Model-Based Reinforcement Learning for energy management in smart buildings

Andrea Blushi

June 2025

Contents

1 Introduction

The management of electrical grids, as well as energy management in buildings, has recently been facing a series of changes, primarily due to the advent of renewable energy resources. This has led to the need to modify and update current methodologies, with the goal of reducing consumption and carbon emissions.
The arrival of new algorithms in this sector has necessitated a standardization and simplification of the problem for technological exploration. CityLearn [Nweye et al., 2024] is an open-source Gymnasium environment that addresses this issue, providing an environment that accurately simulates the real-world dynamics of buildings and neighbourhoods, providing devices like air conditioners and solar panels. CityLearn allows the benchmarking of various advanced distributed energy resource control algorithms, providing useful parameters for comparing these techniques from various points of view, like discomfort, consumption and emissions.
Our research in this domain led us to Deep Reinforcement Learning (DRL), a field that has demonstrated significant success in complex challenges. Starting from a model-free Reinforcement Learning (RL) algorithm as Soft Actor-Critic (SAC) [Haarnoja et al., 2018], already tested and implemented in this environment, we expanded upon this by experimenting with model-based RL (MBRL) algorithms. By benchmarking various techniques, such as Model-Based Policy Optimization (MBPO) [Janner et al., 2021], Masked Model-based Actor-Critic (M2AC) [Pan et al., 2020], and Model-based Actor-Critic with Uncertainty-aware Rollout Adaption (MACURA) [Frauenknecht et al., 2024], we aimed to introduce a new level of sample efficiency to the basic SAC framework, experimenting the strengths of these model-based approaches.
In this text, we analyse CityLearn's dataset chosen for training and evaluating algorithms, and present the distinct mechanisms and performance of each model-based algorithm in this domain.

2 Environment

CityLearn [Nweye et al., 2024] is a collection of building energy models that constitute a district (also called a neighbourhood), which primarily includes electrical sources, energy accumulators, batteries, and electrical devices. It's also important to note that each building is modelled as a single thermal zone, and not all buildings necessarily implement all the types of devices described.
Buildings are equipped with a combination of thermal accumulators and batteries that provide energy flexibility, and these storage devices can replace main grid energy sources during less convenient periods. In some models, photovoltaic systems can be found, which completely or partially replace electricity consumption from the grid by generating energy autonomously. Furthermore, besides the main controller (managed by the agent), there's a backup controller internal to each device, which provides basic and automatic controls. This and many other characteristics are resumed in Figure 1.
We'll consider, for both training and evaluation, the building models from the CityLearn Challenge 2023 [AIcrowd et al., 2023]. The goal for participants was to develop an agent capable of managing the cooling system, electrical batteries, and DHW (Domestic Hot Water) storage of a set of three buildings. We assume the environment of the challenge to be represented by a Markov decision process (MDP) defined by $\mathcal{M} = \big(\mathcal{S}, \mathcal{A}, r, p, \rho_0\big)$, with $\mathcal{S}$ the observation and $\mathcal{A}$ the action space, while a transition model $p(s' \mid s, a)$

describes the thermal and energetic dynamics of the building. Then we define a reward $r \in \mathbb{R}$ generated from a reward function $r(s, a)$ that we will use to train the agent and to compare the results of the various RL algorithms.

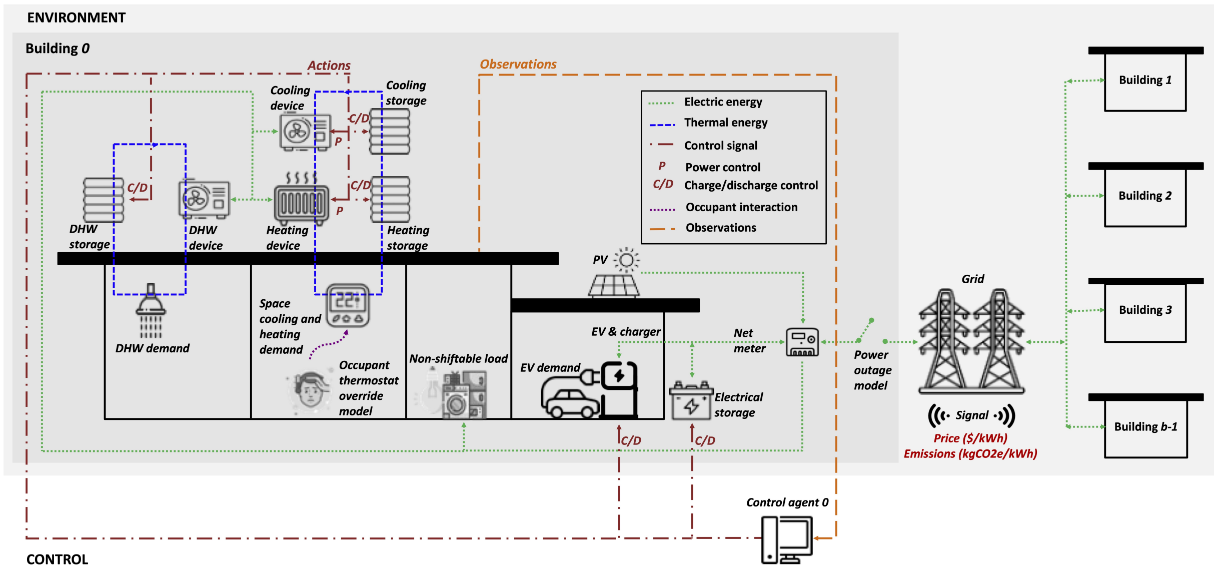

Figure 1: Image from [Nweye et al., 2024]. Here's an overview of the CityLearn environment, illustrating its key components (buildings, electrical grid, storage systems) and how they interact within the model.

Now, we'll describe only the observations, actions and reward used by the CityLearn Challenge 2023 [AIcrowd et al., 2023], and not all the possible combinations that the environment can offer.

2.1 Observations

In the CityLearn [Nweye et al., 2024], the state space coincides with the observations we obtain at each step. We will describe the most relevant information that we can collect. These observations are grouped into four categories:

- **Calendar**: includes time variables such as the month, day of the week, hour, and daylight saving status. All these observations are raw data extracted directly from *building.csv*.

- **Weather**: these are physical measurements and weather forecasts, that include the following attributes:

  – Outdoor dry bulb temperature (°C);

  – Outdoor relative humidity (%);

  – Diffuse solar irradiance (W/m^2);

  – Direct solar irradiance (W/m^2);

  All values are obtained directly as raw data from *weather.csv*.

- **District**: this category collects measurements involving the entire district, and includes the following attributes:

- Carbon intensity: indicates the CO_2 emission rate per total kWh of a district ($kgCO_2$/kWh). It is obtained directly as raw data from *carbon_intensity.csv*;
- Electricity pricing: current electricity price (\$/kWh) obtained as raw data from *pricing.csv*;

- **Building**: this category collects observations specific to each building:
  - Indoor dry bulb temperature: indicates the average indoor temperature of the building (°C). This is a value calculated using a predictive model based on Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997], which simulates the thermal dynamics of the building over time. This temperature depends on the actions taken and also considers other variables to predict how the building thermally reacts over time;

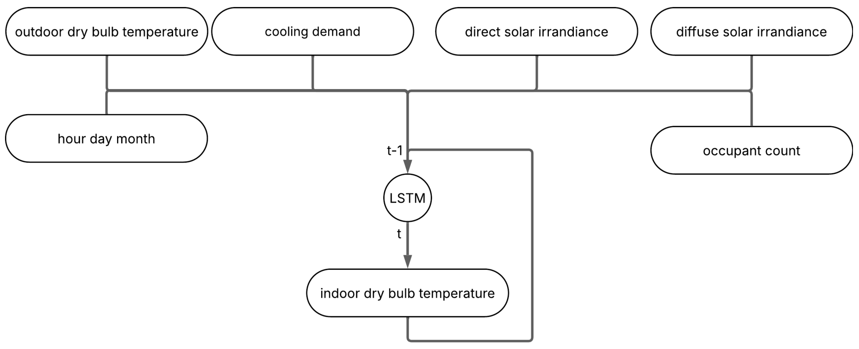
outdoor dry bulb temperature
cooling demand
direct solar irrandiance
diffuse solar irrandiance
hour day month
occupant count
t-1
LSTM
t
indoor dry bulb temperature

Figure 2: LSTM Architecture for Indoor Dry Bulb Temperature Prediction. This figure illustrates the inputs used by the LSTM, which also considers the previous (t-1) indoor dry bulb temperature to calculate the current.

  - Non shiftable load: fixed electrical consumption due to devices like washing machines or refrigerators (kWh). This is a derived value, calculated as the minimum between the actual demand of the devices (available in the *building.csv* dataset) and the maximum permissible electrical consumption by the building (called downward electrical flexibility);
  - Solar generation: electrical energy generated by solar panels (kWh), obtained from the *building.csv* dataset, which is provided as a pre-computed time series input;
  - Cooling device efficiency: which corresponds to the coefficient of performance obtained from the derived calculation based on outdoor temperature (obtained from *weather.csv*) and predefined device characteristics;
  - Cooling demand: energy demand for cooling (kWh). Represents the amount of thermal energy used to maintain the building's internal temperature within comfort limits during warm hours. The derived data is calculated as the sum of thermal energy supplied directly by the cooling device and energy drawn from the cooling accumulator;
  - DHW demand: energy demand for domestic hot water (kWh), obtained similarly to the cooling device;

  - Electrical storage SOC: state of charge of the electrical storage (0 discharged, 1 fully charged). A derived value obtained directly from the electrical energy charged or discharged by the electric battery (influenced by actions), limited by the accumulator capacity and the electrical flexibility available in the time step;
  - Net electricity consumption: total net electricity consumption (kWh), obtained from the derived calculation of the consumption of all devices and batteries, offset by the photovoltaic panel. The consumption of thermal accumulators is ignored as they are charged by their respective devices, whose consumption has already been recorded.
  - Indoor temperature set point: value of the indoor temperature setpoint (°C), obtained as raw data from the *building.csv* file, or dynamically modified in the presence of occupants via predictive and probabilistic models (Logistic Regression Occupant) [Nweye et al., 2024];
  - Occupant count: number of occupants present in the building at the current step, provided as raw data from the *building.csv* dataset;
  - Comfort band: acceptable thermal comfort band for the building (°C), provided as raw data from the *building.csv* dataset;
  - Power outage: indicates whether there is an electrical power outage (0 = no, 1 = yes), provided as random data from a stochastic model based on realistic reliability metrics such as SAIFI (System Average Interruption Frequency Index) and CAIDI (Customer Average Interruption Duration Index) [Nweye et al., 2024];

The state space is primarily continuous, as it consists mostly of continuous variables, with rare exceptions such as power outages. For simplicity, we haven't shown all possible observations, but only those most relevant to our research. In CityLearn, the transition model is predominantly deterministic: if the same actions are repeated under identical conditions, the same results are produced. This is because observations, such as meteorological conditions, are predefined in the datasets. However, there are exceptions, such as occupant behaviour or power outages, which are modelled as random events. We can also treat the indoor temperature as a random variable, since it is not possible to predict the LSTM's output with certainty.

2.2 Actions

In CityLearn, the action space is defined by the control actions that can be applied to various building devices, such as accumulators or HVAC (heating, ventilation, and air conditioning) systems. CityLearn supports central and decentralized agents, that are differentiated by how they manage observations and actions. We focused our application on the centralized agent that keeps in a single array all the possible actions for every building. The possible actions for each building are:

- **cooling_device** [0, 1]: proportion of the nominal power to be delivered by the cooling device;

- **dhw_storage** [−1, 1]: proportion of the domestic hot water storage system's capacity to charge ($a > 0$) or discharge ($a < 0$);

- **electrical_storage** $[-1, 1]$: proportion of the battery's capacity to charge ($a > 0$) or discharge ($a < 0$);

2.3 Reward Function

CityLearn offers a set of basic functions, while also allowing the user to add custom functions [Nweye et al., 2024]. It is important to note that, in CityLearn, the reward function is statically defined within the *schema.json* configuration file. For its efficiency and robustness, we chose to use the Solar Penalty Comfort Reward, which combines two base rewards into a weighted sum, defined as follows:

$$\mathrm{r_{tot}} = \alpha \cdot \mathrm{r_{solar}} + \beta \cdot \mathrm{r_{comfort}} \tag{1}$$

where α and β (by default both 0.5) are the coefficients of the weight of the following reward.

Comfort Reward

$$\begin{cases} -|T_{in} - T_{stp}|^3 & \text{if } T_{in} < (T_{stp} - T_b) \text{ and cooling} \\ -|T_{in} - T_{stp}|^2 & \text{if } T_{in} < (T_{stp} - T_b) \text{ and heating} \\ -|T_{in} - T_{stp}| & \text{if } (T_{stp} - T_b) \leq T_{in} < T_{stp} \text{ and cooling} \\ 0 & \text{if } (T_{stp} - T_b) \leq T_{in} < T_{stp} \text{ and heating} \\ 0 & \text{if } T_{stp} \leq T_{in} \leq (T_{stp} + T_b) \text{ and cooling} \\ -|T_{in} - T_{stp}| & \text{if } T_{stp} \leq T_{in} \leq (T_{stp} + T_b) \text{ and heating} \\ -|T_{in} - T_{stp}|^2 & \text{if } (T_{stp} + T_b) < T_{in} \text{ and cooling} \\ -|T_{in} - T_{stp}|^3 & \text{otherwise} \end{cases} \tag{2}$$

This is a complex function designed to calculate a reward based on the indoor temperature (T_{in}), the setpoint temperature (T_{stp}), and the comfort temperature band (T_b). It considers both cooling and heating devices. In our case it's an exclusive cooling device. Its purpose is to incentivize maintaining the indoor temperature within desired limits, penalize excessive deviations, and ultimately optimize thermal comfort.

Solar Penalty Reward

$$\sum_{i=0}^{n} -\left(\left(1 + \frac{e}{|e|} \times \textbf{storage}^{SoC}\right) \times |e|\right) \tag{3}$$

It's designed to incentivize behaviour that minimize electricity consumption from the grid and maximize the use of solar energy. The following formula is analysed below:

- $\frac{e}{|e|}$: this term serves to distinguish between net consumption ($e > 0$) and net production ($e < 0$), returning 1 or -1 respectively;
- $\textbf{storage}^{SoC}$: this represents the state of charge of the energy storage system, and in this context, it serves to modulate the reward based on the available storage capacity;

- $\left(1 + \frac{e}{|e|} \times \textbf{storage}^{SoC}\right)$: this component is used to increase the penalty when the building consumes energy and the storage system is not fully charged, or when the building produces energy to the grid and the storage system is not fully discharged;
- $|e|$: this takes the absolute value of net consumption to ensure that the penalty is proportional;

The summation iterates through all available n batteries and accumulators in the environment.

3 Background

Below, we will introduce and discuss the fundamental components of the Model-Based Reinforcement Learning (MBRL) algorithms we'll be covering. We'll start with their foundation, the Soft Actor-Critic (SAC) algorithm [Haarnoja et al., 2018], and then specifically introduce Dyna-Style architectures [Sutton, 1990].

3.1 Soft Actor Critic

Soft Actor-Critic (SAC) [Haarnoja et al., 2018] is a Deep Reinforcement Learning (DRL) algorithm characterized by its off-policy approach and entropy regularization. Furthermore, SAC is designed to operate effectively with continuous action spaces, making it ideal for complex environments like CityLearn, and justifying our choice.
The Soft Actor-Critic builds upon typical DRL principles but includes distinct features, notably entropy. Entropy in this context defines the randomness of the policy, the higher it is, the more unpredictable the policy's output becomes. While standard RL aims to maximize the expected sum of rewards, SAC goes further by also maximizing entropy. This encourages the policy to maintain a high degree of randomness, by promoting more robust exploration of the environment. The following formula of the optimal policy:

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho^{\pi}} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right] \tag{4}$$

where $\mathcal{H}(\pi(\cdot|s_t))$ it's the entropy of the policy and α it's a weight to choose its importance. The SAC algorithm also employs an Actor-Critic architecture, where a neural network (the actor) learns a stochastic policy that generates actions while maintaining a high degree of randomness. Meanwhile, one or more neural networks (the critic) estimate the Q-Value by evaluating the effectiveness of the executed actions. This approach is inspired by the policy iteration algorithm, which alternates between evaluation and improvement phases, allowing for continuous evolution of the policy. The implementation of SAC also performs an approximation of the Q-function and the policy, through the use of neural networks and stochastic gradient descent within the Actor-Critic architecture, to allow us to work with continuous spaces. Finally, the algorithm adopts an off-policy approach, using a replay buffer to store experiences collected during interaction with the environment. This allows the networks to be updated even by using data from policies different from the current one. CityLearn [Nweye et al., 2024] provides an implementation directly adapted to the environment, which performs normalizations and uses two critic neural networks to avoid overestimating the Q-value.

3.2 Probabilistic Ensemble Models

At the core of Model-Based Reinforcement Learning (MBRL) and the derived Dyna-Style concepts [Sutton, 1990] is the use of models to learn and approximate the real dynamics of the environment $p(s_{t+1} \mid s_t, a_t)$, creating a simulated $\tilde{p}(s_{t+1} \mid s_t, a_t)$. All the methodologies we've gathered refer to the use of probabilistic ensemble models (PE) [Lakshminarayanan et al., 2017]. In these models, you have E probabilistic neural networks (PNNs), each with parameters θ_e ($e \in \{1, ...E\}$), which are trained on a bootstrapped dataset via negative log-likelihood loss, to approximate the next Gaussian distribution for the next state.

$$\tilde{p}_{\theta_e}(s_{t+1} \mid s_t, a) = \mathcal{N}\big(\mu_{\theta_e}(s, a),\, \sigma^2_{\theta_e}(s, a)\big) \tag{5}$$

Specifically, in the practical implementation, all models in the dynamic ensemble are trained on the same buffer of real experiences, achieving their diversity through random weight initialization and the use of stochastic mini-batches sampled from the replay buffer ($\mathcal{D}_{\text{env}}$).

3.3 Dyna-Style Model-Based Reinforcement Learning

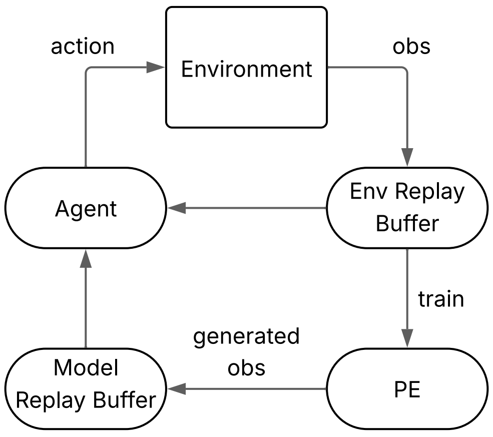

Figure 3: Scheme to summarize the behaviour of a Dyna-style MBRL

Dyna-style MBRL [Sutton, 1990] serves as the conceptual foundation for all the algorithms described in the following sections. In practice, a model-free agent (in our case, SAC) interacts with the environment, gathering real experiences into a replay buffer $\mathcal{D}_{env}$. Then, the collected data is used to train the dynamics model, $\tilde{p}$, which then generates experiences in branched model-based rollouts. These simulated experiences fill a separate replay buffer for modelled data $\mathcal{D}_{mod}$. At last, both replay buffers are used to train the model-free agent. The main characteristic of this approach is that it reduces the amount of data needed for training, increasing sample efficiency. However, this presents a critical challenge: the quality of the generated data and models must be high to produce valid samples for training.

4 Model-Based Policy Optimization

Model-based Policy Optimization (MBPO) [Janner et al., 2021] represents the state-of-the-art approach for Dyna-style MBRL [Sutton, 1990]. It extends the Dyna-style approach

by combining short branched model rollouts with an ensemble of probabilistic dynamics models to mitigate compounding prediction errors. At each iteration, a model-free SAC agent collects real transitions into a replay buffer, which are then used to train an ensemble of PNNs. From states sampled from the real replay buffer $\mathcal{D}_{env}$, MBPO executes M rollouts of T_{max}-step under the learned models, generating modelled trajectories that are stored in a separate model buffer $\mathcal{D}_{mod}$. Additionally, by updating the SAC agent with G_{max} gradient steps at each step [Chen et al., 2021], we can achieve improved sample efficiency.
The main contribution of MBPO [Janner et al., 2021] was to answer the question "When to trust your model?" by studying epistemic uncertainty, or uncertainty of the model parameters, which is crucial when data for model training are scarce and the model is exploited by the policy. One of the main solutions is to generate a prediction from the probabilistic ensemble models (PE), by uniformly choosing a random model e, allowing different modelled transitions $\tilde{p}$ in a single rollout to enhance inference.
Finally, they confirmed with experimentation [Janner et al., 2021] that fixing the rollout length T_{max} at 1 for the duration of training retains the bigger horizon. This, as demonstrated by theoretical analysis, is necessary to decrease compounding error.
In Figure 4, we try to summarize the behaviour that MBPO enacts in generating multiple rollouts. It should be noted that in experiments, T_{max} of 1 was used, in line with the developers' findings. However, for conceptual simplification, we show a T_{max} of 10 in the image. Furthermore, in the MBPO algorithm, there's no real evaluation of uncertainty, therefore, the ellipse representing the entire observation space should be treated as an ideal approximation of trajectory uncertainty. This will be useful for subsequent comparison with the following algorithms.

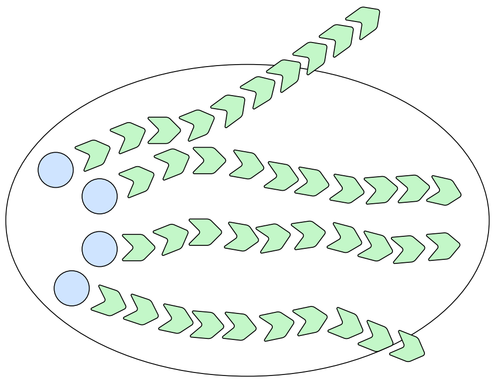

Figure 4: MBPO Rollout scheme example. Blue spheres represent initial observations sampled from the real environment. The larger ellipse represents the entire observation space of the environment. Arrows show a model-generated rollout. If it's green, it means it has been added to $\mathcal{D}_{\text{mod}}$.

Finally, let's define the implementation of the MBPO algorithm (Algorithm 1). This representation is derived from the MACURA developers (`https://github.com/Data-Science-in-Mechanical-Engineering/macura`), which is inspired by (`https://github.com/jannerm/mbpo`). It includes slight theoretical variations to standardize the notation across all discussed algorithms and to use a level of abstraction closer to the actual implementation. For completeness, the addition of noise [Eberhard et al., 2023] to the policy sample was defined in the pseudocode, but in reality, all experiments were

conducted without any type of noise, considering only the base entropy of SAC [Haarnoja et al., 2018].
Following a step taken in the real environment based on the generated action, every R (steps before retraining the model), the rollout mechanism, with its associated model management, is activated. This component begins with the training of the model ensemble, as described in [Lakshminarayanan et al., 2017], with the clarification that in our experiments, the reward function is also modelled. To simplify understanding the link between the pseudocode and the real implementation, it's important to note that M rollouts are not iterated sequentially in reality, but are handled in parallel through the use of batches. The rest of the pseudocode accurately describes the behaviour previously defined. All subsequent algorithms build their implementation upon these foundations.

Algorithm 1 Model-Based Policy Optimization (MBPO)

```
1: Initialize: dynamics model $\tilde{p}_\theta$, RL policy $\pi$, environment replay buffer $\mathcal{D}_{\text{env}} \leftarrow \emptyset$,
   model replay buffer $\mathcal{D}_{\text{mod}} \leftarrow \emptyset$, steps before retraining model $R$, number of model-
   based rollouts $M$, maximum SAC update steps $G_{\max}$, fixed $T_{\max}$, $E$ number of dy-
   namics model.
2: for each iteration do
3:     $s_0 \sim \rho_0(s)$
4:     for each environment step do
5:         $a_t \sim \pi(\cdot \mid s_t)$ with correlated exploration noise [Eberhard et al., 2023]
6:         $s_{t+1} \sim p(\cdot \mid s_t, a_t)$
7:         $r_{t+1} = r(s_t, a_t)$
8:         $\mathcal{D}_{\text{env}} \leftarrow \mathcal{D}_{\text{env}} \cup \{(s_t, a_t, r_{t+1}, s_{t+1})\}$
9:         if environment step % $R = 0$ then
10:            for each epoch do
11:                Train $\tilde{p}_\theta$ on $\mathcal{D}_{\text{env}}$
12:            Evict old data from $\mathcal{D}_{\text{mod}}$
13:            for each m in M model rollouts do
14:                $s_0^m \sim \mathcal{U}(\mathcal{D}_{\text{env}})$
15:            for t = 0,...,$T_{max}$ - 1 do
16:                for each m in M model rollouts do
17:                    $e_t^m \sim \mathcal{U}(1, \ldots, E)$
18:                    $a_t^m \sim \pi(\cdot \mid s_t^m)$
19:                    $s_{t+1}^m \sim \tilde{p}_{\theta_{e_0^m}}(s_{t+1} \mid s_t^m, a_t^m)$
20:                    $r_{t+1}^m \sim r(s_0^m, a_0^m)$
21:                    $\mathcal{D}_{\text{mod}} \leftarrow \mathcal{D}_{\text{mod}} \cup \{(s_t^m, a_t^m, r_{t+1}^m, s_{t+1}^m)\}$
22:        for $G_{\max}$ gradient steps do
23:            Update $\pi$ on $\mathcal{D}_{\text{mod}} \cup \mathcal{D}_{\text{env}}$
```

5 Masked Model-based Actor-Critic

In continuous control, MBPO [Janner et al., 2021] yields comparable results to state-of-the-art model-free methods with significantly fewer samples. However, MBPO seemingly contradicts the fundamental concept that MBRL should be most beneficial in low-data scenarios. In fact, MBPO suggests using the model when it is accurate and employing

short rollout horizons, which limits its applicability in deterministic and non-noisy environments. Thus, the question "When to trust your model?" remains open. Masked Model-based Actor-Critic (M2AC) [Pan et al., 2020] expands upon the MBPO idea by adding a masking mechanism that eliminates unreliable model-generated samples and performs effectively with longer T_{max} horizons.
A crucial component in this mechanism is the calculation of the model uncertainty. When choosing a random $\tilde{p}_{\theta_e}$ (where $e \in \{1, ...E\}$ is the index of the model chosen), we consider all the unchosen $\tilde{p}_{\theta}$, to compute a measure of both aleatoric and model uncertainties. The M2AC developers [Pan et al., 2020] present different methodologies, but we consider the suggested and used one. The One-vs-Rest (OvR) uncertainty estimation is a simple method that estimates model error by confronting it with the remaining models in the ensemble. Given (s_t, a) and a chosen index e we define the uncertainty as:

$$u_e(s_t, a) = D_{\mathrm{KL}} \left[\tilde{p}_{\theta_e}(\cdot \mid s_t, a) \,\|\, \tilde{p}_{\theta_{rest}}(\cdot \mid s_t, a) \right] \tag{6}$$

where:

- D_{KL} it's the KL-divergence between two Gaussian to estimate the uncertainty.
- $\tilde{p}_{\theta_e}(\cdot \mid s_t, a)$ refers to Equation 5 for the chosen dynamics model with index e.
- $\tilde{p}_{\theta_{rest}}(\cdot \mid s_t, a)$ is the merge of the rest dynamics model Gaussian distribution in the ensemble calculated:

$$\tilde{p}_{\theta_{rest}}(s_{t+1} \mid s_t, a) = \mathcal{N}\big(\mu_{\theta_{rest}}(s, a),\, \sigma^2_{\theta_{rest}}(s, a)\big)$$

$$\mu_{rest}(s, a) = \frac{1}{E-1} \sum_{i \neq e}^{E} \mu_{\theta_i}(s, a)$$

$$\sigma^2_{rest}(s, a) = \frac{1}{E-1} \sum_{i \neq e}^{E} \Big(\sigma^2_{\theta_i}(s, a) + \mu^2_{\theta_i}(s, a) \Big) - \mu^2_{rest}(s, a)$$

Knowing that, we can describe the masking mechanism that plays a key role in this method. Consider that the algorithm runs a minibatch of M model rollouts in parallel and has generated M trajectories (we consider all the trajectories at the same step t of the T_{max} horizon length) and to each we assign the computed uncertainty u_e. We rank these rollouts by their u_e score, and keep the first wM trajectories, where $w \in (0, 1]$ is the masking rate. In other words, it's the number that defines how many rollouts we keep. This is done for all the t-step made.
This is the core of the algorithm, but other enhancements have been proposed. You can add a linear function that applies a decay to w, reducing transition selection as the horizon increases and providing stronger filtering during the phase where error accumulates. The formula is as follows, starting with $w_t = 0.5$ at $t = 1$:

$$w_t = \frac{T_{\max} - t}{2\,\big(T_{\max} + 1\big)} \tag{7}$$

In addition, a model error penalty coefficient can be added to the reward, which, as suggested in the paper [Pan et al., 2020], is set to $\alpha = 0.001$.

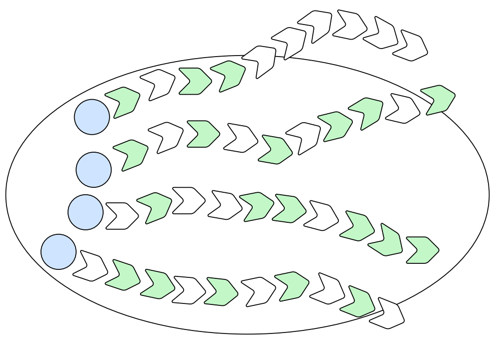

Figure 5: M2AC Rollout scheme example. Blue spheres represent initial observations sampled from the real environment. The larger ellipse represents the entire observation space of the environment. Arrows show a model-generated rollout. If it's green, it means it has been added to $\mathcal{D}_{\text{mod}}$.

In Figure 5, we summarize this behaviour and make it comparable with MBPO. Now, we can define the ellipses of the observation space as an ideal estimation of the uncertainty u_e to better comprehend the algorithm.
The following pseudocode describes the M2AC algorithm (Algorithm 1). Since the original authors did not publish an open-source implementation, we used the MACURA GitHub resources (`https://github.com/Data-Science-in-Mechanical-Engineering/macura`). As with the other methods, this is a simplified and uniform presentation designed to facilitate direct comparison with different MBRL algorithms.

Algorithm 2 Model-based Actor-Critic with Masked Rollouts and Uncertainty Filtering

```
1: Initialize: dynamics model $\tilde{p}_\theta$, RL policy $\pi$, environment replay buffer $\mathcal{D}_{\text{env}} \leftarrow \emptyset$,
   model replay buffer $\mathcal{D}_{\text{mod}} \leftarrow \emptyset$, steps before retraining model $R$, number of model-
   based rollouts $M$, maximum SAC update steps $G_{\max}$, fixed $T_{\max}$. masking rate $w$,
   model error penalty coefficient $\alpha$, $E$ number of dynamics model.
2: for each iteration do
3:     $s_0 \sim \rho_0(s)$
4:     for each environment step do
5:         $a_t \sim \pi(\cdot \mid s_t)$ with correlated exploration noise [Eberhard et al., 2023]
6:         $s_{t+1} \sim \rho(\cdot \mid s_t, a_t)$
7:         $r_{t+1} = r(s_t, a_t)$
8:         $\mathcal{D}_{\text{env}} \leftarrow \mathcal{D}_{\text{env}} \cup \{(s_t, a_t, r_{t+1}, s_{t+1})\}$
9:         if environment step % $R = 0$ then
10:            for each epoch do
11:                Train $\tilde{p}_\theta$ on $\mathcal{D}_{\text{env}}$
12:            Evict old data from $\mathcal{D}_{\text{mod}}$
13:            for each m in M model rollouts do
14:                $s_0^m \sim \mathcal{U}(\mathcal{D}_{\text{env}})$
15:            for $t = 0, \ldots, T_{\max} - 1$ do
16:                for each m in M model rollouts do
17:                    $a_i \sim \pi(s_i)$
18:                    Compute $\{\tilde{p}_{\theta_{e_t^m}}(r, s' \mid s_i, a_i)\}_{i=0}^{E}$
19:                    $e_t^m \sim \mathcal{U}(1, \ldots, E)$
20:                    $a_t^m \sim \pi(\cdot \mid s_t^m)$
21:                    $s_{t+1}^m \sim \tilde{p}_{\theta_{e_t^m}}(\cdot \mid s_t^m, a_t^m)$
22:                    $r_{t+1}^m \sim r(s_t^m, a_t^m)$
23:                    Compute One-vs-Rest uncertainty $u_e^m$ (6)
24:                Rank samples by $u_e^m$, select $\lfloor wM \rfloor$ with smallest uncertainty
25:                $\{s_i, a_i, r_i, s_{i+1}\}_{i=1}^{\lfloor wM \rfloor}$
26:                $\mathcal{D}_{\text{model}} \leftarrow \mathcal{D}_{\text{model}} \cup \{(s_i, a_i, r_i - \alpha u_i, s_i)\}_{i=1}^{\lfloor wM \rfloor}$
27:        for $G_{max}$ gradient steps do
28:            Update $\pi$ on $\mathcal{D}_{\text{mod}} \cup \mathcal{D}_{\text{env}}$
```

6 Model-based Actor-Critic with Uncertainty-aware Rollout Adaption

This section will introduce and analyse the structure of the Model-based Actor-Critic with Uncertainty-aware Rollout Adaption (MACURA) algorithm [Frauenknecht et al., 2024]. The algorithm, starting from Dyna-style MBRL [Sutton, 1990] and MBPO [Janner et al., 2021], provides an uncertainty-aware mechanism to adapt the rollout length dynamically, ensuring stable and monotonic policy improvement. MACURA aims to answer the question "When to trust your model?" by studying the uncertainty of PE models that allow for adaptive rollout length, to terminate those rollouts that have accumulated too much error.

So, at the core of the algorithm is the definition of model uncertainty. To achieve this, the Jensen-Shannon (GJS) divergence [Nielsen, 2019], a symmetric version of the KL divergence, is used and defined as follows:

$$u_{\text{GJS}}(s,a) = \frac{2}{E(E-1)} \sum_{e=1}^{E} \sum_{f=1}^{e-1} D_{\text{GJS}}\left(\mathcal{N}_e \| \mathcal{N}_f\right) \tag{8}$$

where:

- $\mathcal{N}_i = \mathcal{N}\big(\mu_{\theta_i}(s,a), \Sigma_{\theta_i}(s,a)\big)$ similar defined in Equation 5.
- D_{GJS} : Jensen-Shannon geometric divergence defined as:

$$D_{\text{GJS}}\left(\mathcal{N}_e \| \mathcal{N}_f\right) = \frac{1}{2} D_{\text{KL}}\left(\mathcal{N}_e \| \mathcal{N}_{ef}\right) + \frac{1}{2} D_{\text{KL}}\left(\mathcal{N}_f \| \mathcal{N}_{ef}\right) \tag{9}$$

where $\mathcal{N}_{ef}$ it's the merge of two Gaussians defined as:

$$\mathcal{N}_{ef}\big(\mu_{ef}(s,a),\, \Sigma_{ef}(s,a)\big)$$

$$\mu_{ef}(s,a) = \sigma_{ef}^2(s,a)\, \Big(\tfrac{1}{2}\, \sigma_{\theta_e}^2(s,a)^{-1}\, \mu_{\theta_e}(s,a) \;+\; \tfrac{1}{2}\, \sigma_{\theta_f}^2(s,a)^{-1}\, \mu_{\theta_f}(s,a)\Big).$$

$$\Sigma_{ef}(s,a) = \left(\tfrac{1}{2}\, \Sigma_{\theta_e}(s,a)^{-1} \;+\; \tfrac{1}{2}\, \Sigma_{\theta_f}(s,a)^{-1}\right)^{-1}$$

Summing up the previous definition, uncertainty is calculated by measuring how much the models disagree with each other on a given state-action pair. Specifically, it takes all possible pairs of models and, for each pair, calculates the GJS divergence (Equation 9) between their predictions, then measures the average of all these distances.
This definition is fundamental to define $\mathcal{E}$, which represents parts of the observation space where we can trust the model. An ideal $\mathcal{E}$ is not computable, especially in CityLearn, so this uncertainty helps us have a practical definition. Based on this, we can define a threshold κ that allows us to reject all trajectories leaving the space $\mathcal{E}$ (with $u_{GJS} < \kappa$), which is now defined as follows:

$$\mathcal{E} := \{s \in \mathcal{S} \,|\, u_{\text{GJS}}(s,a) < \kappa,\; a \sim \pi(\cdot \mid s)\} \tag{10}$$

In this case, $\mathcal{E}$ can be well represented in Figure 6 as the larger ellipses.

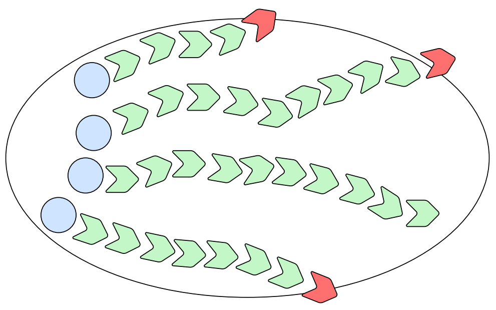

Figure 6: MACURA Rollout scheme example. Blue spheres represent initial observations sampled from the real environment. The larger ellipse represents the entire observation space of the environment. Arrows show a model-generated rollout. If it's green, it means it has been added to $\mathcal{D}_{\text{mod}}$, otherwise if it's red, it means that the rollout has been stopped.

We now focus on the algorithm in detail (Algorithm 0). In the first iteration of M rollouts, only the initial t-step ($t = 0$) is performed for all rollouts, allowing the extraction of the tuple $(s_0^m, a_0^m, e_0^m, s_1^m, r_1^m)$. This tuple allows for the calculation of an initial uncertainty, which is crucial because it defines the threshold κ that stops the rollout horizon. But in this calibration phase, the control $u_{\text{GJS}}(s, a) < \kappa$ cannot be performed; instead, in the absence of the dynamic threshold κ, the current k rollout rounds performed are used. Since this is the first rollout step, high errors can be generated; therefore, this is resolved by imposing this basic initial threshold. Once these constraints are met, $\mathcal{D}_{mod}$ can be populated with the newly generated rollout. Once that's done, it is possible to calculate an initial threshold, which will be a moving average that adapts to the model's confidence. We define it as follows:

$$\kappa = \frac{\xi}{K} \sum_{k=1}^{K} \tilde{u}_{\text{GJS},k} \tag{11}$$

where:

$$\tilde{u}_{\text{GJS},k} = \inf \left\{ u_{\text{GJS}}(s_0, a_0) \in \{u_{\text{GJS}}(s_0^1, a_0^1), \ldots, u_{\text{GJS}}(s_0^M, a_0^M)\} : \zeta \leq \text{CDF}_k(u_{\text{GJS}}(s_0, a_0)) \right\}$$

$\tilde{u}_{\text{GJS},k}$ represents the minimum uncertainty calculated, above which a fraction ζ (e.g., 95%) of the uncertainties calculated in the initial rollout transitions are found. This allows the separation of the more uncertain transitions $(1 - \zeta)$ from the more reliable ones (ζ).
Next, a mean of the uncertainties $\tilde{u}_{\text{GJS},k}$ from all completed K rollout rounds, scaled by a factor ξ, is calculated, becoming the new threshold κ for this rollout round.
Once the uncertainty threshold is calculated, the algorithm begins generating rollouts for each model in the ensemble, up to a maximum horizon of $T_{max} - 1$. If a rollout exceeds the uncertainty threshold, the trajectory is discarded and the rollout is stopped, otherwise, it is saved in $\mathcal{D}_{mod}$. The T_{max} horizon has a new purpose: as defined by the developers [Frauenknecht et al., 2024], it helps optimize code performance by preventing the generation of excessively long rollouts.
After the rollouts are generated, the policy can be updated with $\mathcal{D}mod$ and $\mathcal{D}env$ G times, where G is now defined as:

$$G = \left\lfloor G_{\max} \frac{|\mathcal{D}\text{mod}|}{|\mathcal{D}\text{mod}|\text{max}} \right\rceil \tag{12}$$

This formula dynamically regulates the number of policy update steps for the underlying SAC algorithm [Haarnoja et al., 2018], based on the amount of data available in the model buffer ($|\mathcal{D}mod|$) relative to its capacity ($|\mathcal{D}mod|$max). This allows for a balanced update-to-data ratio, preventing divergence or overfitting [Chen et al., 2021].
The following pseudocode (Algorithm 0) outlines the MACURA algorithm in a clear and structured way. This formulation allows for easier comparison with the other MBRL methods presented in this work.

Algorithm 3 Model-based Actor-Critic with Uncertainty-aware Rollout Adaption (MACURA)

```
1: Initialize: dynamics model $\tilde{p}_\theta$, RL policy $\pi$, environment replay buffer $\mathcal{D}_{\text{env}} \leftarrow \emptyset$, model replay buffer $\mathcal{D}_{\text{mod}} \leftarrow \emptyset$, steps before retraining model $R$, number of model-based rollouts $M$, maximum SAC update steps $G_{\max}$, rounds of rollouts performed thus far $K$, $\pi$, fixed $T_{\max}$, $\zeta$, $\xi$, $E$ number of dynamics model.
2: for each iteration do
3:     $s_0 \sim \rho_0(s)$
4:     for each environment step do
5:         $a_t \sim \pi(\cdot \mid s_t)$ with correlated exploration noise [Eberhard et al., 2023]
6:         $s_{t+1} \sim \rho(\cdot \mid s_t, a_t)$
7:         $r_{t+1} = r(s_t, a_t)$
8:         $\mathcal{D}_{\text{env}} \leftarrow \mathcal{D}_{\text{env}} \cup \{(s_t, a_t, r_{t+1}, s_{t+1})\}$
9:         if environment step % $R = 0$ then
10:            $K \leftarrow K + 1$
11:            $k \leftarrow K$
12:            for each epoch do
13:                Train $\tilde{p}_\theta$ on $\mathcal{D}_{\text{env}}$
14:            Evict old data from $\mathcal{D}_{\text{mod}}$
15:            for each $m$ in $M$ model rollouts do
16:                $s_0^m \sim \mathcal{U}(\mathcal{D}_{\text{env}})$
17:                $e_0^m \sim \mathcal{U}(1, \ldots, E)$
18:                $a_0^m \sim \pi(\cdot \mid s_0^m)$
19:                $s_1^m \sim \tilde{p}_{\theta_{e_0^m}}(s_1 \mid s_0^m, a_0^m)$
20:                $r_1^m \sim r(s_0^m, a_0^m)$
21:                $u_{\text{GJS}}(s_0^m, a_0^m)$ according to (8)
22:                if $u_{\text{GJS}}(s_0^m, a_0^m) < k$ then
23:                    $\mathcal{D}_{\text{mod}} \leftarrow \mathcal{D}_{\text{mod}} \cup \{(s_0^m, a_0^m, r_1^m, s_1^m)\}$
24:                else
25:                    stop rollout $m$ and discard data
26:            updating $\kappa$ according to (11)
27:            for $t = 1, \ldots, T_{\max} - 1$ do
28:                for each $m$ in $M$ model rollouts do
29:                    $e_t^m \sim \mathcal{U}(1, \ldots, E)$
30:                    $a_t^m \sim \pi(\cdot \mid s_t^m)$
31:                    $s_{t+1}^m \sim \tilde{p}_{\theta_{e_t^m}}(\cdot \mid s_t^m, a_t^m)$
32:                    $r_{t+1}^m \sim r(s_t^m, a_t^m)$
33:                    $u_{\text{GJS}}(s_t^m, a_t^m)$ according to (8)
34:                    if $u_{\text{GJS}}(s_t^m, a_t^m) < \kappa$ then
35:                        $\mathcal{D}_{\text{mod}} \leftarrow \mathcal{D}_{\text{mod}} \cup \{(s_t^m, a_t^m, r_{t+1}^m, s_{t+1}^m)\}$
36:                    else
37:                        stop rollout $m$ and discard data
38:            for $G$ gradient steps according to (12) do
39:                Update $\pi$ on $\mathcal{D}_{\text{mod}} \cup \mathcal{D}_{\text{env}}$
```

7 Experiments and Discussion

Now, starting from the theoretical knowledge we've gained, let's discuss the experiments conducted with various algorithms within the CityLearn environment. We'll evaluate their training performance, evaluation metrics, and, where relevant, the parameters that define the building's status. This was made possible by the implementations found at `https://github.com/intelligent-environments-lab/CityLearn/tree/master/citylearn` for the environment and `https://github.com/Data-Science-in-Mechanical-Engineering/macura` for all the algorithms. Additionally, we'll consider the Community-Based Hierarchical Energy Systems Coordination Algorithm (CHESCA) [Garmendia et al., 2024], the winner of the CityLearn Challenge 2023, as our optimal solution benchmark. The CHESCA code was found at `https://github.com/TheLeprechaun25/CHESCA/tree/main`.

7.1 Setup

Let's start by describing the setup and all its functionalities, beginning with the file structure and then analysing the components managed by each file. It's important to note that this entire setup has been designed to be uploaded to Google Drive and subsequently executed on Google Colab. This doesn't prevent local execution, but some adaptations will be necessary, especially for the *.ipynb* files. The choice of Google Colab allows the use of GPUs, which are useful for improving execution times (especially for dynamics models).

In Figure 7, we can see the main structure. This is where we find the main folders and all the *.ipynb* files that execute the various algorithms, returning the general performance metrics of a building (e.g., temperature over time, battery state of charge, or general consumption), along with the performance of the RL and MBRL agents. Theoretically, to run an execution, you only need to create a new *.ipynb* file and follow the instructions.

```
src
├─ rewards
├─ wrappers
├─ agents
│  ├─ model_based
│  └─ chesca
├─ utils
├─ old
└─ MACURASimulation.ipynb
```

Figure 7: General Directory Structure : *MACURASimulation.ipynb* is an example. There a lots more *.ipynb* files

The full CityLearn environment isn't present in the main folders because it's not necessary; you can just install the module via *pip*. If you prefer, you can also download the package

directly from GitHub. For structures and specifics on how to interface with the environment, refer directly to the CityLearn documentation (`https://www.citylearn.net/`) or use the other *.ipynb* files as examples. The most important dependency to manage for importing CityLearn is NumPy, in fact, it's important to load version *1.23.5*, to avoid other problems.
Starting with the *rewards* folder, it contains all the *.py* files representing various rewards used for different tests. Prior to this work, a reward analysis was conducted, and its results are in the *old* folder. The reward we focused on is in *CityLearnReward.py.* Theoretically, these rewards are already present in the environment's module (`https://github.com/intelligent-environments-lab/CityLearn/blob/master/citylearn/reward_function.py`), but due to some compatibility issues with CityLearn version 2.3.0, these were resolved and uploaded in this separate file. The same applies to the wrappers inside the *wrappers* folder, which have no external purpose beyond this. Instead, the *utils* folder contains all the components useful for plotting data for various parameters, which was made possible using the Pandas and Matplotlib libraries.
Finally, there's the *agents* folder, which contains the CHESCA agent and all the remaining MBRL algorithms. Starting with CHESCA, the folder was downloaded directly into the setup without any issues. Interfacing with it is also notably suitable for CityLearn, as it directly has a callable *agent.py* file that is completely compatible with CityLearn, just like the base-implemented agents.
Regarding the model-based folder (downloaded from `https://github.com/Data-Science-in-Mechanical-Engineering/macura`), it wasn't as straightforward. Since it was designed to work with Mujoco environments [Todorov et al., 2012], some modifications were necessary within the files. Consequently, we'll describe the structures in detail where principally modified for our setup (Figure 8). This module needs the installation of PyTorch libraries. Starting with the *exp* folder, it stores all the results from the MBRL and SAC algorithm executions, with the purpose of collecting all the information and grouping it for comparative plotting. For each algorithm, we store:

- In *train.csv*, all parameters related to the training of the SAC agent are stored, including the average batch reward, critic loss, and actor loss;
- In *model_train.csv*, all parameters related to the training of the ensemble of probabilistic neural networks, including model loss and validation score, are stored;
- In *results.csv*, all parameters related to the evaluation of the learned policy are stored, which principally consider sum reward;

In addition, we can find some *.pth* files that are used to load checkpoints of previous executions. Obviously, SAC only has the train and results files.
The core of the algorithms is located in the *mbrl* folder. Starting with the *util* subfolder, you'll find all the fundamental files for execution, notably *common.py* and *env.py.* The most crucial modifications for our setup were made in *env.py.* This file is responsible for loading and adapting environments within the code. We've integrated CityLearn into this component by adding it to the list within the *_legacy_make_env* function, implementing a series of changes to ensure compatibility. After initializing the environment and statically setting the Solar Penalty and Comfort Reward, we then applied two wrappers already present within CityLearn's *wrappers.py* module. First, we apply the *NormalizedSpace* wrapper to normalize the state and action spaces, making the application of models and agents as efficient as possible for the provided code. Then, we apply the *StableBaselines3*

wrapper, which is designed to make the environment compatible with algorithms from the StableBaselines3 library; it also provides a Gym interface that can be used by the MACURA developers' code.

```
model_based
├─ exp
│  ├─ macura
│  │  ├─ train.csv
│  │  ├─ results.csv
│  │  └─ model_train.csv
│  ├─ mbpo
│  ├─ m2ac
│  └─ sac
└─ mbrl
   ├─ algorithms
   ├─ example
   │  ├─ overrides
   │  ├─ algorithm
   │  ├─ dynamics_model
   │  └─ main.yaml
   ├─ third_party
   └─ util
      ├─ common.py
      └─ env.py
```

Figure 8: MBRL Directory Structure

Meanwhile, *common.py* has remained unchanged, but it's important to note that this module contains all the useful and basic functions common to all algorithms, such as executing a step and adding to the replay buffer (*step_env_and_add_to_buffer*).
Regarding the actual implementations of the algorithms, these are contained in the *algorithm* folder. Each file (for example, *macura.py*) has an almost identical structure; the differentiation occurs within the rollout function, which is unique to each MBRL algorithm. For SAC, we opted to use MACURA's implementation instead of CityLearn's one, for a more accurate comparison and to leverage our existing logging system. Upon download, there wasn't a similar direct implementation for SAC like MACURA, so it was necessary to create a new one. Originally, SAC was located within

the *third_party* folder, which the MACURA developers collected from this GitHub repository: https://github.com/pranz24/pytorch-soft-actor-critic. However, it wasn't callable like the other algorithms. Therefore, we created this new module, which reuses the logic of the existing one, but makes it easier for us to call. Finally, a crucial modification was made to all these algorithms: during the policy update, there was a *reverse_mask* flag set to true, which led to performance degradation, so it has been deactivated for all algorithms.
Finally, we have the *example* folder, which contains all the essential *.yaml* files for configuring the hyperparameters of the various algorithms. To use these files, you'll need to install the Hydra and OmegaConf dependencies. Inside, each algorithm has its own *main.yaml* file that references the directories of its components. The *overrides* folder contains files with the main parameters that tune algorithms for specific environment, we'll look at the specifics in later sections. In the algorithm and *dynamics_model* folders, you'll find the fixed parameters for each algorithm. We've removed the configuration files related to MuJoCo for organizational purposes. However, on the GitHub repository, you can easily find all the configurations used, including those for MuJoCo, and an example *.yaml* file to help you understand the meaning of each parameter https://github.com/Data-Science-in-Mechanical-Engineering/macura/blob/master/How_to_start_experiments/Config_files_explained.yaml.
Beyond what we've discussed, there is also an initial attempt at building-specific environment factorization and applying MACURA in a multi-agent-like scenario, called F-MACURA. This is an intermediate and undefined implementation, but it serves as a starting point for future works.
Lastly, there are experiments with various algorithms under different noise conditions [Eberhard et al., 2023], but these aren't considered in this particular text.

7.2 Hyperparameters

Before introducing the results, we need to describe the various parameters identified through grid-search, justifying their selection in some cases. First, we want to remind that the choice of the training (*citylearn_challenge_2023_phase_2_local_evaluation*) and evaluation (*citylearn_challenge_2023_phase_2_online_evaluation_3*) datasets is motivated by our desire to place ourselves in the same situation as the competitors of the CityLearn Challenge 2023 [AIcrowd et al., 2023], providing a solid basis for comparison with the winner CHESCA [Garmendia et al., 2024]. We don't evaluate on the private dataset like the challengers did, mainly due to the difficulty of adapting a policy for three buildings to one with six, but this could be future work.
The choice for the total number of steps per epoch and per episode was (Table 1) driven by the need for consistent log outputs and enough data for result analysis. An evaluation episode (2207 steps long, not shown in the table) is run every 1015 training steps, and this frequency defines the duration of each epoch. The initial exploration steps are performed solely to initially populate the real replay buffer for training the dynamics model, and they do not modify the policy.
Next, through a grid-search, we identified the values for R and M to meet the implementation constraint of M2AC (simplified as: Steps per epoch = $R \times M$) and to ensure consistency across all experiments. Finally, we set $G_{max} = 10$ for all methods, except for MACURA. For MACURA, we recommend $G_{max} = 20$, as a variable G_{max} value could excessively reduce the number of updates. As defined by the developers, we opted

to set a real data ratio ($\mathcal{D}_{env}$) to 5%, using model-generated data ($\mathcal{D}_{mod}$) exclusively for the remaining 95% of updates. In MACURA, following the paper's recommendations [Frauenknecht et al., 2024], we identified $\xi = 0.3$ as the ideal and slightly conservative value for trajectory retention.
In MBPO, the implementation by the MACURA developers includes a rollout horizon schedule (e.g., it performs 1 rollout during the first 20 epochs, while in the following epochs it increases up to 15 rollouts). In our hyperparameters, we chose to adopt the MBPO [Janner et al., 2021] suggested choice of fixing the rollout horizon to one.

Table 1: Hyperparameters Algorithm

Parameter	**MBPO**	**M2AC**	**MACURA**
Number of steps		22330	
Steps per epoch		1015	
Steps per episode		719	
Initial exploration step		719	
Frequency of training model (R)		29	
Effective model rollouts per step (M)		35	
SAC Updates G_{max}	10	10	20
Real data ratio		0.05	
Masking Rate (w)	N/A	0.5	N/A
Xi (ξ)	N/A	N/A	0.3
Zeta (ζ)	N/A	N/A	95
Rollout length T_{max}	1	10	10

When it comes to the models (Table 2), we opted to keep the default parameters, as they proved to be efficient for our application, despite having conducted various tests.

Table 2: Hyperparameters Model

Parameter	**Model**
Ensemble size	7
Number of elite models	7
Patience	10
Improvement threshold	0.01
Weight Decay	0.00005
Learning Rate	0.0003
Validation Ratio	0.2
Model layers	4
Model Hidden Size	256
Model Batch Size	256

The parameters in the Table 3, however, are the same for all the considered algorithms and were determined through multiple attempts. The most interesting parameter is the target entropy, which is defined as $-(dim(A))$, where A is the action space. In our case, this corresponds to -9 (as there are 3 actions per building). This was suggested by the CityLearn implementation [Nweye et al., 2024]. To tune the model-free SAC, we also chose to include $G_{max} = 10$ to allow for a more precise comparison with the model-based version.

Table 3: Hyperparameters SAC

Parameter	**SAC**
Gamma (γ)	0.99
Tau (τ)	0.005
Alpha (α)	0.2
Layer Norm	True
Learning Rate	0.003
Policy Distribution	Gaussian
Target Entropy	-9
Critic Layers	3
Critic Hidden Size	1024
Critic Batch Size	256
Actor Layers	3
Actor Hidden Size	1024
Actor Batch Size	256

7.3 Results

Given the setup and the chosen hyperparameters, we can now begin our analysis of the various results. Starting with Figure 9, we've opted to plot the average batch rewards against the number of updates performed by SAC, in accordance with MACURA's paper [Frauenknecht et al., 2024]. We want to highlight that this doesn't necessarily coincide with the number of steps taken; in fact, MACURA is an exception, which we will analyse later. Furthermore, we chose to use CHESCA as a baseline to assess the optimality of convergence. However, since it is primarily a heuristic [Garmendia et al., 2024], it's not possible to have equivalent training values. Therefore, for training, we used the average reward of the entire episode, while for evaluation, we used the sum of the rewards of the episode.
As we might expect, the algorithms achieve identical, or at least similar, convergence values, but they differ at the beginning. It's important to note that in the initial phase (0-20000), probably due to the limited amount of information in the buffer, the models are more uncertain, which leads to greater prediction errors.
This significantly impacts M2AC, which, lacking a sufficiently suitable filtering mechanism, tends to retain uncertain data, leading to worse performance even than SAC. This is also true for MBPO, which, however, having a short horizon, avoids accumulating too much error, preventing excessive worsening of results. MACURA, on the other hand,

shows good performance from the start, outperforming SAC even from the very first updates. This is mainly due to its mechanisms that block and preserve performance from these initial errors, keeping it in a waiting phase until it gains more certainty about the models. This will be detailed better in Figure 11.

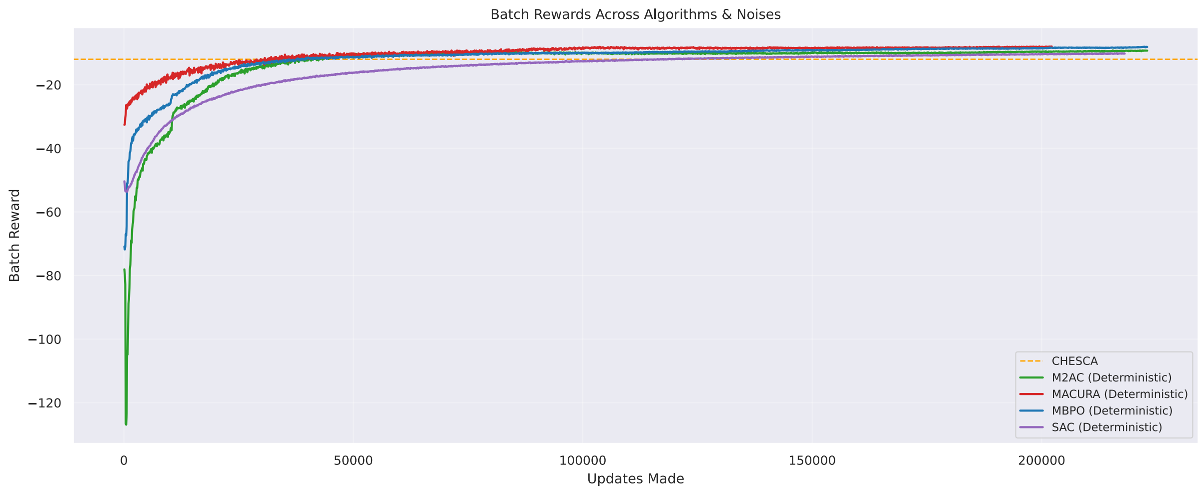

Figure 9: Batch reward across model-based RL algorithms: each curve shows the mean batch reward per update step for MBPO, M2AC, MACURA and SAC (solid lines). The dashed horizontal line marks the baseline performance of CHESCA.

To better analyse the behaviours following this initial phase, we've decided to cut the first 20,000 steps, arriving at Figure 10. This allows us to zoom in and conduct a more in-depth analysis. At this point, the models are already more confident, leading all MBRL algorithms to achieve better performance compared to SAC.
Regarding MACURA, the situation remains the same, showing a lower number of updates by the end. This is due to the previously mentioned situation in which MACURA tends to be more conservative, thereby reducing the number of updates performed (we recall that the steps in the environment are equivalent for all algorithms).
Meanwhile, the behaviour of M2AC and MBPO is more dynamic. Initially, MBPO seems to maintain an advantage over M2AC (0-50000), until they reach a point of equivalent or even slightly better performance for M2AC (50000-80000). However, this doesn't last long, as MBPO achieves better results towards convergence. This is likely due to the fact that, in the beginning, M2AC's models generate more confident data with a longer horizon, potentially leading to more training data than MBPO at a certain point. Unfortunately, this isn't enough to compensate for the accumulated error, which results in a very low convergence value and seems to not allow for improvement over base SAC. However, we must clarify that M2AC adds a model error penalty (α) to the reward, which might lead to an apparently lower convergence during training, even when it might be equivalent or even superior in the final phases. Unfortunately, it was not possible to eliminate this coefficient as it modifies the core of the algorithm. Therefore, while M2AC is surely worse during the initial phase, it is probably capable of the same or even better performance than MACURA later on. Meanwhile, MBPO's short horizon rollout, with increased model safety, adds a slight generalization that leads to better convergence compared to SAC, but loses in sample efficiency. At this point, MACURA and MBPO are similar, with the difference that MACURA achieves this point with fewer updates. The initially more

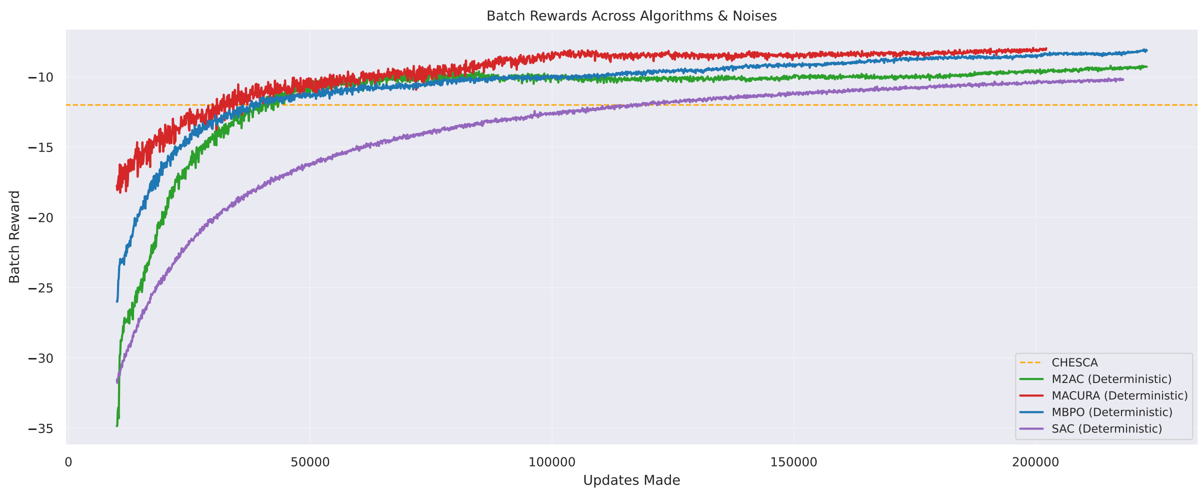

Figure 10: Batch reward across model-based RL algorithms: each curve shows the mean batch reward per update step for MBPO, M2AC, MACURA and SAC. This time we cut the first 10000 to zoom out on convergence performance. The dashed horizontal line marks the baseline performance of CHESCA.

permissive use of M2AC's model is justified by its masking mechanism. Although this mechanism can select a more confident slice of the rollouts, it is still not sufficient to correct the unfiltered models whose reliability is not certain.
Overall, we can observe significant sample efficiency compared to the base SAC, which justifies our initial goal. This leads us to consider MACURA, at least during training, as the algorithm that best and most safely leverages the model-based architecture.
Regarding the test results (shown in the Figure 11), logs were taken every 1015 steps in the real training environment, and we display the sum of rewards for the entire episode in the evaluation environment, after which we reset the execution. Here, at first, the results seem completely opposite to those before (Figure 9), but a real analysis must be performed.
As for MACURA, although we observed significantly better performance at the very start of training, it initially turns out to be the worst algorithm in those initial stages. As mentioned earlier, MACURA employs mechanisms that justify this behaviour. In the early phases, when the model is still highly uncertain, MACURA enters a sort of "waiting" state, during which it performs no updates even as environment steps continue. This results in a policy that has hardly been updated at the beginning, causing it to underperform during evaluation. However, MACURA quickly recovers: after this unstable initial phase, during which it competes with MBPO and SAC, it stabilizes and ultimately outperforms both in the final results.
Speaking of M2AC, we can see that its initial instability lines up with the training phase, but convergence tells a very different story. Whereas before (Figure 10) M2AC converged to the worst performance of all, it now emerges as the best, despite its slight stability issues. This makes sense if we revisit the earlier discussion about the penalty coefficient. In fact, the values were previously penalized, but in this test they are no longer present. That suggests, during training, that at convergence M2AC achieves results equal to or better than MACURA.
Meanwhile, for MBPO and SAC, their behaviours remain similar. While their initial

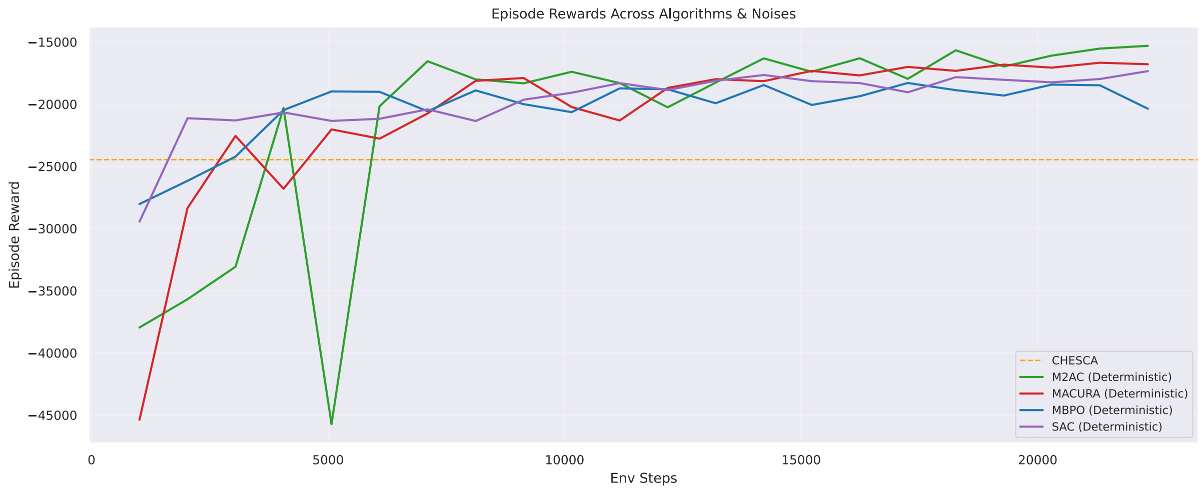

Figure 11: Final episode sum rewards for each algorithm: plot summarizing the evaluated episode reward in environment step from MBPO, M2AC, MACURA and SAC. The dashed horizontal line marks the baseline performance of CHESCA.

performance aligns with the training phase, SAC appears to slightly outperform MBPO at convergence. This is likely due to limited generalization or inefficiency resulting from using a single-horizon rollout, which proves insufficient. Nonetheless, the gap between the two is minimal, reinforcing the idea that, apart from the slight advantage in sample efficiency, there's little else to distinguish them.
A question that might arise, however, is: All results surpass CHESCA, so have we beaten the winners? The answer is absolutely not. While we used the Solar Penalty and Comfort Reward as our comparison metric for all algorithms, the CityLearn Challenge 2023 was based on far more complex building parameters. With our reward, we focus almost exclusively on discomfort and consumption, whereas the challenge evaluated other parameters such as emissions and blackout resilience. Below, we provide the general score formula; for more details, please consult the challenge page [AIcrowd et al., 2023]:

$$Score_{Control} = 0.3 \cdot Score^{Comfort}_{Control} + 0.1 \cdot Score^{Emissions}_{Control} + 0.3 \cdot Score^{Grid}_{Control} + 0.3 \cdot Score^{Resilience}_{Control}$$

An example of this can be the use of batteries. Considering MBPO (Figure 13), which does not differ from the other algorithms discussed, we can see how battery usage is completely disincentivized, as it can be complex to manage and can easily lead to inadequate consumption behaviour for an RL algorithm. Meanwhile, for CHESCA (Figure 12), which is a more complex heuristic, it makes regular use of batteries, leading, for example, to reduced emissions and increased resilience. However, this is not reflected by the reward, which favours MBPO given its reduced consumption. Even if we underestimate CHESCA's performance, it still provides a good baseline against which to compare the algorithms studied, but it's important to remember this detail.

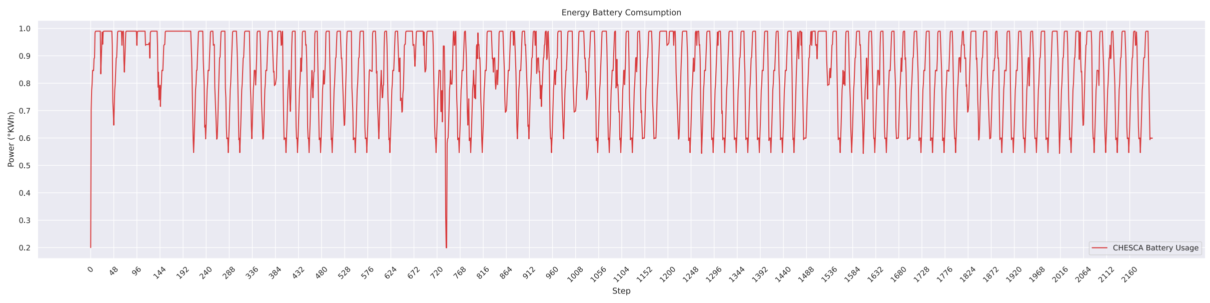

Figure 12: CHESCA: Battery state of charge over environment steps

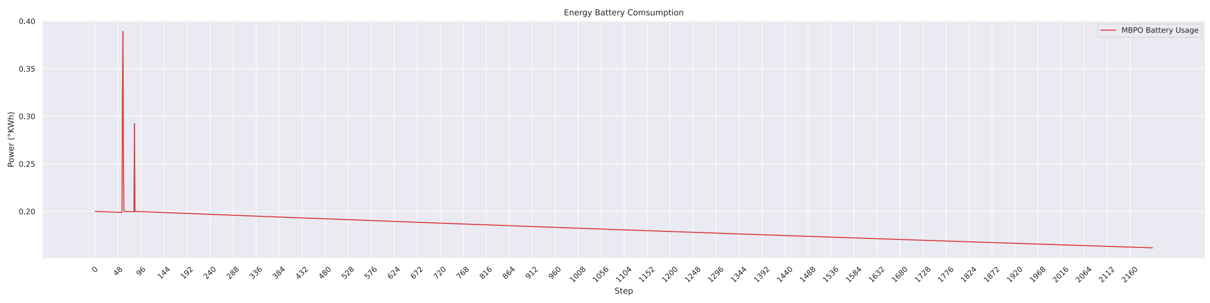

Figure 13: MBPO: Battery state of charge over environment steps

8 Conclusion

At this point, after analysing each algorithm and its performance, we can say that MBRL algorithms can provide real sample efficiency in CityLearn, thus requiring fewer steps to learn an optimal policy. In conclusion, we can consider two algorithms. If we simply desire good sample efficiency, MACURA can be the most stable and efficient from this perspective. However, if we also want an improvement in performance, at the cost of more steps, M2AC addresses this problem as shown in the results. That said, these changes are insufficient to truly compete in the CityLearn Challenge 2023, but they could be a good starting point for future works.

References

[AIcrowd et al., 2023] AIcrowd, Intelligent Environments Lab, and Energy Efficient Cities Initiative (2023). Neurips 2023 citylearn challenge. `https://www.aicrowd.com/challenges/neurips-2023-citylearn-challenge`.

[Chen et al., 2021] Chen, L., Lu, K., and Abbeel, P. (2021). Understanding the role of update-to-data ratio in model-free reinforcement learning. *Advances in Neural Information Processing Systems*, 34.

[Eberhard et al., 2023] Eberhard, O., Hollenstein, J., Pinneri, C., and Martius, G. (2023). Pink noise is all you need: Colored noise exploration in deep reinforcement learning. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR)*.

[Frauenknecht et al., 2024] Frauenknecht, B., Eisele, A., Subhasish, D., Solowjow, F., and Trimpe, S. (2024). Trust the model where it trusts itself – model-based actor-critic with uncertainty-aware rollout adaption.

[Garmendia et al., 2024] Garmendia, A., Morri, F., Cappart, Q., and Cadre, H. (2024). Winning the 2023 citylearn challenge: A community-based hierarchical energy systems coordination algorithm.

[Haarnoja et al., 2018] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Janner et al., 2021] Janner, M., Fu, J., Zhang, M., and Levine, S. (2021). When to trust your model: Model-based policy optimization.

[Lakshminarayanan et al., 2017] Lakshminarayanan, B., Pritzel, A., and Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Nielsen, 2019] Nielsen, F. (2019). On the geometric jensen-shannon divergence. *Entropy*, 21(5):485.

[Nweye et al., 2024] Nweye, K., Kaspar, K., Buscemi, G., Fonseca, T., Pinto, G., Ghose, D., Duddukuru, S., Pratapa, P., Li, H., Mohammadi, J., Lino Ferreira, L., Hong, T., Ouf, M., Capozzoli, A., and Nagy, Z. (2024). Citylearn v2: energy-flexible, resilient, occupant-centric, and carbon-aware management of grid-interactive communities. *Journal of Building Performance Simulation*, 0(0):1–22.

[Pan et al., 2020] Pan, F., He, J., Tu, D., and He, Q. (2020). Trust the model when it is confident: Masked model-based actor-critic.

[Sutton, 1990] Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the seventh international conference on machine learning*, pages 216–224.

[Todorov et al., 2012] Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE.