MBRL-Lib: A Modular Library for Model-based Reinforcement Learning

Luis Pineda Brandon Amos Amy Zhang Nathan O. Lambert
Roberto Calandra
Facebook AI Research
University of California, Berkeley
{lep,bda,amyzhang,rcalandra}@fb.com, nol@berkeley.edu

Abstract

Model-based reinforcement learning is a compelling framework for data-efficient learning of agents that interact with the world. This family of algorithms has many subcomponents that need to be carefully selected and tuned. As a result the entry-bar for researchers to approach the field and to deploy it in real-world tasks can be daunting. In this paper, we present MBRL-Lib – a machine learning library for model-based reinforcement learning in continuous state-action spaces based on PyTorch. MBRL-Lib is designed as a platform for both researchers, to easily develop, debug and compare new algorithms, and non-expert user, to lower the entry-bar of deploying state-of-the-art algorithms. MBRL-Lib is open-source at https://github.com/facebookresearch/mbrl-lib.

1 Introduction

Model-based Reinforcement Learning (MBRL) for continuous control is a growing area of research that investigates agents explicitly modeling and interacting with the world. MBRL is capable of learning to control rapidly from a limited number of trials, and enables us to integrate specialized domain knowledge into the agent about how the world works. While methods are largely exploratory and unsettled, some shared components of these systems are emerging that work in conjunction and harmony together: the forward dynamics model, the state-action belief propagation, the reward function, and the planner/policy optimizer.

Albeit promising, MBRL suffers from reproducibility and computational issues stemming from the broader issues in the field [Henderson et al., 2018]. Moreover, MBRL methods can be even more difficult to debug and get working due to the depth of individual components and complex interactions between them, *e.g.* between the model losses and control objectives [Lambert et al., 2020].

In this work we present the library MBRL-Lib to provide shared foundations, tools, abstractions, and evaluations for continuous-action MBRL, providing: 1) software abstractions for lightweight and modular MBRL for practitioners and researchers, 2) debugging and visualization tools 3) re-implementations of the state-of-the-art MBRL methods that can be easily modified and extended. Our goal is to see MBRL-Lib grow as an open source project where new algorithms will be developed with this library and then added back. Our hope is that with these established baselines, growth in MBRL research will match the recent progress seen in model-free methods.

2 Related Work on Reinforcement Learning Software

Many reinforcement learning libraries have been publicly released [Dhariwal et al., 2017, Kostrikov, 2018, Guadarrama et al., 2018] that provide high-quality implementation of the complex components

Preprint. Under review.

involved in training RL agents. However, the vast majority of this focus on model-free reinforcement learning and lack crucial components of implementing MBRL algorithms. The availability of these libraries has opened up model-free RL methods to researchers and practitioners alike, providing opportunities to researchers that we believe contribute to the noted empirical gap in performance between model-free and model-based methods – in spite of theoretical evidence that model-based methods are superior in sample complexity [Tu and Recht, 2019].

In MBRL, the code landscape consists mostly of a relatively limited number of specific algorithm implementations that are publicly available [Chua et al., 2018, Janner et al., 2019, Wang and Ba, 2019]. Yet, none of these implementations provide a general and flexible environment that span multiple algorithms. A closely related effort is [Wang et al., 2019], which provides a unified interface for benchmarking MBRL methods, but it does not provide a unified code base and it is not designed to facilitate future research.

To foster the development of novel and impactful methods, substantial development and software design are required. Aside from MBRL-Lib, there are two other notable efforts to create centralized libraries for MBRL: Baconian [Linsen et al., 2019] and Bellman [McLeod et al., 2021], both of which are implemented in Tensorflow [Abadi et al., 2016]. Bellman, in particular, has been developed concurrently with MBRL-Lib and includes implementations for recent algorithms such as PETS, MBPO, and ME-TRPO, some of which are also included in MBRL-Lib. In contrast to these libraries, MBRL-Lib is the first MBRL library built to facilitate research in PyTorch [Paszke et al., 2019].

3 Background on Continuous Model-based Reinforcement Learning

Here we briefly discuss the foundations of model-based reinforcement learning for *continuous control* of *single-agent systems* and cover problem formulation, modeling the transition dynamics, and planning or policy optimization.

Problem Formulation MBRL often formulates the prediction problem as a Markov Decision Process (MDP) [Bellman, 1957]. We consider MDPs with continuous states $s_t \in \mathbb{R}^n$, continuous actions $a_t \in \mathbb{R}^m$, a reward function $r(s_t, a_t) \in \mathbb{R}$, and a transition function $p : \mathbb{R}^{n\times m\times n} \mapsto [0, \infty)$ representing the probability density of transitioning to state s_t given that action a was selected at state s_t.[1] A solution to a finite-time MDP is a policy/controller, $\pi^*(\cdot)$, mapping environment states to distributions over actions, which maximizes the expected sum of rewards:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{s_{t+1}\sim p(s_t,a_t), a_t\sim\pi(s_t)} \left[\sum_{t=0}^{T} r(s_t, a_t) \right]. \tag{1}$$

Modeling and Learning the Transition Dynamics MBRL agents learn a forward dynamics model

$$\hat{s}_{t+1} \sim f_\theta(s_t, a_t)\,, \tag{2}$$

to approximate the true transition function of the environment; often, rewards function and termination functions are also learned, which we omit here for clarity. Frequently, when learning the transition function, one can also model the delta transition between the current and next states as

$$\hat{s}_{t+1} \sim s_t + f_\theta(s_t, a_t)\,. \tag{3}$$

The model is trained on a dataset $\mathcal{D} = \{(s_i, a_i, s_{i+1}, r_i, d_i)\}_{i=1}^N$, where r_i is a sampled reward and d_i is a termination indicator denoting the end of the episode. How to best collect this dataset is an open research question. In practice, it is often collected by a mix of random exploration followed by iterative control policies that optimize within simulated model dynamics.

An important modeling choice is whether the learned model is deterministic or probabilistic over the next state predictions. For *deterministic models*, the standard approach is to train the model to minimize Mean Square Error (MSE) between the predicted and true states, as

$$l_{\text{MSE}} = \sum_{n=1}^{N} \| f_\theta(s_n, a_n) - s_{n+1} \|_2^2\,. \tag{4}$$

[1] Problems with high-dimensional observations, both continuous and discrete, *e.g.* pixels, are within the target scope of MBRL-Lib. Here we focus on the continuous-state case to simplify the presentation.

Probabilistic models typically optimize the Negative Log Likelihood (NLL). For example, a Gaussian policy predicts a distribution over next states as $\hat{s}_{t+1} \sim \mathcal{N}\big(\mu_\theta(s_t, a_t), \Sigma_\theta(s_t, a_t)\big)$), and the NLL is

$$l_{\text{NLL}} = \sum_{n=1}^{N} [\mu_\theta(s_n, a_n) - s_{n+1}]^T \Sigma_\theta^{-1}(s_n, a_n)[\mu_\theta(s_n, a_n) - s_{n+1}] + \log \det \Sigma_\theta(s_n, a_n). \tag{5}$$

Chua et al. [2018], Janner et al. [2019] incorporate uncertainty over model predictions, *i.e. epistemic* uncertainty, by using ensembles of independently trained models rather than a single model. These ensembles are typically trained with bootstrapping [Efron and Tibshirani, 1994] and involves training each model f_{θ_j} on its own copy of the dataset, $\mathcal{D}_j$, sampled with replacement from the original dataset $\mathcal{D}$; all models are trained independently. The final prediction for an ensemble of M models is

$$\hat{s}_{t+1} = \frac{1}{M} \sum_{j=1}^{M} f_{\theta_j}(s_t, a_t). \tag{6}$$

Planning or Policy Optimization Once a forward dynamics model is available, the next step is to use some planning or policy optimization algorithm to compute the next action as

$$a_{t+1} \sim \pi(s_t, f)\,. \tag{7}$$

In MBRL, policies are obtained by maximizing the sum of expected rewards over a predictive horizon. Due to approximation errors, long horizon prediction is not yet practical in MBRL, so, many methods choose the next action to take by unrolling the model predictions over a finite horizon h, which can be much shorter than the MDP horizon T. Here, the goal for a MBRL controller is to maximize the predicted expected return over the horizon. For deterministic policies, this can be done by solving

$$a^*_{t:t+h} = \operatorname*{arg\,max}_{a_{t:t+h}} \sum_{i=0}^{h-1} \mathbb{E}_{\hat{s}_{t+i}} \left[r(\hat{s}_{t+i}, a_{t+i}) \right] , \tag{8}$$

wherein the expectation over the future states $\hat{s}_t$ is usually computed numerically by sampling some number of trajectories from the model. In the case of ensembles, rather than using the mean ensemble prediction as in Eq. (6), it is useful to propagate the uncertainty by uniformly sampling a model from the ensemble, and then sample a prediction from the sampled member of the ensemble.

Optimizing Eq. (8) is a key distinction between MBRL methods. Some popular approaches are:

- **Trajectory sampling and controller distillation:** A set of open-loop trajectories are sampled independently, and the best one is chosen [Chua et al., 2018, Hafner et al., 2019, Lambert et al., 2019, Nagabandi et al., 2020]. Popular methods are random sampling and the Cross-Entropy Method (CEM) [De Boer et al., 2005] and extensions as in Wang and Ba [2019], Amos and Yarats [2020], Nagabandi et al. [2020]. Often, to avoid overfitting to model idiosyncrasies, it's beneficial to choose the mean over a set of top trajectories, rather than the best trajectory itself. Furthermore, Levine and Koltun [2013], Weber et al. [2017], Pascanu et al. [2017], Byravan et al. [2019], Lowrey et al. [2018] guide or condition policy training with model-based rollouts.
- **Differentiate through the model:** For non ensemble models, one can use a parameterized policy together with automatic differentiation through the dynamics model to find actions that maximize model performance [Deisenroth et al., 2013, Levine and Koltun, 2013, Heess et al., 2015, Henaff et al., 2018, Byravan et al., 2019, Amos et al., 2021].
- **Use a model-free policy:** In this case a model-free learner, such as SAC [Haarnoja et al., 2018], is used over the predicted dynamics, often by populating a populating a replay buffer with "imagined" trajectories obtained from the model [Gu et al., 2016, Kurutach et al., 2018, Janner et al., 2019].

Some of these methods can also be used in a Model Predictive Control (MPC) fashion [Camacho and Alba, 2013], wherein a full sequence of actions is obtained at each step, but only the first action in the computed trajectory is chosen; this is the standard approach when using trajectory sampling-based methods. The above is also not a comprehensive list of the wide variety of existing methods for generating continuous-action policies over a learned model.

Model-based methods in discrete systems We have focused this section on continuous spaces and refer to Kaiser et al. [2019], Schrittwieser et al. [2020], Hamrick et al. [2020] for some starter references on model-based methods in discrete spaces.

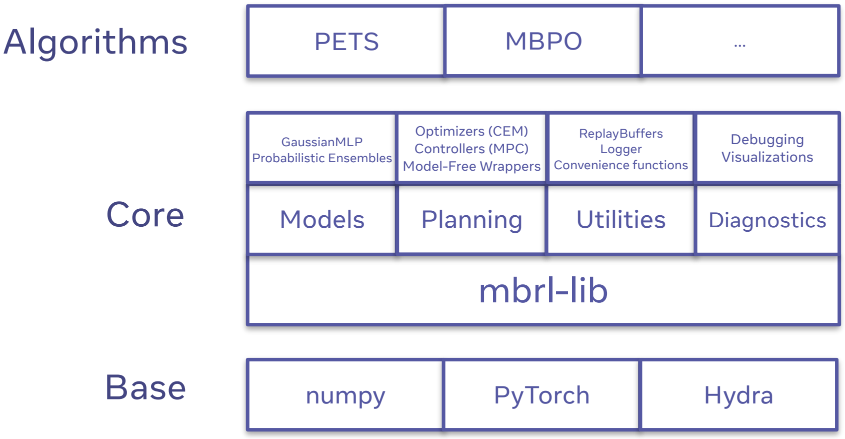


Figure 1: High-level structure of MBRL-Lib.

4 Software Architecture

In this section we detail the various design choices made and relevant modules needed to implement different MBRL algorithms.

4.1 Design Choices

We designed MBRL-Lib with few well-defined principles of MBRL development in mind:

- **Modularity** MBRL algorithms involve the interplay of multiple components whose internal operations are often hidden from each other. Therefore, once such an algorithm has been written, it should be possible to alter some of the component choices (*e.g.*, replace the trajectory sampling method) without affecting the others. Thus, MBRL-Lib seeks to enable a "mix-and-match" approach, where new algorithms or variants of existing ones can be easily written and tested without a lot of code involved.
- **Ease-of-use** We favor minimizing the amount of code required for users of the library, by heavy use of configuration files and a large set of utilities to encapsulate common operations, with an effort to make them general enough to accommodate new functionality. All functions are thoroughly documented and we provide examples of use. Using the library should be as friction-less as possible.
- **Performance** We consider crucial to provide high performance in terms of sample complexity and running time, and well-tuned hyperparameters for provided algorithms. More importantly, when performance gaps exists, future improvements to our components should be transparent to the end-users.

Figure 1 shows the high-level structure of MBRL-Lib. The library is built on top of numpy [Van Der Walt et al., 2011] and PyTorch [Paszke et al., 2019] for numeric computation, and Hydra [Yadan, 2019] for managing configurations. Following the modularity principle, MBRL-Lib is divided into the following four packages, which are described in more details in the next few subsections:

- **mbrl.models:** Dynamics model architectures, trainers, wrappers for data pre- and post-processing.
- **mbrl.planning:** Planners and optimizers of action choices.
- **mbrl.util:** General data-management utilities (*e.g.*, replay buffer, logger) and common operations.
- **mbrl.diagnostics:** Visualizations and tools for diagnosing your models and algorithms.

4.2 Models Package

The core data abstraction for training models in MBRL-Lib is `TransitionBatch`, a data class that stores a batch of transitions of observations, actions, next observations, rewards and terminal indicators; the underlying format can be either `numpy` arrays or `torch` tensors. On top of this data class, the `mbrl.models` package is built around three core classes: `Model`, `ModelTrainer`, and `ModelEnv`.

4.2.1 Model

`Model` is the abstract class defining all models that can be trained with `ModelTrainer` and simulated with `ModelEnv`. We have intentionally kept the required functions and parameters of `Model` interface minimal to account for the variety in choices for modeling dynamics in the literature. Following the PyTorch convention, all models have a `model.forward(TransitionBatch | torch.Tensor)` method that can return any number of output tensors (*e.g.*, mean, log-variance, log-probabilities). Additionally, the interface defines two abstract methods that all subclasses must implement:

- **`loss(TransitionBatch | torch.Tensor, target=Optional[torch.Tensor])`** computes and returns a tensor representing the loss associated to the input. Using this, `Model` provides a default `update(ModelInput, target=None)` method that updates the model using back-propagation on the result of `loss`. `ModelTrainer` will rely on `model.update(trasition_batch)` during model training. For models that are trained without back-propagation, such as a Gaussian Processes, users can opt to override the `update` method directly.
- **`eval_score(TransitionBatch | torch.Tensor, target=Optional[torch.Tensor])`** computes and returns a tensor representing an evaluation score for the model. This method will be used by `ModelTrainer` to compute a validation score.

Note that the input for these two methods can be both a `TransitionBatch` or a `torch.Tensor`. While, the `ModelTrainer` will only send `TransitionBatch` objects during training (in `numpy` format), we allow the model to receive torch tensors if this is more natural (*e.g.*, when using a feed-forward NN). Also, the `target` tensor for `loss` and `eval_score` is defined as an optional argument, since this won't be needed for models taking a `TransitionBatch` input. To solve the data mismatch with the trainer when models using `torch.Tensor` as inputs, we provide model instances that function as data processing wrappers to convert transition data into appropriate input tensors. One such example is `OneDimTransitionRewardModel`, which also provides convenient data processing capabilities, such as setting up delta targets as in Eq. (3), easily toggle between learning rewards or not, provides an input normalizer, as well as the option to add any custom observation pre-processing function.

Listing 1 exemplifies a typical usage pattern:

```python
import mbrl.models as models

net = models.GaussianMLP(in_size=5, out_size=4, device="cuda:0")
wrapper = models.OneDimTransitionRewardModel(net, target_is_delta=True)
trainer = models.ModelTrainer(wrapper)
trainer.train(*trainer_args)
model_env = models.ModelEnv(env, wrapper, *model_env_args)
```

Listing 1: Example of how to train a neural net to model 1-D transitions and rewards a Gaussian distributions in MBRL-Lib

Finally, the following methods are used by `ModelEnv` to simulate trajectories in the environment. They are optional for subclasses—basic default implementations are provided—, but can be useful for algorithms with specialized simulation pipelines (*e.g.*, latent variable models). They both receive a `[TransitionBatch | torch.Tensor]` input and return an observation to give to agents. For example, a model could receive a high-dimensional observation (*e.g.*, pixels), and return a low dimensional vector to use for control. Their differences are:

- **`reset:`** it should use only observation information and initialize any internal state the model needs over a trajectory.
- **`sample:`** it will typically use both observation and action information, and does not need to do any initialization.

4.2.2 ModelTrainer

The model trainer provides a training loop for supervised learning. In particular, its `train` method repeatedly updates the model for some number of epochs, and keeps track of the best model found during the training. The training and (optional) validation datasets are provided as objects of type `TransitionIterator`, which decouples the model trainer from the specifics of the replay buffer data storage; this allows users to train models on customized datasets without altering the data collection process. One such example is the `BootstrapIterator`, which provides transition batches with an additional model dimension that can be used to train ensembles of bootstrapped models; each model's batch is sampled from its own bootstrapped version of the data in the iterator.

Additionally, a common pattern in MBRL methods using ensemble models is to keep track of the best n models of the ensemble. This is handled by the trainer (using validation scores) if the `Model` instance has defined a `model.set_elite()` method.

Also, note that given the flexibility provided by the use of `TransitionBatch`, it is possible to use `ModelTrainer` to train other types of models, including, for example, imitation learning agents; the only requirements is to define `update()` properly.

4.2.3 ModelEnv

A standard approach to use dynamics models is to rollout (or "imagine") trajectories of actions using the model. To facilitate this, `ModelEnv` is a class that wraps the dynamics model into a reinforcement learning environment with an OpenAI's `gym`-like interface [Brockman et al., 2016]. It is then easy to use the model for planning with agents of different type, by following the well known `reset()` and `step()` usage pattern, as Listing 2 illustrates.

```python
import gym
import mbrl.models as models
import numpy as np
from mbrl.env.termination_fns import hopper

# Initialize the model
env = gym.make("Hopper-v2")
net = models.GaussianMLP(in_size=14, out_size=12, device="cuda:0")
wrapper = models.OneDimTransitionRewardModel(
    net, target_is_delta=True, learned_rewards=True)
# Construct the model environment
model_env = models.ModelEnv(
    wrapper, *model_env_args, term_fn=hopper)
# Simulate one step of the environment using the model
obs = env.reset()
model_obs = model_env.reset(obs[np.newaxis, :])
action = env.action_space.sample()
next_obs, reward, done, _ = model_env.step(
    action[np.newaxis, :], sample=True)
```

Listing 2: Example of use of `ModelEnv`.

The constructor for `ModelEnv` requires specifying a termination function to compute terminal indicators; the user can also pass an optional reward function, to use if rewards are not learned. The `reset` method receives a batch of observations to start the episode from, and `step` a batch of actions. When doing a step, the user can also specify if the model should return a random sample (if possible) or not.

4.3 Planning Package

The planning package defines a basic `Agent` interface to represent different kinds of decision-making agents, including both model-based planners and model-free agents. The only method required by the interface is `act(np.ndarray, **kwargs)`, which receives an observation and optional arguments, and returns an action. Optionally, the user can define a `plan` method, which receives an observation and returns a sequence of actions; `Agent`'s default implementation of `plan` just calls the `act` method.

Currently, we provide the following agents:

- **`RandomAgent:`** returns actions sampled from the environments action space.
- **`TrajectoryOptimzerAgent:`** An agent that uses a black-box optimizer to find the best sequence of actions for a given observation; the optimizer choice can be changed easily via configuration arguments. The user must provide a function to evaluate the trajectories. We currently provide

a function that evaluates trajectories using `ModelEnv`, and returns their total predicted rewards, essentially implementing a Model Predictive Control (MPC) [Camacho and Alba, 2013] agent. We also provide a Cross-Entropy Method [De Boer et al., 2005] (CEM) optimizer.

- **`SACAgent`:** a wrapper for a popular implementation of Soft Actor-Critic [Yarats and Kostrikov, 2020, Haarnoja et al., 2018].

4.3.1 The Cross-Entropy Method (CEM)

Here we quickly define CEM, which is a popular algorithm for non-convex continuous optimization [De Boer et al., 2005]. In general, CEM solves the optimization problem $\arg\min_x f(x)$ for $x \in \mathbb{R}^n$. Given a *sampling distribution* g_ϕ, the hyper-parameters of CEM are the number of *candidate points* sampled in each iteration N, the number of *elite candidates* k to use to fit the new sampling distribution to, and the number of iterations T. The iterates of CEM are the *parameters* ϕ of the sampling distribution. CEM starts with an *initial* sampling distribution $g_{\phi_1}(X) \in \mathbb{R}^n$, and in each iteration t generates N samples from the domain $[X_{t,i}]_{i=1}^N \sim g_{\phi_t}(\cdot)$, evaluates the function at those points $v_{t,i} := f_\theta(X_{t,i})$, and re-fits the sampling distribution to the top-k samples by solving the maximum-likelihood problem

$$\phi_{t+1} := \arg\max_\phi \sum_i \mathbb{1}\{v_{t,i} \leq \pi(v_t)_k\} \log g_\phi(X_{t,i}), \tag{9}$$

where the indicator $\mathbb{1}\{P\}$ is 1 if P is true and 0 otherwise, $g_\phi(X)$ is the likelihood of X under the distribution g_θ, and $\pi(x)$ sorts $x \in \mathbb{R}^n$ in ascending order so that

$$\pi(x)_1 \leq \pi(x)_2 \leq \ldots \leq \pi(x)_n.$$

We can then map from the final distribution g_{ϕ_T} back to the domain by taking the mean of it, *i.e.* $\hat{x} := \mathbb{E}[g_{\phi_{T+1}}(\cdot)]$, or by returning the best sample.

In MBRL, we define the objective $f(x)$ to be the control optimization problem in Eq. (8) over the actions. MBRL also often uses multivariate isotropic Gaussian sampling distributions parameterized by $\phi = \{\mu, \sigma^2\}$. Thus as discussed in, *e.g.*, Friedman et al. [2001], Eq. (9) has a closed-form solution given by the sample mean and variance of the top-k samples as $\mu_{t+1} = (1/k)\sum_{i\in\mathcal{I}_t} X_{t,i}$ and $\sigma^2_{t+1} = (1/k)\sum_{i\in\mathcal{I}_t} (X_{t,i} - \mu_{t+1})^2$, where the top-$k$ indexing set is $\mathcal{I}_t = \{i : v_{t,i} \leq \pi(v_t)_k\}$.

4.4 Utilities

Following the ease-of-use principle, MBRL-Lib contains several utilities that provide functionality that's common to many MBRL methods. For the utilities we heavily leverage `Hydra` [Yadan, 2019] configurations; an example of a typical configuration file (saved in YAML format) is shown in Listing 3, which we will use as a running example.

Listing 4 shows examples of some of the utilities provided by MBRL-Lib. Function `make_env` in the `util.mujoco` instantiates the environment, supporting both gym and `dm_control` [Tassa et al., 2020], as well as custom versions of some of these environments, available in `mbrl.env`. Function `util.create_one_dim_tr_model` creates a base model (here, `mbrl.models.GaussianMLP`) with the correct input/outputs for this experiment, and creates a `OneDimTransitionRewardModel` wrapper for it. Function `create_replay_buffer` creates a replay buffer of appropriate size (other optional configuration options are available), and function `rollout_agent_trajectories` populates the buffer with steps taken in the environment with a random agent. Note also that creation utilities can be given a path to load data from a directory, making it easy to re-use data from previous runs. Finally, `train_model_and_save_model_and_data` creates `TransitionIterators` for the trainer, normalizes the data if the model provides a normalizer, runs training, and saves the result to the given save directory.

Overall, our utilities replace a lot of the scaffolding associated with running MBRL experiments, and make it easy for users to concentrate on the high-level details of algorithm and model design.

4.5 Diagnostics Package

The final package is a set of visualization and debugging tools to develop and compare algorithms implemented in MBRL-Lib. The current version contains the following diagnostics:

```
# dynamics model configuration
dynamics_model:
    model:
      _target_: mbrl.models.GaussianMLP
      device: "cuda:0"
      num_layers: 4
      in_size: "???"  # our utilities will complete this info
      out_size: "???"  # our utilities will complete this info
      ensemble_size: 5
      hid_size: 200
      use_silu: true
      deterministic: false
      propagation_method: "fixed_model"

# algorithm specific options
algorithm:
    initial_exploration_steps: 5000
    learned_rewards: false
    target_is_delta: true
    normalize: True

# experiment specific options
overrides:
    env: "gym___Hopper-v2"
    term_fn: "hopper"
    trial_length: 1000,
    num_trials: 125,
    model_batch_size: 256,
    validation_ratio: 0.05

agent:
    # agent's configuration, omitted for space considerations
optimizer:
    # optimizer's configuration for the agent, if it uses one
```

Listing 3: Example `Hydra` configuration for MBRL-Lib. The string "???" is a special Hydra sequence that indicates this should be completed at runtime. Our utilities take care of handling these inputs according to the environment selected.

- **`Visualizer`** Can be used to generate videos of trajectories generated by the agent the model was trained on (and optionally another reference agent). Figure 3 shows an example output of this tool.
- **`Finetuner`** Can be used to train a model (new or saved from a previous run) on data generated by a reference agent. For example, take a model trained with data gathered while doing MPC, and retrain it with data from an optimal controller (*e.g.*, a pre-trained model-free agent). This can be useful to assess if the model is capable of learning correctly around trajectories of interest.
- **`DatasetEvaluator`** Takes a trained model and evaluates it in a saved dataset (not necessarily the same one used for training). The tool produces plot comparing the predicted values to the targets, one for each predicted dimension. An example is shown in Fig. 4, where the same model is evaluated in two different datasets.

We also provide a script for doing CEM control using `TrajectoryOptimizerAgent` on the true environment, which uses Python multiprocessing to launch multiple copies of the environment to evaluate trajectories. This script can be easily extended to other members of `mbrl.planning` and can be useful to assess the quality of new optimization algorithms or evaluation functions without any concern over training a perfect model. Figure 2 illustrates the result of applying this controller to gym's HalfCheetah-v2 environment, which ends up breaking the simulator due to overflow error.

```python
import mbrl.planning as planning
import mbrl.util.common as mbrl_util
import mbrl.util.mujoco as mujoco_util

# Initialize the model, trainer and the model environment
env, term_fn, reward_fn = mujoco_util.make_env(cfg)
obs_shape = env.observation_space.shape
act_shape = env.action_space.shape
# takes care of model input/output size, and
# adding the 1-D model wrapper
model = mbrl_util.create_one_dim_tr_model(cfg, obs_shape, act_shape, model_dir=None)
model_env = mbrl.models.ModelEnv(env, model, term_fn, reward_fn)
trainer = mbrl.models.DynamicsModelTrainer(model)

# create a replay buffer for the run
# automatically sets the size based on the number of trials to run
# and their length, but user can also override with a provided size
replay_buffer = mbrl_util.create_replay_buffer(
    cfg, obs_shape, act_shape, load_dir=None)

# rollout trajectories in the environment and
# optionally populate a replay buffer
mbrl_util.rollout_agent_trajectories(
    env,
    cfg.algorithm.initial_exploration_steps,
    planning.RandomAgent(env),
    {}, # keyword arguments for agent
    replay_buffer=replay_buffer,
)
# create a trajectory optimizer agent that
# evaluates trajectories in the ModelEnv
agent = planning.create_trajectory_optim_agent_for_model(
    model_env, cfg.algorithm.agent)

# create iterators for current buffer data, update normalization stats,
# run training, log results
mbrl_util.train_model_and_save_model_and_data(
    model, trainer, cfg.overrides, replay_buffer, savedir)
```

Listing 4: Example of utilities provided by MBRL-Lib

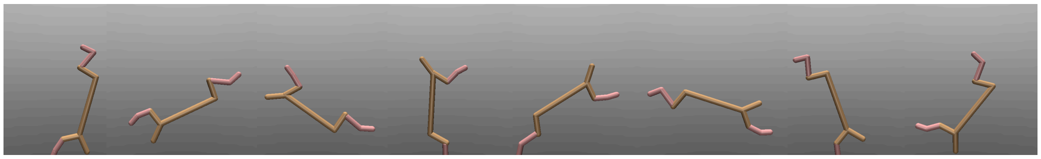


Figure 2: Final frames of an episode run with our implementation of a CEM-based trajectory optimizer on HalfCheetah-v2 environment. The simulation crashed with an overflow error after 679 steps, with an accumulated reward of 20215.78.

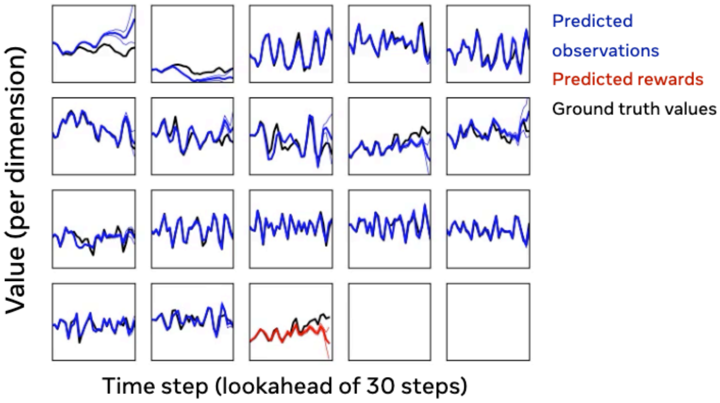


Figure 3: Example result of `mbrl.diagnostics.Visualizer`'s output on an MPC controller with CEM for HalfCheetah. Each subplot corresponds to predictions for a model dimension for 30 time steps, compared with the result of applying the same actions to the true environment. Uncertainty can be visualized via multiple model samples (best visible in the upper left subplot).

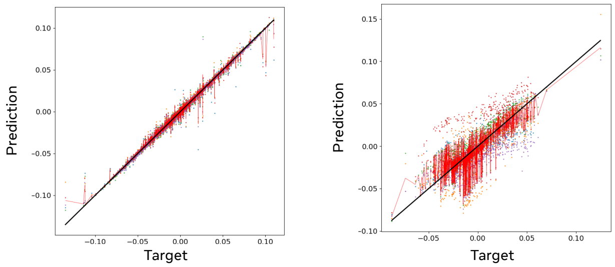


Figure 4: Example result of `mbrl.diagnostics.DatasetEvaluator`'s output. The model was trained on HalfCheetah using MPC control with CEM, the output corresponds to the first observation dimension. Left: model evaluated on the dataset it was trained on. Right: model evaluated on dataset collected with a SAC agent. Note that the predictions of this particular model are bad around optimal trajectories, highlighting problem in learning and data collection.

5 Experimental Results

As proof of concept for MBRL-Lib, we provide implementations for two state-of-the-art MBRL algorithms, namely, PETS [Chua et al., 2018] and MBPO [Janner et al., 2019]. We start by briefly describing these algorithms:

- **PETS:** The method consists of repeatedly applying two alternating operations: 1) train a probabilistic ensemble on all observed transitions, 2) run a trial of a trajectory sampling controller (CEM) over model-generated trajectories.
- **MBPO:** Like PETS, a probabilistic ensemble is trained on all observed transitions. However, rather than using the model for control, short simulated trajectories τ originating from points in the environment dataset, $\mathcal{D}$, are added to the replay buffer of a SAC agent, which is trained on the simulated data.

Unless otherwise specified, we used ensemble sizes of 7 models trained with Gaussian NLL, as described in Equation (5), with predictions using the top 5 "elite" models, according to validation score during training. All models predict delta observations as defined in Equation (3), and observations are normalized using all the data in the replay buffer before each training episode. Unless otherwise specified, the dynamics model is re-trained on all the available data every 250 steps. For MBPO, we use 20% of the data for validation, with a new split generated before each training loop starts (as it's done in the original MBPO implementation). For PETS we do not use validation data, following the original implementation, and elites are computed using the training MSE score.

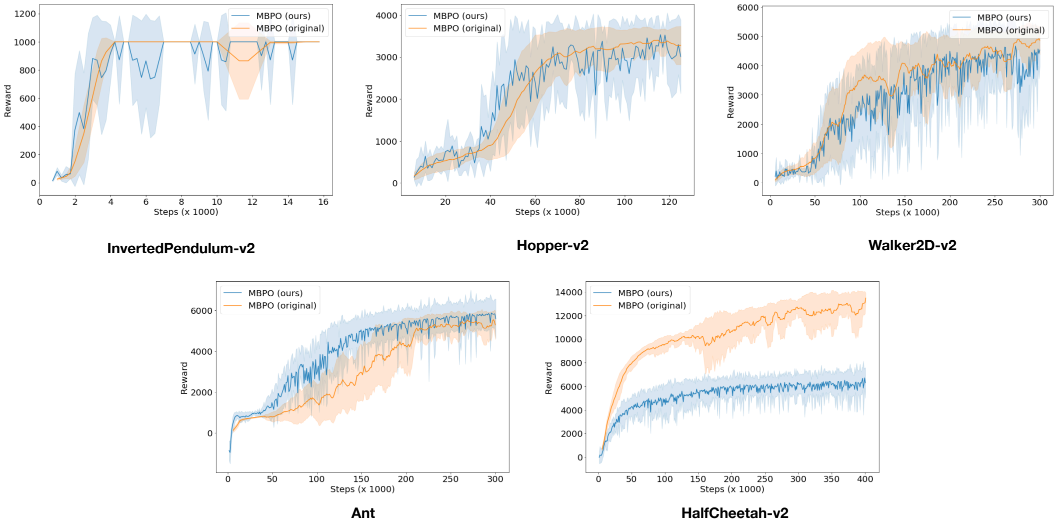


Figure 5: Results of our implementation of MBPO in five Mujoco environments. For the Ant environment, forces are removed from the observations, as was done in the original MBPO implementation.

Fig. 5 shows a comparison of the results obtained with our MBPO implementation and the original on five Mujoco 2.0 environment; the shaded regions correspond to standard deviation using 10 different random seeds (original plots were done with 5 seeds). As the plots show, for the majority of environments our code matches well the original results, and in one case (Ant) the average return observed is higher. A notable exception is HalfCheetah environment, where our results are significantly lower than the original; we are still investigating potential causes for this discrepancy. Our implementation also experiences trouble in the Humanoid-v2 environment, where the SAC losses tend to explode.

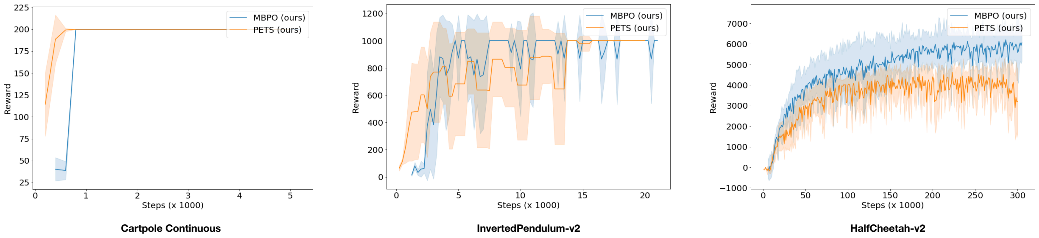
On the other hand, we have had more trouble reproducing the results of the original PETS implementation and have thus omitted a direct comparison with the original, as the large gap in performance is not particularly informative. Moreover, the original PETS code uses handcrafted transformations for the inputs and a given reward function, which makes comparison with other methods confusing, so we opted for a more standard learning setup in our experiments.

Fig. 6 shows a comparison of our implementations of PETS and MBPO in three environments, namely, Inverted Pendulum, HalfCheetah, and a continuous version of CartPole; the shaded region corresponds to standard deviation over 10 seeds. The performance in Cartpole and InvertedPendulum is close between the two algorithms, although PETS tends to be more unstable in InvertedPendulum. Interestingly, we found that deterministic ensembles allowed PETS to learn faster in CartPole environment; for the other environments, a probabilistic ensemble worked better.

However, the performance in HalfCheetah is consistently lower in PETS than in MBPO. Note that, in contrast to the original PETS implementation, for this experiments we used unmodified observations for all environments and learned rewards. Adding these modifications improves performance in the HalfCheetah environment, but not enough to match that of the original Tensorflow implementation [Zhang et al., 2021].

6 Conclusion

We introduce MBRL-Lib – the first PyTorch library dedicated to model-based reinforcement learning. This library has the double goal of providing researchers with a modular architecture that can be used for quickly implement, debug, evaluate and compare new algorithms, and to provide easy-to-use state-of-the-art implementations for non-expert users. We open-source MBRL-Lib at https://github.com/facebookresearch/mbrl-lib, with the aim of foster and support the model-based reinforcement learning community.



Cartpole Continuous

InvertedPendulum-v2

HalfCheetah-v2

Figure 6: Comparison between the MBRL-Lib implementations of PETS and MBPO in three different environments.

Acknowledgments and Disclosure of Funding

R.C. thanks Omry Yadan for useful discussions regarding software engineering and for work on an early version of MBRL library. L.P. thanks Shagun Sodhani, Olivier Delalleau, Po-Wei Chou, and Raghu Rajan for early feedback on the library, Michael Janner for answering questions about MBPO, and Xingyu Lin, whose PyTorch MBPO implementation was a great resource for debugging our own.

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

B. Amos and D. Yarats. The differentiable cross-entropy method. In *International Conference on Machine Learning*, pages 291–302. PMLR, 2020.

B. Amos, S. Stanton, D. Yarats, and A. G. Wilson. On the stochastic value gradient for continuous reinforcement learning. In *L4DC*, 2021.

R. Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

A. Byravan, J. T. Springenberg, A. Abdolmaleki, R. Hafner, M. Neunert, T. Lampe, N. Siegel, N. Heess, and M. Riedmiller. Imagined value gradients: Model-based policy optimization with transferable latent dynamics models. *arXiv preprint arXiv:1910.04142*, 2019.

E. F. Camacho and C. B. Alba. *Model predictive control*. Springer science & business media, 2013.

K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems (NIPS)*, pages 4754–4765, 2018.

P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

M. P. Deisenroth, D. Fox, and C. E. Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE transactions on pattern analysis and machine intelligence*, 37(2): 408–423, 2013.

P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

S. Gu, T. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

S. Guadarrama, A. Korattikara, O. Ramirez, P. Castro, E. Holly, S. Fishman, K. Wang, E. Gonina, N. Wu, E. Kokiopoulou, L. Sbaiz, J. Smith, G. Bartók, J. Berent, C. Harris, V. Vanhoucke, and E. Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. URL https://github.com/tensorflow/agents. [Online; accessed 25-June-2019].

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pages 2555–2565. PMLR, 2019.

J. B. Hamrick, A. L. Friesen, F. Behbahani, A. Guez, F. Viola, S. Witherspoon, T. Anthony, L. Buesing, P. Veličković, and T. Weber. On the role of planning in model-based deep reinforcement learning. *arXiv preprint arXiv:2011.04021*, 2020.

N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015.

M. Henaff, A. Canziani, and Y. LeCun. Model-predictive policy learning with uncertainty regularization for driving in dense traffic. In *International Conference on Learning Representations*, 2018.

P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. *Thirty-Second AAAI Conference On Artificial Intelligence (AAAI)*, 2018.

M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems*, 2019.

L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

I. Kostrikov. Pytorch implementations of reinforcement learning algorithms. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail, 2018.

T. Kurutach, I. Clavera, Y. Duan, A. Tamar, and P. Abbeel. Model-ensemble trust-region policy optimization. *arXiv preprint arXiv:1802.10592*, 2018.

N. O. Lambert, D. S. Drew, J. Yaconelli, S. Levine, R. Calandra, and K. S. Pister. Low-level control of a quadrotor with deep model-based reinforcement learning. *IEEE Robotics and Automation Letters*, 4(4):4224–4230, 2019.

N. O. Lambert, B. Amos, O. Yadan, and R. Calandra. Objective mismatch in model-based reinforcement learning. In *L4DC*, volume 120 of *Proceedings of Machine Learning Research*, pages 761–770. PMLR, 2020.

S. Levine and V. Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.

D. Linsen, G. Guanyu, L. Yuanlong, and W. Yonggang. Baconian: A unified opensource framework for model-based reinforcement learning. *arXiv preprint arXiv:1904.10762*, 2019.

K. Lowrey, A. Rajeswaran, S. Kakade, E. Todorov, and I. Mordatch. Plan online, learn offline: Efficient learning and exploration via model-based control. *arXiv preprint arXiv:1811.01848*, 2018.

J. McLeod, H. Stojic, V. Adam, D. Kim, J. Grau-Moya, P. Vrancx, and F. Leibfried. Bellman: A toolbox for model-based reinforcement learning in tensorflow. *arXiv:2103.14407*, 2021. URL https://arxiv.org/abs/2103.14407.

A. Nagabandi, K. Konolige, S. Levine, and V. Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, pages 1101–1112. PMLR, 2020.

R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. Reichert, T. Weber, D. Wierstra, and P. Battaglia. Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*, 2017.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. URL http://arxiv.org/abs/1912.01703.

J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Y. Tassa, S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, and N. Heess. dm_control: Software and tasks for continuous control, 2020.

S. Tu and B. Recht. The gap between model-based and model-free methods on the linear quadratic regulator: An asymptotic viewpoint. In A. Beygelzimer and D. Hsu, editors, *Proceedings of the Thirty-Second Conference on Learning Theory*, volume 99 of *Proceedings of Machine Learning Research*, pages 3036–3083, Phoenix, USA, 25–28 Jun 2019. PMLR. URL http://proceedings.mlr.press/v99/tu19a.html.

S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

T. Wang and J. Ba. Exploring model-based planning with policy networks. *arXiv preprint arXiv:1906.08649*, 2019.

T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba. Benchmarking model-based reinforcement learning. *arXiv preprint arXiv:1907.02057*, 2019.

T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*, 2017.

O. Yadan. Hydra - a framework for elegantly configuring complex applications. Github, 2019. URL https://github.com/facebookresearch/hydra.

D. Yarats and I. Kostrikov. Soft actor-critic (sac) implementation in pytorch. https://github.com/denisyarats/pytorch_sac, 2020.

B. Zhang, R. Rajan, L. Pineda, N. Lambert, A. Biedenkapp, K. Chua, F. Hutter, and R. Calandra. On the importance of hyperparameter optimization for model-based reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 4015–4023. PMLR, 2021.