init** (*bool, optional*) – should call *complete_init* method. Defaults to True.

complete_init(*cfg_to_load_model: Optional[omegaconf.dictconfig.DictConfig]*)
Complete the init process.

The derived classes should implement this to perform different post-processing steps.

Parameters **cfg_to_load_model** (*ConfigType*) – config to load the model.

get_component_name_list_for_checkpointing() → List[Tuple[torch.nn.modules.module.Module, str]]
Get the list of tuples of (model, name) from the agent to checkpoint.

Returns list of tuples of (model, name).

Return type List[Tuple[ModelType, str]]

get_last_shared_layers(*component_name: str*) → Optional[List[torch.nn.modules.module.Module]]
Get the last shared layer for any given component.

Parameters **component_name** (*str*) – given component.

Returns list of layers.

Return type List[ModelType]

get_optimizer_name_list_for_checkpointing() → List[Tuple[torch.optim.optimizer.Optimizer, str]]
Get the list of tuples of (optimizer, name) from the agent to checkpoint.

Returns list of tuples of (optimizer, name).

Return type List[Tuple[OptimizerType, str]]

load(*model_dir: Optional[str]*, *step: Optional[int]*) → None
Load the agent.

Parameters

- **model_dir** (*Optional[str]*) – directory to load the model from.
- **step** (*Optional[int]*) – step for tracking the training of the agent.

sample_action(*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*)
Sample the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for sampling the action.

Returns sampled action.

Return type np.ndarray

save(*model_dir: str*, *step: int*, *retain_last_n: int*, *should_save_metadata: bool = True*) → None
Save the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.

- **should_save_metadata** (*bool, optional*) – should training metadata be saved. Defaults to True.

save_components (*model_dir: str*, *step: int*, *retain_last_n: int*) → None
Save the different components of the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.

save_optimizers (*model_dir: str*, *step: int*, *retain_last_n: int*) → None
Save the different optimizers of the agent.

Parameters

- **model_dir** (*str*) – directory to save.
- **step** (*int*) – step for tracking the training of the agent.
- **retain_last_n** (*int*) – number of models to retain.

select_action (*multitask_obs: Dict[str, torch.Tensor]*, *modes: List[str]*)
Select the action to perform.

Parameters

- **multitask_obs** (*ObsType*) – Observation from the multitask environment.
- **modes** (*List[str]*) – modes for selecting the action.

Returns selected action.

Return type np.ndarray

train (*training: bool = True*)
Set the agent in training/evaluation mode

Parameters **training** (*bool, optional*) – should set in training mode. Defaults to True.

update (*replay_buffer: mtrl.replay_buffer.ReplayBuffer*, *logger: mtrl.logger.Logger*, *step: int*, *kwargs_to_compute_gradient: Optional[Dict[str, Any]] = None*, *buffer_index_to_sample: Optional[numpy.ndarray] = None*)
Update the agent.

Parameters

- **replay_buffer** (ReplayBuffer) – replay buffer to sample the data.
- **logger** (Logger) – logger for logging.
- **step** (*int*) – step for tracking the training progress.
- **kwargs_to_compute_gradient** (*Optional[Dict[str, Any]], optional*) – Defaults to None.
- **buffer_index_to_sample** (*Optional[np.ndarray], optional*) – if this parameter is specified, use these indices instead of sampling from the replay buffer. If this is set to *None*, sample from the replay buffer. buffer_index_to_sample Defaults to None.

Returns

index sampled (from the replay buffer) to train the model. If buffer_index_to_sample is not set to None, return buffer_index_to_sample.

Return type np.ndarray

Module contents

7.1.2 mtrl.app package

Submodules

mtrl.app.run module

This is the main entry point for the running the experiments.

`mtrl.app.run.`**`run`**(*config: omegaconf.dictconfig.DictConfig*) → None
Create and run the experiment.

Parameters **`config`** (*`ConfigType`*) – config for the experiment.

Module contents

7.1.3 mtrl.env package

Subpackages

mtrl.env.gym_1 package

Subpackages

mtrl.env.gym_1.utils package

Submodules

mtrl.env.gym_1.utils.framestack module

mtrl.env.gym_1.utils.sticky_observation module

Module contents

Submodules

mtrl.env.gym_1.cartpole module

mtrl.env.gym_1.half_cheetah module

Module contents

Submodules

mtrl.env.builder module

mtrl.env.builder.**build_dmcontrol_vec_env**(*domain_name: str*, *task_name: str*, *prefix: str*, *make_kwargs: omegaconf.dictconfig.DictConfig*, *env_id_list: List[int]*, *seed_list: List[int]*, *mode_list: List[str]*) → *mtrl.env.vec_env.VecEnv*

mtrl.env.builder.**build_metaworld_vec_env**(*config: omegaconf.dictconfig.DictConfig*, *benchmark: metaworld.Benchmark*, *mode: str*, *env_id_to_task_map: Optional[Dict[str, metaworld.Task]]*) → Tuple[gym.vector.async_vector_env.AsyncVectorEnv, Optional[Dict[str, Any]]]

mtrl.env.types module

Collection of types used in the env.

mtrl.env.vec_env module

class mtrl.env.vec_env.**MetaWorldVecEnv**(*env_metadata: Dict[str, Any]*, *env_fns*, *observation_space=None*, *action_space=None*, *shared_memory=True*, *copy=True*, *context=None*, *daemon=True*, *worker=None*)

Bases: gym.vector.async_vector_env.AsyncVectorEnv

Return only every *skip*-th frame

create_multitask_obs(*env_obs*)

property ids

property mode

reset()

Reset all sub-environments and return a batch of initial observations.

Returns **observations** – A batch of observations from the vectorized environment.

Return type sample from *observation_space*

step(*actions*)

Take an action for each sub-environments.

Parameters **actions** (iterable of samples from *action_space*) – List of actions.

Returns

- **observations** (sample from *observation_space*) – A batch of observations from the vectorized environment.
- **rewards** (*np.ndarray* instance (dtype *np.float_*)) – A vector of rewards from the vectorized environment.
- **dones** (*np.ndarray* instance (dtype *np.bool_*)) – A vector whose entries indicate whether the episode has ended.
- **infos** (*list of dict*) – A list of auxiliary diagnostic information dicts from sub-environments.

class mtrl.env.vec_env.**VecEnv**(*env_metadata: Dict[str, Any]*, *env_fns*, *observation_space=None*, *action_space=None*, *shared_memory=True*, *copy=True*, *context=None*, *daemon=True*, *worker=None*)

Bases: gym.vector.async_vector_env.AsyncVectorEnv

Return only every *skip*-th frame

property ids

property mode

reset()

Reset all sub-environments and return a batch of initial observations.

Returns **observations** – A batch of observations from the vectorized environment.

Return type sample from *observation_space*

step(*actions*)

Take an action for each sub-environments.

Parameters **actions** (iterable of samples from *action_space*) – List of actions.

Returns

- **observations** (sample from *observation_space*) – A batch of observations from the vectorized environment.
- **rewards** (*np.ndarray* instance (dtype *np.float_*)) – A vector of rewards from the vectorized environment.
- **dones** (*np.ndarray* instance (dtype *np.bool_*)) – A vector whose entries indicate whether the episode has ended.
- **infos** (*list of dict*) – A list of auxiliary diagnostic information dicts from sub-environments.

Module contents

mtrl.env.**register_once**(*id*, *entry_point*, ***kwargs*)

7.1.4 mtrl.experiment package

Submodules

mtrl.experiment.dmcontrol module

Class to interface with an Experiment

class mtrl.experiment.dmcontrol.**Experiment**(*config: omegaconf.dictconfig.DictConfig*, *experiment_id: str = '0'*)

Bases: *mtrl.experiment.multitask.Experiment*

Experiment Class to manage the lifecycle of a multi-task model.

Parameters

- **config** (*ConfigType*) –
- **experiment_id** (*str, optional*) – Defaults to “0”.

evaluate_vec_env_of_tasks(*vec_env:* mtrl.env.vec_env.VecEnv, *step: int*, *episode: int*)
Evaluate the agent's performance on the different environments, vectorized as a single instance of vectorized environment.

Since we are evaluating on multiple tasks, we track additional metadata to track which metric corresponds to which task.

Parameters

- **vec_env** (*VecEnv*) – vectorized environment.
- **step** (*int*) – step for tracking the training of the agent.
- **episode** (*int*) – episode for tracking the training of the agent.

get_action_when_evaluating_vec_env_of_tasks(*multitask_obs: Dict[str, Union[numpy.ndarray, str, int, float]]*, *modes: List[str]*) → torch.Tensor

mtrl.experiment.experiment module

Experiment class manages the lifecycle of a model.

class mtrl.experiment.experiment.**Experiment**(*config: omegaconf.dictconfig.DictConfig*, *experiment_id: str = '0'*)
Bases: *mtrl.utils.checkpointable.Checkpointable*

Experiment Class to manage the lifecycle of a model.

Parameters

- **config** (*ConfigType*) –
- **experiment_id** (*str, optional*) – Defaults to "0".

build_envs() → Tuple[Dict[str, *mtrl.env.vec_env.VecEnv*], Dict[str, gym.spaces.box.Box]]
Subclasses should implement this method to build the environments.

Raises **NotImplementedError** – this method should be implemented by the subclasses.

Returns Tuple of environment dictionary and environment metadata.

Return type Tuple[EnvsDictType, EnvMetaDataType]

close_envs()
Close all the environments.

load(*epoch: Optional[int]*) → Any
Load the object from a checkpoint.

Returns Any

periodic_save(*epoch: int*) → None
Perioridically save the experiment.

This is a utility method, built on top of the *save* method. It performs an extra check of wether the experiment is configured to be saved during the current epoch. :param epoch: current epoch. :type epoch: int

run() → None
Run the experiment.

Raises **NotImplementedError** – This method should be implemented by the subclasses.

save(*epoch: int*) → Any
Save the object to a checkpoint.

Returns Any

startup_logs() → None
Write some logs at the start of the experiment.

mtrl.experiment.experiment.**get_env_metadata**(*env: gym.vector.async_vector_env.AsyncVectorEnv*, *max_episode_steps: Optional[int] = None*, *ordered_task_list: Optional[List[str]] = None*) → Dict[str, gym.spaces.box.Box]
Method to get the metadata from an environment

mtrl.experiment.experiment.**prepare_config**(*config: omegaconf.dictconfig.DictConfig*, *env_metadata: Dict[str, gym.spaces.box.Box]*) → omegaconf.dictconfig.DictConfig
Infer some config attributes during runtime.

Parameters

- **config** (*ConfigType*) – config to update.
- **env_metadata** (*EnvMetaDataType*) – metadata of the environment.

Returns updated config.

Return type ConfigType

mtrl.experiment.metaworld module

Class to interface with an Experiment

class mtrl.experiment.metaworld.**Experiment**(*config: omegaconf.dictconfig.DictConfig*, *experiment_id: str = '0'*)
Bases: *mtrl.experiment.multitask.Experiment*

Experiment Class

Experiment Class to manage the lifecycle of a multi-task model.

Parameters

- **config** (*ConfigType*) –
- **experiment_id** (*str, optional*) – Defaults to "0".

build_envs()
Build environments and return env-related metadata

collect_trajectory(*vec_env:* mtrl.env.vec_env.VecEnv, *num_steps: int*) → None
Collect some trajectories, by unrolling the policy (in train mode), and update the replay buffer. :param vec_env: environment to collect data from. :type vec_env: VecEnv :param num_steps: number of steps to collect data for. :type num_steps: int

create_env_id_to_index_map() → Dict[str, int]

create_eval_modes_to_env_ids()
Map each eval mode to a list of environment index.

The eval modes are of the form *eval_xyz* where *xyz* specifies the specific type of evaluation. For example. *eval_interpolation* means that we are using interpolation environments for evaluation. The eval moe can also be set to just *eval*.

Returns

dictionary with different eval modes as keys and list of environment index as values.

Return type Dict[str, List[int]]

evaluate_vec_env_of_tasks(*vec_env:* mtrl.env.vec_env.VecEnv, *step: int*, *episode: int*)
Evaluate the agent's performance on the different environments, vectorized as a single instance of vectorized environment.

Since we are evaluating on multiple tasks, we track additional metadata to track which metric corresponds to which task.

Parameters

- **vec_env** (VecEnv) – vectorized environment.
- **step** (*int*) – step for tracking the training of the agent.
- **episode** (*int*) – episode for tracking the training of the agent.

mtrl.experiment.multitask module

Experiment class manages the lifecycle of a multi-task model.

class mtrl.experiment.multitask.**Experiment**(*config: omegaconf.dictconfig.DictConfig*, *experiment_id: str = '0'*)
Bases: *mtrl.experiment.experiment.Experiment*

Experiment Class to manage the lifecycle of a multi-task model.

Parameters

- **config** (*ConfigType*) –
- **experiment_id** (*str, optional*) – Defaults to "0".

build_envs() → Tuple[Dict[str, *mtrl.env.vec_env.VecEnv*], Dict[str, gym.spaces.box.Box]]
Build environments and return env-related metadata

collect_trajectory(*vec_env:* mtrl.env.vec_env.VecEnv, *num_steps: int*) → None
Collect some trajectories, by unrolling the policy (in train mode), and update the replay buffer. :param vec_env: environment to collect data from. :type vec_env: VecEnv :param num_steps: number of steps to collect data for. :type num_steps: int

create_eval_modes_to_env_ids() → Dict[str, List[int]]
Map each eval mode to a list of environment index.

The eval modes are of the form *eval_xyz* where *xyz* specifies the specific type of evaluation. For example. *eval_interpolation* means that we are using interpolation environments for evaluation. The eval moe can also be set to just *eval*.

Returns

dictionary with different eval modes as keys and list of environment index as values.

Return type Dict[str, List[int]]

run()
Run the experiment.

mtrl.experiment.utils module

mtrl.experiment.utils.**clear**(*config: omegaconf.dictconfig.DictConfig*) → None
Clear an experiment and delete all its data/metadata/logs given a config

Parameters **config** (*ConfigType*) – config of the experiment to be cleared

mtrl.experiment.utils.**get_dirs_to_delete_from_experiment**(*config: omegaconf.dictconfig.DictConfig*) → List[str]

Return a list of dirs that should be deleted when clearing an experiment

Parameters **config** (*ConfigType*) – config of the experiment to be cleared

Returns List of directories to be deleted

Return type List[str]

mtrl.experiment.utils.**prepare_and_run**(*config: omegaconf.dictconfig.DictConfig*) → None
Prepare an experiment and run the experiment.

Parameters **config** (*ConfigType*) – config of the experiment

Module contents

7.1.5 mtrl.utils package

Submodules

mtrl.utils.checkpointable module

Interface for the objects that can be checkpointed on the filesystem.

class mtrl.utils.checkpointable.**Checkpointable**
Bases: abc.ABC

All classes that want to support checkpointing should extend this class.

abstract load(**args*, ***kwargs*) → Any
Load the object from a checkpoint.

Returns Any

abstract save(**args*, ***kwargs*) → Any
Save the object to a checkpoint.

Returns Any

mtrl.utils.config module

Code to interface with the config.

mtrl.utils.config.**dict_to_config**(*dictionary: Dict*) → omegaconf.dictconfig.DictConfig
Convert the dictionary to a config.

Parameters **dictionary** (*Dict*) – dictionary to convert.

Returns config made from the dictionary.

Return type ConfigType

mtrl.utils.config.**get_env_params_from_config**(*config: omegaconf.dictconfig.DictConfig*) → omegaconf.dictconfig.DictConfig
Get the params needed for building the environment from a config.

Parameters **config** (*ConfigType*) –

Returns params for building the environment, encoded as a config.

Return type ConfigType

mtrl.utils.config.**make_config_immutable**(*config: omegaconf.dictconfig.DictConfig*) → omegaconf.dictconfig.DictConfig
Set the config to be immutable.

Parameters **config** (*ConfigType*) –

Returns

Return type ConfigType

mtrl.utils.config.**make_config_mutable**(*config: omegaconf.dictconfig.DictConfig*) → omegaconf.dictconfig.DictConfig
Set the config to be mutable.

Parameters **config** (*ConfigType*) –

Returns

Return type ConfigType

mtrl.utils.config.**pretty_print**(*config*, *resolve: bool = True*)
Prettyprint the config.

Parameters

- **config** (*[type]*) –
- **resolve** (*bool, optional*) – should resolve the config before printing. Defaults to True.

mtrl.utils.config.**process_config**(*config: omegaconf.dictconfig.DictConfig*, *should_make_dir: bool = True*) → omegaconf.dictconfig.DictConfig
Process the config.

Parameters

- **config** (*ConfigType*) – config object to process.
- **should_make_dir** (*bool, optional*) – should make dir for saving logs, models etc? Defaults to True.

Returns processed config.

Return type ConfigType

mtrl.utils.config.**read_config_from_file**(*config_path: str*) → omegaconf.dictconfig.DictConfig

Read the config from filesystem.

Parameters **config_path** (*str*) – path to read config from.

Returns

Return type ConfigType

mtrl.utils.config.**set_struct**(*config: omegaconf.dictconfig.DictConfig*) → omegaconf.dictconfig.DictConfig

Set the struct flag in the config.

Parameters **config** (*ConfigType*) –

Returns

Return type ConfigType

mtrl.utils.config.**to_dict**(*config: omegaconf.dictconfig.DictConfig*) → Dict[str, Any]

Convert config to a dictionary.

Parameters **config** (*ConfigType*) –

Returns

Return type Dict

mtrl.utils.config.**unset_struct**(*config: omegaconf.dictconfig.DictConfig*) → omegaconf.dictconfig.DictConfig

Unset the struct flag in the config.

Parameters **config** (*ConfigType*) –

Returns

Return type ConfigType

mtrl.utils.types module

Collection of types used in the code.

mtrl.utils.utils module

Collection of utility functions

mtrl.utils.utils.**chunks**(*_list: List[T]*, *n: int*) → Iterator[List[T]]

Yield successive n-sized chunks from given list. Taken from https://stackoverflow.com/questions/312443/how-do-you-split-a-list-into-evenly-sized-chunks

Parameters

- **_list** (*List[T]*) – list to chunk.
- **n** (*int*) – size of chunks.

Yields *Iterator[List[T]]* – iterable over the chunks

mtrl.utils.utils.**flatten_list**(*_list: List[List[Any]]*) → List[Any]

Flatten a list of lists into a single list

Parameters **_list** (*List[List[Any]]*) – List of lists

Returns Flattened list

Return type List[Any]

mtrl.utils.utils.**get_current_commit_id**() → str
Get current commit id.

Returns current commit id.

Return type str

mtrl.utils.utils.**has_uncommitted_changes**() → bool
Check if there are uncommited changes.

Returns wether there are uncommiteed changes.

Return type bool

mtrl.utils.utils.**is_integer**(*n: Union[int, str, float]*) → bool
Check if the given value can be interpreted as an integer.

Parameters **n** (*Union[int, str, float]*) – value to check.

Returns can be the value be interpreted as an integer.

Return type bool

mtrl.utils.utils.**make_dir**(*path: str*) → str
Make a directory, along with parent directories. Does not return an error if the directory already exists.

Parameters **path** (*str*) – path to make the directory.

Returns path of the new directory.

Return type str

mtrl.utils.utils.**set_seed**(*seed: int*) → None
Set the seed for python, numpy, and torch.

Parameters **seed** (*int*) – seed to set.

mtrl.utils.utils.**split_on_caps**(*input_str: str*) → List[str]
Split a given string at uppercase characters. Taken from: https://stackoverflow.com/questions/2277352/split-a-string-at-uppercase-letters

Parameters **input_str** (*str*) – string to split.

Returns splits of the given string.

Return type List[str]

mtrl.utils.video module

Utility to record the environment frames into a video.

class mtrl.utils.video.**VideoRecorder**(*dir_name*, *height=256*, *width=256*, *camera_id=0*, *fps=30*)
Bases: object

Class to record the environment frames into a video.

Parameters

- **dir_name** (*[type]*) – directory to save the recording.
- **height** (*int, optional*) – height of the frame. Defaults to 256.
- **width** (*int, optional*) – width of the frame. Defaults to 256.

- **camera_id** (*int, optional*) – id of the camera for recording. Defaults to 0.
- **fps** (*int, optional*) – frames-per-second for the recording. Defaults to 30.

init (*enabled=True*)
Initialize the recorder.

Parameters **enabled** (*bool, optional*) – should enable the recorder or not. Defaults to True.

record (*frame*, *env=None*)
Record the frames.

Parameters **env** (*[type]*) – environment to record the frames.

save (*file_name*)
Save the frames as video to *self.dir_name* in a file named *file_name*.

Parameters **file_name** (*[type]*) – name of the file to store the video frames.

Module contents

7.2 Submodules

7.3 mtrl.logger module

class mtrl.logger.**AverageMeter**
Bases: *mtrl.logger.Meter*

update (*value*, *n=1*)

value ()

class mtrl.logger.**CurrentMeter**
Bases: *mtrl.logger.Meter*

update (*value*, *n=1*)

value ()

class mtrl.logger.**Logger** (*log_dir*, *config*, *retain_logs: bool = False*)
Bases: object

dump (*step*)

log (*key*, *value*, *step*, *n=1*)

class mtrl.logger.**Meter**
Bases: object

update (*value*, *n=1*)

value ()

class mtrl.logger.**MetersGroup** (*file_name*, *formating*, *mode: str*, *retain_logs: bool*)
Bases: object

dump (*step*, *prefix*)

log (*key*, *value*, *n=1*)

mtrl.logger.**np_float32** (*val*)

mtrl.logger.**np_int64**(*val*)

mtrl.logger.**serialize_log**(*val*)
mtrl.logger.**serialize_log**(*val: numpy.float32*)
mtrl.logger.**serialize_log**(*val: numpy.int64*)
Used by default.

7.4 mtrl.replay_buffer module

class mtrl.replay_buffer.**ReplayBuffer**(*env_obs_shape*, *task_obs_shape*, *action_shape*, *capacity*, *batch_size*, *device*)

Bases: object

Buffer to store environment transitions.

add(*env_obs*, *action*, *reward*, *next_env_obs*, *done*, *task_obs*)

delete_from_filesystem(*dir_to_delete_from: str*)

is_empty()

load(*save_dir*)

reset()

sample(*index=None*) → *mtrl.replay_buffer.ReplayBufferSample*

sample_an_index(*index*, *total_number_of_environments*) → *mtrl.replay_buffer.ReplayBufferSample*
Return env_observations for only the given index

save(*save_dir*, *size_per_chunk: int*, *num_samples_to_save: int*)

class mtrl.replay_buffer.**ReplayBufferSample**(*env_obs: torch.Tensor*, *action: torch.Tensor*, *reward: torch.Tensor*, *next_env_obs: torch.Tensor*, *not_done: torch.Tensor*, *task_obs: torch.Tensor*, *buffer_index: torch.Tensor*)

Bases: object

action

buffer_index

env_obs

next_env_obs

not_done

reward

task_obs

7.5 Module contents

CHAPTER

EIGHT

COMMUNITY

Ask questions in the chat or GitHub issues.

To contribute, open a Pull Request (PR)

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[CBLR18] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. Gradnorm: gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning*, 794–803. PMLR, 2018.

[GKB+19] Carles Gelada, Saurabh Kumar, Jacob Buckman, Ofir Nachum, and Marc G Bellemare. Deepmdp: learning continuous latent space models for representation learning. In *International Conference on Machine Learning*, 2170–2179. PMLR, 2019.

[PSDV+18] Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. Film: visual reasoning with a general conditioning layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32. 2018.

[TBC+17] Yee Whye Teh, Victor Bapst, Wojciech Marian Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: robust multitask reinforcement learning. *arXiv preprint arXiv:1707.04175*, 2017.

[YXWW20] Ruihan Yang, Huazhe Xu, Yi Wu, and Xiaolong Wang. Multi-task reinforcement learning with soft modularization. *arXiv preprint arXiv:2003.13661*, 2020.

[YKG+20] Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. *arXiv preprint arXiv:2001.06782*, 2020.

[YQH+20] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: a benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*, 1094–1100. PMLR, 2020.

[ZSKP20] Amy Zhang, Shagun Sodhani, Khimya Khetarpal, and Joelle Pineau. Multi-task reinforcement learning as a hidden-parameter block mdp. *arXiv preprint arXiv:2007.07206*, 2020.

PYTHON MODULE INDEX

m

INDEX

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

Z