TECHNISCHE UNIVERSITÄT
CHEMNITZ

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

Winfried Lötzsch, Julien Vitay and Fred Hamker

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 DOF robotic arm and discuss its applications to more realistic scenarios.

Contents

Chapter 1

Introduction

Reinforcement learning (RL, Sutton and Barto (1998)) 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. Mnih et al. (2015) 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. 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 (Zhang et al., 2015). Alternatively, deep networks can be used as a submodule of the complete system: Levine et al. (2016) 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 (Heess et al., 2015), 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 (Silver et al., 2014; Lillicrap et al., 2015)). 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 (Levine et al., 2016; Gu et al., 2017). Multiplying robots reduces the acquisition time, but does not impact the learning properties.

In the discrete domain, Mnih et al. (2016) 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 Mnih et al. (2016) with the DDPG algorithm of Lillicrap et al. (2015) 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 2 DOF robotic arm, using raw pixels as inputs and joint angles as continuous outputs, and discuss its applicability to more complex problems.

Chapter 2

Related work

2.1 Background in deep 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 $r(s_t, a_t) : \mathcal{S} \times \mathcal{A} \to \Re$. The policy is defined as a mapping of the state space into the action space: a stochastic policy $\pi_\theta : \mathcal{S} \to P(\mathcal{A})$ defines the probability distribution $P(\mathcal{A})$ of performing an action, while a deterministic policy $\mu_\theta(s_t)$ is a discrete mapping of $\mathcal{S} \to \mathcal{A}$. $\theta \in \Re^n$ 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:

$$Q^{\pi_\theta}(s, a) = \mathbb{E}_{\pi_\theta}(R_t | s_t = s, a_t = a) \tag{2.1}$$

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:

$$\mathcal{L}(\theta) = \mathbb{E}_{\pi_\theta}[r_t + \gamma Q^{\pi_\theta}(s_{t+1}, a_{t+1}) - Q^{\pi_\theta}(s_t, a_t)]^2 \tag{2.2}$$

Policy gradient methods directly learn to produce the policy (stochastic or not). The goal of the neural network is to maximize an objective function $J(\theta) = \mathbb{E}_{\pi_\theta}(R_t)$. The *stochastic policy gradient theorem* (Sutton et al., 1999) provides a useful estimate of the gradient that should be given to the neural network:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)] \tag{2.3}$$

2.2 Continuous action spaces

Eq. 2.3 depends on the unknown Q-value $Q^{\pi_\theta}(s,a)$, as policy gradient methods only output the policy π_θ. $Q^{\pi_\theta}(s,a)$ could be approximated by the actual return R_t after that action, leading to the REINFORCE learning rule (Williams, 1992), widely used in deep RL. Another option is to use a second neural network to learn to approximate the Q-value, similarly to Eq. 2.2. 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 Silver et al. (2014) 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. Silver et al. (2014) derived the *deterministic policy gradient theorem* allowing the deterministic actor to learn in this framework:

$$\begin{aligned}\nabla_\theta J(\theta) &\approx \mathbb{E}_{\pi_\theta}[\nabla_\theta Q(s,a|\theta)|_{s=s_t,a=\mu_\theta(s_t)}] \\ &= \mathbb{E}_{\pi_\theta}[\nabla_a Q^{\pi_\theta}(s,a)|_{s=s_t,a=\mu_\theta(s_t)} \times \nabla_\theta \mu_\theta(s)|_{s=s_t}] \qquad (2.4)\end{aligned}$$

Lillicrap et al. (2015) 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 Lillicrap et al. (2015) 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 (Levine et al., 2016; Gu et al., 2017). Mnih et al. (2016) 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! (Niu et al., 2011). 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.

Chapter 3

Methods

3.1 Simulated robotic arm

As a proof of concept, we chose to control a simulated arm with multiple 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 [1] 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 (an example can be seen on Fig. 4.1-A) and its outputs control changes of the two joint angles with continuous values. 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.

3.2 Architecture

The actor-critic architecture of our algorithm is partially based on the A3C algorithm (Mnih et al., 2016). 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. 3.1.

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 activation functions to model the physical states.

---

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

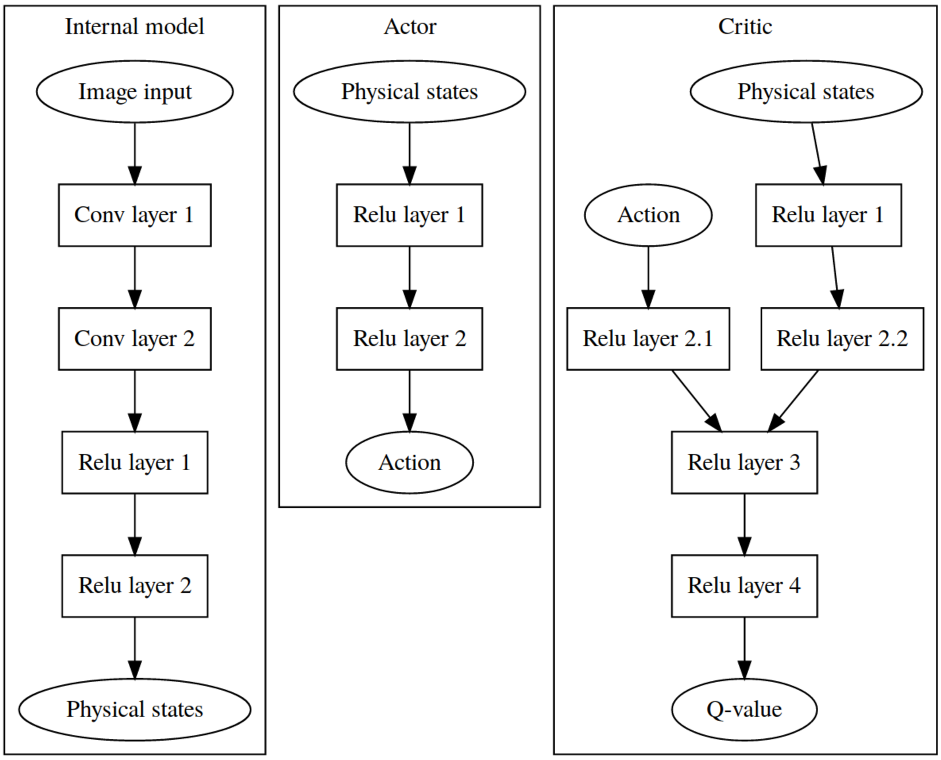

Figure 3.1: Architecture of the complete network model. The internal model transforms the visual information into a physical state representation. The actor outputs an action sampled from the continuous action space for a given state. The critic receives state and generated action as input and outputs the action value.

The 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. Per iteration each arm segment can be maximally moved by 2 degrees in either direction, whereby the actions are scaled to the interval $[-1, 1]$. The output layer has only two neurons with a hyperbolic activation function. 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 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 (as in Eq. 2.2). 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. The critic network is used to directly approximate the action value function $Q(s, a)$. We will however need its gradient w.r.t the action when training the actor (see Eq. 2.4), but it can be easily obtained as the critic is implemented with tensorflow.

$$\mathcal{L}(\theta_Q) = \sum_{t=1}^{5} [r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1})) - Q(s_t, a_t)]^2 \tag{3.1}$$

Equation 2.2 uses a Q-value prediction for the next state-action pair $Q^{\pi_\theta}(s_{t+1}, a_{t+1})$ which depends on the output of both the actor (for the next action) and the critic (for its Q-value). Since Mnih et al. (2015), 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 introduced and used to estimate the expected return R_t (Eq. 3.1). Contrary to the classical approach (Mnih et al., 2015), 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 Lillicrap et al. (2015):

$$\theta' \leftarrow \tau\, \theta + (1 - \tau)\, \theta' \tag{3.2}$$

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 Lillicrap et al. (2015) is finally used to train the actor μ with minibatches of 5 steps:

$$\nabla_{\theta_\mu} J(\theta_\mu) = \sum_{t=1}^{5} \nabla_a Q(s_t, \mu(s_t)) \times \nabla_{\theta_\mu} \mu(s_t) \tag{3.3}$$

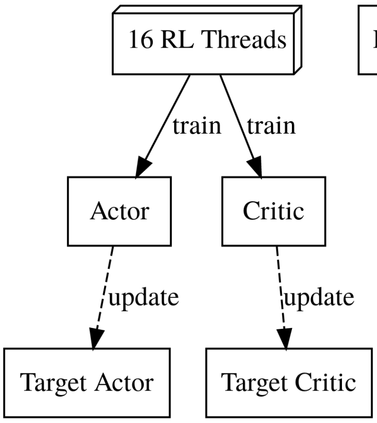

Figure 3.2: Summary of the parallel learning process. A single thread pre-trains the internal model, whereas 16 threads run the reinforcement learning based training for the actor and critic in parallel. Actor and critic networks slowly update their respective target networks at each time step.

There are 16 parallel learners exploring the environment with different initial configurations, all updating simultaneously the same actor and critic networks as in Mnih et al. (2016). Instead of having 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. 3.2 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 Listing 3.1.

```
while t < 1000 do
    Execute action $a_t$ according to policy $\mu(s_t) + \epsilon\mathcal{N}$
    Receive reward $r_{t+1}$ and observe new state $s_{t+1}$
    Compute $R_t = \begin{cases} r_{t+1} + \gamma \cdot Q'(s_{t+1}, \mu'(s_{t+1})) & \text{if not terminal} \\ r_{t+1} & \text{otherwise.} \end{cases}$
    Store $(s_t, a_t, R_t)$ in buffer
    if t % 5 == 0 then
        Update critic: $\mathcal{L}(\theta_Q) = \sum_{i=1}^{5} (R_i - Q(s_i, a_i)]^2$
        Update actor: $\nabla_{\theta_\mu} J(\theta_\mu) = \sum_{i=1}^{5} \nabla_{\mu(s_i)} Q(s_i, \mu(s_i))) \times \nabla_{\theta_\mu} \mu(s_i)$
        Update target critic: $\theta_{Q'} \leftarrow \tau\theta_Q + (1-\tau)\theta_{Q'}$
        Update target actor:  $\theta_{\mu'} \leftarrow \tau\theta_\mu + (1-\tau)\theta_{\mu'}$
```

```
            Empty buffer
        end if
    end while
```

Listing 3.1: Algorithm for each actor learner per episode. The outer loop is broken when the target is reached.

The algorithm has been implemented using the tensorflow library and has been 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 (Kingma and Ba, 2014) 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 Lillicrap et al. (2015), 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 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 from 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.

Chapter 4

Results

For each experiment, we used 2,500,000 samples to train the internal model and 1,200,000 iterations 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 iterations. One important point besides proving that the network architecture in general is able to solve the designed 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 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.

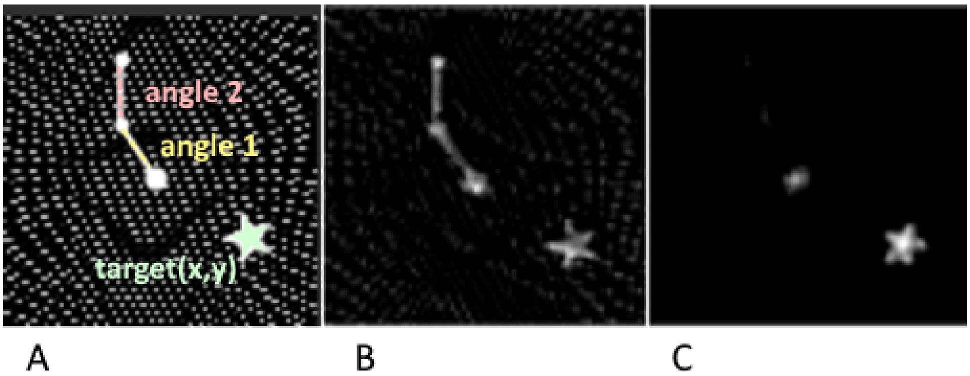

Figure 4.1: A: Physical states of the robot arm in the two-dimensional space with two degrees of freedom. B: Output of one neuron of the first convolutional layer. C: Output of one neuron of the second convolutional layer selective for the target position.

Experiments showed, that the usage of convolutional layers effectively eliminates background noise such as the white spots spread in the image. The performance during evaluation only slightly decreased by 2 % from 57% to 55%, when adding the background noise. Figure 4.1-C illustrates this, 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 (by 30%) or only one (by 19%) is used. This confirms the importance of convolutions to process visual input.

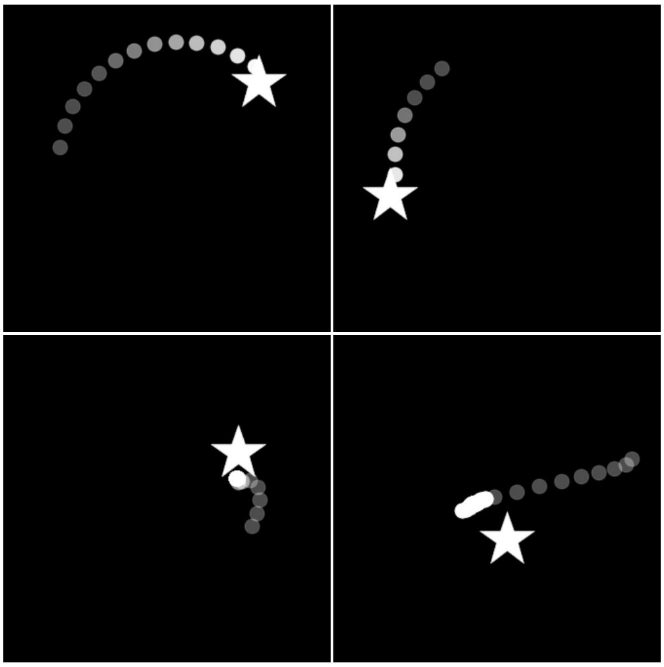

Figure 4.2: Movement of the gripper for two successful episodes (top row) and two failed attempts (bottom row). The brighter the color of a point, the later the gripper was at that position during the episode.

The best performance of 77.3% 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, see Tab. 4. 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 4.2 illustrates the problem. The reinforcement learning process on the real physical states however reaches a high success rate over 90%.

Table 4.1: Experimental results

Experiment	Amount of successful episodes
without background	57 %
without background, noise added	77.3 %
with background	55 %
only one convolutional layer	38 %
no convolutional layers	27 %

Videos showing a random selection of 50 successful episodes and 5 failed episodes can be found on Youtube[1]. The unsuccessful episodes last much longer, because they exploit all possible 1000 iterations 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 iterations. The videos consist of screenshots taken from

---

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

the environment at regular intervals. Both are generated from an actor trained on environments without background and with two convolutional layers. There is no random noise added during evaluation.

Chapter 5

Discussion

We proposed to extend the deep deterministic policy gradient algorithm (Lillicrap et al., 2015) with asynchronous parallel learners (Mnih et al., 2016) 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. 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.

Bibliography

Gu, S., Holly, E., Lillicrap, T., and Levine, S. (2017). Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates. In *Proc. ICRA*.

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Tassa, Y., and Erez, T. (2015). Learning continuous control policies by stochastic value gradients.

Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In *Proc. ICLR*, pages 1–13.

Levine, S., Pastor, P., Krizhevsky, A., and Quillen, D. (2016). Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection. In *Proc. ISER*.

Lillicrap, T., Hunt, J., Pritzel, A., et al. (2015). Continuous control with deep reinforcement learning. *CoRR*.

Mnih, V., Badia, A., Mirza, M., et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. In *Proc. ICML*.

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

Niu, F., Recht, B., Re, C., and Wright, S. (2011). HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Proc. Adv. Neural Inf. Process. Syst.*, page 21.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. In Xing, E. P. and Jebara, T., editors, *Proceedings of ICML*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Beijing, China. PMLR.

Sutton, R. and Barto, A. (1998). *Reinforcement learning: An introduction*, volume 28. MIT press.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Neural Inf. Process. Syst. 12*, pages 1057–1063.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256.

Zhang, F., Leitner, J., Milford, M., Upcroft, B., and Corke, P. (2015). Towards Vision-Based Deep Reinforcement Learning for Robotic Motion Control. In *Proc. Acra*.