An Adaptive Architecture for Modular Q-Learning

Takayuki Kohri and Kei Matsubayashi* and Mario Tokoro†
Department of Computer Science,
Faculty of Science and Technology, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, JAPAN

Abstract

Reinforcement learning is a technique to learn suitable action policies that maximize utility, via the clue of reinforcement signals: reward or punishment. Q-learning, a widely used reinforcement learning method, has been analyzed in much research on autonomous agents. However, as the size of the problem space increases, agents need more computational resources and require more time to learn appropriate policies. Whitehead proposed an architecture called *modular Q-learning*, that decomposes the whole problem space into smaller subproblem spaces, and distributes them among multiple modules. Thus, each module takes charge of part of the whole problem.

In modular Q-learning, however, human designers have to decompose the problem space, and create a suitable set of modules manually. Agents with such a fixed module architecture cannot adapt themselves to dynamic environments. Here, we propose a new architecture for reinforcement learning called *AMQL (Automatic Modular Q-Learning)*, that enables agents to obtain a suitable set of modules by themselves using a selection method.

Through experiments, we show that agents can automatically obtain suitable modules to gain a reward. Furthermore, we show that agents can adapt themselves to dynamic environments efficiently, through reconstructing modules.

1 Introduction

Reinforcement learning is one of the techniques by which agents can learn suitable action policies that maximize their utilities, via reinforcement signals of reward or punishment. There have been various investigations of autonomous agents using this learning technique. However, reinforcement learning with monolithic methods lacks scalability, since agents require much more time to learn suitable action policies when dealing with more complex and larger problems. Additionally, agents use more computational resources to memorize the utilities of all states and actions.

To tackle this problem, some methods that decompose the problem space have been suggested. Whitehead proposed an architecture called *modular Q-learning* [1993] that decomposes the whole problem space into smaller subproblem spaces and distributes them among multiple modules. The goals of multiple goal problems are decomposed into subgoals, which are then distributed as subgoals among multiple modules. Since each module learns only to accomplish its own goal, the number of states that each module can take decreases in comparison with monolithic Q-learning. As a result, Whitehead showed that modular Q-learning improves on learning time and requires less computational resources.

Modular Q-learning and other methods that decompose the problem into modules need a human designer to decompose the whole problem and design an appropriate set of modules. This is because the learning performance depends on the design of the module set. However, agents with fixed modules might not be able to adapt to dynamic environments flexibly.

In this paper, we propose a new architecture that enables agents to obtain a suitable set of modules through interaction with the environment, so that agents can flexibly adapt to dynamic environments. Also, we experimentally evaluate this architecture. As an example of a learning problem that is computationally intractable for agents, we take the *pursuit game* problem. We investigate the suitability of this architecture and its adaptability under dynamic environments.

2 Learning with the Modular Approach

In this section, we describe the basics of Q-learning and modular Q-learning. Also, we show an example that uses modular Q-learning.

*Also with "Research for the Future" Project, Faculty of Science and Technology, Keio University, Shin-Kawasaki-Mitsui Building West 3F, 890-120 Kashimada, Saiwai-ku, Kawasaki, 221, Japan

†Also with Sony Computer Science Laboratory Inc. 3-14-13 Higashi-Gotanda, Shinagawa-ku, Tokyo, 141, Japan

2.1 Reinforcement Learning (Q-learning)

Autonomous creatures generally receive reinforcement signals from the environment for their actions or action sequences. Reinforcement learning is a technique for an agent to learn suitable action policies that maximize a utility from the clues of reinforcement signals. Utility is often represented by the discounted cumulative reinforcement in the future, as follows:

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

where V_t is the discounted cumulative reinforcement from time t through the future, r_t is the reward received at time t, and $\gamma (0 \leq \gamma \leq 1)$ is the temporal discount factor.

Q-learning [Watkins, 1989] is a widely used method of reinforcement learning, where learning corresponds to building a precise Q-function.

$$< Q(state, action) \rightarrow utility >$$

The Q-function gives the estimated utilities that agents receive when they take an *action* in a *state*, and the agents refer to this function to decide on their actions. For deterministic domains, the utility of an action a in response to a state x is equal to the immediate payoff r plus the best utility that can be obtained from the next state y. However, during the course of learning, the Q-function may not be true. Therefore, the value of $Q(x, a)$ is updated in the following way:

$$Q(x, a) \leftarrow Q(x, a) + \beta(r + \gamma \max_b Q(y, b) - Q(x, a))$$

where $\beta (0 \leq \beta \leq 1)$ is the learning rate, $\max_b Q(y, b)$ is the maximum Q-value at state y, and γ is the discount rate.

The simplest way to express the Q-function is to make a Q-table that contains each Q-value for every pair of state and action. However, as the size of the problem space increases, this takes more computational resources and requires much more time for learning.

2.2 Modular Q-Learning

Some research shows that methods which decompose a problem space improve learning performance in reinforcement learning [Dayan and Hinton, 1993] [Singh, 1992]. One investigation of the modular approach shows the improvement of learning performance [Whitehead *et al.*, 1993].

A modular Q-learning architecture contains a number of modules, each of which dedicates itself to the corresponding subproblem. This architecture makes the Q-table smaller and improves learning time for problems with multiple goals.

In modular Q-learning, each module has its own subgoal and decides its action policy according to its subgoal. An arbiter is used to mediate global action with a certain strategy (decision by majority, etc.). Each module includes a Q-table for the pairs of partial states and actions. Modules learn to achieve their respective goals through updating their Q-values. Since individual modules cannot consider other modules' goals, learning agents may yield suboptimal performance. What is worse, even if human designers succeed in designing an appropriate set of modules for a situation, such a predetermined method reduces agents' ability to adapt to dynamic environments.

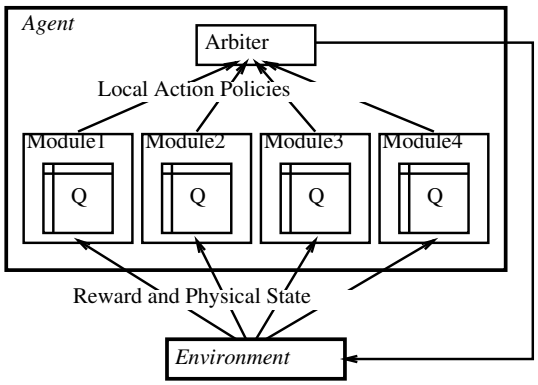

Figure 1: Modular Q-Learning Architecture

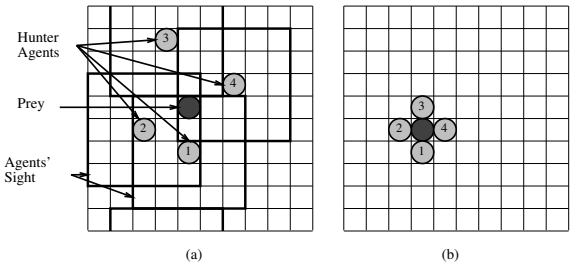

Figure 2: Pursuit Game

Ono [1996] applied modular Q-learning to the learning of cooperative behavior of multi-agents in a pursuit game (Figure 2(a)). The pursuit game that was presented by Benda [1985] is a basic problem of distributed artificial intelligence. The purpose of this problem is to capture a prey agent by making four hunter agents surround the prey (Figure 2(b)). In Ono's research, hunter agents learned the optimal action policy to capture the prey. Ono's learning agents consist of three modules. Each module takes the relative position of the prey and of one of the other hunters as its state.

Considering the state space, for example, in the case of each hunter agent having a limited visual field of depth (5×5), as shown in Figure 2(a), the possible number of states that agents may encounter reaches $(5^2 + 1)^4 = 456976$. This means a learning agent with a monolithic Q-table needs enormous memory resources. However, when only two agents organize separate Q-tables, the possible number of states decreases to $(5^2 + 1)^2 = 676$. Through experimenting with architectures in the pursuit game, Ono showed that agents with modular Q-learning

not only solve state space issues but also learn cooperative behavior. However, the method which Ono took was to decompose the state space and allocate suitable modules of modular Q-learning in advance.

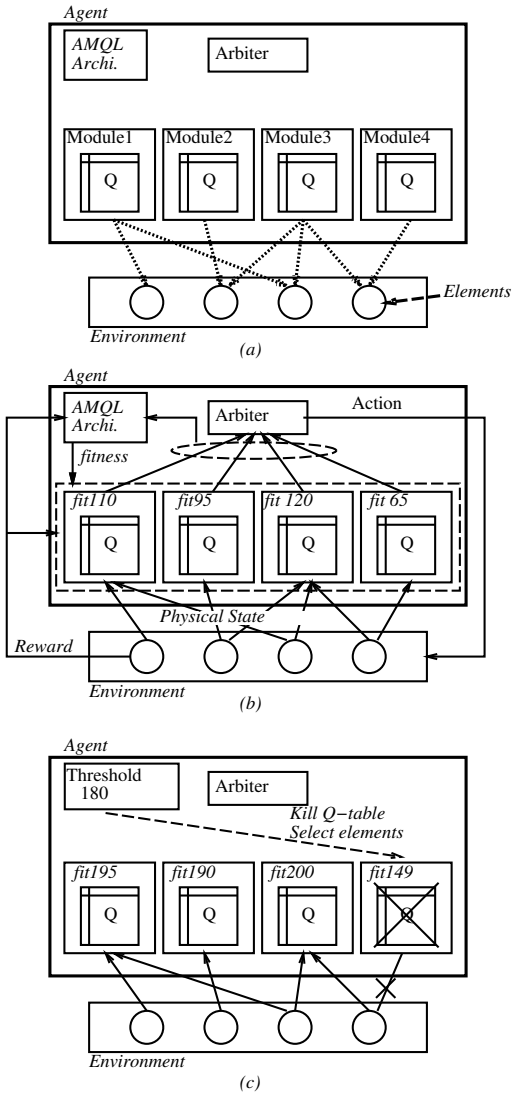

Figure 3: AMQL architecture: (a) element selection phase, (b) fitness allocation phase, and (c) module selection phase.

3 AMQL Architecture

In modular Q-learning, human designers have made modules suitable for the environment. The resulting agents with fixed modules cannot adapt to dynamically changing environments. Here, we propose a new architecture, called *AMQL (Automatic Modular Q-Learning)*, that enables agents to obtain a suitable set of modules by themselves. Also, we experimentally show the availability of this architecture.

The AMQL architecture has three phases when it obtains the set of modules. Firstly, modules in agents select elements of the environment, and make a Q-table (Figure 3(a)). While agents learn, the AMQL mechanism provides fitness estimates for modules that contribute a reward acquisition action (Figure 3(b)). At selection time, modules are evaluated and selected according to fitness (Figure 3(c)). After repeating these three phases, agents obtain a suitable set of modules. In short, AMQL executes module selection like a genetic algorithm. The specific mechanisms are as follows.

3.1 Element Selection Phase

In the AMQL architecture, agents have a fixed number of modules. Each agent recognizes the environment as a state of n elements$(E_1, E_2, \cdots, E_n)$. In other words, the input of an agent is as follows.

$$\text{Agent input } S = S_1, S_2, \cdots, S_n$$

Where S_i is a state of element E_i, elements correspond to the sensors of robot agents. Each module randomly selects $1 - m$ elements $(1 \leq m < n)$ from the set of elements E, and makes a Q-table for the states of selected elements. Through this process, the state space is allocated to multiple modules.

3.2 Fitness Allocation Phase

When agents receive a reward by an action, the modules that contribute to this action are allocated a fitness. This fitness indicates the degree of contribution of a module.

Agents decide their action by using *the greatest mass* (GM) strategy, proposed by Whitehead [1993].

$$f_{gm}(S) = argmax_{a \in A} \sum_{i=1}^{l} Q^i(S, a)$$

The expression, where $f_{gm}(S)$ is an action policy of an agent, represents that the Q-values of l modules are summed up for all possible actions, and the action that has the largest sum is selected.

When agents receive a reward at state S and action a, the Q-value at state S is compared for each module. Modules that have the Q-value of action a have the largest sums. In other words, all modules that desire action a increase their fitness. The fitness is cleared at every stage of module selection, and is added whenever the module contributes to the acquisition of a reward.

Modules do not refer to other modules' Q-tables and each updates its own Q-table individually. The goal of the module is to learn appropriate action policies for inputs of elements to which the module should pay attention. Modules that pay attention to the appropriate subset of elements can learn appropriate action policies. Therefore, appropriate modules achieve greater fitness.

3.3 Module Selection Phase

At every period of module selection T, agents evaluate their own modules. In evaluating modules, selection mechanisms assess the fitness of each module. Modules

whose fitness is over the threshold remain, but if fitness is under the threshold, modules select elements of the environment again, and make a new Q-table. Threshold at module selection is represented as follows:

$$k = (\text{number of reward acquisition}) \times p$$

In the case that $p(0 < p < 1)$ is a constant that determines how good modules must be to be retained, the closer p is to 1, the higher the quality of the module selected.

3.4 Algorithm

The AMQL architecture consists of the above three phases. The actual algorithm is as follows:

1. Each module selects $1-m$ elements from n elements that organize the environment. Modules then make a Q-table and set an initial Q-value.
2. Set the fitness of all modules to 0.
3. Set the execution counter t of agents to 0.
4. Set agents to initial states.
5. Agents receive the current state x.
6. If state x is the goal state, then $t = t + 1$ and go to 11.
7. Agents select their own action by the GM strategy and execute it.
8. Update Q-tables of each modules.
9. If agents receive a reward with action a, then set the fitness of the module, that Q-value of action a is the biggest in every action, to $fitness = fitness + 1$.
10. Go to 5.
11. If $t < T$ (the period of select), then go to 4.
12. Calculate a fitness threshold and evaluate fitness of all modules. Modules where fitness is under the threshold select $1 - m$ elements again. These modules then make a Q-table and set an initial Q-value. Go to 2.

4 Implementation and Evaluation

In this paper, we use the pursuit game as a problem that is computationally intractable. Specifically, we investigate the following. Firstly, we compare the performance of the AMQL architecture with previous architectures. Also, we investigate the adaptability of AMQL architecture in dynamic environments compared to fixed module Q-learning.

4.1 Simulation Environment

We assume that the fundamental simulation environment of the pursuit game is as follows:

- The environment consists of a 10×10 grid. Edges are connected (torus).
- Initial positions of each agent are determined randomly.

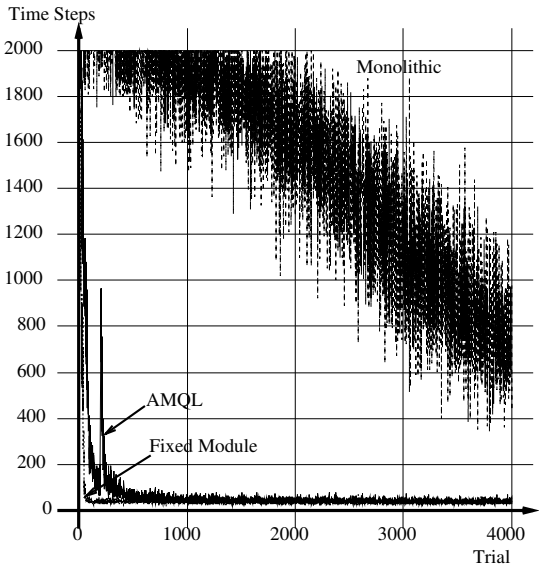

Figure 4: Time steps till caption at each trial

- At each time step, agents choose an action from any of 5 actions (move north, east, south, west, or stay at their current position). More than one hunter agent can share the same grid. However, hunter agents cannot share with the prey. Therefore, hunter agents that try to move to the grid already occupied by the prey, cannot move and must stay at their current positions. The prey agent selects its own action randomly.
- Hunter agents have a 5×5 sight, as shown in Figure 2(a). Each agent is assigned an identifier (Agent $1-4$). A hunter agent can recognize the relative position and identifier of any other agents in its sight.
- A trial ends when the goal is accomplished (the prey agent is captured) or at $2,000$ time steps.

Parameters for the AMQL architecture are as follows. A hunter agent has four modules. The period of module selection T is 200 trials. The constant p that determines the threshold of the module is 0.9. When modules are initialized or there are some modules that are regarded as not adaptable at the module selection phase, these modules select any $1-3$ other agents randomly and make a Q-table.

Parameters for Q-learning are as follows. The learning rate is $\beta = 0.1$, the discount factor is $\gamma = 0.9$, a reward is $r = 1$, and the initial value of the Q-value is 0.1.

4.2 Comparison among AMQL and Previous Architectures

We have compared the AMQL architecture with previous architectures: monolithic Q-learning and a fixed module architecture. Hunter agents with the monolithic architecture have one Q-table for all possible states. In

the case of fixed modules, hunter agents have three modules that pay attention to each different hunter agent and the prey agent. This set of modules is the same as Ono's [1996]. One execution is performed in up to 4000 trials. Figure 4 shows the time steps taken for capture at each trial. Each experiment is repeated ten times, and the averages of these are plotted.

The results of the monolithic architecture show that it takes a long time to learn suitable behaviours. This is because the number of states that the monolithic Q-table has to take increases to $26^4 = 456976$, so that it takes a long time to update Q-values suitably. However, using AMQL or the fixed module architecture, agents can capture the prey agent with less time steps. In almost every experiment, the AMQL architecture selected the module that takes states of two or three agents as an input. Therefore, the number of states that AMQL modules take becomes $26^2 = 676$ or $26^3 = 17576$, and agents with the AMQL architecture can learn to capture the prey faster. Comparing learning speed and quality of solution for this simple case, the fixed module architecture previously designed by human designers is better than the AMQL architecture. After learning, agents with AMQL took about 15 time steps more on average than the well formed modules. However, in every experiment, agents with AMQL succeeded in obtaining a suitable set of modules.

The reason why convergence of AMQL is slower is that the AMQL architecture takes time to find a suitable set of modules. In other words, in the case of fixed module architecture, human designers are needed to design a suitable set of modules. The AMQL architecture cannot only learn faster, but can also obtain suitable modules automatically. These two features are important for autonomous agents. This result shows that the AMQL architecture has the capability to obtain suitable sets of modules with improvement of learning speed due to the modular architecture.

4.3 Adaptation to Changing Environments

In the next experiment, we evaluate the adaptability to problem change. To change the problem dynamically, we set the following additional conditions.

- There are four hunter agents and two prey agents in the grid field. Each prey agent has an identifier (prey1, prey2).
- From the beginning to trial 2000, hunter agents receive a reward only when the prey agent 1 is captured, but no reward when the prey agent 2 is caught. However, from trial 2001, the environment changes so that hunter agents receive a reward only when prey 2 is captured.

In this experiment, we compare the adaptability to dynamic environments, between the AMQL architecture and the fixed module architecture that is designed by human designers. The fixed module architecture in this experiment is the same as that in the previous experiment: hunter agents have three modules that pay attention to each different hunter agent and prey agent 1. No modules pay attention to prey agent 2.

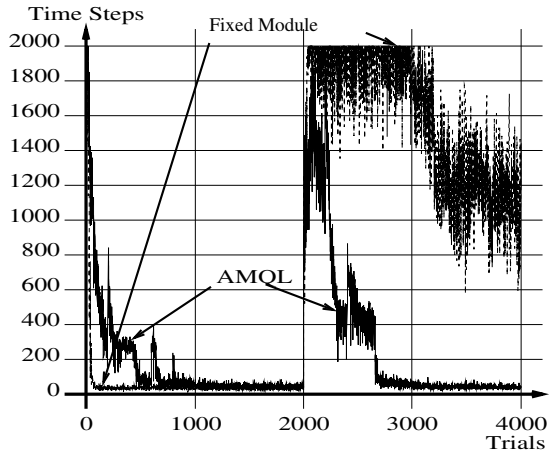

Figure 5: Time steps till caption at each trial. Comparison between AMQL and fixed module architecture in a changing environment.

Table 1: Elements of modules of hunter1 in a changing environment

Module	Before change	After change
Module1	Hunter2 Hunter3 Prey1	Hunter2 Hunter4 Prey2
Module2	Hunter2 Hunter3 Hunter4	Hunter2 Hunter3 Prey2
Module3	Hunter3 Hunter4 Prey1	Hunter3 Prey1 Prey2
Module4	Hunter2 Hunter3	Hunter4 Prey2

Figure 5 shows the time steps for capture at each trial. Each experiment is also repeated ten times, and the averages of these are plotted. Before the problem is changed, the convergence of learning with the fixed module architecture is faster than with the AMQL architecture. However, the fixed module architecture cannot capture well in the environment after the problem changes. This is due to the set of modules being designed for the environment before the change. However, agents with the AMQL architecture can capture well even after changing the problem, because the architecture changes the modules when the target is changed.

Table 1 shows the changes of elements to which the modules of hunter agent1 pay attention. This result shows that agents have some modules that pay attention to prey1 before changing, and that these modules pay attention to prey2 after changing, because agents always have to pay attention to elements that are re-

lated to reward. Moreover, when agents pay attention to elements that are not related to reward, agents waste learning time and computational resources.

Human designers cannot design a suitable set of modules in dynamically changing environments, since they cannot predict the future environment. However, agents with AMQL architecture obtain suitable, though still not optimal, sets of modules that consider their own reward and learning performance in a current environment.

Figure 5 shows that convergence of AMQL, immediately after changing environment, takes longer to converge than at the beginning. This is because Q-functions immediately after changing have learned to catch prey1. This obstructs the module in learning the new problem. However, this feature can be an advantage, if what modules have learned can be used even after the environment has changed.

These experimental results show that an agent with the AMQL architecture is able to obtain a suitable set of modules and to adapt according to changing problems.

5 Discussion

There has been much research dealing with modular architectures for reinforcement learning. Thrun and Schwartz offered the SKILLS algorithm which obtains the structure of problem space that can be used among multiple tasks and called skills, and considered both performance loss and description length [1995]. However, to obtain useful skills takes more time than to find optimal policies with a monolithic Q-function. AMQL makes the description length smaller and learning convergence faster than monolithic architecture, though not optimal. We consider that the idea of considering performance and description length is useful for AMQL.

At the present time, modules in AMQL have lookup tables, because these are easy to treat and to analyse. Also, general function approximators such as neural networks have been applied to reinforcement learning. Sabes and Jordan discussed the association between reinforcement learning and expert networks [1996]. We consider that this model also can be applied to AMQL. In this case, each module obtains an appropriate part of input using AMQL. Such an architecture can be expected to have faster learning convergence and more adaptability than a monolithic architecture.

It is important to know how autonomous agents should design their adaptive mechanism for the environment. The real world is an open system, and changes dynamically. Thus, it is difficult to predict the behavior of future environments perfectly and to design optimal structures. In that respect, AMQL seems a promising architecture, because it can dynamically and automatically obtain suitable module structures through interactions with the environment, and learn faster than monolithic Q-learning architectures. We consider the AMQL architecture to be applicable to various autonomous agents in order to improve both the adaptability and the learning time.

6 Conclusions and Future Work

In this paper, we proposed an AMQL architecture that obtains a suitable set of modules through the interaction with the environment. We showed the availability of the architecture and its adaptability to dynamic environments, through experiments on pursuit game problems. Simulation results showed that agents with the AMQL architecture can not only learn faster but also can obtain suitable sets of modules automatically. Moreover, through experiments where the problem changed dynamically, we showed that agents with the AMQL architecture can adapt themselves to dynamically changing environments, in a way that was impossible for agents with the previous fixed method. These features enable autonomous agents to adapt more flexibly and efficiently.

When the environment changes drastically, the current AMQL architecture may abandon modules which have learned strategies that are no longer relevant. It seems useful to keep such modules, in case a similar situation arises in the future. We plan to investigate this architecture.

References

[Benda *et al.*, 1985] M. Benda, V. Jagannathan, and R. Dodhiawalla. On Optimal Cooperation of Knowledge Sources. Technical Report BCS-G2010-28, Boeing AI Center, 1985.

[Dayan and Hinton, 1993] Peter Dayan and Geoffrey E. Hinton. Feudal Reinforcement Learning. In *Advances in Neural Information Processing Systems 5*, pages 271–278, 1993.

[Ono and Fukumoto, 1996] Norihiko Ono and Kenji Fukumoto. Multi-agent Reinforcement Learning: A Modular Approach. In *Proceedings Second International Conference on Multi-Agent Systems*, pages 252–258, 1996.

[Sabes and Jordan, 1996] Philip N. Sabes and Michael I. Jordan. Reinforcement Learning by Probability Matching. In *Advances of Neural Information Processing Systems 8*, 1996.

[Singh, 1992] Satinder P. Singh. Reinforcement Learning with a Hierarchy of Abstract Models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202–207, 1992.

[Thrun and Schwartz, 1995] Sebastian Thrun and Anton Schwartz. Finding Structure in Reinforcement Learning. In *Advances in Neural Information Processing Systems 7*, pages 385–392, 1995.

[Watkins, 1989] C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, 1989.

[Whitehead *et al.*, 1993] Steven Whitehead, Jonas Karlsson, and Josh Tenenberg. Learning Multiple Goal Behavior via Task Decomposition and Dynamic Policy Merging. In *ROBOT LEARNING*. Kiluwer Academic Press, 1993.