Continuous-Time Model-Based Reinforcement Learning
Supplementary Material

Çağatay Yıldız 1  Markus Heinonen 1  Harri Lähdesmäki 1

S1. Technical Discussion

S1.1. The Form of the Differential Function

For notational convenience, we defined the problem as a first-order ODE system:

\[
\frac{ds(t)}{dt} = f(s(t), a(t)).
\]

We now explain possible extensions to this formulation to inject problem-specific prior knowledge.

Linearity w.r.t. action In many Newtonian systems, the action affects the time differential linearly, a widespread assumption in CT control literature (Doya, 2000). Estimating such systems with an arbitrary function of state and action would tie the action and the dynamics in a nonlinear way, which would render the learning problem unnecessarily complicated. Therefore, we propose to decompose the differential function as follows (Vamvoudakis and Lewis, 2010):

\[
\frac{ds(t)}{dt} = f(s(t)) + h(s(t)) \cdot a(t).
\]

where \( h : \mathbb{R}^d \rightarrow \mathbb{R}^{d \times m} \) and \( \cdot \) denotes the standard matrix-vector product. Above formulation assumes an additive differential function of dynamics and control component. The former aims to learn the evolution of the system under zero force whereas the latter defines a manifold the action is projected onto. Since we approximate these functions with neural networks, both the dynamics and the manifold can be arbitrarily complicated.

Second-order dynamics Most dynamical systems can be expressed in terms of position \( s \in \mathbb{R}^d \) and velocity \( v \in \mathbb{R}^d \) components. Such decomposition of the state space is shown to better explain the phenomena of interest if the underlying physics is indeed second-order (Yildiz et al., 2019). Formally, a second-order dynamical system with control is defined as follows:

\[
\frac{ds(t)}{dt} = v(t), \quad \frac{dv(t)}{dt} = f(s(t), v(t), a(t)).
\]

Here, \( f : \mathbb{R}^{2d+m} \rightarrow \mathbb{R}^d \) is referred to as acceleration field.

Hamiltonian dynamics Zhong et al. (2020) already describes Hamiltonian dynamics in RL context very clearly; however, we include this subsection for completeness. Hamiltonian mechanics reformulate classical physical systems in terms of canonical coordinates \((q, p)\) with \( q, p \in \mathbb{R}^d \), where \( q \) denotes generalized coordinates and \( p \) is their conjugate momenta. The time evolution of a Hamiltonian system is defined as

\[
\frac{dq}{dt} = \frac{dH}{dp} \quad \frac{dp}{dt} = -\frac{dH}{dq}
\]

where \( H(q, p) : \mathbb{R}^{2d} \rightarrow \mathbb{R} \) denotes the Hamiltonian. For simplicity, we assume a time-invariant Hamiltonian, which also corresponds to the total energy of the system. Typically, Hamiltonian is decomposed into a sum of kinetic energy \( T \) and potential energy \( V \):

\[
H = T + V, \quad T = \frac{p^TM^{-1}(q)p}{2m}, \quad V = V(q)
\]

where \( m \) denotes the mass. In simple systems such as pendulum, the mass matrix \( M(q) \) is an identity matrix, implying Euclidean geometry. More complicated systems like cart-pole requires learning the geometry. Given a Hamiltonian decomposing like above, the dynamics become

\[
\frac{dq}{dt} = M^{-1}(q)p, \quad \frac{dp}{dt} = -\frac{dV}{dq}
\]

The dynamics learning problem reduces to learning a potential energy function \( V(q) : \mathbb{R}^d \rightarrow \mathbb{R} \), whose derivative gives the time evolution of momentum, and estimating the geometry through its Cholesky decomposition \( L, i.e., LL^{-1} = M \), via an additional neural network.

S1.2. Greedy policy

A greedy policy is defined as the one that minimizes the Hamilton–Jacobi–Bellman equation:

\[
V^*(s) = \min_a \left[ \frac{dV(s)}{ds} \cdot f(s, a) + r(s, a) \right]
\]
The greedy policy can be expressed in closed form if (i) the reward is of the form \( r(s, a) = r_a(s) + r_d(a) \) with an invertible action reward and (ii) the system dynamics is linear with respect to the action as in eq. (S1.1) (Doya, 2000; Tassa and Erez, 2007):

\[
a_t^* = -\frac{dr_a^{-1}}{da} \left( f(s_t, a)_T dV(s)_T \right)
\]

(2)

Consequently, given the above assumptions are satisfied, the optimal policy can be expressed in closed form for a given value function, which would obviate the need for an actor. We leave the investigation of model-based greedy policy as an interesting future work.

**S2. Experiment Details**

This section consists of experimental details which are not included in the main text.

**S2.1. Environments**

The environment-specific parameters are given in Table S1. In all environments, the actions are continuous and restricted to a range \([-e_{\text{max}}, e_{\text{max}}]\). Similar to Zhong et al. (2020), we experiment with the fully actuated version of the Acrobot environment since no method was able to solve the under-actuated balancing problem.

**S2.2. Reward Functions**

Assuming that each observation \( s = (q, p) \) consists of state (position) \( q \) and velocity (momentum) \( p \) components, the differentiable reward functions have the following form:

\[
r(q, p, a) = \exp \left( -\frac{||q - s^{\text{goal}}||^2}{2} - c_p ||p||^2 - c_a ||a||^2 \right)
\]

where \( c_p \) and \( c_a \) denote environment-specific constants. The exponential function aims to restrict the reward in a range \([0,1]\) minus the action cost, which aids learning (Deisenroth and Rasmussen, 2011). The constants \( c_p \) and \( c_a \) are set so that they (i) penalize large values, and (ii) do not enforce the model to stick at trivial local optima such as the initial state.

**Goal states**

The goal state \([0, \ell]\) in Pendulum environment corresponds to \( x \) and \( y \) coordinates of pole’s tip, with \( \ell \) being the length of the pole. The reward is maximized when the pole is fully upright. In CartPole, this state is concatenated with \( 0 \), which represents the cart’s target location. Finally in Acrobot, the goal state involves the \( x \) and \( y \) coordinates of the second link only (hence \( 2D \)).

**S2.3. Dataset**

**Initial dataset**

Each experiment starts with collecting an initial dataset of \( N_0 \) trajectories at observation time points with length \( T = 50 \). In all environments, the initial state is distributed uniformly:

\[
s_0 \sim \mathcal{U}([-s^{\text{box}}, s^{\text{box}}]).
\]

Initial random actions are drawn from a Gaussian process:

\[
a_t \sim \mathcal{G}\mathcal{P}(0, K(t, t')).
\]

The inputs to the GP are the observation time points. We opt for a squared exponential kernel function with \( \sigma = 0.5 \) and \( \ell = 0.5 \). The output of the GP is followed by a TANH function and multiplied with \( d_{\text{max}} \).

**Data collection**

After each round, the policy is executed once in the environment starting from an initial state drawn from the environment. This is followed by the execution of \( N_{\text{exp}} \) exploring policy functions. Similar to the idea of perturbing policies with an Ornstein–Uhlenbeck process for exploration (Lillicrap et al., 2016), we add draws from a zero-mean Gaussian process to the policy for smoother perturbations:

\[
\pi^{\text{explore}}(s(t), t) := \pi(s(t)) + z(t)
\]

\[
z(t) \sim \mathcal{GP}(0, k(t, t'))
\]

where the input to the GP is a set of time points. We again use a squared exponential kernel function with \( \sigma = 0.1 \) and \( \ell = 0.5 \). We choose the initial values for the exploring data sequences from the previous experience dataset randomly, where each state has a weight proportional to the dynamics estimator’s variance at that state.

**S2.4. Training Details**

We used ADAM optimizer to train all the model components (Kingma and Ba, 2014). More explanation is as follows:

- **NODE Dynamics**: We initialize the differential function by gradient matching:

\[
f(s_t, a_t) \approx \frac{s_{t+1} - s_t}{t_{t+1} - t_t}
\]

Regardless of how observation time points are distributed, we use subsequences of length \( t_s = 5 \) to train the dynamics model. We randomly pick 5 subsequences from each data trajectory to reduce gradient stochasticity. While training the neural ODE model, we start with an initial learning rate of 1e-4, gradually increase it to 1e-3 in 100 iterations, and then proceed \( N_{\text{dyn}} = 1250 \) iterations with the latter learning rate.

- **Discrete Dynamics**: We train PETS and deep PILCO with the algorithms given respective papers.

- **Actor-Critic**: In each round, we form a dataset of \( N_0 \) initial values from the experience dataset. We chose to
exclude the sequences collected with exploring policy
as they may explore states that are undesirably far from
the goal. We set $N_{ac} = 250$.

S2.5. Neural Network Architectures

The dynamics, actor and critic functions are approximated
by multi-layer perceptrons. In all methods and environ-
ments, we used the same neural network architectures,
which are given as follows:

- **Dynamics**: 3-hidden layers, 200 hidden neurons and
ELU activations. We experimentally observed that
dynamics functions with ELU activations tend to ex-
trapolate better on test sequences. Therefore, training
the dynamics model with an augmented dataset (after
each round) becomes much more robust.

- **Actor**: 2-hidden layers, 200 hidden neurons and
RELU activations. Based on the idea that optimal
policies can be expressed as a collection of piece-wise
linear functions, we opt for RELU activations. Neural
network output goes into TANH activation and multi-
plied with $\sigma_{max}$.

- **Critic**: 2-hidden layers, 200 hidden neurons and
TANH activations. Since the state-value functions must
be smooth, TANH activation is more suitable compared
to other activations. We empirically observed critic net-
works with RELU activations easily explode outside
the training data, which deteriorates the learning.

S2.6. Additional Results

Predictive dynamic errors on shorter sequences are illus-
trated in Figure S2. We see that future MSEs are much
lower compared to $H = 2$ while they still cannot directly
predict overall model performance $V(s_0)$.

S3. ODE Solver Comparison

In this ablation study, we ask two questions in relation with
the simulation environment and numerical integration: (i)
Which numerical ODE solver one should use, (ii) To what
extent our continuous framework differ from its discrete
counterparts? To answer, we have built a simple experiment
on CartPole environment where several ODE solvers are
compared: three adaptive step solvers (dopri5 (RK45),
RK23 and RK12), five fixed step solvers (RK4 with 1/10
intermediate steps, and Euler with 10/100/1000 interme-
diate steps), as well as discrete transitions. Due to the lack
of a closed-form ODE solution, true ODE solutions are ob-
tained by Runge-Kutta 7(8) solver, the numerical integrator
which achieves the smallest local error to the best of our
knowledge (Prince and Dormand, 1981).

Each ODE solver takes as input the same set of initial values
as well as twenty different policy functions, some of which
solve the problem whereas some are sub-optimal. Figure
S1 demonstrates the distance between the true state solu-
tions and those given by different ODE solvers. The most
striking observation is that discrete transitions of the form
$s_{t+1} - s_t = h \cdot f(s_t, a_t)$ are highly erroneous. Moreover,
adaptive solvers as well as fixed-step solvers with suffi-
ciently many intermediate steps attain practically zero error.
Unsurprisingly, approximate state solutions deteriorate over
time since the error accumulates. In our experiments, we
use RK78 to mimic the interactions with the real world, and
dopri5 to forward simulate model dynamics.

![Figure S1. Error estimates of different numerical integration meth-
ods plotted against integration time.](image)

S4. Additional Related Work

**Model-based RL** The majority of model-based reinforce-
ment learning methods assumes auto-regressive transitions,
which effectively learn a distribution over the next state
given the current state and action. Unknown transitions
are typically approximated by a Gaussian process (Kocijan
et al., 2004; Deisenroth and Rasmussen, 2011; Kamthe and
Deisenroth, 2017; Levine et al., 2011), multi-layer percep-
tron (MLP) (Gal et al., 2016; Depeweg et al., 2017; Naga-
bandi et al., 2018; Chua et al., 2018) or recurrent neural
network (Ha and Schmidhuber, 2018; Hafner et al., 2018;
2020). Such models are typically developed in conjunction
with model predictive control (Richards, 2005) used for
planning or with a parametric policy.

**Continuous-time RL** In addition to the works discussed
earlier (Baird, 1993; Doya, 2000), Bradtke and Duff (1994)
developed Q-functions and temporal different learning in
the context of semi-Markov decision processes. Abu-Khalaf
and Lewis (2005) proposed a policy-iteration algorithm for
the optimal control of CT systems with constrained con-
trollers. An online version of this algorithm was derived in
(Vrabie and Lewis, 2009), which was extended in a series
of papers (Luo et al., 2014; Modares et al., 2016; Zhu et al.,
2016; Lee and Sutton, 2019). A direct least-squares solu-
tion to Hamilton-Jacobi-Bellman equation was studied in
Continuous-Time Model-Based Reinforcement Learning
Supplementary Material

Table S1. Environment specifications

<table>
<thead>
<tr>
<th>Environment</th>
<th>$N_0$</th>
<th>$N_{exp}$</th>
<th>$c_p$</th>
<th>$c_a$</th>
<th>$a_{max}$</th>
<th>$b_{box}$</th>
<th>$b_{goal}$</th>
<th>Max. execution time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PENDULUM</td>
<td>3</td>
<td>1e-2</td>
<td>1e-2</td>
<td>2</td>
<td></td>
<td></td>
<td>0, [3]</td>
<td>12</td>
</tr>
<tr>
<td>CARTPOLE</td>
<td>5</td>
<td>2</td>
<td>1e-2</td>
<td>3</td>
<td></td>
<td>[0.05, 0.05, 0.05, 0.05]</td>
<td>[0, 0.1]</td>
<td>24</td>
</tr>
<tr>
<td>ACROBOT</td>
<td>7</td>
<td>3</td>
<td>1e-4</td>
<td>4</td>
<td></td>
<td>[0.1, 0.1, 0.1, 0.1]</td>
<td>[0, 2]</td>
<td>24</td>
</tr>
</tbody>
</table>

Figure S2. Predictive mean squared error of different dynamics models, computed after each round on a fixed test set.

(Tassa and Erez, 2007), requiring no forward integration for value estimation but a bag of tricks to deal with numerical instabilities. In a related work, Mehta and Meyn (2009) proposed an adaptive controller for nonlinear CT systems via a continuous-time analog of the Q-function. Above-mentioned methods are either built upon known dynamics or they are model-free. In either case, the dynamics are assumed to be linear with respect to the action, a premise needed for closed-form optimal policies.

Neural ODEs In their ground-breaking work, Chen et al. (2018) show that simple multi-layer perceptrons (MLP) can be utilized for learning arbitrary continuous-time dynamics. The resulting model, called Neural ODEs (NODEs), have been shown to outperform its RNN-based, discrete counterparts in interpolation and long-term prediction tasks (Chen et al., 2018). The vanilla NODE model paved the way for advances in continuous-time modeling, such as second-order latent ODE models (Yildiz et al., 2019), augmented systems (Dupont et al., 2019), stochastic differential equations (Jia and Benson, 2019; Li et al., 2020), and so forth. NODE framework also allows encoding prior knowledge about the observed phenomena on the network topology, which leads to Hamiltonian and Lagrangian neural networks that are capable of long-term extrapolations, even when trained from images (Greydanus et al., 2019; Cranmer et al., 2020).

Neural CTRL The NODE breakthrough has opened a new research avenue in CTRL. In particular, physics-informed continuous-time dynamical systems have gained popularity. For example, Lagrangian mechanics are imposed on the architecture presented in (Lutter et al., 2019), which results in near-perfect real-time control of a robot with seven degrees of freedom. Hamiltonian framework is proven useful for inferring controls from generalized coordinates and momenta (Zhong et al., 2020). Later in Zhong and Leonard (2020), an interpretable latent Lagrangian dynamical system and controller were trained from images. In addition to dynamics learning, above-mentioned methods describe model-specific recipes for learning controls.

References


