Training a deep policy gradient-based neural network with asynchronous learners on a simulated robotic problem

Winfried Lötzsch[1], Julien Vitay[1] and Fred H. Hamker[1]

Abstract: Recent advances in deep reinforcement learning methods have attracted a lot of attention, because of their ability to use raw signals such as video streams as inputs, instead of pre-processed state variables. However, the most popular methods (value-based methods, e.g. deep Q-networks) focus on discrete action spaces (e.g. the left/right buttons), while realistic robotic applications usually require a continuous action space (for example the joint space). Policy gradient methods, such as stochastic policy gradient or deep deterministic policy gradient, propose to overcome this problem by allowing continuous action spaces. Despite their promises, they suffer from long training times as they need huge numbers of interactions to converge. In this paper, we investigate in how far a recent asynchronously parallel actor-critic approach, initially proposed to speed up discrete RL algorithms, could be used for the continuous control of robotic arms. We demonstrate the capabilities of this end-to-end learning algorithm on a simulated 2 degrees-of-freedom robotic arm and discuss its applications to more realistic scenarios.

Keywords: deep learning; reinforcement learning; robotics; policy gradient methods; actor critic; asynchronous learning; continuous action space

1 Introduction

Reinforcement learning (RL, [SB98]) is an important learning method used since decades in many control problems, including robotics, to map states into actions, in order to maximize the amount of reward received on the long-term. This state-action mapping can even be performed through function approximators such as neural networks. However, deep neural networks suffer from highly temporally correlated training samples and non-stationary objective functions, which are inherent to RL. This prevented the use of complex multidimensional state spaces such as raw images in deep RL and limited its applicability to tasks where the sensors could be efficiently pre-processed and reduced to a limited number of states. [Mn15] proposed a solution to this problem by introducing an *experience replay memory*, where episodes are stored and randomly fed in mini-batches to the neural network, as well as the use of *target networks* to increase the stationarity of the objective function. This *Deep Q Network* (DQN) was applied on a series of Atari 2600 games and managed to learn successfully efficient strategies with pixels as inputs and discrete actions as policy.

[1] Technische Universität Chemnitz, Professur für Künstliche Intelligenz, Str. der Nationen 62, 09111 Chemnitz, Deutschland. {winfried.loetzsch,julien.vitay,fred.hamker}@informatik.tu-chemnitz.de

doi:10.18420/in2017_214

However, DQN can only be applied to problems with discrete action spaces (DAC), such as pressing left or right buttons, or initiating complex motor primitives. For control problems requiring continuous action spaces (CAS, for example the joint space of robotic arms), discretizing the action space does not work well, as a fine-grained discretization of outputs would require an excessive amount of exploration [Zh15]. Alternatively, deep networks can be used as a submodule of the complete system: [Le16] for example used a deep network that predicts the probabilities of success of grasping attempts. This probability is then used by a separate control algorithm to generate the optimal action.

End-to-end deep RL approaches called *policy gradient* methods have recently received a lot of attention since they allow to directly learn continuous policies from high dimensional state spaces. One major issue being ensuring a sufficient level of exploration during learning, these methods either focus on learning stochastic policies [He15], which are by definition able to explore different actions during learning, or on learning deterministic policies, but exploring using a separate behavior (e.g. deep deterministic policy gradient - DDPG [Si14, Li15]). Although policy gradient methods have been successfully applied to the control of robotic arms, they require huge amounts of training data to converge (*sample complexity*). A simple but expensive solution is to use multiple robots exploring in parallel and sharing asynchronously their experiences in a common pool, which is then used to train a single neural network [Le16, Gu17]. Multiplying robots reduces the acquisition time, but does not impact the learning properties.

In the discrete domain, [Mn16] introduced the idea of using multiple parallel learners sharing their acquired knowledge, not just experiences, with each other. Their asynchronous advantage actor-critic (A3C) algorithm has been shown to be both superior in performance and in training time to the classical DQN algorithms on a variety of tasks. In this paper, we combine the idea of multiple parallel learners of [Mn16] with the DDPG algorithm of [Li15] to form a new asynchronously parallel continuous control learning algorithm. Additionally, the sample complexity of the algorithm is further reduced by pre-training an internal model of the effector. We apply it to a simple reaching task with a simulated robotic arm with 2 degrees-of-freedom, using raw pixels as inputs and joint angles as continuous outputs, and discuss its applicability to more complex problems.

2 Background and related work

We start by explaining the general problem of RL and setting the notations in section 2.1. We present deterministic policy gradient algorithms in section 2.2, as they allow to solve RL problems with continuous action spaces. Finally, we present asynchronous methods in section 2.3 which allow to train multiple actor-critic learners in parallel and reduce the overall training time.

2.1 Basics of RL

Reinforcement learning problems are modeled as *Markov Decision Processes* (MDP), with a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition dynamics model with density $p(s_{t+1}|s_t, a_t)$ and a reward function $r(s_t, a_t) : \mathcal{S} \times \mathcal{A} \rightarrow \Re$. The policy is defined as a mapping of the state space into the action space: a stochastic policy $\pi_\theta : \mathcal{S} \rightarrow P(\mathcal{A})$ defines the probability distribution $P(\mathcal{A})$ of performing an action, while a deterministic policy $\mu_\theta(s_t)$ is a discrete mapping of $\mathcal{S} \rightarrow \mathcal{A}$. $\theta \in \Re^n$ is a vector of parameters defining the policy, typically the weights of a neural network when using function approximators.

The policy can be used to explore the environment and generate trajectories of states, rewards and actions. The performance of a policy is determined by calculating the *expected discounted return*, i.e. the sum of all rewards received from time step t onwards: $R_t = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1}$, where $0 < \gamma < 1$ is the discount rate and r_{t+1} represents the reward obtained during the transition from s_t to s_{t+1}. The Q-value of an action a is defined as the expected discounted reward if the agent takes a from a state s and follows the policy distribution π_θ thereafter:

$$Q^{\pi_\theta}(s, a) = \mathbb{E}_{\pi_\theta}(R_t | s_t = s, a_t = a) \tag{1}$$

The goal of the agent is to find the optimal policy maximizing the expected return from every state. *Value-based* methods (such as DQN) achieve that goal by estimating the Q-value of each state-action pair. Discrete algorithms transform these Q-values into a stochastic policy by sampling from a Gibbs distribution (softmax) to obtain the probability of choosing an action. The Q-values can be approximated by a deep neural network, by minimizing the quadratic error between the predicted Q-value $Q^{\pi_\theta}(s, a)$ and an estimation of the real expected return R_t after that action. Alternatively, *policy gradient* methods directly learn to produce the policy (stochastic or not). The goal of the neural network is then to maximize the objective function $J(\theta) = \mathbb{E}_{\pi_\theta}(R_t)$, i.e. to maximize directly the expected discounted return by finding the optimal policy.

2.2 Continuous action spaces

The *stochastic policy gradient theorem* [Su99] provides an estimate of the gradient that should be given to the neural network in policy gradient methods, but it depends on the unknown Q-value $Q^{\pi_\theta}(s, a)$. $Q^{\pi_\theta}(s, a)$ could be approximated by the actual return R_t after that action, leading to the REINFORCE learning rule [Wi92]. Another option is to use a second neural network to learn to approximate the Q-value. Such algorithms are called *actor-critic* architectures, as the actor learns to produce a policy π_θ based on the state alone, while the critic learns to evaluate the Q-value of an action and sends this value to the actor to improve the policy.

The deterministic policy gradient (DPG) method proposed by [Si14] is an example of such an actor-critic architecture. The main novelty is that the actor produces a deterministic policy $\mu_\theta(s_t)$, what reduces the sample complexity and improves the training time. Exploration is ensured by using an *off-policy* strategy, i.e. the actions actually taken are chosen by a behavioral stochastic policy, different from the learned policy. [Si14] derived the *deterministic policy gradient theorem* allowing the deterministic actor to learn in this framework. [Li15] were able to use deep networks with the DPG architecture. The resulting deep deterministic policy gradient (DDPG) algorithm is model-free, as it simply follows the gradient of the Q-values, and thus can be applied independently from the systems dynamics. DDPG has been successfully applied to a huge variety of continuous control problems, including learning from raw pixels, and beats state-of-the-art performance on many of them.

2.3 Asynchronous methods

A rather practical problem originates from how the sample data is presented during the learning phase. The straightforward way would be to immediately process the sample data as it arrives from the simulated environment. However, robotics environments induce strong correlations between samples that are temporally close to each other, since a robot might behave very similarly in close situations. This means that the variance of the trained estimator will be very high, negatively affecting the training performance of the deep network. A standard method to reduce sample correlation is experience replay, where one basically stores each sample in a buffer for a while and then randomly samples from the buffer to increase the variability of the training data, at the cost of slowing down the learning process. This is the approach chosen by [Li15] for the DDPG algorithm.

Another way to effectively overcome the problem of data correlation is to execute multiple instances of the environment in parallel, by simultaneously running several simulations or even using many similar real robots [Le16, Gu17]. [Mn16] has shown that this idea is not only able to replace classical experience replay, but even outperforms its benefits. The diversity of the sample data produced by the threads executed in parallel can be further increased by chosing different exploration rates and starting conditions for the threads. Each thread independently uses an online learning method to compute updates of the network weights, which have to be synchronized during training. By weighting the updates with a relatively small learning rate and accumulating some updates before synchronization, one can reduce the risk to override changes from other threads and thus use a lock-free algorithm such as Hogwild! [Ni11]. The resulting algorithm, asynchronous advantage actor-critic (A3C), can replicate or even improve state-of-the-art performance on many different experiments while learning much faster.

3 Methods

We first present the simulated environment in section 3.1, followed by the architecture of the network (section 3.2), the learning procedure (section 3.3), a summary of the algorithm (section 3.4) and the reward function used for the simulated environment (3.5).

3.1 Simulated robotic arm

As a proof of concept, we chose to control a simulated arm with two degrees-of-freedom, where the gripper of the robotic arm, which is basically the end point of the last arm segment, should be guided to reach a target. The original idea stems from the OpenAI Reacher problem [2] and has been reproduced in essence using matplotlib for licensing reasons. Our experiments for this paper restrict to the 2D space and only use two arm segments with two degrees of freedom. The input to the network is a 84x84 grayscale image and its outputs control changes of the two joint angles with continuous values. Fig. 1 illustrates the simulation environment and the underlying physical states. The generated images include a white dotted background to make the vision task more difficult. Important parts in the image such as the segments of the arm and the target are displayed as bright spots in the images and correspond to a high activation of the networks' input neurons.

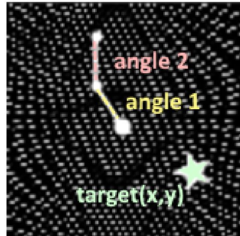

Fig. 1: Physical states of the robot arm in the two-dimensional space with two degrees of freedom.

3.2 Architecture

The actor-critic architecture of our algorithm is partially based on the A3C algorithm [Mn16]. As the algorithm should work on raw pixels, the first layers of both the actor and the critic are convolutional, but without pooling as the spatial information is critical for the task here. It proved difficult to directly train the complete network as convolutional layers need many samples to converge. We settled for a hybrid approach, where the convolutional layers were first trained to reproduce the physical states in a supervised manner, which consist of the arm segments angles and the target position (4 variables in our simulated task). After training, the *internal model* can predict the physical state for a given image, which is then used as an input for the actor and the critic. The complete architecture is depicted in Fig. 2.

[2] `https://gym.openai.com/envs/Reacher-v1`

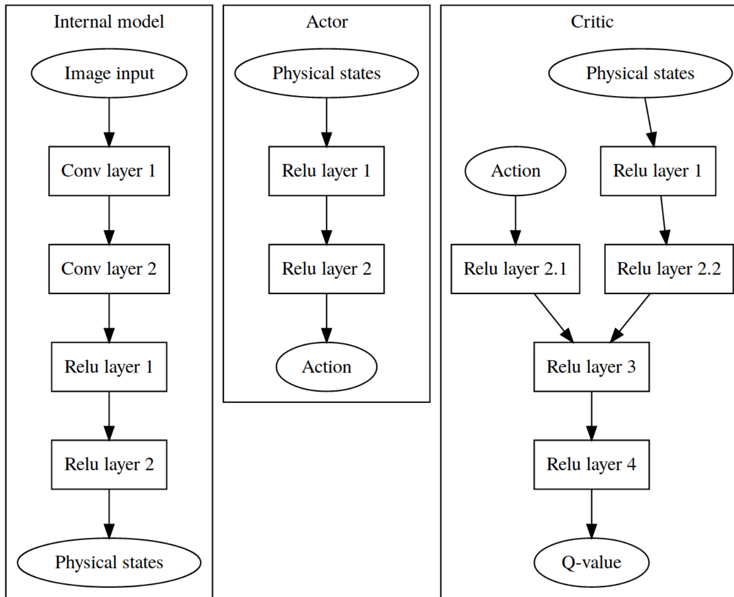

Fig. 2: Architecture of the complete network model. The internal model transforms the visual information into a physical state representation. The actor outputs an action sampled from the continuous action space for a given state. The critic receives state and generated action as input and outputs the action value.

The internal model uses two convolutional layers: the first convolutional layer has 8 filters and a kernel size of 4x4, whereas the second one has 32 filters with a kernel size of 6x6. The two following fully connected layers extract information about the physical states of the arm and the target position: they are composed of rectified linear units with 90 and 50 neurons respectively. The output layer consists of 4 neurons with hyperbolic tangent activation functions to model the physical states.

The actor network is used to predict an action, which is a two-dimensional vector in our setup, defining the two components of the desired angular motion for the first and the second segment of the arm respectively. At each step, each arm segment can be maximally moved by 2 degrees in either direction, where the actions are scaled to the interval $[-1, 1]$. The output layer has only two neurons with hyperbolic tangent activation functions. Both hidden layers of the actor model consist of 200 rectified linear units. The critic has only one linear output neuron, which represents the Q-value for the combined two-dimensional movement vector passed into the network. An action chosen from the continuous action space is seen as a transition from one position of the gripper to another. All hidden layers of the critic consist of 200 rectified linear units.

3.3 Learning procedure

The learning procedure is organized into episodes, where the initial arm position and the target position are randomly selected at the beginning. At each time step, the input image is fed into the internal model, which outputs a prediction of the state variables s_t. It is then fed into the actor, which outputs a continuous action $\mu(s_t)$. The selected action a_t is the sum of this action and of a random variable taken from an Ornstein Uhlenbeck process ($\mu = 0$, $\theta = 0.3$ and $\sigma = 0.4$). This additive noise encourages exploration, as the learned policy is deterministic.

The critic Q with weights θ_Q can be trained with classical Q-learning [SB98]: after each action, the expected return is approximated with $R_t = r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}))$. This target value for the critic is then saved in a buffer together with the state s_t and the action a_t, as the neural network is trained in minibatches of 5 steps:

$$\mathcal{L}(\theta_Q) = \sum_{t=1}^{5} [r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1})) - Q(s_t, a_t)]^2 \tag{2}$$

Equation 2 uses a Q-value prediction for the next state-action pair $Q'(s_{t+1}, \mu'(s_{t+1}))$ which depends on the output of both the actor (for the next action) and the critic (for its Q-value). Since [Mn15], it is known that training stability can be improved by not using the current weights of the networks to perform this prediction, but those from an older version: the *target network*. Target networks for both the actor μ' and the critic Q' are used to estimate the expected return R_t (Eq. 2). Contrary to the classical approach [Mn15], the target networks will not be updated after a certain amount of training steps, but gradually replicate the changes made to the actor and critic, as in [Li15]:

$$\theta' \leftarrow \tau \, \theta + (1 - \tau) \, \theta' \tag{3}$$

The update rate for the target networks is set to a relatively low value of $\tau = 0.001$. Early experiments showed that a higher update rate in combination with the continuous reward function would lead to fast rising Q-values, which means very high output values from the critic and unstable learning. The deterministic policy gradient algorithm of [Li15] is finally used to train the actor μ with minibatches of 5 steps:

$$\nabla_{\theta_\mu} J(\theta_\mu) = \sum_{t=1}^{5} \nabla_a Q(s_t, \mu(s_t)) \nabla_{\theta_\mu} \mu(s_t) \tag{4}$$

There are 16 parallel learners exploring the environment with different initial configurations, all updating simultaneously the same actor and critic networks as in [Mn16]. Instead of having

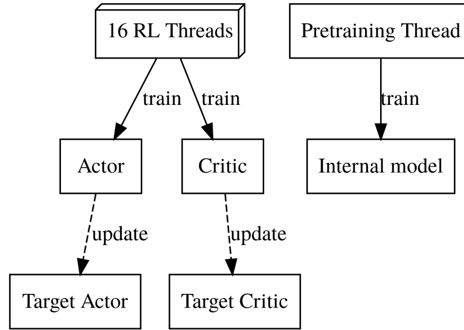

Fig. 3: Summary of the parallel learning process. A single thread pre-trains the internal model, whereas 16 threads run the reinforcement learning based training for the actor and critic in parallel. Actor and critic networks slowly update their respective target networks at each time step.

an episode split into minibatches of 5 steps, what would generate highly correlated inputs, the networks therefore receive small minibatches coming randomly from 16 uncorrelated environments. This could be seen as a distributed experience replay memory and greatly improves the convergence of the networks. Ideally, the parallel learners should update the networks after each step, but the risk of collision between the threads would become too important. Fig. 3 summarizes the parallel execution of the algorithm.

3.4 Overview of the algorithm

The algorithm executed by each parallel learner during a single episode is described in List. 1. The network has been implemented using the tensorflow library and simulated on a shared-memory system with 16 cores and 8 Tesla K20m GPUs. The training of the internal model works faster on the GPU, because larger batch sizes can be used and thus training can be more effectively parallelized by tensorflow. The Adam learning rule is used [KB14] to train all networks, with a learning rate of 10^{-4} for the actor and critic networks, and $8 \cdot 10^{-4}$ for the internal model. As the network structure is very similar to [Li15], most of the weight initialization parameters and learning rates were preserved. To limit the increase of the Q-values during training, an L2 regularization penalty (coefficient 0.02) is introduced for the weights to the output neuron of the critic. A discount factor of 0.97 is used, as episodes last maximally 1000 steps.

3.5 Reward function

The environment returns a reward for each executed action, consisting of two parts: a distance-related part and a control part. To effectively control the gripper, one would like

```
while t < 1000 do
    Execute action a_t according to policy μ(s_t) + εN
    Receive reward r_{t+1} and observe new state s_{t+1}
    Compute R_t = { r_{t+1} + γ · Q'(s_{t+1}, μ'(s_{t+1})) if not terminal
                  { r_{t+1} otherwise.
    Store (s_t, a_t, R_t) in buffer
    if t % 5 == 0 then
        Update critic: L(θ_Q) = Σ_{i=1}^{5} (R_i − Q(s_i, a_i))^2
        Update actor: ∇_{θ_μ} J(θ_μ) = Σ_{i=1}^{5} ∇_{μ(s_i)} Q(s_i, μ(s_i)) ∇_{θ_μ} μ(s_i)
        Update target critic: θ_{Q'} ← τθ_Q + (1 − τ)θ_{Q'}
        Update target actor:  θ_{μ'} ← τθ_μ + (1 − τ)θ_{μ'}
        Empty buffer
    end if
    t = t+1
end while
```

List. 1: Algorithm for each actor learner per episode. The outer loop is broken when the target is reached.

to reward actions getting the gripper closer to the target as fast as possible. The distance-related part provides reward based on the distance between the gripper and the target: $r_{dist} = e^{-|x_{gripper} - x_{target}|}$. If we only use this reward, a possible strategy for the algorithm is to circle around the target until the end of the episode, without reaching it: this way it gathers as much reward as possible, as experimentally observed. So we added a control term $r_{ctrl} = |x_{gripper-old} - x_{target}| - |x_{gripper} - x_{target}|$ which rewards the algorithm if the arm endpoint is closer to the target after the action than before, and punishes it otherwise. The total reward is the product of these two terms $r = r_{ctrl} \cdot r_{dist}$ and is normalized to the interval $[-1, 1]$. When r_{dist} falls below 0.1, which is sufficiently small as both axes of the simulated environment reach from -1 to 1, the episode is considered as terminated.

4 Results

For each experiment, we used 2,500,000 samples to train the internal model and 1,200,000 steps for the asynchronous reinforcement learning algorithm. Evaluation was then conducted over 500 episodes with random starting conditions and the percentage of successful episodes was monitored. An episode is considered successful when the gripper reaches the target within 1000 steps. An important point besides proving that the network architecture is able to solve the robotic problem was to investigate the influence of the convolutional layers on the performance of the model. Therefore, we performed the whole learning and evaluation

Architecture	Background noise	Action noise	Successful episodes
2 conv layers	no	no	57 %
2 conv layers	yes	no	55 %
2 conv layers	no	yes	**77** %
2 conv layers	yes	yes	67 %
1 conv layer	no	no	45 %
1 conv layer	yes	no	34 %
1 conv layer	no	yes	65 %
1 conv layer	yes	yes	61 %
no conv layers	no	no	31 %
no conv layers	yes	no	29 %
no conv layers	no	yes	60 %
no conv layers	yes	yes	59 %

Tab. 1: Experimental results. Percentage of successful episodes (target reached before 1000 steps) when varying the number of convolutional layers in the internal model, the presence of a noisy background in the image and the presence of additive noise in the action selection.

process for two different environments, with or without background noise in the image. Furthermore, we tested one internal model with the last convolutional layer replaced by a fully connected layer with 200 rectified linear units and another with both convolutional layers replaced by feed-forward layers consisting of 200 rectified linear units each.

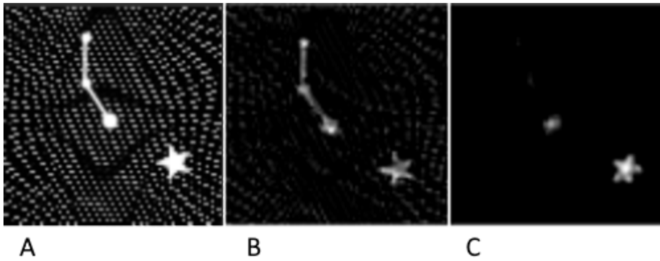

Fig. 4: Convolutional layers are trained to effectively remove background noise and focus on specific features. A: Input to the first convolutional layer with background noise. B: Output of one neuron of the first convolutional layer. C: Output of one neuron of the second convolutional layer selective for the target position.

The experimental results showed in Tab. 1 confirm that using convolutional layers effectively eliminates background noise such as the white spots spread in the image. The performance during evaluation without additive noise in the action selection and with two convolutional layers only slightly decreased by 2 % from 57% to 55%, when adding the background noise. Figure 4-C illustrates this observation, as the background noise disappears in the second convolutional layer. The figure also shows the ability of convolutional layers to extract only relevant information from the image like the target position and thus adapt to the specific task. In contrast, the performance substantially decreases, when no convolutional layers or

only one is used, independent of the presence of different kinds of noise. This confirms the importance of convolutions to process visual input.

The best performance of 77% was reached when the actions generated from the actor during evaluation were not directly executed, but additional noise sampled from an Ornstein-Uhlenbeck process was added as during training. Analysis of the videos showed that most of the unsuccessful episodes failed because of the inaccurate identification of the physical states by the internal model, which means that the arm reaches for a target with a relatively small offset from the real target. Figure 5 illustrates the problem. The reinforcement learning process directly on the real physical states however reaches a high success rate over 90%.

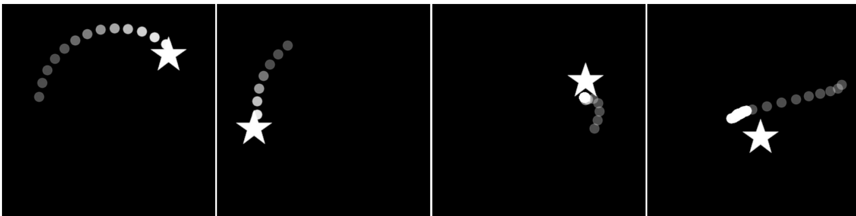

Fig. 5: Movement of the gripper for two successful episodes (first two pictures) and two failed attempts (last two pictures). The brighter the color of a point, the later the gripper was at that position during the episode.

Videos showing a random selection of 50 successful episodes and 5 failed episodes can be found on Youtube[3]. The unsuccessful episodes last much longer, because they exploit all possible 1000 steps and thus less are recorded. The simulated arm with its two segments is shown in its random starting position together with the position of the target. All movements of the arm are shown, which either lead to the successful end of an episode as the gripper reaches the target or a failed episode, when the gripper does not reach the target after 1000 steps. The videos consist of screenshots taken from the environment at regular intervals. Both were generated from an actor trained on environments without background and with two convolutional layers. There was no random noise added during the action selection.

5 Discussion

We proposed to extend the deep deterministic policy gradient algorithm [Li15] with asynchronous parallel learners [Mn16] to allow end-to-end learning in continuous action spaces from raw pixels. We applied this novel algorithm to a simplified continuous control task, with a simulated 2-DOF robotic arm and showed that it is able to achieve a satisfying performance. We also further reduced the sample complexity of the algorithm by pretraining an internal model whose role is to transform images into abstract representations. The algorithm therefore combines a model-free actor-critic architecture with a model-based internal model, what could prove beneficial for problems where such models can be learned.

[3] `https://youtu.be/PTdfxGde69s` and `https://youtu.be/u8aMe6M9jMI`

However, analysis of failed trials shows that wrong state estimations by the internal model are the main source of failure in our setup. Future work will address improving the state representation needed by the actor-critic: the four variables used here may represent a too strong bottleneck for the architecture and intermediate representations could improve the performance of the algorithm.

Acknowledgments: This work was supported by the DFG Major Research Instrumentation Programme (INST 270/221-1 FUGG).

References

[Gu17] Gu, S.; Holly, E.; Lillicrap, T.; Levine, S.: Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates. In: Proc. ICRA. 2017.

[He15] Heess, N.; Wayne, G.; Silver, D.; Lillicrap, T.; Tassa, Y.; Erez, T.: , Learning continuous control policies by stochastic value gradients, 2015.

[KB14] Kingma, D.; Ba, J.: Adam: A Method for Stochastic Optimization. In: Proc. ICLR. pp. 1–13, 2014.

[Le16] Levine, S.; Pastor, P.; Krizhevsky, A.; Quillen, D.: Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection. In: Proc. ISER. 2016.

[Li15] Lillicrap, T.P.; Hunt, J.J.; Pritzel, A. et al.: Continuous control with deep reinforcement learning. CoRR, 2015.

[Mn15] Mnih, V.; Kavukcuoglu, K.; Silver, D. et al.: Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, feb 2015.

[Mn16] Mnih, V.; Badia, A.P.; Mirza, M. et al.: Asynchronous Methods for Deep Reinforcement Learning. In: Proc. ICML. feb 2016.

[Ni11] Niu, F.; Recht, B.; Re, C.; Wright, S.J.: HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. 1, p. 21, 2011.

[SB98] Sutton, R.S.; Barto, A.G.: Reinforcement learning: An introduction, volume 28. MIT press, 1998.

[Si14] Silver, D.; Lever, G.; Heess, N.; Degris, T.; Wierstra, D.; Riedmiller, M.: Deterministic Policy Gradient Algorithms. In (Xing, Eric P; Jebara, Tony, eds): Proc. ICML. volume 32 of Proceedings of Machine Learning Research, PMLR, Bejing, China, pp. 387–395, 2014.

[Su99] Sutton, R.S.; McAllester, D.A; Singh, S.P.; Mansour, Y.: Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: Neural Inf. Process. Syst. 12. pp. 1057–1063, 1999.

[Wi92] Williams, R. J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach. Learn., 8:229–256, 1992.

[Zh15] Zhang, F.; Leitner, J.; Milford, M.; Upcroft, B.; Corke, P.: Towards Vision-Based Deep Reinforcement Learning for Robotic Motion Control. In: Proc. Acra. 2015.