Parallel On-Line Temporal Difference Learning for Motor Control

Wouter Caarls, *Member, IEEE,* and Erik Schuitema

***Abstract*—Temporal difference (TD) learning, a key concept in reinforcement learning, is a popular method for solving simulated control problems. However, in real systems this method is often avoided in favor of policy search methods because of its long learning time. But policy search suffers from its own drawbacks, such as the necessity of informed policy parameterization and initialization. In this paper, we show that temporal difference learning can work effectively in real robotic systems as well, using parallel model learning and planning. Using locally weighted linear regression and trajectory sampled planning with 14 concurrent threads, we can achieve a speedup of almost two orders of magnitude over regular TD control on simulated control benchmarks. For a real-world pendulum swing-up task and a two-link manipulator movement task, we report a speedup of 20x-60x, with a real-time learning speed of less than half a minute. The results are competitive with state-of-the-art policy search.**

***Index Terms*—reinforcement learning, motion control, predictive models, parallel programming, real-time systems**

I. Introduction

REINFORCEMENT learning [1] has a long history of being applied to motor control problems, although mostly in simulation. Actual real-world applications suffer from many problems, such as noise, control delay, wear and tear, and timing constraints [2]. However, the most pressing problem is the huge amount of experience required to find a good control policy. As the collection of experience cannot be sped up, it is necessary to reduce the required amount.

A popular solution has been to increase the amount of prior knowledge by searching in a reduced space of parameterized control policies [3]. This has proven very effective, leading to quick learning and smooth control, but the required knowledge reduces the generalizability and autonomy of the learning system [4]. Value-based methods such as temporal difference (TD) learning require less prior knowledge, but have the aforementioned problem of requiring many samples, especially in continuous state spaces. We show that a multi-threaded implementation of the DYNA framework [5] for on-line model learning and planning dramatically increases the sample efficiency of TD control. By doing *mental rehearsal* on the learned model we can quickly estimate a value function based on just a few samples, enabling the use of TD methods to control real dynamical systems. Our main contribution is doing this in parallel, in a continuous state space using function approximation and a model suitable for motor control. Parallel processing is needed because the number of cores is the only vector along which computing power still increases according to Moore's law [6].

The paper is organized as follows: we start with reviewing related work and relevant methods, in Sections II and III. Next, we introduce our proposed algorithms and system models, in Sections III and IV, respectively. We continue with the experiments and their validation, in Sections VI and VII. Finally, Section VIII presents concluding remarks.

II. Related work

Temporal difference learning has been applied to real robotic systems before (see [3] for a survey). Often this is done in discrete state spaces [7], [8]. This works well in small state spaces (a few hundred states), when the actions themselves are feedback controllers, or when the required accuracy is low. Another tactic is pre-training in simulation or priming with a predefined or demonstrated policy [9], [10]. By such an initialization of the value function, generally a good policy can be found with just local changes. However, this introduction of prior knowledge is what we wanted to avoid.

Parameterized control policies may also be used in the TD setting, leading to algorithms that use an estimated value function to update an explicitly represented control policy (actor-critic, [11]). Such algorithms share many of the properties of policy search, but have lower variance in the computed gradients and allow for very loosely-defined policy parameterizations [12], reducing prior knowledge. The DYNA framework was first formulated for actor-critic systems (DYNA-PI), but showed more promise without an explicit policy representation (DYNA-Q) [13].

A good strategy to reduce the number of required samples is to use batch-mode reinforcement learning [14], [15], in which all samples are used collectively to estimate the value function. This allows for the use of advanced supervised learning methods, which can have better generalization properties. However, while the samples are being processed off-line, the system is not controlled, or at least controlled with an outdated policy.

Manuscript received December 9, 2014; revised March 9, 2015 and May 21, 2015; accepted May 31, 2015. Date of publication Xxxx 00, 2015; date of current version Xxxx 00, 2015. This work was supported in part by the European Union within the 7th Framework Program under Grant Agreement No. 611909, and in part by CAPES, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil, in the context of the Science without Borders program – CsF.

W. Caarls is with the Postgraduate Program in Informatics (DCC/NCE), Federal University of Rio de Janeiro, Rio de Janeiro, Brazil and the Department of Biomechanical Engineering, Delft University of Technology, Delft, The Netherlands (e-mail: wouter@caarls.org)

E. Schuitema is with ESSD Engineering Software, Delft, The Netherlands (e-mail: erik@essd.nl)

Digital Object Identifier 00.000/TNNLS.0000.0000000

Instead of using the samples to estimate the value function directly, we can also use them to learn the system dynamics [16]. This learned model may then be used to increase the efficiency of direct updates [12], or – the solution adopted in this paper – to generate simulated experience. Just like real experience, simulated experience can be used in batches [17], or it can be intermixed with regular value function updates [5]. In the latter case, all processing is done on-line, and the most up-to-date policy is always available for controlling the agent. Such an *anytime* approach has proven successful in previous work with discrete value functions [18], [19].

To vastly increase the number of model updates between control steps, we employ parallel processing by running many planning threads in parallel. Such parallelization has until now only been used to learn from multiple agents [20] or to learn multiple tasks [21], and not to speed up learning a single task with a single agent. Although DYNA has previously been applied in a parallel setting [22], it was not itself parallelized. Instead, each agent ran a separate instance of the DYNA algorithm.

A different parallelization approach, orthogonal to ours, is to parallelize the readout or update of the function approximator itself. This is most efficient when using heavy weight function approximators such as used in batch-mode techniques [23], but on-line temporal difference learning may benefit as well [24].

III. Methods

A. Reinforcement Learning

In reinforcement learning [1], we try to find an optimal policy $\pi : \mathcal{S}, \mathcal{A} \rightarrow [0, 1]$ for a Markov Decision Process $< \mathcal{S}, \mathcal{A}, T, R >$, with $\mathcal{S}$ a set of states s, $\mathcal{A}$ a set of actions a, $T : \mathcal{S}, \mathcal{A}, \mathcal{S} \rightarrow [0, 1]$ a state transition function and $R : \mathcal{A}, \mathcal{S} \rightarrow \mathbb{R}$ a reward function. In model-free temporal difference (TD) learning, this is done by estimating a state-action value function $Q^\pi : \mathcal{S}, \mathcal{A} \rightarrow \mathbb{R}$, which indicates for each state-action pair (s, a) the expected return R_t of taking action a in state s and following π afterwards. The return is the discounted sum of future rewards r starting from the current timestep t:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1}$$

with γ a discount rate that determines the planning horizon. In TD control, the policy is derived from the value function by choosing the action with the highest expected return. To encourage exploration, an ϵ-greedy policy is often used, which chooses a random action with probability ϵ:

$$\pi(s, a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \max_{a'} Q^\pi(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{2}$$

The optimal value function Q^*, defining the optimal policy π^*, is the unique solution to the Bellman equation

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left(R(a, s') + \gamma \max_{a'} Q^*(s', a') \right). \tag{3}$$

It is calculated by sampling state transitions $(s, a) \rightarrow (r, s')$ along a trajectory and updating Q towards minimizing the temporal difference error δ. In the SARSA algorithm:

$$\delta = R(a, s') + \gamma Q(s', a') - Q(s, a) \tag{4}$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta \tag{5}$$

with α a learning rate that sets the parameter of an exponential moving average filter determined by the stochasticity of T and R.

B. Linear function approximation

In motor control, $\mathcal{S}$ is a continuous space with states $s \in \mathbb{R}^n$, in general consisting of joint positions and velocities. It is therefore necessary to approximate the value function Q. Linear approximations are especially favored, because they have the best convergence guarantees [25]. In this case, a state-action pair (s, a) is mapped onto a vector $\phi(s, a)$ of feature activations, and $Q(s, a)$ is approximated by

$$\hat{Q}(s, a) = \theta^T \phi(s, a), \tag{6}$$

where θ is the vector of learned parameters (weights). Both $\mathcal{S}$ and $\mathcal{A}$ may thus be continuous, although in general the $\max$ operator in Eq. 2 is still implementated by iterating over a discrete number of actions.

Updates to θ can be made using gradient descent, such that Eq. 5 becomes

$$\theta \leftarrow \theta + \alpha \nabla_\theta \hat{Q}(s, a)\delta \tag{7}$$

$$\leftarrow \theta + \alpha\phi(s, a)\delta. \tag{8}$$

In particular, in tile coding function approximation [26], each state-action pair activates one binary feature in a fixed number g of overlapping grids. The main advantage is that ϕ is sparse, which both increases the computational efficiency and the locality of updates.

C. DYNA

The SARSA algorithm uses each sample only once. Even when using SARSA(λ), which adds eligibility traces that cause recently visited states to be updated as well, this seems a waste. In the DYNA architecture [5], this problem is addressed by building a model of the state transition and reward functions using all experienced samples. After each TD update, K planning updates are performed by sampling experiences from the model. This may be done randomly, using prioritized sweeping [27], or by sampling trajectories of simulated experience. If the model generalizes well, the planning updates will move the value function in the direction of the optimal value function, speeding up the learning process. However, this is subject to a bias-variance trade-off and can not be guaranteed.

DYNA has shown great promise for control in discrete state spaces [18] and policy evaluation in continuous state spaces using function approximation, for which a convergence proof is available [28]. However, this is not the case for *control* with value function approximation. Additionally, the model learned by the original algorithm is a generic state transition model, requiring $O(|\mathcal{S}|^2|\mathcal{A}|)$ storage (although this may be sparse and

represented in the same feature space as the value function). We can exploit the limited stochasticity of motor control by using a more specific model.

In order to be computationally efficient the model should be fast to learn and evaluate, such that the planning updates can use the most recent information and many of them can be performed per control step. To avoid fluctuations in the induced value function, successive approximations should exhibit stable convergence, without asympotic bias, such that in the limit the planning updates coincide with experience.

Note that, as the model generalizes over unseen state transitions, it might erroneously predict low-reward dynamics in a region of state space. The low values induced by the erroneous model subsequently reduce the probability of visiting that region, which in turn prohibits the model from being corrected. Various "exploration bonus" strategies exist to remedy this situation [5], [29], but we have found that in our experiments the ϵ-greedy strategy provided sufficient exploration.

D. Locally weighted regression

As model approximator we use locally weighted linear regression (LWR, [30]), which satisfies our requirements and has proven very well suited to continuous motor control tasks [31], [32]. LWR is a nonparametric regression method that predicts the next state by fitting a linear model through experienced state transitions close to the query point.

All dimensions were scaled to their maximum range. For velocities we took a range such that it is not exceeded during normal operation, except for the two-link manipulator problem where they were saturated at $\pm 2\pi$ rad/s in the original formulation. Let

$$q = [s, a] \tag{9}$$

be a normalized query point for which we want to evaluate the model. We find the

$$k = 4 \cdot (\dim(\mathcal{S}) + \dim(\mathcal{A})) \tag{10}$$

nearest neighbors using approximate nearest neighbor search [33]. Each point in the model stores a state transition, including the originating state s, action a, difference in state $\Delta s = s' - s$, resulting reward r and whether the episode terminated t. The nearest neighbor search returns the input and output matrices

$$N_I = \begin{bmatrix} s_1, a_1 \\ s_2, a_2 \\ \cdots \\ s_k, a_k \end{bmatrix}, N_O = \begin{bmatrix} \Delta s_1, r_1, t_1 \\ \Delta s_2, r_2, t_2 \\ \cdots \\ \Delta s_k, r_k, t_k \end{bmatrix} \tag{11}$$

and the distances to the query point

$$d_i = \|[s_i, a_i] - q\|_2. \tag{12}$$

We use nearest neighbor bandwidth selection, so that the weight of each neighbor is given as

$$w_i = \sqrt{e^{-\left(\frac{d_i}{h}\right)^2}}, \tag{13}$$

where

$$h = \max_i d_i. \tag{14}$$

Defining W as the matrix with diagonal elements w_i, the weighted input and output matrices then become

$$\begin{aligned} A &= W[N_I, \mathbf{1}] \\ B &= W N_O \end{aligned} \tag{15}$$

and we solve

$$(A^T A)X = A^T B \tag{16}$$

using the Cholesky decomposition with Tikhonov regularization. The mean of the prediction is then simply

$$[\Delta s, r, t] = [q, 1]X. \tag{17}$$

The residual of the model fit

$$E = AX - B \tag{18}$$

can be used to calculate the prediction variance

$$\hat{\sigma_j}^2 = \frac{\sum_i E_{ij}^2}{n_{\mathrm{LWR}} - p_{\mathrm{LWR}}}, \tag{19}$$

where

$$n_{\mathrm{LWR}} = \sum_i w_i^2 \tag{20}$$

is a weighted measure of the number of data points used in the fit and

$$p_{\mathrm{LWR}} = \sum_i w_i^2 A_i^T (A^T A)^{-1} A_i \tag{21}$$

is a measure of the local number of free parameters in the model. The prediction of the next state and reward is then drawn from a normal distribution defined by the calculated mean and variance. The termination prediction, being a binary variable, is drawn from a Bernoulli distribution using only the mean (a linear probability model). Angles are wrapped between $-\pi$ and π.

The choice of distance function (scaling), number of neighbors to consider (Eq. 10), weighting function (Eq. 13) and bandwidth selection (Eq. 14) is heuristic. We experimented with various settings, but did not experience a large sensitivity to these parameters.

E. Sparse online Gaussian processes

To validate the choice of an LWR model, we compare it against another real-time model building algorithm, sparse online Gaussian processes [34]. Gaussian processes have recently proven successful in policy search [35], and are therefore a good candidate. A disadvantage is their complexity, which is cubic in the number of samples. SOGP therefore maintains a fixed set of basis vectors, automatically adding and removing appropriate samples when necessary. We used the default settings of the SOGP C++ library by Dan Grollman[1] (Gaussian RBFs with width w=0.1, $\sigma_0^2 = 0.1$), and 500 basis vectors.

[1]Code available at https://code.google.com/p/brown-rlab

IV. Algorithms

We will now introduce two new parallel reinforcement learning algorithms, parallel DYNA and parallel locally linear fitted Q-iteration. But first we will discuss specific issues related to parallelizing any algorithm which features concurrent access to a shared value function.

A. Parallelization

We have chosen not to parallelize the process of evaluating or updating the function approximator itself, but instead to perform multiple planning updates in parallel. As the linear function approximators usually employed in on-line temporal difference control have low computational complexity ($O(g)$ in tile coding), the synchronization overhead would be too high.

To avoid serializing the process, we do not employ mutual exclusion locks. This prevents us from performing collision detection on the hash table that usually stores the parameter vector for linear function approximators. As a consequence, a feature may participate in more than the intended region of state-space. While this does not affect convergence, it affects the approximation error. We chose the table size sufficiently large such that this rarely occurs in practice.

Evaluating the function approximator requires reading a number of separate weights. Without locking, these may be in an inconsistent state and lead to unpredictable results. However, as long as the approximated value from such an inconsistent state is a weighted average of the original and updated values, it can simply be interpreted as an update that was performed with a lower effective learning rate α. From Eqs. 6 and 8, this is clearly the case for linear function approximation if all elements of ϕ have the same sign. In the nonlinear case, the effective α is arbitrary and may lead to divergence.

Finally, the sets of weights written by two threads may overlap, in which case updates by one of the threads are partially lost. This again reduces the efficiency of the parallelization. It is therefore important for ϕ to be sparse, such that, in general, threads will be operating on different parts of the parameter vector. Tile coding satisfies this requirement, but using eligibility traces severely reduces the locality of updates.

B. Parallel DYNA

We are now ready to present parallel DYNA, our parallel implementation of the DYNA architecture that aims to drastically increase the number of planning updates per control step K. A schematic representation is depicted in Figure 2, while the algorithm itself is given in Algorithm 1. Instead of a single control loop, we have separate threads for the agent, model creation and planning [19]. The agent thread uses a policy derived from the current value function estimate $\hat{Q}$ and adds experienced transitions to the queue $\mathcal{T}$, which is used by the model thread to update the model $\hat{M}$. With all remaining computational power, the P DYNA threads use the model to plan using simulated experience, updating $\hat{Q}$.

The final value of K depends on P, the control step time, and the computational complexities of the model and value

Fig. 1. On-policy parallel DYNA on trajectories

Obtain initial model approximation $\hat{M}$ and action-value function approximation $\hat{Q}$
$\mathcal{T} \leftarrow \emptyset$
continuous loop AgentThread
  Initialize s, a
  repeat
    Take action a, observe r, s'
    Add $(s, a) \rightarrow (r, s')$ to $\mathcal{T}$
    Draw $a' \sim \pi(s', *)$
    Update $\hat{Q}(s, a)$ towards $r + \gamma \hat{Q}(s', a')$
    $s \leftarrow s', a \leftarrow a'$
  until episode ends
end loop
continuous loop ModelThread (1 Hz)
  for each $(s, a) \rightarrow (r, s')$ **in** $\mathcal{T}$ **do**
    Update $\hat{M}(s, a)$ towards (r, s')
    Remove $(s, a) \rightarrow (r, s')$ from $\mathcal{T}$
  end for
end loop
continuous loop DynaThread (P instances)
  Initialize s_m, a_m
  repeat
    Predict $\hat{r}, \hat{s}'_m$ using $\hat{M}$
    Draw $a'_m \sim \pi(\hat{s}'_m, *)$
    Update $\hat{Q}(s_m, a_m)$ towards $\hat{r} + \gamma \hat{Q}(\hat{s}'_m, a'_m)$
    $s_m \leftarrow \hat{s}'_m, a_m \leftarrow a'_m$
  until episode ends or $\hat{s}'_m$ becomes unreliable
end loop

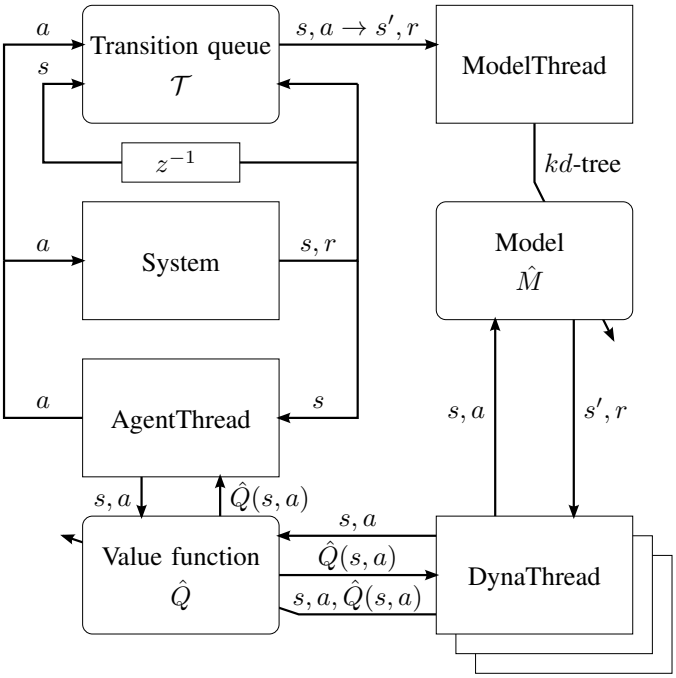


Fig. 2. Schematic representation of parallel DYNA. As LWR is instance-based, the model thread is only concerned with building the kd-tree necessary for fast nearest neighbor search.

function approximators. In our implementation, the model approximation $\hat{M}$ is made using locally weighted linear regression (LWR, [30]), which is dominated by the nearest neighbor search with complexity $O(k \log |\hat{M}|)$ and the Cholesky decomposition with complexity $O((\dim(\mathcal{S}) + \dim(\mathcal{A}))^3)$. The value function $\hat{Q}$ is approximated using hashed tile coding with $g = 16$ tilings of regularly displaced rectangles[2], which requires $O(|\mathcal{A}|g)$ to evaluate all actions. The linear approximation and sparse feature activations allow safe and efficient lock-free multithreaded interaction, which we will verify with experiments.

The policy is ϵ-greedy, and value function updates are made using SARSA(λ) with replacing eligibility traces truncated below 10^{-4}. Planning updates do not use eligibility traces, for reasons described in the previous section.

We use trajectory sampling of predicted experience starting from an experienced start state, meaning that we interact with the approximate model as if it were the real system. Planning continues until the prediction variance exceeds the range of the state parameters, marking an unreliable prediction; in that case, the planning episode is terminated and a new one started. This typically happens when extrapolating beyond the support of the model.

As we mentioned before, there is no proof of convergence for using DYNA with function approximation for control, and in fact using single-step random samples or prioritized sweeping showed only modest gains [28]. However, SARSA updates with trajectory sampling are known to converge [25] with a static environment. We can therefore expect DYNA to perform well if the model changes sufficiently slowly and does itself converge to an unbiased estimation. In that case, the framework may be viewed as successively solving increasingly accurate models of the environment [36].

C. Parallel locally linear fitted Q-iteration

Because of the success of batch-mode reinforcement learning algorithms in the Fitted Q-iteration (FQI) family [37], [38], we would like to compare our DYNA implementation with an algorithm of that kind. In FQI, batches of observed transitions $(s, a) \rightarrow (r, s')$ are iteratively converted into a sample set $(s, a) \rightarrow r + \gamma \max_{a'} \hat{Q}(s', a')$ by maximizing over the next action. This sample set is then used collectively to train the next iteration of the approximator $\hat{Q}$ using any supervised learning algorithm.

However, as stated before, we do not wish to interrupt the real-time control due to batch processing. This can be avoided by using an incremental supervised learning algorithm to train the approximator, for example kernel-based techniques [39]. The approximator can then be evaluated continuously, in parallel with the control loop. Such algorithms are similar to experience replay [40], but only store the values of the experienced states. Other values are interpolated by applying the kernel function to nearby states.

In our implementation we use LWR as the kernel regressor, so we call this locally linear fitted Q-iteration (LLFQI), shown in Algorithm 3. For a fair comparison, we use a parallel implementation with expected on-policy updates [41].

[2]The implementation was based on http://webdocs.cs.ualberta.ca/~sutton/tiles2.html

Fig. 3. Parallel locally linear fitted Q-iteration

1: $\mathcal{Q} \leftarrow \emptyset$
2: **continuous loop** AgentThread
3: Initialize s, a
4: **repeat**
5: Take action a, observe r, s'
6: **for each** $a' \in \mathcal{A}$ **do**
7: Calculate $\hat{Q}(s', a')$ using LWR on $\mathcal{Q}$
8: **end for**
9: Add (s, a, r, s') to $\mathcal{Q}$ with target value $r + \gamma \sum_{a'} \pi(s', a') \hat{Q}(s', a')$
10: Draw $a' \sim \pi(s', *)$
11: $s \leftarrow s', a \leftarrow a'$
12: **until** episode ends
13: **end loop**
14: **continuous loop** EvaluationThread (P instances)
15: Select random $q = (s, a, r, s') \in \mathcal{Q}$
16: **for each** $a' \in \mathcal{A}$ **do**
17: Calculate $\hat{Q}(s', a')$ using LWR on $\mathcal{Q}$
18: **end for**
19: Set target value of q to $r + \gamma \sum_{a'} \pi(s', a') \hat{Q}(s', a')$
20: **end loop**

Note that, as with parallel DYNA, we use approximate nearest neighbor search to determine the neighboring points. This requires the construction of a kd-tree, which is done periodically, similar to the ModelThread in Algorithm 1. Furthermore, the neighbors of a sample and part of the regression step are cached when the sample is added and only recalculated when a new kd-tree is built. In particular, Eq. 16 can be split into

$$(A^T A)Y = A^T \quad (22)$$

$$X = YB, \quad (23)$$

where the first part only depends on the neighbor positions and not their values.

The scheme is therefore very similar to Figure 2, except that the model $\hat{M}$ does not return next states, but abstract local models Y. The value function is then consulted to find the target values of neighboring points that make up B. The largest difference therefore is that parallel DYNA uses LWR to extrapolate in *transition space*, while parallel LLFQI extrapolates in *value space*.

Note that each update only writes a single (atomic) variable, the target value. As such, no inconsistencies due to larger weight sets as described in Section IV-A occur. The only consequence of parallelization is that a point may be updated with an unpredictable mix of older and newer data. However, that also occurs in the sequential case due to the random sequencing of updates.

V. Systems

We investigate the performance of the parallel DYNA algorithm with four classic simulation setups. Additionally, we

validate the simulation results on two real-world setups.

A. Simulation

The simulated systems are pendulum swing-up, two-link manipulator, cart-double-pole balancing and cart-pole swing-up. They are depicted schematically in Figure 4.

1) Pendulum swing-up: We use the pendulum system described in [42] (Section 4.5.3), which consists of a DC motor with a round plate attached to the outgoing axis, see Figure 5a. On the plate, a weight is fixed in a decentered location, creating a pendulum (Figure 4a). The motor is voltage controlled with $u \in \{-3, 0, 3\}$ V.

The system may be modeled as in Equation (24)

$$J\ddot{\theta} = mgl\sin(\theta) - \left(b + \frac{K^2}{R}\right)\dot{\theta} + \frac{K}{R}u \tag{24}$$

with state $s = [\theta, \dot{\theta}]^T$ and constants $J = 1.91 \cdot 10^{-4}$ kgm^2, $m = 0.055$ kg, $g = 9.81$ m/s^2, $l = 0.042$ m, $b = 3 \cdot 10^{-6}$ Nms/rad, $K = 0.0536$ Nm/A and $R = 9.5\Omega$.

The task of the system is to swing up the weight, although the motor does not have enough torque to do this in one swing. We used a discount rate $\gamma = 0.97$, reward function $r = -5\theta^2 - 0.1\dot{\theta}^2 - u$ and 3 s trials. Each trial was started at $s_0 = [\pi, 0]$. The control step time was 0.03 s.

2) Two-link manipulator: The two-link manipulator [42] (Section 4.5.2) is similar to the pendulum, but has two links and is located in the horizontal plane (Figure 4b). The motors are controlled by $\tau = [\tau_1, \tau_2]^T$, $\tau_1 \in \{-1.5, 0, 1.5\}$ Nm, $\tau_2 \in \{-1, 0, 1\}$ Nm.

This system is described by the fourth-order continuous-time nonlinear model

$$M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} = \tau \tag{25}$$

where

$$\begin{aligned} M(\theta) &= \begin{pmatrix} P_1 + P_2 + 2P_3\cos\theta_2 & P_2 + P_3\cos\theta_2 \\ P_2 + P_3\cos\theta_2 & P_2 \end{pmatrix} \\ C(\theta, \dot{\theta}) &= \begin{pmatrix} b_1 - P_3\dot{\theta}_2\sin\theta_2 & -P_3(\dot{\theta}_1 + \dot{\theta}_2)\sin\theta_2 \\ P_3\dot{\theta}_1\sin\theta_2 & b_2 \end{pmatrix} \end{aligned} \tag{26}$$

with state $s = [\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2]$ and $P_1 = m_1c_1^2 + m_2l^2 + I_1$, $P_2 = m_2c_2^2 + I_2$, and $P_3 = m_2lc_2$. The physical parameters are length of the pole $l = 0.4$ m, masses $m_1 = 1.25$ kg, $m_2 = 0.8$ kg, inertias $I_1 = 0.066$ kgm^2, $I_2 = 0.043$ kgm^2, centers of mass locations $c_1 = c_2 = 0.2$ m, and damping $b_1 = 0.08$ kg/s and $b_2 = 0.02$ kg/s.

The task here is to balance the arms in the straight position, hampered by the nonlinear dynamics of the two links affecting eachother. For this system we used a discount rate $\gamma = 0.98$, reward function $r = -5\theta_1^2 - 0.05\dot{\theta}_1^2 - 5\theta_2^2 - 0.05\dot{\theta}_2^2$ and 3 s trials. Each trial was started at a random state $s_0 = [\eta_1, 0, \eta_2, 0]$ with $\eta \in [0, 2\pi)$. The control step time was 0.05 s.

3) Cart-double-pole balancing: To get a higher-dimensional system, we place the two links on a cart, this time in the vertical plane again (Figure 4c). The cart may be pushed with a force $F \in \{-20, 0, 20\}$ N. This results in the cart-double-pole balancing problem, for which we use the formulation as in [43]:

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = Q_q \tag{27}$$

where

$$\begin{aligned} M(q) &= \begin{pmatrix} h_1 & h_2\cos\theta_1 & h_3\cos\theta_2 \\ h_2\cos\theta_1 & h_4 & h_5\cos(\theta_1 - \theta_2) \\ h_3\cos\theta_2 & h_5\cos(\theta_1 - \theta_2) & h_6 \end{pmatrix} \\ C(q, \dot{q}) &= \begin{pmatrix} 0 & -h_2\dot{\theta}_1\sin\theta_1 & -h_3\dot{\theta}_2\sin\theta_2 \\ 0 & 0 & h_5\dot{\theta}_2\sin(\theta_1 - \theta_2) \\ 0 & -h_5\dot{\theta}_1\sin(\theta_1 - \theta_2) & 0 \end{pmatrix} \\ q &= \begin{pmatrix} x \\ \theta_1 \\ \theta_2 \end{pmatrix}, g(q) = \begin{pmatrix} 0 \\ -h_7\sin\theta_1 \\ -h_8\sin\theta_2 \end{pmatrix}, Q_q = \begin{pmatrix} F \\ 0 \\ 0 \end{pmatrix} \end{aligned} \tag{28}$$

with

$$\begin{aligned} h_1 &= m_c + m_1 + m_2 & h_5 &= m_2c_2l \\ h_2 &= m_1c_1 + m_2l & h_6 &= m_2c_2^2 + I_2 \\ h_3 &= m_2c_2 & h_7 &= m_1c_1g + m_2lg \\ h_4 &= m_1c_1^2 + m_2l^2 + I_1 & h_8 &= m_2c_2g \end{aligned} \tag{29}$$

and state $s = [x, \dot{x}, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2]$ and constants $m_c = 0.5$ kg, $m_1 = m_2 = 0.5$ kg, $g = 9.82$ m/s^2 and $l = 0.6$ m. The centers of mass $c_1 = c_2 = 0.3$ m were set in the middle; the inertias of the poles are therefore $I_1 = m_1l^2/3$ and $I_2 = m_2l^2/3$.

The goal is to balance the poles in the upright position, starting from a slightly perturbed state $s_0 = [\eta_1, 0, \eta_2, 0, \eta_3, 0]$ with $\eta \in [-5, 5]$ mm or mrad. The reward is 1 for each timestep in which $[-2.4, -0.7, -0.7] < [x, \theta_1, \theta_2] < [2.4, 0.7, 0.7]$ and the episode ends with a reward of 0 otherwise. The episode also ends after 5 s of successful balancing. The control step time was 0.05 s, with a discount rate $\gamma = 0.97$.

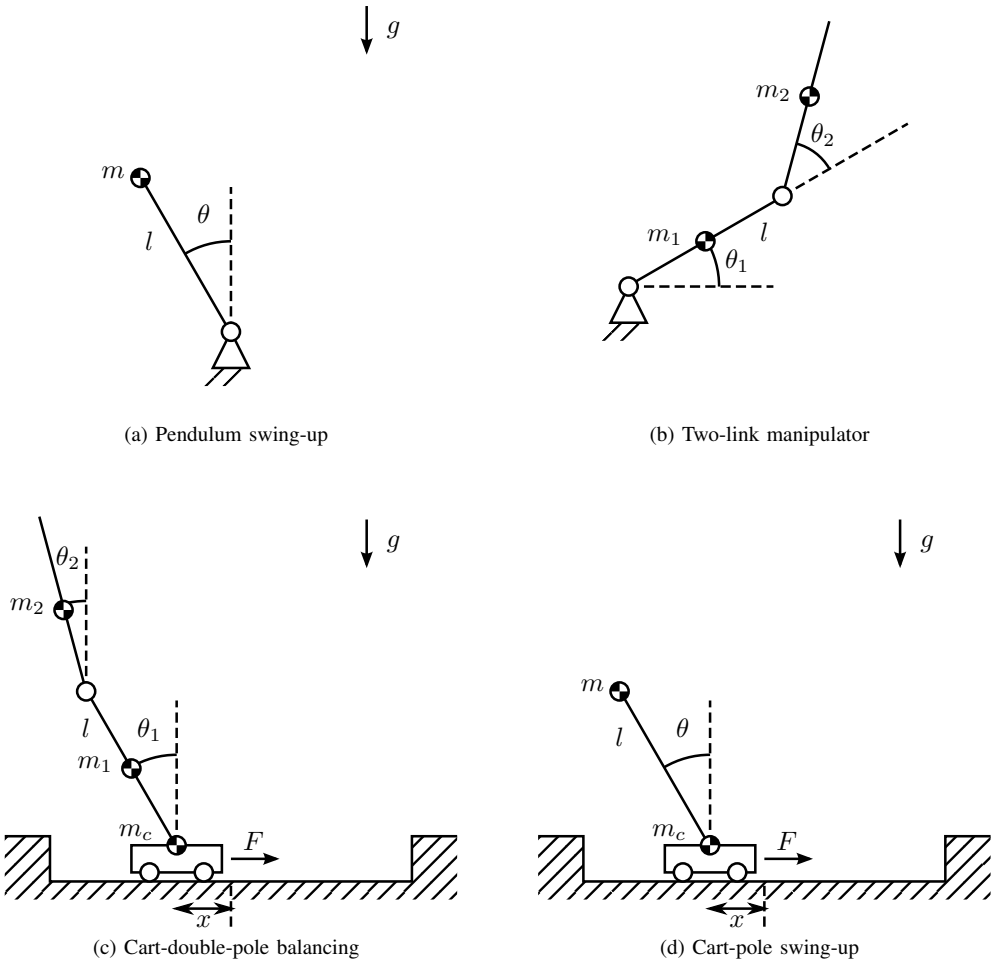
4) Cart-pole swing-up: The last simulated system is the cart-pole swing-up problem (Figure 4d). Although the system has less degrees of freedom than cart-double-pole, the problem is harder because the system is not started in the balanced position. Instead, the cart has to be moved back and forth until the balanced position is achieved, and subsequently maintained.

Our cart-pole system is that of [44], but with the pole starting in the down position and without modelling friction. The equations of motion are:

$$\begin{aligned} \ddot{\theta} &= \frac{g\sin\theta + \cos\theta\left[\frac{-F - ml\dot{\theta}^2\sin\theta}{m_c + m}\right]}{l\left[\frac{4}{3} - \frac{m\cos^2\theta}{m_c + m}\right]} \\ \ddot{x} &= \frac{F + ml\left[\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right]}{m_c + m} \end{aligned} \tag{30}$$

with state $s = [x, \dot{x}, \theta, \dot{\theta}]$ and constants $g = 9.8$ m/s^2, $m_c = 1$ kg, $m = 0.1$ kg and $l = 0.5$ m.

The applied force on the cart $F \in \{-10, 10\}$ N and the state $s = [x, \dot{x}, \theta, \dot{\theta}]$. Each trial started at a slightly perturbed state



(a) Pendulum swing-up

(b) Two-link manipulator

(c) Cart-double-pole balancing

(d) Cart-pole swing-up

Fig. 4. Sketches of the four simulated systems.

$s_0 = [0, 0, \pi + \eta, 0]$ with $\eta \in [-50, 50]$ mrad and lasted 10 s or until the position x left the interval $[-2.4, 2.4]$, in which case a reward of -10^4 was given. We used a discount rate of $\gamma = 0.97$ with reward function $r = -\theta^2 - 0.1\dot{\theta}^2 - 2x^2 - 0.1\dot{x}^2$. The control step time was 0.05 s.

B. Real-world

Our real-world setups are realizations of the pendulum swing-up and two-link manipulator systems from section V-A, shown in Figure 5.

1) Pendulum swing-up: The pendulum swing-up system (Figure 5a) is a faithful implementation of the model of Section V-A1. Because the plate is directly driven by the DC motor, there is a minimum of backlash and encoder discrepancies.

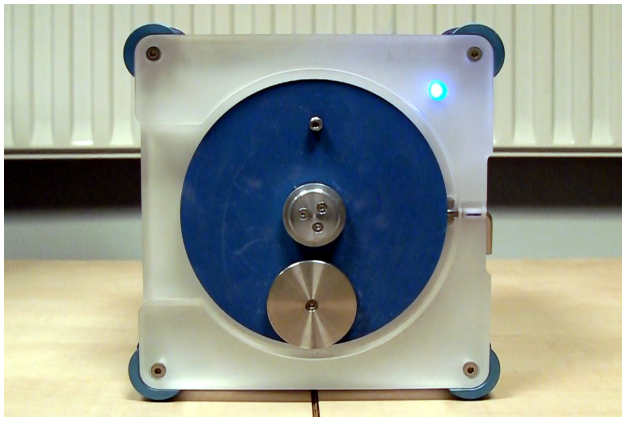
2) Two-link manipulator: The real two-link manipulator setup is slightly different than the one presented in Section V-A2, with Dynamixel RX-28 motors actuating the joints with $\tau_1, \tau_2 \in \{-25, 0, 25\}$ % of maximum torque, and the task being to move from a "pick" position at $s_0 = [-\pi/4, 0, \pi/4, 0]$ to a "place" position $p = [\pi/4, 0, -\pi/4, 0]$, see Figure 5b. The reward function was therefore centered on p instead of the zero position. Additionally, the trials were terminated when the target position was reached, and a penalty of -100 given when the workspace of $[-1.25, 1.25]$ rad was exceeded.

TABLE I
LEARNING PARAMETERS FOR THE SIMULATED TASKS. THE PARAMETERS FOR THE REAL VARIANTS OF TASKS V-A1 AND V-A2 WERE THE SAME AS IN SIMULATION.

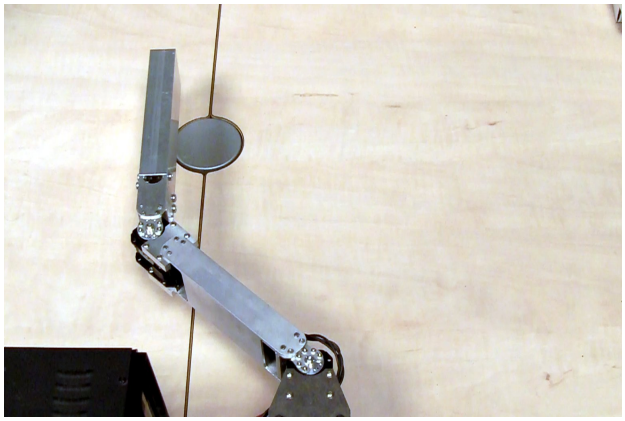
Task	γ	λ	α	ϵ	Tile widths
V-A1	0.97	0.65	0.2	0.05	$\pi/10$, π
V-A2	0.98	0.92	0.4	0.05	$\frac{\pi}{10}$, π, $\frac{\pi}{10}$, π
V-A3	0.99	0.92	0.2	0.02	2.5,2.5,$\frac{\pi}{40}$, $\frac{\pi}{4}$,$\frac{\pi}{40}$, $\frac{\pi}{4}$
V-A4	0.97	0.65	0.2	0.05	2.5,2.5,$\frac{\pi}{20}$, $\frac{\pi}{2}$

VI. EXPERIMENTS

We now present the results of our simulation experiments, comparing them with other algorithms, as well as investigating the impact of some design choices. In all cases, the learning rate for the planning threads was one tenth the learning rate of the agent thread, to account for model inaccuracies. No eligibility traces were used during planning. The experiments were performed on a dual 8-core Intel Xeon E5-2665 machine,



(a) Pendulum swing-up



(b) Two-link manipulator (horizonal plane)

Fig. 5. The two real-world setups in their respective start positions.

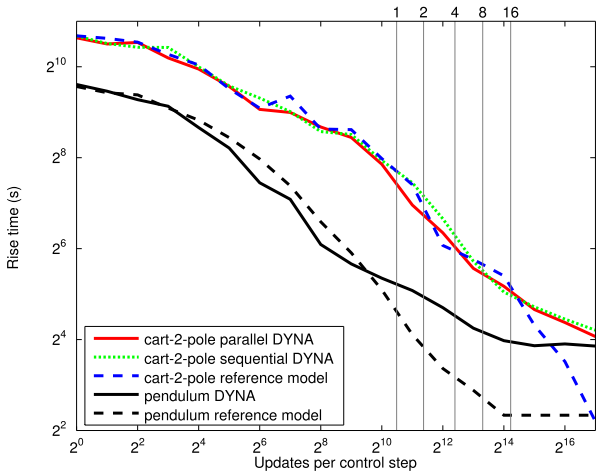


Fig. 6. Rise time versus number of updates per control step for the cart-double-pole balancing and pendulum swing-up problems. Also depicted are the number of updates per control step performed by 1, 2, 4, 8 and 16 parallel threads for the cart-double-pole system. The hard plateau in the pendulum reference model line occurs because the system learns the task in the first trial.

with the parallel algorithms using 14 threads. The learning parameters of all tasks are summarized in Table I. The basic parameters for the pendulum and two-link manipulator were derived from the originating papers, while the others were based on these and coarsely optimized for SARSA(λ) performance.

The learning trials were interrupted by regular test trials, in which exploration was set to zero. We only report the cumulative reward of these test trials. We characterize the rise time by the time required to reach a system-specific cumulative reward, measured by taking the average over all runs of the first time this value is passed for three or more consecutive test trials. The end performance was averaged over the last 10 test trials of all runs.

A. *Parallel efficiency*

We start by looking at the efficiency of the parallelization. Figure 6 plots the rise time of the cart-double-pole balancing and pendulum swing-up problems against the number of updates per control step K. K scales well with the number of threads (vertical lines), with a maximum speedup of 13.3 for 16 threads, validating the lock-free implementation.

Looking at the solid and dotted lines for the cart-double-pole balancing problem, we can see that doing the updates in parallel results in the same performance as doing them sequentially with a single thread. Updates with internally inconsistent features therefore do not measurably impact the learning.

In general, doing more updates per control step decreases the rise time, but there are diminishing returns. The learned model is only valid around the experienced data points. How well these points are generalized is system and model-dependent. When we compare the results for our learned model with those of a perfect reference model (dashed lines) we see that our model only "supports" a limited number of updates per control step, after which the performance gains taper off. Nonlinearities in the transition and reward functions reduce this value. For example, the pendulum swing-up (black lines) supports less updates per control step because it is nonlinear in the transition from start to goal state.

B. *Performance evaluation*

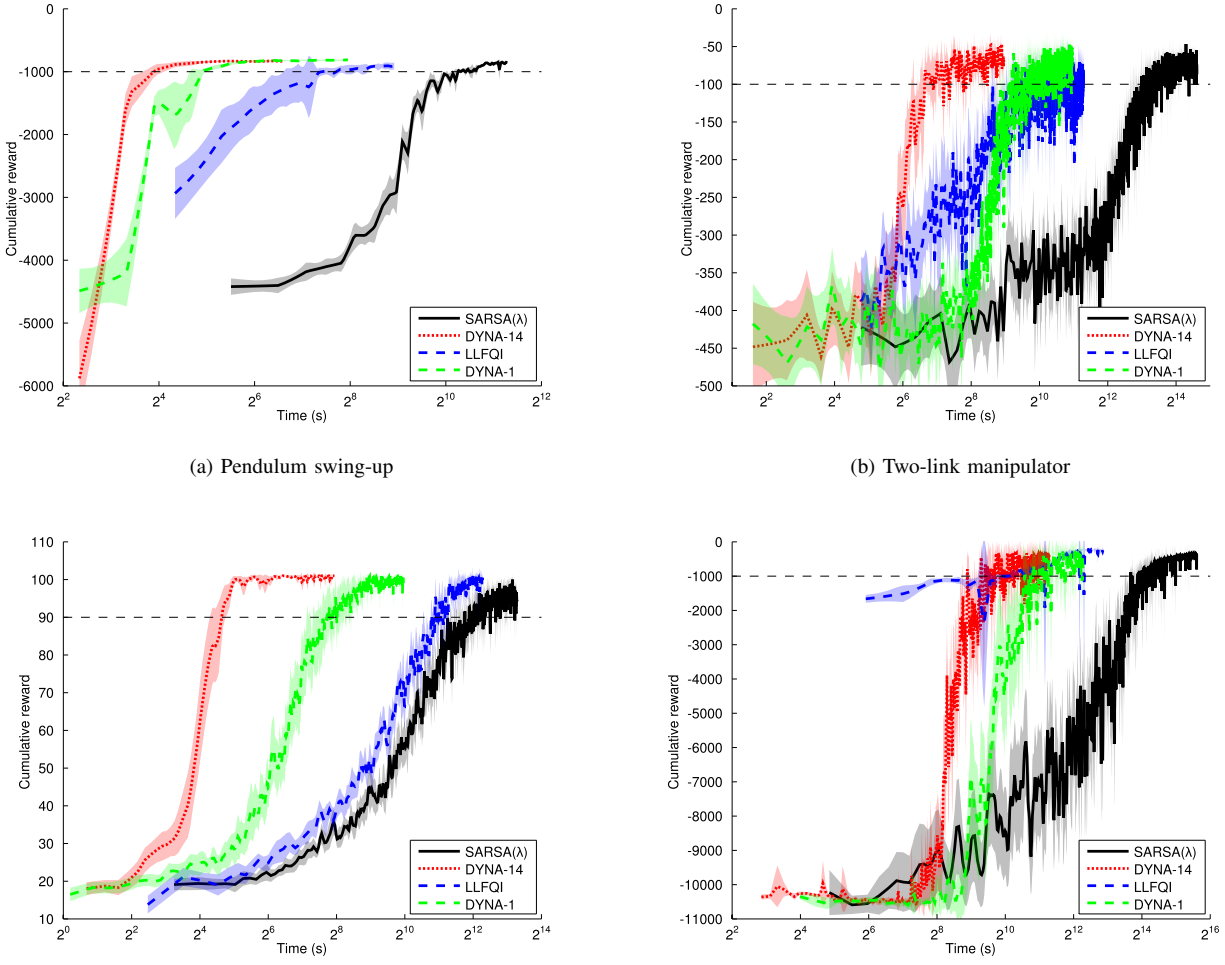
We now analyze the performance of parallel DYNA with respect to SARSA(λ) and our FQI variant, parallel LLFQI. As can be seen in Figure 7 and Table II, both parallel DYNA and parallel LLFQI are significantly faster than SARSA(λ), although the results are very system-dependent. The difference is most clear for the cart-double-pole balancing problem, where the speedup of parallel DYNA is 94x over SARSA(λ) and 69x over parallel LLFQI. Note, however, the superior end performance of parallel LLFQI for the cart-pole swing-up problem. In particular, it very quickly learns not to go off track, which is why the learning curve starts higher.

TABLE II

SIMULATION RESULTS. UNLESS OTHERWISE STATED, THE ALGORITHM IS PARALLEL DYNA WITH 14 THREADS USING LWR MODEL APPROXIMATION WITH STOCHASTIC READOUT, WITH TRAJECTORIES STARTING AT AN EXPERIENCED START STATE AND WITHOUT REFERENCE REWARDS. THE NAMING INDICATES WHICH PART OF THIS DEFAULT SETUP CHANGED. GIVEN ARE THE MEANS AND 95% CONFIDENCE INTERVALS OVER 25 RUNS. THE VALUES THAT ARE WITHIN THE 95% CONFIDENCE INTERVAL OF THE BEST VALUE FOR THAT SYSTEM ARE HIGHLIGHTED.

	pendulum swing-up		two-link manipulator		cart-2-pole balancing		cart-pole swing-up	
	rise (s)	end perf	rise (s)	end perf	rise (s)	end perf	rise (s)	end perf
SARSA(λ)	1201 (106)	-870 (20)	4242 (439)	-77.7 (4.6)	2424 (227)	95.0 (1.5)	11697 (765)	-437 (92)
Parallel DYNA (14 threads)	14.9 (2.0)	-832 (6.6)	82.6 (8.3)	**-65.7** (5.6)	25.8 (4.1)	100.6 (0.3)	342 (25)	-521 (143)
Parallel locally linear FQI	156 (33)	-926 (37)	325 (79)	-99.8 (12)	1780 (177)	99.3 (1.1)	1151 (216)	**-257** (26)
Single threaded DYNA	34.7 (2.5)	-817 (10)	429 (17)	-70.2 (8.8)	182 (35)	99.7 (0.6)	837 (84)	-635 (253)
SOGP model	70.7 (7.6)	-818 (7.4)			706 (137)	96.9 (1.4)		
Deterministic LWR model	**14.7** (1.4)	-830 (7.3)	83.4 (6.7)	**-64.7** (5.3)	19.6 (2.0)	100.7 (0.3)		
Current start states	16.2 (5.9)	-868 (21)	77.8 (17)	-72.4 (4.9)	22.8 (2.1)	**101** (0.04)		
Random start states	**14.3** (2.2)	-831 (11)	84.3 (9.7)	**-63.4** (6.0)	27.1 (4.0)	100.5 (0.4)		
Known reward function	17.4 (2.3)	**-805** (3.9)	**65.8** (8.8)	-72.2 (6.9)	24.9 (3.1)	**101** (0.07)	**23.0** (2.1)	-298 (4.8)
PILCO*	**9.9** (5.0)	-832 (50)			**16.5** (1.5)	100.5 (0.7)	37.0 (14)	-418 (32)

*Averaged over 10 successful runs.



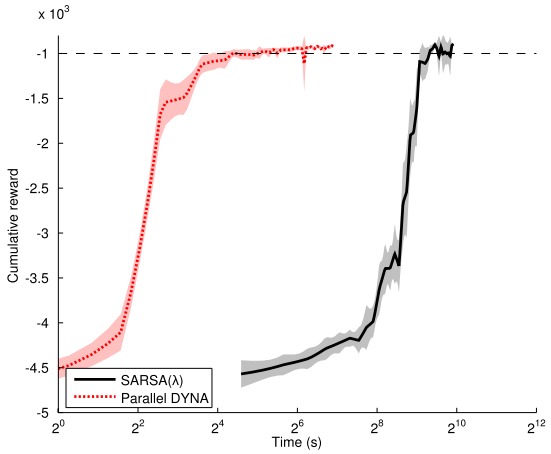
(a) Pendulum swing-up

(b) Two-link manipulator

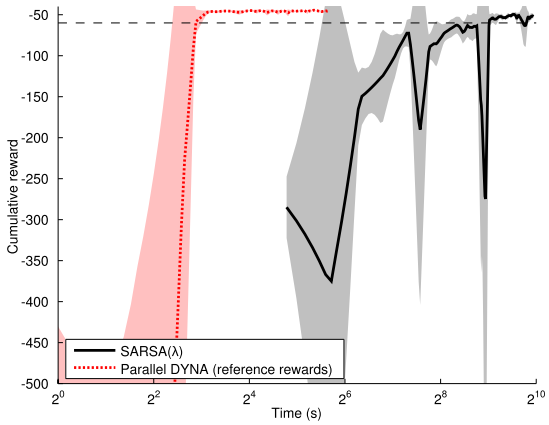
(c) Cart-double-pole balancing

(d) Cart-pole swing-up

Fig. 7. Performance evaluation of parallel DYNA compared to SARSA(λ), parallel LLFQI and single-threaded DYNA. Shown are the mean and 95% confidence interval over 25 independent runs. The horizontal line is the point at which the rise time is measured. For all systems, parallel DYNA is significantly faster (note the logarithmic time axis). The end performance is higher as well, except for the cart-pole swing-up problem.



(a) Pendulum swing-up. Shown are the mean and 95% confidence interval over 25 independent runs.



(b) Two-link manipulator. Shown are the mean and 95% confidence interval over 12 (DYNA) or 6 (SARSA(λ)) independent runs.

Fig. 8. Comparison of SARSA(λ) and parallel DYNA on the real setups.

The performance of parallel DYNA on the pendulum is also significantly faster than that reported for various model-learning actor-critic variants in [12] (14.9s vs ∼5-10min), and than online LSPI on the two-link manipulator as reported in [42] (82.6s vs ∼30min). The speedup over single-threaded DYNA is less dramatic (2x-7x), but still significant. Note that a single thread of our algorithm performs ∼1500 updates per control step, much more than the ∼10 commonly used in literature.

C. Model choice

We have chosen to use locally weighted linear regression with stochastic readout as the model approximator for parallel DYNA. Comparing it to sparse online gaussian processes and LWR with deterministic readout (second set of results in Table II), we can see that LWR does indeed perform better than SOGP. One of the reasons for the worse rise time of SOGP is the slower readout compared to LWR. While with LWR we can achieve up to $5.0 \cdot 10^5$ model updates/s, with SOGP this is only $3.4 \cdot 10^3$.

We could not find good SOGP parameter settings for the two-link manipulator problem. On our machine, the number of basis vectors that could be handled in real-time was ∼700, but using more vectors or different widths also did not prove sufficient. We also refrained from testing the cart-pole swing-up problem due to prohibitively long learning times.

The performance of deterministic LWR readout (taking only the mean) is very similar to the stochastic version. As all the investigated systems are deterministic, a perfect model does not require stochasticity, but modelling the model uncertainty is also not essential in this case. In this regard, an advantage of our approach over model-based policy search is that the temporal correlation [32] between predictions is less important, as each SARSA(0) planning update is based on just a single one-step prediction, instead of the whole trajectory.

D. Choice of start state

We opted to start each planning episode in a random experienced start state of the system. This most closely resembles the actual agent, but might not be optimal. We therefore compared starting from experienced start states to starting from the current agent state (similar to TD search [45]) and from random experienced points.

In general, the results (third set in Table II) are not significantly different, although starting from the current state has a lower end performance for the pendulum swing-up problem. As, after some initial learning, the agent spends most of its time balancing the pendulum, starting from the current state means that the planner doesn't try to decrease the swing-up time. However, it is faster for the cart-double-pole balancing problem, perhaps because it is spending more effort in states where the pole is already slightly out of balance.

E. Reference rewards

Finally, we compare the performance of learning a full system model (state transitions, rewards and termination criterion) to only learning the system dynamics. In many cases the reward function is known beforehand, and it is unproductive not to use this information. We therefore use the actual reward function in planning.

The result is significantly faster learning for the two-link manipulator and cart-pole swing-up problems, and better end performance for the pendulum swing-up problem. In particular, with reference rewards, the cart-pole swing-up problem can be solved in only 23s.

Using reference rewards allows us to make a meaningful comparison to the state-of-the-art model learning policy search algorithm PILCO [35], which requires known, differentiable reward functions. PILCO is a good reference, because it outperforms many other current algorithms by at least an order of magnitude.

Being a policy search algorithm, it is easy to record and use the *best* policy instead of the final one. The PILCO rise

time was therefore calculated as the first time the cumulative reward (averaged over 10 initial positions) of a generated policy passed the set reward, instead of three consecutive such policies. Additionally, the end performance is the maximum over the entire run instead of the average over the last trials.

The learning parameters were set to the closest matching example in the PILCO software[3] and optimized from there. In particular, we used the original reward functions for learning, while we report the end performance on the rewards from Section V. Although both optimize towards the same goal state, the results can be slightly different.

The rise times (last set in Table II) are competitive, with a difference of less than a factor of two either way on the tested systems. PILCO was faster on the pendulum swing-up and cart-double-pole balancing tasks, but slower on the cart-pole swing-up. However, the PILCO software was unable optimize the two-link manipulator task due to the broad starting state distribution.

Note that the PILCO results are given for the *successful* runs only. The success rate was 70% for the pendulum, 100% for the cart-double pole balancing, and 60% for the cart-pole swing-up. Furthermore, the rise time is the required *experimentation time*, but PILCO typically requires hours of additional computation whereas parallel DYNA runs in real-time. It would be interesting to see how close PILCO can get to real-time operation when using a parallel implementation.

VII. Validation

We now validate our simulation results with real-world variants of the pendulum swing-up and two-link manipulator systems. The resulting learning curves are plotted in Figure 8. In both cases, parallel DYNA is significantly faster than SARSA(λ), the difference being 20x for the pendulum and 60x for the manipulator. Note that it was impossible to do more than 6 runs for SARSA(λ) on the manipulator, as excessive jitter caused too much damage to the gearboxes. This could be alleviated by limiting the bandwidth of the action signal [46].

Especially on the two-link manipulator, there is significant play in the joints as well as noise on the encoder values. The fact that parallel DYNA still performs well on this problem shows that it is robust against these effects, which are common in robotic systems.

VIII. Conclusions

We have presented parallel DYNA, an on-line temporal difference learning method suited for motor control. Using locally weighted linear regression to estimate a model of the system dynamics, it runs many *planning threads* in parallel to exploit this information. We have shown that parallel DYNA works well on simulated control tasks, achieving almost two orders of magnitude speedup over the standard on-line algorithm, SARSA(λ), and taking the same amount of experimentation time as the state-of-the-art policy search method PILCO but without the additional computation time. We have validated the simulation results on two real-world robotic setups, where similar speedups were reached.

[3]http://mlg.eng.cam.ac.uk/pilco/release/pilcoV0.9.zip

As future work we would like to scale up to more degrees of freedom, and to investigate systems with discontinuities in the transition function, such as impacts.

Acknowledgment

The authors would like to thank Martijn Wisse and Pieter Jonker for fruitful discussions, as well as Robert Babuška and the Delft Center for Systems and Control for the use of the real-world setups described in this paper.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.

[2] E. Schuitema, M. Wisse, and P. Jonker, "The design of 'LEO': a 2d bipedal walking robot for online autonomous reinforcement learning," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Syst.*, Taipei, Taiwan, October 2010, pp. 3238–3243.

[3] J. Kober and J. Peters, "Reinforcement learning in robotics: a survey," in *Reinforcement Learning: State of the Art*, M. Wiering and M. van Otterlo, Eds. Berlin, Germany: Springer Verlag, 2012, pp. 579–610.

[4] F. Stulp and O. Sigaud, "Robot skill learning: From reinforcement learning to evolution strategies," *Paladyn*, vol. 4, no. 1, pp. 49–61, 2013.

[5] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[6] K. Olukotun and L. Hammond, "The future of microprocessors," *ACM Queue*, vol. 3, no. 7, pp. 26–29, 2005.

[7] J. Morimoto and K. Doya, "Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning," *Robot. Auton. Syst.*, vol. 36, no. 1, pp. 37–51, 2001.

[8] T. Martínez-Marín and T. Duckett, "Fast reinforcement learning for vision-guided mobile robots," in *Proc. IEEE Int. Conf. Robotics and Automation*, Barcelona, Spain, April 2005, pp. 4170–4175.

[9] H. Benbrahim and J. Franklin, "Biped dynamic walking using reinforcement learning," *Robot. Auton. Syst.*, vol. 22, no. 3, pp. 283–302, 1997.

[10] W. Smart and L. Pack Kaelbling, "Effective reinforcement learning for mobile robots," in *Proc. IEEE Int. Conf. Robotics and Automation*, Washington, DC, May 2002, pp. 3404–3410.

[11] I. Grondman, L. Busoniu, G. A. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Trans. Syst. Man Cybern. C, Appl. Rev.*, vol. 42, no. 6, pp. 1291–1307, November 2012.

[12] I. Grondman, M. Vaandrager, L. Busoniu, R. Babuska, and E. Schuitema, "Efficient model learning methods for actor-critic control," *IEEE Trans. Syst. Man Cybern. B, Cybern.*, vol. 42, no. 3, pp. 591–602, June 2012.

[13] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Proc. Int. Conf. Machine Learning*, Austin, TX, June 1990, pp. 216–224.

[14] R. Hafner and M. Riedmiller, "Neural reinforcement learning controllers for a real robot application," in *Proc. IEEE Int. Conf. Robotics and Automation*, Roma, Italy, April 2007, pp. 2098–2103.

[15] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Auton. Robot.*, vol. 27, no. 1, pp. 55–73, 2009.

[16] T. Hester and P. Stone, "Learning and using models," in *Reinforcement Learning: State of the Art*, M. Wiering and M. van Otterlo, Eds. Berlin, Germany: Springer Verlag, 2012, pp. 111–141.

[17] T. Lampe and M. Riedmiller, "Approximate model-assisted neural fitted q-iteration," in *Proc. Int. Joint Conf. Neural Networks*, Beijing, China, July 2014, pp. 2698–2704.

[18] B. Bakker, V. Zhumatiy, G. Gruener, and J. Schmidhuber, "Quasi-online reinforcement learning for robots," in *Proc. IEEE Int. Conf. Robotics and Automation*, Orlando, FL, May 2006, pp. 2997–3002.

[19] T. Hester and P. Stone, "TEXPLORE: real-time sample-efficient reinforcement learning for robots," *Mach. Learn.*, vol. 90, no. 3, pp. 385–429, 2013.

[20] R. M. Kretchmar, "Reinforcement learning algorithms for homogenous multi-agent systems," in *Proc. Workshop Agent and Swarm Programming*, Cleveland, OH, October 2003, pp. 1–10.

[21] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," in *Proc. Int. Conf. Autonomous Agents and Multiagent Syst.*, Taipei, Taiwan, May 2011, pp. 761–768.

[22] T. Tateyama, S. Kawata, and Y. Shimomura, "Parallel reinforcement learning systems using exploration agents and Dyna-Q algorithm," in *SICE Annu. Conf.*, Takamatsu, Japan, September 2007, pp. 2774–2778.
[23] V. Palmer, "Scaling reinforcement learning to the unconstrained multi-agent domain," Ph.D. dissertation, Texas A&M University, 2007.
[24] Y. Li and D. Schuurmans, "Mapreduce for parallel reinforcement learning," in *Proc. European Workshop Reinforcement Learning*, Athens, Greece, September 2011, pp. 309–320.
[25] F. S. Melo, S. P. Meyn, and M. I. Ribeiro, "An analysis of reinforcement learning with function approximation," in *Proc. Int. Conf. Machine Learning*, Helsinki, Finland, July 2008, pp. 664–671.
[26] J. S. Albus, "A new approach to manipulator control: the cerebellar model articulation controller (CMAC)," *J. Dyn. Sys., Meas., Control*, vol. 97, no. 3, pp. 220–227, 1975.
[27] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less real time," *Mach. Learn.*, vol. 13, no. 1, pp. 103–130, 1993.
[28] R. Sutton, C. Szepesvári, A. Geramifard, and M. Bowling, "Dyna-style planning with linear function approximation and prioritized sweeping," in *Proc. Conf. Uncertainty in Artificial Intell.*, Helsinki, Finland, July 2008, pp. 1–9.
[29] A. L. Strehl and M. L. Littman, "An empirical evaluation of interval estimation for markov decision processes," in *Proc. IEEE Int. Conf. Tools with Artificial Intell.*, Boca Raton, FL, November 2004, pp. 128–135.
[30] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning," *Artif. Intell. Rev.*, vol. 11, no. 1, pp. 11–73, 1997.
[31] ——, "Locally weighted learning for control," *Artif. Intell. Rev.*, vol. 11, no. 1, pp. 75–113, 1997.
[32] J. A. Bagnell and J. Schneider, "Autonomous helicopter control using reinforcement learning policy search methods," in *Proc. IEEE Int. Conf. Robotics and Automation*, Seoul, Korea, May 2001, pp. 1615–1620.
[33] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *J. ACM*, vol. 45, no. 6, pp. 891–923, 1998.
[34] L. Csató and M. Opper, "Sparse on-line gaussian processes," *Neural Comput.*, vol. 14, no. 3, pp. 641–668, 2002.
[35] M. Deisenroth and C. Rasmussen, "PILCO: A model-based and data-efficient approach to policy search," in *Proc. Int. Conf. Machine Learning*, Bellevue, WA, June/July 2011, pp. 465–472.
[36] C. Atkeson and J. Santamaria, "A comparison of direct and model-based reinforcement learning," in *Proc. IEEE Int. Conf. Robotics and Automation*, Albuquerque, NM, April 1997, pp. 3557–3564.
[37] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *J. Mach. Learn. Res.*, vol. 6, pp. 503–556, April 2005.
[38] M. Riedmiller, "Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method," in *Proc. European Conf. Machine Learning*, Porto, Portugal, October 2005, pp. 317–328.
[39] D. Ormoneit and S. Sen, "Kernel-based reinforcement learning," *Mach. Learn.*, vol. 49, no. 2–3, pp. 161–178, 2002.
[40] S. Adam, L. Busoniu, and R. Babuska, "Experience replay for real-time reinforcement learning control," *IEEE Trans. Syst. Man Cybern. C, Appl. Rev.*, vol. 42, no. 2, pp. 201–212, March 2012.
[41] H. van Seijen, H. van Hasselt, S. Whiteson, and M. Wiering, "A theoretical and empirical analysis of expected sarsa," in *Proc. IEEE Symp. Adaptive Dynamic Programming and Reinforcement Learning*, Nashville, TN, March/April 2009, pp. 177–184.
[42] L. Busoniu, R. Babuska, B. de Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximation*. Boca Raton, FL: CRC Press, 2010.
[43] W. Zhong and H. Rock, "Energy and passivity based control of the double inverted pendulum on a cart," in *Proc. IEEE Int. Conf. Control Applications*, Mexico City, Mexico, September 2001, pp. 896–901.
[44] A. Barto, R. Sutton, and C. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst. Man Cybern.*, vol. 13, no. 5, pp. 834–846, September/October 1983.
[45] D. Silver, R. Sutton, and M. Müller, "Temporal-difference search in computer go," *Mach. Learn.*, vol. 87, no. 2, pp. 183–219, 2012.
[46] H. Meijdam, M. Plooij, and W. Caarls, "Learning while preventing mechanical failure due to random motions," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Syst.*, Tokyo, Japan, November 2013, pp. 182–187.

Wouter Caarls received his M.Sc. degree (with honors) in artificial intelligence from the University of Amsterdam, Amsterdam, The Netherlands. He obtained a Ph.D. from the Delft University of Technology, Delft, The Netherlands on the subject of the automatic optimization of a parallel computer architecture for smart cameras. He is currently a visiting researcher at the Postgraduate Program in Informatics, Federal University of Rio de Janeiro, Brazil, investigating the applications of reinforcement learning in robotics and computer networks. His research interests include robotics, machine learning, optimization, parallel algorithms, and image processing. Dr. Caarls is a member of the IEEE.

Erik Schuitema received the M.Sc. degree (with honors) in applied physics from the Delft University of Technology, Delft, The Netherlands, in 2006. He obtained the Ph.D. degree in the Department of Biomechanical Engineering, Delft University of Technology, on real-time reinforcement learning techniques for the control of a real bipedal walking robot. His research interests include intelligent hardware and software for robotics, machine learning, autonomous agents, and software engineering.