Accelerating reinforcement learning on a robot by using subgoals in a hierarchical framework

Bart van Vliet Wouter Caarls Erik Schuitema Pieter Jonker

Delft University of Technology, Mekelweg 2, 2628CD Delft

Abstract

Reinforcement learning is a way to learn control tasks by trial and error. Even for simple motor control tasks, however, this can take a long time. We can speed up learning by using prior knowledge, but this is not always available, especially for an autonomous agent. One way to add limited prior knowledge is to use subgoals, defining points that the controller should aim for on the way to reaching the real goal. In this study, we use the MAXQ hierarchical framework to specify subgoals. This decreased the learning time by a factor two on a robot leg step-up task and we show that tests on a real robot give similar results. The worse end performance that is a result of the reduced solution space can be partially canceled out by hierarchical greedy execution. To our knowledge, this is the first time the MAXQ framework is applied to a real robot.

1 Introduction

As robots become more versatile, it becomes harder to design their controllers. Instead, we could use the reinforcement learning (RL) framework. In this framework, the robot learns to perform tasks by maximizing the *rewards* it receives by interacting with the environment. In order to optimize, the robot explores by performing random actions. In this way, RL algorithms can find the optimal solution without the need of knowledge about the system. However, learning a task with RL can take a long time. This is why researchers are looking for ways to speed up the learning process. This can be achieved by the use of prior knowledge. However, prior knowledge is not always available, especially for autonomous robots, and can lead to suboptimal solutions. To accomplish a faster learning process without using a lot of prior knowledge, while still aiming for an optimal solution, we look into the use of subgoals. These subgoals can be regarded as guiding waypoints, placed on the path to the goal of the task, and are either given or discovered by an algorithm, like L-Cut [11].

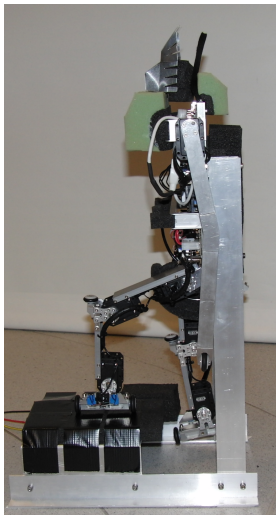


Figure 1: Leo, a 2D bipedal autonomous robot, in its testing harness. Made for walking, its task in this paper is to learn how to make a step onto the platform.

In this paper we test the use of subgoals on a task performed by a single leg of our robot LEO; see Fig. 1. We start by discussing related work in Sec. 2, theory in Sec. 3 and the experimental setup in Sec. 4. In Sec. 5.1 we show that subgoals speed up the learning process. Also, we show that the robot learns even faster when adding a region for each subgoal where it is allowed to be executed, and that hierarchical greedy execution indeed helps to bring the solution closer to the optimal solution. Then we show the influence of bad subgoal placement in Sec. 5.2. Lastly, we compare simulation results with results of the real robot in Sec. 5.3, which is, to our knowledge, the first time the hierarchical framework MAXQ [1] is applied to a real

robot. We present our conclusions in Sec. 6.

2 Related work

One way to implement subgoals in the RL framework is to use reward shaping, by giving extra reward for reaching a subgoal. In this way, the learner receives more frequent positive feedback on its path to the goal. In [6] a robot had to grasp a puck and drop it at home. By giving a reward for grasping the puck, the learning speed was improved. However, the robot can quickly discover it can gather many rewards by continuously grasping and dropping the puck, each time collecting the subgoal reward. Such behavior was shown in [8] and to avoid this, a penalty has to be given for such behavior. In [6] this was done by giving a penalty for dropping the puck away from home, but it is not always this straightforward to find a way to penalize the repetitive behavior. In [7] and [4], more general methods to avoid the cyclic movements have been presented, however, these methods require much prior knowledge.

Another way to solve the problem is by adding an extra state variable, which tells which subgoal has been reached. In [12] this is done by making a subtask for each subgoal. The root task learned to execute these subtasks in the correct order. Such a hierarchical approach is believed to be a good way to make RL suitable for complex tasks and more general frameworks have been developed later, of which Options [15] and MAXQ [1] are the most popular. These frameworks also add a region in state space for each subtask, where the subtask is allowed to be executed. This limits the choice for the root task, which improves the learning speed. Another advantage of these frameworks is that they can be extended with hierarchical greedy execution (HGE). When this is used, the root task is allowed to go to the next subgoal, while the previous subgoal has not been reached yet. This is beneficial when the subgoals are not placed on the optimal path, since in contrast to [12], the subgoals do not necessarily have to be visited. In this case, HGE can bring the solution closer (or make it equal) to the optimal solution.

3 Theory

In the reinforcement learning (RL, [14]) framework, an agent interacts with the environment. The agent performs actions and perceives the state and rewards. The goal of the agent is to maximize the sum of rewards it perceives. The interaction is modeled as a Markov decision process (MDP). It is a discrete time process where, each time step t, the agent perceives a state s_t and chooses an action a_t. The next time step, the agent ends up in a new state s_{t+1} with probability $T(s_t, a_t, s_{t+1})$ and receives a reward $r_t = R(s_t, a_t, s_{t+1})$. This reward function R is often constructed by the designer, while the state transition probability function T is generally unknown. The agent chooses actions based on the state it is in, according to: $a_t = \pi(s_t)$, where π is the *policy* of the agent (deterministic in our case). A policy should exploit its current knowledge, but should also explore to find better solutions. To do so, we chose to use an ϵ-greedy policy, which simply chooses a random action with probability ϵ and a greedy action otherwise. A greedy action is the one where the policy expects the highest sum of future rewards. In order to find the greedy action, the policy uses the action-value function $Q(s, a)$. It contains the expected sum of discounted rewards for each state-action pair when taking action a in state s and following policy π afterward:

$$Q(s,a) = E\left\{\sum_{i=1}^{i_{end}} \left[\gamma^{i-1} r_{t+i}\right] | s_t = s, a_t = a\right\} \tag{1}$$

where $\gamma \in [0, 1]$ is the time discount factor. The actual learning is done by *updating* the Q values such that they make successively better approximations of (1) (Q(λ)-Learning, [16]), for $k = 0...t$:

$$\begin{aligned} Q(s_{t-k}, a_{t-k}) &\leftarrow Q(s_{t-k}, a_{t-k}) + \alpha\gamma^k\lambda^k\delta \\ \delta &= r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \end{aligned} \tag{2}$$

with α the learning rate and λ the *eligibility trace discount factor*. Eligibility traces are used to speed up the learning process. In short, the trace contains a history of state-action pairs. Instead of only updating the Q values of the last state-action pair, the whole history can be updated. One can argue that state-action pairs further back in history are less responsible for the last rewards. Therefore, the update of a state-action pair of k steps ago, is discounted by λ^k, with trace discount factor $\lambda \in [0, 1]$. However, some state-action pairs can occur multiple times in the trace. To avoid such pairs getting an update that is too large, we limit the net trace discount factor to 1, as in [13] (*replacing traces*).

3.1 Hierarchical RL

At the basis of hierarchical reinforcement learning (HRL) lies the Semi-MDP (SMDP) framework [15]. An SMDP is equal to an MDP, except that it allows an action to take multiple time steps. The rewards collected after taking an action that took N time steps, have to be discounted by γ^N.

The SMDP framework allows us to break up a single task into smaller tasks: a hierarchy of subtasks. We used the MAXQ framework [1], which we extended in [9] to include eligibility traces: MAXQ-Q(λ). In this framework, each subtask has its own state space, action space and goal. A subtask can only be executed when the current state is inside the subtask's region. The subtask at the top of the hierarchy, the root task, tries to solve the original task, while the other subtasks only try to reach their own local goal. The root task usually is an SMDP; its actions consist of executing a subtask on a lower level. Once the root task has chosen a subtask to execute, this subtask is in control until it reaches its goal or gets outside of its region. Then the root task can select another subtask to execute. The subtasks on the lowest level of the hierarchy perform actions from the original task, which are called primitive actions. A MAXQ hierarchy with subgoals can be visualized in a graph as shown in Fig. 2.

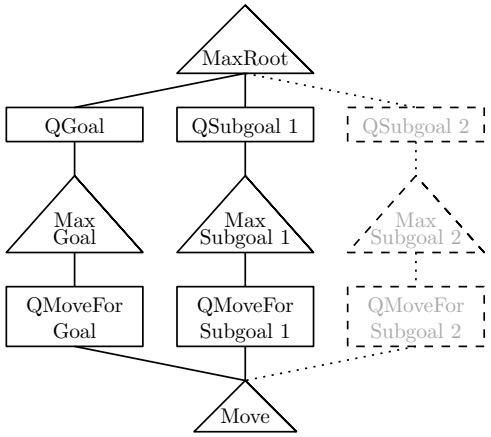


Figure 2: The MAXQ graph for one subgoal, each extra subgoal adds three nodes, like the dotted nodes.

To make subtasks reach their own goal, each subtask i has its own pseudo reward function $\tilde{R}_i(s, a, s')$. These pseudo rewards are not visible to other subtasks, they merely influence the policy of the subtask.

Once the subtasks have learned sufficiently, they will always reach their goal. However, these goals may not lie on the optimal path of the overall task, the problem that the root task has to solve. To get closer to the optimal path, we can use hierarchical greedy execution (HGE) as described in [1]. The idea is to interrupt a currently active subtask and give the root task the opportunity to choose another subtask to execute. In this way, a new subtask can be given control before the previous executed subtask could reach its goal.

However, interrupting a subtask influences the learning process. For a parent task, the Q-value of executing a subtask can only be updated when the subtask has completed, leading to slower learning. In addition, the subtask itself will learn slower when it is refrained from reaching its goal. Therefore, we should not use HGE from the beginning but enable it after the subtasks have sufficiently learned to reach their goals.

In [1], HGE is gradually enabled by interrupting the currently executed subtask after a decreasing amount of time steps on a fixed schedule. We chose to slowly increase the *chance* of an interruption, according to:

$$p_{interruption} = \left\{ \begin{array}{ll} \frac{\kappa}{(t_{hge}-t)+\kappa} & \text{for } t < t_{hge} \\ 1 & \text{for } t \geq t_{hge} \end{array} \right\}, \tag{3}$$

where κ is a parameter to modify the curvature and t_{hge} the time when HGE is fully active (i.e., when there is an interruption each time step). In case the root task has no other option but to choose the same subtask again, interrupting would not lead to a different choice, it would only clear the eligibility trace. Therefore, we do not interrupt in such a case. More information about the MAXQ-Q(λ) algorithm can be found in appendix A.

4 Experiment

To test the use of subgoals on a robot, we performed experiments on a simulated and a real robot leg. The task we gave it is a motor skill task without a straightforward solution.

4.1 Setup

The robot leg is part of a humanoid robot, Leo, of approximately half a meter tall [10]. For this research we built a stand (see Fig. 1) to which we fixed the torso of the robot; we only used the left leg. Each joint is actuated by a Dynamixel RX-28 motor (Robotis Co., Ltd., Seoul, Korea). Although this is a servo motor, we bypass the internal controller and operate it in voltage mode. To detect foot contact, there are two force sensors at the bottom of the foot, one at the toe and one at the heel. The simulation of the leg is made with

Joint	Angle	Velocity
Hip	0.0476 rad	6.67 rad/s
Knee	0.0769 rad	9.09 rad/s
Ankle	0.100 rad	11.1 rad/s

Figure 3: State space resolution per tiling

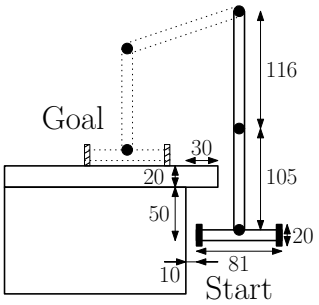


Figure 4: The task to perform: make a step up. Measures are in mm.

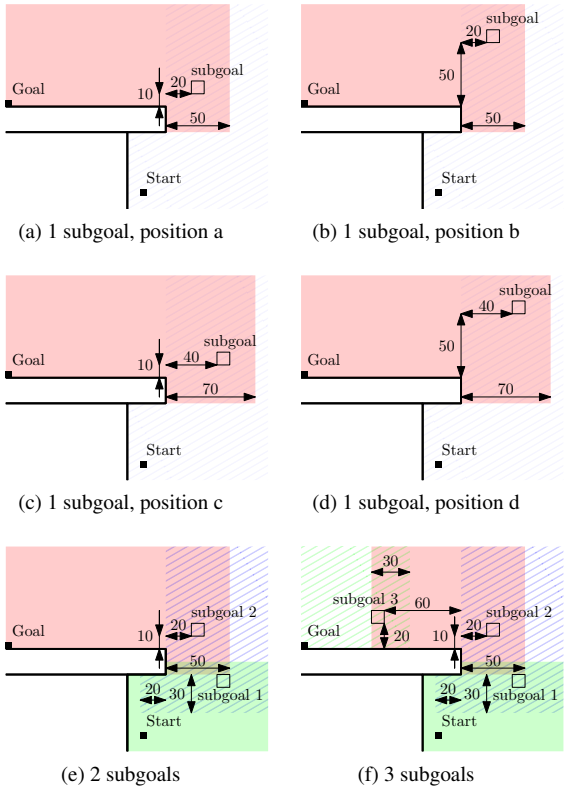


Figure 5: Placement of the subgoals (squares) and their regions where they are allowed to be executed (colored areas), measures are in mm.

ODE [3]. We used dimensions, masses and inertias equal to those of the real robot, and a simulation of the torque behavior of the motors [10].

The primitive actions consist of choosing between 5 voltages in the range $[-10.7, 10.7]$V for the hip and knee motors, resulting in 25 possible actions. The ankle motor was pre-programmed to keep the foot perpendicular to the lower leg. The sampling frequency was 20 Hz. The state of the robot consists of 7 variables: the foot contact and the angle plus angular velocity of each joint. The state was approximated by tile coding [14] with 16 tilings, the resolutions are listed in table 3. The contact variable is binary; at a certain force at the toe and heel, the variable is true.

4.2 Task

The task of the robot is to make a step up, as shown in Fig. 4. An episode starts with the leg in a straight down position and ends when the goal is reached, or after 60 (simulated) seconds, equal to 4500 time steps. The goal is reached when the foot makes contact at both the toe and heel, which is only possible in a position near the position shown in Fig.4.

We used a simple reward scheme, where a reward of 100 was given when the task was completed. Each time step a reward of -3.75 was given, to make the robot minimize the time to reach the goal. A subtask receives a pseudo reward of 100 when reaching its subgoal and a pseudo reward of -100 when leaving its region. A subgoal was defined as a small position area, irrespective of speed. We defined subgoals in this way to avoid suboptimal goal speed.

The Q values were initialized with random values between -1 and 1. In this way, a path that has timed out, left the region, or took too much time to reach the subgoal obtains lower values than the initial values, since these paths gathered many negative rewards. This results in more exploration at the early learning stage, as the agent will avoid these negative paths.

The learning parameters were not tweaked, since the initial choice was satisfying. We chose $\alpha = 0.25$, $\epsilon = 0.05$ and $\lambda = 0.69$. Since the task is episodic and there is a time penalty, we set $\gamma = 1$. For the HGE parameters, we used $t_{hge} = 300$ s and $\kappa = 5$ s.

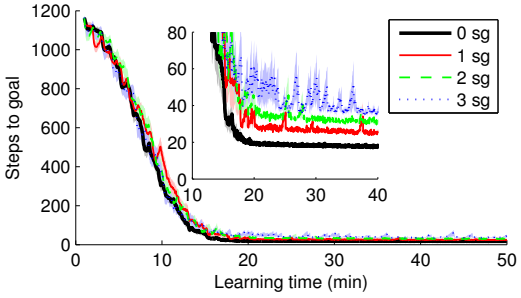


Figure 6: Learning curves with standard error bars at $p = 0.05$ ($n = 100$) for flat learning with 0 to 3 subgoals, and zoomed inset. A fast and slow learning stage can be distinguished.

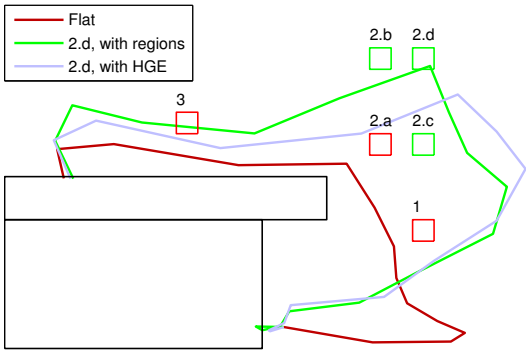


Figure 7: Subgoal positions and solutions found by different learning strategies: no subgoals, using subgoal 2.d with regions, and adding hierarchical greedy execution. The lines represent the position of the toe over time.

4.3 Experiments

To see the effect of using subgoals, we compared learning without subgoals to learning with one, two and three subgoals. The locations of the subgoals and their regions are shown in Fig. 5. Each subgoal is defined as an area of 10mm by 10mm, which the toe should reach. Since the subgoals usually do not lie on the optimal path, we will also take a look at the influence of the placement of a single subgoal, by placing it at different positions. Position a is the same as the single subgoal test explained above, while positions b, c and d are further away from the optimal path.

Apart from the simulation tests, we also performed tests on the real robot, to see if the simulation results match the results of the robot. Due to the long time real tests take, we only performed the tests for 0 and 2 subgoals, using HGE.

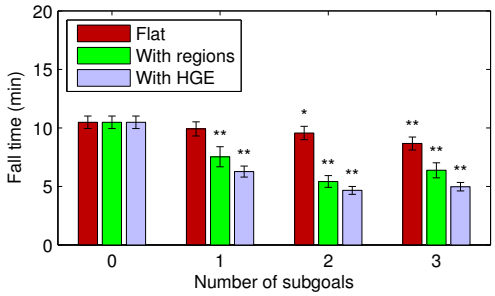
5 Results

The performance is defined as the inverse of the average number of steps it takes to reach the goal. Typically the performance increases drastically in a short time period, after which it slowly increases, as shown in Fig. 6. Therefore, we stopped simulating after 50 minutes and do not know the optimal performance. Consequently, we defined the end performance as the smoothed performance after 50 minutes. The fall time is defined as the time it takes until the 3-sample moving average of the number of steps reached 10% of the number of steps at the start of the learning. Each test is averaged over 100 runs. An example solution, found by flat learning without subgoals, shows that the chosen subgoals most likely lie off the optimal path, see Fig. 7 (lower line). On average, $6 * 10^4$ values were stored each run, with no significant dependency on the number of subgoals.

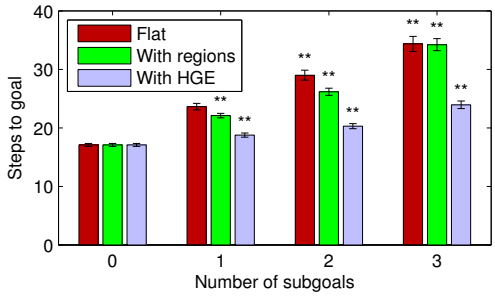
5.1 Number of subgoals

We compared the efficacy of learning with subgoals using three methods: flat learning using an extra state variable to indicate the current subgoal, MAXQ-Q(λ) (which adds regions), and MAXQ-Q(λ) with hierarchical greedy execution.

When using flat learning without regions in which the subgoals are allowed to be executed, the fall time decreases slightly when using more subgoals (left bars in each group of Fig. 8a). However, the end performance decreases dramatically (Fig. 8b). If we add regions using MAXQ-Q(λ) (middle bars), the fall times become considerably smaller. For 2 subgoals, it is halved compared to 0 subgoals. The small difference of fall time between 2 and 3 subgoals can be explained by the ease of learning the last part of the task; once the robot managed to reach subgoal 2, the path to the end goal was quickly discovered. However, the end performance is not much better than learning without regions. When using HGE (right bars), the fall times are slightly better than just using regions, but the end performance is drastically increased, especially when using more subgoals. By allowing the policy to "cut corners" (see Fig. 7, upper two lines), HGE approaches, but not quite reaches, the end performance of learning without subgoals.

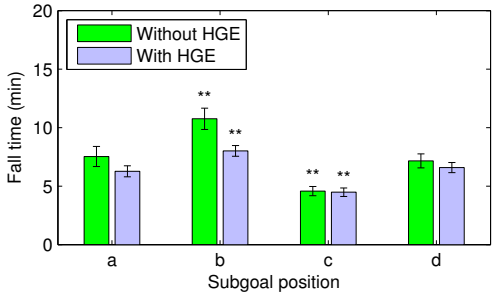


(a) Fall time. Except for flat learning, it is significantly decreased for more subgoals.

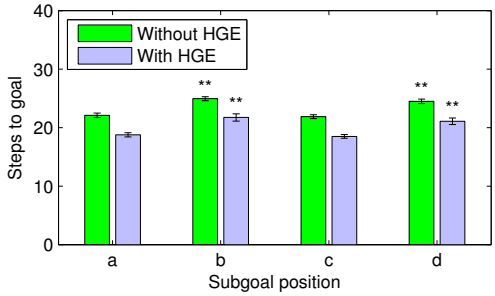


(b) Average steps to goal after 50 minutes. Performance decreases for more subgoals, but HGE reduces this effect.

Figure 8: Comparing different numbers of subgoals. Results differing significantly from 0 are marked ($\star$ for $p < 0.05$, $\star\star$ for $p < 0.01$; $n = 100$).



(a) Fall time. Position b learns significantly slower. This position also has the greatest improvement when using HGE. Position c learns faster.



(b) Average steps to goal after 50 minutes. Positions b and d perform worse than positions a and c.

Figure 9: Comparing subgoal positions, using MAXQ-Q(λ). Results differing significantly from position a are marked ($\star$ for $p < 0.05$, $\star\star$ for $p < 0.01$; $n = 100$).

5.2 Subgoal placement

The placement of subgoal 2 was tested with MAXQ-Q(λ), with and without HGE.

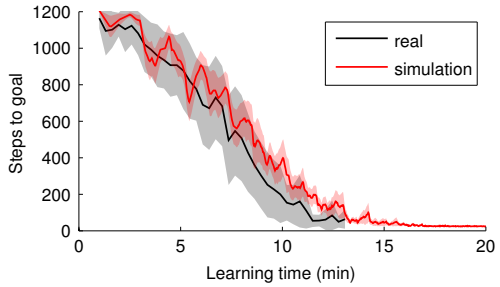
In Fig. 9a, we see that position b learns significantly slower than the original position. This may be expected, as it lies far off the optimal path (see Fig. 7). However, positions c and d don't show this effect. As these are off to the right, it is possible that placing the subgoal further away from the step leads to less collisions and therefore faster learning. If we look at Fig.9b, we see that the subgoals furthest away from the optimal path (b and d) have the worst end performance.

HGE has a positive effect on the fall time and end performance of all positions, but from this data we cannot draw conclusions about its dependence on the distance of the subgoal to the optimal path: the end performance seems to be improved by roughly the same amount in all cases.

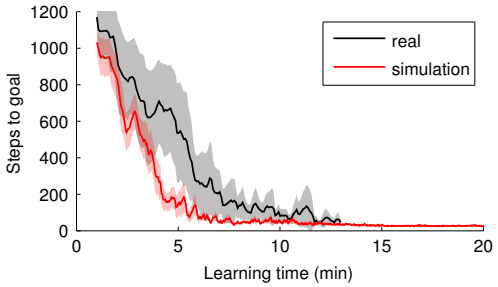
5.3 Tests on Leo

On the real robot, the tests were performed multiple times. After removing runs in which hardware problems occurred, we could average the tests over 12 runs. The hardware problems that occurred were a broken gearbox at the hip actuator and a toe force sensor that got stuck multiple times, which lead to the robot receiving a reward for kicking its behind. Due to a software bug, the runs were aborted after 13 minutes of learning time.

The learning curves of the simulation with the same settings as we used on the real robot, together with the average learning curves of the real robot, are plotted in Fig. 10a and 10b. When comparing simulation results with the results of the real robot, we can see that the learning curves without subgoals do not significantly differ. The curve of 2 subgoals, however, has a lower fall time in simulation than on the real robot. This can be explained by observations we made when the robot was learning: we noticed how the robot was



(a) Comparing real-world learning curve with a simulation for flat learning, using the same settings. The results are not significantly different.



(b) Comparing real-world learning curve with a simulation for 2 subgoals with HGE, using the same settings. The real robot learns slower than the simulation.

Figure 10: Learning curves with standard error bars at $p = 0.05$ ($n = 12$) verifying the simulation results using a real robot. The real world trials exhibit a larger variation than the simulation.

shaking when nearly reaching a subgoal, indicating it was hard to reach the subgoal. We suspect this was caused by sensor noise and backlash, which makes it harder to precisely reach the small subgoal area.

6 Conclusion

When learning motor skills with RL, the use of subgoals can lead to faster learning. However, this will usually lead to a worse performance after convergence. A hierarchical framework like MAXQ adds a region for each subgoal in which the subgoal is allowed to be executed. These regions decrease the state space, and we showed that this speeds up the learning process. Another advantage of MAXQ is the possibility to use HGE. In our experiments, this resulted in a policy that was much closer to the optimal policy. The real robot also benefits from subgoals, but performed worse than the simulation due to the small subgoal areas. We conclude that the use of subgoals is very useful for RL on a robot, as long as the subgoals lie reasonably close to the optimal path and are not too small in comparison with the robot's precision.

We believe that the performance may be further improved using the "all goals updating" method described in [2]. With this method, not only the subtask which is in control gets Q updates, but other subtasks are updated as well. In this way a subtask is learning while not in control, leading to faster convergence; novel techniques that converge with off-policy learning under function approximation have made this possible[5]. If the last subtask (the one that reaches the goal) is allowed to learn (and later execute) outside of its region, it could even reach the same end performance as flat learning.

References

[1] T.G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proc. 15th ICML*, pages 118–126, 1998.

[2] T.G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[3] Russel Smith et al. `http://ode.org/`. last visited in Oct 2010.

[4] A.D. Laud. *Theory and application of reward shaping in reinforcement learning*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[5] H. R. Maei, Cs. Szepesvari, S. Bhatnagar, and R. S. Sutton. Toward off-policy learning control with function approximation. In *Proc. 27th ICML*, 2010.

[6] M.J. Mataric. Reward functions for accelerated learning. In *Proc. 11th ICML*, volume 189, 1994.

[7] A.Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proc. 16th ICML*, pages 278–287, 1999.

[8] J. Randløv. *Solving complex problems with reinforcement learning*. PhD thesis, University of Copenhagen, 2001.

[9] Erik Schuitema. Hierarchical reinforcement learning. Master's thesis, Delft University of Technology, 2006.

[10] Erik Schuitema, Martijn Wisse, Thijs Ramakers, and Pieter Jonker. The design of LEO: a 2D bipedal walking robot for online autonomous reinforcement learning. In *Proc. IROS*, 2010.

[11] Ö. Şimşek, A.P. Wolfe, and A.G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proc. 22nd ICML*, pages 816–823. ACM New York, NY, USA, 2005.

[12] S.P. Singh. Transfer of learning across compositions of sequential tasks. In *Proc. 8th ICML*, pages 348–352. Morgan Kaufmann, 1991.

[13] S.P. Singh and R.S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1):123–158, 1996.
[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
[15] R.S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.
[16] C. Watkins. *Learning from Delayed Rewards*. Phd thesis, King's College, 1989.

A The MAXQ-Q(λ) algorithm

In MAXQ, the value function of subtask i is defined as

$$Q_i(s,a) = V(s,a) + C_i(s,a) \tag{4}$$

where $C_i(s,a)$ is the completion function for subtask i, defined as the expected discounted cumulative reward of completing subtask i after invoking subtask a in state s. $V(s,a)$ is the projected value function for action a in state s, defined as

$$V(s,a) = \begin{cases} Q_a(s,\pi_i(s)) & \text{if } a \text{ is a subtask} \\ V_p(s,a) & \text{if } a \text{ is primitive} \end{cases} \tag{5}$$

Since the actual policy of a subtask is based on rewards plus pseudo rewards, each subtask also has a pseudo value function $\tilde{C}_i(s,a)$, which predicts the expected pseudo plus real reward. Thus, the greedy action of the policy for subtask i is:

$$\pi_{i,greedy}(s) = \arg\max_{a'} \left[\tilde{C}_i(s,a') + V(s,a')\right] \tag{6}$$

The eligibility trace is stored in $e_i(s,a)$, which contains the discount factor for each state-action pair. A subtask terminates when it reached its goal or when it got outside of its region. It also has to terminate when its parent has to terminate. In all those cases $T_i(s)$ is true. The pseudo code of the MAXQ-Q(λ) algorithm for the tabular case, including HGE, can be found below.

function MAXQ-Q(λ)(MaxNode i, State s)
if i is a primitive MaxNode **then**
  execute primitive action i, receive reward r, and observe result state s'
  $V_p(s,i) \leftarrow (1-\alpha_i)\cdot V_p(s,i) + \alpha_i \cdot r$
  return 1
else
  $count \leftarrow 0$, initialize $e_i(s,a) = 0$ for all s, a {Reset trace}
  choose an action a according to the current policy $\pi_i(s)$
  while $T_i(s)$ is false **do**
    $N \leftarrow$ MAXQ-Q(λ)(a,s) {Recursive call}
    observe result state s', reward $r = R_i(s,a,s')$ and pseudo-reward $\tilde{r} = \tilde{R}_i(s,a,s')$
    choose an action a' according to the current policy $\pi_i(s')$
    $a^* \leftarrow \arg\max_b[\tilde{C}_i(s',b) + V(s',b)]$
    if s' is terminal and absorbing **then**
      $\tilde{\delta} \leftarrow \gamma^N[r+\tilde{r}] - \tilde{C}_i(s,a)$, $\delta \leftarrow \gamma^N r - C_i(s,a)$
    else
      $\tilde{\delta} \leftarrow \gamma^N[r+\tilde{r}+\tilde{C}_i(s',a^*)+V(s',a^*)] - \tilde{C}_i(s,a)$, $\delta \leftarrow \gamma^N[r+C_i(s',a^*)+V(s',a^*)] - C_i(s,a)$
    $e_i(s,a) \leftarrow 1$ {Replacing trace}
    for all s,a with $e_i(s,a) > e_{min}$ **do** {Limit trace length}
      $\tilde{C}_i(s,a) \leftarrow \tilde{C}_i(s,a) + \alpha_i\tilde{\delta}e_i(s,a)$, $C_i(s,a) \leftarrow C_i(s,a) + \alpha_i\delta e_i(s,a)$
      if $a' = a^*$ **then** {Greedy action}
        $e_i(s,a) \leftarrow \gamma^N\lambda(i)e_i(s,a)$ {Decay trace}
      else
        $e_i(s,a) \leftarrow 0$ {Cut off trace on exploration}
    $count \leftarrow count + N$, $s \leftarrow s'$; $a \leftarrow a'$
    if HGE **and** $i \neq$ rootnode **and** rootnode has other subtasks to choose than i **then**
      break
return $count$