Multi-Agent Reinforcement Learning: Weighting and Partitioning

Ron Sun
Todd Peterson
The University of Alabama
Department of Computer Science
Tuscaloosa, AL 35487

April 7, 1999

Contact author: Ron Sun, The University of Alabama, Department of Computer Science, Tuscaloosa, AL 35487. Fax: 205-348-2109. Phone: 205-348-6363. Email: rsun@cs.ua.edu.

Running title: Multi-Agent Reinforcement Learning

ACKNOWLEDGEMENT: This work was conducted while the first author was funded by the Office of Naval Research grant N00014-95-1-0440 and by a supplemental research grant from the University of Alabama (which included a research assistantship for the second author). Thanks to the anonymous reviewers for their comments.

Multi-Agent Reinforcement Learning: Weighting and Partitioning

Abstract

This paper addresses weighting and partitioning in complex reinforcement learning tasks, with the aim of facilitating learning. The paper presents some ideas regarding weighting of multiple agents and extends them into partitioning an input/state space into multiple regions with differential weighting in these regions, to exploit differential characteristics of regions and differential characteristics of agents to reduce the learning complexity of agents (and their function approximators) and thus to facilitate the learning overall. It analyzes, in reinforcement learning tasks, different ways of partitioning a task and using agents selectively based on partitioning. Based on the analysis, some heuristic methods are described and experimentally tested. We find that some off-line heuristic methods performed the best, significantly better than single-agent models.

Keywords: weighting, averaging, neural networks, partitioning, gating, reinforcement learning,

1 Introduction

Multiple agents can be used in many problems in lieu of a single agent. The goal is to make a complex learning task easier and/or to achieve better performance (Whitehead 1993), through combining the outcomes of multiple agents (see e.g., Breiman 1996 a, b, c, Schapire et al 1997, Wolpert 1992, Jacobs et al 1991, and Jordan and Jacobs 1994). According to the existing literature, combination can be done in various ways in accordance with the problem characteristics: in classification problems, voting can be used; in numerical prediction problems, averaging can be used. In combining outcomes, weighting can also be adopted so that each agent carries a different weight (Krogh and Vedelsby 1995, Tresp and Taniguchi 1995, Breiman 1996b). The goal is to make sure that the combination is optimal in some sense. Furthermore, weighting need not be done uniformly throughout the problem space (Tresp and Taniguchi 1995): Uniform weighting may be disadvantageous because of differing characteristics of different regions in the problem space (in which case an agent may perform well in a particular region of the space and should ideally be weighted more in that region). Thus, we may want to adopt the partitioning of the problem space in addition to weighting so that we can have agents that specialize to local regions (which may or may not overlap each other; see Jacobs et al 1991, Jordan and Jacobs 1994, Jacobs 1997, Tresp and Taniguchi 1995). An added advantage of partitioning is that we can use simpler agents: local agents usually turn out to be a lot simpler than a monolithic, global agent; e.g., simpler functional forms can be used when we approximate polynomial functions using multiple agents (Schaal and Atkeson 1996); or much fewer numbers of hidden units are needed in backpropagation networks when multiple such networks are used. Consequently, partitioning can potentially improve learning to a significant degree.

These methods have not been applied extensively to tasks other than simple regression/prediction and classification problems. Existing proposals related to more involved tasks such as reinforcement learning (RL) (i.e., learning by autonomous agents on the basis of only simple feedback from the environment) are limited (such as Tham 1995, Singh 1994, Dayan and Hinton 1993, Humphrys 1996, Dorigo and Gambardella 1995, Dietterich 1997; more discussions of them later). Therefore, there is a need for further exploration of multi-agent approaches in tasks such as reinforcement learning. In this paper, our main aims are (1) to present a uniform perspective on various multi-agent approaches (including weighting and partitioning, as mentioned earlier) in reinforcement learning, and (2) to present our new methods motivated and developed in light of this perspective. In the remainder of this paper, first, we will analyze weighting and show the optimality of several weighting schemes. We will then relate, under the rubric of partitioning, "Mixture of Expert" gating (Jacobs et al 1991), boosting (Freund and Schapire 1996), decision trees (McCallum 1996, Chrisman 1993) and so on, and address some issues concerning their optimality. In light of these analyses, we will proceed to propose several new ways of achieving optimal partitioning and optimal weighting. Experiments of these new methods will then be presented and discussed.

The advantage of our methods lies in the fact that they help to make a learning task easier and more manageable and to achieve a better performance (Whitehead 1993), and moreover, they require little a priori domain-specific knowledge to begin with, unlike many existing approaches that require detailed *domain* knowledge to initialize partitioning (such as in the case of "knowledge-based" RBF networks; Taniguchi and Tresp 1997, Kubat 1997), a priori partitioning of the problem space (such as in Singh 1994 and Humphrys 1996), a priori domain-specific division of subsequences (such as in the case of gating of reinforcement learners; Tham 1995), or a priori determination of domain-specific goal/subgoal structures (such as in the hierarchical RL approaches of Dayan and Hinton 1993 and Dietterich 1997). Our methods are suitable for incremental reinforcement learning in which changes of domain structures and characteristics can occur, because it does not involve a priori structuring and tends to make changes easy. In this respect, they are similar to the ideas of Jacobs et al (1991), Jordan and Jacobs (1994), Tresp and Taniguchi (1995), and Blanzieri and Katenkamp (1996) (see detailed discussions later). However, these methods did not deal with RL directly. Our methods search for optimal partitioning of state space, along with, or separate from, the learning of individual agents, so as to manage the overall complexity of both the learning of individual agents and the combinations of these agents. In particular, our *off-line* methods that learn partitioning separately from the learning of individual agents tend to reduce the overall complexity by isolating to certain extent the two aspects of learning.

2 Weighting

We want to analyze the optimality issues associated with simple averaging and weighted averaging. For simple averaging, we have

$$a(x) = \frac{\sum_k a_k(x)}{n} \tag{1}$$

where x is an input, k denotes an agent ($k \in [1, n]$), $a_k(x)$ is the output of agent k, and $a(x)$ is the averaged output. For weighted averaging, we have

$$a(x) = \frac{\sum_k w_k a_k(x)}{\sum_k w_k} \tag{2}$$

where x is an input, k denotes an agent ($k \in [1, n]$), $a_k(x)$ is the output of agent k, w_k is the weight for agent k, and $a(x)$ is the combined output. Note that $a(x) = \sum_k w_k a_k(x)$, if $\sum_k w_k = 1$. [1]

In the context of either regression or classification tasks, suppose $y(x)$ is the correct output, where x is the input, and suppose the output of agents are $a_k(x)$, where k indicates an agent k. Then the average error (for training or for cross validation) is

$$\begin{aligned} avg_{k=1}^n (y(x) - a_k(x))^2 &= y(x)^2 - 2y(x) avg_{k=1}^n a_k(x) + avg_{k=1}^n a_k(x)^2 && (3) \\ &\geq (y(x) - avg_{k=1}^n a_k(x))^2 && (4) \end{aligned}$$

where $avg_{k=1}^n a_k = \sum_{k=1}^n a_k / n$. This is because

$$avg_{k=1}^n a_k^2(x) \geq (avg_{k=1}^n a_k(x))^2 \tag{5}$$

Summing over all data points, we have

$$avg_{k=1}^n \sum_x (y(x) - a_k(x))^2 \geq \sum_x (y(x) - avg_{k=1}^n a_k(x))^2 \tag{6}$$

[1]While the issue of optimality of averaging has been discussed extensively along the line of bias/variance decomposition, we will extend Breiman (1996 a)'s analysis (the analysis of "bagging", which is simple averaging of agents trained with re-sampling of instances) to address the optimality of weighted averaging. The analysis is limited to showing that averaging or weighted averaging is better than single agents *on average*, not that it is better than or equal to the best single agent. For some suggestions regarding the latter point, see Breiman (1996b).

Note that $avg_{k=1}^{n} a_k(x)$ is the output of averaging. This means that the average error of an individual agent is always greater than or equal to the error of the combined outcome (i.e., the averaging of all the agents). [2]

In the above scheme (simple averaging), the weights for all the agents are identical (i.e., $w_k = 1/n$). But we may vary these weights in the hope of getting better results (i.e., using weighted averaging instead, as suggested by Wolpert 1992, Breiman 1996 b). To make sure that a differential weighting scheme is indeed beneficial, we need to show that

$$avg_{k=1}^{n} \sum_x (y(x) - a_k(x))^2 \geq \sum_x (y(x) - \sum_{k=1}^{n} w_k a_k(x))^2 \tag{7}$$

That is, the weighting scheme of w_k's reduces the error, where w_k's are weights, subject to the constraints $w_k \geq 0$ and $\sum w_k = 1$. A direct way to guarantee that is to minimize (e.g., through gradient descent)

$$error = \sum_x error(x) = \sum_x (y(x) - \sum_{k=1}^{n} w_k a_k(x))^2 \tag{8}$$

with weights subject to the above constraints. Obviously, with such minimization, we are guaranteed that the needed inequality (7) always holds (i.e., the total error is always reduced), because as derived earlier, we have $avg_{k=1}^{n} \sum_x (y(x) - a_k(x))^2 \geq \sum_x (y(x) - \sum_{k=1}^{n} w_k a_k(x))^2$, if $w_k = 1/n$ for all k.

An alternative way of optimization, which is also common, is to minimize

$$error' = \sum_x error'(x) = \sum_x \sum_{k=1}^{n} w_k (y(x) - a_k(x))^2 \tag{9}$$

with weights subject to the same constraints. As shown by Krogh and Vedelsby (1995),

$$\sum_x \sum_{k=1}^{n} w_k (y(x) - a_k(x))^2 \tag{10}$$

$$= \sum_x (y(x) - \sum_{k=1}^{n} w_k a_k(x))^2 + \sum_x \sum_{k=1}^{n} w_k (a_k(x) - \sum_{k=1}^{n} w_k a_k(x))^2$$

That is, minimizing this criterion by adjusting combination weights is equivalent to minimizing the overall sum squared error of the combined outcome (through the first term, which is the same as the earlier approach) plus minimizing the weighted averages of the variances of the agents (through the second term). The second term contributes to the reduction of error by individual agents, in addition to the reduction of error of the combined outcome (which is also dealt with by the first term).

We will refer to the latter approach as the *local-error* approach (or more specifically, the weighted-average-of-local-errors approach), and the former approach as the *overall-error* approach. As will be discussed later (in the context of reinforcement learning), these two different ways constitute two different algorithms for obtaining weights in weighted averaging: that is, either performing gradient descent on $(y(x) - \sum_{k=1}^{n} w_k a_k(x))^2$ or performing gradient descent on $\sum_{k=1}^{n} w_k (y(x) - a_k(x))^2$. That is, either

$$\Delta w_k \propto \frac{\partial \sum_x (y(x) - \sum_{k=1}^{n} w_k a_k(x))^2}{\partial w_k} \tag{11}$$

[2] As usual, we hope that the estimated errors for an individual or averaged agent are indicative of the true errors.

or

$$\Delta w_k \propto \frac{\partial \sum_x \sum_{k=1}^{n} w_k (y(x) - a_k(x))^2}{\partial w_k} \tag{12}$$

Beside ensuring the combined outcome being better than an individual agent (on average), these two ways go further and ensure that the combination weights are optimal in the sense of minimizing the weighted errors as expressed in the two forms given above. [3]

An issue closely related to weighting is diversity. The precept of choosing a diverse set of agents (i.e., uncorrelated agents) as opposed to a set of identical or highly similar agents in the averaging or weighted averaging schemes has been justified theoretically on the basis of bias-variance decomposition (see e.g. Breiman 1996c, Ueda and Nakano 1996, Raviv and Intrator 1996, and so on). The heuristics of creating independent agents has been embedded in a number of well-known approaches, such as "bagging", in which diversity is achieved through repeated random re-sampling of the training data set and the use of "unstable" (easily varied) agents (Breiman 1996a), and in "boosting", in which diversity is achieved through repeated re-sampling with changing sampling probabilities in favor of those data points that are misclassified or mispredicted (Freund and Schapire 1996, Drucker 1997).

Certainly other combination functions beside the linear combination can be equally applicable here. For example, beyond averaging and weighted averaging is the idea of "stacking" as proposed by Wolpert (1992) (see also Breiman 1996b). Instead of weighted averaging of outcomes from different agents, arbitrarily complex combination functions can be adopted that allow more flexible combinations of outcomes, such as the use of a backpropagation network for combining outcomes of agents, trained using gradient descent (as usual for backpropagation networks) based on cross-validation errors. However, due to the complexity of such combination methods, it would be harder to ensure accuracy and convergence to optima than simple averaging or weighted averaging.

The above analysis is limited to learning only the combination weights (for weighted averaging) without involving the learning of agents themselves. In many circumstances, we train both the combination weights and the agents at the same time. This analysis is also limited to an input space, or a part of an input space, in which combination weights remain constant. When there are multiple regions in an input space, each of which is dynamically formed and acquires dynamically a set of combination weights (for combining the agents within the region), the situation is more complex, due to the fact that we need to consider the partitioning of the space into regions (when this division is not pre-given) in addition to optimizing the weights for the agents in each region.

3 Weighting and Input Space Partitioning

Now let us analyze different ways of partitioning. A number of recent proposals involving multiple learned agents are concerned with partitioning input spaces into multiple regions. Here the word "region" is used in a generalized sense and refers generically to a subset of inputs.

First of all, the "Mixture of Experts" gating model produces a "soft" partitioning of spaces among different experts, with different experts being weighted differently, overlapping each other

[3] A variation to the afore-discussed weighting schemes is to interpret combination weights as probabilities; that is, instead of combining outcomes of individual agents using these weights, we can select probabilistically the outcome of an individual agent, using the relative weight of an agent as the probability of selecting it.

in their domains of expertise (Jacobs et al 1991, Jordan and Jacobs 1994, Xu et al 1995). According to the gating model, using a least-squares approach in a prediction task setting, we attempt to find optimal agents a_k, $k \in [1,n]$, and the corresponding optimal weights w_k, so as to minimize the weighted average of errors (Jacobs et al 1991):

$$error = \sum_k error_k = \sum_k \sum_{x \in S} w_k(x)(y(x) - a_k(x))^2 \tag{13}$$

where $y(x)$ is the target output for input x in the input space S, $a_k(x)$ is the output of agent k for input x, and $w_k(x)$ is the weight for agent k and input x (notice that weights vary across the input space).

To see how this model works, let us make some simplifying assumptions. If we can divide the elements of S into "equivalence classes" or regions, S_j's, in each of which $w_k(x)$ ($k \in [1,n]$) remains approximately the same (denoted as w_{kj}). We have $\cup_j S_j = S$ and $S_i \cap S_j = \emptyset$, if $i \neq j$. Let e_{k,S_j} denote $\sum_{x \in S_j}(y(x) - a_k(x))^2$. Then, we have

$$error_k = \sum_j \sum_{x \in S_j} w_k(x)(y(x) - a_k(x))^2 \approx \sum_j w_{kj} \sum_{x \in S_j}(y(x) - a_k(x))^2 = \sum_j w_{kj} e_{k,S_j} \tag{14}$$

Thus, we have

$$error = \sum_k error_k \approx \sum_k \sum_j w_{kj} e_{k,S_j} \tag{15}$$

We further assume $\sum_k w_k(x) = 1$ for each $x \in S$, and thus $\sum_k w_{kj} = 1$ for each j. We can encourage "binarization" of w_{kj}, by using steep sigmoidal function to represent these weights and using competitive learning schemes to encourage "winner-take-all" among different agents in each region, or by using feature based logical formulas for specifying regions (see more discussion regarding this formulation in section 4). If, for each j, we wind up having one k with $w_{kj} \approx 1$ and other k's with $w_{kj} \approx 0$, supposing $w_{kk} \geq w_{kj}$ for all j, then

$$w_{kj} \approx \begin{cases} 1 & \textit{if j=k} \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

Thus, we have

$$error \approx \sum_k \sum_j w_{kj} e_{k,S_j} \approx \sum_k w_{kk} e_{k,S_k} \approx \sum_k e_{k,S_k} \tag{17}$$

From this formula, it is clear that, with the simplification, the gating model is equivalent to minimizing the local error $\sum_x (y(x) - a_k(x))^2$ in each of the regions $x \in S_k$. In so doing, we try to optimize (1) the partitioning of S into regions S_k's, each of which is handled by one agent, and (2) each of the agents a_k (which can be a neural network, and can be optimized over its chosen region S_k with regard to the sum squared error: $\sum_{x \in S_k}(y - a_k(x))^2$).

Without the above simplifying assumption, instead of a "hard" partitioning into mutually exclusive and exhaustive regions, the least-squares gating will carry out a "soft" partitioning, resulting in graded boundaries and overlapping regions (that are specified by $w_k(x) \in [0,1]$). But the optimization objective is essentially the same. In this case, we interpret the combination weights as weighting agents differently at each point in the input space and as specifying the "soft" boundaries of regions. [4]

[4]Note that this is not necessarily the case, especially when hard partitioning of regions is used and multiple agents are active in each region; see section 5.2. Note also that this is different from the type of partitioning used in Singh et al (1994) in which partitioning has hard boundaries but inputs are assigned to regions probabilistically.

We can also look into the other error function discussed in section 2:

$$error = \sum_{x \in S} (y(x) - \sum_k w_k(x) a_k(x))^2 \tag{18}$$

In this case, we encourage "binarization" as before. Then, let $S_k = \{x | w_k(x) \approx 1; w_j(x) \approx 0, \forall j \neq k\}$ and $e_{k,S_k} = \sum_{x \in S_k} (y(x) - a_k(x))^2$. We have

$$error = \sum_k \sum_{x \in S_k} (y(x) - \sum_k w_k(x) a_k(x))^2 \approx \sum_k \sum_{x \in S_k} (y(x) - a_k(x))^2 = \sum_k e_{k,S_k} \tag{19}$$

So the resulting optimization problem is basically the same: optimizing the partitioning and optimizing the agents simultaneously. Yet another error function, which is a variation of the first, is as follows (Jacobs et al 1991):

$$error = -\sum_{x \in S} (log \sum_k w_k(x) e^{-(y(x) - a_k(x))^2}) \tag{20}$$

The same derivation leads to the same conclusion in this case as well. (We will later apply these methods to RL.)

Second, boosting (as in Freund and Schapire 1996), in a way, can also be viewed as "soft" partitioning of the input space. This is because in boosting, each agent focuses on different "regions" of the input space, due to the fact that at each iteration of boosting, different inputs are weighted differently so as to create a focus on a particular "region" (broadly defined) of the input space. However, in this case, unlike in gating with "soft" partitioning, the combination weights are fixed with respect to agents, not a function of inputs. Formally, at each iteration, instances are sampled according to a sampling weight distribution and are used to train an agent (a "weak learner"; Freund and Schapire 1996). Starting with an uniform distribution of sampling weights for training instances (i.e., $w_i^{(0)} = 1$, for all instances i), weights for the next iteration $k+1$ are modified based on the performance of the current agent A_k at the current iteration:

$$w_i^{(k+1)} = \begin{cases} w_i^{(k)} * \frac{\epsilon_k}{1-\epsilon_k} & \text{if instance } i \text{ is correctly classified} \\ w_i^{(k)} & \text{otherwise} \end{cases} \tag{21}$$

where k is the iteration (i.e., agent) number and ϵ_k is the total error rate for the current agent A_k (we should have $\epsilon_k < 0.5$, or the algorithm should be terminated). That is, weights are reduced for those instances that are handled correctly by the currently trained agent, so they are less likely to be sampled for the next iteration. The re-sampling probability for the next iteration is calculated according to $p_i(k+1) = \frac{w_i^{(k+1)}}{\sum_i w_i^{(k+1)}}$. Breiman (1996c) proposed a similar boosting algorithm (which he termed Arcing): it proceeds as in the original boosting algorithm, except the re-sampling probability is calculated as follows: $p_i(k+1) = \frac{1+m_i^4}{\sum_i (1+m_i^4)}$, where m_i is the misclassification rate of the ith instance by all the previous agents: A_1, ..., A_k. [5] For combining the outputs in boosting, Freund and Schapire (1996) proposed a weighted voting scheme, in which the k agent is weighted by $log \frac{1-\epsilon_k}{\epsilon_k}$, where ϵ_k is error rate for agent k. Breiman (1996c), however, showed that equal voting is equally effective.

[5]The performance of this variation is comparable to the original boosting algorithm. Thus the conclusion was drawn that the specific re-weighting scheme was inconsequential (Breiman 1996c).

In either of the two boosting schemes, at each iteration, the graded focus on some instances creates a form of "region" in the input space: those instances that have the highest weights are the centers of the region (because an agent assigned to the region will handle the instances in accordance with these weights), while instances having lower weights constitute (graded) boundaries of the region in ways much like a radial basis function (Poggio and Girosi 1990). [6] One agent is trained for each such region, and the final classification/prediction regarding a point in the input space is given by the voting of all the agents whose "regions" cover the point in question. In order to strengthen our argument regarding the role of partitioning in boosting, we can "strengthen" the afore-described boosting process: instead of the above described change to weights at each iteration, we can perform a more radical adjustment of weights, that is,

$$w_i^{(k+1)} = \begin{cases} 0 & \text{if instance } i \text{ is correctly classified} \\ 1 & \text{otherwise} \end{cases} \tag{22}$$

so that only those instances that were not correctly handled by the previous agent (obtained from the previous iteration) are focused on during the current iteration. What this amounts to is the *hard* partitioning of the inputs into winner-take-all regions, with each region handled by an individual agent, which is the logical extreme of the original boosting method. [7] According to the analysis by Breiman (1996c), which showed that variations of the original boosting algorithm have little impact on performance, we conjecture that this variation should perform approximately as well as the original method proposed by Freund and Schapire (1996). However, in this case, the nature of this method as partitioning inputs becomes apparent.

Third, radial basis function networks constitute yet another way for soft partitioning of the input space (Poggio and Girosi 1990). Such functions have the highest activation at their specified centers and gradually taper off until having near zero activation at a certain distance away from their respective centers. Each of the (non-exclusive and overlapping) regions is handled by an individual agent (i.e., function). The overall classification or prediction can be calculated by weighting individual agents in accordance with the distance from the center of each agent to a particular point in question. Similarly, locally weighted regression (Atkeson et al 1997) can also be viewed as soft partitioning of the input space.

On the other hand, a decision tree (Quinlan 1986) can be viewed as a "hard" partitioning of the input space into multiple (relatively) homogeneous regions in the sense that each region ideally leads to a unique classification. In decision trees, there is no weighting involved. Starting with one node (the root) that contains all the instances, we incrementally create more and more nodes by splitting a current node. The basic idea is that a node should be split to maximally gain information (or equivalently, to reduce entropy). We thus choose the input feature with the maximum information gain for use in splitting. Successive splitting leads to a tree structure. CART (Breiman et al 1984) is similar to decision trees in this regard. It can also be viewed as a form of "hard" partitioning, which is based on minimizing the total variance. Starting with one node (the root) that contains all the instances, we incrementally split nodes by determining

[6] In other words, the vector of weights specifies a graded region with its shape fully determined by weight values: that is, given $W = (w_1, w_2, ..., w_n)$, we obtain a region: $W' = (w'_1, w'_2, ..., w'_n)$, where $w'_1 = \max_i w_i$, and $w'_j = \max_{i \notin \{w'_1, ..., w'_{j-1}\}} w_i$. Thus, W' specifies the instances from the centers to the furtherest boundaries.

[7] In this method, it may be the case that multiple agents are assigned to identical regions or overlapping regions. This is because weights on some or all instances may oscillate back and forth between 0 and 1 and thus generate identical or overlapping regions.

dimensions/types	compatible with	mechanisms
Boundary types:		
soft	overlapping	graded boundary function
hard	overlapping, non-overlapping	0/1 boundary function
Relation to others regions:		
overlapping	WTA, non-WTA; soft, hard	overlapping partition function
non-overlapping	WTA; hard	non-overlapping partition function
Outcome combination methods:		
WTA	soft, hard; overlapping, non-overlapping	WTA
non-WTA	soft, hard; overlapping	averaging or weighted averaging
Instance-to-region assignment:		
deterministic	soft, hard; overlapping, non-overlapping; WTA, non-WTA	deterministic assignment function
probabilistic	hard; WTA, non-WTA; overlapping	stochastic assignment function

Figure 1: Types of partitioning along several dimensions.

a split point so as to minimize:

$$error = \sum_{x:x_i<x_s} (y(x) - avg_{x:x_i<x_s} y(x))^2 + \sum_{x:x_i \geq x_s} (y(x) - avg_{x:x_i \geq x_s} y(x))^2 \qquad (23)$$

where x_i is a dimension of the input, x_s is a split point along that dimension, and $(y(x) - avg_{x:x_i<x_s} y(x))^2$ is the variance of the predicted values on the one side of the splitting point and $(y(x) - avg_{x:x_i \geq x_s} y(x))^2$ is the variance on the other side. We choose an x_s along a dimension i that minimizes that measure. As with decision trees, incrementally splitting nodes leads to a tree structure. (These methods will lead to some new methods for RL to be discussed later.)

In all, many methods exist for partitioning the input space given feedback information from the task to be learned, whether it is classification, prediction, or reinforcement learning (which is to be discussed later). In prediction tasks, the feedback is the value to be predicted; in classification tasks, the feedback is the correct class label; in reinforcement learning tasks, the feedback is (sparse and delayed) reward/punishment, i.e., an indication of how well a sequence of actions achieved its objective. Regardless of the types of feedback, the objective of partitioning is to divide up the world structurally, in ways that best facilitate the performance of the tasks. It is advantageous to group similar inputs into the same regions and separate dissimilar inputs into different regions (i.e., "cutting the world at its joint"). The partition can be either hard, without graded boundaries, or soft, with graded boundaries delineated by either sampling weights (as in boosting) or updating weights (as in gating). Regions may be overlapping (which is a must for soft partitioning), or may be mutually exclusive (which may be the case for some types of hard partitioning; e.g., decision trees). The output can be generated in a winner-take-all (WTA) fashion (including the cases in which each region is handled solely by one agent). On the other hand, in non-winner-take-all (non-WTA) combinations, each region is handled by a set of agents (or all of the agents) with each weighted differently in accordance with its performance characteristics in the region, in order to enhance the overall performance (see section 2 regarding the optimality of weighted averaging; Breiman 1996b). The mapping from an input to a region in a partitioning can be either probabilistic or deterministic. See Figure 1 for the table cataloging these differences.

4 Partitioning and Reinforcement Learning

4.1 Review of Reinforcement learning

First of all, a brief review of single-agent reinforcement learning (see Bellman 1957, Bertsekas and Tsitsiklis 1996, Kaelbling et al 1995) is in order. Assume there is a discrete-time system

$(t = 0, 1, 2, ...)$ in which the state transitions are dependent on controls (or actions) performed by an agent. That is,

$$P : S \longrightarrow U$$

$$T : S, U \longrightarrow S$$

where S is the set of state, U is the set of controls (actions), T is the state transition function that maps the current state and the current control to a new state in the next time step, and P is the (reactive) *policy* that determines the control (action) at the current time step given the current state: $P(x_t) = u_t$. A Markovian process determines a new state x_{t+1} (resulting from a state transition) after control/action u_t is performed in state x_t:

$$prob(x_{t+1}|x_t, u_t, x_{t-1}, u_{t-1}, ...) = prob(x_{t+1}|x_t, u_t) = p_{x_t,x_{t+1}}(u_t) \tag{24}$$

In this process, costs (or its opposite, rewards) can occur at certain states. That is,

$$J(x_0) = \lim_{N\rightarrow\infty} E(\sum_{t=0}^{N-1} \gamma^t g(x_{t+1})|x_0) \tag{25}$$

where J is the cost/reward estimate for a starting state x_0, controls/actions are selected by a fixed policy P, g denotes cost, E denotes expectation, and $\gamma \in (0, 1]$ is a discount factor. The idea of discounting is that costs/rewards incurred in the future matter less than that incurred now; the further off a cost/reward is, the less important it is. When $\gamma = 1$, there is in effect no discount at all.

To find a cost/reward estimate that results from following a particular policy of control (action), there are a number of algorithms, one of which is value iteration, which iteratively updates value estimates J:

$$J(x_t) = \max_u \sum_{x_{t+1}\in S} p_{x_t,x_{t+1}}(u)(g(x_{t+1}) + \gamma * J(x_{t+1})) \tag{26}$$

where x_t is any state and x_{t+1} is the new state resulting from action u (determined by a policy P). The updating of J for different states can be done asynchronously (i.e., states can be updated in any order), as long as all states are updated infinitely often (Bertsekas and Tsitsiklis 1996).

A variation is to keep track of the component inside "*max*" in the right-hand side of this equation, using the notation $Q(x_t, u_t)$:

$$\begin{aligned} Q(x_t, u_t) &= \sum_{x_{t+1}\in S} p_{x_t,x_{t+1}}(u_t)(g(x_{t+1}) + \gamma * J(x_{t+1})) && (27) \\ &= \sum_{x_{t+1}\in S} p_{x_t,x_{t+1}}(u_t)(g(x_{t+1}) + \gamma * \max_u Q(x_{t+1}, u)) && (28) \end{aligned}$$

where u_t ranges over all possible controls/actions for state x_t.

In reinforcement learning (with Q-values as specified above), updating can be done completely on-line, without explicitly using probability estimates. It is done based on actual state transitions; that is, on-line "simulation" is performed. The updating is also incremental, necessitated by the fact that we use only the information about the current state transition. That is,

$$\begin{aligned} Q(x_t, u_t) &:= (1 - \alpha)Q(x_t, u_t) + \alpha(g(x_{t+1}) + \gamma * \max_{u_{t+1}} Q(x_{t+1}, u_{t+1})) && (29) \\ &= Q(x_t, u_t) + \alpha(g(x_{t+1}) + \gamma * \max_{u_{t+1}} Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t)) && (30) \end{aligned}$$

Or

$$\Delta Q(x_t, u_t) = \alpha(g(x_{t+1}) + \gamma * \max_{u_{t+1}} Q(x_{t+1}, u_{t+1}) - Q(x_t, u_t)) \tag{31}$$

where $\alpha \in (0, 1)$ is the learning rate and u_t is determined by an action policy, such as $u_t = argmax_u Q(x_t, u)$, or using the Boltzmann distribution $prob(u_t) = \frac{e^{Q(x_t,u_t)/\tau}}{\sum_u e^{Q(x_t,u)/\tau}}$ (where τ is the temperature). With enough sampling, the transition frequency from x_t and u_t to x_{t+1} should approach $p_{x_t,x_{t+1}}(u_t)$, and thus provides an estimation. Therefore, the results will be the same values as in the earlier specification of Q-values. This updating formula is commonly referred to as Q-learning (Watkins 1989).

With either Q-values or J-values, we need function approximators when state spaces are large. We assume that a neural network, such as a backpropagation network, is used (see, e.g., Lin 1992, Sutton 1990). Although we are aware of potential unpredictability of this type of function approximation (see Boyan and Moore 1995, Sutton 1996), we hope that partitioning can help to remedy the problem, in lieu of using a local approximator such as a RBF network (see Comparisons later).

In relation to other more frequently studied types of learning tasks, we note that the output of a reinforcement learning agent, such as that of Q-learning as described above, can be interpreted in two different ways:

- As prediction. In Q-learning systems, the output $Q(x_t, u)$, where x_t is the current state and u is the chosen action in x_t, can be viewed as predicting the discounted cumulative reinforcement:

  $$Q(x_t, u) = \sum_{t'} \gamma^{t'} g(x_{t+t'})$$

  or equivalently, in terms of step-wise updating, as predicting one-step lookahead values:

  $$Q(x_t, u) = g(x_{t+1}) + \gamma \max_v Q(x_{t+1}, v)$$

  where $g(x_{t+1})$ is the reinforcement received after action u, and x_{t+1} is the new state resulting from action u. However, we should note that in RL, the target for prediction is a moving target, because with learning, $\max_v Q(x_{t+1}, v)$ may change over time. Thus, $Q(x_t, u)$ may need to change as well.

- As classification. In Q-learning systems, $Q(x_t, u)$ indicates, in state x_t, whether an action u is desirable or not. If an action is desirable, then its Q-value should be high, or close to 1. If an action is undesirable, then its Q-value should be low, or close to 0. We assume that stochastic decision making is done based on Q-values in choosing an action to be performed. Thus, the final outcome can be interpreted as stochastic classification decisions. However, this interpretation ignores the point that Q-values are estimates of discounted cumulative reinforcement, and leads to the "coarsening" of Q-values. It can work in some circumstances and may help to simplify learning.

These two interpretations serve as the foundation in the present work for addressing partitioning and weighting in reinforcement learning.

4.2 Partitioning in reinforcement learning tasks

With the above overview of reinforcement learning, we are now ready to extend it to multi-agent learning settings. Reinforcement learning can be difficult, due to, among others things, complex value functions and large state spaces as a result of complex real-world scenarios. Even when function approximation is used for value functions, the accuracy of approximation and the complexity of learning are seriously affected by the complexity and the lack of smoothness of value functions to be approximated, as discussed by e.g. Boyan and Moore (1995). Thus we need to find ways to reduce the complexity of value functions. Partitioning a reinforcement learning task is one way that can help to reduce the complexity and to improve learning (in terms of, e.g., speeding up learning).

Let us discuss all the possibilities of partitioning reinforcement learning tasks, synthesizing various existing proposals (Breiman 1996a, b, c, Wolpert 1992, Jacobs et al 1991, Singh 1994, Tham 1995) and new possibilities:

- Partition the input space (the state space), so that different inputs located in the different regions of the state/input space can be handled by different agents (e.g., the gating model; see Jacobs et al 1991, Jordan and Jacobs 1994).

- Partition a sequence, so that each subsequence is handled by a different agent (e.g., Wiering and Schmidhuber 1996, Singh 1994, Tham 1995, Thrun and Schwartz 1995). The partitioning of subsequences can either be predetermined (as in Singh 1994), or better yet, automatically determined as part of reinforcement learning (as in Wiering and Schmidhuber 1996).

- Partition actions (i.e., the action space, when we deal with action-oriented tasks), so that each agent will be responsible for only certain limited types of actions. For example, Dayan and Hinton (1993) limited an agent to certain actions at a certain level of abstraction. Sun et al (1996) and Sun and Peterson (1997) divided actions into two types, speed and turn, and each type was handled by a separate agent.

- Partition the goal of a task, when there are multiple *explicit* goals to be achieved in reinforcement learning. For example, in Dayan and Hinton (1993), the goal of an agent is determined on the fly, through the action of a higher-level agent. In Reddy and Tadepalli (1997), the partitioning of the goal is learned through using externally provided examples through depth-first search. Static partitioning of goals can also be done (e.g., Sun et al 1996, Sun and Peterson 1997, 1998).

- Partition the reinforcement (in reinforcement learning), so that certain reinforcements are given in association with achieving certain aspects of a goal (e.g., Sun et al 1996, Mataric 1995). This can be viewed as a form of the partitioning of goals.

- Partition outputs. We divide up outputs into multiple sets (either overlapping or disjoint, either probabilistically or deterministically). The partitioning can be done in a variety of ways: for example, (1) based on a set of equal agents each over the entire input space (which is the simplest way, discussed in section 2; see also Breiman 1996a regarding "bagging"), (2) based on inputs (and thus the input space is also partitioned and the partitioning of outputs is the result of partitioning inputs; e.g., see Jacobs et al 1991; see also the discussion later of our methods), or even (3) based on the particular outputs of some subsets of agents (e.g., see Erickson and Kruschke 1996). [8] The combination of the outputs of the agents

[8]In Erickson and Kruschke (1996), which involves the combination of "exemplar nodes" and "rule nodes", gating is determined based on the "exemplar nodes", independent of the "rule nodes".

involved in partitioning can be done by averaging (Breiman 1996a), by weighted averaging (Breiman 1996b), or even by more complex methods (Wolpert 1992). [9]

As has been discussed in the two previous sections, in this work, we are mainly concerned with partitioning outputs through averaging and weighted averaging (i.e., weighting), and on top of that, partitioning the input space for the sake of better weighted averaging (or a better single agent, as an extreme case) in each region. [10]

The basic motivation for partitioning inputs/outputs in RL is that it can make learning easier, especially when a neural network is used for each agent. Divide-and-conquer is generally a good idea and can lead to improved performance. In particular, Whitehead (1993) showed that learning time in RL is dependent on the size of the state space and the minimum distance between the starting state and the goal state. By dividing the input/state space into multiple (more or less) separate subspaces, the learning in each subspace is facilitated because of the smaller size of a subspace, and potentially the overall learning is also facilitated. Another way to see this (when neural networks are used) is that, since the ease of convergence of a BP network is dependent on the number of input/output patterns to be learned and the complexity of the set of patterns (i.e., the complexity of the underlying function; Hertz et al 1991), by dividing the set of training patterns into multiple (more or less disjoint) sets and using a separate network for learning each of them, we have less patterns for each network and thus likely improve the convergence of the networks. In addition, due to the tendency in BP networks of smoothing the outputs (because of generalization), more homogeneous patterns being assigned to each network will lead to more accurate outputs (generalization). Partitioning does lead to assigning similar patterns to each network, and thus more homogeneous mappings are to be learned by each network.

5 Improving Partitioning for Reinforcement Learning

Given the perspective in the previous sections, there are the following two possibilities for achieving better partitioning:

- On-line optimization: on-line, soft partitioning can be carried out based on a chosen optimality criterion/method, such as least squares (LS), maximum likelihood (ML) (Jordan and Jacobs 1994), or expectation-maximization (EM) (Jordan and Jacobs 1994). Partitioning is done on-line, at the same time as individual agents are trained, and it is learned as part of optimizing the same criterion as that used in training individual agents. Gradient descent/ascent is usually used to minimize/maximize a criterion.

- Off-line optimization: partitioning is obtained separately from the training of individual agents, but based on the performance of these agents (during training or during a separate testing/cross-validation phase).

Below we will explore some possibilities in on-line and off-line optimization for reinforcement learning, inspired by ideas discussed in sections 2 and 3.

[9] In case of reinforcement learning tasks, we partition the Q (or J) value outputs so that they combine to generate values that are proper or even optimal in some sense (in terms of either a prediction or classification interpretation; see the more detailed discussion later in section 5.1).

[10] Other types of partitioning, e.g., partitioning sequences, actions, goals, and reinforcement, are dealt with elsewhere; see e.g. Sun et al (1996), Sun (1997), and Sun and Peterson (1997, 1998).

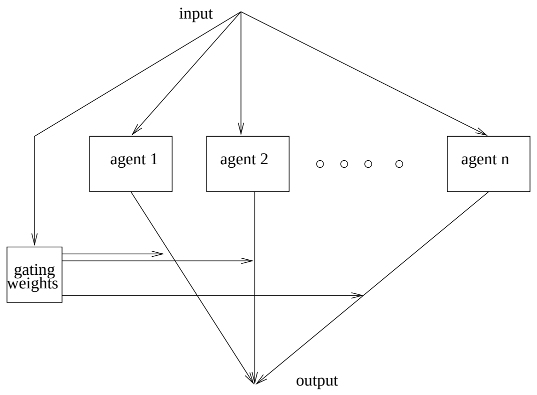


Figure 2: On-line partitioning (gating).

5.1 On-line Optimization

Let us discuss *on-line* optimization first. The general structure of the system for soft, on-line partitioning with multiple agents (i.e., gating) is shown in Figure 2. In this system, we assume that the outputs (Q-values) from multiple agents are combined using the weighted average, and then the Boltzmann distribution is used to select an action based on the combined outputs (Q-values).

In on-line optimization, the learning rule for partitioning is often derived from the same learning criterion in exactly the same way as the learning rule for individual agents. We will look into various possibilities for a learning criterion along several dimensions: target values (i.e., the desired outputs, which hinge on whether we view reinforcement learning as a prediction task or a classification task; see section 4), error functions (measuring the overall difference between target values and actual outputs; see section 2), and optimization methods (such as LS, ML, or EM), based on which gradient descent/ascent is then used to derive learning rules.

First of all, target values can be the Q-values for the new states (discounted or not, along with reinforcements received), in which case the task is viewed as predicting such Q-values at each step of a task (see the earlier discussion regarding the two views of Q-values); or they can be either 0 or 1 depending on a measure that determines the desirability of the action, in which case the task is viewed as a classification task in which at each step (state) we separate desirable actions (which are "positive instances" and have a target value close to 1) from undesirable actions (which are "negative instances" and have a target value close to 0). Let $d_k(x_t, u)$ denote the Bellman residual (i.e., the Q-learning updating amount) for agent k. With the prediction task interpretation, we define

$$d_k(x_t, u) = \gamma \max_v Q_k(x_{t+1}, v) + g(x_{t+1}) - Q_k(x_t, u) \tag{32}$$

where x_{t+1} is the state resulting from action u in state x_t. With the classification task interpretation, we define

$$d_k(x_t, u) = class(x_{t+1}) - Q_k(x_t, u), \tag{33}$$

where $class(x_{t+1}) = 1$ if a certain criterion of success is met [11] or $class(x_{t+1}) = 0$ if otherwise.

[11] The criterion is: $\gamma \sum_k \frac{w_k(x_{t+1})}{\sum_j w_j(x_{t+1})} * \max_v Q_k(x_{t+1}, v) + g(x_{t+1}) - \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} * Q_k(x_t, u) > \nu$. See Sun and Peterson (1998) for the discussion of its justification.

Either as a prediction difference or as a misclassification, the quantity d_k is a local error measure for agent k, The global error measure can be derived as a combination of local error measures.

In terms of deriving a global error measure used for deriving learning rules, there are a number of possibilities (see Section 2 for justifications concerning these methods). First we consider combining separate error measures of individual agents (that is, the *local-error* approach). We can combine local errors based on either the weighted average (Breiman 1996b) or the exponentiated weighted average (Jacobs et al 1991), both of which treat each agent as a separate entity and tries to minimize each of their errors separately in proportion to their contributions to the overall error (the justifications in terms of reducing the error by the weighted average of outputs as well as reducing the variance of each agent were discussed in section 2; Krogh and Vedelsby 1995). Different from the analysis in section 2, here we use normalized weights, because it is easier this way to keep the total (normalized) weight at 1, which was assumed in the previous analysis (section 2), and to create a form of competition among agents. That is, the combined error is defined as follows:

$$error(x_t, u) = \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} (d_k(x_t, u))^2 \tag{34}$$

or

$$error(x_t, u) = -log \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} e^{-(d_k(x_t,u))^2} \tag{35}$$

Another way is to use an overall-error measure directly (that is, the *overall-error* approach; section 2), instead of summing individual error measures in ways that lead to trying to minimize each individually. In this case, the overall-error measure is the squared overall Bellman residual:

$$error(x_t, u) = (d^o(x_t, u))^2 \tag{36}$$

where

$$d^o(x_t, u) = \gamma \sum_k (\frac{w_k(x_{t+1})}{\sum_j w_j(x_{t+1})} * \max_v Q_k(x_{t+1}, v)) + g(x_{t+1}) - \sum_k (\frac{w_k(x_t)}{\sum_j w_j(x_t)} * Q_k(x_t, u)) \tag{37}$$

if we use a prediction task interpretation. That is, we generate a prediction of the weighted average of the next-state Q-values from the agents by a weighted average of individual predictions, instead of each agent predicting its own next-state Q-value (the justification of it in terms of reducing the average error by the weighted average of outputs was discussed in section 2; Breiman 1996b). [12] With a corresponding classification task interpretation, we have the following overall Bellman residual:

$$d^o(x_t, u) = class(x_{t+1}) - \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} * Q_k(x_t, u) \tag{38}$$

where x_{t+1} is the next state resulting from action u in state x_t and $class(x_{t+1})$ is determined in ways specified earlier. The overall-error measure has the characteristics of encouraging

[12]Note that $\frac{w_k(x_{t+1})}{\sum_j w_j(x_{t+1})}$ is different from $\frac{w_k(x_t)}{\sum_j w_j(x_t)}$ if $x_t \neq x_{t+1}$, because weights are dependent on states/inputs. Thus $\gamma \sum_k \frac{w_k(x_{t+1})}{\sum_j w_j(x_{t+1})} * \max_v Q_k(x_{t+1}, v) + g(x_{t+1}) - \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} * Q_k(x_t, u) \neq \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} * d_k(x_t, u)$.

the agents to cooperate with each other (in making prediction or classification), rather than to compete with each other as in the case of the first two measures (Jacobs et al 1991).

In terms of optimization methods, we can use e.g. least squares (LS), maximum likelihood (ML), or expectation-maximization (EM). We will limit ourselves to LS, because ML and EM require some assumptions regarding underlying probability distributions and thus lead to parametric statistical models (Jordan and Jacobs 1994). [13] For deriving learning rules based on LS, we may use steepest descent, Newton's method, Quasi-Newton method, Gauss-Newton method, or other numerical methods (Bertsekas and Tsitsiklis 1996). Here we will focus on (on-line) incremental steepest descent approach (i.e., updating is done based on gradients after only one or a few steps). We need some form of soft partitioning with graded boundaries determined by weights, for the sake of calculating the gradients of partitioning. We need to derive the learning rules for two types of weights: combination weights w_k and individual agent network weights w^k_{net} for agent k.

Let us first look into the weighted-average-of-local-errors measure. Using incremental steepest descent, after experiencing (x_t, u) (i.e., performing action u in state x_t), we update the two types of weights as follows: [14]

$$\Delta w_k(x_t) = \alpha(-\frac{\partial error(x_t,u)}{\partial w_k(x_t)}) = \alpha * \frac{\sum_{j\neq k} w_j(x_t)(d_j(x_t,u)^2 - d_k(x_t,u)^2)}{(\sum_j w_j(x_t))^2} \tag{39}$$

$$\Delta w^k_{net} = \beta(-\frac{\partial error(x_t,u)}{\partial Q_k(x_t,u)}\frac{\partial Q_k(x_t,u)}{\partial w^k_{net}}) = \beta * d_k(x_t,u) * \frac{w_k(x_t)}{\sum_j w_j(x_t)} * \frac{\partial Q_k(x_t,u)}{\partial w^k_{net}} \tag{40}$$

where α and β are learning rates. When the gating weights $w_k(x_t)$ are generated using a lookup table, then $\Delta w_k(x_t)$ can be viewed as the amount of updating applied to the corresponding entry of the table. When the gating weights $w_k(x_t)$ are generated by a gating network (based on inputs x_t), $\Delta w_k(x_t)$ can be viewed as the error to be minimized by the gating network. A learning rule concerning the internal weights of the gating network (not specified here) can then be easily derived. Similarly, the learning rule for Δw^k_{net} depends on the neural network implementing Q-learning, because the derivative $\frac{\partial Q_k(x_t,u)}{\partial w^k_{net}}$ depends on the particular structure of the neural network chosen for an agent.

Using the exponentiated weighted-average-of-local-errors measure, we similarly derive the following incremental steepest descent rules:

$$\Delta w_k(x_t) = \alpha(-\frac{\partial error(x_t,u)}{\partial w_k(x_t)}) = \alpha * \frac{1}{\sum_j w_j(x_t) e^{-d_j(x_t,u)^2}} * \frac{\sum_{j\neq k} w_j(x_t)(e^{-d_k(x_t,u)^2} - e^{-d_j(x_t,u)^2})}{\sum_j w_j(x_t)} \tag{41}$$

$$\Delta w^k_{net} = \beta(-\frac{\partial error(x_t,u)}{\partial Q_k(x_t,u)}\frac{\partial Q_k(x_t,u)}{\partial w^k_{net}}) = \beta * d_k(x_t,u) * \frac{w_k(x_t) e^{-d_k(x_t,u)^2}}{\sum_j w_j(x_t) e^{-d_j(x_t,u)^2}} * \frac{\partial Q_k(x_t,u)}{\partial w^k_{net}} \tag{42}$$

[13]In addition, although EM and ML may learn faster, there is no indication of better eventual performance compared with LS (Jordan and Jacobs 1994).

[14]In deriving the learning rules, in accordance with either the prediction or the classification interpretation, we treated the target values as constants, although strictly speaking they were not. The same applies to other derivations later.

Using the overall-error measure, we have the following learning rules:

$$\Delta w_k(x_t) = \alpha(-\frac{\partial error(x_t,u)}{\partial w_k(x_t)}) = \alpha * d^o(x_t,u) * \frac{\sum_{j\neq k} w_j(x_t)(Q_k(x_t,u) - Q_j(x_t,u))}{(\sum_j w_j(x_t))^2} \tag{43}$$

$$\Delta w_{net}^k = \beta(-\frac{\partial error(x_t,u)}{\partial Q_k(x_t,u)}\frac{\partial Q_k(x_t,u)}{\partial w_{net}^k}) = \beta * d^o(x_t,u) * \frac{w_k(x_t)}{\sum_j w_j(x_t)} * \frac{\partial Q_k(x_t,u)}{\partial w_{net}^k} \tag{44}$$

In some cases, combination weights tend to keep falling or rising simultaneously. Thus, competition among weights is necessary in order to keep the total weight constant to avoid saturation (underflow or overflow). We derived heuristically the following competition: given weight updating amount $\Delta w_k(x_t)$, the new updated weight is $w_k^{t+1}(x_t) = w_k^t(x_t) + \Delta w_k(x_t)$, where t indicates the current step. Then the *adjusted* weight updating amount is :

$$\Delta' w_k(x_t) = \frac{w_k^{t+1}(x_t)}{\sum_k w_k^{t+1}(x_t)} - w_k^t(x_t) = \frac{w_k^t(x_t) + \Delta w_k(x_t)}{\sum_k (w_k^t(x_t) + \Delta w_k(x_t))} - w_k^t(x_t) \tag{45}$$

which keeps the total at 1. Although this is not a principled solution, it helps the computational implementation.

Care must also be taken in setting learning rates for the gating network and the agent networks and in setting their respective schedules of changes. In general, we want to initialize the learning rate of the gating network to a higher value compared with that of the agent networks, and then quickly reduce it to a very low value (or 0), so that gating can learn and stabilize faster than the agent networks, so as to provide a stable structure (a stable division of subtasks) in which agents can learn and become stable (see Experiments).

Given the current Q-values, the decisions on actions to be performed at each step are made in the following way. At each step, each agent receives the same input and computes its outputs (i.e., Q-values). Then weighted averaging is used for combining the Q-values of all the agents: $Q(x_t,u) = \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} Q_k(x_t,u)$. An action is then selected based on the maximum Q-value (i.e., $u_t = argmax_u Q(x_t,u)$) or the Boltzmann distribution of Q-values.

The complete but generic specification of the algorithm is in Figure 3. The details of the learning rules and error measures (and their different interpretations and implications) have been discussed above in text and thus omitted from the specification in the figure. The optimality of the above methods in terms of minimizing the total error is only guaranteed to the extent that the (on-line) incremental steepest descent method is likely to be optimal with respect to find local minima/maxima.

Note that different inputs can be given to the gating network compared with those given to individual agents. This may potentially improve learning, since potentially different information may be needed for individual agents performing a task and the gating mechanism that assigns subtasks to different agents. The inputs for partitioning must reflect the useful structure of the state/input space of a task, and enable meaningful division of the space into regions. For example, instead of a local view that is given to agents as their inputs, we can use x-y coordinates for partitioning (see Experiments). The inputs can be either full or partial descriptions of current states, and thus either full state-based or partial observation-based RL may be used. (This is essentially the same for off-line methods, to be discussed next.)

```
1. Each agent receives the same input and computes its output Q-values
2. Combine the Q-values of all the agents through weighted averaging: $Q(x_t, u) = \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} Q_k(x_t, u)$. An action is then selected based on the maximum Q-values or the Boltzmann distribution
3. Perform the selected action
4. Each agent receives the same new input and computes its new Q-values
5. Generate the global error with one of the following two ways:
5.1. generate a local error measure for each agent based on either the Q-updating formulas: $\gamma \max_v Q_k(x_{t+1}, v) + g(x_{t+1}) - Q_k(x_t, u)$, or the abstract Q-updating formula: $class(x_{t+1}) - Q_k(x_t, u)$. Then combine the local errors based on either the weighted average or the exponentiated weighted average
5.2. generate a global error measure directly based on the weighted combined Q-updating formula: $\gamma \sum_k \frac{w_k(x_{t+1})}{\sum_j w_j(x_{t+1})} * \max_v Q_k(x_{t+1}, v) + g(x_{t+1}) - \sum_k \frac{w_k(x_t)}{\sum_j w_j(x_t)} * Q_k(x_t, u)$
6. Apply an appropriate updating rule derived from the adopted global error measure to the gating network, adjusting its weights
7. Apply an appropriate updating rule derived from the adopted global error measure to the individual agents, adjusting their internal weights
8. Go to step 1
```

Figure 3: The on-line gradient descent algorithm.

5.2 Off-line Optimization

Off-line optimization provides better alternatives for partitioning the state/input space (for the sake of producing better overall performance). This is because off-line methods enable a learning/partitioning decomposition [15] — separating the two issues and optimizing them separately, which facilitates the whole task. There are many possible ways for off-line optimization of partitioning. However, what is especially important is the fact that we can use *hard* partitioning in off-line optimization; this is because in the off-line case, we do not need to calculate gradients of partitioning and are thus free to choose either hard or soft partitioning, and hard partitioning is easy to do off-line. [16] In addition, according to the analysis by Meir (1995), in many cases, hard partitioning with *non-overlapping* regions can be superior to overlapping (hard or soft) partitioning. Krogh and Vedelsby (1995) also showed the advantage of hard, non-overlapping partitioning, based on their analysis of ensemble generalization with the decomposition of the generalization error into the weighted average of individual generalization errors and the ensemble ambiguity (see section 2). An added advantage of off-line partitioning (the learning-partitioning decomposition) is that we can use different criteria, based on different inputs, for partitioning, different from the learning of agents (and their specific inputs and learning criteria).

We can sum up the main process behind different methods for off-line, hard, non-overlapping partitioning in a unified way: The algorithm is described in generic terms in Figure 5. The algorithm basically generates one or more plausible partitions and test agents on the partition(s) to measure performance; based on the performance, it selects one or more better partitions to pursue further, by generating variations of these selected partitions for testing; the loop goes on

[15] Here, learning refers to the learning of individual agents.

[16] This is unlike the case of on-line optimization, in which the use of gradient descent basically dictates that soft partitioning be used (otherwise, it would be difficult, if not impossible, to calculate gradients).

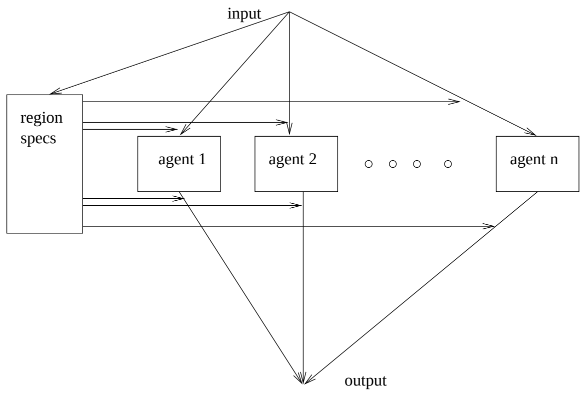


Figure 4: Off-line partitioning.

until some termination conditions are satisfied. This algorithm is, in a way, a variation of the gating model of Jacobs et al (1991), but it is off-line in terms of partitioning the input/state space while the gating method is on-line. The general structure of a multi-agent system when hard, non-overlapping partitioning is used is shown in Figure 4. We assume that at each step, the input to the system is checked against region specifications and directed to the agent responsible for handling the region to which the input belongs. The output (the Q-values) from the selected agent is used as the final outcome of the whole system. The Boltzmann distribution is used to select an action based on these Q-values. We use the prediction task interpretation here for agent learning.

In terms of means for generating partitions and for selecting better partitions, we have the following possibilities:

- We can apply gradient descent/ascent (based on LS, ML, or EM methods) in a batch mode (with large batches). For example, assuming a gating network is used, while we train individual agents in a completely on-line fashion, we train the gating network after each long period of training of individual agents during which we collect updates that are later used to adjust the gating network.
- Or, if the space of all possible partitionings is relatively small, we can perform an explicit search of possible partitionings, e.g., with hill-climbing.
- A more promising search technique is the genetic algorithm (GA), which can be either applied to the gating network (with weights of the gating network being mutated and crossed over during search), or applied directly to partitions (with regions of a partition being mutated and crossed over).
- Yet another possibility is a decision tree like procedure, in which progressive splitting is applied that leads to an optimal partitioning of the input/state space (Sanger 1991, Blanzieri and Katenkamp 1996). The criterion for splitting can be based on one of the error functions mentioned earlier.

Let us discuss the last two possibilities below. First, the decision tree like region-splitting algorithm is described in Figure 6. In the algorithm, a region in a partition (non-overlapping and with hard boundaries) is handled exclusively by a single agent (a neural network) (see Figure 4).

The algorithm looks at one partition at a time and attempts to find a better partitioning by splitting regions incrementally when a certain criterion is satisfied (e.g., Quinlan 1986, Breiman et al 1984). The splitting criterion is based on the total magnitude of the errors that incurred in a region during training and also based on the consistency of the errors (which concerns the distribution of the directions of the errors, either positive or negative). These two considerations can be combined (Blanzieri and Katenkamp 1996). Specifically, in the context of Q-learning, error is defined as the Q-value updating amount (the Bellman residual). We select those regions to split that have high *sums of absolute errors* (or alternatively, sums of squared errors), which are indicative of the high magnitude of the errors (the Bellman residuals), but have low *sums of errors*, which together with high sums of absolute errors are indicative of low error consistency (i.e., that Q-updates/Bellman residuals are distributed in different directions; Blanzieri and Katenkamp 1996, Sanger 1991). That is, our combined criterion is

$$consistency(r) = |\sum_{x \in r} error(x)| - \sum_{x \in r} |error(x)| < threshold_1 \tag{46}$$

where x refers to the data points encountered during previous training that are within the region r to be split. We define $error(x) = \max_{u'} Q_{k'}(x', u') + g(x') - Q_k(x, u)$, where x is a (full or partial) state description, u is the action taken, x' is the new state resulting from action u in state x, k is the agent responsible for x, and k' is the agent responsible for x'.

Next, we select a dimension to be used in splitting, within each region to be split. Instead of being random, we again use the heuristics of high sums of absolute errors but low error consistency. Since the sum of the absolute errors remains the same regardless what we do, what we can do is to best split a dimension to increase the overall error consistency, i.e., the sums of errors (which is analogous to CART; see Breiman et al 1984). Specifically, we compare for each dimension i in the region r the following measure: the increase in consistency if a dimension is optimally split, that is,

$$\Delta consistency(r, i) = \max_{v_i}(|\sum_{x \in r: x_i < v_i} error(x)| + |\sum_{x \in r: x_i \geq v_i} error(x)|) - |\sum_{x \in r} error(x)| \tag{47}$$

where v_i is a split point for a dimension i, x refers to the points within region r on the one side or the other of the split point, when projected to dimension i. This measure indicates how much more we can increase the error consistency if we split a dimension i optimally. The selection of dimension i is contingent upon

$$\Delta consistency(r, i) > threshold_2 \tag{48}$$

Among those dimensions that satisfy $\Delta consistency(r, i) > threshold_2$, we choose the one with the highest $\Delta consistency(r, i)$. For a selected dimension i, we then optimize the selection of a split point v_i' based on maximizing the sum of the absolute values of the total errors on both sides of the split point:

$$v_i' = argmax_{v_i}(|\sum_{x \in r: x_i < v_i} error(x)| + |\sum_{x \in r: x_i \geq v_i} error(x)|) \tag{49}$$

where v_i' is the split point for dimension i. Such a point is optimal in the exact sense that error consistency is maximized (Breiman et al 1984). Then, we split the region r using a boundary created by the split point: We create a split hyperplane using the selected point $spec = x_j \; < \; v_j$.

1. Initialize one, or a set of, partition(s), each of which contains one or more regions that together cover the entire input/state space
2. Train a set of agents on each of the partition(s); record the performance on each partition
3. Modify the partition(s) through operations such as merge, split, and recombination, based on the performance on each partition
4. Train a set of agents again on each of the new partition(s); record the performance on each partition
5. If a termination condition is satisfied, stop; else, go to 3

Figure 5: The off-line optimization algorithm. Here is the generic description of off-line optimization. Specific instances will be described later.

We then split the region using the hyperplane: $region_1 = region \cap spec$ and $region_2 = region \cap \neg spec$, where *region* is the specification of the original region. Replicating the existing agent for handling the old region, two new agents are created to handle the two new regions. After splitting a region, if the number of regions exceeds R, we combine some (randomly selected) existing regions until the number is right (preferring adjacent regions for combination). After a combination of two regions, one of the two agents involved is deleted.

From the above description, this algorithm is clearly related to a number of partitioning algorithms we examined in section 3, such as decision trees (Quinlan 1986) and CART (Breiman et al 1984), in addition to being inspired by the gating model (Jacobs et al 1991). Note that it is different from stochastic hard partitioning (such as Singh et al 1994), which assigns inputs to different regions probabilistically and is better suited for on-line partitioning (using e.g. gradient descent).

The region-splitting algorithm is domain independent in the sense that no domain knowledge (e.g., concerning which dimension to split and at which point) is needed. However, domain knowledge, when available, can be useful. For example, in a navigation setting, if we know which input dimension (such as a particular instrument reading) is more important, then we can use that dimension first. This way not only we are more likely to find a good partition, but also we can save much computation.

Let us now turn to an algorithm more complex than region-splitting, a GA-inspired algorithm with direct manipulations of regions, which is described in Figure 7. It is more complex, because it considers simultaneously a *set* of different partitions and then select a subset of them, modify them, and test them further. It thus involves a much larger search space. In this algorithm, we use two types of operations inspired by GA: mutation and crossover. In the *mutation* operation, the selection of regions to be split, dimensions to be split and split points as well as the selection of regions to be combined are guided by the same criteria prescribed for the previous algorithm (which will not be repeated, but see Figure 6). Thus, different from the usual GA, mutation is not random but follows the consistency measure; mutation therefore has a strong likelihood of improving a partition. Because of this, mutation can be safely applied to all the partitions in a population. In the *crossover* operation, we use the performance of a partition (i.e., the success rate) as the "fitness" value in selecting partitions to be crossed over. But the selection of the split hyperplane for crossover (see Figure 7) is random. Duplication operations in the usual GA are not explicitly specified here but implied in the *crossover* operation (as detailed in Figure 7): Successful partitions will be copied into the next generation. See Figure 7 for the full details of

```
1. Initialize one partition to contain only one region that covers the whole input/state space
2. Repeat for $n$ times (each time with a different randomly generated initial condition):
   Train an agent on the partition for $m$ trials.
3. Further split the partition
4. Repeat for $n$ times (each time with a different randomly generated initial condition for
   each agent): Train a set of agents, with each region assigned to a different agent, each
   for $m$ trials.
5. If no more splitting can be done, stop; else, go to 3

Further splitting a partition:

   For each region that satisfies $consistency(r) < threshold_1$ do:
   1. Select a dimension $j$ in the input/state space that maximizes $\Delta consistency$, provided
      that $\Delta consistency(r, j) > threshold_2$
   2. In the selected dimension $j$, select a point (a value $v_j$) lying within the region and
      maximizing $\Delta consistency(r, j)$
   3. Using the selected point in the selected dimension, create a split hyperplane: $spec =$
      $x_j < v_j$
   4. Split the region using the newly created hyperplane: $region_1 = region \cap spec$ and
      $region_2 = region \cap \neg spec$, where $region$ is the specification of the original region; create
      two new agents for handling these two new regions by replicating the agent for the original
      region
   5. If the number of regions exceeds $R$, keep combining regions until the number is right:
      randomly select two regions (preferring two adjacent regions) and merge the two; keep
      one of the two agents responsible for these two regions and delete the other
```

Figure 6: The region-splitting algorithm. An instance of off-line optimization. See text for details.

1. Randomly generate a population (i.e., a set) of different partitions, each of which contains a certain number of regions (randomly determined between 1 and the upper limit R)
2. Repeat for n times (each time with a different randomly generated initial condition for each agent): Train a set of agents on a corresponding partition in the partition population (with each region in the partition assigned to a different agent), each for m trials; do so for each partition. Record the average performance on each partition
3. Perform *crossover* on the entire population of partitions; perform *mutation* on each partition in the resulting population
4. If the number of iteration is less than a preset limit, goto 2; else, stop

See Figure 8 for further details.

Figure 7: The GA-based algorithm. An instance of off-line optimization.

this simplified version of GA.

In all of these above algorithms, regions are made up of hypercubes, each of which is specified by a logical conjunction of simple inequalities each concerning one of the input dimensions. Beside such a simple type of region, we can also use alternative types of regions, for example, hyperspheres as specified by radial basis functions. Or we can use a linear mapping (e.g., by using a linear perceptron network for specifying regions). We can even extend to more general, arbitrarily shaped regions as, for example, specified by a backpropagation network (which can serve as a gating mechanism). Such an extension would be analogous to extending weighted averaging to general "stacking" (Wolpert 1992), although it is on the input side, as opposed to stacking which is on the output side.

If we measure optimality of the above partitioning algorithms by the resulting overall performance (for example, based on a pre-set success rate criterion, the number of steps or trials to reach a success rate criterion, or transfer performance to test generalization abilities), there is no guarantee of optimality. The quality of the results from the region-splitting algorithm is dependent on the quality of the heuristics used. The quality of the results from the GA based algorithm relies on the optimality of GA per se. Empirical evidence in the literature in numerous domains indicates that GA is a good weak search/learning algorithm although there is no formal guarantee of being optimal. On the other hand, if we measure optimality by the maximization of $\Delta consistency$ (and by the increase of *consistency*, which is a result of the maximization of $\Delta consistency$), then there is some potential: if only local minima are sought, the region-splitting algorithm is optimal in this sense. [17]

While the above description of the specific algorithms is for hard partitioning, soft partitioning as in the case of the on-line algorithms is also possible. In a way this is a generalization of bagging and boosting algorithms, as well as a generalization of the gating method, due to the use of overlapping regions. However, computationally there is little justification for it since we are doing partitioning off-line and thus are not required to use soft partitioning. Another alternative is to keep hard partitioning but use multiple agents in each region. This way we can maintain the simplicity of hard, off-line partitioning but at the same time improve the overall

[17] Note that for the sake of simplicity, we assumed that errors were fixed values to be distributed to different regions and thus to different agents in order to reduce inconsistency; in reality these values change as a function of partitioning and are not fixed, which further complicates analysis.

```
Crossover:
    Do the following for $l$ times (where $l$ is randomly selected between 0 and $l_2$) over the entire
        current-generation population:
    1. Select two partitions: $p1$ and $p2$, with the probability of selecting a partition determined
        by a Boltzmann distribution of success rates of different partitions
    2. Divide the input space into two half spaces ($s1$ and $s2$):
    2.1. Randomly select a dimension $i$ and a split point $v_i$ in the dimension that can be used
        to split both partitions;
    2.2. For any $x$: if $x_i > v_i$, then $x$ belongs to s1; otherwise, it belongs to s2;
    3. Crossover $p1$ and $p2$ (by combining all the regions in $p1$ that are in $s1$ and all the regions
        in $p2$ that are in $s2$): any $region$ is in the resulting partition, if and only if $region \in p1$
        and $region \in s1$ or $region \in p2$ and $region \in s2$ (any $region$ that crosses the border
        between $s1$ and $s2$ is pre-split into two regions in the two half spaces respectively); each
        region keeps its corresponding agent
    4. If the number of regions exceeds $R$, do the following until the number of regions is at or
        below $R$: select two regions (preferring adjacent regions) and combine them; keep one of
        the two agents responsible for the two original regions and delete the other
    5. Put the resulting partition into the next-generation population
    6. Remove an existing partition that has the lowest performance from the current-generation
        population
    In the end, move all the remaining partitions in the current-generation population into the
        next-generation population
Mutation:
    For each partition, repeat for $l$ times (where $l$ is randomly selected between 0 and $l_1$):
    1. Split a selected region, along a selected dimension at a selected point (using the prescribed
        criteria; see Figure 6 for details); create two new agents for handling these two regions
        by replicating the agent for the original region
    2. If splitting leads to exceeding the limit on the number of regions in a partition ($R$),
        combine two randomly selected regions in the partition (preferring adjacent regions) (see
        Figure 6); keep one of the two agents responsible for the two combined regions
Random generation of a partition (of $R'$ regions):
    Initialize the partition to contain one region $region_1$ that covers the entire input/state space.
    For i= 2 to $R'$ do:
    1. Randomly select a region $r$ where $r \in [1, i-1]$
    2. Randomly select one dimension $j$ in the input/state space that can be used to split region
        $r$
    3. Randomly select a point (a value $v_j$) in dimension $j$ that can be used to split region $r$
    4. Create two regions out of the original region $r$ by using the selected value point: $spec_r :=$
        $spec_r \cap x_j < v_j$ and $spec_i := spec_r \cap x_j \geq v_j$ (whereby one new region is used to replace
        the original region and the other new region is set to be region $i$)
    Create $R'$ agents, each responsible for one region.
```

Figure 8: Some details of the GA-based algorithm.

```
1. Initialize the partition to contain only one region that covers the whole input space
2. Train a set of K agents on the region.
3. Further split the partition
4. Repeat for $n$ times (each time with a different randomly generated initial condition for
   each agent): Train K agents on each region, while learning a set of combination weights
   on each region.
5. If no more splitting can be done, stop
6. Else, go to 3

Further splitting a partitioning (up to a total of R regions): see Figure 6.
```

Figure 9: The region-splitting algorithm with weighted averaging. A special case of off-line optimization.

performance in each region by combining the outcomes of multiple agents (see section 2; Breiman 1996 a,b). For example, consider an extension to the region-splitting algorithm: the region-splitting algorithm with weighted-averaging (i.e, with weighted averaging being added to the region-splitting algorithm described in Figure 6) shown in Figure 9. Since this is a straightforward extension, we will not repeat the details of the region splitting algorithm. However, the learning rules need to be specified. For combination weights in each region, using incremental steepest descent on the weighted-average-of-local-errors measure, we obtain

$$\Delta w_k = \alpha * \frac{\sum_{j \neq k} w_j (d_j(x_t, u)^2 - d_k(x_t, u)^2)}{(\sum_j w_j)^2} \tag{50}$$

Different from the gating algorithm (see section 5.1 and Figure 3), however, is the fact that the combination weights w_k's are not completely input-dependent; rather, they are *uniform* throughout a region. We do not need to consider partitioning when we learn the combination weights — the partitioning is done separately, off-line, with hard boundaries. For internal weights of agents in a region, we have,

$$\Delta w_{net}^k = \beta * d_k(x_t, u) \frac{w_k}{\sum_j w_j} * \frac{\partial Q_k(x_t, u)}{\partial w_{net}^k} \tag{51}$$

The derivative $\frac{\partial Q_k(x_t,u)}{\partial w_{net}^k}$ is dependent on the network type and structure chosen for an agent. Other error measures discussed before can also be adopted; see section 5.1 for details. The overall performance from combining these agents can benefit from the diversity among these agents (see Appendix) as in the case of weighted averaging (section 2).

6 Experiments

6.1 Tasks

We looked into two maze tasks, with different difficulty levels in their layouts, one easy and one hard. For each maze, we also varied the size of the layout. Maze 1 of the small size is shown in Figure 10. Maze 2 of the small size is shown in Figure 11. The median size is the double of the original size, and the large size is 3 times the original size. We performed experiments with all the combinations of the task parameters: difficulty level of maze layout (easy, hard), and size

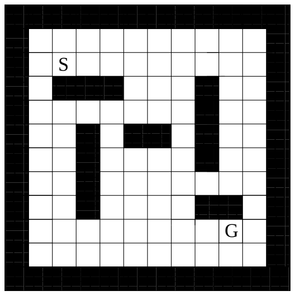


Figure 10: The easy maze.

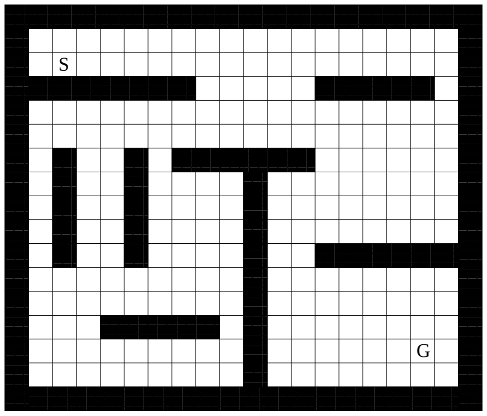


Figure 11: The hard maze.

of maze layout (large, median, small). The task setting is as follows: an agent has views of five sides: left, soft left, front, soft right, and right, and can tell in each direction whether there is an obstacle or not, up to a distance of two cells away. (Thus, the input is orientation-dependent.) It has also the information of the distance and the bearing to the goal. There is a total of 20 binary inputs (and thus more than 10^6 possible inputs/states). The separate inputs for gating or region specifications consist of x-y coordinates. An agent can move by selecting two output parameters: turn (left, right, or no turn) and speed (0 or 1). 400 steps are allowed for each episode. If agents fail to reach the goal within the limit, failure is declared. Reinforcement is provided (1) when the target is reached, the value of which is 1 in this case, (2) when the time runs out, the value of which is -1, and (3) when any move is made, the value of which is determined as follows: when the agent is going toward the target, the reinforcement is $gr = 1/c * ((x_2 - x_1)/x)^4$, where $c = 5.0$, $x_2 - x_1$ is the distance traveled in the target direction in one step, and x is set to 40. When the agent is going away from the target, the reinforcement is $gr' = -0.5gr$.

6.2 Algorithms and Parameters

We tested the following multi-agent RL algorithms: we tested on-line partitioning (gating), with either the prediction or the classification interpretation, with the weighted-average-of-local-errors combination, the exponentiated weighted-average-of-local-errors combination, or the overall-error (thus, there is a total of 6 algorithms, denoted as PWA, PEWA, PO, CWA, CEWA, CO, where P indicates *prediction*, C indicates *classification*, and WA, EWA or O indicate respective error combination methods). We also tested the batch version of gating (denoted as BA), in which we used the prediction interpretation with the weighted-average-of-local-errors measure (which resembled off-line algorithms; section 5.2). We tested off-line partitioning, including region-splitting (denoted as RS) and GA. We also tested a variation of region-splitting with multiple agents for each region combined with weighted averaging, trained with the weighted-average-of-local-errors measure (denoted as MRS). For comparison purposes, we also tested simple averaging and weighted averaging (without partitioning) of multiple agents (denoted as SA and WA respectively), based on the prediction interpretation and the weighted-average-of-local-errors measure. In constructing agents for these two types of models, random variations of the initial internal structures of individual agents (including initial weights, numbers of hidden units, and learning rates) were used to generate uncorrelated agents (see Appendix; Raviv and Intrator 1997, Breimen 1996a). We also tested single-agent Q-learning (denoted as Q).

The parameters for different algorithms were set as follows: The learning rate for each individual agent was set at $\alpha_0 = 0.05$ initially and gradually lowered by $\alpha_t := \alpha_0 * \epsilon^t$, where $\epsilon = 0.996$. and t denotes episode numbers. 7 hidden units were used for the backpropagation network in each agent. The discount factor γ was set at 0.95. The initial temperature τ for the Boltzmann distribution action selection was set at 0.04 and the temperature changed according to $\tau_t = \tau_0 * \epsilon^t$, where $\epsilon = 0.9999$. Specifically, in gating algorithms (i.e., on-line partitioning, including PWA, PEWA, PO, CWA, CEWA, CO), we used a linear network for gating (as in Jordan and Jacobs 1994), and we used the *x*-*y* coordinates as input to the gating network while local orientation dependent views (described earlier) were given to agents. In the classification-based versions, the error criterion ν was set at 0.05. The number of agents was set at 5 (but we also tried other numbers). We set the learning rate of the gating network to be higher than the individual agent networks initially and reduced it toward 0 more quickly; that is, $\alpha_0 = 0.07$ and $\alpha_t := \alpha_0 * \epsilon^t$, where $\epsilon = 0.985$. In BA, we updated gating weights every 20 episodes (while updating agents immediately; section 5.2). In RS (region splitting), $threshold_1$ was set at -100, and $threshold_2$ at 5. The maximum number of regions (R) was set at 20, and the number of repetition n was set at 1. We trained agents on a partition for 20 episodes before the partition was changed. In MRS, we set the number of agents in each region (K) at 5 (but we also tried other numbers), and the other parameters were the same as used in RS (specified before). In GA, the population size of different partitions was set at 10, l_1 at 20, and l_2 at 10. We trained agents on each partition in a generation for 20 episodes (but we also tried other numbers). The maximum number of regions in each partition (R) was set at 20, and the number of repetition n was set at 1. The thresholds in the mutation operation were set the same way as in RS. In SA and WA, 5 agents were used.

6.3 Results

Let us compare the different algorithms discussed earlier by their test performance in each maze. The test performance was measured by the average success rates over 100 "test" episodes, conducted after sufficient training of each algorithm (that is, when their averaged learning curves leveled off), using their respectively best parameter settings within the range of parameter settings we tested. Roughly, on average, the on-line partitioning algorithms (PWA, PEWA, PO, CWA, CEWA, CO) took a total of 1000 training episodes, the weighting/averaging algorithms (SA and WA) took 1000 episodes, the single-agent algorithm (Q) took 1000 episodes, the off-line partitioning algorithm RS (and MRS) also took 1000 episodes (due to the use of non-overlapping regions in RS, there is no need for additional training episodes), but GA took 2000 episodes (due to having 10 partitions in each generation, with each partition trained for 20 episodes, and training for 10 generations). Note also that the on-line algorithms and the weighting/averaging algorithms require far more updatings than the other algorithms, because of their use of overlapping or identical regions and thus the simultaneous updating of multiple (5) agents at each step. Figure 12 shows the performance of all the algorithms after training in each of the six different mazes (with all the combinations of difficulty levels and sizes). Overall, we can see that some partitioning methods improved performance compared with single agent systems. Compared with all the other multi-agent algorithms, RS fared the best: it was better than or comparable to all these other algorithms, and it was better than others in more difficult settings (i.e., the hard mazes in large sizes), which demonstrated the merit of this algorithm.

Comparing on-line and off-line partitioning. As shown in Figure 12, the on-line (gating) algorithms performed worse than RS and GA (the off-line algorithms). The performance differences were statistically significant. This is because the on-line (gating) algorithms performed both agent learning and partitioning at the same time (both with gradient descent) and thus complicated the overall process. On the other hand, the off-line algorithms were able to separate the two aspects of learning and thus facilitated the overall learning process as discussed earlier.

Comparing RS and GA. GA in general conducts a more thorough search than RS, and thus incurs a higher cost, but we would expect it to achieve a better performance. However, judging from the experimental data, performance-wise there was little difference between the two. This was probably because of the randomness introduced by the crossover operation. It appeared that in this particular setup there was not much advantage in introducing such randomness.

Comparing RS and MRS. The two algorithms performed comparably (although MRS was slightly better, the difference was not statistically significant). Similarly, WA and SA did not outperform Q.

Comparing gating algorithms. As mentioned earlier, the classification interpretation leads to inaccurate Q-values, and thus we expect worse performance from it compared with the prediction interpretation. This conjecture was borne out by the data: PWA performed better than CWA, PEWA performed better than CEWA, and PO performed better than CO. The differences were statistically significant. The exponentiated versions performed at an equal or better level compared with the non-exponentiated versions (by comparing PEWA with PWA and PO, and comparing CEWA with CWA and CO). The batch version performed no better than the corresponding non-batch version (that is, BA performed comparably to PWA). Overall, the gating algorithms were not better than single agent Q-learning, simple averaging, or weighted averaging.

Note that in the above experiments, the x-y coordinates provided to the multi-agent algorithms for the purposes of partitioning were also provided to single-agent Q-learning as part of its input, so as to avoid putting the single-agent algorithm at a disadvantage. In so doing, we compared the performance of the single-agent algorithm with vs. without the x-y coordinates, and found no significant performance difference.

Examining RS. Let us look into RS specifically. Figure 13 shows the learning curve (in terms of success rate for each block of 20 episodes). Figure 14 shows the consistency curve (in terms of the consistency measure used in RS). Both demonstrate a gradual improvement of a multi-agent system using RS over the course of learning. Figure 15 shows a partition of regions

algorithm \ task	1xEasy	2xEasy	3xEasy	1xHard	2xHard	3xHard
RS	100.0(0.0)	94.6(5.1)	93.4(6.5)	89.0(3.5)	57.6(8.3)	42.8(6.5)
MRS	100.0(0.0)	98.2(2.4)	96.0(3.4)	91.2(8.2)	64.4(8.7)	48.8(5.0)
GA	97.2(5.6)	97.2(2.4)	96.8(2.3)	86.6(8.6)	49.4(38.5)	21.8(13.2)
PWA	100.0(0.0)	94.2 (3.3)	95.2(6.8)	77.8(9.5)	35.4(19.8)	6.6(7.0)
PEWA	100.0(0.0)	99.6 (0.5)	93.4(2.1)	79.2(6.6)	37.2(10.2)	6.0(8.9)
PO	100.0(0.0)	99.4 (0.8)	98.2(1.2)	70.2(13.3)	19.4(10.1)	12.0(5.2)
CWA	99.6(0.8)	50.4(37.9)	38.6(36.4)	27.8(32.5)	0.6(0.4)	0.4(0.8)
CEWA	100.0(0.0)	94.4 (2.3)	92.2(3.3)	63.2(7.0)	9.0(9.0)	9.5(4.5)
CO	100.0(0.0)	75.8(38.8)	66.8(37.8)	38.6(14.2)	3.6(4.0)	0.4(0.5)
BA	100.0(0.0)	97.6(1.4)	97.6(1.9)	76.2(5.7)	20.2(17.5)	8.8(10.9)
SA	100.0(0.0)	99.4(0.8)	97.3(0.9)	56.6(16.3)	22.0(6.2)	10.6(1.5)
WA	100.0(0.0)	99.8(0.4)	92.6(5.5)	64.4(14.2)	18.6(3.1)	12.6(4.2)
Q	100.0(0.0)	95.0(6.6)	93.4(6.5)	73.8(12.5)	34.6(14.3)	15.6(5.5)

Figure 12: Comparisons of different algorithms in terms of average test performance (average numbers of successful episodes out of 100 test episodes). The standard deviations are in parentheses. See text for explanation.

as the result of RS, and also some trajectories through these regions of the maze to reach the goal (at the end of learning).

As conjectured earlier, using multiple agents may reduce the requirement regarding the complexity of individual agents. That is, when multiple agents are used, we may be able to learn the same task equally (or more) effectively using simpler individual agents. Because we used backpropagation networks, the complexity of the agents was determined by the number of hidden units in their networks. Our data (see Figure 20) shows that when we gradually reduced the number of hidden units in Q (a single-agent algorithm) and RS (a multi-agent algorithm) respectively, Q performed worse and worse, but the performance of RS was hardly affected.

Recall that the point of using different agents for different regions was to be able to specialize each agent to a different region, in order to exploit differential characteristics of regions and to develop differential characteristics in the corresponding agents (and thus to reduce the complexity of individual agents). Figures 16, 17, 18, and 19 show the average Q-values for each agent in four typical settings. Comparing these different agents (each for a different region), we found that the average Q-values of different agents were different, which implied that different action policies were formed (comparing Figures 16, 17, 18, and 19). Each agent was indeed specialized to its corresponding region because its Q-values were specifically concerned with actions in that specific region.

7 Discussions

Let us retrace the development of various ideas in this paper. The discussion of gating [18] led to the adoption of on-line partitioning for RL (with all its variations in terms of error measures; Figure 3). Off-line algorithms for RL (Figure 5 as well as Figures 6 and 7) were then formulated as alternatives to the on-line algorithms that enable the decomposition of learning (of agents) and

[18]The analysis of bagging led to the analysis of weighted averaging (an extension of simple averaging), which in turn led to partitioning in general and gating in particular.

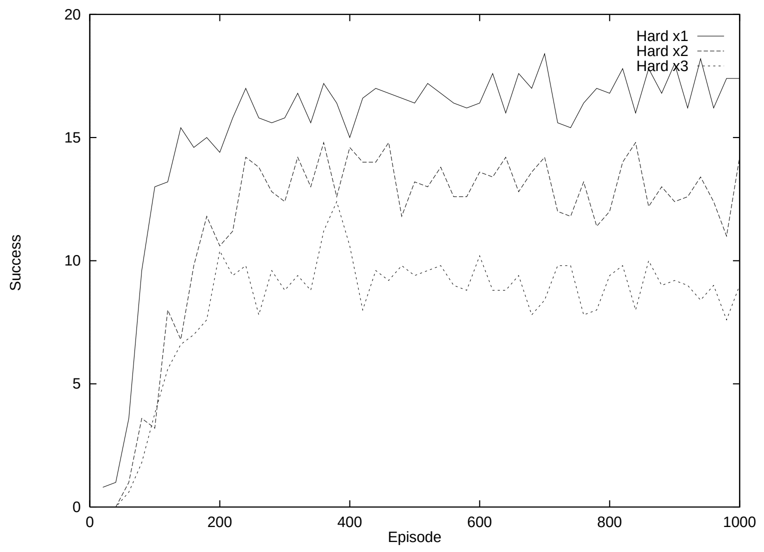


Figure 13: The learning curve of RS (in terms of success rates) in three hard mazes.

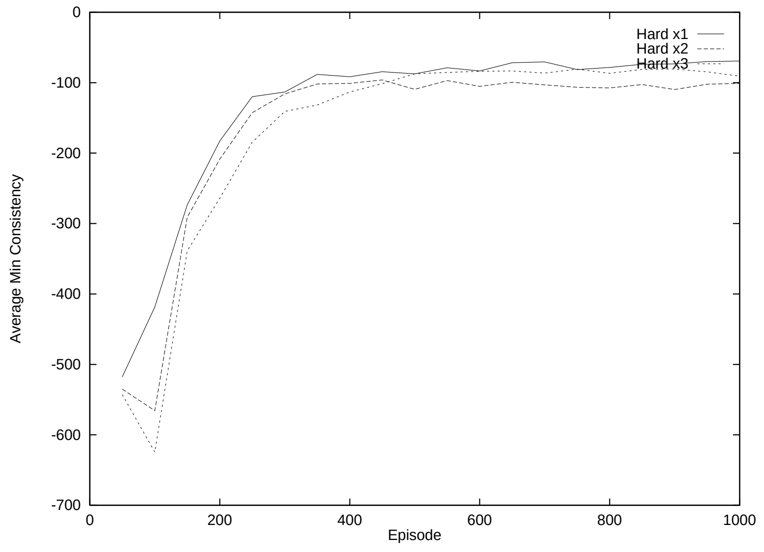
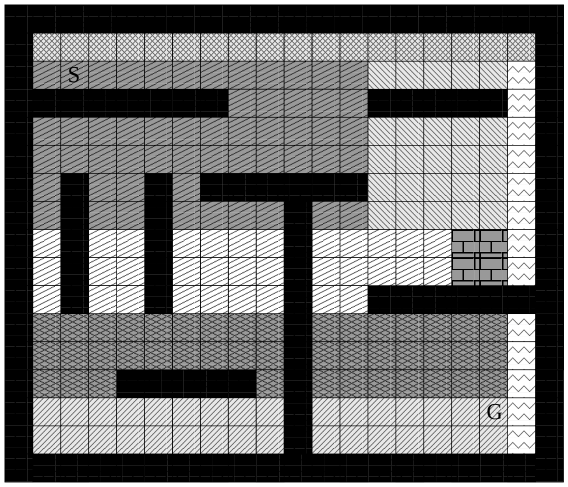


Figure 14: The consistency measure over the course of learning in three hard mazes.



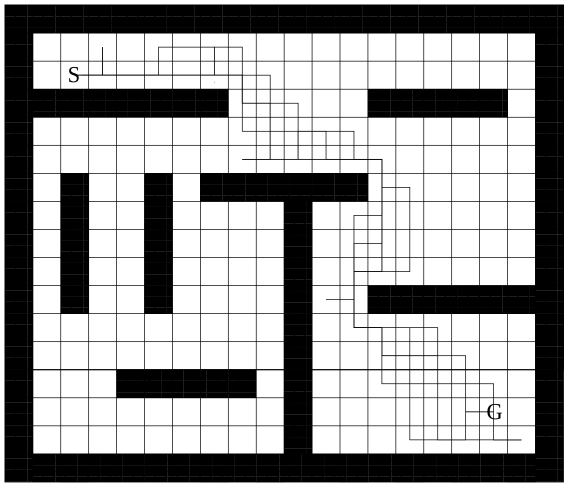


Figure 15: A partition of regions from RS and ten sample successful trajectories through these regions (as tested after training).

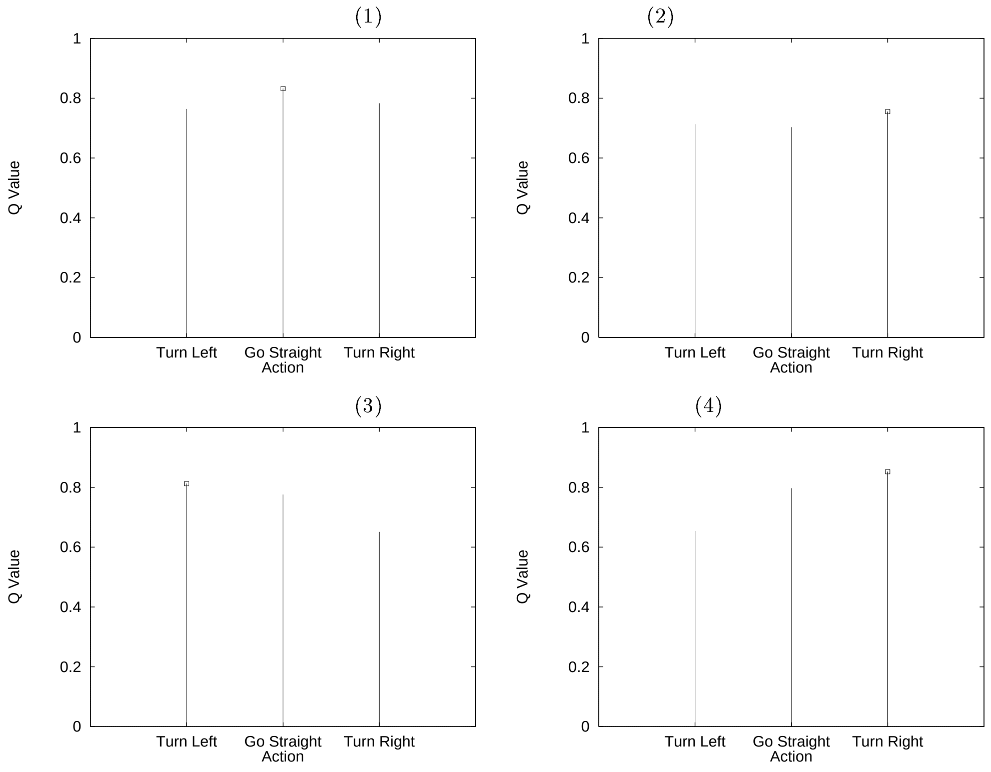


Figure 16: The averaged Q-values of an agent (agent 1) in each of the four major categories of states: (1) no wall in sight, (2) wall in front, (3) wall on right, and (4) wall on left. For the purpose of comparing different agents, the Q-values (for "turn left", "go straight", and "turn right" respectively) of all the input states that fit into one of the above four categories were averaged for each agent.

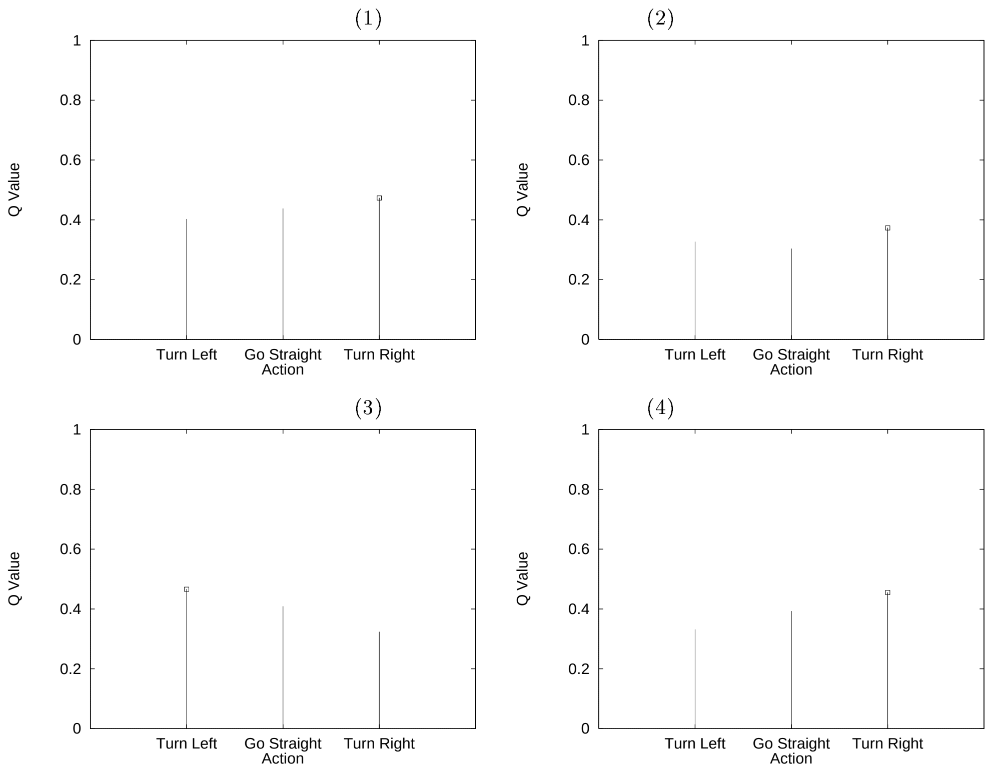


Figure 17: The averaged Q-values of an agent (agent 2) in each of the four major categories of states: (1) no wall in sight, (2) wall in front, (3) wall on right, and (4) wall on left. For the purpose of comparing different agents, the Q-values (for "turn left", "go straight", and "turn right" respectively) of all the input states that fit into one of the above four categories were averaged for each agent.

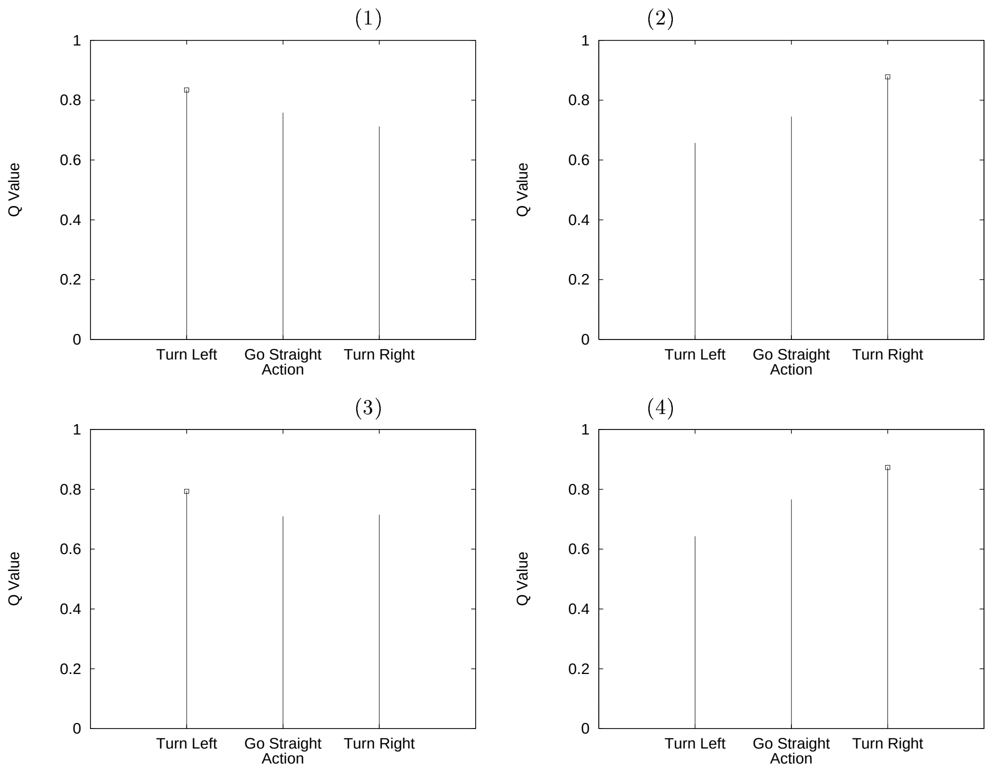


Figure 18: The averaged Q-values of an agent (agent 3) in each of the four major categories of states: (1) no wall in sight, (2) wall in front, (3) wall on right, and (4) wall on left. For the purpose of comparing different agents, the Q-values (for "turn left", "go straight", and "turn right" respectively) of all the input states that fit into one of the above four categories were averaged for each agent.

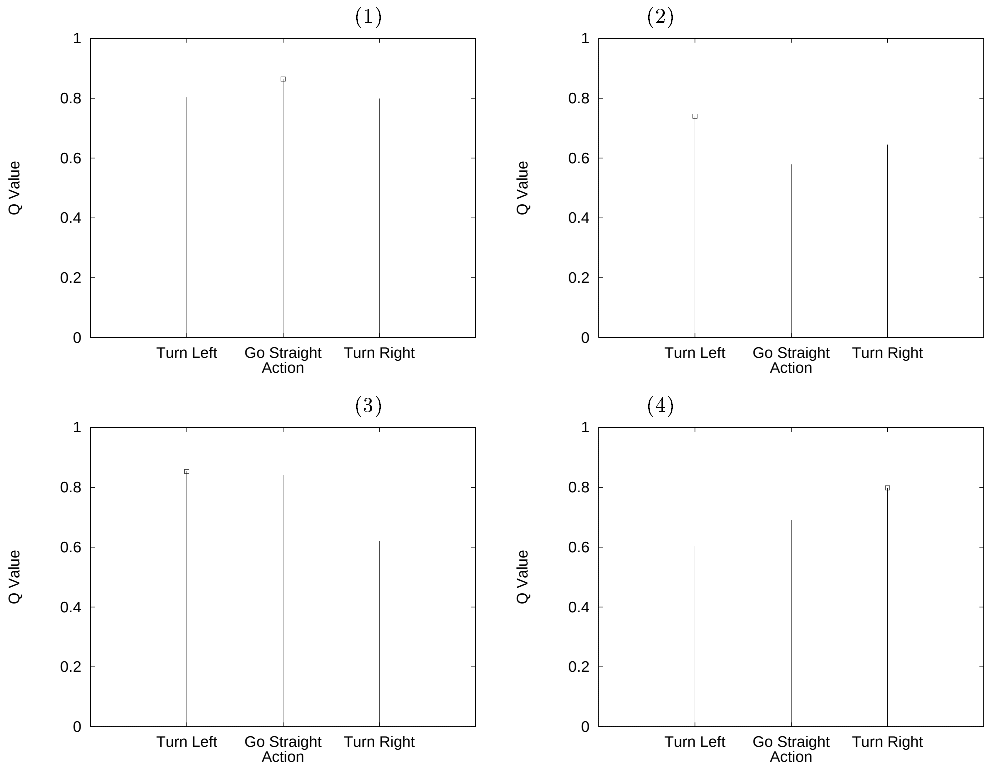


Figure 19: The averaged Q-values of an agent (agent 4) in each of the four major categories of states: (1) no wall in sight, (2) wall in front, (3) wall on right, and (4) wall on left. For the purpose of comparing different agents, the Q-values (for "turn left", "go straight", and "turn right" respectively) of all the input states that fit into one of the above four categories were averaged for each agent.

algorithm/hidden units	1xHard	2xHard	3xHard
Q /15	85.2 (11.8)	23.8 (3.3)	19.2 (3.3)
Q /7	73.8 (12.5)	34.6 (14.3)	15.6 (5.5)
Q /3	61.8 (12.4)	26.0 (2.8)	13.8 (7.4)
Q /2	41.0 (18.0)	12.9 (9.5)	2.4 (1.9)
RS /15	86.4 (5.5)	61.0 (15.4)	42.0 (9.6)
RS /7	89.0 (3.5)	57.6 (8.3)	42.8 (6.5)
RS /3	83.2 (5.7)	73.4 (6.7)	49.6 (10.9)
RS /2	79.8 (10.3)	55.8 (15.7)	52.4 (16.0)

Figure 20: The effect of the number of hidden units in backpropagation networks when single-agent and multi-agent algorithms were used. The success rates are shown here. The standard deviations are in parentheses.

partitioning, drawing inspirations also from other methods discussed. Specifically, the discussion of decision trees and CART (and similar ideas by Sanger 1993, Blanzieri and Katenkamp 1996) led to the formulation of the region-splitting algorithm. The region-splitting algorithm with weighted averaging (Figure 9) was an extension of it. The GA-based algorithm was another extension of it (and also a straight application of GA).

We can sum up the optimization issues in different settings as follows:

- On-line optimization of soft partitioning (such as gating discussed in section 5.1): we optimize the two sets of parameters, those concerning combination weights w_k's and internal agent weights w_{net}^k's, together. The partitioning is the direct result of w_k's. With an algorithm such as gradient descent, we are guaranteed to reach local optima. If w_{net}^k's are guaranteed to reach global optima (with proper internal structures and learning algorithms), then the learning of w_k's (using e.g. gradient descent) can reach global optima (if we use the linear combination of agents as in section 5.1 and thus there is no problem of local optima with regard to w_k's).

- Off-line optimization of soft partitioning: we optimize the two sets of parameters separately, w_k's and w_{net}^k's. If the learning of w_{net}^k's is guaranteed to reach global optima, then the overall learning can reach global optima (if we use the linear combination of agents). Otherwise, with gradient descent, local optima can be reached. Compared with the previous method, the cost will be much higher due to the fact that for each adjustment of w_k's, we have to train w_{net}^k's to convergence. We can, of course, interleave combination weight learning and agent training by having a certain amount of agent training without necessarily training agents to convergence before each combination weight change. In that case, the method is essentially a batch version of the on-line method just discussed.

- Off-line optimization of hard partitioning (such as region-splitting and GA as discussed in section 5.2): in this case, we optimize different parameters separately: w_k's, w_{net}^k's, and P (the partitioning into regions). Iteratively, we adjust P, and then w_{net}^k's and w_k's (only if multiple agents are used in each region as in MRS) on the basis of the current P. The optimality can be ensured in this case only in very limited cases (with respect to region-splitting as discussed in section 5.2). To avoid high cost that would incur in order to train the system to convergence for each partitioning change, we interleave partitioning and training of agents (including the training of w_k's when used).

- On-line optimization of hard partitioning: in this case, we optimize three sets of parameters together simultaneously: w_k's, w_{net}^k's, and P (the partitioning). When gradient descent is used, local optima can be reached. In order to perform on-line optimization of P, we need to select a type of partitioning so that on-line learning methods can be applied. In particular, when gradient descent is used, we need to create a continuous, differentiable partitioning function, and thus we need to use soft partitioning (as discussed before), or a probabilistic function (Singh et al 1994; which resembles soft partitioning except with weights being interpreted as probabilities instead of as soft boundaries). Due to this difficulty, on-line optimization of hard partitioning was not adopted in this work.

8 Comparisons

With regard to averaging or weighted averaging, in addition to various theoretical analyses mentioned earlier (such as Breiman 1996 a, b, Raviv and Intrator 1996, Uedo and Nakano 1996), empirically, there have been demonstrations of performance advantages resulting from combining a set of (diversified) learners, for example, Hashem (1993), Perrone (1993), Parmanto, Munro and Doyle (1996), Rosen (1996), Tumer and Ghosh (1996), and Taniguchi and Tresp (1997). There are also many empirical demonstrations of bagging and boosting in particular (such as Ting and Witten 1997, Quinlan 1996, Drucker 1997, Margineantu and Dietterich 1997, etc). However, the afore-mentioned work did not deal with reinforcement learning.

With regard to partitioning, there are other variations besides what we discussed earlier: for example, variance-based weighting (that is, setting a gating weight at each input point to be the inverse of the variance of the corresponding agent at that point), error-based weighting (that is, setting a gating weight to be the inverse of the residual error of the corresponding agent at each input point), or density-based weighting (that is, using the conditional probability estimate $P(agent_i|x)$ as the gating weight for agent i at input point x), and their various combinations thereof (e.g., as discussed by Tresp and Taniguchi 1995). Also very relevant, especially to our off-line partitioning methods, is the work by Chrisman (1993), McCallum (1996), Blanzieri and Katenkamp (1996), and Sanger (1991). In these approaches, different criteria for splitting were adopted, for example, based on differences in Q-value distribution with regard to different actions (McCallum 1996, Chrisman 1993), based on the variance of error (Sanger 1991), or based on the amount of error (Blanzieri and and Katenkamp 1996), or based on error consistency (as used in our methods; i.e., the ratio or difference between the sum of absolute errors vs. the sum of errors; Blanzieri and Katenkamp 1996). van der Smagt and Groen (1995) formed tree-like structures for multi-resolution hierarchies through splitting (when error exceeds a preset threshold) and merging, after training with self-organizing-map (SOM) algorithms. Rosca (1997) devised a evolutionary divide-and-conquer method that was independently developed but similar to our GA-based method. [19]

Comparing with most of the other weighting and partitioning work (which rarely dealt with RL; such as Breiman 1996b, Wolpert 1992, van der Smagt and Greon 1995, Jacobs et al 1991, Jordan and Jacobs 1994, Blanzieri and Katenkamp 1996, Sanger 1991), our work extends into reinforcement learning tasks, which are more complex because of the lack of any clear learning target. In fact, in RL, we only have moving targets that are changing constantly during learning. However, some of these models, such as Chrisman (1993) and McCallum (1996), were specifically designed for reinforcement learning, especially Q-learning. Comparing with the existing work involving RL, our approach has some differences and/or relative advantages. Our approach does not require a priori partitioning of the input/state space such as done in Singh (1994), and Humphrys (1996). [20] Our approach does not require a priori division of a task into subtasks as in e.g. Dietterich (1997) and Tadeppali and Dietterich (1997), which is one way of simplifying learning, but it requires some a priori decisions that determine preset subgoals or predetermined subsequences and is very different from our approach of learning to partition the input/state

[19] However, his method relied on complex fitness functions, and used clustering for grouping together different agents. It is not only more costly computationally, but it also tends to produce irregular regions that are composed of disjoint parts.

[20] For example, Humphrys (1996) used pre-wired, differential input features and reward functions for different agents. Singh (1994) used separate training on different subtasks for different agents.

space. Our approach does not even require knowledge to initialize partitioning, as in the case of "knowledge-based" RBF networks in Taniguchi and Tresp (1997) and Kubat (1997). Our approach is not limited to selecting an agent for an entire (sub)sequence as in Tham (1995) and Dayan and Hinton (1993), so different agents may alternate in dealing with a sequence. In terms of assigning agents to regions, our approach [21] appears to be better justified algorithmically than more ad hoc methods such as Humphrys (1996) and Dorigo and Gambardella (1995), which involve purely heuristic methods for competition to determine a winner agent for a region. Our approach also differs from feature selection approaches such as McCallum (1996) and Chrisman (1993) (which use decision trees to select input features in order to create useful states on which reinforcement learning is based), because such work does not divide up the input/state space into regions for *different* agents to learn (and thus makes learning tasks easier overall), although they do divide up input/state space through using decision trees. Our approach differs from radial-basis functions (such as in Blanzieri and Katenkamp 1996, Schaal and Atkeson 1996, Peterson and Sun 1998, and van der Smagt and Greon 1995), in that (1) we use hypercubes or other region forms different from the spherical form used by RBF and (2) more importantly, instead of a Gaussian function as in the RBF approach, we use a more powerful approximator in each region, which is capable of arbitrary functional mappings and thus eliminates the need for highly overlapping regions (especially when hard partitioning is used). [22] Our approach also differs from CMAC (Albus 1975, Lane et al 1992, Sutton 1996), in that we use a more powerful approximator in each region, thus avoiding highly overlapping placement of regions again. The same point applies also to fuzzy logic based methods (see e.g. Takagi and Sugeno 1985).

Our approach is somewhat more suitable for incremental learning that involves changes over time, unlike some of the existing work that predetermines partitioning and thus makes it hard or impossible to undergo changes. It is especially suitable for learning situations in which the world changes in a minor way throughout the course of learning. The changes can be quickly accommodated due to the use of localized regions which make each individual mapping (as a part of the overall mapping) to be learned by each approximator (i.e., neural network) simpler and thus make the overall learning easier. Localized regions also tend to group together inputs/states that have similar value distributions (with regard to actions) and thus are easier to adjust. When major change occurs that cannot be localized, our approach can also accommodate them by creating corresponding drastic changes in the allocations of local agents. However, such changes are costly.

9 Concluding Remarks

This work is concerned with weighting and partitioning in reinforcement learning tasks. We developed various multi-agent approaches for the purpose of facilitating reinforcement learning tasks, through partitioning a input/state space into different regions and/or weighting multiple agents differently. In this work, various multi-agent learning approaches were viewed as (implicit)

[21] In our on-line methods, agents are assigned to regions by nonconstant weighting with weights learned on-line with respect to different points in the input space, and in our off-line methods, by incrementally creating agents whenever the splitting of a region occurs,

[22] Note that each individual radial basis function is not capable of arbitrary mappings, and thus overlapping placement of such functions throughout the input/state space is necessary in order to approximate arbitrary functions.

optimization of the partitioning of the input/state space to achieve better learning performance. Such optimization can be done either on-line or off-line. Partitioning can be with done either hard or soft boundaries. However, the goal is always the same: to exploit differential characteristics of regions and differential characteristics of agents to reduce the learning complexity of agents (and their function approximators) and thus to facilitate learning overall. We experimentally tested various approaches discussed in the paper in a reinforcement learning setting, using several tasks differing in state space size and difficulty level. To summarize our findings from the experiments, off-line algorithms, especially region-splitting, performed the best; on-line algorithms had varied performance; some good multi-agent methods (especially region-splitting) indeed facilitated learning and reduced the requisite complexity of individual agents.

Appendix: Diversity in Weighting

The precept of choosing a diverse set of agents (i.e., uncorrelated agents) as opposed to a set of identical or highly similar agents in the averaging or weighted averaging schemes has been justified on the basis of bias-variance decomposition (see e.g. Breiman (1996c), Ueda and Nakano (1996), Raviv and Intrator (1996), and so on). That is,

$$variance(avg_i a_i) = avg_x(y(x) - avg_i a_i(x))^2$$

$$= avg_x(y(x) - avg_x y(x))^2 + (avg_x avg_i a_i(x) - avg_x y(x))^2 + avg_x(avg_i a_i(x) - avg_x avg_i a_i(x))^2$$

So, the total error is determined mainly by the bias $avg_x avg_i a_i(x) - avg_x y(x)$ and variance $avg_x(avg_i a_i(x) - avg_x avg_i a_i(x))^2$ of the averaged outcome of the agents. While averaging may not reduce the bias of individual agents, it may reduce the variance of individual agents and thereby improve the performance of the aggregated system. The variance can be decomposed as follows (Raviv and Intrator 1996):

$$avg_x(avg_i a_i(x) - avg_x avg_i a_i(x))^2$$

$$= avg_x(avg_i a_i(x))^2 - (avg_x avg_i a_i(x))^2$$

$$= 1/n^2 * \sum_i (avg_x a_i^2(x) - (avg_x a_i(x))^2) + 2/n^2 * \sum_{i<j} (avg_x(a_i(x) a_j(x)) - avg_x a_i(x) * avg_x a_j(x))$$

where n is the number of agents. If a_i are independent and identically distributed, then

$$variance(avg_i a_i) = 1/n^2 * \sum_i (avg_x a_i^2(x) - (avg_x a_i(x))^2) = 1/n * variance(a_i)$$

where $variance(a_i) = avg_x(a_i(x) - avg_x a_i(x))^2 = avg_x a_i^2(x) - (avg_x a_i(x))^2$ and $variance(a_i) = variance(a_j)$ for all i and j. That is, the variance is reduced by a factor of n. On the other hand, if a_i's are identical, i.e., $a_i(x) = a(x)$, for all i and j, then

$$variance(avg_i a_i) = 1/n * variance(a) + \frac{2}{n^2} * \frac{n(n-1)}{2} variance(a) = variance(a)$$

where $variance(a) = avg_x(a(x) - avg_x a(x))^2 = avg_x a^2(x) - (avg_x a(x))^2$. That is, there is no reduction. Most averaging methods fall somewhere in between. The heuristics of creating independent agents has been embedded in a number of well-known approaches, such as “bagging”, in which diversity is achieved through repeated random re-sampling of the training data set and the use of "unstable" (easily varied) agents, and in "boosting", in which diversity is achieved through repeated re-sampling with changing sampling probabilities in favor of those data points that are misclassified (or mispredicted, Drucker 1997). This idea is also relevant to gating (Jacobs et al 1991). Jacobs (1997) aimed at achieving not only uncorrelated agents but anti-correlated (i.e., negatively correlated) agents.

References

J. Albus, (1975). A new approach to manipulator control: the cerebellar model articulation control. *Journal of Dynamic Systems Measure and Control*, 97, 270-277.

C. Atkeson, A. Moore, and S. Schaal, (1997). Locally weighted regression. *Artificial Intelligence Review.*

R. Bellman, (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

D. Bertsekas and J. Tsitsiklis, (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA.

E. Blanzieri and P. Katenkamp, (1996). Learning radial basis function networks on-line. *Proc. of International Conference on Machine Learning.* 37-45. Morgan Kaufmann, San Francisco, CA.

J. Boyan and A. Moore, (1995). Generalization in reinforcement learning: safely approximating the value function. in: J. Tesauro, and D. Touretzky, and T. Leen, (eds.) *Neural Information Processing Systems 7*, 369-376, MIT Press, Cambridge, MA.

L. Breiman, L. Friedman, and P. Stone, (1984). *Classification and Regression.* Wadsworth, Belmont, CA.

L. Breiman, (1996a). Bagging predictors. *Machine Learning*, 24, 123-140.

L. Breiman, (1996b). Stacked regressions. *Machine Learning*, 24, 49-64.

L. Breiman, (1996c). Bias, variance and arcing classifiers. Technical Report 460. University of California, Berkeley.

L. Chrisman, (1993). Reinforcement learning with perceptual aliasing: the perceptual distinction approach. *Proc. of AAAI.* 183-188. Morgan Kaufmann, San Francisco, CA.

P. Dayan and G. Hinton, (1993). Feudal reinforcement learning. *Neural Information Processing Systems*, MIT Press, Cambridge, MA.

T. Dietterich, (1997). Hierarchical reinforcement learning with MAXQ value function decomposition. ftp://www.cs.orst.edu

M. Dorigo and L. Gambardella, (1995). Ant-Q: a reinforcement learning approach to combinatorial optimization. Technical Report 95-01. Universite Libre de Bruxelles. Belgium.

H. Drucker, (1997). Improving regressors using boosting techniques. 107-115. *Proc. of ICML'97.* Morgan Kaufmann, San Francisco, CA.

M. Erickson and J. Kruschke, (1996). Rules and Examplars in Category Learning. Manuscript.

Y. Freund and R. Schapire, (1996). Experiments with a new boosting algorithm. 148-156. *Proc. of ICML'97.* Morgan Kaufmann, San Francisco, CA.

S. Hashem, (1993). *Optimal Linear Combinations of Neural Networks.* Ph.D. Thesis, Purdue University. Purdue, Indiana.

M. Humphrys, (1996). W-learning: a simple RL-based society of mind. Technical report 362, University of Cambridge, Computer Laboratory. Cambridge, UK.

J. Hertz, A. Krogh, and R. Palmer, (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley, Reading, MA.

R. Jacobs, (1997). Bias/variance analysis of mixtures-of-experts architectures. *Neural Computation.* 9, 369-383.

R. Jacobs, M. Jordan, S. Nowlan, and G. Hinton, (1991). Adaptive mixtures of local experts. *Neural Computation.* 3, 79-87.

M. Jordan and R. Jacobs, (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural Computation.* 6, 181-214.

L. Kaelbling, M. Littman, and A. Moore, (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4, 237-285.

A. Krogh and J. Vedelsby, (1995). Neural network ensembles, cross validation, and active learning. *Neural Information Processing Systems*, 231-238. MIT Press, Cambridge, MA.

M. Kubat, (1997). Decision trees can initialize radial-basis-function networks. Manuscript.

S. Lane, D. Handelman, and J. Gelfand, (1992). Theory and development of higher-order CMAC neural networks. *IEEE Control Systems*, pp.23-31. April, 1992.

L. Lin, (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning.* Vol.8, pp.293-321.

D. Margineantu and T. Dietterich, (1997). Pruning adaptive boosting. *Proc. of ICML*, 211-218. Morgan Kaufmann, San Francisco, CA.

M. Mataric, (1995). Reward functions for accelerated learning. *Proc. Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, MA.

A. McCallum, (1996). Learning to use selective attention and short-term memory in sequencetial tasks. *Proc. Conference on Simulation of Adaptive Behavior.* 315-324. MIT Press, Cambridge, MA.

R. Meir, (1995). Bias, variance and the combination of least sqaures estimators. *Neural Information Processing Systems*, 295-302. MIT Press, Cambridge, MA.

B. Parmanto, P. Munro and H. Doyle, (1996). Reducing variance of committee prediction with resampling techniques. *Connection Science*, 8 (3/4), 405-426.

M. Perrone, (1993). *Improving Regression Estimation: Averaging Methods for Variance Reduction with Extensions to General Convex Measure Optimization.* Ph.D. Thesis, Brown University. Providence, RI.

T. Peterson and R. Sun, (1998). An RBF network alternative to a hybrid architecture. *Proceedings of IEEE International Conference on Neural Networks*, Anchorage, Alaska. May 4-9, 1998. IEEE Press, Piscataway, NJ.

T. Poggio and F. Girosi, (1990). Networks for approximation and learning. *Proceedings of IEEE*, 78 (9), 1481-1497.

R. Quinlan, (1986). Inductive learning of decision trees. *Machine Learning*. 1, 81-106.

R. Quinlan, (1996). Bagging, Boosting and C4.5. *Proc. of AAAI'96*. 725-730. Morgan Kaufmann, San Francisco, CA.

Y. Raviv and N. Intrator, (1996). Bootstrapping with noise: an effective regularization technique. *Connection Science*, 8 (3/4), 355-372.

C. Reddy and P. Tadepalli, (1997). Learning goal-decomposition rules using exercises. *Proc of ICML'97*. 278-286. Morgan Kaufmann, San Francisco, CA.

J. Rosca, (1997). *Hierarchical Learning with Procedural Abstraction Mechanisms*. Ph.D Thesis, Department of Computer Science, University of Rochester, Rochester, NY.

B. Rosen, (1996). Ensemble learning using decorrelated neural networks. *Connection Science*, 8 (3/4), 373-384.

T. Sanger, (1991). Tree-structured adaptive networks for function approximation in high-dimensional spaces. *IEEE Transaction on Neural Networks*, 2 (2), 285-293.

S. Schaal and C. Atkeson, (1996). From isolation to cooperation: an alternative view of a system of experts. *Advances in Neural Information Processing Systems 8*. pp.605-611. MIT Press. Cambridge, MA.

R. Schapire, Y. Freund, P. Bartlett, and W. Lee, (1997). Boosting the margin: a new explanation for the effectiveness of voting methods. *Proc.of International Conference on Machine Learning*. 322-330. Morgan Kaufmann, San Francisco.

S. Singh, (1994). *Learning to Solve Markovian Decision Processes*. Ph.D Thesis, University of Massachusettes, Amherst, MA.

S. Singh, T. Jaakkola, and M. Jordan, (1994). Reinforcement learning with soft state aggregation. In: S.J. Hanson J. Cowan and C. L. Giles, eds. *Advances in Neural Information Processing Systems 7*. Morgan Kaufmann, San Mateo, CA.

R. Sun, (1997). Planning from reinforcement learning. Technical report TR-CS-97-0027, University of Alabama, Tuscaloosa, AL.

R. Sun and L. Bookman, (eds.) (1994). *Computational Architectures Integrating Neural and Symbolic Processes* . Kluwer Academic Publishers. Norwell, MA.

R. Sun and T. Peterson, (1997). A hybrid agent architecture for reactive sequential decision making. In: R. Sun and F. Alexandre, (eds.) *Connectionist-Symbolic Integration*. Lawrence Erlbaum Associates. Hillsdale, NJ.

R. Sun and T. Peterson, (1997). A hybrid model for learning sequential navigation. *Proc. of IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA'97).* Monterey, CA. pp.234-239. IEEE Press. Piscateway, NJ.

R. Sun and T. Peterson, (1998). Some experiments with a hybrid model for learning sequential decision making. *Information Sciences*. 111, 83-107. October, 1998.

R. Sun, T. Peterson, and E. Merrill, (1996). Bottom-up skill learning in reactive sequential decision tasks. *Proc.of 18th Cognitive Science Society Conference*, Lawrence Erlbaum Associates, Hillsdale, NJ. pp.684-690. 1996.

R. Sutton, (1988). Learning to predict by the methods of temporal difference. *Machine Learning.* Vol.3, 9-44.

R. Sutton, (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proc.of Seventh International Conference on Machine Learning.* Morgan Kaufmann, San Meteo, CA.

R. Sutton, (1996). Generalization in reinforcement learning: successful examples using sparse coarse coding. *Neural Information Processing Systems 8*, MIT Press, Cambridge, MA.

P. Tadepalli and T. Dietterich, (1997). Hierarchical explanation-based reinforcement learning. *Proc. International Conference on Machine Learning.* 358-366. Morgan Kaufmann, San Francisco.

T. Takagi and M. Sugeno, (1985). Fuzzy identification of systems and its applications to modeling and control. *IEEE Transactions on Systems, Man and Cybernetics.* 15, 1. 116-132.

M. Taniguchi and V. Tresp, (1997). Averaging regularized estimators. *Neural Computation*, 9, 1163-1178.

C. Tham, (1995). Reinforcement learning of multiple tasks using a hierarchical CMAC architecture. *Robotics and Autonomous Systems.* 15, 247-274.

S. Thrun and A. Schwartz, (1995). Finding structure in reinforcement learning. *Neural Information Processing Systems 7*, MIT Press, Cambridge, MA.

W. K. Ting and I. Witten, (1997). Stacking bagged and dagged models. 367-375. *Proc. of ICML'97.* Morgan Kaufmann, San Francisco, CA.

K. Tumer and J. Ghosh, (1996). Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8 (3/4), 385-404.

V. Tresp and M. Taniguchi, (1995). Combining estimators using non-constant weighting functions. *Neural Information Processing Systems 7*, 419-426. MIT Press, Cambridge, MA.

N. Ueda and R. Nakano, (1996). Generalization error of ensemble estimators. *IEEE International Conference on Neural Networks*, pp.90-95. IEEE Press. Piscateway, NJ.

P. van der Smagt and F. Groen, (1995). Approximation with neural networks. *Proc. of 1995 International Conference on Neural Networks*, Perth, Australia. IEEE Press, Piscateway, NJ.

C. Watkins, (1989). *Learning with Delayed Rewards.* Ph.D Thesis, Cambridge University, Cambridge, UK.

M. Wiering and J. Schmidhuber, (1996). HQ-learning. TR IDSIA-95-96.

S. Whitehead, (1993). A complexity analysis of cooperative mechanisms in reinforcement learning. *Proc. AAAI'93*, 607-613. Morgan Kaufmann, San Francisco, CA.

D. Wolpert, (1992). Stacked generalization. *Neural Networks*, 5, 241-259.

L. Xu, M. Jordan and G. Hinton, (1995). An alternative model for mixtures of experts. *Neural Information Processing Systems 7*, 633-640. MIT Press, Cambridge, MA.