REINFORCEMENT LEARNING IN SYSTEM IDENTIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

System identification, also known as learning forward models, transfer functions, system dynamics, etc., has a long tradition both in science and engineering in different fields. Particularly, it is a recurring theme in Reinforcement Learning research, where forward models approximate the state transition function of a Markov Decision Process by learning a mapping function from current state and action to the next state. This problem is commonly defined as a Supervised Learning problem in a direct way. This common approach faces several difficulties due to the inherent complexities of the dynamics to learn, for example, delayed effects, high non-linearity, non-stationarity, partial observability and, more important, error accumulation when using bootstrapped predictions (predictions based on past predictions), over large time horizons. Here we explore the use of Reinforcement Learning in this problem. We elaborate on why and how this problem fits naturally and sound as a Reinforcement Learning problem, and present some experimental results that demonstrate RL is a promising technique to solve these kind of problems.

1 INTRODUCTION

Learning forward models has been an active area of research in past decades, with abundant contributions on the application of Machine Learning (ML) techniques to the "system identification problem" (see for instance, Werbos, 1989; Fu & Li, 2013; Zhang, 2014; Abdufattokhov & Muhiddinov, 2019; Roehrl et al., 2020).

Particularly, it is a recurring topic of research within Reinforcement Learning (RL, see Sutton, 1991; Sutton & Barto, 1998; Polydoros & Nalpantidis, 2017; Moerland et al., 2020), where forward models usually represent the transition function $s_{t+1} = \mathcal{T}(s_t, a_t)$ of some Markov Decision Process (MDP). We denote an MDP as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $\mathcal{S}$ denotes the state space, $\mathcal{A}$ denotes the action space, $\mathcal{T}$ denotes the transition function and $\mathcal{R}$ denotes the reward function. Thus, $s_{t+1} = \mathcal{T}(s_t, a_t)$ represents the immediate state after the evolution of the system, starting at time t with state s_t and conditioned by an action a_t. Hence, $\mathcal{T}$ can be represented by a deterministic or stochastic mapping function $s_{t+1} = \mathcal{F}(s_t, a_t)$.

Learning a forward model, i.e., learning a mapping function $\hat{s}_{t+1} = f(s_t, a_t)$, is commonly defined as a Supervised Learning problem in a direct way (Jordan & Rumelhart, 1992; Moerland et al., 2020), since there are well-defined observations $\mathbf{X} = \{(s_t, a_t), ...\}$, labels $\mathbf{y} = \{s_{t+1}, ...\}$, and a loss, e.g., $\mathcal{L} = ||f(s_t, a_t) - s_{t+1}||_2$, which are the core parts of such problems. However, we must note that what we are *given* is a set of observations solely, and it is just a reasonable *assumption* defining the labels as $\mathbf{y} = \{s_{t+1}, ...\}$. In practice, this approach faces several difficulties due to the inherent complexities of the dynamics to learn, thus we propose that the problem of learning forward models can be more naturally defined, and indeed effectively solved, as an RL problem.

2 MOTIVATION AND PROBLEM DEFINITION

Why learning forward models with RL? The domains, tasks, and problems to which forward modeling is being applied are of increasing complexity, including time delayed dynamical effects, high degree of non-linearity, partial observability (POMDPs), and in general, complex dynamics.

This situation raised the need of additional techniques to adapt the Supervised Learning framework to deal with this increasing complexity, for instance:

1. Rollout testing for beyond single step learning robustness.
2. Loss accumulation over rollouts for large horizon prediction.
3. Recurrent networks, frame-stacking, or neural Turing Machines for partial observability.
4. Curriculum learning over increasing horizons to aid learning convergence.
5. Data augmentation to aid learning symmetries in data.
6. Ensembles of stochastic neural networks to increase the prediction accuracy and reduce bias.

These additional techniques cause an "over complexification" of the Supervised Learning setting. On the other hand, RL has intrinsic features that provides a natural way to deal with many of those complexities, such as:

1. Rollout learning by working with episodic tasks.
2. Minimization of error accumulation by optimizing in the long-run.
3. Stochastic scenarios.
4. Partial observability.
5. Solving the sequential credit assignment problem.
6. Continuous learning from new experience without requiring a full retraining.

Unreasonable extra search cost? Solving a regression problem by RL has an extra cost due to the required exploration, i.e. searching for a point $y_i \in \mathbf{y}$ (label) that is indeed already known. Thus, why solving this problem with RL? Is it an unreasonable cost? May we have now two problems instead of one?

Let's suppose that we run bootstrapped rollouts of certain length (time horizon), while minimizing the errors between each predicted point $x_t = f(s_t, a_t)$ and its corresponding "true target" (label) $y_t \in \mathbf{y}$. Since the predictions in the rollouts trajectories are *sequentially dependent* (by bootstrapping), then we face a temporal credit assignment problem. The RL framework deals with this issue naturally through the Bellman's optimality criteria (Bellman 1957). Also, we shall emphasize that we are not given "true targets", instead we are *assuming* that the next observed states are the targets. Moreover, since we are relying on bootstrapping, then, in the Supervised Learning setting, we need the corresponding observation points x_t for the predicted labels $\hat{y_t} = f(x_t)$, however, these are not in the dataset, since x_t are predictions as well.

How to learn forward models with RL? Learning forward models with RL can be achieved directly just by translating a regression problem to an RL problem, where the observation (state) is formed by the current state of the system and the previous observed action, the actions of the agent represent the predictions of the next state, and the reward signal is just the negated total/cumulative prediction error. Note that now we don't have to assume that next observations are targets.

More formally, given an MDP, $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, the forward learning problem is defined as the MDP $\mathcal{M}_{\mathcal{F}} = (\mathcal{S}_{\mathcal{F}}, \mathcal{A}_{\mathcal{F}}, \mathcal{D}|\mathcal{O}, \mathcal{L}_{\mathcal{F}})$, where $\mathcal{D}|\mathcal{O}$ refers to a time series of transitions stored in a dataset ($\mathcal{D}$) or observed ($\mathcal{O}$) from a real world process, $\mathcal{L}_{\mathcal{F}} = ||s_{t+1} - \hat{s}_{t+1}||$ is a loss function of the true observed next state ($s_{t+1} \leftarrow \mathcal{D}|\mathcal{O}$) and the predicted next state ($\hat{s}_{t+1}$), $\mathcal{S}_{\mathcal{F}}$ denotes the state space formed by tuples of $(s, a) \in \mathcal{S} \times \mathcal{A}$, and $\mathcal{A}_{\mathcal{F}}$ is the action space defined by elements $\Delta s = \hat{s}_{t+1} - \hat{s}_t$, where $s \in S$. Figure 1 shows the diagrams for the MDPs $\mathcal{M}$ and $\mathcal{M}_{\mathcal{F}}$, respectively.

Hence, to apply RL to the forward model learning problem, we just need a "transposition" of the problem and its observations, where the final objective is to obtain, by RL, a policy $\pi_{\hat{\mathcal{M}}}$ on the approximated dynamics $\pi_{\mathcal{M}_{\mathcal{F}}} \leftarrow \mathcal{M}_{\mathcal{F}}$, such that $\pi_{\hat{\mathcal{M}}}$ is as close as possible to the optimal policy

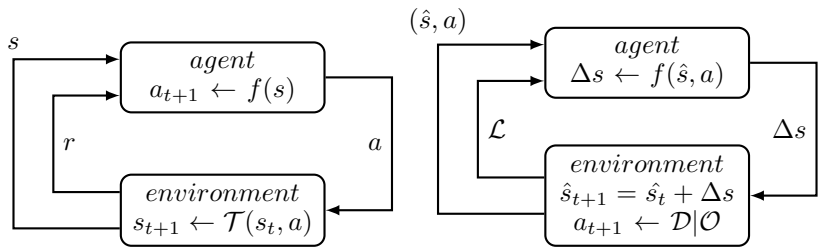

Figure 1: (Left) the typical RL setting. (Right) RL flow for learning a forward model, where $\mathcal{D}|\mathcal{O}$ refers to a time series of transitions stored in a dataset ($\mathcal{D}$) or observed ($\mathcal{O}$) from a real world process, and $\mathcal{L} = ||s_{t+1} - \hat{s}_{t+1}||$ is a loss function of the true observed next state ($s_{t+1} \leftarrow \mathcal{D}|\mathcal{O}$) and the predicted next state ($\hat{s}_{t+1}$), in the form of a reward signal to be maximized.

$\pi_{\mathcal{M}}$ in the original MDP, as shown in Equations 1 to 4.

$$\pi_{\mathcal{M}} \leftarrow \mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}) \tag{1}$$

$$\pi_{\mathcal{M}_{\mathcal{F}}} \leftarrow \mathcal{M}_{\mathcal{F}} = (\mathcal{S}_{\mathcal{F}}, \mathcal{A}_{\mathcal{F}}, \mathcal{D}|\mathcal{O}, \mathcal{L}_{\mathcal{F}}) \tag{2}$$

$$\pi_{\hat{\mathcal{M}}} \leftarrow \hat{\mathcal{M}} = (\mathcal{S}, \mathcal{A}, \pi_{\mathcal{M}_{\mathcal{F}}}, \mathcal{R}) \tag{3}$$

$$||\pi_{\mathcal{M}} - \pi_{\hat{\mathcal{M}}}|| \propto ||\mathcal{T} - \pi_{\mathcal{M}_{\mathcal{F}}}|| \tag{4}$$

3 Derivation and method

Starting from the one-step definition of a forward model (Eq. 5):

$$\hat{s}_{t+1} = f(s_t, a_t), \tag{5}$$

where s_t is the *state* of the system at time step t and a_t being the control actions applied to the system at t, we obtain a one-step (*forward*) prediction of the next *state* $\hat{s}_{t+1}$ of the system. Learning such a forward model $f(s, a)$ implies learning the approximator function $f \approx \mathcal{T}$ with any statistical or ML method (e.g., by fitting the parameters θ of a parametric estimator f_θ). For this, a loss function is defined as $\mathcal{L}$ (Eq. 6),

$$\mathcal{L} = ||s_{t+1} - \hat{s}_{t+1}||, \tag{6}$$

where $\mathcal{L}$ is minimized during training, for instance, using gradient descent techniques by computing the gradient $\nabla_\theta \mathcal{L}$. Thus, the problem setting naturally appears as a Supervised Learning problem (regression) $\hat{y}_i = f(x_i)$, where the collection of inputs $\mathbf{X}$ is formed by tuples $x_i = (s_t, a_t)$ and the targets collection (labels) $\mathbf{y}$ are commonly defined by the next observed states $y_i = s_{t+1}$.

With this definition, a Supervised Learning task is completely defined and can be then easily achieved through a "vanilla" ML pipeline. However, as we noted in Section 2, the results can suffer from several issues, requiring thus, a battery of adhoc methods and other "over complexifications".

For instance, Fleming (2018) gives an extensive list of common tips and additions to consider. In this work, we consider and apply the following ones:

1. Predict the deltas from the current state, not the next state directly.
2. Use frame-stacking-like window of past observations.
3. Use data-augmentation techniques.
4. Use special regularization techniques such as noise or Dropout.
5. Evaluate the model accuracy using rollout evaluations.
6. More importantly, use bootstrapped rollout training (predictions over past predictions).

From the immediate above list as number 1, by predicting deltas, Eq. 5 becomes of the following form (c.f. Eq. 7):

$$\hat{s}_{t+1} = s_t + f(s_t, a_t), \text{ and hence,} \tag{7}$$

$$\frac{\Delta \hat{s}}{dt} = f(s, a), \tag{8}$$

which expresses a well-known recursive relation in the field of Dynamical Systems, by expressing the evolution of the system in terms of its past state plus its derivative (residual).

The last point, number 6 on the above list, appears as the most difficult part from the learning perspective. Moreover, we claim it is a requirement for a sound and robust definition of a forward model learning problem. Additional features are also inherent to a rollout training when applying RL, for instance, it creates, trough exploration, an implicit data-augmentation over the source "ground-truth" data ($\mathcal{D}|\mathcal{O}$), which is a common practice done as an extra step in the supervised ML setting to improve generalization and reduce over-fitting (van Dyk & Meng, 2001; Hernández-García & König, 2018; Iwana & Uchida, 2021). This implicit data augmentation, instead of enlarging a training dataset from existing data using various translations, acts as a kind of "spatio-temporal" data augmentation, as in the case of autoencoders, that generate new training data as shown by Tu et al. (2018), contributing to obtain robust policies to noisy inputs, aid in learning problem symmetries and generalization, and helping to avoid overfitting to the fixed dataset or observations.

Function approximation: Gaussian networks Focusing on the most recent works on model-based RL (Janner et al. 2019; Yu et al. 2020), we selected a stochastic Gaussian network as the function approximator to learn the policy $f \approx \mathcal{T}$ such that:

$$f = \frac{\Delta \hat{s}}{dt} \sim \mathcal{N}\left(\mu_{s,a},\, \sigma^2_{s,a}\right), \tag{9}$$

this allows us to approximate the dynamics through a stochastic model. There is, perhaps, good margin for improvements over Gaussian networks as pointed out by Chou et al. (2017). However, the original SAC algorithm, which we use for our experiments, uses a squashed Gaussian Normal Policy for the actor network (see, Haarnoja et al. 2018).

Rollout loss and evaluation An effective forward model should predict not only the next state, based on the true past state and true action accurately, but should allow as well to simulate the system by running rollouts over its own predictions. The rollout process then implies a bootstrapping process, that is, predicting the next state based on a previous predicted state. Thus, by running a rollout $\mathrm{E}(s_t, (\mathrm{a})_t^h, h)$ of length h simulation steps over a fixed sequence of actions $(\mathrm{a})_t^h = (a_t ... a_{t+h})$ from an initial state s_t, we obtain:

$$\hat{s}_{t+h} = s_t + \sum_{i=0}^{h} f\left(\hat{s}_{t+i}, a_{t+i}\right)\Big|_{t=0, \hat{s}_0 = s_0}, \tag{10}$$

obtaining a bootstrapped prediction $\hat{s}_{t+h}$ of length h and a rollout trajectory $\hat{y} = (s_t, \hat{s}_{t+1}, ..., \hat{s}_{t+h})$. Also, equations 7 to 10 define well known recurrent relations that can be seen as the equivalent of recurrent connections, computed with a loop over a sequence input batch where loss is calculated at the end, as well as dynamical system modeling through Neural ODEs (ODENet, Chen et al. 2018), its augmented version (Teh et al. 2019), and ResNets (He et al. 2016).

Thus, the rollout loss $\mathcal{L}_E(s_t, (\mathrm{a})_t^h, y, \hat{y}, h)$ can then be defined as follows:

$$\begin{aligned} \mathcal{L}_E(s_t, (\mathrm{a})_t^h, y, \hat{y}, h) &= \sum_{i=1}^{h} ||s_{t+i} - \hat{s}_{t+i}||, \text{ with } s_{t+i} \in y, \hat{s}_{t+i} \in \hat{y},\ or, \\ &= \mathcal{Z}_E(y, \hat{y}), \end{aligned} \tag{11}$$

being $\mathcal{Z}_E(y, \hat{y})$ a signal (trajectory) similarity function. Thus, in rollout learning, the network is trained with the rollout loss (Eq. 11), instead of the common supervised loss (c.f. Eq. 6).

The signal similarity function $\mathcal{Z}_E(y, \hat{y})$ is an open choice, however, it changes the problem to be solved, as it is the goal of the RL problem. In particular, here the problem is sequential and thus the objective is to evaluate how "close" in terms of shape (pattern recognition) is the trajectory generated vs. the true trajectory (both y and $\hat{y}$ can be interpreted as two signals). We must note that (r)mse for this problem is just a weak approximation and that doing signal/pattern recognition using (r)mse alone to measure signal/shape similarity is definitely not a good general measure, see for instance Pandit & Schuller (2019). Also, there are many measures for signal similarity (e.g., KL-divergence, statistical (invariant) moments, signal correlations, etc.), and usually a combination works better depending on the type of the signals and which signal's features are more important for the problem.

Policy learning with the Actor-Critic architecture Finally, a natural way to implement rollout learning is episodic learning, where sequences of transitions are divided in episodes, and rollouts are run over such episodes using loss accumulation for computing the gradient. This approach naturally conducts to thinking on the temporal credit assignment problem (see, Minsky 1961; Sutton 1984; Sutton & Barto 1998), which is solved effectively through TD-Learning (Sutton 1988). That is, to solve a sequential decision problem optimizing a scalar signal over an episodic task (a rollout). For this it looks natural to use the Actor-Critic method (Sutton & Barto, 1998; Degris et al., 2012) to train a policy network and, in particular, a stochastic one to predict the deltas (residuals) of the next system state. This can be achieved trough methods like the SAC algorithm (Haarnoja et al., 2018).

4 EXPERIMENTAL EVALUATION

We use the Soft Actor-Critic (SAC) algorithm (Haarnoja et al., 2018) as the base of our trainable agents, where the actor networks represent the learned models. Refer to Appendix A.2 for additional details about hardware and software used. We test the proposed approach on three different MuJoCo environments from OpenAI Gym control suite: Hopper-v2, Walker2d-v2, and Halfcheetah-v2. A specific Gym environment $\mathcal{M}_{\mathcal{F}}$ has been developed, which basically returns rollouts predictions as steps plus a reward value based on the trajectory error.

In order to define the state space $\mathcal{S}_{\mathcal{F}}$, a window w (stacking) of 20 past time steps observations is used. Every rollout prediction relies on this stack of present and past positions plus the taken actions stored in the datasets. At each step, every new prediction is incorporated into w in a FIFO way. Also, for giving more variability in the training data, each episodes is started randomly from a time step in the range $[0, 30]$, so that the agent is not seeing the same sequences all the time.

All MuJoCo environments used here rely on continuous state and action spaces. The state variables can be divided into two main categories: positions ($qpos$) and velocities ($qvel$). Hence, position can be derived from velocity and vice-versa. Thus, we opted to predict only the deltas of the positions $\Delta qpos$, and then infer the original velocities of the simulation. The predicted $qpos$ and its corresponding $qvel$ are calculated according to:

$$qpos_{t+1} = qpos_t + \Delta qpos \tag{12}$$

$$qvel_{t+1} = \frac{\Delta qpos}{dt}, \tag{13}$$

D4RL (Fu et al., 2020) datasets of the MuJoCo environments have been used to train the forward models. Further details on the data preparation can be found in Appendix A.4.

All the experiments presented share the same hyperparameters: Actor and Critic encoders architectures are based on 6 hidden layers of 512 elements each. Mish (Misra, 2019) has been used as activation function. Quantile regression (Dabney et al., 2017) is used for the Q-function networks. Batch-size for each network and learning step is $B = 5 \times 2^{10}$ data-points. Other information about specific training parameters like the number of episodes run per experiment, or gradient steps can be found at Table 1.

Table 1: Experiment's training parameters and main test metrics results.

Environment	Training Parameters				Testing - last values (average)		
	Episodes	#Params	dt	Grad. Steps	Cr. loss	Act. loss	rmse
Hopper-v2	$12K$	$1425K$	0.008	$\sim 121K$	0.49	2.05	0.09
Walker2d-v2	$12K$	$1488K$	0.008	$\sim 122K$	5.76	10.63	0.19
Halfcheetah-v2	$12K$	$1488K$	0.05	$\sim 121K$	69.06	33.00	0.28

For the general experiments, we use a pseudo-sparse reward signal as follows:

$$r_t(s,a) = \begin{cases} 1 - \mathcal{Z}_E(y, \hat{y})(\text{signal similarity function}) & \text{at rollout ends,} \\ -\|s_t - \hat{s}_t\|_{L_2} & \text{otherwise,} \end{cases} \tag{14}$$

being the signal similarity function $\mathcal{Z}_E(y, \hat{y})$:

$$\mathcal{Z}_E(y, \hat{y}) = (1 + \|y - \hat{y}\|_{L_2})\,(1 + \|\nabla y - \nabla \hat{y}\|_{corr})\,(1 + \|y - \hat{y}\|_{KL})\,, \tag{15}$$

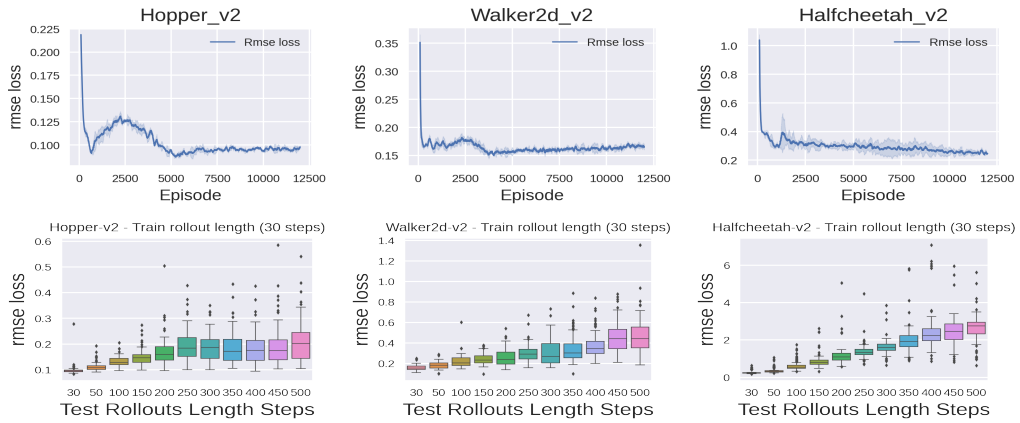

Figure 2: Hopper, Walker2D and HalfCheetah: RSME per episode and rollout test performance

where, $\|y - \hat{y}\|_{L_2}$ is the sum of the squared error between true and the predicted values, $\|\nabla y - \nabla \hat{y}\|_{corr}$ is the correlation distance between the derivatives of both predicted and true values and $\|y - \hat{y}\|_{KL}$ is the Kullback-Leibler divergence between both true and predicted values, over all the the current rollout steps.

For Hopper, Figure 2 shows the overall RMSE of predicted variables and the total reward for the forward environment. After running a total of 12K episodes and $\sim$121k gradient steps, the overall RMSE converged close to 0.10 out-of-sample. Additionally, both actor and critic loss curves show that the SAC agent is learning a good policy from episodes 1.000 to 4.000 (see 2 for additional experimental results). After these episodes, both curves converge asymptotically, suggesting a pseudo optimal policy has been achieved. Walker2d has similar number of gradient steps and episodes as Hopper, it is noticeable how errors are higher in terms of RMSE out-of-sample (Fig. 2) as well as actor and critic losses (see Appendix 2) than Hopper. This is basically due to the higher number of predicted variables (almost double), and the higher complexity of the environment. We can also notice this increase when the model is tested with different rollout-steps. However, the asymptotic behavior of these curves is still maintained, suggesting the finding of a stable sub-optimal policy. Finally, HalfCheetah is the hardest environment tested. Despite having the same number of target variables than Walker2d, these are noisier as the dynamic of the robot is more complex and the results seems no to be as good as expected.

4.1 Comparison of Reinforcement Learning vs. "vanilla" Supervised Learning approaches

To compare both Supervised Learning and Reinforcement Learning approaches we selected the Hopper environment and designed a dedicated setup (as fair as we were able to do).

Reinforcement Learning setup: Two SAC based RL forward models have been trained (RL-v1 and RL-v2) with different frame stack windows. More precisely, RL-v1 uses a framestack of $w = 10$, while RL-V2 uses a framestack of $w = 30$. The reward function used for this experiments is a full-sparse reward, that is, $r = 0$ for every time step, and $r =$ signal similarity measure (between both true and predicted trajectories) at every rollout end. This has been designed in this way to show RL in it's extreme version, despite this sparse reward function will hurt convergence speed.

The signal similarity measure (RL goal) used in this experiments is a simplified but effective similarity measure. It incorporates explicit measures from time-domain similarity, frequency-domain similarity as well as power-domain similarity.

Supervised Learning setup: Two Supervised Learning models (SL-v1 and SL-v2) have been trained and tested with different framestack windows as well. SL-v1 has a framestack window of $w = 10$, while SL-v2 a window of $w = 30$, same as RL setup. The Supervised Learning model is a "vanilla" training approach following the same structure of steps/rounds of the RL training setup.

In particular, we use the same SAC Actor Network for the Supervised Learning policy. However, we sample from the true dataset and not from a replay buffer as RL does. Each sample to train is of batch-size $= 5 \times 2^{10}$ and, at every step/round, 100 mini-batches are randomly sampled and shuffled to train the network using the Pytorch MSE Loss.

Thus, both approaches use the same Policy Networks but trained using two different methods. Special attention have to be paid to the fact that these two approaches solves different problems, since they have different optimization objectives: the RL training uses as its goal a signal similarity function over entire rollouts while the "vanilla" SL training optimizes the MSE over random mini-batches.

4.1.1 TRAINING METRICS

A total of 25.000 steps/rounds have been performed per experiment. Figure 3 shows four different plots:

- ⌜ Critic Loss (upper-left) of the RL approach per evaluation-round. The Critic Loss for SL is 0 since no critic is trained for the SL experiments.
- ⌝ Supervised loss MSE (upper-right) per evaluation-round for all policies (RL+SL) since it is possible to evaluate an RL-actor policy network over the true dataset in the same way as the SL policy.
- ⌞ RMSE rollout-metric (lower-left) which is the RMSE evaluated at the current evaluation-round over all the steps of a random episode e_r from the dataset, comparing the predicted trajectory of the entire episode vs. the true trajectory of the entire episode.
- ⌟ Mean rollout reward (lower-right) shows the mean of the rewards obtained by the evaluation of the signal similarity measure at each rollout end of the random episode e_r.

Some conclusions can be elaborated from observing the results in Figure 3.

1. SL trained policies converge very fast to very low values of the Supervised loss MSE and low values of the RMSE rollout-metric (SL-v1 and SL-v2 lines almost converge in the firsts training rounds). However, it is not the same for the Mean rollout reward (signal similarity measure).
2. The SL-v2 version ($w = 30$) achieves better optimality than SL-v1 ($w = 10$).
3. The RL trained policies convergence speed is notably slower (orders of magnitude), however both versions of RL policies achieves better optimality than SL in Rmse rollout-metric and Mean rollout reward.
4. Although RL-v2 policy of ($w = 30$) shows a better optimality in the RMSE rollout-metric than the RL-v1 ($w = 20$), it is not clear that this is true for Mean rollout reward metric.
5. It is somehow natural that the trained SL policies does not perform well for the Mean rollout reward metric, since they are not trained with this learning objective. However, both RL policies are also converging to low enough values of the Supervised loss MSE metric, even when they are not trained with this objective function.
6. Finally, from the whole figure, it can be argued that beyond some "epochs" RL starts to gain advantage over the vanilla SL approach, not only achieving better values in the RMSE rollout-metric and Mean rollout reward metrics, but also with significantly less standard deviations, as shown in the shaded regions enclosing all curves.

4.1.2 ROLLOUT SIZE EVALUATION

With the corresponding trained policies we tested each one by predicting trajectories on random episodes for different rollout lengths $h \in \{50, 100, 200, 300, \ldots, 1000\}$ (the rollout length during training RL was 50 episode steps), as shown in Figure 3. According to the results, it is noticeable how the SL policies (SL-v1 and SL-v2) perform better than RL for shorter rollouts ($\approx< 300$). However, as we increase the length of the rollouts, the SL trained policies suffer an exponentially increasing compounding error. Nevertheless, the RL approach seems to be more robust against rollout lengths, suggesting, perhaps, that RL polices are able to better grasp the underlying dynamics,

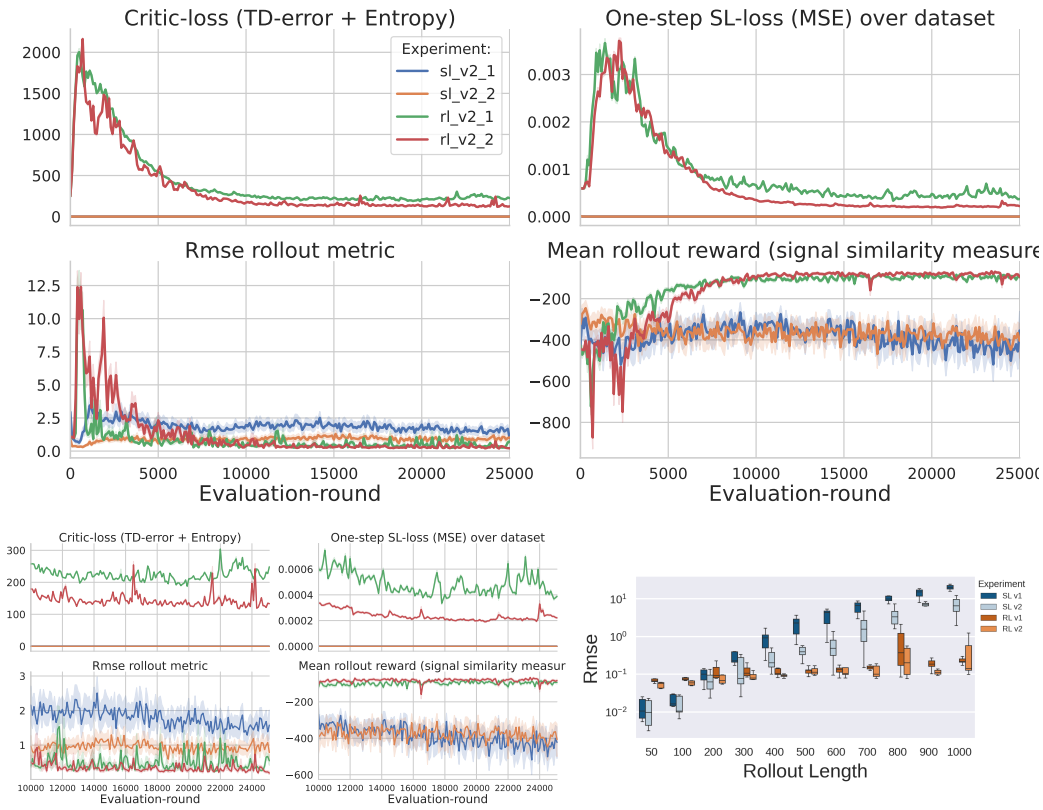

Figure 3: SL and RL Training metrics (upper) and zoom-in last episodes (lower-left). Rollout tests (lower-right) performed with different lengths (logarithmic scale).

which is crucial for predicting full trajectories, especially those that are highly multimodal. Figures 4 and 5 show examples of Hopper's random episodes with the true trajectories vs. predicted ones, for rollouts of 50 and 500 steps for both for SL-v2 and RL-v2.

5 Conclusion and Future Work

This paper presents a general framework to learn forward models with Reinforcement Learning. We have shown significant reasons to consider forward-model learning as an RL problem instead of the common definition as a Supervised Learning task. We have also defined and described how to model this problem with RL, and finally, this ideas were tested over three well-known environments from the MuJoCo Gym collection. In addition, we have compared our approach against a commonly-used Supervised Learning framework, with significant results for large horizon simulations.

Other contributions to consider are that forward-model learning can be used as new kind of benchmark for RL algorithms, besides the control paradigm, enforcing the requirements for fast but deep exploration and convergence optimization. Using RL for forward-model learning also brings homogeneity to model-based RL.

Future work includes the use of these forward models to train RL control agents. This will allow exploration in offline RL, which is a very desirable feature to get optimal and robust enough policies, and thus compare more fairly their performance against agents trained in the real environment. However, to do so, it is still required to perform additional experiments to validate the performance of the learned forward models against real dynamics, as well as understanding the limitations of the learned models.

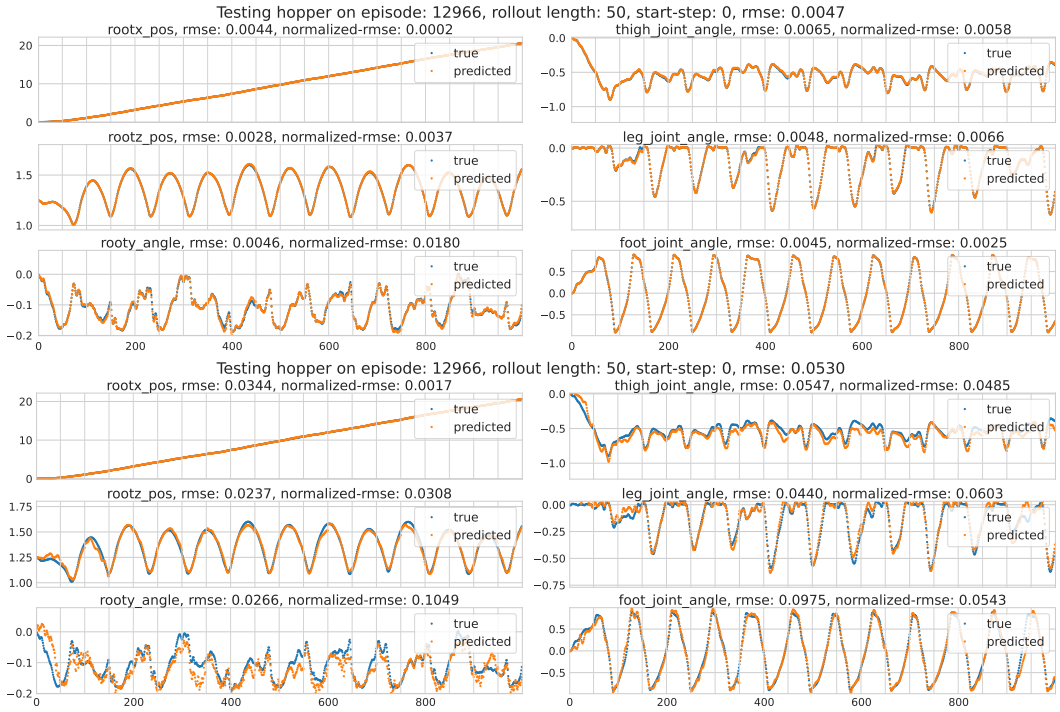

Figure 4: Random episodes of rollouts length 50 steps for the Supervised Learning (upper) and Reinforcement Learning (lower) policies.

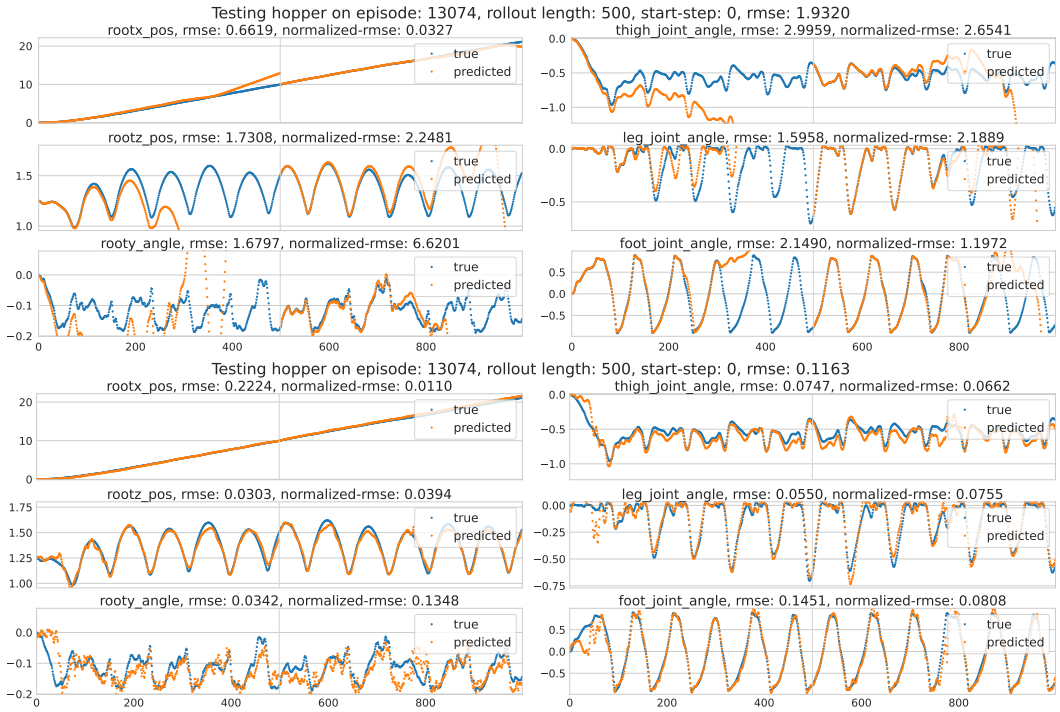

Figure 5: Random episodes of rollouts length 500 steps for the Supervised Learning (upper) and Reinforcement Learning (lower) policies.

REFERENCES

Shokhjakhon Abdufattokhov and Behzod Muhiddinov. Stochastic approach for system identification using machine learning. In *2019 Dynamics of Systems, Mechanisms and Machines (Dynamics)*, pp. 1–4, 2019. doi: 10.1109/Dynamics47113.2019.8944452.

R.E. Bellman. *Dynamic Programming*. Dover Books on Computer Science Series. Dover Publications, 1957. ISBN 9780486428093. URL https://books.google.es/books?id=fyVtp3EMxasC.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

Po-Wei Chou, Daniel Maturana, and Sebastian Scherer. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 834–843. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/chou17a.html.

Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression, 2017. URL https://arxiv.org/abs/1710.10044.

Thomas Degris, Martha White, and Richard Sutton. Off-policy actor-critic. In *International Conference on Machine Learning*, 2012.

Jim (Fomoro) Fleming. Forward models tutorial, 2018. URL https://github.com/fomorians/forward-models.

Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning, 2020.

Li Fu and Pengfei Li. The research survey of system identification method. In *2013 5th International Conference on Intelligent Human-Machine Systems and Cybernetics*, volume 2, pp. 397–401, 2013. doi: 10.1109/IHMSC.2013.242.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1861–1870. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/haarnoja18b.html.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Alex Hernández-García and Peter König. Data augmentation instead of explicit regularization. *arXiv preprint arXiv:1806.03852*, 2018.

Brian Kenji Iwana and Seiichi Uchida. An empirical survey of data augmentation for time series classification with neural networks. *PLOS ONE*, 16(7):1–32, 07 2021. doi: 10.1371/journal.pone.0254841. URL https://doi.org/10.1371/journal.pone.0254841.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, volume 32, Red Hook, NY, USA, 2019. Curran Associates Inc.

Michael I Jordan and David E Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive science*, 16(3):307–354, 1992.

Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

Diganta Misra. Mish: A self regularized non-monotonic neural activation function. *arXiv preprint arXiv:1908.08681*, 2019. URL https://arxiv.org/abs/1908.08681.

Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based reinforcement learning: A survey. *CoRR*, abs/2006.16712, 2020. URL https://arxiv.org/abs/2006.16712.

Vedhas Pandit and Björn Schuller. The Many-to-Many Mapping Between the Concordance Correlation Coefficient and the Mean Square Error. *arXiv e-prints*, art. arXiv:1902.05180, February 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

Manuel A. Roehrl, Thomas A. Runkler, Veronika Brandtstetter, Michel Tokic, and Stefan Obermayer. Modeling system dynamics with physics-informed neural networks based on lagrangian mechanics. *IFAC-PapersOnLine*, 53(2):9195–9200, 2020. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2020.12.2182. URL https://www.sciencedirect.com/science/article/pii/S2405896320328354. 21st IFAC World Congress.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3 (1):9–44, 1988.

Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*. MIT press Cambridge, 1998.

Richard Stuart Sutton. *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst, 1984.

Michita Imai Takuma Seno. d3rlpy: An offline deep reinforcement library. In *NeurIPS 2021 Offline Reinforcement Learning Workshop*, December 2021.

Y Teh, E Dupont, and A Doucet. Augmented neural odes. *Advances in Neural Information Processing Systems 32 (NIPS 2019)*, 32(2019), 2019.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012.

Juanhui Tu, Hong Liu, Fanyang Meng, Mengyuan Liu, and Runwei Ding. Spatial-temporal data augmentation based on lstm autoencoder network for skeleton-based human action recognition. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pp. 3478–3482. IEEE, 2018.

David A van Dyk and Xiao-Li Meng. The art of data augmentation. *Journal of Computational and Graphical Statistics*, 10(1):1–50, 2001. doi: 10.1198/10618600152418584.

P.J. Werbos. Neural networks for control and system identification. In *Proceedings of the 28th IEEE Conference on Decision and Control,*, pp. 260–265 vol.1, 1989. doi: 10.1109/CDC.1989.70114.

Hiroyuki Yamada. cpprb, 1 2019. URL https://gitlab.com/ymd_h/cpprb.

Tianhe Yu, Garrett Thomas, Lantao Yu, Stefano Ermon, James Y Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma. Mopo: Model-based offline policy optimization. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 14129–14142. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/a322852ce0df73e204b7e67cbbef0d0a-Paper.pdf.

Wenle Zhang. Mimo system identification with extended madaline neural network trained by levenberg-marquardt method. In *The 26th Chinese Control and Decision Conference (2014 CCDC)*, pp. 1191–1196, 2014. doi: 10.1109/CCDC.2014.6852347.

A APPENDIX

A.1 EXTENDED DISCUSSION AND CONCLUDING REMARKS

Real Life motivation Reinforcement Learning is a very powerful technique for solving Industrial Control Problems. However, in the Industrial setting, there are many mission critical assets where RL cannot be applied in its canonical form, for security or operational risks, i.e., the risk of unsafe exploratory actions or the interruption of asset operation. Among the last lines of research on this matter, the "Offline Reinforcement Learning" formulation drops many barriers in the successful application of RL to these problems. Offline RL uses observations from the true industrial asset operation trough a dataset of trajectories (including control actions). This dataset is then used to perform offline RL to learn a control policy.

However, it is very common, by causes of operational reasons, that not all control actions are well represented in the dataset (out of distribution actions –OOD actions–), and thus the estimated value of such actions may be largely over/under estimated. The reason is that the returns of such OOD actions can not be learnt (penalized/rewarded) during training, since there is no interaction with the true environment. Hence, OOD actions become untested actions with maximal uncertainty in its estimated values. This fact, induces poor (or completely wrong) policies withing the canonical RL setting, and the performance of the current state of the art Offline RL algorithms is being actively investigated. However, if a forward model is provided then RL algorithms can explore on it, and avoid or mitigate the problem of ODD actions.

Along this line, the natural improvement over the presented results, is being able to understand how good or bad our forward models represent the real environment dynamics (beyond the information contained in source datasets and error metrics). In order to analyse this key point, the goal is to train standard RL agents on these models and compare their performance against real environment trained ones.

Although we have done initial steps on this direction, setting up, and controlling the experimental setup of such experiments must be further studied, since our initial tests did not yield the expected results. We argue that once these tests are completed, their results will give us a better understanding of the robustness of our approach.

The claim First, we want to remark that the presented comparison between Supervised Learning and Reinforcement Learning is not about algorithms, but problem paradigms. We do not deny that Supervised Learning, with enough extra features, can be used to solve the problem as effectively as any Reinforcement Learning approach or even better. However, what we argue is that the procedure to build such specialized Supervised Learning method signals the quest to solve a problem that is more naturally stated as a Reinforcement Learning problem or a general sequential decision problem.

The proposed method There are many other aspects to consider for further optimization of the learning procedure, for instance:

Does this problem impose a special constraint to the Q-function? Since we know that a sequence of perfect actions should return 0 as rollout error, it can be argued that we know in advance the right returns of many (s, a) pairs if we do not consider signal noise and a source of randomness in the collected data. We can even consider that these sources exist, but are negligible so we can then explicitly impose such constraints into the learning of the Q-function.

Along this line, one tested trick is to select random tuples $(s, a, s', r = 0)$ from the datasets and update the critic network with such tuples with the aim of improving learning convergence. However, in our initial experiments, using this trick cause the Q-function to diverge if the ratio of this updates vs. the updates from the replay buffer is not controlled.

Transferring the knowledge from the policy to the Q-function Another trick is to try training the policy network via Supervised Learning with some random (sa, s') pairs so that the policy can take advantage of Supervised Learning as well. But this will require a special procedure to update the critic network specifying confidence on some (s, a, R) tuples. This looks analogous to the proce-

dures used by offline-RL algorithms to constraint the Q-values on out-of-distribution (OOD) points. We have tested this trick by helping the policy learning using random (sa, s') to try to improve learning, however without a clear procedure to update the critic network as well, there is a fierce competition between the updates to the policy to optimize the critic and the Supervised Learning ones. Besides, thee results show also a divergence in the Q-function and the policy learning.

Avoiding the hacker agent against a learned model A known common concern is the idea that an agent trained over a learned forward model may exploit weaknesses of the model to gain unfair advantage in optimizing the reward function (the cheating effect). Some authors have argued that this can be alleviated due to the inherent noise contained in the learned forward-model. Another simple idea is to over constraint the reward function for the learned forward models. We think that a future research direction is how to prevent this cheating effect. However, in our early experiments we can not conclude that the agents systematically gain unfair advantages. Instead, what we observe is just slightly different behaviors that of course yield different returns, but not a clear trend to optimize over the model, degrading the performance in the original environment.

A.2 HARDWARE AND SOFTWARE

For all the experiments we have used a Linux virtual machine with enough system RAM and 4 Nvidia Tesla-T4 GPUs. The experiments presented here rely on the following Deep Reinforcement Learning specific software: for the environments (problems to solve) OpenAI Gym (Brockman et al. (2016)) and MuJoCo (Todorov et al. (2012)) are used. For Deep RL algorithms, the SAC implementation in the d3rlpy library (Takuma Seno (2021)) is used. For implementing a replay buffer we do not use the d3rlpy's replay buffer implementation, instead the cpprb library (Yamada (2019)) is used. Finally, as the Deep Learning backend and framework, Pytorch (Paszke et al. (2019)) is used. D4RL datasets (Fu et al. (2020)) are used as as the source of dynamics trajectories. Every library and asset mentioned in this paper and used in this research has the license to be used without restriction.

A.3 ENVIRONMENTS' SPACES

In this section, we present the list of variables of the state spaces for both the original problem $\mathcal{M}$ and the forward model approach $\mathcal{M}_{\mathcal{F}}$ for the three MuJoCo environment used in the experimental evaluation. We recall that in the case of the forward model environment we have decided to predict only the position's state variables (target variables) because the velocities can be derived from them in order to reduce complexity of the problem. Those variables have been marked with an (*).

Table 2: State space variables for MuJoCo and derived forward environments

Hopper	Walker2d	Halfcheetah
rootx pos*	rootx pos*	rootx pos*
rootz pos*	rootz torso*	rootz pos*
rooty angle*	rooty torso angle*	rooty pos*
thigh joint angle*	thigh joint angle*	bthigh angle*
leg joint angle*	leg joint angle*	bshin angle*
foot joint angle*	foot joint angle*	bfoot angle*
	thigh left joint angle*	fthigh angle*
	leg left joint angle*	fshin angle*
	foot left joint angle*	ffoot angle*
rootx vel	rootx vel	rootx vel
rootz vel	rootz vel	rootz vel
rooty angle vel	rooty angle vel	rooty angle vel
thigh joint angle vel	thigh joint angle vel	bthigh angle vel
leg joint angle vel	leg joint angle vel	bshin angle vel
foot joint angle vel	foot joint angle vel	bfoot angle vel
	thigh left joint angle vel	fthigh angle vel
	leg left joint angle vel	fshin angle vel
	foot left joint angle vel	ffoot angle vel

A.4 TRAINING DATA PREPARATION

Training data for the forward model have been extracted from D4RL repositories (Fu et al. (2020)). For each example, several datasets from this library have been included in order to get a large and diverse collection of trajectories (see Table 3). Additionally, and for training purposes they have been filtered only those experiences with episodes longer than a minimum number of steps (500 for the main experimental results and 100 for the additional tests). This filtering aims to find longer and more stable episodes to learn.

Table 3: D4RL source datasets used per experiment.

Hopper-v2	Walker2d-v2	Halfcheetah-v2
hopper-random-v2	walker2d-random-v2	halfcheetah-random-v2
hopper-medium-v2	walker2d-medium-v2	halfcheetah-medium-v2
hopper-expert-v2	walker2d-expert-v2	halfcheetah-expert-v2
hopper-medium-replay-v2	walker2d-medium-replay-v2	halfcheetah-medium-replay-v2
hopper-medium-expert-v2	walker2d-medium-expert-v2	halfcheetah-medium-expert-v2

A.5 TRAINING PROCEDURE

Here, you can find the main algorithms of the presented forward models (Algorithms 1, 2, and 3).

Algorithm 1 Forward model main training loop

```
Initialize replay-buffer (RB), SAC algorithm (SAC) and forward model GYM environment (FW)

total steps = 0
for episode do
    steps = 0
    total reward = 0
    results = collect(FW, SAC) {(described at algo.2)}
    for sample in results do
        RB ← sample {(append)}
        steps += sample[steps]
        total reward += sample[reward]
    end for
    total steps += steps
    for i in range(10) do
        samples = RB.sample(SAC.batchsize)
        loss = SAC.update(samples)
    end for
    Update metrics
    if episode % 100 == 0 then
        Save SAC model and policy
    end if
end for
```

Algorithm 2 Collect Algorithm

Initializate $buffer$. env and SAC included as argument.

state = env.reset()
for n in $range(10000)$ **do**
  if explore **then**
    action = SAC.sample(state)
  else
    action = SAC.predict(state)
  end if
  step = env.step(a) {described at algo.3}
  state = step[s]
  $buffer \leftarrow step$ {append}
end for
return $buffer$

Algorithm 3 Forward model environment step algorithm

Initializate observation as $stack$, predicted state as $buffer$ and true state as $buffer$. dataset included as argument, with the real experiences from d4rl datasets. action performed by SAC agent (sac action) is passed as argument.

step counter $+ = 1$
rollout step counter $+ = 1$
true obs = dataset[step counter][observation]
true action = dataset[step counter][action]
predicted state = observation + sac action
observation = predicted state
if rollout step counter $>=$ rollout step **then**
  observation = true obs
  rollout step counter = 0
  rollout terminal = True
end if
predicted $\leftarrow$ predicted state, action, rollout terminal {append}
true state $\leftarrow$ true obs, true action, terminal {append}
if step counter $>=$ total episode length **then**
  terminal = True
else
  terminal = False
end if
reward = getReward(predicted state, true state, terminal)
return observation, reward, terminal

A.6 ADDITIONAL EXPERIMENTAL RESULTS

In this section, there are some detailed additional results that may result useful in order to understand not only the training stage (i.e., actor and critic losses figures, as well as detailed training main metrics Table 1), but also the testing of our forward model on dataset actions.

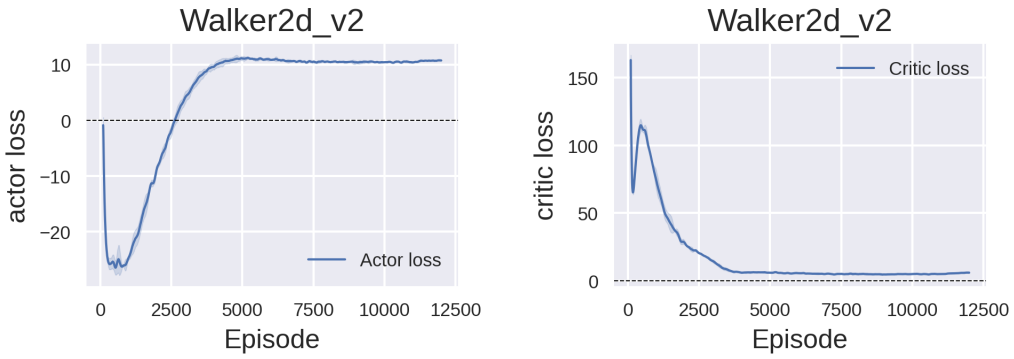

Figure 6: Walker2d SAC Actor loss and Critic loss (training) per episode

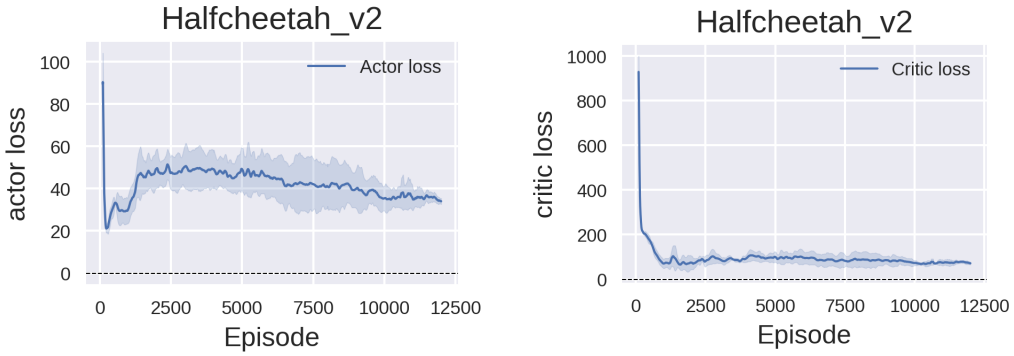

Figure 7: HalfCheetah SAC Actor loss and Critic loss (training) per episode

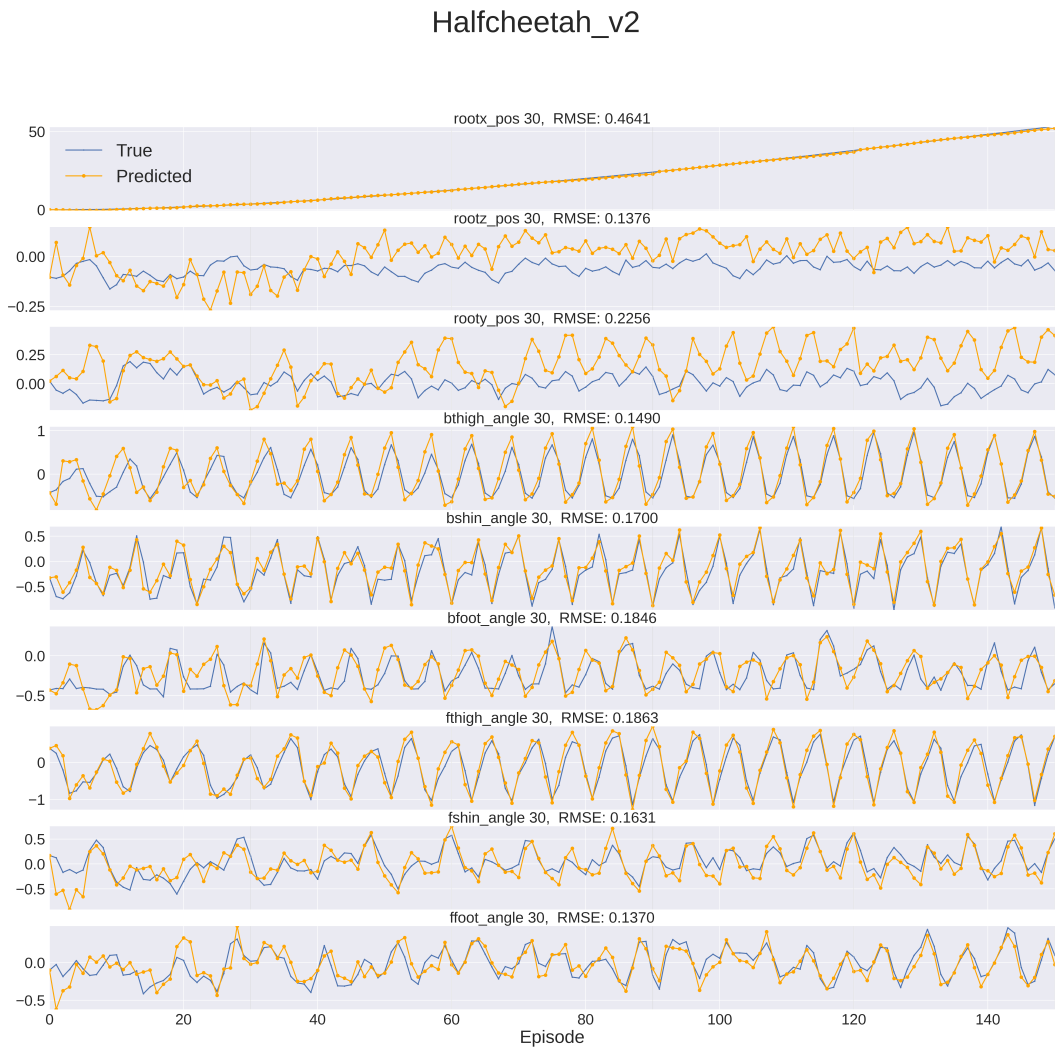

Figure 8: Halfcheetah forward model prediction of target variables, compared to real values. Rollouts of 30 steps. Sample of 150 steps (5 full rollouts of 30 steps each) obtained from a full random episode.

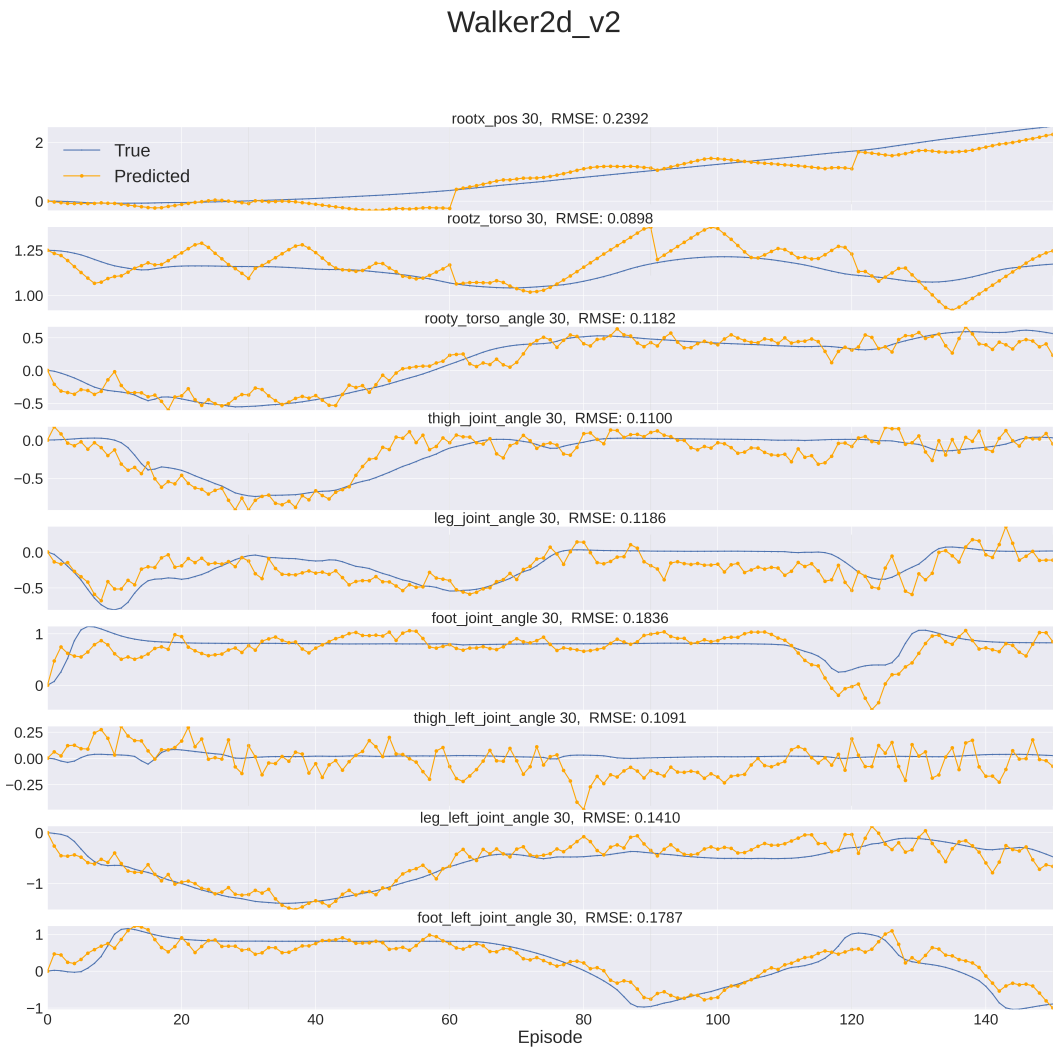

Figure 9: Walker3d forward model prediction of target variables, compared to real values. Rollouts of 30 steps. Sample of 150 steps (5 full rollouts of 30 steps each) obtained from a full random episode.

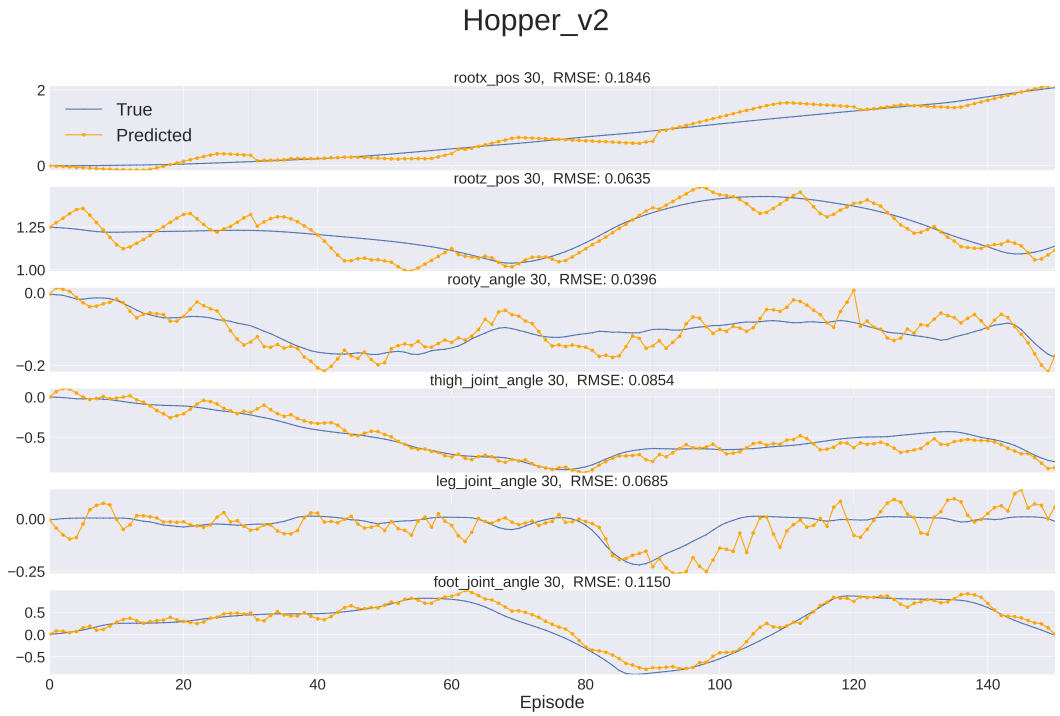

Figure 10: Hopper forward model prediction of target variables, compared to real values. Rollouts of 30 steps. Sample of 150 steps (5 full rollouts of 30 steps each) obtained from a full random episode.