Hierarchical Learning of Robot Skills by Reinforcement

Long-Ji Lin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
e-mail: ljl@cs.cmu.edu

***Abstract*—**

Because learning from delayed reinforcement is hard and generally a slow process, a straightforward application of reinforcement learning would be generally impractical for complex problems in which state spaces are huge and reinforcement signals are sparse. This paper shows how we can make reinforcement learning practical for complex problems by introducing hierarchical learning: The agent at first learns elementary skills for solving elementary problems. To learn a new skill for solving a complex problem later on, the agent can ignore the low-level details and focus on the problem of coordinating the elementary skills it has developed. A physically-realistic mobile robot simulator is used here to demonstrate the success and importance of hierarchical learning. For fast learning, artificial neural networks are used to generalize experiences, and a teaching technique is employed to save many learning trials of the simulated robot.

I. Introduction

Reinforcement learning is an unsupervised learning method for sequential decision making. In this learning paradigm, the learning agent receives a scalar performance feedback called *reinforcement* or *payoff* after each action execution. The objective of learning is to construct a *control policy* so as to maximize the *discounted cumulative reinforcement* in the future or, for short, *utility*:

$$V_t = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k}$$

where V_t is the discounted cumulative reinforcement starting from time t throughout the future, r_t is the reinforcement received after the transition from time t to $t+1$, and $0 \leq \gamma \leq 1$ is a *discount factor*, which adjusts the importance of long-term consequences of actions.

Q-learning is a widely-used reinforcement learning method [12, 5, 4]. The idea of Q-learning is to construct an evaluation function called *Q-function*:

$$Q(\mathit{state}, \mathit{action}) \rightarrow \mathit{utility}$$

The Q-function is used to predict the discounted cumulative reinforcement (i.e., utility) for each state-action pair given that the agent is in that state and executes that action. Given an optimal Q-function and a state x, the optimal control policy is simply to choose the action a for which $Q(x,a)$ is maximal over all actions.

For generalization, the Q-function can be represented by multi-layer neural networks called ***Q-nets***, and incrementally trained by a combination of the error back-propagation algorithm and *temporal difference* (TD) methods [8]. Given a state transition (x, a, y, r) meaning that an action a in response to a state x results in a new state y and reinforcement r, the Q-nets can be adjusted as follows:

1. $u \leftarrow Q(x,a)$; $u' \leftarrow r + \gamma \cdot Max\{Q(y,k) | k \in actions\}$;
2. Adjust the network corresponding to $Q(x,a)$ by back-propagating the error $(u' - u)$ through it;

Given a sequence of state transitions, the agent simply applies the above learning procedure to every two successive states along the sequence. Note that the above procedure uses the simplest form of TD methods, TD(0). This research in fact uses TD($\lambda > 0$), which was found more effective than TD(0). See [3, 4] for further details.

Reinforcement learning has been successfully applied solving nontrivial learning problems [5, 10]. A serious problem, however, is that learning from sparse reinforcement signals is hard and generally a slow process. There are several ways to speed up reinforcement learning, such as generalization [1, 3, 4, 6, 10], using action models [9, 5], teaching [5, 3], etc.

This paper describes another way to speed up reinforcement learning; that is, *hierarchical learning*. It is well known that no planning system can scale up well without *hierarchical planning*. Similarly, no learning agent can successfully cope with the complexity of the real world without hierarchical learning of skills. This paper dis-

0-7803-0999-5/93/$03.00 ©1993 IEEE

cusses a way of doing hierarchical reinforcement learning, and presents a case study using a physically-realistic mobile robot simulator. The task of the robot is to learn a control policy for finding a battery charger and electrically connecting to it. The task is so complex that without using hierarchical learning, the robot was unable to learn a good control policy within a reasonable time. By decomposing the task into subtasks and solving them separately, the robot was able to solve the learning task effectively.

II. Hierarchical Reinforcement Learning

The primary virtue of hierarchical learning/planning is the great reduction in the complexity of problem solving. The essential idea is based on ***problem decomposition*** and the use of ***abstraction***. To efficiently solve a complex learning problem, humans often decompose the problem into simpler subproblems. Once we have learned skills to solve each subproblem, solving the original problem becomes easier, because we now can concentrate on the high-level features of the problem and ignore the low-level details, which presumably will be filled in when the previously learned skills are actually applied. This idea can generalize to multiple levels of abstraction. At each level, we focus on a different level of details. Another virtue of hierarchical learning is that subproblems shared among high-level problems need be solved just once and their solutions can be reused. Basically three steps are involved in hierarchical reinforcement learning:

1. **Task Decomposition.** A complex task is decomposed into multiple elementary tasks. The original complex task is thus reduced to the task of integrating the solutions to the elementary tasks to form the solution to the original task. Note that task decomposition involves designing a reward function for each elementary task.
2. **Learning elementary skills.** An elementary skill needs to be learned to solve each elementary task. Here Q-learning can be used, and each elementary skill corresponds to a Q-function: $Q(state, action) \rightarrow utility$.
3. **Learning a high-level skill.** A high-level skill for coordinating the elementary skills needs to be learned in order to solve the original task. Learning a high-level skill is conceptually similar to learning an elementary skill. Again, Q-learning can be used, and the high-level skill corresponds to a Q-function: $Q(state, skill) \rightarrow utility$.

A good task decomposition often demands either lots of domain knowledge or lots of experience of solving related problems. This research assumes that task decomposition is performed by human designers. A formal model of hierarchical reinforcement learning can be found in [4], in which a complexity analysis for deterministic and discrete domains is also presented. The analysis shows a theoretical complexity reduction by using hierarchical learning.

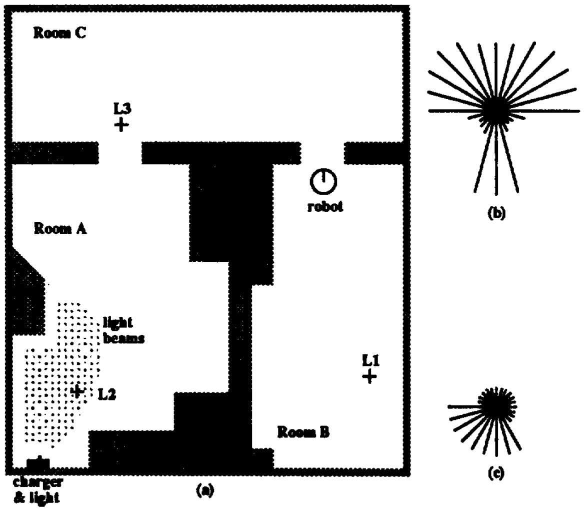

Figure 1: (a) A simulated robot and its environment. The radius of the robot is 12 inches. The size of the environment is 360 inches by 420 inches. (b) Sonar readings received at L3. (c) Light readings at L2.

III. The Domain: A Simulated Mobile Robot

The domain used in this research was a mobile robot simulator (Figure 1), which was intended to mimic a real robot called Hero and a real office environment. The simulated environment consists of three rooms, A, B and C, and two narrow doorways connecting Room C to Rooms A and B. A battery charger is located in the left-bottom corner of Room A. A light is placed on top of the charger so that the robot can distinguish the charger from other obstacles. The robot has 6 actions: turn ±15°, turn ±60°, and move ±12 inches. It has a sonar sensor and a light intensity sensor mounted on its top. Both sensors can rotate and collect 24 readings (separated by 15 degrees) per rotation. The sonar sensor returns a distance reading between 0 and 127 inches, with a resolution of 1 inch. In addition, the robot has a compass and four collision sensors placed around its body. To be realistic, about 8% control error is added to the robot's actuators and about 10% sensing error added to the sensors.

The robot's task is to learn a control policy for finding the battery charger and connecting to it. It may start with any orientation and from any location in the environment. The robot begins with no knowledge about its effectors, sensors, and the environment. The reward function for the task is 100 when the robot successfully connects to the charger and 0 otherwise. This battery recharging task is hard for several reasons: (1) The state space is continuous and large, (2) the optimal control policy is complex, and (3) the reward delay is long. It takes the robot approximately 100 steps to reach the battery charger starting from Location L1 (Figure 1).

IV. Task Decomposition

In this work, the battery recharging task is decomposed into four elementary tasks:

- following walls while keeping them on the robot's right/left hand side (WFR/WFL),
- passing a door starting from places near the door (DP), and
- docking on the charger starting from the light area where the robot can sense the light (DK).

For instance, to connect to the battery charger from Location L1 (Figure 1), the robot simply executes WFR to get to the vicinity of the right door, executes DP to get out of Room B, executes WFL to reach the left door, executes DP again to get into Room A, executes WFR again to get close to the charger, and finally executes DK to get connected to the battery charger.

To define an elementary task, we need to specify:

- a reward function specifying the goal, and
- an *application space* specifying the portion of the state space for which a skill will be trained.

For example, the reward function used here for docking is: -10 if collision occurs, 100 if the robot successfully connects to the charger, and 0 otherwise. Note that the application space of a skill can be thought of as the *precondition* of the skill. For example, here I want to train a robot to dock on the charger as long as the robot can detect the light source on top of the charger, therefore I define the application space of docking to roughly correspond to the light area. Similarly, I define the application space of door passing to roughly correspond to the locations where a door opening can be detected. The application space for wall following is simply the whole state space. Since a skill is only trained for its own application space, if a skill is executed when the agent is not in the application space, the agent may never achieve the goal of the skill. For instance, a door passing skill is not expected to achieve its goal, if there is no door nearby.

V. Learning Elementary Skills and Teaching

Here let us consider in detail how the robot could learn a docking skill; other elementary skills can be learned similarly. The state representation for the docking task includes 24 real-valued sonar readings, 24 binary light readings, and 4 binary collision readings. The learning process consisted of many trials. In each trial, the robot started with a random orientation and a random position mostly in the light area (i.e., the application space of docking). For the purpose of exploration, the robot chose actions randomly, but preferred to choose actions having high utilities. Each trial ended when the robot achieved the goal or else after 30 steps. The experience of each trial was recorded, and the learning procedure mentioned in Section I.was applied to train the Q-function, which was represented by multi-layer neural networks.

In the first attempt, the performance of the robot was found poor: In 2 out of 7 runs, the robot was unable to dock after 300 learning trials. In the other 5 runs, the robot took 250 trials on average to learn an acceptable docking skill. The poor performance was due to two difficulties. First, reinforcement learning agents learn by trial and error. If the agent is lucky, it may learn quickly. If it is not lucky, the trial and error process may take a long time. Second, docking and obstacle avoidance are kind of contradictory; docking requires the robot to collide with the charger. Once the robot learns to avoid obstacles, it may never learn to dock— it is stuck in a local optimum.

A possible solution to both difficulties is to teach the robot. By teaching, we take control over the robot, and demonstrate a few times how the target task can be solved from different initial states. The demonstrated solutions are used to train the Q-function by using the learning procedure described in Section I. This teaching technique was found very effective. With just 10 teaching examples, the robot generally could learn a very good docking skill within 150 trials. Using this technique, the robot could also learned the door passing and wall following skills effectively. See [4] for details.

VI. Learning High-Level Skills

During learning the elementary skills, the robot assumes that all primitive actions terminate automatically. Thus, if the robot wants to see the effect of an action, it simply executes that action and waits to see the outcome. If each elementary skill has a termination condition and can terminate just like primitive actions, then learning a high-level skill is just like learning an elementary skill, and the learning technique used to learn the docking skill can be directly applied to learning a high-level skill for the battery recharging task.

A. Never-ending Skills

Unfortunately, in practice there is often a complication: ***Once activated, a skill may not terminate automatically.*** For example, a wall following skill has no termination condition at all and can last forever once it is activated. Even though the door passing skill has a termination condition (i.e., getting into another room), it also may never stop under certain circumstances. For example, the robot's sensors may mistakenly detect a non-existing door (due to noise), causing an inappropriate activation of the skill. Or the skill may not have been trained well enough to deal with every situation that the skill is supposed to take care of. Under either circumstance, the door passing skill may never achieve its goal and terminate.

Since skills may not terminate on their own, the agent has to decide ***when*** to switch skills as well as ***which skill*** to apply next. Because optimal timings for switching skills may occur at any time, the agent must check to see which skill to choose frequently such that optimal timings will not be missed. For example, while following walls, the robot needs to check frequently to see if a door opening suddenly comes into sight such that it will not miss the doorway it wants to enter. Another reason that frequent checks may be necessary is this: The robot may mistakenly detect a non-existing door opening and activate the door passing skill. If the robot checks frequently, it may quickly find that the skill is no longer appropriate and stop it right away.

This work took an extreme— the decision about which skill to apply is made by the robot on ***each primitive step.*** In other words, the robot checks to see which skill is most appropriate; applies it for one single step; determines the next appropriate skill; applies that skill for another step; and so on. In the worst case, the robot needs a high-level control policy to decide the optimal skill for each state. In other words, the agent needs to learn an evaluation function of $Q(\boldsymbol{state}, \boldsymbol{skill})$. If hierarchical learning is not used, the robot has to learn $Q(state, action)$. Is $Q(\boldsymbol{state}, \boldsymbol{skill})$ much easier to learn than $Q(state, action)$? The answer seems positive, because the former Q-function is less complex than the latter, which is easily seen by noting that the robot need switch actions much more frequently than switch skills.

B. Effective Exploration

To learn $Q(\boldsymbol{state}, \boldsymbol{skill})$ effectively, the agent needs a good strategy to explore the state space, which may be large. Fortunately, the agent has knowledge (i.e., Q-functions) about the elementary skills it has learned previously. It can use this knowledge for effective exploration. Two exploration rules are discussed below. Although the discussion of both rules is specific to the battery recharging task, they seem quite general and applicable to many other domains as well.

Applicability Rule. *Choose only applicable skills.* A skill is applicable only if its application space covers the current state. But since the robot is not provided with a procedure for determining the application spaces, how can the robot decide if a skill is applicable? For the wall following skill, the answer is quite simple— they are always applicable. Whether the door passing skill is applicable can be determined by examining the Q-function (called Q_{dp}) that defines the door passing skill. There are basically two circumstances where the output (i.e., Q-value) of Q_{dp} is low: either the robot is far away from any door or the skill is unable to handle the given situation. In either case, it is inappropriate to apply the skill. But, what Q-values should be considered low or high? In this work, a simple thresholding was used. A proper threshold could be learned, but here it was chosen off-line after the skill was learned. Similarly, by examining the Q-function of the docking skill, we can determine whether it makes sense to apply the skill— low Q-values often correspond to situations where the robot does not detect the light on top of the charger. Note that because it is fine to activate an inapplicable skill, the selection of the threshold values was not found critical as long as they were not too high to rule out the optimal skill.

Persistence Rule: *Do not switch skills unless some significant change to the robot's situation has occurred.* The following two changes can be considered significant: (1) The goal of the docking or door passing skill is achieved. (2) A previously inapplicable skill becomes applicable (for instance, a door opening suddenly comes into sight, making the door passing skill become applicable). The basic idea behind this rule is the following: To make good progress, the robot should avoid switching skills frequently. Imagine that the robot is close to a door. The door passing skill and both wall following skills are all applicable at the moment. The robot can choose any of them for experimentation, but it should either keep following the wall until another doorway comes into sight or dedicate itself to door passing until the door is completely passed. If the robot keeps changing skills, it will often end up being stuck in the same place [4].

VII. Experimental Results

This section reports two simulation experiments. In the first experiment, hierarchical learning was not used; the robot learned a monolithic Q-function, $Q(state, action)$, for the battery recharging task. The second experiment is about learning a high-level control policy, $Q(\boldsymbol{state}, \boldsymbol{skill})$, for battery recharging. The elementary skills (including wall following, door passing, and docking) had been trained beforehand. Again, the learning process consisted of many trials. In each trial, the robot started with a random orientation and location in the 3-room environment, and was allowed to take 200 primitive steps at a maximum. See [4] for detailed descriptions of the exploration rules, the input representations, and the experiments.

Non-hierarchical learning. The robot was provided with 10 teaching examples to start with. The same learning technique used to train the docking skill was employed

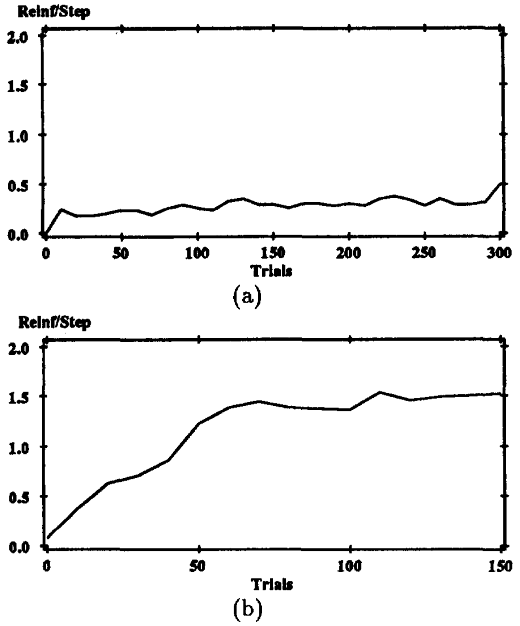

Figure 2: The learning curves of the robot (a) without and (b) with hierarchical learning. The Y axis indicates the average reinforcement per primitive step, and the X axis indicates the number of trials attempted so far.

to train the recharging skill. Figure 2.a shows the mean performance of the robot over 7 runs. After 300 learning trials, the robot was generally unable to accomplish the task starting from Rooms B and C. Besides, it collided with obstacles very often (more than 5% of the time). In this experiment, the robot was not punished when it collided with obstacles. Slightly better performance was observed when the robot was punished for collisions, but it still performed poorly. The robot in fact never learned a systematic way to carry out the recharging task within 300 trials no matter what reward function was used.

Hierarchical learning. The robot was not provided with any teaching example except those for training the elementary skills. The two exploration rules described in Section B.were used. Figure 2.b shows the robot's mean performance over 7 runs. Within 20 trials, the robot generally had developed a good high-level skill for connecting to the charger from most of locations in Rooms A and C. A nearly optimal skill for all situations was generally learned within 60 trials. The learned skills could adapt to small unexpected environmental changes without additional training [4]. (The robot's performance was poor when the Persistence Rule was not used [4].)

Although the numbers of teaching steps (about 500) involved in both experiments are approximately equal, it is unfair to make a point-by-point comparison between the two learning curves in Figure 2, because in the case of hierarchical learning, the robot had already taken many action executions to learn the elementary skills before it started to learn the high-level skill. However, when both experiments ended, the robot executed approximately the same number of actions (about 40000) in both cases (taking into account the action executions during learning the elementary skills). Given a similar number of teaching steps and a similar number of experimental actions, the one using hierarchical learning was apparently superior to the one without hierarchical learning.

VIII. Discussion

Why did the robot perform significantly better with hierarchical learning than without it? Two explanations:

Better state representations. Good input representations are crucial to efficient learning, because better representations support better generalization and thus faster learning. One criterion for being a good representation is that it includes only the information relevant to the task. When the recharging task was not decomposed, the robot had to use a large state representation that described the robot state in sufficient detail, even though not all of the detailed information would be needed at the same time during carrying out the task. Such a state representation is less desirable. When the task was decomposed, the robot only needed to use a small state representation during solving each subtask, resulting in efficient learning. For example, the light readings, which were needed for docking, would not be needed for door passing and thus could be excluded from the state representation for the door passing skill. The high-level skill also used a small state representation, which carried abstract information derived from the low-level sensory inputs. See [4] for details.

Easier-to-learn Q-nets. Training multiple simple networks is often easier than training a single complex network [11]. Without task decomposition, the optimal Q-net for the recharging task is so complex that it could hardly be trained to the desired accuracy at all. With task decomposition, the optimal Q-nets for the elementary and high-level tasks are simple and can be learned rapidly.

Being able to do hierarchical learning, the robot has been provided with additional domain-specific knowledge, including a proper selection of elementary tasks and a proper selection of a reward function and an application space for each elementary task. It seems inevitable that we need domain knowledge to buy great learning speed.

IX. Related Work

Mahadevan and Connell [6] presented a box-pushing robot, which learned elementary skills from delayed rewards. Their approach was based on Q-learning. In their case, the skill coordination policy was not learned but hard-wired by humans beforehand.

Lewis et al. [2] studied a six legged insect robot, which successfully learned to walk. The robot was controlled by neural networks with weights determined by genetic algorithms. The robot developed its walking skill in two stages: It developed first an oscillation behavior for each of the six legs, and then a walking behavior that coordinates the six oscillation behaviors. The robot might not have developed a successful walking skill without dividing the learning process into such two stages. Except that we used different optimization techniques (genetic algorithms vs. reinforcement learning), we have shared the same idea of hierarchical learning.

Singh [7] described an efficient way to construct a complex Q-function from elementary Q-functions. However, to use his technique requires a few strong assumptions to be met. For example, it could not handle skills that do not have termination conditions (e.g., wall following). On the other hand, Singh has demonstrated for a simple case that his architecture could learn a new elementary skill that was not explicitly specified. This suggests a possible way for robots to discover on their own a proper way to decompose a task.

X. Conclusion

This paper has demonstrated that it is possible for robots to acquire complex skills by reinforcement learning. However, it would be impractical to apply reinforcement learning in a straightforward manner, because it would be too slow for solving very complex tasks. By introducing hierarchical learning, the complexity of reinforcement learning can be greatly reduced. Indeed, the simulation experiments have demonstrated that the robot with hierarchical learning was able to solve a complex problem, which otherwise was hardly solvable within a reasonable time.

Learning to choose optimal elementary skills is conceptually similar to learning to choose optimal primitive actions, but there is one complication: A skill may not terminate automatically once it is activated. This paper discussed two effective exploration rules for learning to coordinate skills that may not have a termination condition. By taking advantage of previously acquired knowledge (i.e., the Q-functions for the elementary skills), the exploration rules allowed the simulated robot to acquire a nearly optimal high-level skill effectively.

Hierarchical learning relies on a careful decomposition of tasks. To decompose a task properly often requires some domain-specific knowledge, which may come from humans or the agent's own experiences of solving related problems before. In this paper, task decomposition was done by a human. A great challenge would be to let the robot discover a proper task decomposition on its own.

Acknowledgments. I thank Tom Mitchell, Sebastian Thrun, Ryusuke Masuoka, and Chi-Ping Tsang for their valuable comments on a draft of this paper.

References

[1] D. Chapman and L.P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of IJCAI-91*, pages 726–731, 1991.

[2] M.A. Lewis, A.H. Fagg, and A. Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 2618–2623, 1992.

[3] Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of AAAI-91*, pages 781–786, 1991.

[4] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, School of Computer Science, 1993.

[5] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[6] S. Mahadevan and J. Connell. Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Proceedings of the Eight International Workshop on Machine Learning*, pages 328–332, Evanston, Illinois, 1991. Morgan Kaufmann.

[7] S.P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[8] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[9] R.S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, Austin, Texas, 1990.

[10] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[11] A. Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1:39–46, 1989.

[12] C.J.C.H. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, 1989.