On the Importance of Hyperparameter Optimization for Model-based Reinforcement Learning

Baohe Zhang
University of Freiburg

Raghu Rajan
University of Freiburg

Luis Pineda
Facebook AI Research

Nathan Lambert
UC Berkeley

André Biedenkapp
University of Freiburg

Kurtland Chua
Princeton University

Frank Hutter
University of Freiburg and Bosch Center for AI

Roberto Calandra
Facebook AI Research

Abstract

Model-based Reinforcement Learning (MBRL) is a promising framework for learning control in a data-efficient manner. MBRL algorithms can be fairly complex due to the separate dynamics modeling and the subsequent planning algorithm, and as a result, they often possess tens of hyperparameters and architectural choices. For this reason, MBRL typically requires significant human expertise before it can be applied to new problems and domains. To alleviate this problem, we propose to use automatic hyperparameter optimization (HPO). We demonstrate that this problem can be tackled effectively with automated HPO, which we demonstrate to yield significantly improved performance compared to human experts. In addition, we show that tuning of several MBRL hyperparameters dynamically, i.e. during the training itself, further improves the performance compared to using static hyperparameters which are kept fixed for the whole training. Finally, our experiments provide valuable insights into the effects of several hyperparameters, such as *plan horizon* or *learning rate* and their influence on the stability of training and resulting rewards.

Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS) 2021, San Diego, California, USA. PMLR: Volume 130. Copyright 2021 by the author(s).

1 Introduction

Model-based Reinforcement Learning (MBRL) is a powerful framework for learning to solve continuous control tasks. Recent work in MBRL has demonstrated performance comparable to traditional model-free algorithms, using only a fraction of the data (Williams et al., 2017; Chua et al., 2018; Kurutach et al., 2018; Amos et al., 2018). Modern MBRL algorithms are complex tools which rely on several sub-components: the training of a dynamics model, the prediction and propagation of state-action trajectories, and the nested optimization of a policy or the planning of the action sequence. Each of these sub-components, in turn, requires the tuning of several design parameters that can significantly impact performance. As a result, MBRL algorithms require a high degree of human expertise to configure before they can be applied to new tasks.

The observation that hyperparameter tuning can be a tedious and domain-specific task which requires high levels of expertise and computational cost is not unique to MBRL. Hyperparameter Optimization (HPO) emerged as a promising and quickly growing approach to automate the training of machine learning models (see e.g. the survey by Feurer and Hutter, 2019). HPO has been successfully applied to several domains such as natural language processing (Melis et al., 2018) or deep RL (Henderson et al., 2018) to improve the performance and robustness of trained models which are highly dependent on their hyperparameter settings. Although HPO has been previously used in conjunction with model-free RL algorithms (Jaderberg et al., 2017; Henderson et al., 2018; Paul et al., 2019), to the best of our knowledge, HPO has not yet been evaluated in the MBRL setting. Using HPO with MBRL poses challenges as MBRL is often used in the low-data

regime, which constrains HPO to be as data-efficient as possible, and MBRL typically has a larger number of hyperparameters that have unknown interaction effects between each other. Additionally, MBRL provides another exciting challenge to HPO algorithms through its potential non-stationarity, where during the learning process different hyperparameter settings are required to improve learning quality. Commonly, HPO assumes a stationary process, where a single hyperparameter setting works best during the whole learning process. In this paper, we propose to utilize (dynamic) HPO for MBRL to drastically reduce the need for human expertise and further improve performance.

Specifically, we demonstrate: 1) *automatic* HPO can outperform human experts in tuning MBRL algorithms, achieving new state-of-the-art performance (to the point of training in one experiment an agent so successful that it "breaks down" the simulator); 2) *dynamic* HPO can further improve the performance compared to *static* HPO; 3) the best hyperparameter configurations differ across MBRL scenarios; 4) the dramatic effect that different hyperparameter choices have in MBRL.

By demonstrating the value that HPO can bring to MBRL approaches, we envision a future where HPO can be widely adopted to drive further progress in the MBRL community, while at the same time creating a new research direction and challenges to stimulate the HPO community.

2 Related Work

HPO Random search and grid search are simple and commonly used methods for performing HPO (Bergstra and Bengio, 2012). A more informed framework for HPO is Bayesian Optimisation (BO; Snoek et al., 2012; Bergstra et al., 2011; Springenberg et al., 2016; Hutter et al., 2011). BO sequentially optimizes an algorithm's hyperparameters by using previous evaluation results to select the next set of hyperparameters to evaluate. The previous evaluations are used to build a surrogate which models the relationship between hyperparameter values and resulting performance. Then, an acquisition function is used to trade off exploration and exploitation. The sequential nature of BO and use of full function evaluations however can make it very costly. Multi-fidelity HPO (Kandasamy et al., 2019) speeds up HPO by using evaluations of an algorithm on a cheaper, low fidelity to judge how an algorithm would perform when evaluated on an expensive, high fidelity. For example Klein et al. (2017) used the number of datapoints as fidelity, when tuning the hyperparameters of an SVM. On smaller subsets of the data many more hyperparameter configurations can be evaluated, compared to evaluating only a few configurations on the full data for the same cost, given that performance on the small dataset is correlated with performance on the full dataset. Successive Halving (SH; Jamieson and Talwalkar, 2016) used random search to sample a set of configurations to be run on the lowest fidelity and only the best performing configurations continue to be optimized further on higher ones. This allows SH to speed up the search for well performing hyperparameters. However, it is not always clear what minimum budgets would serve as good fidelities for the full budget. To overcome this, Hyperband (Li et al., 2017) launches multiple Successive Halving runs with different minimum budgets. BOHB (Falkner et al., 2018) further builds on Hyperband and BO by using a surrogate model to guide Hyperband's search procedure. The approaches discussed so far only search for *static* hyperparameter settings. Population based training (PBT; Jaderberg et al., 2017), on the other hand, was proposed to facilitate *dynamic* optimization of non-stationary processes. Instead of resulting in a static configuration, which is constant throughout a whole run of the optimizee, PBT changes the configuration at multiple stages. Thus, PBT can adapt hyperparameters to a potentially changing search landscape.

HPO for model-free RL RL is known for its sensitivity to its hyperparameters as well as its neural architecture choices. Islam et al. (2017) described the impact of hyperparameters and network architectures for policy gradient methods. They showed, for example, the impact the activation can have on the achievable reward of TRPO and DDPG agents. Henderson et al. (2018) followed with a broader study, including more algorithms with a larger search space. Multiple hyperparameter optimization methods have been used to optimize model-free reinforcement learning agents on a broad variety of tasks. Falkner et al. (2018) proposed to use BOHB to efficiently search for well performing hyperparameter settings and neural architectures of PPO. Similarly, under the heading of AutoRL, Runge et al. (2019) used BOHB for joint optimization of an RL agent's hyperparameters and neural architecture, whereas Chiang et al. (2019) proposed a two stage approach to optimize the agent hyperparameters and the network architectures. The presented methods so far, however, ignored a crucial issue for RL: the possible non-stationarity of the RL problem induces a possible non-stationarity of the hyperparameters. Thereby, at different stages of the learning process, different hyperparameter settings might be required to behave optimally. For A3C-like model-free RL methods on various tasks, PBT dynamically optimized hyperparameters and achieved better performance over a random search baseline (Jaderberg et al., 2017). Paul et al. (2019) proposed an alternative to PBT for policy gradient

algorithms by utilizing the properties of policy gradient methods and learn a schedule for the hyperparameters via importance sampling.

HPO for model-based RL To the best of our knowledge, we are the first to study hyperparameter optimization in the context of model-based reinforcement learning. Wang et al. (2019) presented an extensive benchmarking study of different MBRL agents. To compare the agents fairly, they only used static grid searches over the hyperparameters. They report only the performance of the optimized agents and do not quantify the impact of the used grid-searches.

3 Background

Among the HPO methods, we introduce PBT here in more detail because we provide an in-depth evaluation of novel settings for its training data usage and HPO methodology later.

3.1 Population Based Training

Population based training (PBT) is an evolutionary approach for dynamic HPO and allows to optimize hyperparameters during the training of the members of its population. PBT starts out with a randomly initialized population. As a result, all members of its population start from different regions in the hyperparameter configuration space. The members are ranked according to their current performance at regular intervals. The worst performing members in the population are then replaced by the best ones, by copying over their parameters (network weights, in case of neural networks NNs)), as well as their hyperparameters in an *exploitation step*. It is unclear from the original PBT paper, however, whether the data on which the members are trained are also copied over. We discuss ablations of this setting in the experiments section. To allow searching for potentially better performing hyperparameter configurations, the copied hyperparameters are perturbed to allow for small steps in their immediate neighborhoods in an *exploration step*. Over time, different configurations are evaluated *during* the training process and potentially kept if they improve performance. PBT comes with its own hyperparameters. In the original paper, members from the population are selected using *truncation* during the exploitation step. Thereby, agents from the bottom 20% of the population are replaced by agents from the top 20%. Additionally, continuous hyperparameter values are multiplied at random by 0.8 or 1.2 in the exploration step.

Algorithm 1: Generic Model-based RL framework

Input: Number of trials, n, dataset, D, policy, π
Output: Dynamics model $\hat{P}(s_{t+1}|s_t, a_t)$
Initialize dataset, D, from random actions
Set trial counter, $i = 0$
while $i < n$ **do**
    Train dynamics model $\hat{P}(s_{t+1}|s_t, a_t)$ on D
    while *trial not done* **do**
        Optimize objective using $\hat{P}$ to obtain π
        Execute π and collect data D' with controller using P in real environment
        Aggregate dataset $D = D \cup D'$

3.2 Model-based Reinforcement Learning

We briefly introduce the reinforcement learning (RL) framework in the finite time horizon setting. Let $(\mathcal{S}, \mathcal{A}, P, r)$ be a Markov Decision Process, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, P represents transition dynamics, and r is the reward function. Furthermore, let T denote a finite time horizon. The objective is then to find a policy π that maximizes the expected sum of rewards given by

$$\mathbb{E}\left[\sum_{t=0}^{T} r(s_t, a_t)\right], \tag{1}$$

where $s_{t+1} \sim P(\cdot|s_t, a_t)$ and $a_t \sim \pi(\cdot|s_t)$.

RL algorithms can be roughly divided into two categories, *model-based* and *model-free*, depending on whether they do or do not build a model of the environment dynamics, respectively. In the model-based approach, considered in this work, we first use data to learn a dynamics model, $\hat{P}$, approximating the true transition dynamics, P. Once this model is learned, a model-based method then optimizes the objective (Equation 1), using $\hat{P}$ in place of P. The resulting policy is then executed in the environment, typically for a limited number of steps to reduce the effect of compounding modeling errors, and the optimization/execution process is repeated in an inner iteration loop termed a *trial*. The outer loop of learning and trials are interleaved for n trials. Algorithm 1 summarizes the general MBRL framework. For more information on MBRL we refer the reader to (Deisenroth and Rasmussen, 2011; Chua et al., 2018).

4 HPO for MBRL

Reinforcement learning involves challenging optimization problems due to the stochasticity of evaluation, high computational cost and possible non-stationarity of the hyperparameters. Therefore, we now discuss

some significant design decisions facing an RL practitioner.

4.1 Comphrehensive Considerations

Transferability across environments When transferability across *environments* is the preferred goal of a set of hyperparameters, users expect them to be broadly applicable to related tasks. For example, when training agents to play a variety of Atari games, hyperparameters are set such that the agent performs well on a variety of different games (Mnih et al., 2015). Alternatively, we could choose the performance on a specific task to be the main focus of hyperparameter tuning. In the context of MBRL this could, for example, be motivated by the dynamics model which is specific to the environment at hand, often requiring very different hyperparameter settings to be learned as accurately and data-efficiently as possible (Wang et al., 2019). Thus hyperparameter settings might not always transfer between environments. Conversely, desirable hyperparameter settings might be those that give us the best performance on specific tasks. As a consequence, when choosing between available HPO methods for MBRL, we need first to decide if we require robust configurations that transfer or if we require highly specific configurations that achieve the best possible result on a particular task.

Transferability across multiple runs The required robustness of the found configuration has consequences on the nature of the HPO method best suited for the current task. Dynamic configurations can be expected to be less transferable across even *multiple runs* than static ones. By design, dynamic configurations make many more choices about the parameter settings than static configurations, thus making it very challenging to tune dynamic configurations by hand and without automatic HPO methods. With an increasing number of decision points, it becomes more and more likely that each choice we make is specifically tailored to the environment and even the current run at hand.

4.2 Design Choices

Considered Methods Based on the mentioned considerations, we decided to investigate three approaches for the hyperparameter tuning of MBRL agents. Namely, we use Hyperband as a multi-fidelity approach; PBT as a dynamic configuration method; and random search as a baseline in HPO. As mentioned in Section 3, for PBT, we additionally evaluate the effect of copying not only the model but also the training data when members are exploited, such that the dynamics model is not learned from drastically different data. We further consider a variation of PBT which we dub *PBT with backtracking (PBT-BT)*, to gain further insights into dynamic configuration. PBT-BT is similar to the work by Dasagi et al. (2019), which allows agents to backtrack to checkpoints before performance dropped significantly, with the exception that PBT-BT not only backtracks the model parameters but also the hyperparameters. We expect this modification to reduce the typical performance drops seen for RL training curves and to better explore unexplored parts of the hyperparameter search space from previously seen good configurations. For implementation details and hyperparameter settings of the used HPO methods, we refer to Appendix A.2.

Handling Stochasticity of Evaluation In the context of MBRL, it is possible to optimize the hyperparameters based on the model loss, which is usually the Mean Squared Error (or likelihood for probabilistic models) of one-step predictions from fitting the model. However, Lambert et al. (2020) showed that the correlation between model fitting and final returns can be weak. Alternatively when tuning the hyperparameters we can use the average returns over n steps as HPO objective. If n is too small we might be prone to accept highly volatile hyperparameter settings as well performing, whereas if n is too large we might make little progress in identifying truly well-performing configurations.

Handling Computational Demands As discussed in Section 2, traditional HPO methods such as BO can be computationally demanding. Thus, when dealing with expensive MBRL problems we need to decide how we can limit the additional computational overhead required for HPO. From the HPO literature we know that multi-fidelity approaches (Li et al., 2017) as well as PBT allow for efficient parallelization and provide significant improvements.

5 Experimental Evaluation

To address the importance of automatic HPO and validate our hypothesis of the non-stationarity of MBRL, we compare automatic HPO results to hyperparameters tuned by human experts and analyze the performance gain of dynamic tuning over static tuning.

5.1 Setting

Optimizee To demonstrate the importance of HPO for MBRL we use the current state-of-the-art *Probabilistic Ensembles With Trajectory Sampling (PETS)* (Chua et al., 2018) algorithm as optimizee. PETS uses an ensemble of neural networks to learn a model of the environment which provides aleatoric and epistemic

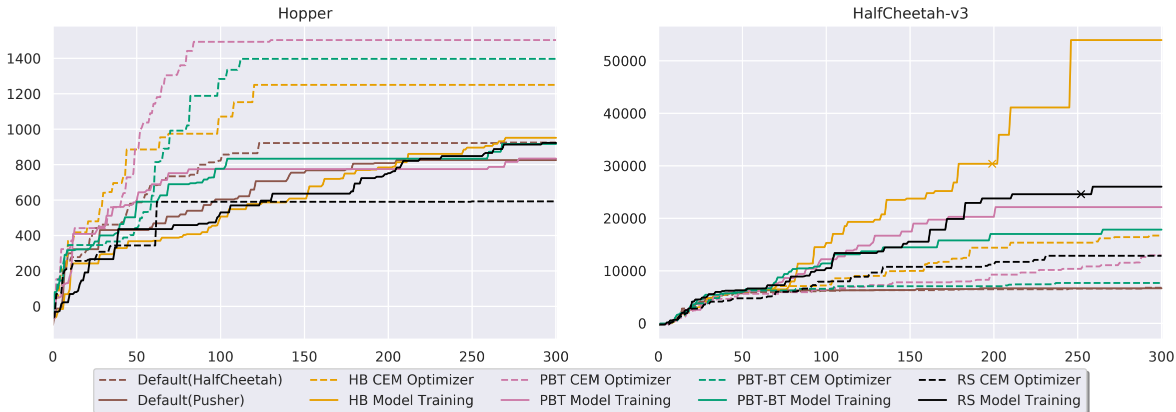


Figure 1: Evaluation of the *best* found hyperparameters and schedules from all search methods. Scores (y-axis) are the maximum of the average return over 5 evaluations over number of trials (x-axis). Brown lines are the baselines using the hyperparameters tuned by a human expert. Markers in HalfCheetah show that the evaluation started to encounter a bug in Mujoco which caused numerical overflow. Hyperparameter schedules do not transfer across environments or even across runs on the same environment (for the latter, compare performances in the transfer experiments here with the actual HPO experiments in Figure 4).

uncertainty estimates. In PETS, the dynamics model is chosen to be an ensemble of neural networks whose outputs parameterize anisotropic Gaussians. Model Predictive Control (MPC) is then used to get the policy, by directly optimizing the expected sum of rewards over a fixed planning horizon. PETS, in particular, performs model predictive control with a cross-entropy method (CEM) optimizer for action selection. To evaluate an action sequence, PETS first samples a model from the ensemble, and rolls out the action sequence using the selected model, and computes the sum of rewards. Action sequences are then evaluated by performing this process multiple times iteratively and averaging the simulated returns over the ensemble members.

Environments For the experiments, we consider four test environments: *Pusher*, *Reacher*, *Hopper*, *Halfcheetah* from MuJoCo (Todorov et al., 2012) and a simulation environment of *Daisy*, a robot hexapod to accomplish locomotion tasks. The reward signal for *Daisy* is similar to Hopper and HalfCheetah, where we use the forward speed as the reward signal. The number of trials is fixed to 80 for pusher and 300 for the rest, with rollout lengths of 150 and 1 000 steps, respectively. The actions in the initial trial are sampled randomly to collect data to train the model before it is used by PETS in future trials. We use Hopper and HalfCheetah for illustrative plots in the main paper; quantitative results for the other three tasks can be found in Appendix A.5.

Configuration Space We split the hyperparameters of PETS into two groups: (i) *Model Training* and (ii) *CEM Optimizer*; to clearly differentiate the influence these parameter spaces have in the MBRL setting. This allows optimizers to learn interaction effects within each group. When optimizing one group of hyperparameters, the others are set to the default value of the best manually tuned PETS hyperparameters as reported by Chua et al. (2018). We also optimized all hyperparameters together in Appendix A.4. The full configuration spaces are given in Appendix A.3.

HPO Objective We use the average returns of the 3 most recent trials as the objective for all HPO methods. This gives us a better estimate of the noisy reward in the MBRL tasks. As discussed in Section 4, we consider two scenarios: 1) the transferability of the hyperparameter schedule (static or dynamic) learned by the HPO methods across environments; and 2) where we are interested in the final learned model and policy reward across multiple runs. In the first scenario, we consider the mean performance of the top 5 members during the search. For the latter scenario, we use the best found schedules of each HPO method and report the performance of PETS over 5 seeds.

5.2 Importance of HPO

To demonstrate the importance of HPO in the context of MBRL, we use the default hyperparameters tuned (on Pusher and on HalfCheetah) by a human expert (Chua et al., 2018) as a baseline. We compare the performance of these baselines to the best hyperparameter settings found through means of each HPO method, see Figure 1. If HPO was of low importance in MBRL, we would expect the hand-crafted hyperparameter settings to transfer well between environments

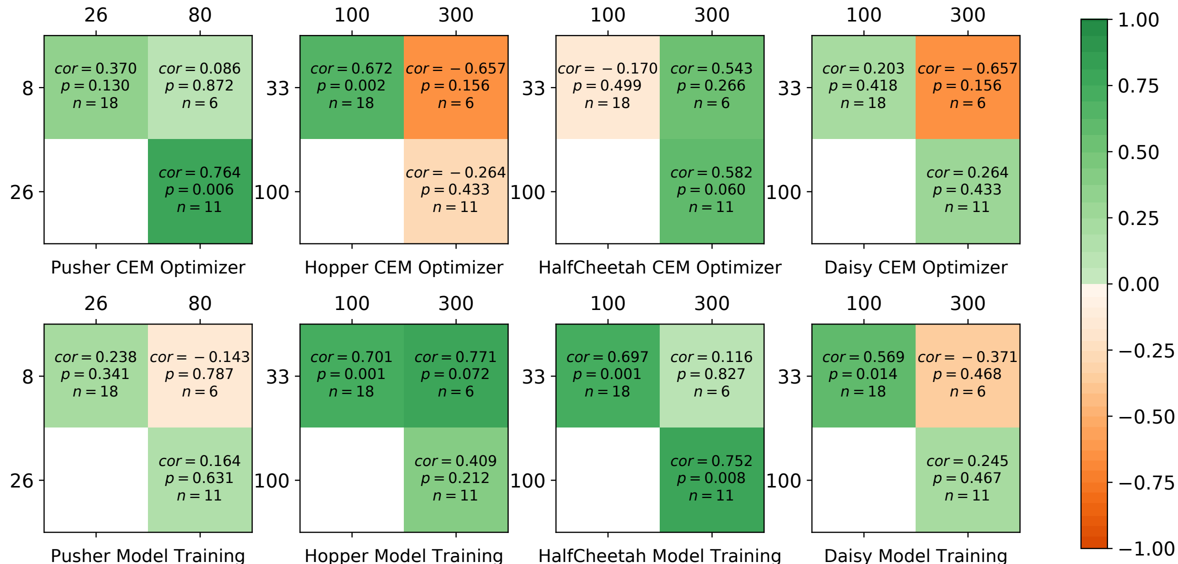


Figure 2: Spearman rank correlation of hyperparameters sampled by Hyperband across fidelities (i.e., number of trials) when training agents with 8 and 80 trials as the minimal/maximal budgets for Pusher and 33 and 300 trials for the rest of the environments. *cor* is the correlation coefficient, p is the p-value for the hypothesis test and n gives the number of configurations trained in both fidelities. Weak or even negative correlation across different budgets are found in most of the environments, which shows different configurations perform best for different training durations and that dynamic tuning may be needed.

and to perform competitively with the hyperparameter settings found by HPO. Instead, we can observe that both default hyperparameter settings are outperformed by configurations obtained either through static or dynamic HPO. In Figure 1, the Hopper and HalfCheetah environments show respectively a $1.5 - 2\times$ and $10\times$ improvement in final returns of HPO tuned agents over manually configured baselines which themselves involved substantial manual effort. The gains over manually tuned agents are not always so significant as can be seen in Appendix A.5, especially for simpler environments, however, automatic tuning always outperforms manual tuning and saves significant overhead.

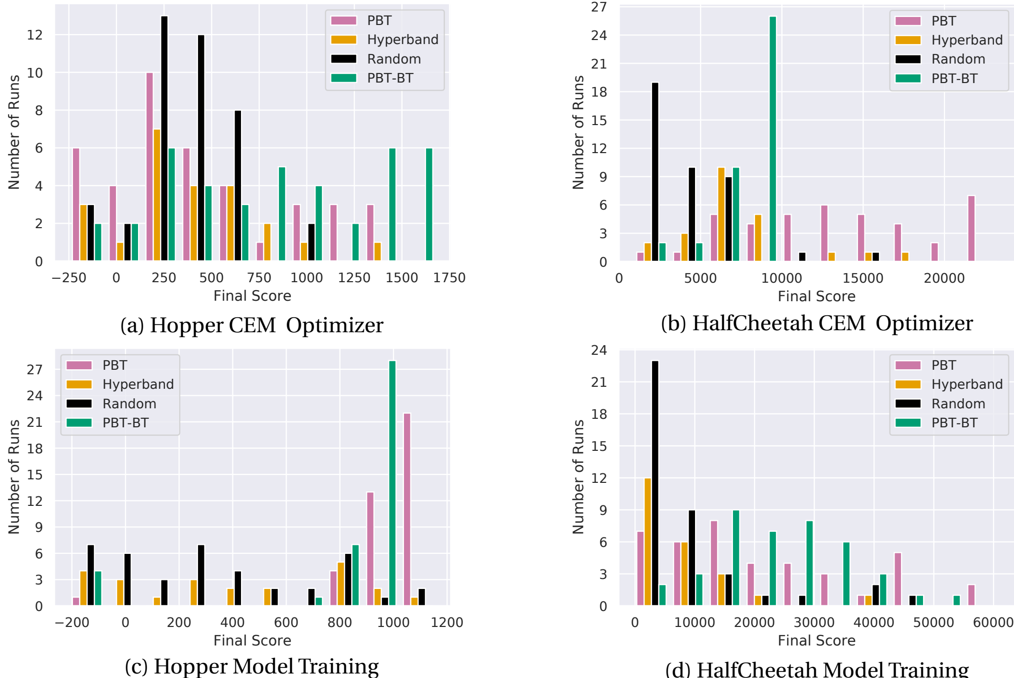
5.3 Need for Dynamic Tuning

We now use Hyperband to motivate the need for dynamic tuning. As fidelities in this setting we consider the number of trials used to train PETS. We show that different configurations perform well across different fidelities in many instances. This reveals the possible need for dynamic tuning. We calculate the Spearman correlation between the ranks of different configurations based on different fidelities of a Hyperband run (see Figure 2). The correlations are very small or even negative, indicating that *static* hyperparameter settings that are best for smaller fidelities do not perform as well or even worse on higher fidelities. Further we can observe for nearly all environments that configurations found on the lowest fidelity have a lower correlation to the highest fidelity than to the middle fidelity. This suggests that different hyperparameter configurations work best for different training durations and that we may need to tune hyperparameters dynamically in order to perform better. The outlier seems to be the model training parameters on hopper, which are highly correlated throughout. One possible reason is the impact of model training parameters on the performance is relatively low in Hopper, also shown in Figure 1.

5.4 Breaking the Simulation

When tuning the hyperparameters of PETS we observed two surprising results in the HalfCheetah environment. The task for the agent in this environment is to control a "Cheetah" such that it runs as fast as possible to the right. Before learning to run, the agent first has to learn to move right, then learn to walk and finally it can learn to run. Besides those expected gaits some of our tuned agents were able to learn a new *rolling* gait[1]. With this gait the Cheetah continually cartwheels and can build up far higher speeds than with running, allowing it to reach extremely high rewards. Some runs are so fast in fact that the simulator breaks with an overflow error (see Figures 1 & 4).

[1] Video provided in the supplementary material.



(a) Hopper CEM Optimizer

(b) HalfCheetah CEM Optimizer

(c) Hopper Model Training

(d) HalfCheetah Model Training

Figure 3: Histogram of achieved final episode return on Hopper and HalfCheetah over all configurations of CEM hyperparameters (top) and model training (bottom). We see that the importance of hyperparameters differs in different environments and that PBT can allocate more resources in well-performing areas during exploitation.

5.5 Effects of Hyperparameters

We observed that the effect of HPO of model training vs CEM optimizer hyperparameters varied between the environments. While HPO of CEM hyperparameters led to more significant gains on Hopper, for HalfCheetah model parameters were more important to tune.

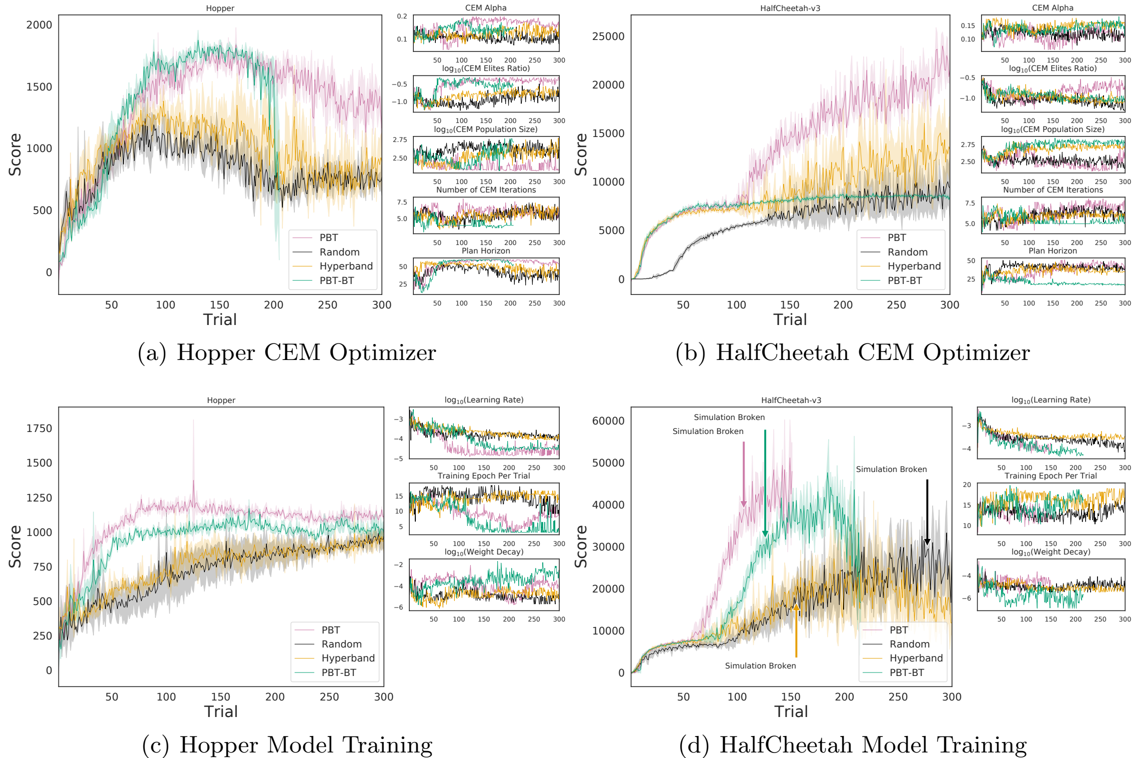
CEM Hyperparameters The impact on final performance of the CEM optimizer hyperparameters is not as large as that of model hyperparameters (see Figures 3). Comparatively fewer runs for both Hopper and HalfCheetah result in low rewards (around 0) for the HPO of CEM hyperparameters, which shows that an inaccurate model will lead to catastrophic performance, whereas a weak planner will still be able to have mediocre performance. Tuning *CEM optimizer* hyperparameters using PBT-BT, however does not allow agents to learn the rolling behaviour in HalfCheetah: most PBT-BT and random search agents result in a final score around 10 000. This can be attributed to the backtracking mechanism, which replaces stumbling or falling agents, resulting in a worse reward, with ones that have a better running behavior. Unexpectedly, falling and stumbling can be converted into the rolling gait, resulting in much higher rewards.

Model Training Hyperparameters Figures 3c & 3d show that results vary widely between different hyperparameter configurations, and the score of the best configurations can be orders of magnitude larger than the score of the worst ones. Relying on static tuning methods, including random search or Hyperband, to tune the model parameters results in most evaluated configurations not being able to learn any significant behaviour, most often resulting in close to zero reward. This suggests that the model parameters need to be chosen much more carefully to train a successful agent.

5.6 Trends of Hyperparameters

Figure 4 depicts the learning performance of agents on the Hopper and HalfCheetah tasks. Additionally, for each hyperparameter, we plot plausible hyperparameter schedules by taking the mean of the top 5 members' values for those hyperparameters.

Trends on Hopper We observe that dynamic tuning methods perform much better than static tuning methods for comparable budgets. Regarding individual hyperparameters, we observe that the *Plan Horizon* does not have a clear trend in the beginning but increases later for all methods. This might be due to the fact that when the model has been trained for more epochs, it becomes more reliable, allowing for better long-term planning. We discuss more on this in Appendix A.6. Similar to Wang et al. (2019) we observe that the best static configurations choose a final plan



(a) Hopper CEM Optimizer

(b) HalfCheetah CEM Optimizer

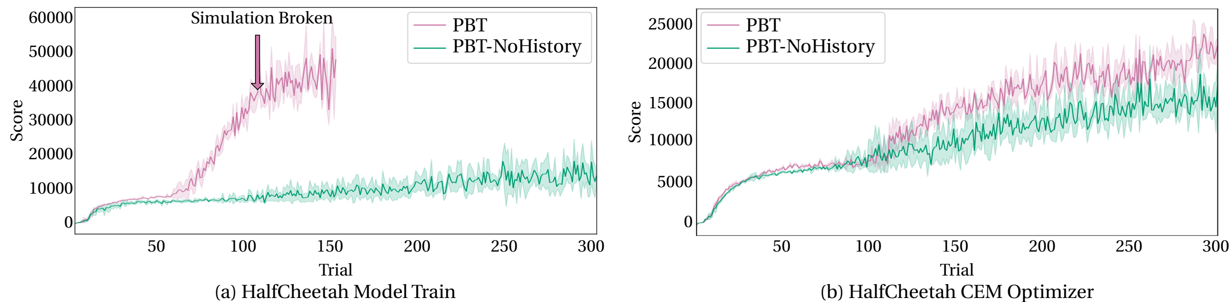
(c) Hopper Model Training

(d) HalfCheetah Model Training

Figure 4: Training curves for PBT, Hyperband and Random Search on Hopper and Halfcheetah. We reported the mean and standard deviation of the *top* 5 members at each time step. On the x-axis we show the number of trials and on the y-axis the received reward. The stacked columns right of the learning curves show the average hyperparameters value of the top 5 members in the population at each time step, which indicates the trends of the hyperparameters across the training. Note, our algorithm discovers a bug in HalfCheetah environment which allows the agent's forward speed up to infinity and breaks the simulation. Therefore, we stop plotting the curves when 1/3 members in the population stop training due to this error. PBT outperforms Random Search and Hyperband in all four environments during the search.

horizon between 20 and 40. *CEM Alpha* and *CEM elites ratio* seem to have a clear mismatch between the schedules extracted from static and dynamic tuning methods. PBT and PBT-BT tend to be more greedy as these two values increase over time, whereas static tuning methods do not show this behavior. We also note that both PBT variants learn a decreasing learning rate schedule, similarly as reported by Jaderberg et al. (2017) without using any learning rate scheduler.

Trends on HalfCheetah On the HalfCheetah task as well, the optimization methods do not always find the same parameter schedules for the CEM hyperparameters. For example, all methods result in a mostly static value for *CEM Alpha*, but converge to different values, whereas only PBT-BT over time slowly decreases the *Plan Horizon*. Interestingly, PBT tends to reduce *CEM Population Size*, which does not match our intuition. We can observe this phenomenon also in the Hopper environment. Furthermore, we can observe that PBT is able to train agents which learn much faster. The large difference in score is because a few well performing agents are able to successfully learn the rolling gait while PBT-BT quickly prevents the "Cheetah" from falling over, a necessary precondition before learning to roll. However, the whole PBT-BT population reliably learns a fast running gait. An interesting observation when jointly optimizing all the hyperparameters is that Random Search is better than PBT as can be seen in Appendix A.4. However, we can see that PBT successfully reaches an impressive episode return with around 50 000 for the top 5 members, see Figure 4(d). Two possible reasons are (i) during optimizing with PBT, good trajectories are propagated by PBT's exploitation mechanism which copies not only the weights of the neural network but also the hyperparameters of the network and (ii) PBT searches more in the region that gives better returns through its explore mechanism. Again both PBT variants automatically adapt the learning rate.



(a) HalfCheetah Model Train

(b) HalfCheetah CEM Optimizer

Figure 5: PBT on HalfCheetah showing the average performance of the top 5 members. The green curves are obtained by not copying the history when members are replaced by the top members. The performance drops 25% and 75% in HalfCheetah for *CEM Optimizer* and *Model Training* hyperparameters respectively, indicating the need for copying the history when applying PBT to MBRL algorithms

Transferability of Hyperarameter Schedules The dynamic schedules learned on the environments are clearly not transferable across even runs while the static ones don't transfer across environments, as is shown in Figure 1. This is in stark contrast to dynamic runs usually giving the best performing results in those experiments. As discussed in Section A, environment specific configurations can allow an MBRL agent to learn a well performing dynamics model but this might not transfer well across runs or even at all to other environments. This might be attributed to the intialization of RL agents, which differs from the intialization during the HPO run. However, most search methods still outperform the hand tuned baselines for both sets of hyperparameters. Only random search in Hopper with CEM Optimizer does not show a significant gain. One explanation may be that random search does not sample good configurations with the given budget. Moreover, dynamic tuning schedules performed best in Hopper whereas in HalfCheetah, random search and Hyperband outperformed PBT and PBT-BT.

5.7 Copying History across PBT Members

As mentioned before, the original PBT paper (Jaderberg et al., 2017) does not address the importance of the history/replay buffer, which is critical in MBRL. We performed further experiments to shed light on this aspect and discovered that it is crucial to the performance (see Figure 5). When not copying the history to other members during the exploitation step of PBT, the performance has significant drops for both sets of hyperparameters. This also reveals that in the context of MBRL, the history has a significant impact on model training and implicitly influences the planning part.

Summary of Experimental Results Based on this empirical evaluation we revisit the choices (see Section 4) MBRL practicioners have to face before making use of HPO. To this end, for the PETS method we suggest to make use of dynamic tuning, if the most important evaluation criterion is final performance. If however, a robust configuration that is transferable across multiple runs is required, static configuration provides the desired result. Moreover, the history is an essential factor for MBRL during training. When applying PBT to MBRL, we suggest to also copy the history during the exploitation step.

6 Conclusion

In this paper, we thoroughly investigated the applicability of HPO to MBRL and explored different design choices MBRL practitioners have to make when tuning hyperparameters. We empirically evaluated the influence of these design choices using HPO on the PETS algorithm, and observed not only that different hyperparameters in MBRL can drastically influence the final performance, but also that HPO significantly outperforms manually tuned hyperparameter settings. Our experiments support the conclusion that automatic hyperparameter optimization can help researchers in finding settings that allow agents to learn faster and achieve better final rewards aross different tasks. In addition, we evaluated the use of dynamic hyperparameters that change through the learning process and observed that they clearly outperform static hyperparameters configurations (e.g., for *plann horizon*).

We believe that our results validate the significant impact of hyperparameter tuning on learning performance. As a result, we want to encourage the MBRL community to apply automatic configuration methods, and especially dynamic tuning methods to reduce the need for human-expertise and to take into account the non-stationarity of the RL problem. Exciting future research includes the joint and dynamic search for hyperparameters together with neural architectures, and the automatic search of alternative surrogate losses with HPO to further improve data-efficiency of MBRL.

Acknowledgements

André, Baohe, Frank and Raghu gratefully acknowledge support by the Bosch Center for Artificial Intelligence, and by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation program under grant no. 716721, as well as by the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG. They would like to thank their group for helpful feedback and discussions. Raghu additionally gratefully acknowledges support by the BMBF grant DeToL. The authors would like to thank the reviewers for their time and effort for reviewing and improving the paper.

References

Amos, B., Rodriguez, I. D. J., Sacks, J., Boots, B., and Kolter, J. Z. (2018). Differentiable MPC for end-to-end planning and control. In *Advances in Neural Information Processing Systems*, pages 8299–8310.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research (JMLR)*, 13:281–305.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554.

Chiang, H. L., Faust, A., Fiser, M., and Francis, A. (2019). Learning navigation behaviors end-to-end with AutoRL. *IEEE Robotics and Automation Letters*, 4(2):2007–2014.

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

Dasagi, V., Bruce, J., Peynot, T., and Leitner, J. (2019). Ctrl-z: Recovering from instability in reinforcement learning. *arXiv preprint arXiv:1910.03732*.

Deisenroth, M. P. and Rasmussen, C. E. (2011). PILCO: A Model-Based and Data-Efficient Approach to Policy Search. In *International Conference on Machine Learning (ICML)*, pages 465–472.

Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning (ICML)*, volume 80, pages 1436–1445.

Feurer, M. and Hutter, F. (2019). Hyperparameter optimization. In *AutoML: Methods, Sytems, Challenges*, pages 3–38.

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

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer.

Islam, R., Henderson, P., Gomrokchi, M., and Precup, D. (2017). Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. (2017). Population based training of neural networks. *arXiv preprint arXiv:1711.09846*.

Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *International Conference on Artificial Intelligence and Statistics, (AISTATS)*, pages 240–248.

Kandasamy, K., Dasarathy, G., Oliva, J. B., Schneider, J. G., and Póczos, B. (2019). Multi-fidelity gaussian process bandit optimisation. *J. Artif. Intell. Res.*, 66:151–196.

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2017). Fast bayesian optimization of machine learning hyperparameters on large datasets. *International Conference on Artificial Intelligence and Statistics, (AISTATS)*, 54:528–536.

Kurutach, T., Clavera, I., Duan, Y., Tamar, A., and Abbeel, P. (2018). Model-ensemble trust-region policy optimization. In *International Conference on Learning Representations (ICLR)*.

Lambert, N., Amos, B., Yadan, O., and Calandra, R. (2020). Objective mismatch in model-based reinforcement learning. *Learning for Dynamics and Control (L4DC)*, pages 761–770.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research (JMLR)*, 18(1):6765–6816.

Melis, G., Dyer, C., and Blunsom, P. (2018). On the state of the art of evaluation in neural language models. In *International Conference on Learning Representations (ICLR)*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D.

(2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Paul, S., Kurin, V., and Whiteson, S. (2019). Fast efficient hyperparameter tuning for policy gradient methods. In *Advances in Neural Information Processing Systems*, pages 4618–4628.

Runge, F., Stoll, D., Falkner, S., and Hutter, F. (2019). Learning to design RNA. In *International Conference on Learning Representations (ICLR)*.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. (2016). Bayesian optimization with robust bayesian neural networks. In *Advances in Neural Information Processing Systems*, pages 4134–4142.

Todorov, E., Erez, T., and Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 5026–5033.

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

Williams, G., Wagener, N., Goldfain, B., Drews, P., Rehg, J. M., Boots, B., and Theodorou, E. A. (2017). Information theoretic MPC for model-based reinforcement learning. In *International Conference on Robotics and Automation (ICRA)*.

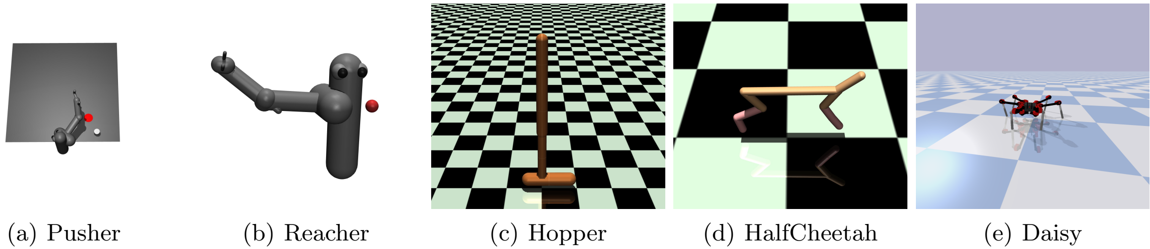
A SUPPLEMENTARY MATERIAL

A.1 Environment Overview

We summarize the dimensionality, task horizon and the number of trials for experiments on each environment in Table 1. Figure 6 shows a screenshot of each of the environments.

Name	Observation Space Dimension	Action Space Dimension	Horizon	Number of trials
Pusher	20	7	150	80
Reacher	17	7	150	80
Hopper	12	3	1000	300
HalfCheetah	18	6	1000	300
Daisy	24	18	1000	300

Table 1: Dimensions of the observation and action space, task horizon and number of trials for all environments.



(a) Pusher (b) Reacher (c) Hopper (d) HalfCheetah (e) Daisy

Figure 6: Test environments used for the experiments

Algorithm 2: Hyperband Algorithm

Input: budgets b_{min} and b_{max}, η
$s_{max} = \lfloor \log_\eta \frac{b_{max}}{b_{min}} \rfloor$;
for $s \in \{s_{max}, s_{max} - 1, \cdots, 0\}$ **do**
  $n = \lceil \frac{s_{max}+1}{s+1} \cdot \eta^s \rceil$;
  sample n configurations C;
  $b = \eta^s \cdot b_{max}$;
  while $b \leq b_{max}$ **do**
    $L = evaluate(C, b)$;
    $k = \lfloor n_i / \eta \rfloor$;
    $C = top_k(C, L)$;
    $b = \eta \cdot b$;
  end
end
Output: Configuration with the smallest loss

A.2 Implementation Details of Search Strategies

Random Search For the random search experiments, we sample 40 independent configurations for each experiment and run them in parallel. Each configuration is evaluated using the full budget (i.e. number of trials) and the total amount of trials are 3200 and 12000 for Pusher/Reacher and the other environments respectively.

Hyperband Hyperband randomly samples configurations and utilizes Successive Halving to quickly reject poorly performing configurations. Moreover, Hyperband sets the budget for each fidelity such that the amount

of time cost in each stage is approximately the same under the assumption that the time cost will be linear proportional to the amount of budgets. Compared to random search, Hyperband is slower than random search by a constant factor at most because of its dynamical trade-off between exploration and exploitation. A detailed algorithm is shown in Algorithm 2. We use the implementation from Falkner et al. (2018). The *minimal/maximal* budgets for Pusher are 8/80 and 33/300. η is equal to 3. We set the *number of iterations* to 15 in order to have a fair comparison with the other methods. This resulted in 3130 and 11982 trials for Pusher/Reacher and the other environments respectively.

PBT For PBT, we deploy exploitation and exploration every 4 and 5 trials for Pusher/Reacher and the other environments respectively. In the *exploit* step, runs with performance in the bottom 20% are replaced by runs from the top 20%. In the *explore* step, for the replaced runs, 75% perturb the hyperparameters and the remaining 25% randomly resample from the search space.

PBT-BT We reuse the hyperparameters of PBT in the *exploit* and *explore* step. To perform backtracking for PBT, we maintain a population of elites across all the timesteps elapsed so far. These are then compared to the current population members after every 30 PBT timesteps and poorly performing members are replaced with elite members sampled randomly from the elite population. When a member is replaced by an elite member, its hyperparameters are modified in such a way as to ensure no other members reuse the same hyperparameters. We also relax the restriction of standard PBT in which members are only exploited by the other members which are in the same time step. PBT-BT can also be regarded as a more greedy version of PBT, since it performs additional exploitation steps using backtracking to elite members.

A.3 Search Space

We use the same search space for all environments to ensure a fair comparison. Tables 2 &3 & 4 show the considered search spaces (for Pusher & Reacher & Hopper, HalfCheetah and Daisy respectively) where the default value follows (Chua et al., 2018) as our baseline.

Catagory	Hyperparameter	Range	Default Value	Log-Transform
Model Train	Learning Rate	[3e-5, 3e-3]	1e-3	True
	Weight Decay	[1e-7, 1e-1]	4e-4	True
	Training Epochs	[3, 20]	5	False
CEM Optimizer	Number of CEM Iterations	[3, 8]	5	False
	CEM Population Size	[100, 700]	500	True
	CEM Alpha	[0.01, 0.5]	0.1	False
	CEM Elites Ratio	[0.04, 0.5]	0.1	True
	Plan Horizon	[5, 40]	30	False

Table 2: Search space of PBT, Hyperband and random search for Pusher

Catagory	Hyperparameter	Range	Default Value	Log-Transform
Model Train	Learning Rate	[1e-5, 4e-2]	7.5e-4	True
	Weight Decay	[1e-7, 1e-1]	5e-4	True
	Training Epochs	[3, 20]	5	False
CEM Optimizer	Number of CEM Iterations	[4, 6]	5	False
	CEM Population Size	[200, 700]	400	True
	CEM Alpha	[0.05, 0.4]	0.1	False
	CEM Elites Ratio	[0.04, 0.5]	0.1	True
	Plan Horizon	[5, 40]	25	False

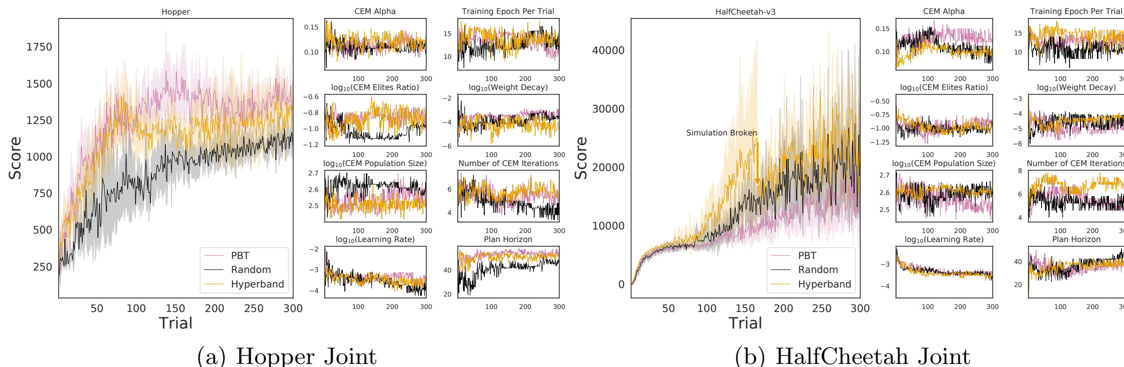
Table 3: Search space of PBT, Hyperband and random search for Reacher

Catagory	Hyperparameter	Range	Default Value	Log-Transform
Model Train	Learning Rate	[1e-5, 4e-2]	1e-3	True
	Weight Decay	[1e-7, 1e-1]	7.5e-5	True
	Training Epochs	[3, 20]	5	False
CEM Optimizer	Number of CEM Iterations	[3, 8]	5	False
	CEM Population Size	[200, 700]	500	True
	CEM Alpha	[0.05, 0.2]	0.1	False
	CEM Elites Ratio	[0.04, 0.5]	0.1	True
	Plan Horizon	[5, 60]	30	False

Table 4: Search space of PBT, Hyperband and random search for Hopper, HalfCheetah, Daisy

A.4 Joint Optimization of All Hyperparameters

In this section, we present the results obtained by jointly optimizing both sets of hyperparameters, namely *Model Training* and *CEM Optimizer*. Since the search space is larger than separately optimizing on two sets of hyperparameters. We increase the budget from 40 configurations to 80 configurations for PBT and random search. For Hyperband, we increase the number of iterations from 15 to 30. Overall, we now doubled the computational cost compared to optimizing only one set of hyperparameters at a time. The results are summarized in Figure 7. From that, we find that PBT still gives the best performance in Hopper. However, it is worse than other two static tuning methods on HalfCheetah. This may have been caused by the fact that PBT is based on evolutionary strategies which generally do not perform well on high-dimensional problems. Therefore, without enough compute budget or a good heuristic, PBT may not be able to find a promising schedule that can outperform the hyperparameter configurations found by static tuning methods. The trends of hyperparameters found during joint optimization are similar to those found by optimizing them separately, e.g. for the learning rate , plan horizon and model weight decay.



(a) Hopper Joint (b) HalfCheetah Joint

Figure 7: Joint optimization of both *Model training* and *CEM Optimizer* hyperparameters on Hopper and HalfCheetah using different search strategies. We show the average performance of the top 5 members.

A.5 Additional Experimental Results on Other Environments

In addition to Hopper and HalfCheetah, we performed additional experiments on the Pusher, Reacher and Daisy environments. As is shown in Figure 4 in the main paper, PBT-BT shows similar behavior in most cases and can get stuck in local optima. Therefore, we dropped PBT-BT to reduce the computational cost.

Pusher Since Pusher is a relatively simple task in our setting with a deterministic optimal final return given the initial state, there is only a small improvement in terms of the performance using PBT comparing to random search and Hyperband during the search, which is shown in Figures 9(a) & (b). For all hyperparameters, the trends of them found by all methods are very close to each other. Specifically, we found that all three strategies give the same increasing trend for *Plan Horizon*, which again suggests that low plan horizon is more beneficial

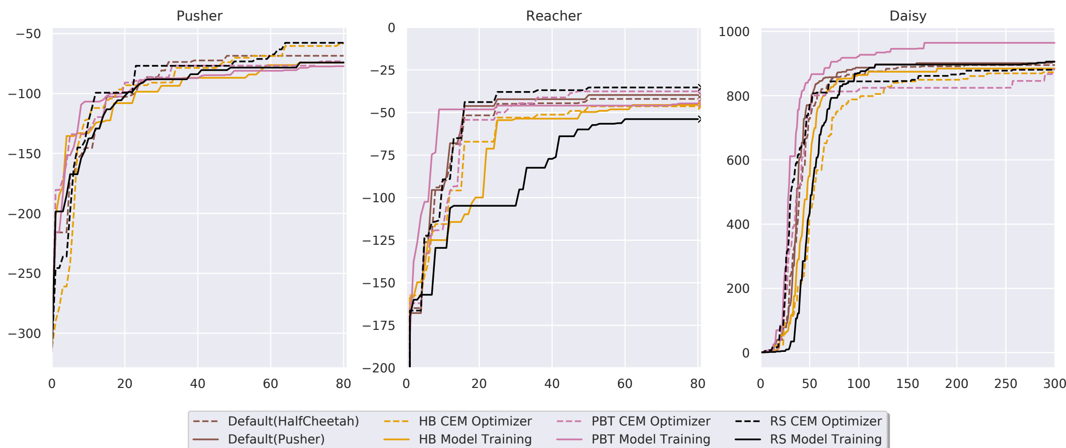


Figure 8: Evaluation of the *best* found hyperparameters and schedules from all search methods on Pusher, Reacher and Daisy. Scores (y-axis) are the maximum of the average return over 5 evaluations over number of trials (x-axis). Brown lines are the baselines using the hyperparameters tuned by human expert.

when the model is not yet fully trained. The evaluation results of all hyperparameters and schedules are shown in Figure 8. There we observe that the dynamic schedules learned by PBT have better performance in the initial training stage, which may be due to using a low plan horizon and larger learning rate in the initial stage.

Reacher Similar to Pusher, Reacher also has a deterministic final return and even lower dimensionality, which makes it easier to sovle compared to pusher. As a result, there isn't any improvement on applying dynamic tuning strategies since all three search strategies give similar results, which is shown in Figures 9(c) & (d). Similar to Pusher, the trends for all hyperparameters found by all the methods are very similar.

Daisy Daisy is the hardest task in our setting due to its relatively high observation and action dimensionalities (see section A.1). Figures 9(e) & (f) show the comparison of performances during the search and how hyperparameters evolve over time. In this environment, in contrast to Pusher, PBT learns a strategy that *decreases Plan Horizon* over time. This may suggest that for this task, long-term planning is not needed and could introduce cumulative noise in the model rollouts as it requires to perform one-step prediction with more iterations compared to short-term planning. Therefore, a short-term greedy policy can be a good choice. In terms of the *Model Training* hyperparameters, PBT learns to increase the *weight decay* over time, which indicates that it might be trying to combat overfitting which can otherwise occur for the model. This is further supported by the decreasing trend of *training epoch*. We found that the evaluation of the hyperparameters and schedules does not cause as big an impact in the final return (See Figure 8) as in the Hopper and HalfCheetah environments. However, the schedule of *Model Training* found by PBT allows the agent to converge faster and results in better rewards than the other evaluated HPO methods. The baselines perform similar to the other search methods. This could be potentially caused by the design of the search space or the low importance of the hyperparameters.

A.6 Objective Mismatch

In order to get an in-depth analysis of the relationship between model uncertainties and planning horizon in the context of MBRL and to address the Objective Mismatch problem found by (Lambert et al., 2020), we train 40 models on HalfCheetah with different plan horizons. The performance and the learned hyperparameter schedules can be found in Figure 10. We find that the performance improvement is even larger compared to optimize all *CEM Optimizer* hyperparameters. This may be because *planning horizon* is more important than the other *CEM Optimizer* hyperparameters. Moreover, models with a low plan horizon perform well in the first 10 trials but do not manage to learn the other gaits which result in higher returns.

To evaluate the obtained models, we first evaluate them on their own training data, which we refer to as on-policy evaluation. This provides more evidence on the objective mismatch problem in MBRL. Figure 11 shows the

on-policy negative log likelihood (NLL) of the top/bottom members' models on the trajectories they collected in the last 20 trials. We find that the bottom members have lower NLL compared to the top members. This is because the bottom members only explored the trajectories that gave similarly poor behaviors (in HalfCheetah, the agents kept falling on the ground) and they overfit to the data generated from such behaviors. This, again, highlights the objective mismatch problem in MBRL - lower losses when fitting the model is not the same as the objective of the planning algorithm and, thus, does not translate to better performance.

Secondly, we evaluate them on *unseen* data generated from other policies, which we refer to as off-policy evaluation. The test dataset consists of trajectories from the other policies. We split the test dataset into 3 subsets, each consisting of 300 trajectories, based on the episodic return, to further analyse the uncertainties of the models in different data regimes. The distribution of the three datasets and evaluation results on these three data regimes can be found in Figure 12. From figures 12(e) & (f), We see that the models' uncertainties increase dramatically for the poorly performing models which consistently explored only the poorly performing behaviors during the training and did not learn any meaningful strategies. For the well performing models which learned good policies, however, the NLL do not explode as for the poorly performing ones. In fact, we see much lower loss for well-performing models compared to poorly performing models when evaluated off-policy while the opposite was the case for the on-policy evaluation. This supports the conclusion that poorly performing models are overfitting to their training data while well-performing models are able to generalise to unseen data. Both, top models and bottom models, have similar NLL in the low return data regime. However, models which have a larger plan horizon have lower returns initially (See Figure 10), which supports our hypothesis that in order to perform long-term planning, models first need to be accurate enough. In words, when models have high uncertainties in an environment, the plan horizon should be relatively low to reduce the uncertainties introduced in the planning algorithms.

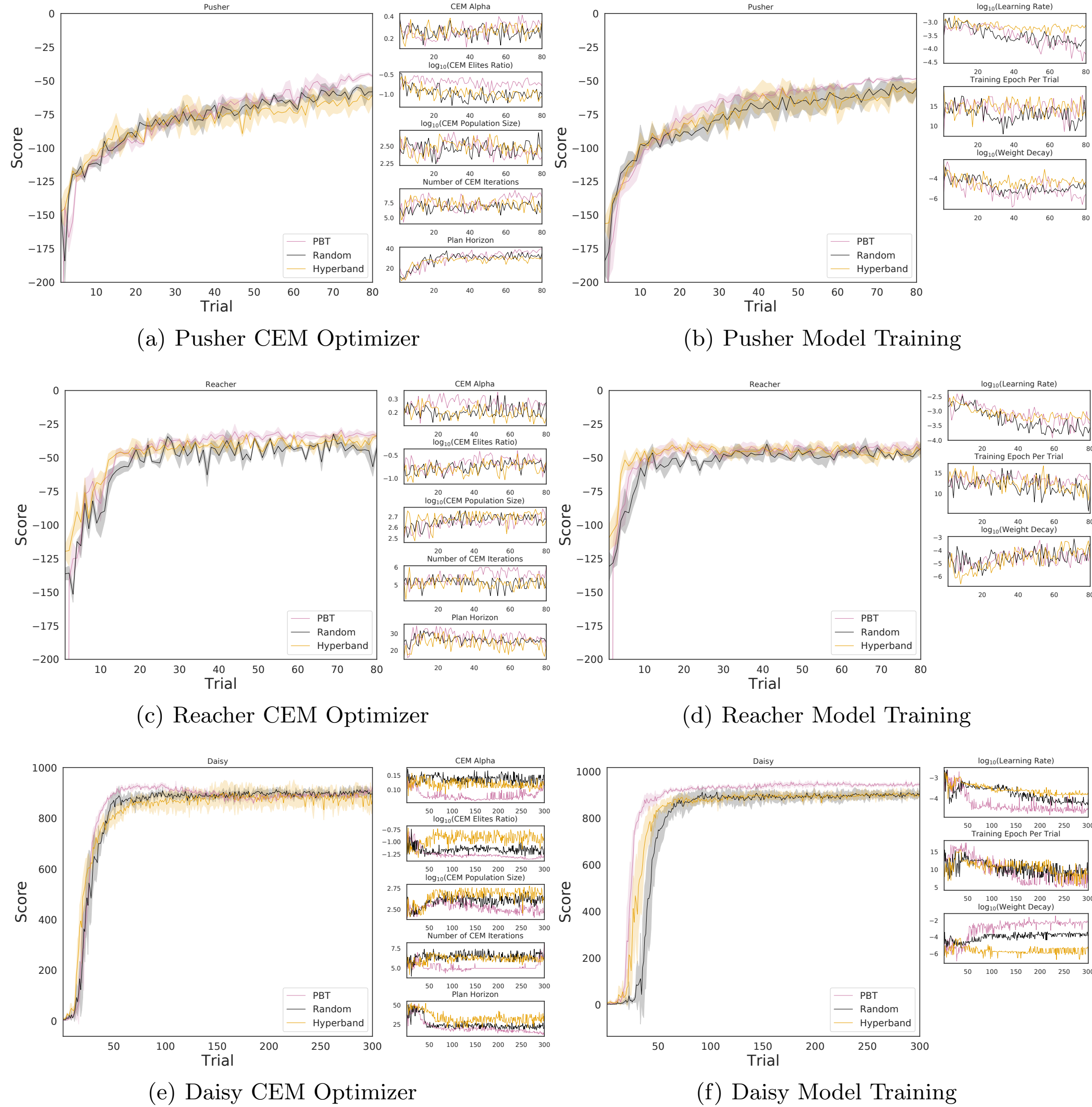
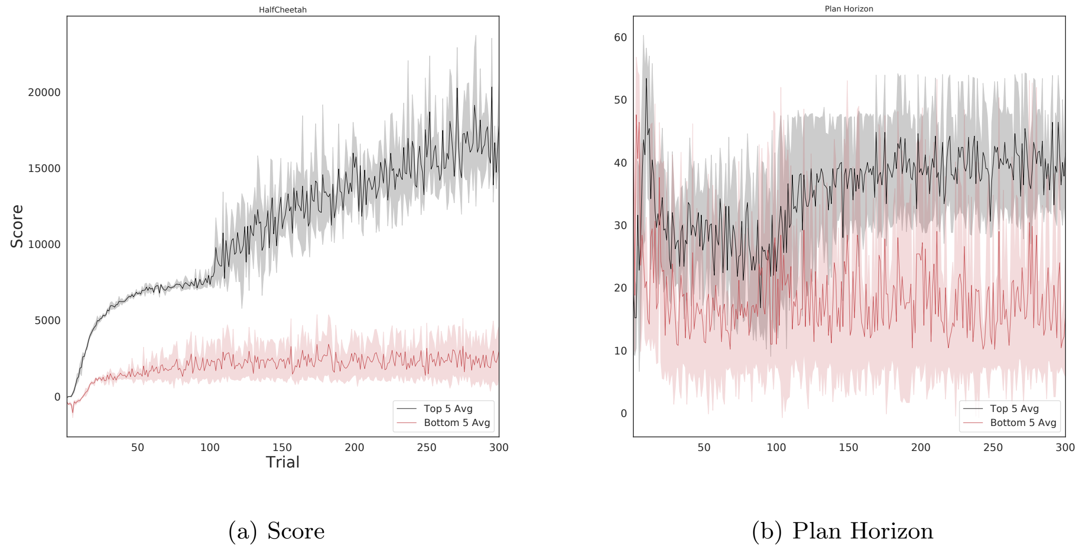


Figure 9: Training curves from PBT, Hyperband and Random Search on Pusher (top), and Reacher (middle) and Daisy (Bottom). We report the mean and standard deviation of the *top* 5 members at each time step. On the x-axis we show the number of trials and on the y-axis the received reward. The stacked columns on the right of the learning curves show the average hyperparameter values of the top members in the population at each time step, which indicates the trend of the hyperparameters across training.



(a) Score

(b) Plan Horizon

Figure 10: Results of using random search to optimize only the plan horizon. 40 models were trained, each with a plan horizon sampled from the configuration space. We plot the average performance and plan horizon of the top 5 members and the bottom 5 members over the training.

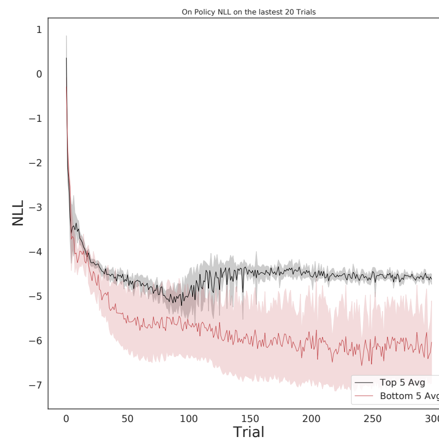
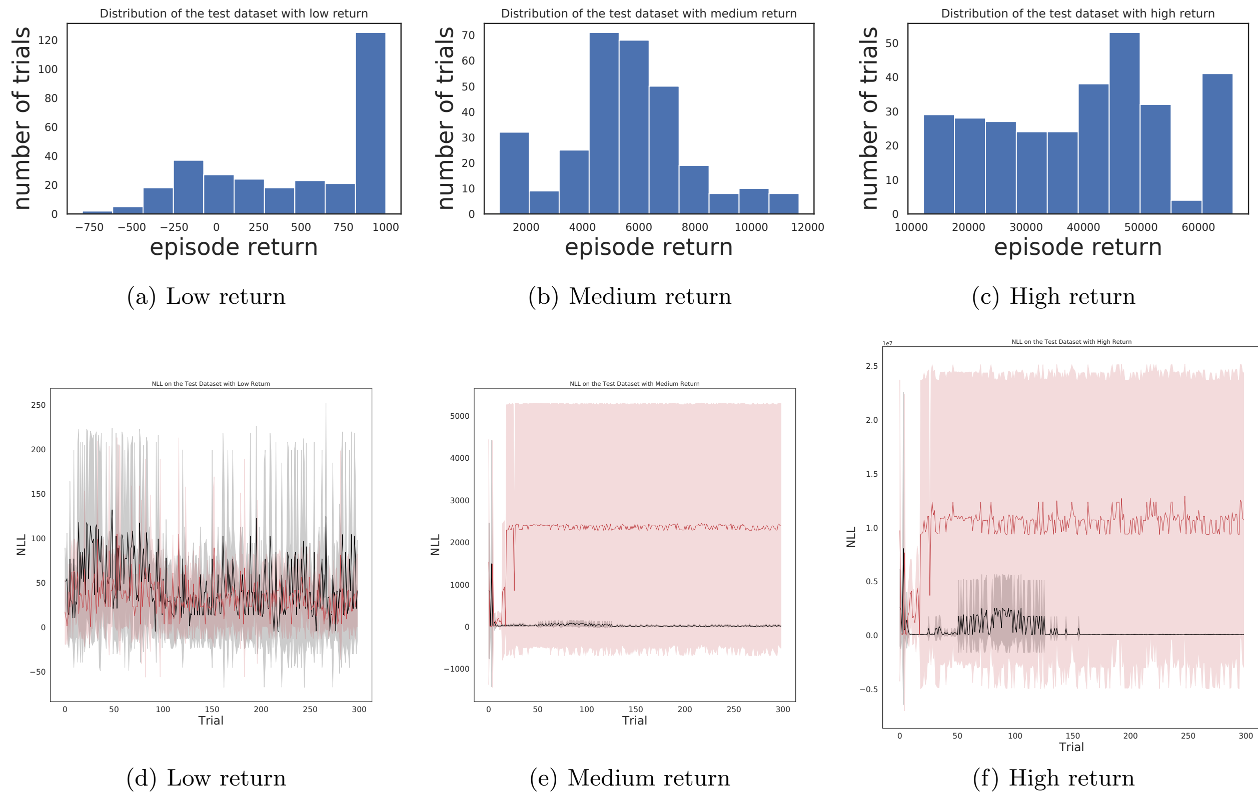


Figure 11: NLL of top and bottom members for the objective mismatch experiment. On the *y-axis*, we show the on-policy NLL of each model by evaluating the model on the last 20 trials collected (if not enough trials are available, then we use all trials collected until that time point) and the number of trials on the *x-axis*.



(a) Low return (b) Medium return (c) High return

(d) Low return (e) Medium return (f) High return

Figure 12: (Upper row) Reward distribution of three "out-of-distribution" test datasets. Data is generated by different policies and therefore not seen by the models during their training. Each dataset has 300 trajectories. (Lower row) The average NLL of the top/bottom members on the test dataset during training.