Chapter 1

A COMPARATIVE STUDY OF DISCRETIZATION APPROACHES FOR STATE SPACE GENERALIZATION IN THE KEEPAWAY SOCCER TASK

Javier Garca[*] ***Ivn Lpez-Bueno***[†] ***Fernando Fernández***[‡] ***Daniel Borrajo***[§]
Computer Science Department, Universidad Carlos III de Madrid
Avenida de la Universidad 30, 28911 Leganés, Madrid, Spain

Keywords: Reinforcement Learning, Temporal difference learning, Evolutionary computation, State space generalization, Discretization, Keepaway soccer

[*]E-mail address: fjgpolo@inf.uc3m.es
[†]E-mail address: ivan.lopezbueno@gmail.com
[‡]E-mail address: ffernand@inf.uc3m.es
[§]E-mail address: dborrajo@ia.uc3m.es

Abstract

There are two main branches of reinforcement learning: methods that search directly in the space of value functions that asses the utility of the behaviors (*Temporal Difference Methods*); and methods that search directly in the space of behaviors (*Policy Search Methods*). When applying Temporal Difference (TD) methods in domains with very large or continuous state spaces, the experience obtained by the learning agent in the interaction with the environment must be generalized. The generalization can be carried out in two different ways. On the one hand by discretizing the environment to use a tabular representation of the value functions (e.g. Vector Quantization Q-Learning algorithm). On the other hand, by using an approximation of the value functions based on a supervised learning method (e.g. CMAC Q-Learning algorithm). Other algorithms use both approaches to benefit from both mechanisms, allowing a higher performance. This is the case of the Two Step Reinforcement Learning algorithm. In the case of Policy Search Methods, the Evolutionary Reinforcement Learning algorithm has shown promising in RL tasks. All these algorithms present different ways to tackle the problem of large or continuous state spaces. In this chapter, we organize and discuss different generalization techniques to solve this problem. Finally, we demonstrate the usefulness of the different algorithms described to improve the learning process in the Keepaway domain.

1. Introduction

Many model-free Reinforcement Learning (RL) techniques rely on learning value functions. Ideally, we could assume that the estimated values of the value function could be represented as a look-up table with one entry for each state or state-action pair. However, complex (most real world) domains represent a challenge to the use of such tables, because they usually have continuous (or very large) state/action spaces. This fact poses two problems: the size of the state-action tables (presenting unrealistic memory requirements); and the correct use of the experience (an agent is not able to visit all states or state-action pairs, or the time needed to fill the look-up table easily becomes too large). This problem is known as the curse of dimensionality and requires some form of generalization. Generalization techniques have been extensively studied before the popularity of RL [52], so many of the existing generalization methods are commonly combined with RL. Some approaches have used decision trees [11], neural networks [30], or variable resolution dynamic programming [38]. Another alternative consists on discretizing the continuous variables. Then, the new discrete version of the problem is solved with RL techniques. However, if we choose a bad discretization of the state space, we might introduce hidden states into the problem, making it impossible to learn the optimal policy. If we discretize too fine grain, we loose the ability to generalize and increase the amount of training data that we need. This is especially important when the task state is multi-dimensional, where the number of discrete states can be exponential in the state dimension.

Thus, it seems reasonable to replace the discrete look-up tables of many RL algorithms with function approximators or by discretizing the environment to use a reduced tabular representation of the value functions, capable of handling continuous variables in several dimensions and generalizing across similar states.

All these traditional techniques require a policy representation through state enumeration. Another group of techniques perform a direct policy search. These methods are often more applicable to problems with large or infinite state spaces, because they do not need a policy representation through state enumeration. Policy search methods represent a policy through a set of parameters, θ. This set of parameters can be expressed in different ways (e.g. weights in a neural network), which generally increase linearly in both the number of dimensions and the size of these dimensions. For this reason, policy search methods are always more amenable to problems whose state space is large or infinite.

The goal of this chapter is to introduce the reader to the generalization problem in complex RL domains and to organize and discuss different generalization techniques to solve this problem. We will review and compare Vector Quantization (VQ) [14], CMAC [3], other generalizations techniques such as decision trees [15] and regression trees [32], and a policy search method [20]. All these techniques, coupled with RL methods, will be applied in the Keepaway Soccer Task [50].

The remainder of the chapter is organized as follows. Section 2 begins with a brief summary of the generalization problem, introducing some of the main approaches used. Section 3 describes how vector quantization is used to solve the generalization problem in the Vector Quantization Q-Learning (VQQL) algorithm. The CMAC Q-Learning algorithm is shown in Section 4. Section 5, where the Two Step Reinforcement Learning algorithm (2SRL) is reviewed, shows two approaches based, one on supervised learning function

approximation (ISQL), and the other on state space discretizations. Section 6 overviews the Evolutionary RL method and Section 7 describes the keepaway task and presents empirical results on the benchmark version of this task for the different algorithms described. Section 8 concludes.

2. Related Work

In [23], the authors characterize two main branches of RL: methods that search directly in the space of value functions that measure the utility of the behaviors (*Temporal Difference Methods*); and methods that search directly in the space of behaviors (*Policy Search Methods*). The authors focuses entirely on the first set of methods. Temporal difference (TD) methods are one popular way to solve RL problems, that learn a value function that estimates the expected reward for taking an action in a particular state. Policy search methods can also address RL problems by searching in the space of behaviors for one that receives the maximal reward [53].

When applying TD methods in domains with very large or continuous state spaces, the experience generated by the learning agent during its interaction with the environment must be generalized. The generalization methods are often based on the approximation of the value functions used to calculate the action policy and tackled in two different forms [46]. On the one hand, by discretizing the state space to use a tabular representation of the value functions. On the other hand, by using an approximation of the value functions based on a supervised learning method.

The first one consists on discretizing the state space to obtain a compacted and discrete one, so tabular representations of the value functions can be used. Uniform discretization has reached good results [41], but only in domains with few features describing the states, where a high resolution does not increase the number of states very much. A very large state space produces unpractical computational requirements, and unpractical amounts of experience in model based methods [44].

Continuous U Trees has also been used for discretizing the state space [58]. Thus, the Continuous U Tree transfers traditional regression trees techniques to RL techniques. In [28], the authors try to improve the applicability and efficacy of RL algorithms by adaptive state space partitioning. They proposed the TD learning with adaptive vector quantization algorithm (TD-AVQ) wich is an online method and does not assume any priori knowledge with respect to the learning task and environment. The paper [5] introduces new techniques for abstracting the state space of a Markov Decision Process (MDP). These techniques extend one of the minimization models, known as $\epsilon - reduction$, to build a partition space. The state space built has a smaller number of states than the original MDP. As a result, the learning policies on the state space built should be faster than on the original state space. The discretization approach is used in Section 3., where the VQQL algorithm is described.

The second approach for learning the Q function is based on the supervised learning of tuples $< s, a, q_{s,a} >$, where s is a state, a is an action, and $q_{s,a}$ is the $\hat{Q}$ value approximated for the state s and the action a. Any supervised learning method can be used, each of them with a set of parameters θ to be computed [9; 57; 7]. Many different methods of function approximation have been used successfully, including CMACs, radial basis functions, and

neural networks [52]. This approach is studied in Section 4. where a CMAC is used as function approximation, and in Section 5., where we define the Iterative Smooth Q-Learning (ISQL) algorithm, based on the Smooth Value Iteration algorithm [22], which will be tested using different function approximators.

Policy search methods, are a reasonable alternative to TD methods. The general idea behind these methods is to search for the optimal policy in the space of all possible policies (behaviors) by directly examining different policy parameterizations, bypassing the assignment of the value. In [29], the authors discuss direct search methods for unconstrained optimization. They give a modern perspective on this classical family of algorithms. In [40], they focus on the application of evolutionary algorithms to the RL problem, emphasizing alternative policy representations, credit assignment methods, and problem-specific genetic operators. Some strengths and weaknesses of the evolutionary approach to RL are discussed, along with a survey of representative applications. In [45], the authors present a method for structuring a robot motor learning task. By designing a suitably parameterized policy, they show that a simple search algorithm, along with biologically motivated constraints, offers a competitive means for motor skill acquisition. Surveys of policy search methods can be found in [1; 43]. This approach is used in Section 6., where the evolutionary RL algorithm is described.

3. Vector Quantization and RL

Vector Quantization (VQ) is a clustering method that permits to find a more compact representation of the state space. VQ appeared as an appropiate way of reducing the number of bits needed to represent and transmit information [17]. This technique is extensively employed for signal analog-to-digital conversion and compression, which have common characteristics to MDP problems. VQ is based on the principle of block coding. In the past, the design of a vector quantizer was considered to be a challenging problem due to the need for multi-dimensional integration. In [31], the authors proposed a VQ design algorithm based on a training sequence. A VQ that is designed using this algorithm is cited in the literature as LBG-VQ (which refers to the initials of the authors: Linde, Buzo and Gray). In the case of large state spaces in RL, the problem is analogous: *how can we compactly represent a huge number of states with very few information?*. As an example, RL and VQ are used in [59] for image compression. In that work, they present the FRLVQ algorithm (Fuzzy Reinforcement Learning Vector Quantization) which is based on the combination of fuzzy K-means clustering and topology knowledge. In each iteration of the RL algorithm, the size and direction of the movement of a codevector is decided by the overall pair-wise competition between the attraction of each training vector and the repellent force of the corresponding winning codevector. In [14], VQ is applied again to the RL problem. The authors present the *VQQL* model, that integrates Q-Learning as the RL technique, and VQ as state generalization technique. They use the Generalized Lloyd Algorithm, a numerical clustering method, for the design of vector quantizers. Figure 1 shows how to represent the action policy following this approach.

Next, we review how to apply VQ to RL. First, we review the main concepts of VQ and, then, the VQQL algorithm is described in detail.

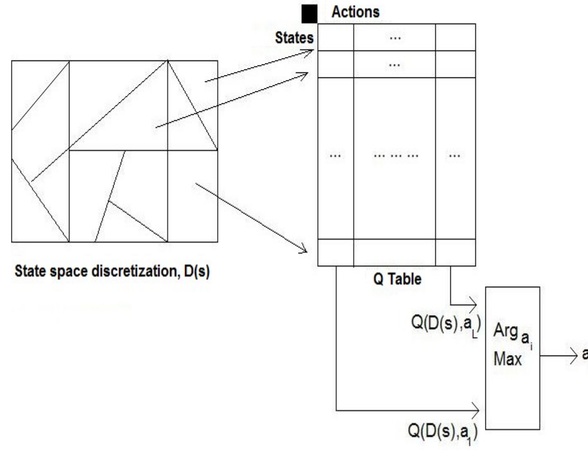


Figure 1. Generalization by discretizing the environment, and using a tabular representation of the value functions.

3.1. Vector Quantization

A vector quantizer *Q* of dimension *K* and size *N* is a mapping from a vector (state or action) in the K-dimensional Euclidean space, R^k, into a finite set *C* containing *N* states. Thus,

$$Q : R^k \rightarrow C$$

where $C = \{y_1, y_2, ..., y_N\}$, $y_i \in R^k$. Given *C*, and a state $x \in R^k$, $VQ(x)$ assigns x to the closest state from C,

$$VQ(x) = \arg\min_{y \in C}\{dist(x, y)\}$$

where $dist(x, y)$ is a distance function in the R^k space (typically the euclidean distance). To design the vector quantizer we use the Generalized Lloyd Algorithm (GLA), also called k*-means*. GLA is a clustering technique that consists of a number of iterations, each one recomputing the set of more appropriate partitions of the input states and their centroids. The centroids, together with the distance metric, define the Voronoi regions. The VQQL algorithm uses each of those regions as a unique state, obtaining a state space discretization, that permits learning with reduced experience. An example of 2-dimensional VQ is shown in Figure 2. There are 16 partitions and a point associated with each partition. In this figure, every pair of values falling in a particular partition are approximated by a grey point associated with that partition.

3.2. VQQL

The integration of VQ and Q-learning yields the VQQL (Vector Quantization Q-Learning) algorithm, that is shown in Table 1. The vector quantizer is designed from the input data

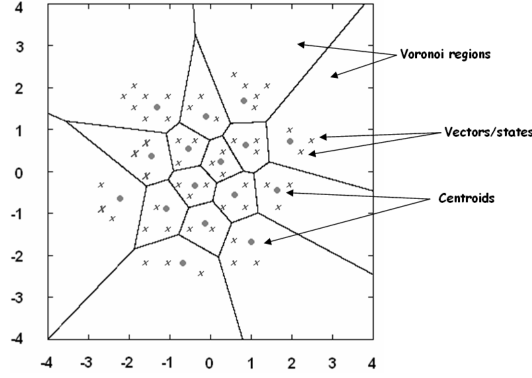


Figure 2. Example of Vector Quantizer.

C obtained during an interaction between the agent and the environment. The data set C is composed of tuples in the form $< s_1, a, s_2, r >$ where s_1 and s_2 belong to the state space S, a belongs to the action space A and r is the immediate reward. In many problems, S is composed of a large number of features. In these cases, we suggest to apply feature selection to reduce the number of features in the state space. Feature selection is a technique for selecting a subset of relevant features for building a new subset. So feature selection is used to select the relevant features of S to obtain a subset $S^{'}$. This feature selection process is defined as $\Gamma : S \rightarrow S'$. The set of states $s^{'} \in S^{'}$, $C^{'}_{s}$, are used as input for the Generalized Lloyd Algorithm to obtain the vector quantizer. The vector quantizer $VQ^{s^{'}}$ is a mapping from a vector $s^{'} \in S^{'}$ into a vector $s^{'} \in D_{s^{'}}$, where $D_{s^{'}}$ is the state space discretization $Ds' = s^{'}_1, s^{'}_2, ..., s^{'}_n$ for $s^{'}_i \in S'$.

In the last part of the algorithm, the Q-table is learned from the obtained discretizations using the set $C^{'}$ of experience tuples by equation 1.

$$Q(s_t, a_t) \rightarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (1)$$

To obtain the set $C^{'}$ from C, each tuple in C is mapped to the new representation. Therefore, every state in C is firstly projected to the space $S^{'}$ and then discretized, i.e. $VQ^{S^{'}}(\Gamma(S))$.

4. CMAC and RL

In RL, though, generalization is commonly achieved through function approximation (as an instance of supervised learning): by learning weights from a set of features over the agent past and present perceptions. In this case, CMAC is often used as the approach for function approximation. CMAC (*Cerebellar Model Arithmetic Computer*) was originally designed for robotic systems and today is still widely used [3]. CMAC architecture was motived by the biological motor control functions of human cerebellum. According to [10], a CMAC is most closely comparable to a neural network that is trained using back-propagation, but almost always outperforms the neural network. So, CMAC can be adapted to function as a data analysis tools beyond its original purpose as robot controller. In [21], the authors

Table 1. VQQL Algorithm.

VQQL
1. Gather experience tuples
1.1. Generate the set C of experience tuples of the type $< s_1, a, s_2, r >$ from an interaction of the agent in the environment, where $s_1, s_2 \in S$, $a \in A$ and $r \in \Re$ is the immediate reward.
2. Reduce (optionally) the dimension of the state space
2.1. Let C_s be the set of states in C
2.2. Apply a feature selection approach using C_s to reduce the number of features in the state space. The resulting feature selection process is defined as a projection $\Gamma : S \rightarrow S'$
2.3. Set $C_s^{'} = \Gamma(C_s)$
3. Discretize the state space
3.1. Use GLA to obtain a state space discretization, $D_{s'} = s_1^{'}, s_2^{'}, ..., s_n^{'}, s_i^{'} \in S^{'}$, from $C_s^{'}$.
3.2. Let $VQ^{S^{'}} : S^{'} \rightarrow D_{s'}$ be the function that, given any state in $S^{'}$, returns the discretized value in D_s.
4. Learn the Q-Table
4.1. Map the set C of experience tuples to a set $C^{'}$. For each tuple $< s_1, a, s_2, r >$ in C, introduce in $C^{'}$ the tuple $< VQ^{S^{'}}(\Gamma(s_1)), a, VQ^{S^{'}}(\Gamma(s_2)), r >$
4.2. Apply the Q-Learning update function defined in equation 1 to learn a Q table Q: $D_{s'} \times a \rightarrow \Re$, using the set of experience tuples $C^{'}$
6. Return Q, Γ and $VQ^{S^{'}}$

adapted the CMAC algorithm for use in representing a general multi-variable function and applied the algorithm to mapping a geological surface. The CMAC discretization of the variable space is quite similar to that used in the Averaged Shifted Histogram (ASH) proposed by [47]. The results presented in [50] indicate that using a CMAC to approximate the value function led to results that were better than hand-coded approaches.

Thus, CMAC has been widely used for applications in function approximation [3; 2; 4; 13]; in robotics for path planning for manipulators [54; 55]; industrial processes [56]; and character recognition [35]. CMAC is widely used in robotic control and it has been used in various adaptative control tasks [34; 37; 27], and it has received extensive attention and extensive mathematical analysis [36]

CMAC is well known as a good function approximator and it local generalization ability has been beneficial in RL. In this section, CMAC together with Q-Learning is reviewed. First, we describe CMAC as function approximation, and later the CMAC-QL algorithm is described in detail.

4.1. CMAC

Consider a problem in which the state set is continuous and 2-dimensional. In this case, a state is a vector with two components. One kind of feature are those corresponding to circles in state space. If a state is in a circle, then the corresponding feature has the value 1, otherwise the feature has the value 0 [52]. Figure 3 shows an example where the binary feature vector corresponding to the state $S^{'} = (S^{'}_1, S^{'}_2)$ is $\phi(S^{'}) = \{0, 1, 0, 0, 1, 0\}$.

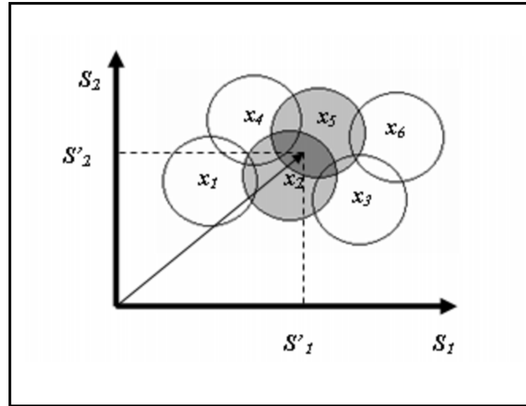


Figure 3. *Tile coding*.

For each state, a vector of binary features is built. This vector represents a coarsely code for the true location of the state in the space. Representing a state with features that overlap in this way is known as coarse coding.

CMAC, also known as tile coding, is a form of coarse coding [50; 52]. In CMAC the features are grouped into partitions of the input state space. Each of such partition is called a *tiling* and each element of a partition is called a *tile*. Each *tile* is a binary feature. The tilings are overlaids, each offset from the others. In each tiling, the state is in one tile. In Figure 4, we present an example of tile-coding which represents a 2-dimensional input state space *S* where two tilings were overlaid, each one with an offset from the other of 1/2 of the tile width. The set of all these active tiles, one per tiling and two per state, is what makes up the binary feature vector. As an example, the binary features vector for the state (0.38, 0.38) is $\phi_s = \{0_0, 0_1, ..., 1_5, 0_6, ..., 1_{26}, 0_{27}, ..., 0_{31}\}$.

4.2. CMAC-QL

When we combine CMAC with Q-Learning, it results in the CMAC-QL algorithm. In CMAC-QL, the approximate value function, Q_a, is represented not as a table, but in a parameterized form with a parameter vector $\vec{\theta}_t$. This means that the approximate value function Q_a totally depends on $\vec{\theta}_t$. In CMAC, each tile has an associated weight. The set of all these weights is what makes up the vector $\vec{\theta}$. The approximate value function, $Q_a(s)$ is then computed using equation 2.

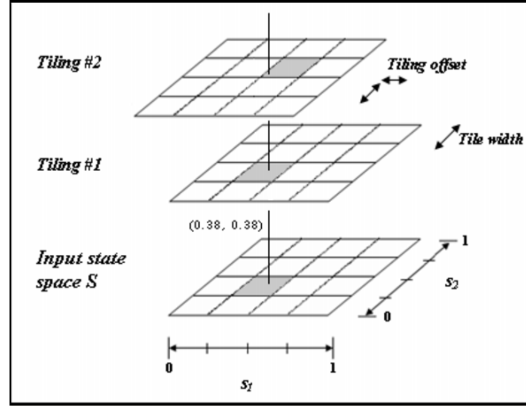


Figure 4. *Tile coding*.

$$Q_a(s) = \vec{\theta}^T \vec{\phi} = \sum_{i=0}^{n} \theta(i)\phi(i) \quad (2)$$

The CMAC process to compute the Q function is shown in Figure 5.

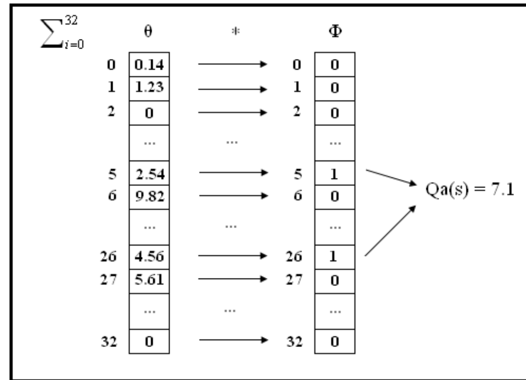


Figure 5. CMAC process to compute the Q value for an action *a*.

The Figure 5 shows a vector $\vec{\theta}_t$ of size 32 an a vector of features ϕ_s of the same size. The Q function is computed by the scalar product of these vectors. The CMAC-QL algorithm, described in Table 2, uses CMAC to generalize the state space. In this case, a data set C is obtained during an interaction between the agent and the environment. This data set C is composed of tuples in the same form as in VQQL, $< s_1, a, s_2, r >$. Again, s can be also composed of a large number of features and feature selection can be used to select a subset $S^{'}$ of the relevant features of S. Later, the CMAC is built from $C^{'}_s$. For each state variable $x^{'}_i$ in $s^{'} \in S^{'}$ the tile width and tiles per tiling are selected taking into account their ranges.

In this chapter, a separate value function for each of the discrete actions is used. Finally, the Q function is approximated using equation 2.

Table 2. CMAC-QL Algorithm.

CMAC-QL
1. Gather experience tuples
1.1. Generate the set C of experience tuples of the type $< s_1, a, s_2, r >$ from an interaction of the agent in the environment, where $s_1, s_2 \in S$, $a \in A$ and $r \in \Re$ is the immediate reward.
2. Reduce the dimension of the state space
2.1. Let C_s be the set of states in C
2.2. Apply a feature selection approach using C_s to reduce the number of features in the state space. The resulting feature selection process is defined as a projection $\Gamma : S \rightarrow S'$
2.3. Set $C_s^{'} = \Gamma(C_s)$
3. Design CMAC
3.1. Design a CMAC function approximator from $C_s^{'}$
4. Approximate the Q function
4.1. Map the set C of experience tuples to a set C'. For each tuple $< s_1, a, s_2, r > \in C$, introduce in C' the tuple $< \Phi(\Gamma(s1)), a, \Phi(\Gamma(s_2)), r >$ where Φ is the binary vector of features
4.2. Update the weights vector θ for the action a using $\Phi(\Gamma(s_1))$, $\Phi(\Gamma(s_2))$ and r.
4.3. Apply the approximate value function defined in equation 2 to approximate the Q function for the action a using θ and $\Phi(\Gamma(C_s))$.
6. Return Q, Γ and θ

5. Decision/Regresion Trees and RL

The value function can be approximated using any general function approximator such as neural network, or decision/regression trees. Two Steps Reinforcement Learning (2SRL), uses decision and regression trees as discretization methods of the state space [15]. It computes the action-value function in model free RL. The method assumes a reduced set of actions and finite trials, where positive and discrete rewards can be obtained only when a goal area is achieved. It is based on finding discretizations of the state space that are adapted to the value function being learned, trying to keep the convergence properties of the discretization methods using non-uniform discretizations [42]. The method is based on two learning phases. The first one is a model free version of the Smooth Value Iteration algorithm [22], that is called Iterative Smooth Q-Learning (ISQL). This algorithm, that executes an iterative supervised learning of the Q function, can be used to obtain a state space

discretization too. This new discretization is used in a second learning phase, that is called Multiple Discretization Q-Learning (MDQL), to obtain an improved policy.

However, the method presented a main drawback: it requires a discrete reward function. Using such discrete reward function permits ISQL to learn a function approximation of the Q function applying classification algorithms such as J48, an algorithm to learn decision trees [61]. However, many domains, like the Keepaway [50], have continuous reward functions. In this case, 2SRL needs to discretize the reward space by hand, and to test different discretizations to obtain an accurate one. To apply the same ideas of 2SRL in domains with continuous rewards requires that the function approximation used in the ISQL phase be a regression approach. As before, the approximation method should generate a discretization of the state space, so such discretization can be used in the second learning phase. Fortunately, there are different approaches in the literature for regression that generate state space discretizations of the input space. Classical ones are M5 [61] or PART [16], which generate regression trees and regression rules respectively.

In this section, first decision and regression trees are introduced as discretizacion techniques. Finally, the Two Step Reinforcement Learning algorithm is described.

5.1. Decision/Regression Trees

An approach for representing the value function in RL is to use a neural network. This approach scales better than others, but is not guaranteed to converge and often performs poorly even on relatively simple problems. Our alternative is to use a decision/regression tree to represent the value function. Also, decision and regression trees divide the state space with varying levels of resolution, as shown in Figure 6.

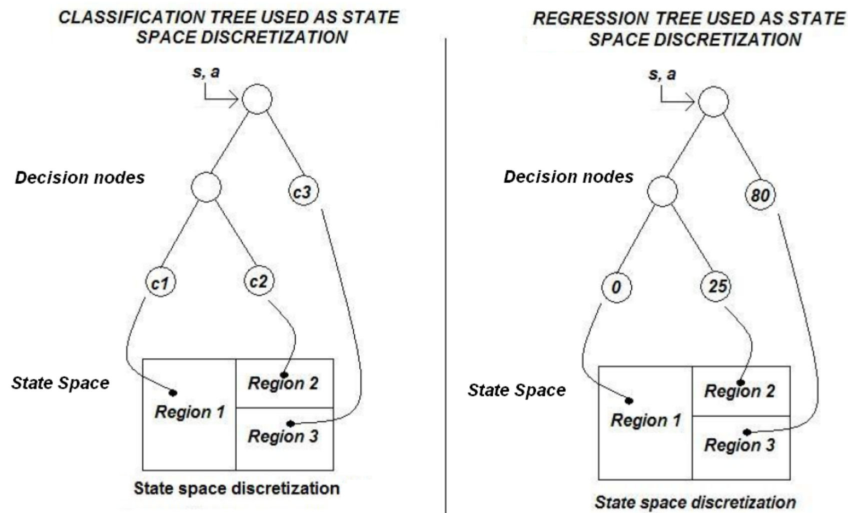


Figure 6. Decision and regression trees allow the state space to be divided.

In a decision tree, each decision node contains a test on the value of some input variable. The terminal nodes of the tree, the leafs, contain the predicted class values. Examples of algorithms to build decision trees are ID3, or J48 [61].

Regression trees may be considered as a variant of decision trees, designed to approximate real-valued functions instead of being used for classification tasks. A regression tree is built through a process known as binary recursive partitioning. This is an iterative process of splitting the data into partitions and then splitting it up further on each of the branches. The process continues recursively on each branch until an end condition is true, and the current node becomes a terminal node. Examples of algorithms to build regression trees are PART [16] and M5 [61].

5.2. Two Steps RL

Two Steps RL (2SRL) computes the action-value function in model free RL. This technique combines function approximation and discretization in two learning phases:

- ISQL phase: iteratively refines a $Q(s,a)$ function approximation. For each iteration, it obtains a state space discretization.
- MDQL phase: runs the Q-Learning algorithm using the state space discretization obtained from the approximation of the $Q(s,a)$ function.

Iterative Smooth Q-Learning algorithm (ISQL) derives from Discrete Value Iteration [8], where a function approximator is used instead of the tabular representation of the value function. Thus, the algorithm can be used in the continuous state-space case. The algorithm is described in Figure 7. The update equation of the Q function used is the stochastic Q-Learning update equation. The figure shows the adapted version of the original ISQL algorithm that allows continuous rewards.

The algorithm assumes a discrete set of L actions, and hence, it will generate L function approximators, $Q_{a_i}(s)$. It requires a collection of experience tuples, T. Different methods can be applied to perform this exploration phase, from random exploration to human driven exploration [48]. In each iteration, from the initial set of tuples, T, and using the approximators $\hat{Q}_{a_i}^{iter-1}(s)$, $i = 1, \ldots, L$, generated in the previous iteration, the Q-learning update rule can be used to obtain L training sets, $T_{a_i}^{iter}$, $i = 1, \ldots, L$, with entries of the kind $< s_j, c_j >$ where c_j is the resulting value of applying the Q update function to the training tuple j, whose state is s_j.

In the first iteration, $\hat{Q}_{a_i}^{0}(s)$ are initialized to 0, for $i = 1, \ldots, L$, and all $s \in S$. Thus, when the respective c_j are computed, they depend only on the possible values of the immediate reward, r. A requirement of the 2SRL approach is that the classification/regression techniques used in the ISQL phase to generate a $Q(s,a)$ function approximation must learn by dividing the space in regions, so that these regions can be used as a state space discretization. In the second phase, the obtained discretizations can be used to tune the action value function generated in the previous phase, following a multiple (one per action) discretization based approach.

Figure 8 shows how to translate the refined function approximation obtained in the first phase to state space discretization in the second phase of 2SRL. An approximation like M5-Rules has been used, and L rule sets have been obtained (one for each action).

The left part of the approximator rules *RuleSet i* will be used as the discretization $D_i(s)$, and the right part of the rules of *RuleSet i* will be located in column i of the generated Q

Iterative Smooth Q-Learning with Regression

- Inputs:
  1. A state space X
  2. A discrete set of L actions, $A = \{a_1, \ldots, a_L\}$
  3. A collection T of N experience tuples of the kind $< s, a_i, s', r >$, where $s \in X$ is a state where action a_i is executed, $s' \in X$ is the next state visited and r is the immediate reward received
- Generate L initial approximators of the action-value function $\hat{Q}^0_{a_i} : X \rightarrow \mathcal{R}$, and initialize them to return 0
- $iter = 1$
- Repeat
  - For all $a_i \in A$, initialize the learning sets $T^{iter}_{a_i} = \emptyset$
  - For $j = 1$ to N, using the jth tuple $< s_j, a_j, s'_j, r_j >$ do
    * $c_j = \alpha c_j + (1 - \alpha) \max_{a_r \in A} \gamma \hat{Q}^{iter-1}_{a_r}(s'_j)$
    * $T^{iter}_{a_j} = T^{iter}_{a_j} \cup \{< s_j, c_j >\}$
  - For each $a_i \in A$, train $\hat{Q}^{iter}_{a_i}$ to approximate the learning set $T^{iter}_{a_i}$
  - $iter = iter + 1$

  Until r_{max} is propagated to the whole domain
- Return $\hat{Q}^{iter-1}_{a_i}$, $\forall a_i \in A$

Figure 7. *Iterative Smooth Q-Learning* algorithm for domains with continuous rewards.

table, providing an initialization of the Q table in the second learning phase.[1] Each RuleSet may have a different number of rules, so the number of rows of the table is given by the maximum number of rules of the L approximators, and zero values can be used to complete the columns with less rules.

If we use J48 as the approximation technique, we obtain a classification tree where the regions of the state space discretization are each leaf of the tree. In Figure 9 we can see how to use a classification tree as space state discretization in the MDQL phase.

If we use M5 as an approximation technique, we obtain a regression tree where its leafs have a numerical value. We can use these numbers to initialize the Q table in the MDQL phase. Each leaf also represents a region of the state space discretization. In Figure 10 we can see how to use the numbers in the leafs of the tree to initialize the Q table.

Once the translation from the first scheme (ISQL) to the second one (MDQL) is done, a new learning phase can be executed using the Q-learning update function shown in equation 1.

The MDQL learning phase can be executed with new experiences or with the same ones used in the first phase. In the experimentation performed next, the first approach is applied using new experiences. This second phase can be executed exactly as it was defined in the

[1] We can also initialize to 0 and do not use the right part of the rules.

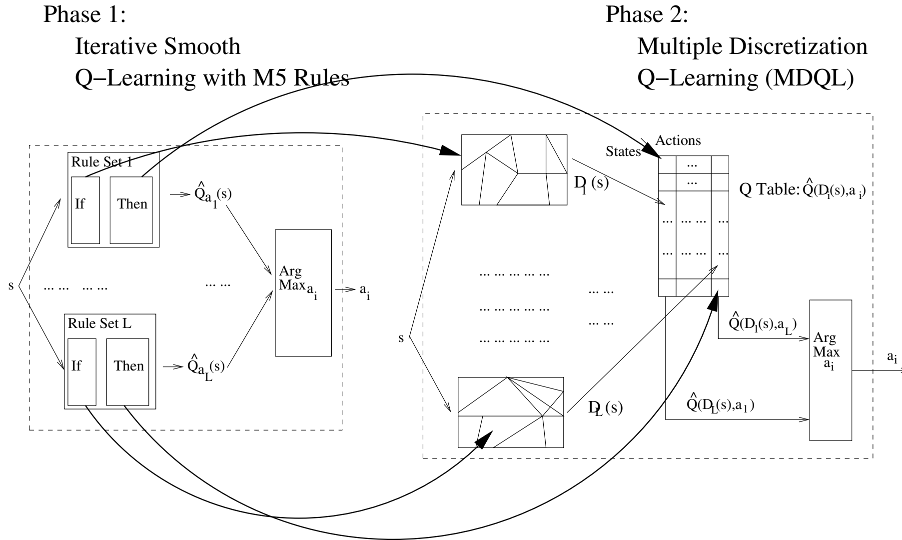
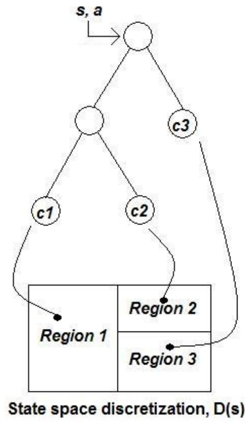


Figure 8. The two steps of the 2SRL algorithm with Regression.



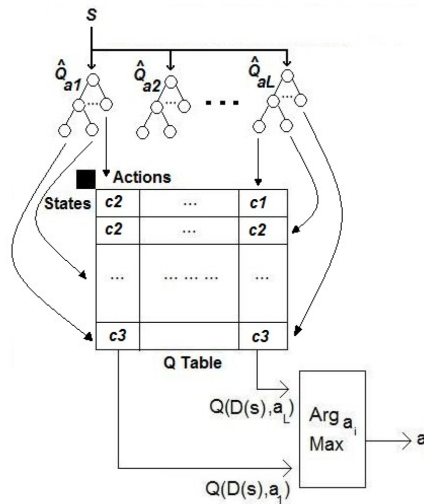


Figure 9. MDQL phase of 2SRL algorithm with classification trees.

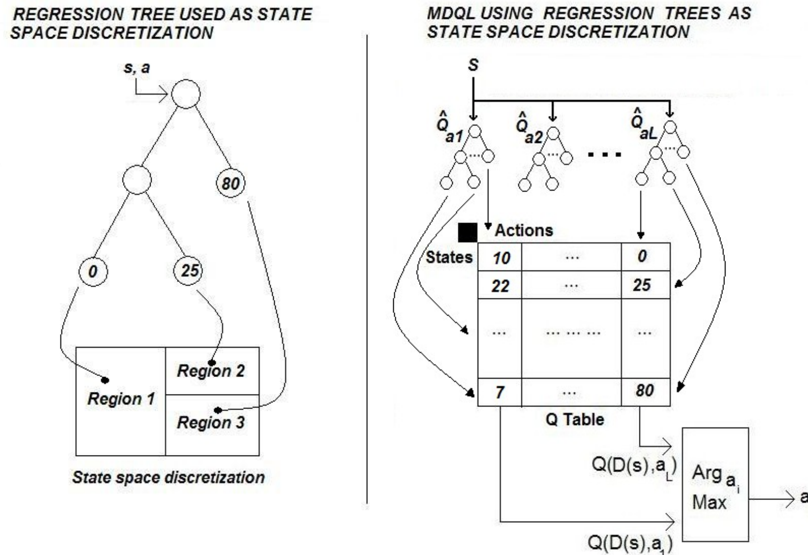


Figure 10. MDQL phase of 2SRL algorithm with regression trees.

original 2SRL algorithm, so additional explanation of this phase, and how to connect both phases, can be found in [15].

6. Evolutionary Computation and RL

Another category of RL methods are policy search methods. The main difference between policy search methods and other RL methods lies on how the search for a best policy is performed. Policy search methods search directly in the space of policies. The advantages of direct search methods compared to standard RL techniques are that they allow direct search in the policy space, whereas most RL techniques are restricted to optimize the policy indirectly by adapting state-action value functions or state-value functions, and they are usually easier to apply in complex problems and more robust.

In [6], the authors use a policy search method as a technique for finding sub-optimal controllers when such structured controllers exist. To validate the power of this approach, they show the presented learning control algorithm by flying an autonomous helicopter. In a similar way, [26] considers several neuroevolutionary approaches to discover robust controllers for a generalized version of the Helicopter hovering problem. In the same research line, [20] present a method to obtain a near optimal neuro-controller for the autonomous helicopter flight by means of an "ad hoc" evolutionary RL method. Evolutionary algorithms are powerful direct policy search methods. They has shown promising in RL tasks. In this section, first evolutionary algorithms are described from a direct policy search method perspective. Finally, the Evolutionary RL algorithm is explained in detail.

6.1. Evolutionary Algorithms as Direct Policy Search Method in RL

In Evolutionary Algorithms RL, the solutions take the form of policies used by decision making agents that operate in dynamic environments. Agents are placed in the world where they make decisions in response to environmental conditions. The Evolutionary Algorithm selects policies for reproduction based on their performance in the task, and applies genetic operators to generate new policies from the promising parent policies.

Evolutionary algorithms learn a decision policy of the form $f(s) \rightarrow a$ that maps a state description s to an action a. So, the fundamental difference between Evolutionary Algorithms and TD methods concerns the decision policy representation. TD methods define a value function of the form $f(s,a) \rightarrow v$ (Figure 11). Evolutionary Algorithm use a direct mapping that associates state descriptions directly with the recommended action.

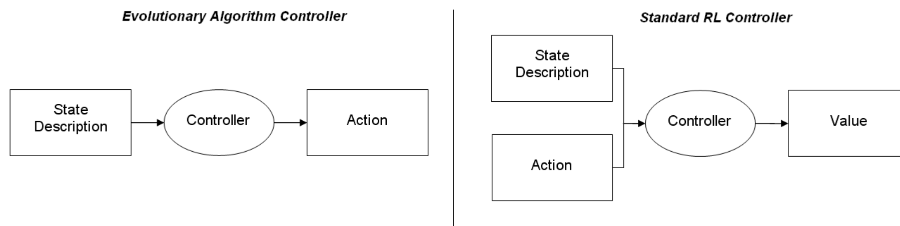


Figure 11. Mappings of Evolutionary Algorithms and TD methods.

6.2. Evolutionary RL

In this chapter, we apply an evolution strategy to the adaptation of the weights of artificial neural networks for RL tasks [20]. In this case, each artificial neural network is used for representing the policy of an agent. When using neural networks for representing the policy directly, the neural network weights parameterize the policies space that the network can learn. This parameterization is usually very complex, since there are strong correlations. The ability of an optimization algorithm to find dependencies between network parameters seems to be an important factor of the good performance.

In Evolutionary RL [20], each agent manages a population of N individual solutions (or candidate solutions). Each individual solution is a artificial neural network, all with the same topology but different weights. Each neural network receives as input the state perceived from the environment and the last action and returns as output the action to be executed. The algorithm evolves towards better solutions using four steps: selection of the best individual solutions, correlation, soft mutation and strong mutation. This evolution strategy adapts the weights of the neural networks in every generation. Figure 12 shows the different steps to evolve a population.

The weights of the initial population are radomly generated and adapted in every evolution. In every generation, each individual solution executes a number of τ episodes obtaining the cumulative reward (the fitness in the evolutionary algorithm). When all the individual solutions have their own fitness, the population is evolved using the evolutionary operators (i.e. selection of the best individual solutions, correlation, soft mutation and hard mutation). In the correlation step, the individual solutions are ordered by their fitness or

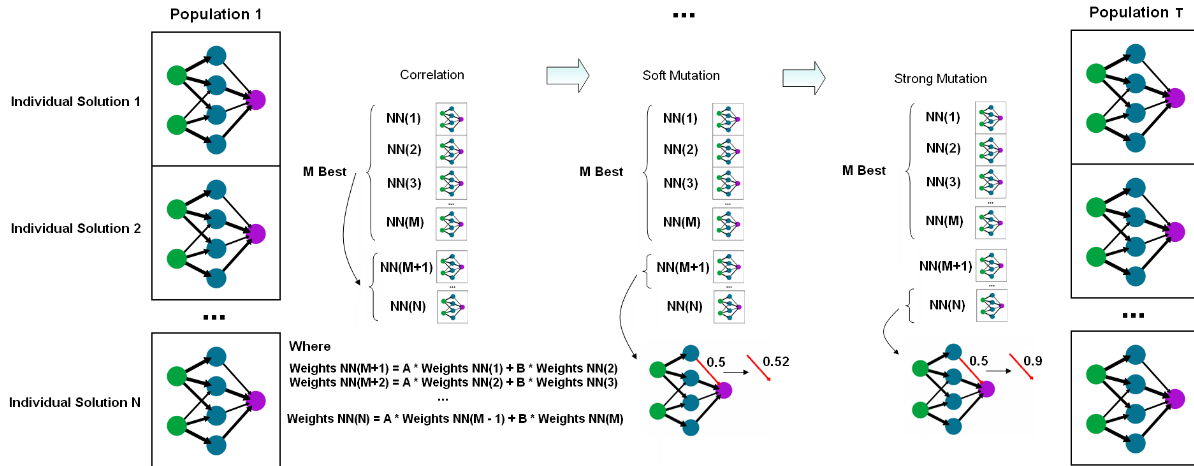


Figure 12. Evolutionary RL with Neural Networks.

cumulative reward. The M best individual solutions propagate their weights to the rest of the population in the way shown in Figure 12. The correlation step is an effective form of guaranteeing offspring viability since weights can be set in order to generate individuals very close to one of its parents. In addition, this strategy guarantees the interrelation of the system since this causes a chain of correlations that propagates over the whole population. In the mutation steps, the algorithm randomly changes the weights of the last $N - M$ individual solutions. In soft mutation this change is a slight change, while in strong mutation it is a big change. The algorithm is shown in Table 3.

Table 3. Evolutionay RL Algorithm.

Evolutionay RL Algorithm
1. Generate a initial population of N individual solutions
1.1. All the neural networks have the same topology
1.2. The weights are randomly initialized
2. For each individual solution
2.1. The agent interacts with the environment for τ episodes
2.2. Obtain the cumulative reward of this interaction using the individual solution.
3. Evolve the population, applying
3.1. Selection of the M best individual solutions
3.2. Correlation of the M best individual solutions
3.3. Soft mutation of the $N - M$ individual solutions
3.4. Strong mutation of the $N - M$ individual solutions
4. Return to 2 if the number of evolutions $< T$
5. Return Population T

7. Evaluation

RoboCup simulated soccer presents many challenges to RL methods: a large state space, hidden and uncertain states, multiple and independent agents learning simultaneously, and long and variable delays in the effects of actions. It has been used as the basis for international competitions and research challenges [25]. It is a distributed, multi-agent domain with teammates and adversaries. There are hidden states, meaning that each agent has only a partial view of the world at any moment. The agents have noisy sensors and actuators, meaning that they do not perceive the world exactly as it is, nor can they affect the world exactly as intended. In addition, the perception and action cycles are asynchronous, prohibiting the traditional AI paradigm of using perceptual input to trigger actions. Communication opportunities are limited, and the agents must make their decisions in real-time. These domain characteristics make simulated robot soccer a realistic and challenging domain. In this chapter, we present the results in the Keepaway domain [50], a subtask of the full and complex RoboCup soccer domain. This subtask of soccer, involves 5-9 players rather than the full 22. In the next section we describe the Keepaway domain. The next sections show the results obtained in this domain for the RL algorithms described previously: VQQL, CMAC-QL, 2SRL and Evolutionary RL.

7.1. Keepaway domain

In this subtask, the keepers try to maintain the possession of the ball within a region, while the takers try to gain it. An episode ends when the takers gain the possession of the ball or the ball leaves the playing region. When an episode ends, the players are reset for another episode. In our experiments each keeper learns independently. From a point of view of a keeper, an episode consists of a sequence of states, actions and rewards in the form:

$$s_0, a_0, r_1, s_1, ..., s_i, a_i, r_{i+1} \tag{3}$$

where a_i is selected based on some perception of s_i. We reward the keepers for each time step they keep possession, so we set the reward r_i to the number of time steps that elapsed while following action $a_{i-1} : r_i = t_i - t_{i-1}$. The keepers goal is to select an action such that the remainder of the episode will be as long as possible, and thus maximizes the total reward.

The state space is composed of several features that can be considered continuous. These features use information derived from distances and angles between the keepers, takers and the center of the playing area. The number of features used for each state depends on the number of players. In 3vs2 Keepaway (3 Keepers against 2 Takers) there are 13 features (Table 4).

The action space is composed of two different macro-actions, *HoldBall()* and *PassBall*(k_i) where k_i is the teammate *i*. The actions are available only when the keeper is in possession of the ball. In all the experiments reported, the 3vs2 keepaway is used and the size of the playing region is 25×25. In addition, the parameter setting for VQQL, CMAC-QL and 2SRL, obtained after an exhaustive experimentation, are the following: $\gamma = 1$ and $\alpha = 0.125$. In all cases, an $\epsilon - greedy$ strategy is applied, increasing the value of epsilon from 0 (random behaviour) to 1 (fully greedy behaviour) by 0.0001 in each episode.

Feature	Description
dist(K_1, C)	Distance between $Keeper1$ and the center of the playing region.
dist(K_2, C)	Distance between $Keeper2$ and the center of the playing region.
dist(K_3, C)	Distance between $Keeper3$ and the center of the playing region.
dist(T_1, C)	Distance between $Taker1$ and the center of the playing region.
dist(T_2, C)	Distance between $Taker2$ and the center of the playing region.
dist(K_1, K_2)	Distance between $Keeper1$ and $Keeper2$.
dist(K_1, K_3)	Distance between $Keeper1$ and $Keeper3$.
dist(K_1, T_1)	Distance between $Keeper1$ and $Taker1$.
dist(K_1, T_2)	Distance between $Keeper1$ and $Taker2$.
Min(dist(K_2, T_1), dist(K_2, T_2))	Minimum distance between $Keeper2$ and $Taker1$ and between $Keeper2$ and $Taker2$
Min(dist(K_3, T_1), dist(K_3, T_2))	Minimum distance between $Keeper3$ and $Taker1$ and between $Keeper3$ and $Taker2$
Min(ang(K_2, K_1, T_1), ang(K_2, K_1, T_2))	Minimum between the ang(K_2, K_1, T_1) and ang(K_2, K_1, T_2)
Min(ang(K_3, K_1, T_1), ang(K_3, K_1, T_2))	Minimum between the ang(K_3, K_1, T_1) and ang(K_3, K_1, T_2).

Table 4. The 13 state variables in 3vs2.

7.2. Results of VQQL

In this setting, we use VQ to discretize the state space. First, for each keeper, we design the vector quantizer composed of N centroids from the input data obtained during a random interaction between the keeper and the environment. Then, the Q function is learned, using Q-Learning, generating the Q table composed of N rows and a column for each action. The size of the discretized state space is 64. In 3vs2, we obtain the results shown in Figure 13. The *y-axis* represent the average time that the keepers are able to keep the ball from the takers and the *x-axis* is training time. In the graphs, there are ten learning curves. The average values raises from 7 seconds up to around 18 seconds.

7.3. Results of CMAC-QL

The intervals of the variables in the 3vs2 case in a 25×25 region are defined in Table 5. In our experiments we use single-dimensional tilings. For each variable, 32 tilings were overlaid, each offset from the others by a 1/32 of the tile width. For each state variable, we specified the width of the tiles based of the width of the generalization that we desired. For example, distances were given widths of about 3.0 meters, whereas angles were given

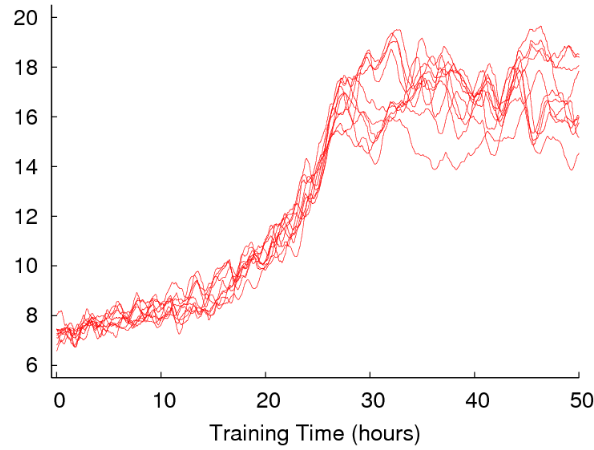


Figure 13. VQQL Results.

widths of about 10.0 degrees. This is the same parameter setting used in [50].

Table 5. Number of tiles per tiling.

Feature	Interval	*Tile* width	Tiles per tiling
dist(K_1, C)	[0, 17.67]	3	6
dist(K_1, K_2)	[0, 35.35]	3	11
dist(K_1, K_3)	[0, 35.35]	3	11
dist(K_1, T_1)	[0, 35.35]	3	11
dist(K_1, T_2)	[0, 35.35]	3	11
dist(K_2, C)	[0, 17.67]	3	6
dist(K_3, C)	[0, 17.67]	3	6
dist(T_1, C)	[0, 17.67]	3	6
dist(T_2, C)	[0, 17.67]	3	6
Min(dist(K_2, T_1), dist(K_2, T_2))	[0, 35.35]	3	11
Min(dist(K_3, T_1), dist(K_3, T_2))	[0, 35.35]	3	11
Min(ang(K_2, K_1, T_1), ang(K_2, K_1, T_2))	[0, 180]	10	18
Min(ang(K_3, K_1, T_1), ang(K_3, K_1, T_2))	[0, 180]	10	18

Then, the size of the primary vector $\vec{\theta}$ in 3vs2 is 4224 $(x1_{tiles} + x2_{tiles} + ... + x13_{tiles})$. In our work, we use a separate value function for each of the discrete actions. So there would be roughly 4224 tiles for the *HoldBall()* action, 4224 tiles for the *PassBall*(k_1) action and 4224 tiles for the *PassBall*(k_2) action (about 12672 in total). We obtain the results shown in Figure 14. In the graph, there are ten learning curves. The average values raises from 7 seconds up to around 25 seconds.

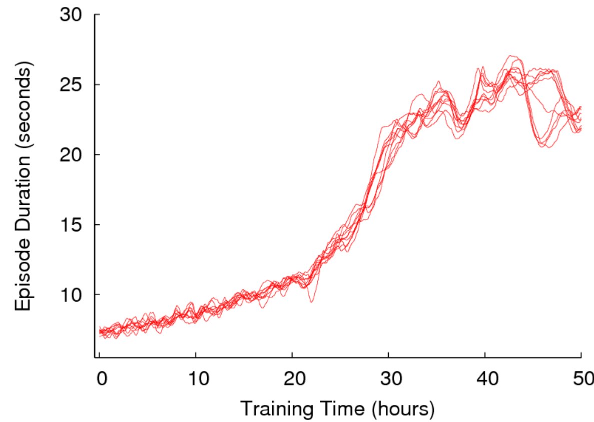


Figure 14. CMAC-QL Results.

7.4. Results of Two Steps RL

We use both M5 (regression) and J48 (C4.5, classification) algorithms to approximate the Q function for each iteration of the ISQL phase. In the case of J48 [61] we have applied the original 2SRL algorithm discretizing the reward function by hand and tested different discretizations to obtain an accurate one. In this section we only show the best results obtained when we discretize the reward function by hand, using 88 classes. For M5 [61], we use the continuous rewards adapted version.

The first tuple set was obtained with a random policy working in the keepaway domain.

We tested two different approaches for the approximation. The first approach uses only one approximation $\hat{Q}_(s, a_i)$ where the action is an input parameter. The second approach uses multiple approximations $\hat{Q}_{a_i}(s)$, one per action, to approximate the Q function. The Q table is initialized using the values obtained in the leaves of the M5/J48 tree. In 3vs2, we obtain the results shown in Figure 15.

For each graph in Figure 15, we can see 10 evolution curves using the same configuration and the same approximation approach. The title at the top of each graph shows the approximation approach used. The *y-axis* is the average episode length and the *x-axis* is the training time. The J48 curves were generated using the best discretization of the reward space obtained by hand with 88 classes. We can observe how the policy improves during the training time.

7.5. Results of Evolutionary RL

In this case, a population of 100 individuals for each keeper is evolved using the evolutionary algorithm. In each generation, each individual plays five episodes in the keepaway domain. The cumulative reward achieved by playing all individuals in the population determines the probability that an individual will be selected when the next generation is created. Each individual is an artificial neural network with 14 nodes in the input layer. Inputs to each network describe the agent's current state and the last action selected (13 inputs for the state features and 1 for the last action selected). There are eight nodes in the hidden layer

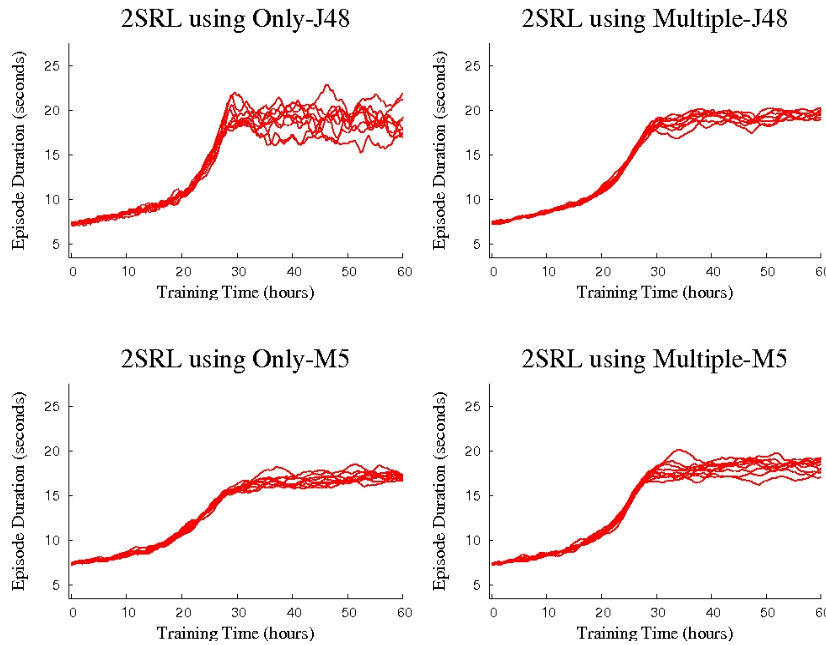


Figure 15. 2SRL algorithm results in Keepaway 3vs2.

and three nodes in the output layer. There is one output for each available action and the agent takes whichever action has the highest activation.

Each candidate network is evaluated by allowing it to control the keepers' behavior and observing how much reward it receives. The policy's fitness is the sum of the rewards accrued while under the network's control. It is necessary to evaluate each member of the population for many episodes to get accurate fitness estimates. In these experiments, each policy in keepaway is evaluated for an average of five episodes. Figure 16 shows 10 learning curves. The average values raise from seven seconds up to around 14 seconds.

7.6. Summary of the Results

The results of the four algorithms described in previous sections (VQQL, CMAC-QL, 2SRL and Evolutionary RL) are summarized in Figure 17. The graph in Figure 17 shows the mean and the standard deviation for the different algorithms. We can see the best result is achieved by CMAC-QL. In this case, the mean value grows from 7 seconds to 25 seconds approximately. Using the keepaway domain, our experiments show that Q-learning with CMAC can significantly improve the performance of other techniques. We believe this performance is approximately the optimal solution, as it matches the best results published by other researchers [50]. In any case, the effectiveness of the CMAC function approximator combined with Q-Learning is widely demonstrated on several control problems [51].

In the second place, we can see the results obtained by 2SRL when J48 trees is used as approximation technique. In this case, the mean value raises from 7 seconds up to around

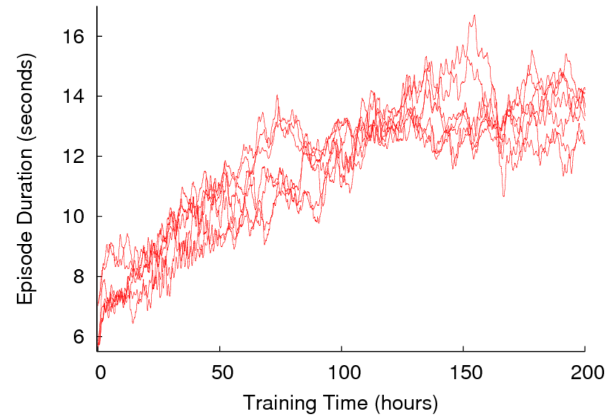


Figure 16. Evolutionary RL Results.

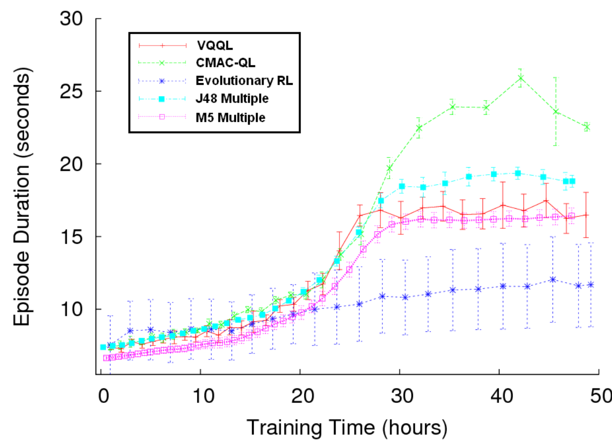


Figure 17. Mean and standard deviation obtained for the different algorithms in the keep-away domain.

20 seconds. The discretization is obtained after an exhaustive experimentation evaluating different discretizations. In contrast, M5 does not need to discretize by hand the reward space and it obtains similar results than J48. When we use M5 as approximation technique, the mean grows from 7 seconds to 18 seconds approximately. We are using one state space discretization per action. This is a difference with previous work based on state space discretizations that typically use the same discretization for each action. Here we show that in model free methods where the action-value function, $Q(s, a)$ must be approximated, it is possible that each action requires a different number of resources or state space discretizations.

In the third place, we find the results of the VQQL algorithm. VQQL's mean grows from 7 seconds to 19 seconds approximately. The results show that VQ is a successful method to solve the states generalization problem of RL algorithms. The experiments show how VQ and the generalized Lloyd algorithm allows us to dramatically reduce the number of states needed to represent a continuous environment (only 64). This technique gives us

more quality in the quantization of this continuous environment.

Finally, we can see the results obtained by Evolutionary RL. In this case, the mean raises from 7 seconds up to around 12 after 50 hours of training. In section 7.5., the performance grows from 7 seconds to 14 seconds approximately after 200 hours of training. The results in the Keepaway task show that Evolutionary algorithms can obtain good policies at the RL task. In this chapter, a comparison between TD methods and Evolutionary algorithm is given. A detailed comparison can be found in [53; 60]. These studies conclude that the choice between using Evolutionary algorithms and a TD method should be made based on some of the given task's characteristics. In deterministic domains, the fitness of an individual can be quickly evaluated and Evolutionary algorithms RL are likely to excel. If the task is fully observable but nondeterministic, TD methods may have an advantage. If the task is partially observable and nondeterministic, each method may have different advantages: TD methods in speed and Evolutionary algorithms in asymptotic performance.

8. Conclusions

This chapter began by suggesting two distinct approaches to solve RL problems with large or continuous state spaces; one can search in the value function space or in the policy space. Different TD methods and Evolutionary Algorithms RL, examples of these two approaches, are reviewed in this chapter. Unfortunately, conventional TD methods cannot be applied to problems with large or continuous state spaces; some form of generalization is required. This generalization is carried out in two different ways: by discretizing the state space to use a tabular representation of the value functions, or by using an approximation of the value function. As an example of the first approach, the paper reviewed the VQQL algorithm which use vector quantization to discretize the state space. For the second approach, the CMAC-QL algorithm, which uses a CMAC function approximator, is explained in detail. There are even algorithms that use both approaches. The 2SRL algorithm is the algorithm representative of this category reviewed in this chapter.

In this chapter, we have also shown that the use of vector quantization for the generalization problem of RL techniques provides a solution to how to partition a continuous environment into regions of states that can be considered the same for the purposes of learning and generating actions. It also solves the problem of knowing what granularity or placement of partitions is more appropriate. In relation to the CMAC-QL algorithm, this chapter reviewed the application of Q-Learning with linear tile-coding function approximation to a complex, multiagent task. CMAC is a function approximation method that strikes an empirically successful balance along representational power, computational cost and ease to use. The sucess of CMAC in practice depends in large part on parameter choices. The results obtained by the CMAC-QL algorithm show that RL can work robustly in conjunction with function approximators.

Then, we have described the 2SRL algorithm, that is composed of two main learning phases based on two approaches: supervised function approximation, using the Iterative Smooth Q-learning algorithm, and state space discretization methods, using the Multiple Discretization Q-Learning. The algorithm has two main advantages. First, the two phases have shown better results over methods based on only one of the approaches. So, this method can be applied to approaches which only execute the first phase. For instance, the

Q-RRL algorithm [12] is a Relational Reinforcement Learning algorithm very similar in structure to the Iterative Smooth Q-Learning algorithm presented here, but using relational data instead of feature based data. Furthermore, Q-RRL uses a logical Q-tree to approximate a Q-table, so a state space discretization is obtained in the same way that was presented in this chapter for J48 tree or M5. So, a second learning phase could be added over that state space representation obtained after executing Q-RRL to tune the Q-Learning approximation obtained. The second advantage is that the number of parameters and/or knowledge that must be introduced in order the algorithm to work correctly is very low, and hence, easily applicable to new domains.

Evolutionary RL algorithms is a general purpose optimization technique and can be applied to a wide variety of problems. We have used an Evolutionary RL algorithm to perform policy search RL and represent population of neural network action selectors. The results show that Evolutionary RL can learn good policies in the keepaway domain, though it requires a high number of evaluations to do so. In this chapter we can see RL problems can also be tackled without learning value functions, by directly search in the space of policies. Evolutionary methods, which simulate the process of Darwinian selection to discover good policies, are one way of conducting this search.

Evolutionary methods have fared better empirically on certain benchmark problems, especially those where the agent's state is partially observable [19; 18; 49]. In contrast, value functions methods have stronger theoretical guarantees [24; 33]. Evolutionary methods have been criticized because they do not exploit the specific structure of the RL problem. There have been few studies that directly compare these methods [18; 39], and in these studies rarely isolate factors critical to performance of each method [53; 60]. Unfortunately, since the TD and evolutionary research communities are disjoint and usually focus on different applications, there are no accepted benchmark problems or evaluation metrics.

Acknowledgements

References

[1] D. Aberdeen. A (revised) survey of approximate methods for solving partially observable markov decision processes. Technical report, National ICT Australia, 2003.

[2] J. S. Albus. Data storage in the cerebellar model articulation controller (cmac). *ASME Journal of Dynamical Systems, Measurement, and Control*, pages 228–233, 1975.

[3] J. S. Albus. A new approach to manipulator control: the cerebellar model articulation controller (cmac. *Journal of Dynamic Systems, Measurement, and Control*, 97:220–227, 1975.

[4] J. S. Albus. *Brains, Behaviour, and Robotics*. Byte Books, Subsidiary of McGraw-Hill, 1981.

[5] M. Asadi and M. Huber. State space reduction for hierarchical reinforcement learning. In V. Barr and Z. Markov, editors, *FLAIRS Conference*. AAAI Press, 2004.

[6] J. A. D. Bagnell and J. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the International Conference on Robotics and Automation 2001*. IEEE, May 2001.

[7] L. Baird and A. Moore. Gradient descent for general reinforcement learning. In *In Advances in Neural Information Processing Systems 11*, pages 968–974. MIT Press, 1998.

[8] R. Bellman. *Dynamic Programming*. Dover Publications, March 2003.

[9] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming (Optimization and Neural Computation Series, 3)*. Athena Scientific, May 1996.

[10] G. Burgin. Using cerebellar arithmetic computers. In *AI Expert 7*, pages 32–41, 1992.

[11] D. Chapman and L. P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. pages 726–731. Morgan Kaufmann, 1991.

[12] S. Dzeroski, L. D. Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43(1/2):7–52, 2001.

[13] E. Ersü and J. Militzer. Software implementation of a neuron-like associative memory system for control applications. In *2nd IASTED Conference on Mini- and Micro-Computer Applications – MIMI '82*, Davos, Switzerland, March 1982.

[14] F. Fernández and D. Borrajo. Vqql. applying vector quantization to reinforcement learning. In *RoboCup-99: Robot Soccer World Cup III*, pages 292–303, London, UK, 2000. Springer-Verlag.

[15] F. Fernández and D. Borrajo. Two steps reinforcement learning. *Int. J. Intell. Syst.*, 23(2):213–245, 2008.

[16] E. Frank and I. H. Witten. Generating accurate rule sets without global optimization. pages 144–151. Morgan Kaufmann, 1998.

[17] A. Gersho and R. M. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[18] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning (ECML)*, 2006.

[19] F. J. Gomez and J. Schmidhuber. Co-evolving recurrent neurons learn deep memory pomdps. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 491–498, New York, NY, USA, 2005. ACM.

[20] J. A. M. H. and J. de Lope Asiaín. Learning autonomous helicopter flight with evolutionary reinforcement learning. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *EUROCAST*, volume 5717 of *Lecture Notes in Computer Science*, pages 75–82. Springer, 2009.

[21] A. Hagens and J. H. Doveton. Application of a simple cerebellar model to geologic surface mapping. *Comput. Geosci.*, 17(4):561–567, 1991.

[22] A. M. Justin Boyan. Generalization in reinforcement learning: Safely approximating the value function. In G. T. . D. T. . T. Lee, editor, *Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.

[23] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH*, 4:237–285, 1996.

[24] M. Kearns. Near-optimal reinforcement learning in polynomial time. In *Machine Learning*, pages 260–268. Morgan Kaufmann, 1998.

[25] H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada. The robocup synthetic agent challenge,97. In *International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997.

[26] R. Koppejan and S. Whiteson. Neuroevolutionary reinforcement learning for generalized helicopter control. In *GECCO 2009: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 145–152, July 2009.

[27] M. Lang. *A real-time implementation of a neural-network controller for industrial robotics*. PhD thesis, Toronto, Ont., Canada, Canada, 1998. Adviser-D'Eleuterio, G. M.

[28] I. S. K. Lee and H. Y. K. Lau. Adaptive state space partitioning for reinforcement learning. *Engineering Applications of Artificial Intelligence*, 17:577–588, 2004.

[29] R. M. Lewis, V. Torczon, Michael, and M. W. Trosset. Direct search methods: Then and now. *Journal of Computational and Applied Mathematics*, 124:191–207, 2000.

[30] L. J. Lin. Scaling up reinforcement learning for robot control. In *ICML*, pages 182–189, 1993.

[31] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *Communications, IEEE Transactions on*, 28(1):84–95, 1980.

[32] I. López-Bueno, J. García, and F. Fernández. Two steps reinforcement learning in continuous reinforcement learning tasks. In *IWANN '09: Proceedings of the 10th International Work-Conference on Artificial Neural Networks*, pages 577–584, Berlin, Heidelberg, 2009. Springer-Verlag.

[33] T. L. D. M. L. Littman and L. P. Kaelbling. On the complexity of solving markov decision processes. In *In Proceedings of the 11th International Conference on Uncertainty in Artificial Intelligence*, 1995.

[34] W. T. Miller. Real-time application of neural networks for sensor-based control of robots with vision. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4):825–831, 1989.

[35] W. T. Miller, K. F. Arehart, and S. M. Scalera. On-line hand-printed character recognition using cmac neural networks. In *In Lendaris, G. G., Grossberg, S., & Kosko, B. (Eds.), World Congress on Neural Networks, WCNN'93*, pages 10–13, 1993.

[36] W. T. Miller, F. H.Glanz, and L. G. Kraft. Cmac: An associative neural network alternative to backpropagation. In *Proc. IEEE*, volume 78, pages 1561–1567, 1990.

[37] W. T. Miller, III, and A. L. Kun. *Dynamic Balance of a Biped Walking Robot*. MA: Academic Press, Boston, 1997.

[38] A. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In L. Birnbaum and G. Collins, editors, *Machine Learning: Proceedings of the Eighth International Conference*, 340 Pine Street, 6th Fl., San Francisco, CA 94104, June 1991. Morgan Kaufmann.

[39] D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. In *Machine Learning*, pages 11–32, 1994.

[40] D. M. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276, 1999.

[41] R. Munos. A study of reinforcement learning in the continuous case by the means of viscosity solutions. *Machine Learning*, 40(3):265–299, 2000.

[42] R. Munos and A. W. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49(2-3):291–323, 2002.

[43] L. M. Peshkin. *Reinforcement learning by policy search*. PhD thesis, Providence, RI, USA, 2002. Adviser-Kaelbling, Leslie.

[44] S. S. Representations, F. F. Andez, and D. Borrajo. On determinism handling while learning reduced. In *In Proceedings of the European Conference on Artificial Intelligence (ECAI 2002*, 2002.

[45] M. T. Rosenstein and A. G. Barto. Robot weightlifting by direct policy search. In *IJCAI'01: Proceedings of the 17th international joint conference on Artificial intelligence*, volume 2, pages 839–844, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[46] J. C. Santamara, J. C. Santamar'ia, R. S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6:163–218, 1998.

[47] D. W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization (Wiley Series in Probability and Statistics)*. Wiley-Interscience, September 1992.

[48] W. D. Smart. *Making reinforcement learning work on real robots*. PhD thesis, Providence, RI, USA, 2002. Adviser-Kaelbling, Leslie Pack.

[49] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:2002, 2001.

[50] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[51] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044. MIT Press, 1996.

[52] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. Mit Pr, May 1998.

[53] M. E. Taylor, S. Whiteson, and P. Stone. Temporal difference and policy search methods for reinforcement learning: An empirical comparison. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence*, pages 1675–1678, July 2007. Nectar Track.

[54] C. K. Tham and R. W. Prager. *Reinforcement learning for multi-linked manipulator control*. Cambridge University Engineering Department, Trumpington Street, Cambridge CB2 1PZ, UK, 1992.

[55] C. K. Tham and R. W. Prager. *Reinforcement learning methods for multi-linked manipulator obstacle avoidance and control*. Cambridge University Engineering Department, Trumpington Street, Cambridge CB2 1PZ, UK, 1993.

[56] H. Tolle, P. C. Parks, E. Ersuand, Hormel, and J. Militzer. Learning control with interpolating memories - general ideas, design-lay-out, theoretical approaches and practical applications. 56(2):291–317, 1992.

[57] J. N. Tsitsiklis and B. V. Roy. Feature-based methods for large scale dynamic programming. In *Machine Learning*, pages 59–94, 1994.

[58] W. T. B. Uther. *Tree based hierarchical reinforcement learning*. PhD thesis, Pittsburgh, PA, USA, 2002. Chair-Veloso, Manuela.

[59] J. Z. W. Xu, A.K. Nandi. A new fuzzy reinforcement learning vector quantization algorithm for image compression. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Hong Kong, June 2003.

[60] S. Whiteson, M. E. Taylor, and P. Stone. Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Journal of Autonomous Agents and Multi-Agent Systems*, 2009.

[61] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2 edition, June 2005.