Transfer of Learning by Composing Solutions of Elemental Sequential Tasks

Satinder Pal Singh
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
SATINDER@cs.umass.edu

March 6, 1992

Abstract

Although building sophisticated learning agents that operate in complex environments will require learning to perform multiple tasks, most applications of reinforcement learning have focussed on single tasks. In this paper I consider a class of sequential decision tasks (SDTs), called composite sequential decision tasks, formed by temporally concatenating a number of elemental sequential decision tasks. Elemental SDTs cannot be decomposed into simpler SDTs. I consider a learning agent that has to learn to solve a set of elemental and composite SDTs. I assume that the structure of the composite tasks is unknown to the learning agent. The straightforward application of reinforcement learning to multiple tasks requires learning the tasks separately, which can waste computational resources, both memory and time. I present a new learning algorithm and a modular architecture that learns the decomposition of composite SDTs, and achieves transfer of learning by sharing the solutions of elemental SDTs across multiple composite SDTs. The solution of a composite SDT is constructed by computationally inexpensive modifications of the solutions of its constituent elemental SDTs. I provide a proof of one aspect of the learning algorithm.

Keywords: Reinforcement Learning, Compositional Learning, Modular Architecture, Transfer of Learning.

Running Head: Transfer of Learning across Sequential Tasks

To appear in the Machine Learning Journal (1992).

1 Introduction

Although building sophisticated autonomous learning systems that operate in complex environments will require learning to perform multiple tasks, most applications of control architectures based on reinforcement learning have focussed on single tasks (e.g., Barto et al., 1991; Sutton, 1990; Whitehead & Ballard, 1990). One way to extend reinforcement learning to multiple tasks is to use different modules to learn the solutions of the different tasks. If the tasks are related, learning them separately will waste computational resources, both memory and time. In addition, reinforcement learning methods do not scale efficiently to large tasks, and thus the "learn each task independently" method is only feasible for simple tasks. Discovering faster learning algorithms for single tasks will certainly help, but algorithms that allow transfer of learning across tasks must also play a crucial role in control systems that have to learn to perform multiple tasks. While achieving transfer of learning across an arbitrary set of tasks may be difficult, or even impossible, there are useful classes of tasks where such transfer is achievable. In this paper I focus on one class of such tasks.

In this paper I consider a learning agent that interacts with a dynamic external environment and faces multiple sequential decision tasks. Each task requires the agent to execute a sequence of actions to control the environment[1], either to bring it to a desired state or to traverse a desired state trajectory over time. In addition to the environment dynamics that define state transitions, such tasks are defined by a payoff function that specifies the immediate evaluation of each action taken by the agent. The agent's goal is to learn a closed loop control policy[2], i.e., a function assigning actions to states, that extremizes some measure of the cumulative payoffs received over time. Sequential decision tasks are difficult to solve because the long-term consequences of taking an action are seldom reflected in the immediate payoff, making it difficult to assign credit to actions. Thus, decisions cannot be viewed in isolation because the agent must account for both the short-term and the long-term consequences of its actions. The framework of sequential decision-making is quite general and can be used to pose a large number of tasks from diverse fields (e.g., Bertsekas, 1987).

Much of everyday human activity involves sequential tasks that have compositional structure, i.e., complex tasks built up in a systematic way from simpler tasks. As an example, consider how many of our daily activities involve the subtasks of lifting an object, opening a door, sitting down, walking, etc. Compositionally-structured tasks allow the possibility of sharing knowledge across the many tasks that have common subtasks. To formulate the problem abstractly, consider an agent that has to learn to solve many different elemental and composite tasks. In general, there may be n elemental tasks labeled $T_1, T_2, \ldots, T_n$. *Elemental* tasks cannot be decomposed into simpler subtasks. *Composite* tasks, labeled $C_1, C_2, \ldots, C_m$, are produced by temporally concatenating a number of elemental tasks. For example, $C_j = [T(j,1)T(j,2)\cdots T(j,k)]$, is composite task j made up of k elemental tasks that have to performed in the order listed. $T(j,i) \in \{T_1, T_2, \ldots, T_n\}$, for

[1] The environment is the system or plant being controlled by the agent.

[2] In the machine learning literature closed loop control policies are also referred to as situation-action rules, or simply as "reactions".

$1 \leq i \leq k$, is the i^{th} elemental task in the list for task C_j. The sequence of elemental tasks in a composite task will be referred to as the *decomposition* of the composite task. I assume that the decomposition of a composite task is unknown to the learning agent.

Compositional learning involves solving a composite task by learning to compose the solutions of the elemental tasks in its decomposition. It is to be emphasized that given the short-term, evaluative nature of the payoff from the environment (often the agent gets informative payoff only at the completion of the composite task), the task of discovering the decomposition of a composite task is formidable. In this paper I propose a compositional learning scheme in which separate modules learn to solve the elemental tasks, and a task-sensitive gating module solves composite tasks by learning to compose the appropriate elemental modules over time (also see Singh, 1992a). I present the results of three simulations using a set of simple navigational tasks that illustrate the advantage of compositional learning, and I prove one aspect of the learning algorithm in Appendix A.

2 Markovian Decision Tasks and Q-learning

One often-studied class of sequential decision tasks consists of Markovian decision tasks (MDTs) in which the agent seeks to control a finite-state, discrete-time, stochastic dynamical system. Let S be the finite set of states and A be the finite set of actions available to the agent.[3] At each time step t, the agent observes the system's current state $x_t \in S$ and executes action $a_t \in A$. As a result, the agent receives a payoff with expected value $R(x_t, a_t) \in \mathbb{R}$ and the system makes a transition to state $x_{t+1} \in S$ with probability $P_{x_t x_{t+1}}(a_t)$. The environment satisfies the Markovian property, i.e., the current state and future actions determine the expected future sequence of states and payoffs independently of the state trajectory prior to the current state. The transition probabilities for all states and actions constitute the environment dynamics. The payoff function that assigns expected payoffs to state-action pairs is defined independently of the environment dynamics.

The agent's *goal* is to determine a closed loop control policy that maximizes some cumulative measure of the payoffs received over time. The number of time steps over which the cumulative payoff is determined is called the *horizon* of the MDT. One measure, called the expected *return* of a policy for a given initial environment state x_0, is the expected value of the sum over an infinite horizon of the discounted payoffs that the agent would receive if it were to use the given policy to select its actions. The expected return is equal to $E\left[\sum_{t=0}^{\infty} \gamma^t R(x_t, a_t)\right]$, where E indicates expected value. The discount factor γ, where $0 \leq \gamma \leq 1$, allows the payoffs distant in time to be weighted less than more immediate payoffs. Any control policy that achieves the agent's goal (and there may be more than one) is called an optimal policy. The optimal policies for the class of infinite horizon MDTs defined above are stationary (Ross, 1983), i.e., are not functions of time, and therefore, I restrict attention to stationary policies. Clearly, both the environment dynamics and the payoff function play a role in determining the set of optimal policies. Changing the payoff function while keeping the same environment dynamics can change the set of policies that are optimal. That this

[3]For ease of exposition I assume that the same set of actions are available to the agent from each state. The extension to the case where different sets of actions are available in different states is straightforward.

is not always true is apparent from the fact that there are a finite number of stationary policies but an infinite number of payoff functions.

If the environment dynamics and the payoff function are known, computational procedures based on dynamic programming can be used to find an optimal policy (Bertsekas, 1987). In particular, DP methods based on the value iteration algorithm (e.g. Ross, 1983) compute an optimal *value function* that assigns to states their scalar expected returns under an optimal policy. Given the optimal value function, an optimal policy can be determined by selecting for each state an action that if executed would maximize the expected value of the sum of the immediate payoff and the expected return for the next state. In the absence of a model of the environment dynamics, reinforcement learning methods that approximate dynamic programming, such as temporal difference (Sutton, 1988; Barto et al., 1983) and Q-learning methods (Watkins, 1989; Barto et al., 1990) can be used to directly estimate an optimal policy without building an explicit model of the dynamics of the environment.

Following Watkins, I define the Q-value, $Q(x,a)$, for $x \in S$ and $a \in A$, as the expected return on taking action a in state x, under the condition that an optimal policy is followed thereafter. Given the Q-values, a greedy policy that in each state selects an action with the highest associated Q-value, is optimal. Q-learning is an asynchronous version of value iteration that maintains an estimate, $\hat{Q}$, of the Q-values for all states and actions. Q-learning is often used on-line, i.e., actual experience interacting with the environment is used to incrementally improve $\hat{Q}$ over time. On executing action a in state x at time t, the resulting payoff and next state are used to update the estimate at time t, $\hat{Q}_t(x,a)$ as follows[4]:

$$\hat{Q}_{t+1}(x,a) \quad = \quad (1.0 - \alpha_t)\hat{Q}_t(x,a) + \alpha_t(R(x,a) + \gamma \max_{a' \in A} \hat{Q}_t(y,a')), \tag{1}$$

where y is the state at time $t+1$, and α_t is the value of a positive learning rate parameter at time t. Watkins and Dayan (1992) prove that under certain conditions on the sequence $\{\alpha_t\}$, if every state-action pair is updated infinitely often using Equation 1, $\hat{Q}$ converges to the true Q-values asymptotically.

Computing a greedy policy with respect to $\hat{Q}$ requires evaluating $\hat{Q}(x,a)$ for all $x \in S$ and $a \in A$. A common practice that eliminates this computation is to cache the control policy in a separate controller module.[5] To ensure asymptotic convergence most control methods based on Q-learning incorporate some exploration strategy (Kaelbling, 1990; Barto & Singh, 1990; Sutton, 1990) that sometimes takes non-greedy actions.

---

[4]Here, I assume that on executing action a in state x the agent receives the expected value of the immediate payoff $R(x,a)$, instead of just a sample.

[5]However, this adds the computational effort of training the controller module. In addition, the mismatch between the controller module and $\hat{Q}$ can adversely affect the time required to learn an optimal policy.

3 Elemental and Composite Markovian Decision Tasks

Several different elemental tasks can be defined in the same environment. In this paper I restrict attention to elemental tasks requiring the agent to bring the environment to a desired final state. Each elemental task has its own final state. All elemental tasks are MDTs that share the the same state set S, action set A, and the same environment dynamics. The payoff function, however, can be different across the elemental tasks. Furthermore, I assume that the payoff function for each elemental task T_i, $1 \leq i \leq n$, $R_i(x,a) = \sum_{y \in S} P_{xy}(a) r_i(y) - c(x,a)$, where $r_i(y)$ is the positive reward associated with the state y resulting from executing action a in state x for task T_i, and $c(x,a)$ is the positive cost of executing action a in state x. I assume that $r_i(x) = 0$ if x is not the desired final state for T_i. Note that the reward function r_i is task-dependent while the cost function c is task-independent. Thus, the elemental tasks share the same cost function but have their own reward functions.

A composite task is defined as an ordered sequence of elemental MDTs. It is clear that a composite task is not itself a MDT because the payoff is a function of both the state and the current elemental task, instead of the state alone. To make composite tasks qualify as MDTs, the set S can be extended as follows:[6] imagine a device that for each elemental task detects when the desired final state of that elemental task is visited for the first time and then remembers this fact. This device can be considered part of the learning system or equivalently as part of a *new* environment for the composite task.

Formally, the new state set for a composite task, S', is formed by augmenting the elements of set S by n bits, one for each elemental task. For each $x' \in S'$ the *projected state* $x \in S$ is defined as the state obtained by removing the augmenting bits from x'. The environment dynamics and cost function, c, for a composite task is defined by assigning to each $x' \in S'$ and $a \in A$ the transition probabilities and cost assigned to the projected state $x \in S$ and $a \in A$. The reward function for composite task C_j, r_j^c, is defined as follows: $r_j^c(x') > 0$ if the projected state x is the final state of some elemental task in the decomposition of C_j, say task T_i, and if the augmenting bits of x' corresponding to elemental tasks before and including subtask T_i in the decomposition of C_j are one, and if the rest of the augmenting bits are zero; $r_j^c(x') = 0$ everywhere else.

Composite tasks as defined above are MDTs, and the results of Watkins and Dayan (1992) extend to them. Thus, Q-learning can be used to learn the Q-values for a composite task. The learning agent has to learn to solve a number of elemental and composite tasks in its environment. At any given time, the task the agent faces is determined by a device that can be considered to be part of the environment or to be a part of the agent. If the device is considered to be part of the environment, it can be thought of as providing a task command to the agent. On the other hand, if the device is part of the agent, it provides a context or internal state for the agent. The two views are formally equivalent; the crucial property is that they determine the payoff function but do not affect the dynamics of the environment.

[6] The theory developed in this paper does not depend on the particular extension of S chosen, as long as the appropriate connection between the new states and the elements of S can be made.

If the task command is considered part of the state description, the entire set of tasks faced by the agent becomes one unstructured Markovian decision task. While an optimal policy for the unstructured MDT can be found by using Q-learning, just as for an elemental MDT, the structure inherent in the set of compositionally structured tasks allows a more efficient solution. In the next section I show that the Q-values for a composite task have a special relationship to the Q-values of the elemental tasks in its decomposition. I then present a learning architecture that uses that relationship to advantage.

4 Compositional Q-learning

Compositional Q-learning (CQ-learning) is a method for constructing the Q-values of a composite task from the Q-values of the elemental tasks in its decomposition. Let $Q^{T_i}(x,a)$ be the Q-value of (x,a), $x \in S$ and $a \in A$, for elemental task T_i, and let $Q^{C_j}_{T_i}(x',a)$ be the Q-value of (x',a), for $x' \in S'$ and $a \in A$, for task T_i when performed as part of the composite task $C_j = [T(j,1) \cdots T(j,k)]$. Let $T(j,l) = T_i$. Note that the superscript on Q refers to the task and the subscript refers to the elemental task currently being performed. The absence of a subscript implies that the task is elemental.

Consider a set of undiscounted ($\gamma = 1$) MDTs that have compositional structure and satisfy the following conditions:
(A1) Each elemental task has a single desired final state.
(A2) For all elemental and composite tasks the expected value of undiscounted return for an optimal policy is bounded both from above and below for all states.
(A3) The cost associated with each state-action pair is independent of the task being accomplished.
(A4) For each task elemental task T_i the reward function r_i is zero for all states except the desired final state for that task. For each composite task C_j, the reward function r^c_j is zero for all states except the final states of the elemental tasks in its decomposition (see Section 3).
Then, for any elemental task T_i and for all composite tasks C_j containing elemental task T_i, the following holds:

$$Q^{C_j}_{T_i}(x',a) \;=\; Q^{T_i}(x,a) + K(C_j,l), \tag{2}$$

for all $x' \in S'$ and $a \in A$, where $x \in S$ is the projected state, and $K(C_j,l)$ is a function of the composite task C_j and l, where T_i is the l^{th} task in the decomposition of C_j. Note that $K(C_j,l)$ is independent of the state and the action. Thus, given solutions of the elemental tasks, learning the solution of a composite task with n elemental tasks requires learning only the values of the function K for the n different subtasks. A proof of Equation 2 is given in Appendix A.

Equation 2 is based on the assumption that the decomposition of the composite tasks is known. In the next Section, I present a modular architecture and learning algorithm that simultaneously discovers the decomposition of a composite task and implements Equation 2.

5 CQ-L: The CQ-Learning Architecture

Jacobs et al. (1991) developed a modular gating architecture that does task decomposition. The gating architecture consists of several expert modules and a special gating module that has an output for each expert module. When presented with training patterns (input-output pairs) from multiple tasks, the expert modules compete with each other to learn the training patterns, and this competition is mediated by the gating module. The best interpretation of this architecture is as an "associative Gaussian mixture model" (see Jacobs & Jordan, 1991 ; Jacobs et al. 1991) of the distribution of training patterns. Learning is achieved by gradient descent in the log likelihood of generating the desired target patterns.

The gating architecture has been used to learn only multiple non-sequential tasks within the supervised learning paradigm (Duda & Hart, 1973). I extend the gating architecture to a CQ-Learning architecture (Figure 1), called CQ-L, that can learn multiple compositionally-structured sequential tasks even when training information required for supervised learning is not available. CQ-L combines CQ-learning and the gating architecture to achieve transfer of learning by "sharing" the solutions of elemental tasks across multiple composite tasks. Only a high level description of the CQ-L is provided in this section; the details are given in Appendix B.

In CQ-L the expert modules of the gating architecture are replaced by Q-learning modules that learn to approximate the Q-values for the elemental tasks. The Q-modules receive as input both the state ($\in S$) and the actions ($\in A$). The gating and bias modules (see Figure 1) receive as input the augmenting bits and the task command used to encode the current task being performed by the architecture. The stochastic switch in Figure 1 selects one Q-module at each time step. CQ-L's output is the output of the selected Q-module added to the output of the bias module.

At time step t, let the output of the Q-module i be $q_i(t)$, and let the i^{th} output of the gating module after normalization be $g_i(t)$. $g_i(t)$ is the *prior* probability that the Q-module i is selected at time step t by the stochastic switch. At each time step, the following steps are performed. For the current task and current state, x_t, the outputs of the gating module are determined and used to select a single Q-module. I use q_t to refer to the output of the Q-module selected at time t. $q_t(x_t, a)$ is evaluated for all $a \in A$. One action is selected probabilistically using the Gibbs distribution, i.e., $\forall a \in A$, $P(a|x_t) = e^{\beta q_t(x_t a)} / \sum_{a' \in A} e^{\beta q_t(x_t a')}$, where the parameter β controls the probability of selecting a non-greedy action. The chosen action, a_t, is then executed, and $q_i(x_t, a_t)$ is evaluated for all i.

The estimated Q-value for the current task and state-action pair, $\hat{Q}(x_t, a_t)$ is set equal to $q_t(x_t, a_t) + K(t)$, where $K(t)$ is the output of the bias module at time t. The bias module implements the function K of Equation 2. The estimate of the error in $\hat{Q}(x_t, a_t)$, which becomes available at time $t+1$, is

$$e(t) \quad = \quad R(x_t, a_t) + \hat{Q}(x_{t+1}, a_{t+1}) - \hat{Q}(x_t, a_t), \tag{3}$$

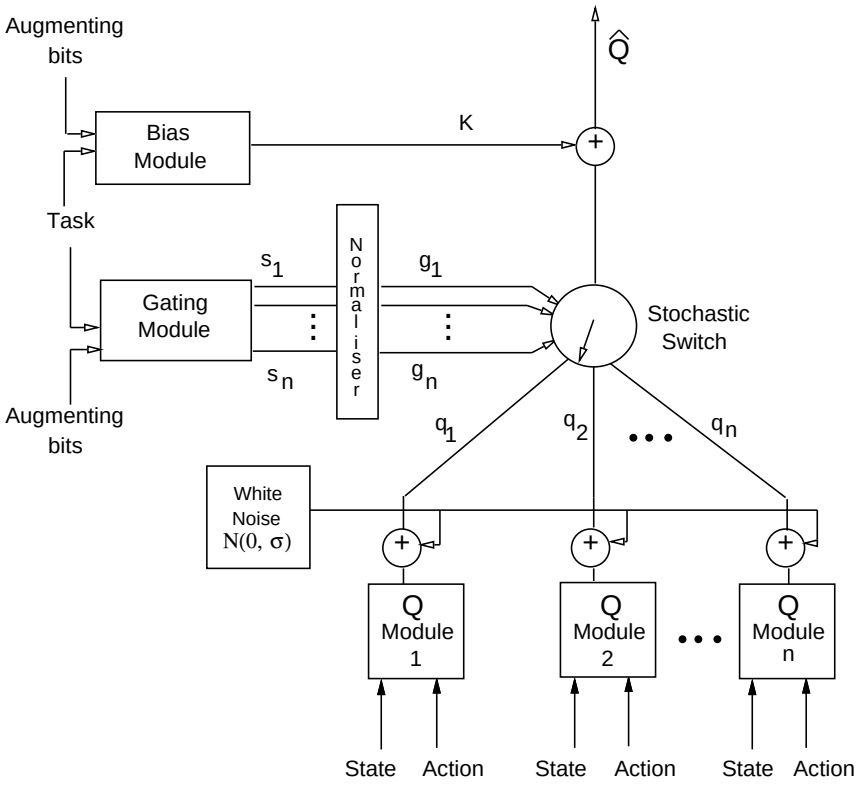

Figure 1: CQ-L: The CQ-Learning Architecture. This figure is adapted from Jacobs et al. (1991). The Q-modules learn the Q-values for elemental tasks. The gating module has an output for each Q-module and determines the probability of selecting a particular Q-module. The bias module learns the function K (see Equation 2).

where $R(x_t, a_t)$ was the expected payoff at time t.[7] Alternatively, the estimate of the desired output of CQ-L at time t can be defined as $D(t) = e(t) + \hat{Q}(t)$. At each time step the learning rules described in Appendix B are used to adjust *all* the Q-modules, the gating module, and the bias module.

Each Q-module is adjusted to reduce the error between its output and $D(t) - K(t)$ in proportion to the probability of that Q-module having produced the estimated desired output. Hence the Q-module whose output would have produced the least error is adjusted the most. The gating module is adjusted so that the *a priori* probability of selecting each Q-module becomes equal to the *a posteriori* probability of selecting that Q-module, given $D(t)$. Over time, different Q-modules start winning the competition for different elemental tasks, and the gating module learns to select the appropriate Q-module for each task. For composite tasks, while performing subtask T_i, the Q-module that has best learned task T_i will have smaller expected error than any other Q-module. The bias module is also adjusted to reduce the error $e(t)$. The parameter β is increased over time so that eventually only the greedy actions are selected.

6 Navigation Tasks: Description and Representation

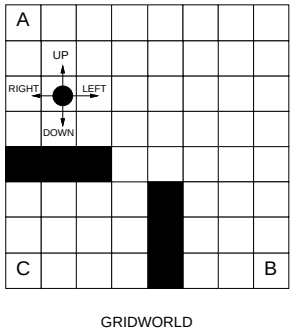

Figure 2: The Grid Room. The room is an 8×8 grid with three desired final locations designated A, B and C. The filled squares represent obstacles. The robot is shown as a circle and has four actions available: UP, DOWN, RIGHT, and LEFT.

I use a set of deterministic navigation tasks to illustrate the performance of CQ-L. Figure 2 shows an 8×8 grid room with three desired final locations designated A, B and C. The robot is shown as a circle, and the filled squares represent obstacles that the robot must avoid. In each state the robot has 4 actions: UP, DOWN, LEFT and RIGHT. Any action that would take the robot into an obstacle or boundary wall does not change the robot's location. There are three elemental tasks: "visit A", "visit B", and "visit C", labeled T_1, T_2 and T_3 respectively. Three composite tasks, C_1, C_2, and C_3 were constructed by temporally concatenating some subset of the elemental tasks. It is to be emphasized that the goal for each task is not merely to find any path that leads

[7] Note that Equation 3 is an accurate implementation of Q-learning only if the action taken at time $t+1$ is the greedy action. As the value of parameter β is increased, the probability of executing the greedy action increases.

Table 1: Tasks. Tasks T_1, T_2, and T_3 are elemental tasks; tasks C_1, C_2, and C_3 are composite tasks. The last column describes the compositional structure of the tasks.

Label	*Command*	*Description*	*Decomposition*
T_1	000001	visit A	T_1
T_2	000010	visit B	T_2
T_3	000100	visit C	T_3
C_1	001000	visit A and then C	T_1T_3
C_2	010000	visit B and then C	T_2T_3
C_3	100000	visit A, then B and then C	$T_1T_2T_3$

to the desired final state, but to find the shortest path. The six different tasks, along with their labels, are described in Table 1. $c(x,a) = -0.05$, for all $x \in S \cup S'$ and $a \in A$. $r_i(x) = 1.0$, if $x \in S$ is the desired final state of elemental task T_i, or if $x \in S'$ is the final state of composite task C_i, and $r_i(x) = 0.0$ in all other states. Thus, for composite tasks no intermediate payoff for successful completion of subtasks was provided.

The representation used for the task command determines the difficulty of discovering the decomposition of a composite task. At one extreme, using "linguistic" representations can reduce the problem of discovering the decomposition to that of "parsing" the task command. At the other extreme, unstructured task command representations force the system to learn the decomposition of each composite task separately. To avoid the issues arising from structured representations I use unit basis vectors to represent the task commands (Table 1). Lookup tables were used to implement all the modules of the CQ-Learning architecture. The elements of the set S are the possible positions of the robot in the grid room. The 3 augmenting bits, one for each the three elemental tasks, and the task command vector form the input to the bias and the gating modules (Figure 1).

7 Simulation Results

In the simulations described below, the performance of CQ-L is compared to the performance of a "one-for-one" architecture that implements the "learn-each-task-separately" strategy. The one-for-one architecture has a pre-assigned distinct module for each task, which prevents transfer of learning, in other words, the one-for-one architecture treats the entire set of tasks as one unstructured MDT. Each module of the one-for-one architecture was provided with the augmented state ($\in S'$).

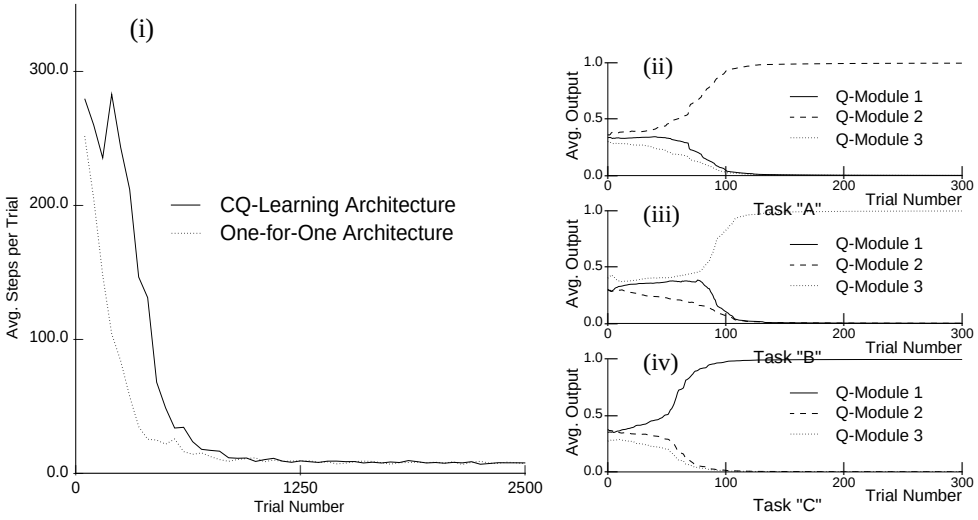

Figure 3: **(i)** Learning Curves for Multiple Elemental tasks. Each data point is the average, taken over 50 trials, of the number of actions taken by the robot to get to the desired final state. **(ii)** Module Selection for Task T_1. The 3 normalized outputs of the gating module are shown averaged over each trial with task T_1. Initially the outputs were about 0.3 each, but as learning proceeded the gating module learned to select Q-module 2 for task T_1. **(iii)** Module Selection for Task T_2. Q-module 3 was selected. **(iv)** Module Selection for Task T_3. Q-module 1 was selected for task T_3.

7.1 Simulation 1: Learning Multiple Elemental Markovian Decision Tasks

Both CQ-L and the one-for-one architecture were separately trained on the three elemental MDTs T_1, T_2, and T_3 until they could perform the three tasks optimally. Both architectures contained three Q-modules. The task for each trial and the starting location of the robot were chosen randomly. Each trial ended when the robot reached the desired final location for that task. Figure 3(i) shows the number of actions, with each data point an average over 50 trials, taken by the robot to get to the desired final state. The one-for-one architecture converged to an optimal policy faster than CQ-L did, because it took time for the gating module's outputs to become approximately correct, at which point CQ-L learned rapidly.

Figures 3(ii), 3(iii), and 3(iv) respectively show the three normalized outputs of the gating module for trials involving tasks T_1, T_2 and T_3. Each data point is the output of the gating module averaged over a complete trial. For each graph, the abscissa is the number of times the corresponding tasks occurred. The gating module outputs started close to 0.3. After approximately 100 trials the gating module had learned to assign a different Q-module to each task. This simulation shows that CQ-L is able to partition its "across-trial" experience and learn to engage one Q-module for each elemental task. This is similar in spirit to the simulations reported by Jacobs (1990), except that he does not apply his architecture to sequential decision tasks. See Appendix C for simulation details.

7.2 Simulation 2: Compositional Learning

Both CQ-L and the one-for-one architecture were separately trained on the six tasks T_1, T_2, T_3, C_1, C_2, and C_3 until they could perform the six tasks optimally. CQ-L contained three Q-modules, and the one-for-one architecture contained six Q-modules. The task for each trial and the starting state of the robot were chosen randomly. Each trial ended when the robot reached the desired final state. Figure 4(i) shows the number of actions, averaged over 50 trials, taken by the robot to reach the desired final state. The one-for-one architecture performed better initially because it learned the three elemental tasks quickly, but learning the composite tasks took much longer due to the long action sequences required to accomplish the composite tasks. The CQ-Learning architecture performed worse initially, until the outputs of the gating module become approximately correct, at which point all six tasks were learned rapidly.

Figures 4(ii), 4(iii), and 4(iv) respectively show the three outputs of the gating module for one trial each of tasks C_1, C_2, and C_3. The trials shown were chosen after the robot had learned to do the tasks, specifically, after $10,000$ learning trials. The elemental tasks T_1, T_2, and T_3 respectively were learned by the Q-modules 1, 3 and 2. For each composite task the gating module learned to compose the outputs of the appropriate elemental Q-modules over time. This simulation shows that CQ-L is able to learn the decomposition of a composite task, and that compositional learning, due to transfer of training across tasks, can be faster than learning each composite task separately. See Appendix C for simulation details.

Figure 4: **(i)** Learning Curves for a Set of Elemental and Composite Tasks. Each data point is the average over 50 trials of the time taken by the robot to reach the desired final state. **(ii)** Temporal Composition for Task C_1. After $10,000$ learning trials, the three outputs of the gating module during *one* trial of task C_1 are shown. Q-module 1 was turned on for the first seven actions to accomplish subtask T_1, and then Q-module 2 was turned on to accomplish subtask T_3. **(iii)** Temporal Composition for Task C_2. Q-module 3 was turned on for the first six actions to accomplish subtask T_2 and then Q-module 2 was turned on to accomplish task T_3. **(iv)** Temporal Composition for Task C_3. The three outputs of the gating module for one trial with task C_3 are shown. Q-modules 1, 3 and 2 were selected in that order to accomplish the composite task C_3.

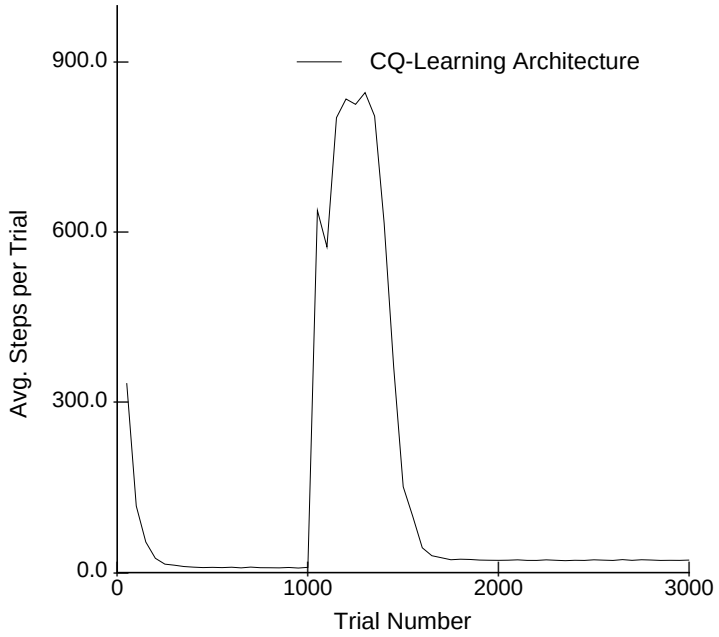

Figure 5: Shaping. The number of actions taken by the robot to reach the desired final state, averaged over 50 trials, is plotted. The CQ-Learning architecture containing one Q-module was trained for 1000 trials on task T_3. Then another Q-module was added, and the architecture was trained to accomplish task C_2. The only task-dependent payoff for the task C_2 was on reaching the desired final location C.

7.3 Simulation 3: Shaping

One approach (cf. Section 7.2) for training a robot to learn a composite task is to train the robot on multiple tasks: all the elemental tasks required in the composite task, and the composite task itself. Another approach is to train the robot on a succession of tasks, where each succeeding task requires some subset of the already learned elemental tasks, plus a new elemental task. This roughly corresponds to the "shaping" procedures (Skinner, 1938) used by psychologists to train animals to do complex motor tasks.

A simple simulation to illustrate shaping was constructed by training CQ-L with one Q-module on one elemental task, T_3, for $1,000$ trials and then training on the composite task C_2. After the first $1,000$ trials, the learning was turned off for the first Q-module and a second Q-module was added for the composite task. Figure 5 shows the learning curve for task T_3 composed with the learning curve for task C_2. The number of actions taken by the robot to get to the desired final state, averaged over 50 trials, were plotted by concatenating the data points for the two tasks, T_3 and C_2. Figure 5 shows that the average number of actions required to reach the final state increases when the composite task was introduced, but eventually the gating module learned to decompose the task and the average decreased. The second Q-module learned the task T_2 without ever being explicitly exposed to it.

A separate shaping simulation was conducted in the same way as above, except that learning was not turned off for the first Q-module after it had learned task T_3. The solution to task T_3 was partially unlearned before the architecture figured out the decomposition for the composite task. As a result, it took more time than in the earlier shaping simulation to learn the elemental task T_2 in the second Q-module. See Appendix C for simulation details.

8 Discussion

Learning to solve MDTs with large state sets is difficult due to the sparseness of the evaluative information and the low probability that a randomly selected sequence of actions will be optimal. Learning the long sequences of actions required to solve such tasks can be accelerated considerably if the agent has prior knowledge of useful subsequences. Such subsequences can be learned through experience in learning to solve other tasks. In this paper, I define a class of MDTs, called composite tasks, that are structured as the temporal concatenation of simpler MDTs, called elemental tasks. I present CQ-L, an architecture that combines the Q-learning algorithm of Watkins (1989) and the modular architecture of Jacobs et al. (1991) to achieve transfer of learning by sharing the solutions of elemental tasks across multiple composite tasks. CQ-L's use of acquired subsequences to solve composite tasks is similar to the use of macro-operators (Korf, 1985) in macro-learning systems (Iba, 1989).

Another architecture similar to CQ-L is the subsumption architecture for autonomous intelligent agents (Brooks, 1989), which is composed of several task-achieving modules along with precompiled switching circuitry that controls which module should be active at any time. Maes and Brooks (1990) showed how reinforcement learning can be used to learn the switching circuitry for a robot with hardwired task modules. Mahadevan and Connell (1990), on the other hand, showed how Q-learning can be used to acquire behaviors that can then be controlled using a hardwired switching scheme. The simulation illustrating shaping with CQ-L demonstrates that not only can the gating module compose sequences of elemental tasks on which it has been trained, it can also learn new elemental tasks in order to solve a composite task. Thus, for compositionally-structured MDTs, CQ-L combines the complementary objectives of Maes and Brooks's architecture with that of Mahadevan and Connell's architecture.

Given a set of composite and elemental MDTs, the sequence in which the learning agent receives training experiences on the different tasks determines the relative advantage of CQ-L over other architectures that learn the tasks separately. The simulation reported in Section 7.2 demonstrates that it is possible to train CQ-L on intermixed trials of elemental and composite tasks, while the simulation involving shaping (Section 7.3) demonstrates that it is possible for CQ-L to learn elemental tasks on which it has not been explicitly trained. Nevertheless, some training sequences on a set of tasks will result in faster learning of the set of tasks than other training sequences. The ability of CQ-L to scale well to complex sets of tasks is still dependent on the choice of the training sequence.

To focus on compositional learning I avoided the issues arising from the use of representations

more structured than unit-basis vectors and function approximators more powerful than lookup tables. In addition, Watkins and Dayan's (1992) theorem on the convergence of Q-learning holds for lookup-table representations; it is not known to hold for other function approximation methods. Nevertheless, future work with connectionist networks may make it possible to use distributed representations for states and thus to take advantage of the ability of networks to generalize across the states within the same task.

According to the definition used in this paper, composite tasks have only one decomposition and require the elemental tasks in their decomposition to be performed in a fixed order. A broader definition of a composite tasks allows it to be an unordered list of elemental tasks, or more generally, a disjunction of many ordered elemental task sequences. CQ-L should work with the broader definition for composite tasks without any modification because it should select the particular decomposition that is optimal with respect to its goal of maximizing expected returns. Further work is required to test this conjecture.

Although compositional Q-learning was illustrated using a set of simple navigational tasks, it is suitable for a number of different domains where multiple sequences from some set of elemental tasks have to be learned. CQ-L is a general mechanism whereby a "vocabulary" of elemental tasks can be learned in separate Q-modules, and arbitrary[8] temporal syntactic compositions of elemental tasks can be learned with the help of the bias and gating modules.

Appendix

A Proof of Equation 2

Consider an elemental deterministic Markovian decision task (MDT) T_i and its desired final state $x_g \in S$. The payoff function for task T_i is $R_i(x,a) = \sum_{y \in S} P_{xy}(a) r_i(y) - c(x,a)$, for all $x \in S$ and $a \in A$. By assumptions A1–A4 (Section 4) we know that the reward $r_i(x) = 0$ for all $x \neq x_g$. Thus, for any state $y \in S$ and action $a \in A$,

$$Q^{T_i}(y,a) \;\; = \;\; r_i(x_g) - c(y,a) - \Phi(y,x_g),$$

where $\Phi(y,x_g)$ is the expected cost of going from state y to x_g under an optimal policy, π_i, for elemental task T_i.

Consider a composite task C_j that satisfies assumptions A1–A4 given in Section 4 and w.l.o.g. assume that for $C_j = [T(j,1)T(j,2)\cdots T(j,k)]$, $\exists 1 \leq l \leq k$, such that $T(j,l) = T_i$. Let the set $L \subset S'$ be the set of all $x' \in S'$ that satisfy the property that the augmenting bits corresponding to the tasks before T_i in the decomposition of C_j are equal to one and the rest are equal to zero. Let $y' \in L$ be the state that has the projected state $y \in S$. Let $x'_g \in S'$ be the state formed from

[8] This assumes that the state representation is rich enough to distinguish repeated performances of the same elemental task.

$x_g \in S$ by setting to one the augmenting bits corresponding to all the subtasks before and including subtask T_i in the decomposition of C_j, and setting the other augmenting bits to zero. Let π'_j be an optimal policy for task C_j. $r^c_j(x')$ is the reward for state $x' \in S'$ while performing task C_j. Then by assumptions A1–A4, we know that $r^c_j(x') = 0$ for all $x' \in L$ and that $c(x', a) = c(x, a)$.

By the definition of C_j, the agent has to navigate from state y' to state x'_g to accomplish subtask T_i. Let $\Phi'(y', x'_g)$ be the expected cost of going from state y' to state x'_g under policy π'_j. Then, given that $T(j, l) = T_i$,

$$Q^{C_j}_{T_i}(y', a) \quad = \quad Q^{C_j}_{T(j,l+1)}(x'_g, b) + r^c_i(x'_g) - c(y', a) - \Phi'(y', x'_g),$$

where $b \in A$ is an optimal action for state x'_g while performing subtask $T(j, l+1)$ in task C_j. Clearly, $\Phi'(y', x'_g) = \Phi(y, x_g)$, for if it were not, either policy π_i would not be optimal for task T_i, or given the independence of the solutions of the subtasks the policy π'_j would not be optimal for task C_j. Define $K(C_j, l) \equiv Q^{C_j}_{T(j,l+1)}(x'_g, b) + r^c_i(x'_g) - r_i(x_g)$. Then

$$Q^{C_j}_{T_i}(y', a) \quad = \quad Q^{T_i}(y, a) + K(C_j, l).$$

Q.E.D.

B Learning rules for the CQ-Learning Architecture

The derivation of the learning rules in this section follows the derivation in Nowlan (1990) closely. The output of Q-module i, q_i, is the mean of a Gaussian probability distributions with variance σ. The i^{th} output of the gating module, s_i, determines the *a priori* probability, $g_i = e^{s_i} / \sum_j e^{s_j}$, of selecting Q-module i.

The final output of the architecture at time step t, is given by $\hat{Q}(t) = q_t(t) + K(t)$, where q_t is the output of the selected Q-module and K is the output of the bias module. The desired output at time t is, $D(t) = R(t) + \hat{Q}(t+1)$, where $R(t)$ is the expected payoff at time t. Then the probability that the Q-module i will generate the desired output is $p_i(D(t)) = \frac{1}{N\sigma} e^{\frac{\|(D(t) - K(t)) - q_i(t)\|^2}{2\sigma^2}}$, where N is a normalizing constant. The *a posteriori* probability that Q-module i was selected given the desired output $D(t)$, is $p(i|D(t)) = \frac{g_i(t) p_i(D(t))}{\sum_j g_j(t) p_j(D(t))}$. The likelihood of producing the desired output, $L(D(t))$, is given by $\sum_j g_j(t) p_j(D(t))$.

The objective of the architecture is to maximize the log likelihood of generating the desired Q-values for the task it is being trained on. The partial derivative of the log likelihood with respect to the output of the Q-module j is $\frac{\partial \log L(t)}{\partial q_j(t)} = \frac{1}{\sigma^2} p(j|D(t))((D(t) - K(t)) - q_j(t))$. The partial derivative of the log likelihood with respect to the i^{th} output of the gating module simplifies to

$\frac{\partial \log L(t)}{\partial s_i(t)} = (p(i|D(t)) - g_i(t))$. Using the above results the update rules for Q-module j, the i^{th} output of the gating module, and the bias module[9] respectively are:

$$\begin{aligned} q_j(t+1) &= q_j(t) + \alpha_Q \frac{\partial \log L(t)}{\partial q_j(t)}, \\ s_i(t+1) &= s_i(t) + \alpha_g \frac{\partial \log L(t)}{\partial s_i(t)}, \text{and} \\ b(t+1) &= b(t) + \alpha_b (D(t) - Q(t)), \end{aligned}$$

where α_Q, α_b and α_g are learning rate parameters.

C Parameter values for Simulations 1, 2 and 3

For all three simulations, the initial values for the lookup tables implementing the Q-modules were random values in the range 0.0–0.5, and the initial values for the gating module lookup table were random values in the range 0.0–0.4. For all three simulations, the variance of the Gaussian noise, σ, was 1.0 for all Q-modules.

For Simulation 1, the parameter values for the both CQ-Learning and the one-for-one architectures were: $\alpha_Q = 0.1$, $\alpha_b = 0.0$, $\alpha_g = 0.3$. The policy selection parameter, β, started at 1.0 and was incremented by 1.0 after every 100 trials.

For Simulation 2, the parameter values for CQ-L were: $\alpha_Q = 0.015$, $\alpha_b = 0.0001$, $\alpha_g = 0.025$, and β was incremented by 1.0 after every 200 trials. For the one-for-one architecture, the parameter values were: $\alpha_Q = 0.01$ and β was incremented by 1.0 after every 500 trials.[10]

For Simulation 3, the parameter values, $\alpha_Q = 0.1$, $\alpha_b = 0.0001$, and $\alpha_g = 0.01$ were used, and β was incremented by 1.0 every 25 trials.

Acknowledgements

This work was supported by the Air Force Office of Scientific Research, Bolling AFB, under Grant AFOSR-89-0526 and by the National Science Foundation under Grant ECS-8912623. I thank Andrew Barto, Rich Sutton, and Robbie Jacobs for their guidance and help in formulating my ideas. Discussions with Richard Yee, Vijaykumar Gullapalli, and Jonathan Bachrach helped focus my ideas. I also thank Andrew Barto for his patience in reading and commenting on many hastily written drafts of this paper. Without his extensive help, no reader would have gotten far enough in this paper to be reading this sentence.

[9] This assumes that the bias module is minimizing a mean square error criteria.

[10] Incrementing the β values faster did not help the one-for-one architecture.

References

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1991). Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57, COINS dept., University of Massachusetts, Amherst, MA.

Barto, A. G. & Singh, S. P. (1990). On the computational economics of reinforcement learning. In *Proc. of the 1990 Connectionist Models Summer School*, San Mateo, CA. Morgan Kaufmann.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE SMC*, *13*, 835–846.

Barto, A. G., Sutton, R. S., & Watkins, C. (1990). Sequential decision problems and neural networks. In Touretzky, D. S. (Ed.), *Advances in Neural Information Processing Systems 2*, pages 686–693, San Mateo, CA. Morgan Kaufmann.

Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models.* Englewood Cliffs, NJ: Prentice-Hall.

Brooks, R. (1989). A robot that walks: Emergent behaviors from a carefully evolved network. *Neural Computation*, *1*, 23–262.

Duda, R. O. & Hart, P. E. (1973). *Pattern Classification and Scene Analysis.* New York: Wiley.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Jacobs, R. A. (1990). *Task decomposition through competition in a modular connectionist architecture.* PhD thesis, COINS dept., Univ. of Massachusetts, Amherst, Mass. U.S.A.

Jacobs, R. A. & Jordan, M. I. (1991). A competitive modular connectionist architecture. In *Advances is Neural Information Processing Systems 3.*

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, *3*(1).

Kaelbling, L. P. (1990). *Learning in Embedded Systems.* PhD thesis, Stanford University, Department of Computer Science, Stanford, CA. Technical Report TR-90-04.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence*, *26*, 35–77.

Maes, P. & Brooks, R. (1990). Learning to coordinate behaviours. In *Proceedings of the Eighth AAAI*, pages 796–802. Morgan Kaufmann.

Mahadevan, S. & Connell, J. (1990). Automatic programming of behavior-based robots using reinforcement learning. Technical report, IBM Research Division, T.J.Watson Research Center, Yorktown Heights, NY.

Nowlan, S. J. (1990). Competing experts: An experimental investigation of associative mixture models. Technical Report CRG-TR-90-5, Department of Computer Sc., Univ. of Toronto, Toronto, Canada.

Ross, S. (1983). *Introduction to Stochastic Dynamic Programming.* New York: Academic Press.

Singh, S. P. (1992a). On the efficient learning of multiple sequential tasks. In Lippman, R., Moody, J., & Touretzky, D. (Eds.), *Advances in Neural Information Processing Systems 4*, San Mteo, CA. Morgan Kauffman.

Singh, S. P. (1992b). Solving multiple sequential tasks using a hierarchy of variable temporal resolution models. Submitted to Machine Learning Conference, 1992.

Skinner, B. F. (1938). *The Behavior of Organisms: An experimental analysis.* New York: D. Appleton Century.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*, 9–44.

Sutton, R. S. (1990). Integrating architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc. of the Seventh International Conference on Machine Learning*, pages 216–224, San Mateo, CA. Morgan Kaufmann.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* PhD thesis, Cambridge Univ., Cambridge, England.

Watkins, C. J. C. H. & Dayan, P. (1992). Q-learning. *Machine Learning.* to appear.

Whitehead, S. D. & Ballard, D. H. (1990). Active perception and reinforcement learning. In *Proc. of the Seventh International Conference on Machine Learning*, Austin, TX. M.