Learning robot control

Max Mindt
Autonomous Learning System Seminar
Department of Computer Science
TU Darmstadt
Darmstadt, Germany
maxmindt@msn.com

Abstract

Learning is essential to expand the capabilities of a robot. But what is the meaning of learning for a robot and how is the implementation of a learning task? This question is a key question and in addition to what should be learned.
This paper defines a general learning model, classify robot learning and develops a learning architecture for locomotion of a simple walker. These learning architecture contains a construction of the simple walker, the Q-Learning algorithm, the development of an appropriate reward function and consider the convergence of the learning system.

1 Introduction

Conventional robots are limited to their programmed abilities and a programmer can not implement all possible tasks. But we need robots which expand their abilities by learning, since intelligent autonomous systems becoming increasingly important in our world [7]. They can learn either by themselves or with the help of human supervision. Application areas are e.g. assistive robots, playmate robots in child education, robots for mentoring and assistance in manipulation tasks, and robots that teach movement exercises [7]. As a result from this variety of areas, robot learning is going to be a key ingredient because they can only find its way into these areas if they can learn new tasks. But what is the meaning of learning for a robot. Human learning is based on mental, physical and social skills. It is a process of change that occurs in behavior, thinking and feeling. Simon et al. [9] defines learning as follows:

> "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and effectively the next time"

Learning for a robot means that he has to realize that a change in his actions can be more efficiently the next time. The consequence from this realization is that he has to adapt or change its strategy. But one of the key question is how to transform this process as a learning task into a robot environment. In addition, the question arises what should be learned at all. First of all we need an abstract description of learning for a robot, a formal specification is described in the next Section 2. Based on this description robot learning can be classified in Section 3. The focus of this paper is the development of a learning architecture for locomotion and is described in Section 4. Especially the construction of a walker, the development of a learning algorithm and the arising problems are described. Subsequently, we consider the convergence of the learning algorithm under different specific parameters. The last Section deals with open questions and gives a summary of the whole paper.

2 The Basic Model

This Section describes an abstract learning model and is based on [6]. The key question is what should be learned. A learning control system should be abstract in adapting new learning strategies. This idea contains the control policy π which depends on the time t, the actual state vector $\mathbf{x}$ of the system and the vector α. The vector α is the problem specific parameter of the policy π and needs to be adjust by the learning system. All these components and dependencies are illustrated in Figure 1. The vector $\mathbf{u}$ represents the result of the control policy π and is described as follows

$$\mathbf{u} = \pi(\mathbf{x}, \alpha, t) \tag{1}$$

Generally, the control system can be expressed as a nonlinear function

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) \underset{(1)}{=} f(\mathbf{x}, \pi(\mathbf{x}, \alpha, t)) \tag{2}$$

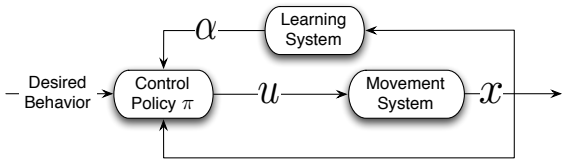


Figure 1: Abstract diagram for learning control

The task of a learning control system is to find a function π which maps the vector $\mathbf{x}$ on the desired behavior. But the control policy can be learned in many different ways which depends on the task. This abstract description of finding a strategy represented as function π allows discussing robot learning in terms of different methods. An overview of these different ways is depicted in the next Section.

3 Classification

An approach to classify robot learning delivers Schaal et al. [8], shown in Figure 2. Schaal makes the subdivision in three different types: direct versus indirect control, learning method and class of task. Topics further out on the arrows can be considered more complex research topics than topics closer to the center. This Section is based on [8].

Suppose you want to learn the Equation (2). One approach is to choose a model like forward -, inverse -, mixed - or multi-step prediction model. Each model has its advantages and disadvantages, exactly studied in [5]. And then compute a controller based on the estimated model. After that, Equation (2) can be learned with function approximation. A Survey of model learning is described by Nguyen-Tuong et al. [5]. Such techniques are summarized under the name Model-Based Learning, Indirect learning, or Internal Model Learning. In contrast, Model-Free Learning of the policy is possible with a reward function. Here, an agent receives a reward and a current state of the environment by performing an action. This principle includes Reinforcement Learning.

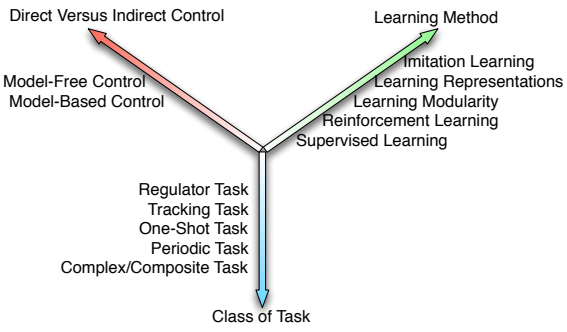


Figure 2: Classification of control, learning method and class of task in Learning robot control

Schaal et al. [8] divides the class of task because it is easier to address the goal of learning. A Regulator Task tries to keep the system in a particular state, e.g. cart-pole is such a task. A task in which a robot follows a desired trajectory is called Tracking Task. An One-shot Task tries to terminate, e.g. grasping a cup of coffee. Locomotion is a typical part of Periodic Movement Task. Complex/Composite Tasks contains complex manipulation, like emptying a dishwasher.

Supervised learning learns the function π of given input and output pairs. Learning from reward and punishment addressed reinforcement learning. The goal is to maximize the reward. All learning methods will benefit and learn from the prior knowledge.

The next Section choose the learning method, the class of task and the control of our application and is based on this classification.

4 Case Study: Reinforcement Learning of Locomotion Controllers

In this Case Study a robot walks on a flat plane, we call this robot simple walker. He has two rigid massless legs at the hip, a point-mass at the hip, no point-masses at the feets, and no upper body. Our task is a Periodic Movement Task and the goal is to build a learning algorithm for stable locomotion. First, we consider some studies about Locomotion in the Subsection 4.1. Subsequently, the Subsection 5 describes the construction and the kinematic model of the simple walker. Afterwards, the learning architecture will be presented which contains the Markov Decision Process and the Q-Learning algorithm. The Subsection 4.4 shows implementation details of this Case Study and the last Subsection 4.5 considers the convergence of the learning algorithm.

4.1 Related Work

There exist several papers about the simple walker and learning of locomotion. Garcia et al. [1] develops a fully dynamic model for a walker and examined the stable running down of a shallow slope. The only free parameter was the ramp slope γ. The model shows stable walking between $0 < \gamma < 0.015$ rad. With increasing γ stable walk becomes more chaotic. However, this dynamic model is to complex for our learning algorithm and walking of a shallow slope is out of scope.
Morimoto et al. [2] develops a model-based Reinforcement Learning algorithm which learns appropriately place the swing leg. His algorithm learns an adequate walking frequency for sloping ground and not how the robot has to walk. However, our goal is to learn locomotion on a flat plane under certain assumptions.
Nakamura et al. [3] develops a new Reinforcement Learning method for a central pattern generator (CPG). This method is called CPG-actor-critic method and rhythmic motor patterns which are controlled by neural oscillators are referred to as CPG. The results of the walker was successful and he walks on an upslope, downslope and rough ground but the learning process was still unstable. However, it is difficult to determine an appropriate movement pattern and adjust the parameters for the CPG. In our case movement patterns are out of scope. We focus on the general construction of a simple walker in the next Subsection.

4.2 Construction

For the simple walker we use the kinematic model (compare Equation (14) from the Appendix) from the SCARA-Manipulator. A construction of our simple walker is depicted in Figure 3. In our case we assume $l_1 = l_2$. Based on the kinematic model, we can determine the position of the head and the foot in the coordinate system S_1. For a representation in S_0 we add the displacement vector $\mathbf{b_0^1}$. But actually the robot is not able to walk, because his base is attached to S_1. This situation is the reason for a little trick. Let us assume we start with the angles $(\theta_1, \theta_2)^T = (3/4\ \pi, 1/2\ \pi)^T$, the foot negotiate the base and ends with $(\theta_1, \theta_2)^T = (1/4\ \pi, 3/2\ \pi)^T$. Now we change the roles and the foot becomes the new base, and vice versa. Additionally, the base vector $\mathbf{b_0^1}$ and the angles $\boldsymbol{\theta}$ needs to be updated as follows[1]

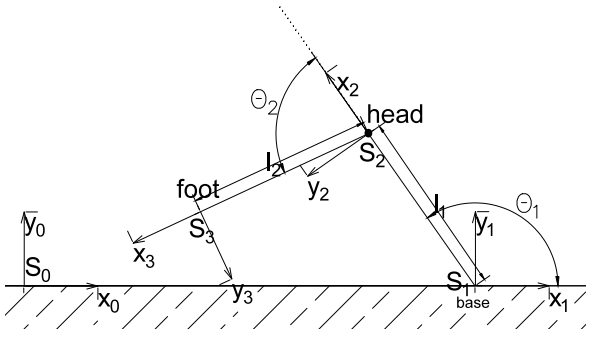


Figure 3: Simple walker as SCARA-Manipulator

$$\boldsymbol{\theta_{new}} = \begin{pmatrix} \theta_{1,new} \\ \theta_{2,new} \end{pmatrix} = \begin{pmatrix} \pi - \theta_{1,old} \\ 2\pi - \theta_{2,old} \end{pmatrix}$$

$$\mathbf{b^1_{0,new}} = \mathbf{f^3_{1,old}} + \mathbf{b^1_{0,old}}$$

At this moment, the robot is, in S_1, in its initial position and can make his next step. The only permanent change is the displacement vector $\mathbf{b_0^1}$. Based on this construction the next Subsection develops an appropriate learning architecture.

[1] The vector b_0^1 is depicted in S_0 and point out the origin of S_1. Similar is the foot vector f_1^3

4.3 Learning Architecture

Consider an autonomous agent acting in its environment. The agent can learn to choose optimal actions, depending on the appropriate receive reward from the environment. This issue addresses Reinforcement Learning and is the learning method of this application. The goal of the agent is to chose sequences of actions that produce the greatest cumulative reward. This problem is based on the theory of the Markov Decision Process (MDP), which is explained in Subsection 4.3.1. Q-Learning is one reinforcement learning technique and can acquire optimal control strategies from delayed reward, also when the agent has no knowledge of the effects of its actions on the environment. This algorithm is explained in the Subsection 4.3.2 but it works only reliably if the defined reward function is appropriately, explained in Subsection 4.4. The implementation details of the learning architecture for the simple walker are also shown in Subsection 4.4. The convergence of the Q-Learning algorithm is described in the last Subsection 4.5.

4.3.1 Markov Decision Process

The Markov Decision Process is a decision problem and the reward for an agent depends on his decisions. The Markov assumption is the probability of reaching a state s' from state s, is only dependent on s and not of his predecessors. This assumption must be satisfied in each state transitions. Ng et al. [4] defines a (finite) MDP is a tuple $(S, A, \{P_{sa}\}, \gamma, r)$, where

- S is a finite set of N **states**
- $A = \{a_1, ..., a_n\}$ is a set of n **actions**
- $P_{sa}(\cdot)$ are the state **transition probabilities** upon taking a action a in sate a
- $\gamma \in [0, 1)$ is the **discount factor**, i.e. a direct reward is better than later reward with the same amount
- $r : S \mapsto \mathbb{R}$ is the **reward function**, bounded in absolute value by $r_{\max}$

In a MDP the agent perceives a set S of distinct states of its environment and has a set of A actions that it can perform. At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it. The environment responds by giving the agent a reward r_t and the next state s_{t+1} depending on the formulas

$$r_t = r(s_t, a_t) \tag{3}$$

$$s_{t+1} = \delta(s_t, a_t) \tag{4}$$

First, we assume the functions r and δ are deterministic and part of the environment. Also the transition probabilities are $P_{sa}(\cdot) \in \{0, 1\}$. The functions are not necessarily known of the agent. The solution of the MDP and task of the agent is to find a policy $\pi : S \to A$. This policy selects the next action a_t, based on the current observed state s_t, or formal $\pi(s_t) = a_t$. The question is how the agent learn the policy? One solution is to require the policy that produces the greatest possible cumulative reward. The cumulative value $V^\pi(s_t)$ for a policy π in an initial state s_t is defined as follows

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{5}$$

With this cumulative value, we can define the learning task. The optimal policy π^* is the maximum of the cumulative value for all states, or formal

$$\forall s \in S.\ \ \pi^*(s) = \arg\max_{\pi} V^\pi(s) \tag{6}$$

Often, to simplify the notation, $V^\pi(s)$ is written as $V^*(s)$. The next step is to develop an algorithm which find an optimal policy π^*. An appropriate algorithm is Q-Learning, described in the next Subsection.

4.3.2 Q-Learning

It is difficult to learn the policy directly because the training data does not provide any examples. The only available training information is the received reward in a sequence. The consequence is to learn a reward function to determine π. One possibility is that the agent tries to learn $V^*(s)$ as reward function. With the condition to chose s_1 instead of s_2 if $V^*(s_1) > V^*(s_2)$. But that would mean that the agent has to decide between states and not actions. Resulting from this decision we define the (Q)uality of a state-action combination as a mapping $Q : S \times A \mapsto \mathbb{R}$. The Q-Function depends on the current reward (3) and the cumulative reward (5) of the successor state

$$Q(s_t, a_t) \underset{(3)+(5)}{=} r(s_t, a_t) + \gamma V^*(s_{t+1}) \underset{(4)}{=} r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t)) \tag{7}$$

And $Q(s_t, a_t)$ can also be maximized instead of $V^*(s_t)$. So we can rewrite the Equation (6) as follows

$$\forall s \in S.\ \ \pi^*(s) = \arg\max_a Q(s, a) \tag{8}$$

The result of rewriting is that the agent learns the Q-Function and choose between actions and does not require any information, in the scope of Equation (8), about the functions δ and r.
The next step is to develop an algorithm that learns Q and implicit π^*. This learning can be accomplished through iterative approximation but first, notice the close relationship between Q and V^*

$$V^*(s_t) = \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{9}$$

With the Equation (9) we can rewrite the Equation (7) and obtain the recursion

$$Q(s_t, a_t) \underset{(9)}{=} r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \underset{(4)}{=} r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(\delta(s_t, a_t), a_{t+1}) \tag{10}$$

This recursion is the core of the algorithm. But the assumption the functions δ and r are deterministic was wrong. There are nondeterministic, because the function r produce different rewards for the same state, also the function δ. In summary, we have a nondeterministic MDP. Therefore we need to adjust a few Equations. The cumulative value $V^*(s_t)$ (see Equation (5)) is now an expectation

$$V^*(s_t) = E\left(\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right)$$

Similar to $V^*(s_t)$, Q is also now an expectation and Equation (7) needs to be updated

$$\begin{aligned} Q(s_t, a_t) &= E(r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= E(r(s_t, a_t)) + E(\gamma V^*(\delta(s_t, a_t))) \\ &= E(r(s_t, a_t)) + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1}) \end{aligned}$$

$P(s_{t+1}|s_t, a_t)$ is the probability to choose an action a_t in state s_t that will produce the next state s_{t+1}. The updating allows us the rewriting of the recursive Equation (10)

$$Q(s_t, a_t) = E(r(s_t, a_t)) + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{11}$$

Now we have a recursive Q-Function for the nondeterministic case with nondeterministic functions δ and r. The Q-Learning algorithm is depicted in algorithm 1 with a learning rate $\alpha(s_t, a_t) \in (0, 1]$. $\widehat{Q}$ is the notation for the estimated value by the learning agent of the function Q. The key idea in this revised rule is, that the changes in $\widehat{Q}$ are more continuously than in the deterministic case (compare Equation (10)). Resulting from this continuity is by reducing α at an appropriate rate, we can achieve convergence to the correct Q function. Notice if $\alpha = 1$ the resulting Equation is (10). But a major component of correct learning is to find an appropriate reward function. This aspect and the associated difficulties is described in the next Subsection.

Algorithm 1 Q-Learning algorithm

For each s, a initialize the Table entry $\widehat{Q}(s, a)$ to zero
Observe the current state s_t
loop
Select an action a_t and execute it
Receive immediate reward $r(s_t, a_t)$
Observe the new state s_{t+1}
Update the Table entry for $Q(s_t, a_t)$ with Equation (11) as follows
$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha_t(s_t, a_t)[r(s_t, a_t) + \gamma \max_{a_{t+1}} \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t)]$
end loop

4.4 Implementation

First, we develop the MDP for the simple walker and in the second step we cater to the reward function. The reward function is a major component of the learning algorithm because correct learning depends on it. This function produces the most errors, therefore we will introduce some of them.
For the MDP we need to define the tuple $(S, A, \{P_{sa}\}, \gamma, r)$. First of all the finite set of N states. The simple walker can stand, move forwards or backwards, reach an illegal position or find a new base. Formally, a set of states can be written as

$$S = \{standing, forward, backward, error, newbase\}$$

The learning algorithm can adjust the angles θ_1 and θ_2. For simplicity, the algorithm can subtract or add one degree of each angle. Formally, a set of actions can be written as

$$A = \{-1, 0, 1\} \times \{-1, 0, 1\}$$

If the action $(-1, 1)$ was selected the new angles are $(\theta_{1,new}, \theta_{2,new})^T = (\theta_{1,old} - 1, \theta_{2,old} + 1)^T$.
For the transition probabilities we implemented the epsilon-greedy strategy. The value of the discount factor γ will be discussed in Section 4.5.
Now we cater to the reward function. Before we deploy the reward function, we make some assumptions

I. The head must always be between the foot and base.
II. The foot must always be below the head and above the bottom.
III. It has to be an adequate distance between foot and head, the foot must have passed the base, and the foot must be in contact with the ground, before the base is changed.

To determine the head and foot we need the coordinates of them. The head is represented by the vector $\mathbf{h_a^2}$ and the foot by the vector $\mathbf{f_a^3}$, in the coordinate system S_a. We need a representation in the coordinate system S_0 but first we can calculate them in S_1 with these formulas

$$T_1^3(\theta_1 + a_{t,1}, \theta_2 + a_{t,2}) = \begin{pmatrix} R_1^3 & \mathbf{f_1^3} \\ \mathbf{0^T} & 1 \end{pmatrix}$$

$$T_1^2(\theta_1 + a_{t,1}, \theta_2 + a_{t,2}) = \begin{pmatrix} R_1^2 & \mathbf{h_1^2} \\ \mathbf{0^T} & 1 \end{pmatrix}$$

from the Appendix A.1. With the displacement vector $\mathbf{b_0^1}$ we get a representation in S_0 as follows

$$\mathbf{f_0^3} = \mathbf{b_0^1} + \mathbf{f_1^3}$$
$$\mathbf{h_0^2} = \mathbf{b_0^1} + \mathbf{h_1^2}$$

With these coordinates and the assumptions we can define some useful functions[2]

$$h(\mathbf{h_0^2}, \mathbf{b_0^1}, \mathbf{f_0^3}) = \begin{cases} 0, & b_{0,x}^1 \leq h_{0,x}^2 \leq f_{0,x}^3 \vee b_{0,x}^1 \geq h_{0,x}^2 \geq f_{0,x}^3 \\ 1, & \text{otherwise} \end{cases}$$

$$c(\mathbf{f_0^3}, \mathbf{g}) = \begin{cases} 1, & f_{0,y}^3 = g_y \\ 0, & \text{otherwise} \end{cases}$$

[2] $\mathbf{h_0^3}, \mathbf{b_0^1}, \mathbf{f_0^3}, \mathbf{g} \in \mathbb{R}^3$ and vector $\mathbf{g}$ describes the ground

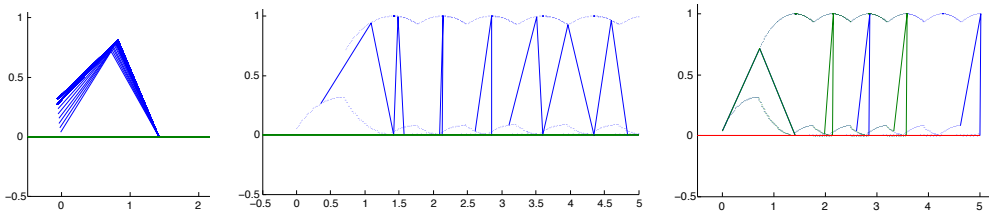
The function h provides zero if the position of the head is between the foot and the base, otherwise it provides one. This function supports the first assumption. The function c provides one if the foot has contact with the ground, otherwise it provides zero. Now the first steps for a reward function. The conditions to get a bad reward are described in I. and II., and to get a good reward are described in III., or formal

$$r(s_t, a_t) = \begin{cases} -1, & h(\mathbf{h_0^2}, \mathbf{b_0^1}, \mathbf{f_0^3}) \vee f_{0,y}^3 < g_y \vee f_{0,y}^3 > h_0^2 \\ 1, & c(\mathbf{f_0^3}, \mathbf{g}) \wedge \|\mathbf{f_0^3} - \mathbf{b_0^1}\| > 1 \wedge f_{0,x}^3 > b_{0,x}^1 \end{cases} \tag{12}$$

But this reward function is not sufficient for a proper walk. Actually, even nothing happens. So we have to show him the "learning way". One choice is to give him a reward for the traveled distance.

$$r(s_t, a_t) = \begin{cases} -1, & h(\mathbf{h_0^2}, \mathbf{b_0^1}, \mathbf{f_0^3}) \vee f_{0,y}^3 < g_y \vee f_{0,y}^3 > h_0^2 \\ 1000, & c(\mathbf{f_0^3}, \mathbf{g}) \wedge \|\mathbf{f_0^3} - \mathbf{b_0^1}\| > 1 \wedge f_{0,x}^3 > b_{0,x}^1 \\ \frac{100}{\|\mathbf{b_0^{1*}} - \mathbf{b_0^1}\|} \|\mathbf{f_0^3} - \mathbf{b_0^1}\|, & \text{otherwise} \end{cases} \tag{13}$$

Here, $\mathbf{b_0^{1*}}$ is the next estimated base and $\|\mathbf{b_0^{1*}} - \mathbf{b_0^1}\| = const$. This approach leads to an inappropriate learning. The result of this reward function is that the leg starts to shake in a particular situation without further move (compare Figure 4(a)). The algorithm has learned that he gets a reward and repeat the process to get again a reward. The question is how we can solve this problem? Our implemented solution is to give him the reward only the first time. The result of this solution is illustrated in Figure 4(b). More implementation details are described in the Appendix A.2.



(a) Tremor of the left leg with Equation (13)

(b) Resulting walk with the updated Equation (13)

(c) Competition between the best and the worst simple walker

4.5 Results

This Subsection shows the results of the Q-Learning algorithm. One important point is the convergence of the algorithm. We can adjust the discount factor $\gamma \in [0, 1)$ and the learning rate $\alpha \in (0, 1]$. Figure 5(a) shows the results with four different discount factors with a fixed learning rate $\alpha = 0.25$ and 500 steps per episode. The convergence with $\gamma = 0.99$ is the fastest one with the highest total reward of $\sim$ 16000 after 100 episodes. All depicted discount factors converge after 500 episodes.

The next step is to consider the convergence with the learning rate. Figure 5(b) shows the results with four different learning rates with $\gamma = 0.99$ and 500 steps per episode. The convergence with $\alpha = 0.25$ is the fastest one with the highest total reward of $\sim$ 16000 after 200 episodes. The best adjustment of the parameters is $\gamma = 0.99, \alpha = 0.25$. The standard deviation of the total reward of this adjustment is $\sim$ 300 or $\sim$ 1.875% of the total reward after 600 episodes and exactly depicted in Figure 4.

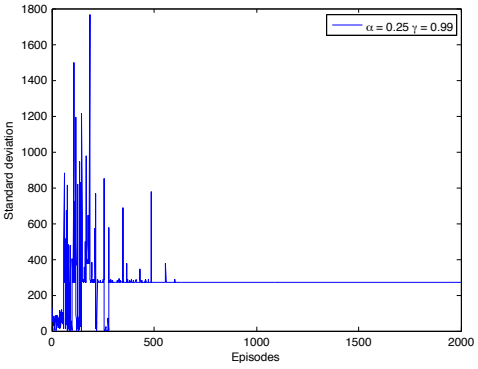
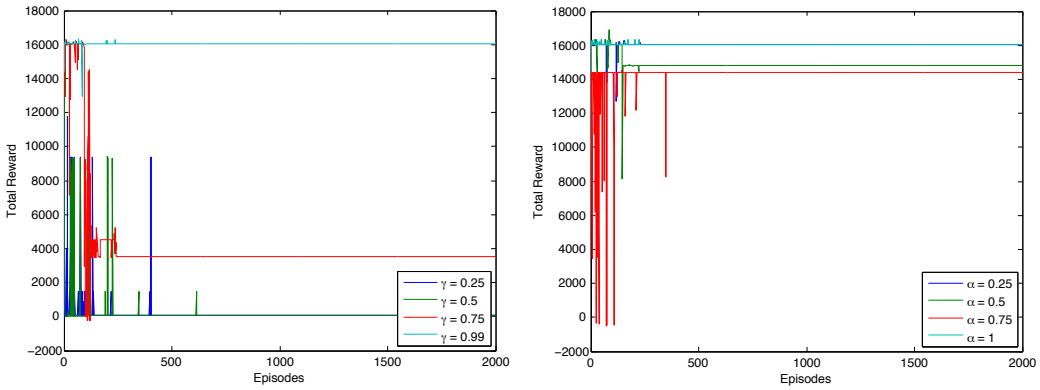


Figure 4: Standard deviation of 20 trials

Lastly, the comparison of a competition between the best and the worst simple walker, represented in Figure 4(c). The picture shows the two walkers at the beginning, after 250 steps and at the end after 500 steps. The result is that the best walker is, in the middle, 0.7 ahead of the worst walker and at the end 1.4. Based on these results, the final Section 5 discusses future tasks and gives a summary of the whole paper.



(a) Convergence with different discount factors

(b) Convergence with different learning rates

5 Conclusion

This paper provides an overview in Learning robot control. We define an abstract learning model and addressed the questions: "How is the implementation of a learning task and what should be learned?", as subdivision and classification of the learning task. Subsequently, we classified the learning task of the simple worker and constructed it with the well-known SCARA-Manipulator. Subsequently, the learning architecture, which contains the MDP and the Q-Learning algorithm, was explained and developed. A major component was the development of an appropriate reward function. Step by step, the problems were explained and the reward function was adapted. We have compared different simple walkers and have shown that the simple walker, with the highest reward, goes faster than others. In addition, the convergence of the learning algorithm was studied. The results show that the learning algorithm converges after 500 episodes and the best combination of the parameters is $\gamma = 0.99, \alpha = 0.25$ with a standard deviation of ~ 300 or $\sim 1.875\%$ of the total reward.
The next steps could be to build a simple walker for an inclined plane or include obstacle. Also, our assumptions can be replaced by a suitable physical model. In addition, the simple walker can get knee joints, torso or swinging arms. There exist a lot of opportunities to expand the system.

References

[1] Ruina A Coleman M. Garcia M, Chatterjee A. The simplest walking model: stability, complexity, and scaling. *Department of Theoretical and Applied Mechanics, Cornell University, Ithaca, NY 14853, USA*, 2:281–8, 1988.

[2] J. Morimoto, G. Cheng, C.G. Atkeson, and G. Zeglin. A simple reinforcement learning algorithm for biped walking. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 3, pages 3030–3035. IEEE, 2004.

[3] Y. Nakamura, M. Sato, and S. Ishii. Reinforcement learning for biped robot. In *Proceedings of the 2nd International Symposium on Adaptive Motion of Animals and Machines, pages ThP–II–5*, 2003.

[4] A.Y. Ng and S. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 663–670, 2000.

[5] D. Nguyen-Tuong and J. Peters. Model learning for robot control: a survey. *Cognitive processing*, pages 1–22, 2011.

[6] S. Schaal. Learning robot control. *The handbook of brain theory and neural networks,*, 2:983–987, 2002.

[7] S. Schaal. The new roboticstowards human-centered machines. *HFSP journal*, 1(2):115–126, 2007.

[8] S. Schaal and C. Atkeson. Learning control in robotics. *Robotics & Automation Magazine, IEEE*, 17(2):20–29, 2010.

[9] H.A. Simon. Why should machines learn. *Machine learning: An artificial intelligence approach*, 1:25–37, 1983.

A Appendix

A.1 Construction

For the simple walker we use the kinematic model from the SCARA-Manipulator with the DH-Parameter (shown in Figure 5)

Link i	θ_i	d_i	a_i	α_i
1	θ_1(variable)	0	l_1	0
2	θ_2(variable)	0	l_2	0

Figure 5: DH-Parameter for the SCARA-Manipulator

and with the simplification $l_1 = l_2 = 1$ we get the following kinematic model

$$T_1^3(\theta_1, \theta_2) = \begin{pmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 & \cos(\theta_1)+\cos(\theta_1+\theta_2) \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 & \sin(\theta_1)+\sin(\theta_1+\theta_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{14}$$

$$T_1^2(\theta_1, \theta_2) = \begin{pmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & \cos(\theta_1) \\ \sin(\theta_1) & \cos(\theta_1) & 0 & \sin(\theta_1) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{15}$$

A.2 MATLAB Code

This Subsections contains the important MATLAB functions of our Case Study.

```
%-------------------------------------------------------------------------
%--- This function calculates the Q table
%-------------------------------------------------------------------------
%--- INPUT
%   maxepisodes [N]     Maximum number of episodes
%   f                   Function of the ground
%   alpha       [N]     Learning rate
%   gamma       [N]     Discount factor
%
%--- OUTPUT
%   Q           [NxN]   Q table
%-------------------------------------------------------------------------
function [Q] = QLearning(maxepisodes,f,alpha,gamma)

maxsteps = 500;                 % maximum steps per episode
epsilon = 0.001;                % probability of a random action selection

%Instantiation
env = environment();

%For each s,a initialize the table entry Q(s,a) to zero
Q = zeros(length(env.s),length(env.a));

for i = 1 : maxepisodes

    [Q] = episode( maxsteps, Q , alpha , gamma , epsilon , env ,f);

    epsilon = epsilon * 0.99;

end
```

```
%-------------------------------------------------------------------------
%--- This function calculates a episode
%-------------------------------------------------------------------------
%--- INPUT
%   maxsteps      [N]     Maximum steps of this episode
%   Q             [NxN]   Q table
%   alpha         [N]     Learning rate
%   gamma         [N]     Discount factor
%   epsilon       [N]     Probability of a random action selection
%   env                   Environment
%   f                     Function of the ground
%
%--- OUTPUT
%   Q             [NxN]   Q table
%-------------------------------------------------------------------------
function [Q] = episode(maxsteps,Q,alpha,gamma,epsilon,env,f)

%s <= actual state
[env,s] = init(env,f);

%choose Action
[a,n] = epsilon_greedy_strategy(Q,s,epsilon,env);

%"do forever"
for i = 1 : maxsteps

    %and execute. r <= actual reward and s' <= new state
    [env,r,s_new] = action(env,a);

    %choose Action
    [a_new,n_new] = epsilon_greedy_strategy(Q,s_new,epsilon,env);

    %Update the Q talbe for nondeterministic environments
    Q(s,n) = Q(s,n) + alpha * ( r + gamma * Q(s_new,n_new) - Q(s,n));

    %Update the current variables
    s = s_new; a = a_new; n = n_new;

   %Basechange
    if s == env.s(5)
        [env.TM , env.q_current , env.base] = basischange(env.TM,
                                              env.q_current,env.base);
        env.next_base = [env.base(1) + sqrt(2) ; 0 ; 0];
    end

end

end

%-------------------------------------------------------------------------
%--- This function calculates max(Q(s_t+1,a_t+1))
%--- with the epsilon-greedy strategy
%-------------------------------------------------------------------------
function [ a ,n ] = epsilon_greedy_strategy( Q,s,epsilon,env )

    if(rand()>epsilon)
        %a = getBestAction(Q,s);
        [v n] = max(Q(s,:));
    else
        n = randint(1,1,length(env.s))+1;
    end
    a = env.a(:,n);
end
```

```
classdef environment

    properties
        TM
        q_init = [(3/4)*pi ; pi/2];
        q_current
        %States
        s = [1;2;3;4;5];
        %Actions
        a = [[-1 ; -1] [-1 ; 0] [-1 ; 1] [0 ; -1]
                [0 ; 0] [0 ; 1] [1 ; -1] [1 ; 0] [1 ; 1]];
        base
        next_base
        f
        x
    end

    methods

        %Initialization
        function [env,s] = init(env,f)
            env.TM = calcDHDirKin(walker(),env.q_init);
            env.q_current = env.q_init;
            env.base = [sqrt(2) ; f(sqrt(2)) ; 0];
            env.next_base = [2*sqrt(2) ; f(2*sqrt(2)) ; 0];
            env.f = f;
            s = env.s(1);
            env.x = 0;
        end

        %-------------------------------------------------------------------
        %-- INPUT
        %   a       [2x1]   Action
        %
        %-- OUTPUT
        %   r       [1]   Reward
        %   s       [1]   State = {standing : 1, forward : 2, backward : 3,
        %                          error : 4, newbase : 5}
        %-------------------------------------------------------------------
        function [env,r,s] = action(env,a)

            %Set new joint position
            env.q_current = [ env.q_current(1) + (1/180)*pi*a(1) ;
                        env.q_current(2) + (1/180)*pi*a(2)];

            %Calculate kinematic model
            env.TM = calcDHDirKin(walker(),env.q_current);

            %Calculate head and foot positions
            head = env.base + env.TM(1).T(1:3,4);
            foot = env.base + env.TM(2).T(1:3,4);

            %Condition for error
            if foot(2) > head(2) -0.5
                        || headOutOfInterval(head,env.base,foot,0.001)
                        || foot(2) < -0.001
                r = -1;
                s = env.s(4);
            %Condition for new base
            elseif contact(env.f,foot,0.001) == 1
                        && norm(foot-env.base) >= sqrt(2)/2
                        && foot(1) > env.base(1)
                r = 1000;
                s = env.s(5);
            else
```

```
            %Calculate reward
            r = (100/(2*sqrt(2)))*norm(env.next_base - foot);

            %Backward
            if r < 0
                s = env.s(3);
            %Standing
            elseif r == 0
                s = env.s(1);
            %Forward
            elseif r > 0
            %Reward just for the first time
                if env.x < foot(1)
                    env.x = foot(1);
                else
                    r = 0;
                end
                s = env.s(2);
            end

            end
        end

    end

end
```