Parallelized deep reinforcement learning for robotic manipulation

Pilot study

Isac Arnekvist
isacar@kth.se
KTH Robotics Perception and Learning
Supervisor: Johannes A. Stork

January 23, 2017

Contents

1 Background

1.1 Objective

The interest in conducting this thesis research started with a series of articles published by researchers at Google in their research blog [1–4]. The main theme in these articles was robotic manipulation learned by gathering experience in real time in non-simulated contexts. In two of these articles [1,2] tasks are learned from scratch without the need for initializing by demonstration. Although, in the article by Gu et al. [1], poses of targets and arms are known from attached equipment. It would be interesting to incorporate estimation of poses from visual feedback in this case to lessen the need for external equipment. Another central theme in these articles is the distributed collection of experience over several robots. This is done in order to decrease the time it takes to collect data and to increase variance of the data and the robustness of the algorithms. The use cases for incorporating and extending these findings could be robotic manipulation tasks with camera as feedback where exact relative positions of objects, manipulators, and sensors need not be fixed. Also, where resources exists to use several robots for speeding up the learning process. Possible readers might be other researchers working with end-to-end machine learning for robotic manipulation. Other interested parties might also be manufacturers where repetitive tasks are a part of the production chain and variations in these make it hard for robots to be easily programmed for those tasks.

1.2 Reinforcement learning

This entire section is descriptions of key concepts from a book on Reinforcement Learning by Sutton and Barto [5].

1.2.1 The three tiers of machine learning

In reinforcement learning (RL) an agent interacts with an environment and tries to maximize how much *reward* it can observe from the environment. To maximize the reward in the long run might require short-time losses, making the problem more complex than just maximizing for one step at a time. To find a good strategy, commonly referred to as a *policy*, the agent uses its experience to make better decisions, this is referred to as *exploitation*. But, it must also find a balance between exploitation and to also try out new things, i. e. *exploration*. These things are specific for RL and therefore distinguishes it from supervised and unsupervised learning making it the third piece of machine learning.

1.2.2 Main elements of RL

Let S_t be the state at time t, R_t be the reward at time t, and A_t the action at time t. The interaction between an agent and its environment in RL is depicted in figure 1. At time step t, the agent reads the environment state and takes an action. The environment changes, maybe stochastically, by responding with a new state and a reward at time $t+1$.

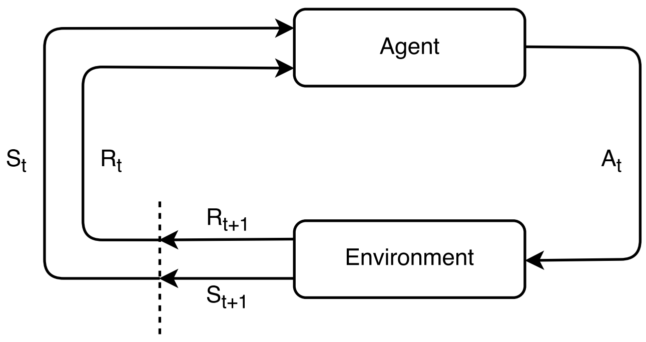

Figure 1: Agent and environment interaction in RL. S_t, R_t, and A_t is the state, reward, and action at time t.

The quantity to maximize is often not the individual rewards, but rather the long term accumulated rewards. Let us call this quantity G_t, or *return*, for an agent at time t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

Some problems imply infinite sequences of actions and rewards and in turn implying that the sum of all rewards be infinite. It is of course a problem to maximize something infinite and that is the reason for the $\gamma \in [0, 1]$ factor above that alleviates this problem if $\gamma < 1$. For lower values of γ the agent tries to maximize short term rewards, and for larger values long-term rewards.

A policy is a function from the state of the environment to probabilities over actions, i. e. the function that chooses what to do in any situation. Since a reward is only short-term, a *value function* tries to estimate the total amount of reward that will be given in the long run for being in some state and following some policy. To enable planning of actions in the environment, RL algorithms sometimes uses a *model* in order to try out actions in this before making decisions. This is usually referred to as *model-based* RL in contrast to *model-free*. TODO: Change "try out" above if not correct.

1.2.3 Finite Markov Decision Processes

In a RL scenario where the environment has a finite number of states, there is a finite number of actions, and the Markov property holds is called a *finite Markov Decision Process* (finite MDP). The dynamics of a finite MDP is completely specified by the probability distribution:

$$p(s', r|s, a) = P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{2}$$

Important functions and terminology that is used throughout RL includes the *state-value function* (abbreviated as value function) and the *action-value function.* The state-value function with respect to some policy informally tells you how good a state is to be in given that you follow that policy:

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t|S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right] \tag{3}$$

To compare the value of different actions in some state, given that you thereafter follow some policy π, is given by the action-value function:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t|S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{4}$$

According to RL theory there is always an optimal policy, i. e. that gives the highest possible expected return given any state. This is often denoted with $*$ and has the corresponding value and action-value functions $v_*(s)$ and $q_*(s,a)$. Given the optimal value or action-value function, it is (depending on the problem) easy to infer the optimal policy, therefore a common approach is to first approximate either of these functions.

1.2.4 Policy and value iteration

One exact method to find the optimal policy, at least in the limit, is called *policy iteration.* This builds on two alternating steps, the first called *iterative policy evaluation.* This estimates a value function given some policy and starts from a random value function v_0, except for any terminal state which is assigned 0. Then we iteratively update new value functions for each step:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s\right] \\ &= \sum_a \pi(a|s) \sum p(s', r|s,a)\left[r + \gamma v_k(s')\right] \end{aligned}$$

As can be seen, the dynamics $p(s', r|s,a)$ needs to be known, which of course is not always the case. The next step is called *policy improvement* and for this we first need to calculate the action-state function given the current policy π:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r|s,a)\left[r + \gamma v_\pi(s')\right] \end{aligned}$$

When we have this, we can get an improved policy π' by:

$$\pi'(s) = \arg\max_a q_\pi(s,a) \tag{5}$$

Iteratively doing these two steps will eventually converge to the optimal policy. There is an alternative way that is done by only approximating the value function, called *value iteration*:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_k(s')\right] \end{aligned}$$

1.2.5 Monte Carlo methods and Time Difference learning

Policy and value iteration are exploring the entire state-action space and finds an optimal policy if the dynamics of the environment are known. Sometimes we are dealing with samples from interacting with a system, and where we do not know the dynamics. For these cases, we can instead estimate the action-value function given a policy. This can be done by *Monte Carlo methods* which basically is averaging of returns for samples that we have attained. The other method is *Time difference methods* which estimates an error for each observed reward and updates the action-value function with this. To be more precise, one famous example of a time difference method is *Q-learning* and the updates are done according to:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{6}$$

Q-learning is an example of an *off-policy* method. This means that you can use a second, or derived, policy for exploration, but the algorithm still finds the greedy optimal policy. The other family of methods is called *on-policy* methods. In this case, the exploring policy is the same as the policy being improved.

1.3 Extending Q-learning to continuous domains

The Q-learning method assumes you can keep a finite set of estimates for all state-action pairs. Continuous state and action spaces require other methods, or modifications to Q-learning to be possible to use. Gu et al. [6] proposes a relatively simple variant of Q-learning they call normalized advantage functions (NAF). They argue after doing simulation experiments that this algorithm is an effective alternative to recently proposed actor-critic methods and that it learns faster with more accurate resulting policies. First, the action-value function is divided into a sum of the value function V and what they call an advantage function A:

$$Q(\mathbf{x}, \mathbf{u}) = A(\mathbf{x}, \mathbf{u}) + V(\mathbf{x}) \tag{7}$$

Here, $\mathbf{x}$ is the state of the environment and $\mathbf{u}$ are controls or actions. The advantage function is a quadratic function of u:

$$A(\mathbf{x}, \mathbf{u}) = -\frac{1}{2}(\mathbf{u} - \mu(\mathbf{x}))^T \mathbf{P}(\mathbf{x})(\mathbf{u} - \mu(\mathbf{x})) \tag{8}$$

There are more terms that need to be defined, but let's look at equation 8 first. The matrix $\mathbf{P}$ is a positive-definite matrix, this makes the advantage function have its maximum when $\mathbf{u} = \mu(\mathbf{x})$. The purpose of μ is to be a greedy policy function, thus $\mathbf{Q}$ is maximized when $\mathbf{u}$ is the greedy action. The purpose of this is that given an optimal Q, we do not need to search for the optimal action, since we know μ. Now the definition of $\mathbf{P}$:

$$P(\mathbf{x}) = \mathbf{L}(\mathbf{x})\mathbf{L}(\mathbf{x})^T \tag{9}$$

Here, $\mathbf{L}$ is a lower-triangular matrix where diagonal entries are strictly positive.

After these definitions, we are left with estimating the functions V, μ, and $\mathbf{L}$. To this end the authors use a neural network, here shown in figure 2. The $\mathbf{L}$ output is a matrix multiplication with previous layer and not passed through an activation function. The diagonal entries of L are exponentiated. Hidden layers consisted of 200 fully connected units with rectified linear units (ReLU) as activation functions except for $\mathbf{L}$ and A as already defined.

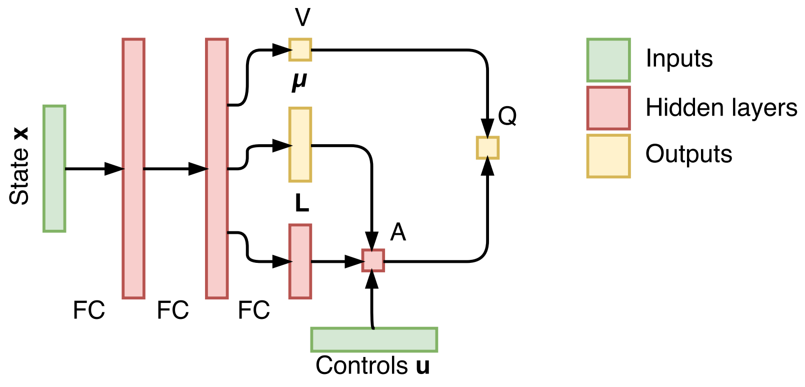

Figure 2: Neural network design for NAF. FC is short for fully connected layers.

The NAF algorithm is listed in algorithm 1. All collected experiences are stored in a replay buffer that optimization is run against. Exploration is done by adding noise to the current greedy policy μ.

1.4 Distributed real-world learning using NAF

Real-world experiments were done by Gu et al. [1] on door opening tasks using the NAF algorithm and 7-DoF torque controlled arms. They extended the algorithm to be distributed on several robots/collectors and one separate trainer thread on a separate machine. They state that this was the first successful real-world experiment with a relatively high complexity problem without human demonstration or simulated pretraining. They used the layout of the network shown in figure 2 but with hidden layers of 100 units each. Also, in this article it was explicitly mentioned that the activation functions for the policy μ was tanh in order to bound the actions. The state input consisted of

Algorithm 1 NAF algorithm

```
Randomly initialize network $Q(x, u|\theta)$
Initialize target network $Q'$, $\theta' \leftarrow \theta$
Initialize replay buffer $R \leftarrow \emptyset$
for episode = 1 to $M$ do
    Initialize random process $\mathcal{N}$ for action exploration
    Receive initial exploration state $x_1$
    for $t = 1$ to $T$ do
        Select action $\mathbf{u_t} = \mu(\mathbf{x_t}|\theta) + \mathcal{N}_\mathbf{t}$
        Execute $\mathbf{u_t}$ and observe $r_t$ and $\mathbf{x}_{t+1}$
        Store transition $(\mathbf{x}_t, \mathbf{u}_t, r_t, \mathbf{x}_{t+1})$ in $R$
        for iteration = 1 to $I$ do
            Sample a random minibatch of $m$ transitions from $R$
            Set $y_i = r_i + \gamma V'(\mathbf{x}_{i+1}|\theta')$
            Update $\theta$ by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(\mathbf{x}_i, \mathbf{u}_i|\theta))^2$
            Update target network $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$
        end for
    end for
end for
```

the arm pose and target pose. The target pose was known from attached equipment. The modified version of NAF is listed in algorithm 2.

The authors conclude that there was an upper bound on the effects of parallelization, but hypothesize that the speed of the trainer thread has a limiting factor in this matter. They used CPU for training the neural network so instead using a GPU might increase the effect of more collectors.

1.5 Motion planning by "Deep visual foresight"

This article [2] trains a convolutional neural network on images together with motion as inputs to predict how the image will change due to that motion. This is later used to plan movement of objects to some target pose.

1.6 Path Integral Guided Policy Search

In this article [4], the authors extend Guided Policy Search and demonstrate two manipulation tasks. These are initialized from demonstrations. To be able to comprehend this article, referenced articles [7–9] would have to be read as well.

Algorithm 2 Asynchronous NAF - N collector threads and 1 trainer thread

```
// trainer thread
Randomly initialize network $Q(\mathbf{x}, \mathbf{u}|\theta)$
Initialize target network $Q'$, $\theta' \leftarrow \theta$
Initialize shared replay buffer $R \leftarrow \emptyset$
for iteration = 1 to $I$ do
    Sample a random minibatch of m transitions from $R$
    Set $y_i = \begin{cases} r_i + \gamma V'(x_i|\theta) & \text{if } t_i < T, \\ r_i & \text{if } t_i = T \end{cases}$
    Update $\theta$ by minimizing the loss: $L = \frac{1}{m}\sum_i (y_i - Q(\mathbf{x}_i, \mathbf{u}_i|\theta))^2$
    Update the target network: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$
end for
// collector thread $n$, $n = 1...N$
for episode = 1 to $M$ do
    Sync policy network weights $\theta_n \leftarrow \theta$
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $x_1$
    TODO
    for $t = 1$ to $T$ do
        Select action $\mathbf{u}_t = \mathbf{mu}$
        Execute $\mathbf{u}_t$ and observe $r_t$ and $\mathbf{x}_{t+1}$
        Send transition $(\mathbf{x}_t, \mathbf{u}_t, r_t, \mathbf{x}_{t+1}, t)$ to $R$
    end for
end for
```

1.7 Collective Robot Reinforcement Learning with Distributed Asynchronous Guided Policy Search

This article [3] distributes learning of door opening across several robots. The exact nature of the tasks are varied across robots to increase robustness. The learning is initialized from demonstration.

1.8 Problem statement

Manipulation tasks that seem trivial to a human can be hard to learn for robots, especially from scratch without initial human demonstration due to high sample complexity. Recent research suggests ways to do this but are based on that you know the poses of the objects and the end-effector. For some scenarios these are non-trivial to find out.

Problems also arise when learning in real time by collecting experience. Robots must be able to evaluate their policies regularly at a high rate which is complicated by adding a deep convolutional neural network for pose detection. Also, learning tasks within a feasible time frame is harder when data collection and policy updates happen in real time. The approach of distributing collection of experience over several robots will be evaluated in this thesis for handling this problem.

1.9 Research question

How can deep and distributed reinforcement learning be used for learning and performing dynamic manipulation tasks with unknown poses.

1.10 Expected scientific results

If all goes well, previous results are verified in new contexts. Also they are extended to also handle unknown target and manipulator poses.

2 Method

2.1 Examination method

Preliminary method is using the mentioned distributed version of NAF and extend it with pose estimates from a convolutional neural network. This network is pretrained as in [3] by randomly placing objects and the end-effector and this way generating training data. Several robots will be used to parallelize the training process. The preliminary manipulation task is pushing of objects to some random target position.

2.2 Conditions

There will be need for several robot setups, each including a robot, computer, and camera. These will have to be able to communicate with a separate computer responsible for training the policies/neural networks. In the ideal case, this computer is supplied and has a graphics card compatible with modern neural network libraries.

2.3 Limitations

A proof of concept should be done with a corresponding report (the thesis). There are no requirements for implementation of code that should be delivered as libraries etc. The main contribution is the thesis. All code used for conducting the experiments will be openly published on GitHub.

3 Schedule

3.1 Weekly plan

V.3	Finalize this document
V.4-7	Pilot study and write down related background sections
V.8	Set up robots, method section will be written in parallel
V.9	End-effector and object pose estimation
V.10-11	Implement reinforcement learning algorithms
V.12-13	Tweak and fix bugs in order to accomplish task
V.14	Record and write down results
V.15	Finish the remainder of the thesis (Conclusions/Future work), hand in for review
V.16-17	Review and adjustment process with supervisor
V.18	All reviews from supervisor and corresponding adjustments done. Ready for presentation/public discussion and approval from examiner
V.20	Oral presentation
V.22	Finishing touches, hand in final report to supervisor and examiner

References

[1] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation. *arXiv preprint arXiv:1610.00633*, 2016.

[2] Chelsea Finn and Sergey Levine. Deep visual foresight for planning robot motion. *arXiv preprint arXiv:1610.00696*, 2016.

[3] Ali Yahya, Adrian Li, Mrinal Kalakrishnan, Yevgen Chebotar, and Sergey Levine. Collective robot reinforcement learning with distributed asynchronous guided policy search. *arXiv preprint arXiv:1610.00673*, 2016.

[4] Yevgen Chebotar, Mrinal Kalakrishnan, Ali Yahya, Adrian Li, Stefan Schaal, and Sergey Levine. Path integral guided policy search. *arXiv preprint arXiv:1610.00529*, 2016.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[6] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *arXiv preprint arXiv:1603.00748*, 2016.

[7] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[8] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. A generalized path integral control approach to reinforcement learning. *Journal of Machine Learning Research*, 11(Nov):3137–3181, 2010.

[9] William H Montgomery and Sergey Levine. Guided policy search via approximate mirror descent. In *Advances in Neural Information Processing Systems*, pages 4008–4016, 2016.