Uncertainty-driven Imagination for Continuous Deep Reinforcement Learning

Gabriel Kalweit
University of Freiburg, Germany
kalweitg@cs.uni-freiburg.de

Joschka Boedecker
University of Freiburg, Germany
jboedeck@cs.uni-freiburg.de

Abstract: Continuous control of high-dimensional systems can be achieved by current state-of-the-art reinforcement learning methods such as the *Deep Deterministic Policy Gradient* algorithm, but needs a significant amount of data samples. For real-world systems, this can be an obstacle since excessive data collection can be expensive, tedious or lead to physical damage. The main incentive of this work is to keep the advantages of model-free Q-learning while minimizing real-world interaction by the employment of a dynamics model learned in parallel. To counteract adverse effects of imaginary rollouts with an inaccurate model, a notion of uncertainty is introduced, to make use of artificial data only in cases of *high* uncertainty. We evaluate our approach on three simulated robot tasks and achieve faster learning by at least 40 per cent in comparison to vanilla DDPG with multiple updates.

Keywords: Reinforcement Learning, Data Augmentation, Uncertainty Estimation

1 Introduction

Deep reinforcement learning had recent successes in a variety of applications, such as super-human performance at playing Atari games from pixels [1], in the game of Go [2], or for robot control [3]. The adaptation of *Deep Q-Networks* to an actor-critic approach addressed the problem of continuous action domains [4], which is an important step towards solving many real-world problems. Yet the high sample complexity of the *Deep Deterministic Policy Gradient* (DDPG) limits its real-world applicability. When applying reinforcement learning approaches on a real robot, the interaction and reset of environment and agent can be expensive, tedious or lead to physical damage. Therefore, collecting millions of transitions is not feasible in the real world. Even though it is possible to alleviate this problem by massive parallelism [5], enhancing data-efficiency for deep reinforcement learning is desirable and beneficial for both parallel and sequential setups to achieve increasingly complex tasks.

One way to achieve better data efficiency in terms of minimal interactions with the system is to augment the collected data. In a reinforcement learning setting, this can be done by artificial transitions, e.g. through a learned dynamics model, under the assumption that the model can be learned faster than the corresponding value-function of a task. Models that can easily (i.e. models of low capacity or complexity) be learned, however, might lack in expressiveness and thus in accuracy. Therefore, an agent has to balance the use of artificial and real data during learning. Here, we introduce a method to reduce system interaction through synthetic data when appropriate to achieve this balance. To counteract adverse effects of inaccurate artificial data, the uncertainty of the agent is measured and incorporated to limit training on the generated data set. This paper brings three contributions to current research. Firstly, we show that a massive increase of updates per step does lead to stability issues in DDPG. Secondly, we extend the replay memory of DDPG by artificial transitions from a neural model and show that this leads to a much smaller demand for real-world transitions. Thirdly, we extend the critic to a bootstrapped neural network, so as to limit articifial data usage for high uncertainty. Together, this leads to the *Model-assisted Bootstrapped DDPG* algorithm, which we evaluate on three simulated robot learning tasks.

1st Conference on Robot Learning (CoRL 2017), Mountain View, United States.

2 Related Work

All approaches incorporating off-policy experience generated by a learned model can be seen as some instantiation of *Dyna-Q* [6]. Lampe and Riedmiller [7] extended *Neural-fitted Q-Iteration* (NFQ) [8] by a neural model and achieved significantly faster learning in terms of required interactions on the *Cartpole Swing up* environment. However, NFQ is a full-batch approach which is not feasible for experience buffers containing millions of samples. The authors suggest the limitation of model-assisted rollouts an open question. In addition, *Approximate Model-Assisted NFQ* is limited to discrete action domains. Gu et al. [9] investigated the impact of *imagination rollouts* within *Normalized Advantage Functions* (NAF). In their results, they present performances of NAF and DDPG with 5 updates per step and compare these to NAF with model-acceleration and $5 \cdot l$ updates per step, l being the rollout length. Furthermore, they were not able to apply neural models successfully and used locally linear models. Whereas locally linear models are very efficient to learn, these models lack in expressiveness and do not generalize as well as neural models. Lastly, the model had to be switched off after a certain amount of episodes, since further training on artificial samples led to performance decrease. This is in contrast to our approach which avoids manual tuning of the time point to switch off model usage, and rather determines the ratio of artificial and real data based on uncertainty of the Q-function. Also related to our work is *Dyna-2* [10], an online-approach from the Dyna-family learning two Q-function approximations based on permanent and transient memories. As in the other approaches, the use of artificial data is not limited, as opposed to our proposal detailed below.

Recently, Popov et al. [11] successfully applied DDPG on a simulated robot environment by incorporating task knowledge in the reward function and by increasing the amount of updates per step. However, the knowledge the system can extract from collected transitions is limited. Hence, this also holds for multiple update steps. In our experiments, increasing the amount of updates to a certain point brought no additional performance or even had negative impact. Lastly, Weber et al. [12] used a predictive model in *Imagination-Augmented Agents* to provide additional context to a policy network. To address the issue of erroneous predictions, the imagined trajectories are processed by an interpretation module.

3 Background

In the reinforcement learning setup, tasks are modelled as MDPs, where the agent is in a state s_t from a continuous, $d_{\mathcal{S}}$-dimensional state space $\mathcal{S}$ and executes action a_t from a continuous, $d_{\mathcal{A}}$-dimensional action space $\mathcal{A}$ to interact with environment $\mathcal{E}$ in each discrete time step t. Applying a_t in s_t, the agent gets into a next state s_{t+1} and receives a reward r_t. Such a transition is described by the dynamics model $\mathcal{M} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{P}[\mathcal{S}]$ of the environment. It maps states and actions to a set of probability measures $\mathcal{P}[\mathcal{S}]$ over the state space, defining the transition dynamics $\rho_{\mathcal{M}}[s_{t+1}|s_t, a_t]$. When the agent chooses its actions, it follows some policy $\pi : \mathcal{S} \mapsto \mathcal{P}[\mathcal{A}]$ which is a mapping from states to probability measures over the action space.

In each time step t, the agent receives a reward r_t which is given by the immediate reward function $r : \mathcal{S} \times \mathcal{A} \longmapsto \mathbb{R}$. The overall reward for starting in state s_t and following a fixed trajectory thereafter is described by the return $R(s_t) = r(s_t, a_t) + \sum_{j=t+1}^{T-1} \gamma^{(j-t)} r(s_j, a_j)$, where $\gamma \in [0, 1]$ is a discounting factor. The expected return for a state s_t when following a policy π is then the expectation over all possible executed actions and successor states, $R^{\pi}(s_t) = \mathbf{E}_{s_{j>t} \sim \mathcal{M}, a_{j \geq t} \sim \pi}[R(s_t)]$. The goal is to find an optimal policy π^*, such that $R^{\pi^*}(s_0) \geq R^{\pi}(s_0)$ for all policies π and start state s_0. The start states $\mathcal{S}_0 \subset \mathcal{S}$ might also follow some probability distribution with underlying $\rho_0[s_0]$. This adds to the definition of $R^{\pi}(\mathcal{S}_0) = \mathbf{E}_{s_0 \sim \rho_0, s_{j>0} \sim \mathcal{M}, a_{j \geq 0} \sim \pi}[R(s_0)]$. π^* therefore corresponds to a policy maximizing the expected return of R^{π} for start state density $\rho_0[s_0]$,

$$\pi^* = \arg\max_{\pi} R^{\pi}(\mathcal{S}_0). \tag{1}$$

Model-free and Model-based Reinforcement Learning. In order to find a solution for Equation (1) directly, the dynamics model of the environment is usually required. However, if the model is not available, there are several *model-free* methods to circumvent this necessity. One way is to learn a *Q-function*, which assigns a value to state-action pairs, describing the expected

value of taking action a_t in state s_t and following a policy π afterwards, i.e. $Q^\pi(s_t, a_t) = \mathbf{E}_{s_{j>t}\sim\mathcal{M}, a_{j>t}\sim\pi}[R(s_t)|a_t]$. The Bellman Equation provides a recursive definition of the Q-function, $Q^\pi(s_t, a_t) = \mathbf{E}_{s_{t+1}\sim\mathcal{M}}[r(s_t, a_t) + \gamma \mathbf{E}_{a_{t+1}\sim\pi}[Q^\pi(s_{t+1}, a_{t+1})]]$. For very large or infinite state or action spaces, the Q-function is usually estimated by some function-approximator $Q^\pi(\cdot, \cdot|\theta^Q)$ with parameters θ^Q. We optimize these parameters by minimizing a loss term based on the squared difference between target value $y_t^Q = r(s_t, a_t) + \gamma \max_a Q^\pi(s_{t+1}, a|\theta^Q)$ and the current Q^π-estimate, i.e. we minimize:

$$\mathcal{L}(\theta^Q) = (y_t^Q - Q^\pi(s_t, a_t|\theta^Q))^2. \tag{2}$$

Given the gradients of the loss w.r.t the Q-function parameters, $\nabla_{\theta^Q}\mathcal{L}(\theta^Q)$, the Q-function approximator can then be updated by gradient descent. Usually, an update is done for a batch of samples in one step, therefore taking the mean loss.

A different way to find a solution for Equation (1) without a model are *Policy Gradient Methods* [13]. Instead of approximating a value-function and deriving an implicit policy from it, the policy itself is represented by some parameterized approximator. The parameters are updated directly according to the gradient referring to its performance, the *policy gradient*. In its basic form, the policy is stochastic. However, gradients for deterministic policies have also been derived [14]. This approach can be embedded in actor-critic or policy iteration algorithms and naturally fits continuous action domains.

When the environment dynamics can be approximated, the task can also be solved with *model-based* reinforcement learning and optimal control on the learned model. However, extracting an optimal policy, given the transition and reward function of an environment, is in general a non-trivial task and might require some very expensive computations at runtime. Prominent planners used in this area are the *iterative Linear Quadratic Regulator* [15] or *Monte-Carlo Tree Search* [16, 17]. Both had recent applications; the former e.g. in [18, 19, 3, 9] and the latter in *AlphaGo* [2]. The advantages of model-free reinforcement learning are its general applicability and the cheap evaluation after learning. On the other hand, model-based reinforcement learning methods often require (orders of magnitude) less samples.

Deep Deterministic Policy Gradient. This paper builds upon the DDPG algorithm, an actor-critic approach that was explicitly designed to handle continuous action domains. Since the policy in DDPG, subsequently μ, is deterministic, it is a direct mapping from states to actions, $\mu : \mathcal{S} \mapsto \mathcal{A}$. It represents the currently best known policy, i.e. $\mu(s_t) = \arg\max_{a_t} Q(s_t, a_t)$. The actor μ, as well as the critic Q, are estimated by corresponding approximators $\mu(\cdot|\theta^\mu)$ and $Q(\cdot, \cdot|\theta^Q)$, parameterized by θ^μ and θ^Q respectively. Following the insights of Deep Q-Networks, the target values for training are calculated using slowly updated target Q- and policy networks, denoted by $Q'(\cdot, \cdot|\theta^{Q'})$ and $\mu'(\cdot|\theta^{\mu'})$. This stabilizes deep Q-learning significantly. For each update step, a minibatch of n samples is generated randomly. Firstly, the target values y_i are computed using target Q- and policy networks, $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.

Then the mean squared error is given by $\mathcal{L}(\theta^Q) = \frac{1}{n}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$ and the policy is updated according to the mean of all samples, as stated in the *Deterministic Policy Gradient Theorem* [14]:

$$\nabla_{\theta^\mu} R^\mu \longleftarrow \frac{1}{n}\sum_i \nabla_a Q(s_i, a|\theta^Q)|_{a=\mu(s_i|\theta^\mu)} \nabla_{\theta^\mu}\mu(s_i|\theta^\mu). \tag{3}$$

The parameters $\theta^{Q'}$ and $\theta^{\mu'}$ of the target networks are slowly moved towards the parameters of their associates in each update step, $\theta^{Q'} \longleftarrow (1-\tau)\theta^{Q'} + \tau\theta^Q$ and $\theta^{\mu'} \longleftarrow (1-\tau)\theta^{\mu'} + \tau\theta^\mu$, with $\tau \in (0, 1]$.

4 Model-assisted Bootstrapped DDPG

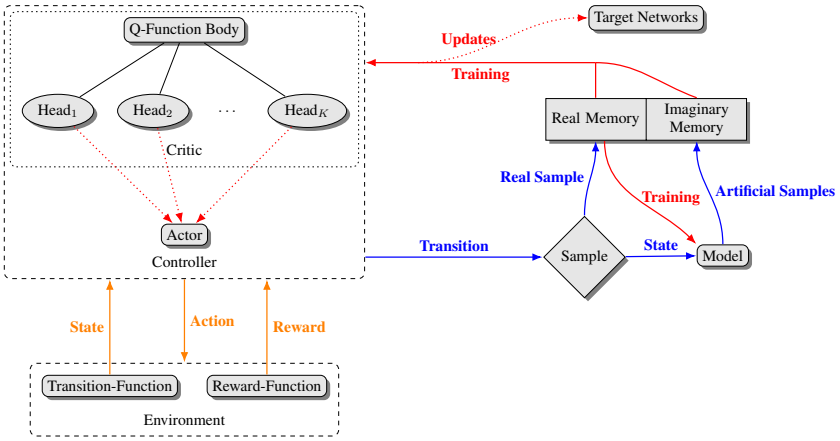

Figure 1: Structure of MA-BDDPG. Interaction with the environment (yellow), internal data processing (blue) and training (red). The K-headed controller collects samples in the real world and measures its variance. From a collected state, artificial rollouts are generated and saved in the imaginary memory. In each cycle, the controller is trained on both memories proportional to its variance.

The main incentive of this work is to keep the generality and optimality of model-free Q-learning while minimizing real-world interaction with a model to enhance data efficiency. The employment of a learned model bears the risk of generating incorrect transitions and thus impairing value function approximation. To counteract these adverse effects, a notion of uncertainty is introduced, to make use of artificial data only when appropriate, i.e. for samples of *high* uncertainty. These ideas result in the *Model-assisted Bootstrapped DDPG* algorithm, abbreviated with MA-BDDPG (see Figure 1 for an overview of the approach). To arrive at the full algorithm, let us first describe the combination of DDPG with a model, giving a variant which we call model-assisted DDPG (MA-DDPG). Following that, we show how to make use of bootstrapping to include a measure of uncertainty in the critic, and how to use it to get an automatic limitation of virtual rollout (subsequently called *imagination*) samples.

Model-assisted DDPG. We approach the problem with a deterministic model and learn to predict the state change: $\mathcal{M}^{\Delta} : \mathcal{S} \times \mathcal{A} \mapsto \Delta_{\mathcal{S}}$. In each time step, the agent collects a real sample s_t, which is the starting point for B imaginary rollouts of specified depth D. For each step, the action a_{bd} is selected based on the current real policy and random noise. For training, target values, loss and target update are equivalent to DDPG. Since actor and critic are updated on imaginary samples, for those updates it comes down to

$$\hat{\mathcal{L}}(\theta^Q) = \frac{1}{n} \sum_i (\hat{y}_i - Q(\hat{s}_i, a_i | \theta^Q))^2, \tag{4}$$

where target values $\hat{y}_i$ are defined as $\hat{y}_i = r(\hat{s}_i, a_i) + \gamma Q'(\hat{s}_{i+1}, \mu'(\hat{s}_{i+1}|\theta^{\mu'})|\theta^{Q'})$ and the occuring states $\hat{s}_i = \mathcal{M}^{\Delta}(\hat{s}_{i-1}, a_{i-1}|\theta^{\mathcal{M}}) + \hat{s}_{i-1}$, with model $\mathcal{M}^{\Delta}$ parameterized by $\theta^{\mathcal{M}}$. The critic is updated on $U_{\mathcal{R}}$ real batches and $U_{\mathcal{I}}$ imaginary batches and thus gets gradients from both loss functions (both of the form of Equation (4), but with real states s rather than synthetic states $\hat{s}$) in each cycle. For an overview and a detailed description of MA-DDPG, see the appendix.

Including Uncertainty. The quantification of uncertainty in the prediction of a neural network is a question of active research. Bayesian Neural Networks [20, 21, 22, 23] put a probability distribution over the network weights and thereby give a measure of uncertainty. As Gal and Ghahramani [24] observe, these models suffer from tremendous computational cost. They suggest to address the problem of model uncertainty by using dropout to approximate a Gaussian Process posterior. While this is a scalable and simple way to measure the uncertainty in predictions, Osband et al. [25] found this approach (and its modification to heteroscedastic model uncertainty) to bring unsatisfactory results. To overcome these issues, they apply bootstrapping [26] to *Deep Q-Networks* to get a distribution over Q-functions and therefore an estimate of the uncertainty. In order to incorporate

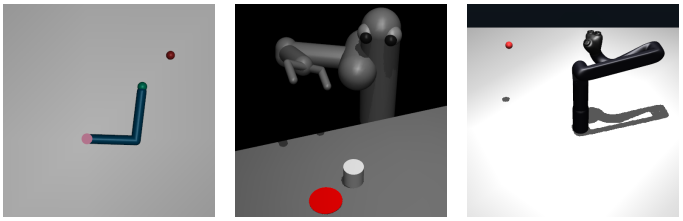

Figure 2: Visualization of the environments. (left) Reacher, (middle) Pusher and (right) Jaco.

the uncertainty estimate and to still be able to adapt to continuous action domains, the *Bootstrapped DQN* architecture is extended to follow the actor-critic scheme of DDPG described above.

The Q-function approximator $Q_{1:K}(\cdot,\cdot|\theta^Q)$ is defined to be a deep neural network with parameters θ^Q composed of an optional single shared body and K independent network heads Q_k. While this corresponds to [25], in our experiments we did not use a shared body. Analogously to DDPG and MA-DDPG, the target values get computed via slowly updated target networks $Q'_{1:K}$ and μ'. When the agent receives a new transition, it generates a random mask $\mathbf{m}_t = (m^1, m^2, \ldots, m^K)_t$ with $m^i \sim \mathbf{P}$, for some probability distribution $\mathbf{P}$. In the experiments, each element of the mask is drawn according to a Bernoulli distribution with sharing probability p, analogously to [25]. This ensures that a sample is only seen by some subset of the heads and, hence, K bootstrapped samples are created. The K heads are then trained on their respective samples. The exact procedure is shown in Algorithm 1. First, the different target values for the K heads are calculated, $y_t^{Q_{1:K}} \leftarrow r_t + \gamma Q'_{1:K}(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'_{1:K}})$, and then the critic gets updated on the combined loss,

$$\mathcal{L}(\theta^{Q_{1:K}}) = \sum_k \frac{1}{n_k} \sum_i m_i^k (y_i^k - Q^k(s_i, a_i|\theta^{Q_k}))^2, \tag{5}$$

where $n_k = \sum_i \delta_{m_i^k \neq 0}$ and δ is the Dirac delta function. Note that the masks modify the loss, so as to redirect the gradients only to the corresponding heads. The parameters θ^μ of single-network actor μ are then updated according to the mean of all respective updates of all heads,

$$\nabla_{\theta^\mu} R \leftarrow \frac{1}{K} \sum_k \frac{1}{n} \sum_i \nabla_a Q^k(s_i, a|\theta^{Q^k})|_{a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s_i|\theta^\mu). \tag{6}$$

After updating critic and actor, the weights of the target networks $Q'_{1:K}$ and μ' get shifted towards the current weights by $\tau \in (0, 1]$, analogously to DDPG. The model-based loss and updates are defined as in the previous section. Based on the distribution over Q-functions, the agent can then estimate its uncertainty by its variance.

Limiting Model Usage. A simple way to limit the use of imagination proportional to current uncertainty is to transform the uncertainty measure to a probability distribution and to induce a biased coin flip. The result of the coin flip then determines whether to train on an imaginary batch or not. The variance of an agent could be task specific. On the other hand, the measure should not depend on prior knowledge over the variance. In order to circumvent this necessity, we put the uncertainty into context of the maximum variance seen so far. Formally, we define the probability of learning on minibatch b from the imaginary replay buffer $\mathcal{I}$ as $\mathbf{P}[b \subset \mathcal{I}] = \hat{\sigma}^2_{e+1} = \frac{\bar{\sigma}^2_e}{\max_E \bar{\sigma}^2_{E<e}}$ in episode e, where $\bar{\sigma}^2_e$ denotes the mean variance in e. This restricts the mean variance ratio to $[0, 1]$ and can be interpreted as follows: if it is unknown whether a value of variance is high or not, the maximum value is a rational reference point. Hence, this underlying probability distribution yields a dimensionless measure of uncertainty and should decrease over time when the different heads of the critic converge. However, this leads to very noisy data generation in the very beginning and limiting model usage by prediction error gives an important direction for future work.

5 Experimental Results

Environments. We evaluated our approach on three robot environments implemented in *MuJoCo* [27]: Reacher, Pusher and Jaco. For the Reacher and Pusher tasks, we used the environments from

Algorithm 1: MA-BDDPG

initialize critic Q and actor μ, targets Q' and μ' and model $\mathcal{M}^{\Delta}$
initialize replay buffer $\mathcal{R}$ and imaginary replay buffer $\mathcal{I}$
$\hat{\sigma}_1^2 = 0$
for $e = 1..E$ **do**
  get initial state s_1
  for $t = 1..T$ **do**
    apply action $a_t = \mu(s_t|\theta^{\mu}) + \xi$ and observe variance $\sigma_t^2(Q)$
    observe s_{t+1} and r_t
    generate random mask $\mathbf{m}_t$
    save transition $(s_t, a_t, s_{t+1}, r_t, \mathbf{m}_t)$ in $\mathcal{R}$
    generate B random imaginary transitions of length D starting from s_t using $\mathcal{M}^{\Delta}$
    store the imaginary transitions in $\mathcal{I}$
    for $u_{\mathcal{M}} = 1..U_{\mathcal{M}}$ **do**
      train $\mathcal{M}^{\Delta}$ on minibatch from $\mathcal{R}$
    for $u_{\mathcal{R}} = 1..U_{\mathcal{R}}$ **do**
      train Q and μ on minibatch from $\mathcal{R}$
      adjust parameters of Q' and μ'
    for $u_{\mathcal{I}} = 1..U_{\mathcal{I}}$ **do**
      generate random number $rand \in [0, 1]$
      if $rand < \hat{\sigma}_e^2$ **then**
        train Q and μ on minibatch from $\mathcal{I}$
        adjust parameters of Q' and μ'
  add variance values in e and set maximum variance seen so far
  set current variance ratio $\hat{\sigma}_{e+1}^2 = \frac{\bar{\sigma}_e^2}{\max_{E} \bar{\sigma}_{E<e}^2}$, where $\bar{\sigma}_e^2$ denotes the mean variance in e

OpenAI Gym [28]. A visualization can be seen in Figure 2 and a detailed state description can be found in the appendix. We tested the approaches on 100 randomly generated initial states per run.

Reacher In this environment, a two-link arm has to reach a given random target in 2D-space. The continuous state-space has 11 dimensions, the continuous action-space has two.

Pusher The goal in this environment is to push a randomly placed object to a fixed goal position using an arm with seven degrees of freedom. The continuous state-space has 23 dimensions, the continuous action-space has seven.

Jaco This environment extends the Reacher task to a 3D random-target reaching task for the Kinova[1] Jaco arm. The continuous state-space has 24 dimensions (excluding the fingers), the continuous action-space has six.

Parameters. To enhance comparability, we kept all shared parameters the same across all approaches, due to the fact that MA-BDDPG was built upon DDPG. Since we experienced stability issues otherwise in both MA-BDDPG and DDPG, we had to lower the learning rates when using multiple updates. For the Reacher and Jaco tasks, we set the default amount of $U_{\mathcal{R}}$ to 20 and the amount of $U_{\mathcal{I}}$ to 80. In case of the Pusher task, these values were set to 5 and 95. Nevertheless, we also compare our results to DDPG with 100 updates, as well as to exemplary runs of the original DDPG hyperparameter setting with one and 20, respectively five, updates per cycle. DDPG with default settings is denoted by DDPG*. Numbers in parentheses denote differing update values, otherwise the same amount of real updates as in MA-BDDPG was used. A more detailed description of the setting is given in the appendix.

Results. The results of the three environments can be seen in Figure 3 and a video can be found at https://youtu.be/f_QC9cj33Ew. Within the given time frames, MA-BDDPG shows significantly faster learning across all tasks. The dynamics of the Reacher environment is the easiest among these three, hence model learning is faster. We believe that this is the reason for MA-DDPG to gain

[1] http://www.kinovarobotics.com/

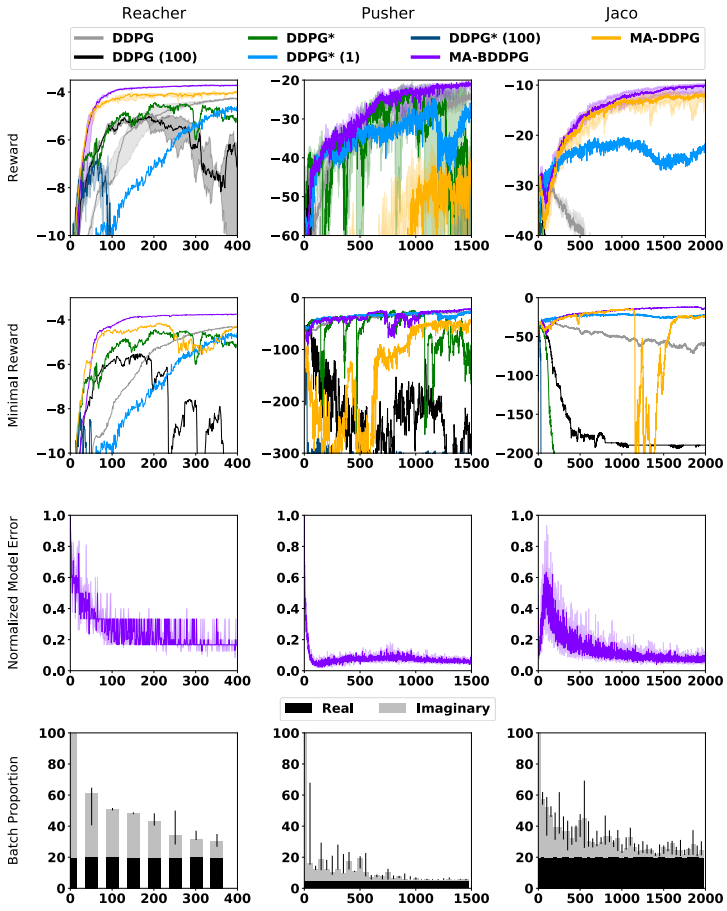

Figure 3: Results of environments. Top row shows median performance and interquartile ranges. The second row shows the worst occuring reward values across all runs. The third row shows the error in prediction over time, normalized to [0, 1]. The last row shows the ratio of real and imaginary batches used by MA-BDDPG. Numbers in parentheses denote differing update values.

better performance in the very beginning, but the accuracy of the model seems to slow learning down after around 100 episodes. At that point, MA-BDDPG already decreased its uncertainty and therefore its use of imagination, so as to focus on the real data. Note the adversary effects of 100 updates in both DDPG and DDPG*, as well as 20 updates in DDPG*, as presented in Table 1. There does not seem to be enough additional information per cycle, such that it can benefit from this amount of updates. However, model-based imagination is no panacea and the use of too much artificial information can lead to tremendous stability issues, as can be seen in the results of MA-DDPG in the Pusher and Jaco experiments. Limiting the use of artificial rollouts leads to significantly more stable performances in all three tasks and achieves robustness against this hyperparameter. This translates to much faster successes, which is listed in Table 2. In regard to the three given tasks, MA-BDDPG saves 92, 44 and 93 per cent training time in comparison to DDPG with the same hyperparameter setting. This translates to speed ups of factor 12.35, 1.77 and 15. Additionally, MA-BDDPG saves 92, 69 and 91 per cent of episodes until reaching a successful performance in comparison to DDPG* (1), which corresponds to speed ups of factor 12.35, 3.22 and 11.13. We increased the amount of updates stepwise in the range of [1, 100] and were not able to find a better hyperparameter setting than the one used, for neither DDPG nor DDPG*.

We noticed a performance decrease and stability issues when using too large imaginary replay buffers. Hence, even if long-term samples and negative examples are needed, learning does be-

	MA-BDDPG	DDPG (100)	DDPG* (100)	DDPG*
Reacher	**243 (-3.74)**	188 (-5.02)	64 (-6.866)	203 (-4.432)
Pusher	**1072 (-20.893)**	33 (-47.118)	0 (-58.378)	957 (-22.014)
Jaco	**1999 (-9.53)**	18 (-32.262)	0 (-28.21)	17 (-32.248)

Table 1: Best results returned by DDPG and DDPG* with 100 updates per step, as well as best results from DDPG* and equal real updates as MA-BDDPG.

	MA-BDDPG	DDPG	speed up $\times$	DDPG* (1)	speed up $\times$
Reacher	**243 (-3.74)**	3000 (-3.848)	12.35	3000 (-3.885)	12.35
Pusher	**1072 (-20.893)**	1902 (-20.579)	1.77	3455 (-20.941)	3.22
Jaco	**1999 (-9.53)**	30000 (-11.816)	15.00	22245 (-9.986)	11.13

Table 2: Median episodes until success of all runs. Next to episodes are rewards. In case there was no success within the considered time frame, the amount was set to the maximal episode, together with the best reward found after that time. Reacher was solved if there was a reward > -3.75, as defined in OpenAI Gym. For Pusher we defined success to get a reward > -21 and for Jaco > -10.

come faster and more stable using imaginary replay buffers that only cover imagination for the last few episodes. We believe that this is due to the differences in state prediction of the model after too many updates. The samples in the replay buffer might not agree and thus have an adverse effect on learning. Setting the size of the imaginary replay buffer so it only covers the last few episodes behaves like implicit prioritized experience replay [29] in the later course of training. It enhances recent successful trajectories and thus enables faster learning. On the other hand, uniform sampling of real transitions still keeps track of negative experience.

There is a big difference in speed up between the Jaco and Pusher tasks. The model error in Figure 3 yields a possible explanation and we believe this is due to several reasons. Firstly, the Pusher task only acts within a very narrow part of the configuration space, whereas the Jaco task covers a much larger part. Secondly, the Pusher task is a two-tier task. The robot has to move towards the obstacle and then push it to the goal. In the beginning, the robot does not move the object, but tries to move towards it. In regard to model learning, only samples are collected in which the object does not move. Hence, when the agent does move the object, the model first has to relearn this part of the environment's dynamics which translates to an increase of model error and a very slow decrease afterwards. This makes the task more difficult in regard to model learning and fast model adaptation is one direction for future work. Lastly, since the agent moves in a very narrow part of the configuration space, the variance gets low rather quickly. Even after the agent relearnt the dynamics of the moving object, imagination is sparingly used and thus has almost no further impact. We experienced much faster learning with a larger rollout length in the very beginning. However, the approach became unstable later on, so we had to reduce it. With a faster adapting model or a flexible rollout length it seems that the benefit could be a lot larger.

6 Conclusion

In this paper, we introduced the *Model-assisted Bootstrapped DDPG* algorithm. It augments the transitions collected in the real world by artificial data generated by a neural model learned in parallel. This transition augmentation is limited by the decreasing uncertainty of the Q-function. We showed that the presented approach leads to significantly faster learning while maintaining stability. Reaching a satisfying performance was at least 40 per cent faster than that of vanilla DDPG with multiple updates.

Going forward, instead of setting a fixed rollout-length, the augmentation depth and model usage should be limited by model-error. Fast model adaptation would be of great use in terms of applicability on environments of higher complexity. Furthermore, the data generation per cycle can be a bottleneck for real-world applications. It could be useful to either move data generation to the end of an episode or to parallelize data generation and execution. The effect of mixed minibatches, instead of distinct real and imaginary, also gives an interesting direction for future work.

Acknowledgments

We would like to thank Jost Tobias Springenberg for the insightful discussions throughout this work and Manuel Watter for his detailed comments and suggestions. This research was supported by the German Research Foundation (DFG, grant number EXC 1086).

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[3] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971.

[5] S. Gu, E. Holly, T. P. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. pages 3389–3396, 2017.

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

[7] T. Lampe and M. Riedmiller. Approximate model-assisted neural fitted q-iteration. In *Neural Networks (IJCNN), 2014 International Joint Conference on*, pages 2698–2704. IEEE, 2014.

[8] M. Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In *Lecture Notes in Computer Science: Proc. of the European Conference on Machine Learning, ECML 2005*, pages 317–328, Porto, Portugal, 2005.

[9] S. Gu, T. P. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. pages 2829–2838, 2016.

[10] D. Silver, R. S. Sutton, and M. Müller. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th international conference on Machine learning*, pages 968–975. ACM, 2008.

[11] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller. Data-efficient deep reinforcement learning for dexterous manipulation. *arXiv preprint arXiv:1704.03073*, 2017.

[12] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*, 2017.

[13] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[14] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

[15] W. Li and E. Todorov. Iterative linear quadratic regulator design for nonlinear biological movement systems. In *ICINCO (1)*, pages 222–229, 2004.

[16] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, pages 72–83. Springer, 2006.

[17] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[18] J. Boedecker, J. T. Springenberg, J. Wülfing, and M. Riedmiller. Approximate real-time optimal control based on sparse gaussian process models. In *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, 2014.

[19] M. Watter, J. Springenberg, J. Boedecker, and M. Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. In *Advances in Neural Information Processing Systems*, pages 2746–2754, 2015.

[20] R. M. Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

[21] D. J. MacKay. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

[22] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

[23] J. M. Hernández-Lobato and R. P. Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. 37:1861–1869, 2015. URL http://jmlr.org/proceedings/papers/v37/hernandez-lobatoc15.html.

[24] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. 48:1050–1059, 2016. URL http://jmlr.org/proceedings/papers/v48/gal16.html.

[25] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy. Deep exploration via bootstrapped dqn. In *Advances In Neural Information Processing Systems*, pages 4026–4034, 2016.

[26] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

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

[28] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[29] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

A MA-DDPG

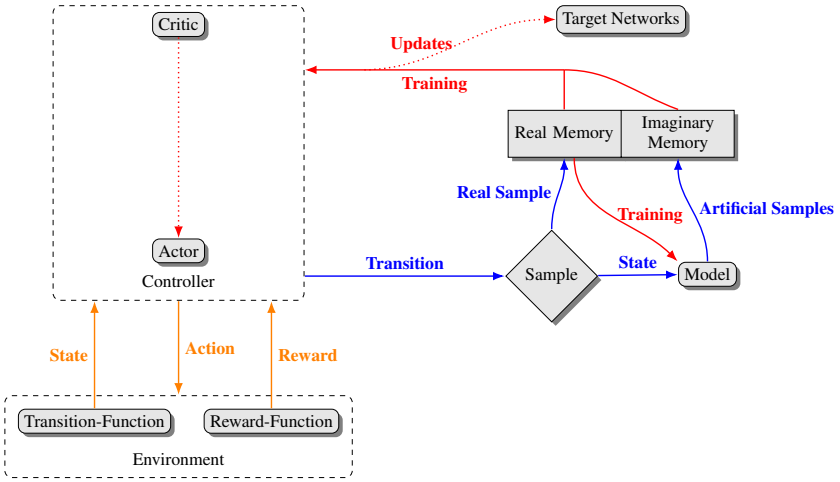

Figure 4: Structure of MA-DDPG. Interaction with the environment (yellow), internal data processing (blue) and training (red).

Algorithm 2: MA-DDPG

initialize critic Q and actor μ, targets Q' and μ' and model $\mathcal{M}^{\Delta}$
initialize replay buffer $\mathcal{R}$ and imaginary replay buffer $\mathcal{I}$
for $e = 1..E$ **do**
  get initial state s_1
  for $t = 1..T$ **do**
    apply action $a_t = \mu(s_t|\theta^{\mu}) + \xi$
    observe s_{t+1} and r_t
    store transition (s_t, a_t, s_{t+1}, r_t) in $\mathcal{R}$
    generate B random imaginary transitions of length D starting from s_t using $\mathcal{M}^{\Delta}$
    store the imaginary transitions in $\mathcal{I}$
    for $u_{\mathcal{M}} = 1..U_{\mathcal{M}}$ **do**
      train $\mathcal{M}^{\Delta}$ on minibatch from $\mathcal{R}$
    for $u_{\mathcal{R}} = 1..U_{\mathcal{R}}$ **do**
      train Q and μ on minibatch from $\mathcal{R}$
      adjust parameters of Q' and μ'
    for $u_{\mathcal{I}} = 1..U_{\mathcal{I}}$ **do**
      train Q and μ on minibatch from $\mathcal{I}$
      adjust parameters of Q' and μ'

B State Descriptions

Reacher The state includes sine and cosine of the angles, as well as the goal position, the velocity of the end effector and the distance between end effector and goal.

Pusher The state space includes the arm's configuration, velocities, the position of the end effector, the position of the target and the position of the goal.

Jaco The state again includes sine and consine of the joint's angles, as well as their velocities, the position of the goal and the distance between end effector and goal.

C Parameters

The critic had two hidden layers of 400 ReLUs, whilst the policy had two hidden layers of 300 ReLUs. The output of the critic was linear and the output layer of the policy used tanh-activation in order to scale the output to $[-1, 1]$. All hidden layers were randomly initialized from $[-0.1, 0.1]$ and

the output layers from $[-0.003, 0.003]$. In terms of optimization, Adam was used with a learning rate of 10^{-4} for the critic and 10^{-5} for the policy. γ was set to 0.99, the target values got updated with $\tau = 10^{-4}$ and the minibatch size was set to 64. The real replay buffer had a size of 10^6, the imaginary replay buffer had a size of 10^5 for Reacher and Jaco and $4 \cdot 10^5$ for Pusher. For Reacher and Jaco, we used 20 updates on real samples and 80 on imaginary samples. In case of the Pusher environment, we used five real and 95 imaginary updates. The augmentation breadth was set to 10 and the augmentation depth was set to 50 for Reacher and Jaco. For the Pusher task, we had to reduce the depth to 20, but increased the breadth to 50. MA-BDDPG used four heads in the critic with a sharing probability of 75 per cent. The model had two hidden layers with 200 leaky ReLUs for Reacher and Jaco and got updated five times, whereas for Pusher it had 400 leaky ReLUs and got updated once. The model got optimized with Adam. The learning rate was set to 10^{-3} for Reacher and to 10^{-4} for Jaco and Pusher. The batch size was 1024 for Reacher and Jaco and 4096 for Pusher. We divided the velocities of the task's states by a constant value following $\{$Reacher: 10, Pusher: 100, Jaco: $100\}$, because we experienced stability issues with MuJoCo otherwise.