Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces*

COINS Technical Report 96-088

December 1996

Juan Carlos Santamaría [†] carlos@cc.gatech.edu

Richard S. Sutton [‡] rich@cs.umass.edu

Ashwin Ram [†] ashwin@cc.gatech.edu

[†] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280

[‡] Lederle Graduate Research Center, University of Massachusetts, Amherst, MA 01002

Abstract

A key element in the solution of reinforcement learning problems is the value function. The purpose of this function is to measure the long-term utility or *value* of any given state and it is important because an agent can use it to decide what to do next. A common problem in reinforcement learning when applied to systems having continuous states and action spaces is that the value function must operate with a domain consisting of real-valued variables, which means that it should be able to represent the value of infinitely many state and action pairs. For this reason, function approximators are used to represent the value function when a close-form solution of the optimal policy is not available. In this paper, we extend a previously proposed reinforcement learning algorithm so that it can be used with function approximators that generalize the value of individual experiences across both, state and action spaces. In particular, we discuss the benefits of using sparse coarse-coded function approximators to represent value functions and describe in detail three implementations: CMAC, instance-based, and case-based. Additionally, we discuss how function approximators having different degrees of resolution in different regions of the state and action spaces may influence the performance and learning efficiency of the agent. We propose a simple and modular technique that can be used to implement function approximators with non-uniform degrees of resolution so that it can represent the value function with higher accuracy in important regions of the state and action spaces. We performed extensive experiments in the double integrator and pendulum swing up systems to demonstrate the proposed ideas.

*This work was supported by the NSF grant ECS-9511805 to Andrew G. Barto and Richard S. Sutton.

Contents

1 Introduction

A wide class of sequential decision-making problems are those in which the states and actions of the dynamic system must be described using real-valued variables. Common examples of these class of problems are found in robotics where agents are required to navigate around obstacles, move manipulators, or grasp tools or objects to accomplish a task. The states of the system in these examples corresponds to a set of variables representing positions and velocities of joints or body parts and the actions are a set of forces or torques that can be applied to the actuators. In this class of problems, an agent must select an action from infinitely many alternatives after every fixed time interval while basing its decision using the currently perceived state, which is also one of infinitively many possible states.

Reinforcement learning can be used to solve sequential decision-making problems. The main idea consists of using experiences to progressively learn the optimal *value function*, which is the function that predicts the best long-term outcome an agent could receive from a given state when it applies a specific action and follows the optimal policy thereafter. An agent can incrementally learn the optimal value function by continually exercising the current, non-optimal estimate of the value function and improving such estimate after every experience. More specifically, the agent can make a decision using the current, non-optimal estimate of the value function by selecting the action leading to the best value at the current state. Then, after observing the result of executing such action, the agent can use a reinforcement learning algorithm such as Sutton's TD(λ) algorithm ([22]) or Watkins' Q-learning algorithm ([26]) to improve the long-term estimate of the value function associated with such state and action. However, in systems having continuous state and action spaces, the value function must operate with real-valued variables representing states and actions, which means that it should be able to represent the value for infinitely many state and action pairs. This makes the learning problem very difficult because it is very unlikely that the agent would experience exactly same situations it has experienced before. Thus, the agent must be able to generalize the value of specific state-action combinations it has actually experienced to the situation it is currently facing so that it can make a good decision about what to do next. Unless a closed-form solution of the optimal value function is known in advance, the choice of the value function's representation is a real challenge. It should be able to handle real-valued variables as inputs, precisely map the state-action pairs to their assigned values, use memory resources efficiently, support learning without too much computational burden, and generalize the immediate outcome of specific state-action combinations to other regions of the state and action spaces.

Value functions are typically represented using function approximators, which use finite resources to represent the value of continuous state-action pairs. Additionally, they have parameters that the agent can use to adjust the value estimates with experience and consequently improve performance. There are several issues related to the design of function approximators. In this research, we explore two issues related to the following two questions in the context of reinforcement learning applied to systems with continuous state and action spaces:

1. **How can the agent use function approximators to generalize the outcome of experiences involving specific state-action pairs to other regions of the state and action spaces?**

   A common approach that have been used to represent the value function is to quantize the state and action spaces into a finite number of cells and aggregate all states and actions within

each cell (e.g., [10, 23, 3]. This is one of the simplest forms of generalization in which all the states and actions within a cell have the same value. Thus, the value function is approximated as a table in which each cell has a specific value. However, there is a compromise between the efficiency and accuracy of this class of tables that is difficult to resolve at design time. In order to achieve accuracy, the cell size should be small to provide enough resolution to approximate the value function. But as the cell size gets smaller, the number of cells required to cover the entire state and action spaces grows exponentially, which causes the efficiency of the learning algorithm to deteriorate because more data is required to estimate the value for all cells.

Another approach is to avoid the problems associated with quantityzing the state space altogether by using other types of function approximators, such as neural networks, that do not rely on quantization and can been used to generalize the value function across states (e.g., [9, 18]). The approach consists of associating one function approximator to represent the value of all the states and one specific action. For this reason it is useful for systems with continuous states and discrete actions. In this way, the agent can generalize the value of one specific state and the given action to other states in the neighborhood and the same action. The approach can be extended to systems with continuous states and actions by quantizing the action space into a finite number of cells and associating one function approximator to each action cell as before. In this way, all actions within an action cell are aggregated into one and assigned the same value, but keeping the state continuous. However, as with the quantized table approach, when the number of action levels increases, more function approximators are required, and the efficiency of the learning algorithm deteriorates due to the inability of the function approximator to generalize across action levels. Moreover, it is not possible to use specific state-action experiences to improve the estimate of more than one function approximator, even when the action value is near the boundary of two different action cells.

In this technical report, we propose an approach that extend the use of function approximators to systems with continuous states and action spaces. The approach consists of using only one function approximator over the combined state and action spaces. Therefore, it does not require quantization of the state nor action spaces and it can be used to generalize state-action experiences to other regions of the states and actions spaces. We evaluate the feasibility of the proposed approach in two widely known systems with continuous state and action spaces: double integrator and pendulum swing up.

2. **What is the effect of designing function approximators with non-uniform resource preallocation?**

   An important decision to consider when designing function approximators deals with resource preallocation across state and action spaces. Resource preallocation refers to the distribution of adjustable memory elements across the input domain. This is an important decision because such distribution affects the way the function approximator represents the value of state-action pairs. In general, the more the adjustable memory elements are available, the more the resolution of the function approximator. A good design strategy when no prior knowledge about the system is available is to preallocate resources uniformly across the entire state and action spaces. In this way, the function approximator will be able to represent the value function with the same resolution everywhere, which is desirable since there is no reason to favor some regions of the state-action space more than others. However, it is possible to design function approximators with non-uniform distribution of resources in order to achieve different degrees of resolution in different regions of the state-action space.

There are several reasons why designing non-uniform function approximators may be more beneficial than designing uniform ones. First, it is unlikely that there is absolutely no prior knowledge of the system available; often the designers know, up to a certain degree, what regions of the state-action space will be used more often. In such cases, the premise that lead to the design of a function approximator with uniform resources preallocation no longer applies. Presumably, it may be an advantage to design the function approximator such that it uses fewer resources in regions of the state-action space that are known to be visited rarely while use more resources in other, more heavily transited regions. Second, given fixed amount of resources, a non-uniform function approximator may lead to better performance and learning efficiency than that achieved with a uniform function approximator just because the former is able to exploit the resources more efficiently than the later. Finally, it may be possible to design function approximators that dynamically allocate more resources in certain regions of the state-action space and increase the resolution in such regions as required to perform on-line.

In this technical report, we describe a modular method that can be used to implement a non-uniform function approximator given the implementation of an uniform one. Thus, the method takes advantage of the conceptually simple and computationally efficient implementation of uniform function approximators and yet produces results equivalent to the non-uniform ones. We explore the feasibility of this idea by implementing uniform and non-uniform versions of three types of sparse coarse-coded function approximators, CMAC, instance-based, and cased-based, and compare the results in learning efficiency and final performance on the double integrator and the pendulum swing up.

Section 2 describes the problem formulation and introduces the concepts that will be used later in the paper. Section 3 presents the role of function approximators in reinforcement learning and also describes in detail the two classes of sparse coarse-coded function approximators used in the experiments. Section 4 introduces the topic of resource preallocation in function approximators and describes a simple technique to obtain non-uniform resource preallocation while keeping the simplicity and computational efficiency of function approximators with uniform preallocation. The results of experiments performed with uniform and non-uniform sparse coarse-coded function approximators in two different reinforcement learning problems is presented in section 5 and section 6 analyzes these results. Section 7 concludes the paper.

2 Reinforcement Learning

2.1 Definition

In reinforcement learning problems, a decision-maker or *agent* attempts to control a dynamic system by choosing actions in a sequential fashion. The agent receives a scalar value or *reward* with every action it executes. The ultimate goal of the agent is to learn a strategy for selecting actions or *policy* such that the expected sum of discounted rewards is maximized. The dynamic system, often referred to as the *environment*, is characterized by a *state*, and its *dynamics*, a function that describes the evolution of the state given the agent's actions. The state of the system captures all the information required to predict the future evolution of the system given agent's actions. It is assumed that the agent can perceive the state of the environment without error and base its current decision using this information.

2.2 The State-Value Function or Value Function

A key element in the solution of reinforcement learning problem is the state-value function (or simply, the value function) $V^\pi(x)$ associated with a given policy $\pi(x)$, which is defined as,

$$V^\pi(x_t) \equiv E\left[\sum_{k=1}^{\infty} \gamma^k r_{t+k}\right] \tag{1}$$

where x_t is the state of the system at time t, r_{t+k+1} is the reward received for performing action $u_{t+k} = \pi(x_{t+k})$ at time $t+k$, and γ is the *discount factor* $(0 < \gamma < 1)$. The state-value function measures the expected discounted sum of rewards or *expected return* the agent will receive when it starts from the given state and follows the given policy; therefore the name of *state-value* function. In particular, the optimal value function, $V^*(x)$ measures the maximum possible expected return the agent could receive when it starts from state x.[1]

When the agent is controlling a deterministic system and knows the one-step reward function $r_{t+1} = R(x_t, u_t)$, the dynamics function $x_{t+1} = F(x_t, u_t)$, and the optimal value function $V^*(x)$ for every state x and action u, then it can decide the optimal action to perform at every decision point by performing the following one-step lookahead search,

$$u_t^* = \pi^*(x_t) = \arg\max_{u_t \in U} \{R(x_t, u_t) + \gamma V^*(F(x_t, u_t))\} \tag{2}$$

The policy defined by Equation 2 is optimal because it selects the actions that maximizes the expected return starting from the given state. Additionally, it is easy to design efficient implementations of the one-step search in most of the cases.

2.3 The Action-Value Function or Q-Function

The one-step search using the value function is useful only when the agent knows the one-step reward and dynamics functions in advance. Alternatively, the action-value function (or simply, the Q-function as Watkins [26] defines it) measures the expected return of executing action u at state x_t, and then following the policy $\pi(\cdot)$ for selecting actions in subsequent states; therefore, the name of *action-value* function. The Q-function corresponding to policy $\pi(x)$ is defined as

$$Q^\pi(x_t, u_t) \equiv r_{t+1} + \gamma Q^\pi(x_{t+1}, \pi(x_{t+1})) \tag{3}$$

The advantage of using the Q-function is that the agent is able to perform the one-step lookahead search without knowing the one-step reward and dynamics functions. The disadvantage is that the domain of the Q-function increases from the domain of states $x \in X$ to the domain of state-action pairs $(x, u) \in X \times U$. When the agent knows the optimal Q-function, Q^*, it can select the optimal action using the following one-step search

$$u_t^* = \pi^*(x_t) = \arg\max_{u_t \in U} \{Q^*(x_t, u_t)\} \tag{4}$$

[1] In some problems, the rewards are always negative and it is more convenient to express them as positive costs. Then, to keep consistency, the agent should seek to minimize the discounted sum of costs instead of maximize the discounted sum of rewards. However, one should keep in mind that both formulations are mathematically equivalent.

The optimal Q-function can be computed using dynamic programming ([4]) when the one-step reward and dynamics functions are known in advance. However, we are interested in methods the agent can use to learn the optimal policy using its own experience. More specifically, in methods the agent can use to incrementally improve the estimates of the Q-function using the outcome of its previous actions.

2.4 Temporal Difference Methods

Temporal difference methods ([22]) exploit Equation 3 to create an update formula that an agent can use to asymptotically learn the Q-function function from observations. More specifically, every time the agent selects action u and observes the next state y and reward r, it can verify whether Equation 3 holds or not by computing the error between the predictions $Q^\pi(x,u)$ and $r + \gamma Q^\pi(y, \pi(y))$. When the error is different from zero, the update formula is used to adjust the estimate of $Q^\pi(x,u)$ such that the error is maximally reduced.

Sutton defines a whole family of update formulas for temporal difference learning called TD(λ), where $0 \leq \lambda \leq 1$ is a weight used to measure the relevance of previous predictions in the current error. The convergence of TD(λ) has been proved under different conditions and assumptions. Watkins [26] shows that the Q-function estimates asymptotically converge to their optimal values in systems having discrete and finite state and action spaces when TD(0) is used to perform the updates. A condition for the convergence is that all states are visited and all actions are executed infinitely often. Tsitsiklis and Van Roy [25] shows that the value function associated with a given policy converges for any linear function approximator and TD(λ) updates. There is no proof showing the convergence of TD(λ) for more complex function approximators, but this has not stopped researchers for trying these methods using different classes of non-linear function approximators. Successful results have been obtained with multi-layer neural networks (e.g., [9], [18]) and sparse coarse coding methods such as Cerebellar Model Articulation Controllers (*CMACs*) (e.g., [23], [24]). The next section describes the role function approximators in reinforcement learning problems with continuous state and action spaces.

3 Function Approximators

The optimal Q-function is sufficient for an agent to optimally perform the given task since the agent can use the one-step lookahead search (i.e., Equation 4) to select the optimal action at any given state. However, the agent does not have the optimal Q-function readily available from the beginning and it must learn such function using its own experience. Thus, the idea is to provide the agent with an initial estimate of the Q-function and let it decide the best action based on the current estimate. Then, the agent can use the outcome of each action to asymptotically improve the estimate towards the optimal action value. One way to accomplish this is to use a function approximators to represent Q-function. The function approximator will provide the agent with estimates of the expected returns of every state-action pair even when the state and action spaces are continuous. More specifically, a function approximator for the Q-function of a given policy is of the form $\hat{Q}_w(x,u)$, where x is the state, u is the action, and w is a set of adjustable parameters or *weights*. Given a current estimate of $\hat{Q}$, the agent can use the one-step search to select the best action at some state x_t according to

this estimate. Additionally, the agent may adjust the current estimate by modifying the weights of the function approximator using TD(λ) updates.

Function approximators are useful because they can generalize the expected return of state-action pairs the agent actually experiences to other regions of the state-action space. In this way, the agent can estimate the expected return of state-action pairs that it has never experienced before. Also, function approximators are able to represent the Q-function even when state and action spaces are continuous. However, note that a function approximator may not be able to accurately represent the Q-function for the entire state and action space due to its finite resources.

There are many classes of function approximators, each with advantages and disadvantages. The choice of a function approximator depends mainly in how accurate it is in generalizing the values for unexplored state-action pairs, how expensive is to store it in memory, and how well supports the computation of the one-step search and learning by TD(λ) updates.

3.1 One-Step Search

A function approximator for the Q-function can indicate the best action the agent can take at any given state. Such action is determined by performing the following one-step search:

$$\hat{u}_t^* = \arg\max_{u_t \in U} \left\{ \hat{Q}_{w_t}(x_t, u_t) \right\} \tag{5}$$

The actual implementation of the one-step search depends on the function approximator. A common technique used in problems with continuous state spaces is to use one function approximator for each action. Then, the one-step search can be performed by simply evaluating each function approximator at the given state. The best action is the one associated with the function producing the maximum value (see, for example, [9, 18, 23]). However, such technique becomes quickly impractical when the number of actions increases due to storage implications and computational efficiency because the number of function approximators is proportional to the number of actions. Additionally, since each function approximators operates in isolation, they are able to generalize the estimated return only across states. Also, they must be evaluated using only the action values from a finite predetermined set.

It is possible to overcome the difficulties of this technique at expense of more elaborate computation while performing the one-step search. The Q-function may be represented with only one function approximator and the optimal action at a given state may be found using some numerical optimization method may. For example, for function approximators that provide a direct measure of $\nabla_u \hat{Q}$, the best action could be found using a gradient descent method. The details in the implementation of the one-step search depends on the class of function approximator. However, the algorithm shown in Figure 1 is applicable for all classes of function approximators.

The computational complexity of Algorithm 1 depends on the step size Δ_u, the cost of evaluating $\hat{Q}_{w_t}(x_t, u_t)$, and the dimensionality of the action space. Additionally, since the resolution of the search in the action space is Δ_u, the resulting action value is not truly continuous throughout the action space. However, the advantage is that only one function approximator is being used to represent the Q-function instead of $O(U/\Delta_u)$. Clearly, this is not the most efficient algorithm but is conceptually simple and widely applicable.

Input: state x_t and weights w_t
Output: best action $\hat{u}^*$
Algorithm:

1. For every action u_i from $u_{\min}$ to $u_{\max}$ at step Δ_u:
   (a) Evaluate $\hat{Q}_{w_t}(x_t, u_i)$.
   (b) Keep the action $\hat{u}^* = u_i$ that produces the maximum value.
2. Return $\hat{u}^*$

Figure 1: One-Step Search Algorithm.

3.2 Learning

An agent following policy $\pi(x)$ can improve the current estimate of $\hat{Q}_{w_t}$ by adjusting the weights after every state transition using TD(λ) updates. However, it is not possible to do so when the agent always chooses the action suggested by the given policy. The agent must also perform some *exploratory* actions to be able to determine their effect. For this purpose, every time the agent selects action u_t at state x_t, and observes the next state x_{t+1} and reward r_{t+1} that results from executing that action, it uses the following TD(λ) formula to update the weights of the function approximator,

$$\Delta w_t = \alpha \left(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t \right) \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{w_k} \hat{Q}_k \tag{6}$$

where $\hat{Q}_{t+1} = \hat{Q}_{w_t}(x_{t+1}, \pi(x_{t+1}))$, $\hat{Q}_t = \hat{Q}_{w_t}(x_t, u_t)$, $\nabla_{w_k}\hat{Q}_k$ is the gradient of $\hat{Q}$ with respect w and evaluated at the state-action pair (x_k,u_k), and α is a learning rate. The interpretation of Equation 6 is as follows: $\hat{Q}_t$ and $\hat{Q}_{t+1}$ represent the long-term outcome (i.e., discounted sum of rewards) associated with the current policy at states x_t and x_{t+1} respectively. The term $\left(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t \right)$ represents the error incurred by $\hat{Q}_t$ in predicting the future according to $\hat{Q}_{t+1}$. In other words, the Q-value of state x_t should be equal to the immediate reward r plus the Q-value of the next state x_{t+1} properly discounted. In case of error, the weights are proportionally modified in the direction of the gradient ∇_{w_t} in order to maximally reduce error. The discounted sum of the previous gradients ∇_{w_k} are also credited for the current error although their influence decays exponentially with λ.

The rightmost sum has the following recursive property:

$$\begin{aligned}
\mathbf{W}_{t+1} &= \sum_{k=0}^{t+1} (\lambda\gamma)^{t+1-k} \nabla_{w_k} \hat{Q}_k \\
&= \nabla_{w_{t+1}} \hat{Q}_{t+1} + (\lambda\gamma) \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{w_k} \hat{Q}_k \\
&= \nabla_{w_{t+1}} \hat{Q}_{t+1} + (\lambda\gamma) \mathbf{W}_t
\end{aligned} \tag{7}$$

An on-line implementation of the update rule given by Equation 6 is possible by taking advantage of this recursive property. The idea is to maintain the value of the rightmost sum in Equation 6 in a variable, $\mathbf{W}_t$, which can be easily update at every step as $\mathbf{W}_{t+1} = \nabla_{w_{t+1}} \hat{Q}_{t+1} + \lambda\gamma \mathbf{W}_t$. Another

Input: initial set of weights w_0
Algorithm:

1. Initialization:
   (a) perceive current state: x_{curr}
   (b) select action: $u_{\text{curr}} \leftarrow \epsilon\text{-}greedy\text{-}policy(x_{\text{curr}})$
2. Accumulate gradient:
   $\mathbf{W}_t \leftarrow \nabla_w \hat{Q}(x_{\text{curr}}, u_{\text{curr}}) + \lambda\gamma\mathbf{W}_{t-1}$
3. Perform action:
   Execute u_{curr}, observe resultant reward, r, and next state, x_{next}.
4. Select next action:
   $u_{\text{next}} \leftarrow \epsilon\text{-}greedy\text{-}policy(x_{\text{next}})$
5. Learn:
   $w_{t+1} \leftarrow w_t + \alpha(r + \gamma\hat{Q}(x_{\text{next}}, u_{\text{next}}) - \hat{Q}(x_{\text{curr}}, u_{\text{curr}}))\mathbf{W}_t$
6. Loop:
   $x_{\text{curr}} \leftarrow x_{\text{next}}$
   $u_{\text{curr}} \leftarrow u_{\text{next}}$
   If x_{curr} is a goal state then terminate else goto 2.

Figure 2: Gradient Descent version of the SARSA Algorithm.

approach to perform this computation is to store each component of the multidimensional variable W_t separately in an *eligibility trace*, e_t. The variable e_t represents the proportion of blame or "eligibility" the associated component in $\mathbf{W}_t$ has in the current error taking into account past information (for details, see, [20]).

In order to improve the Q-values, the policy must be chosen in such a way that both improve as the agent collects information. One way to accomplish this is to choose the action leading to the best estimate $\hat{Q}(x, \hat{u}^*)$ most of the time with ties resolved randomly. However, a small fraction, ϵ, of the time, an action is chosen randomly uniformly from the operating range. This policy is called ϵ-*greedy-policy* (see [23]).

Both performing and learning are put together in the *SARSA* algorithm ([18, 23]). The algorithm uses the ϵ-greedy-policy and the TD(λ) updates to improve the estimates of the Q-function with every experience and consequently the performance of the agent (see Figure 2).

3.3 Sparse Coarse-Coded Function Approximators

The one-step search and SARSA algorithms outlined in the previous section apply to any kind of function approximator. However, not all function approximators are equally efficient. The following are the main characteristics a function approximator should exhibit to be useful and efficient:

- **Generalization**: Refers to the ability of the function approximator to accurately generalize the values for unexplored state-action pairs. In systems with continuous state and action spaces, it is unlikely that the agent experiences exactly the same situation it has experienced before.

Thus, it is very important that the function approximator is able to extrapolate the values of experienced state-action pairs to unseen state-action pairs.

- **Resolution**: Refers to the granularity of the function approximator and its capacity to represent different values in small areas of the input space. Thus, the ability of the function approximator to be able to accurately represent the Q-function depends on the resolution.

- **Storage**: Refers to the memory resources used to implement the function approximator. The more storage the function approximator need the less usable becomes due to the cost associated with its maintenance. However, in most cases, the storage is in compromise with the resolution of the function approximator because the finer resolution the larger the storage.

- **Computational efficiency**: Refers to the complexity and efficiency of the one-step search and SARSA algorithms. An efficient function approximator must provide support for simple and efficient evaluation of state-action pairs and gradient computation because the agent performs these two operations extensively during the action selection and learning.

We concentrate our research in sparse coarse-coded function approximators because their properties nicely support these basic characteristics. Sparse coarse-coded function approximators are the most basic model of associative memory. They represent the content associated with some input using several physical memory locations and generalize the content by sharing these memory locations. The more two different inputs share memory locations, the similar their contents will be. The resolution, storage, and computational efficiency vary according to the specific type of function approximator and its implementation. A detailed description of sparse distributed memory and related models is found in [7].

The following subsections describe three types of sparse coarse-coded function approximators and outline their advantages and disadvantages.

3.3.1 Lookup Tables

The lookup table is one of the most common function approximators used to represent the Q-function when the states and actions spaces are discrete. It represent the extreme in the class of sparse coarse-coded memory function approximators because each input is associated with only one memory location. More specifically, each state-action pair is an element of the table that is used to store the current estimate of the value associated with that pair. The size of the lookup table is $O(NU)$, where N and U are the number of states and actions respectively. Lookup tables represent the extreme of sparse coarse-coded function approximators in which only one memory location is used to represent the Q-value associated with every state-action pair. For system characterized by continuous state and action spaces, each element of the lookup table is mapped into a cell in the state and action spaces. Thus, all states and actions within a region or *cell* are aggregated into one table element and are all assigned the same value. This is the basic form of generalization that lookup tables implement.

Lookup tables quickly become impractical for several reasons. First, state and actions spaces must be quantize into a finite number of cells. It is often difficult to determine an appropriate quantization scheme to provide enough resolution (i.e., accuracy) and low quantization error. Second, the number of cells grows exponentially with the number of variables and geometrically with the number of quantization levels. This creates storage and maintenance problems. Third, the rate of convergence

of the learning algorithm becomes extremely slow as the number of states and actions increases. Additionally, the quantized state and action spaces often create convergence problems because they form a non-Markovian representation of the dynamics of the system (see [12]).

Lookup tables efficiently support one-step searches and TD updates. The best action at a given state is found by indexing the table holding the state constant and performing a sweep across all possible actions values (Equation 4). Since there is only one parameter associated with every state-action pair in the lookup table, the TD updates are also easy to implement. Equation 8 shows a TD(0) update.

$$\Delta Q(x,u) = \alpha(r_{t+1} + \gamma \max_a \hat{Q}(y,a) - \hat{Q}(x,u)) \tag{8}$$

Watkins showed that under certain conditions this update rule can be used to asymptotically learn the optimal Q-values in systems having discrete state and action spaces [26]. Also, Peng and Williams described a way to use TD(λ) in a learning algorithm they call Q(λ) [15], which results in a learning algorithm with faster convergence rate. However, the conditions for convergence in these algorithms rarely hold in quantized versions of continuous state and action spaces.

3.3.2 Cerebellar Model Articulation Controller

Cerebellar Model Articulation Controller or CMAC is a class of sparse coarse-coded memory that models cerebellar functionality [1]. Each input or state-action pair activates a specific set of memory locations or *features*, the arithmetic sum of whose contents is the value of the stored Q-value. The CMAC takes advantage of the continuous nature of the input and is able to store the necessary data in a physical memory of practical size.

A CMAC consists of several overlapping tilings of the state-action space to produce the feature representation. A query is performed by first activating all the features that contain the state-action input and then summing the values of all the activated features. Figure 3 shows a bidimensional example of a CMAC organization. CMACs have been widely used in conjunction with reinforcement learning [26, 23, 24].

The size of a CMAC depends on the number of tilings and the size of each tiling. Tilings are usually large to provide enough resolution and they grow exponentially with the number of variables. A common trick to avoid large tilings is to use a consistent random hashing function to collapse a large number of tiles in a tiling into a smaller set. CMACs with uniform tilings efficiently support one-step searches since the evaluation of the Q-values is not computationally expensive. In naives CMAC implementations, TD(λ) updates are proportional to the number of tiles in the CMAC since each tile holds an eligibility trace. More efficient implementations keep a list of non-zero traces and perform the updates only on those tiles (e.g., [6]). A description of the structure and basic operations of the CMAC follows:

- **Tile structure**: A CMAC consists of a set of N tilings: $\{ T_i, i = 1, \cdots, N \}$. Each tiling T_i consists of a set of N_i tiles or features that cover the entire input space of the tiling contiguously and without overlapping: $T_i = \{ f_{ij}, j = 1, \cdots, N_i \}$. Each tile f_{ij} has a weight w_{ij} and an eligibility e_{ij}.
- **Tile selection**: For each tiling T_i, the tile f_{ij} containing the query point (x_q, u_q) is activated. All active tiles are aggregated in the set $F(x_q, u_q)$.

$$F(x_q, u_q) = \{ f_{ij} \in T_i \mid (x_q, u_q) \in f_{ij} \} \tag{9}$$

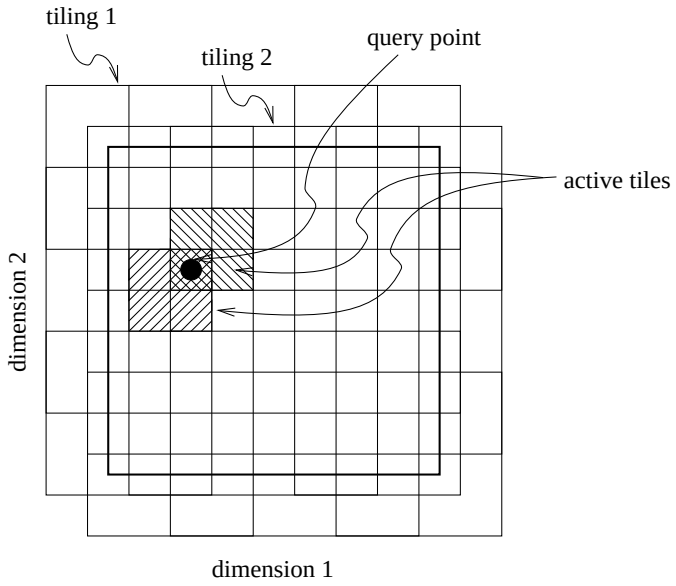

Figure 3: CMAC organization. Every state-action pair, represented by the dot, activates one tile in each tiling. The sum of the weights of all the activated tiles represents the vale associated with that state-action pair.

- **Function evaluation**: All the weight associated with the tiles in $F(x_q, u_q)$ are summed together.

$$\hat{Q}(x,u) = \sum_{f_{ij} \in F(x,u)} w_{ij} \tag{10}$$

- **TD(λ) update**: The learning update is performed to each tile in the CMAC.

$$\Delta w_{ij} = \alpha(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t)\, e_{ij} \quad \forall f_{ij} \in \text{CMAC} \tag{11}$$

  where $\hat{Q}_t$ and $\hat{Q}_{t+1}$ are the values of state-action pairs at (x_t, u_t) and (x_{t+1}, u_{t+1}) respectively.

- **Eligibility update**: The eligibilities of all the tiles in the CMAC are updated according to their contributions.[2]

$$e_{ij} \leftarrow \begin{cases} \frac{1}{|F(x_q,u_q)|} & \text{if } f \in F(x_q,u_q) \\ \lambda\gamma\, e_{ij} & \text{otherwise} \end{cases} \tag{12}$$

  where $|F(x_q, u_q)|$ is a constant[3] that represents the number of tiles in the set $F(x_q, u_q)$. Note that each tile in the CMAC represents a weight of the function approximator and each active tile has a the same contribution, namely $\frac{1}{|F(x,u)|}$, in estimating the value for (x_q, u_q).

The CMAC function approximator use computationally efficient methods for the selection of tiles when the size of each tile within each tiling is constant. The TD(λ) and the eligibilities updates are proportional to the number of tiles in the CMAC. This may deteriorate the performance of the CMAC in situations where some of the tiles are never used. The generalization and resolution of

[2]Equation 12 corresponds to replace of eligibilities (see [20]).

[3]In naive CMAC implementations, there is only one active tile per tiling. Thus, the number of active tiles per query point is always the same and equal to the total number of tilings in the CMAC (i.e., $|F(x_q, u_q)| = N$).

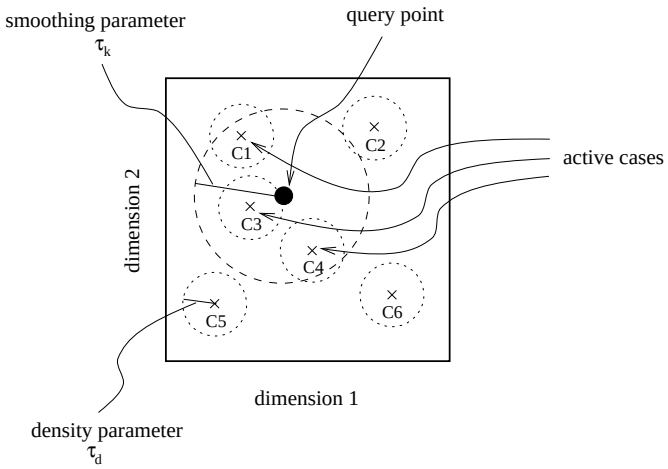

Figure 4: Memory-based function approximator. Every state-action pair, represented by the dot, activates the cases that are close to it according to some similarity metric such as Euclidean distance. The weighted average of the values of all the activated cases represents the vale associated with that state-action pair. Cases that are closer contribute more to the final value than cases that are farther. New cases are created when nearest neighbors are too far away.

the CMAC depends on the number of tilings and the number of tiles within each tiling. The more number of tiles the better the resolution and the more number of tilings the better the generalization. However, the storage is proportional to the total number of tiles.

3.3.3 Memory-Based Function Approximators

Another class of sparse coarse-coded memory is memory-based. Although, memory-based function approximators have not been widely used in conjunction with reinforcement learning, they are common in other tasks such as classification (e.g., [8]) and robot control (e.g., [2]) (but see [16, 14, 11]). In a memory-based function approximator, each memory element represents some of the state-action pairs or a *case* the agent has experienced before. A query is performed by first retrieving the nearest neighbors to the query point according to some similarity metric and then performing a weighted average of their Q-values. Additionally, new cases can be created and added to the memory when the nearest neighbors are too far away from a query point. In this way, the memory expands dynamically and on demand as new regions of the state-action space are being explored.

The size of the memory-based function approximator is dynamic. The memory starts empty and grows according to a *density threshold*, τ_d, that is used to determine when a new case should be added to the memory. Thus, when the distance of the nearest neighbor to a query point is greater than τ_d, a new case is added to the memory placed at the position of the query point. Small thresholds tend to produce function approximators with high resolution but with large amounts of memory due to the high density of cases. The choice of the density threshold depends on the problem. Figure 4 shows a bidimensional example of a memory-based function approximator.

The choice of the similarity metric depends on the application. A common measure of similarity metric in continuous domains is the Euclidean distance. This metric is useful when the Q-function is expected to be continuous and smooth throughout the state-action space. To compute the value associate with some input, a weighted average among nearest neighbors is performed using a kernel

function (e.g., Gaussian) with a *smoothing parameter*, τ_k, that controls the blending of values of nearest neighbors in the average. Generalization in memory-based function approximators results as a combination of the weighted average of nearest neighbors as determined by τ_k and the density of cases in memory as determined by τ_d. A good design is one in which the smoothing parameter is larger than the density parameter because in this way several cases contribute to the value of the query point while, at the same time, large portions of the domain are covered with fewer cases.

We describe the implementation of two memory-based function approximators: instance-based and case-based. They differ in the structure of the memory elements, the selection of the nearest neighbors, and the procedures used to compute the weighted averages. The instance-based version uses less storage and it is useful when the value function is smooth and continuous. The case-based version uses more storage but it is able to perform better generalizations and achieve greater resolution than the instance-based version. The description of the structure and the basic operations each function approximator follows:

- **Instance-Based Function Approximator**

  The cases in the instance-based function approximator have a simple structure. Every case consists of a state and action combination the agent has used in the past and stores the approximated Q-value for such combination. Only one distance function is required to generalize the values of cases across the combined state-action space. However, this may produce over-generalization because only the cases that are similar to both, the query state and the query action, are used to determine the Q-value of a state-action pair.

  - **Case structure**: Every case consists of a some state-action pair (x_i, u_i) the agent has experienced in the past, the associated value Q_i, and the eligibility e_i. That is, $C_i = (x_i, u_i, Q_i, e_i)$.
  - **Nearest neighbors selection**: A similarity function is used to compute the distance d_i from the query point (x_q, u_q) to the point (x_i, u_i) associated with each case C_i in memory. Then, the set of nearest neighbors to the query point, NN_q, is determined using the following rule:
    $$NN_q = \{C_i \in \text{Memory} \mid d_i \leq \tau_k\} \tag{13}$$
  - **Function evaluation**: A kernel function, $K(\cdot)$ (e.g., Gaussian $K(d_i) = \exp(-d_i^2/\tau_k^2)$), is used to compute the relative contribution of case C_i in the value of the query point. The final value is the average of the nearest-neighbor cases weighted using their relative contributions.
    $$\hat{Q}(x_q, u_q) = \sum_{\forall\, C_i \in NN_q} \frac{K(d_i)}{\sum_j K(d_j)} Q_i \tag{14}$$
  - **TD(λ) update**: The learning update is performed to each case in memory.
    $$\Delta Q_i = \alpha(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t)\, e_i \quad \forall C_i \in \text{Memory} \tag{15}$$
    where $\hat{Q}_t$ and $\hat{Q}_{t+1}$ are the values of state-action pairs at (x_t, u_t) and (x_{t+1}, u_{t+1}) respectively.

- **Eligibility update**: The eligibilities of all the cases in memory are updated according to their relative contributions.[4]

$$e_i \leftarrow \begin{cases} \frac{K(d_i)}{\sum_j K(d_j)} & \text{if } C_i \in NN(x,u) \\ \lambda\gamma\, e_i & \text{otherwise} \end{cases} \tag{16}$$

- **Case addition criterion**: A new case is added to the memory when the distance to the closest neighbor, $d_{\min} = \min\{d_i\}$, is larger than the threshold parameter τ_d.

- **Case-Based Function Approximator**

  The case-based function approximator use cases that contain more information than state-action pairs. Each case represents a region of the state space the agent has visited in the past, a set of actions the agent may or may have not executed in that region of the state space, and the set approximated Q-values associated to each action. Additionally, it uses two distance functions and two kernel functions to generalize and blend the values across the state and action spaces independently. For this reasons, the case-based function approximator has better resolution and generalization capabilities than the instance-based function approximator and may prove useful when the Q-function changes abruptly across actions within similar states. However, the case-based version requires more computation to perform the function evaluation and the TD(λ) updates than the instance-based version.

  - **Case structure**: Every case consists of a some state point x_i the agent has experienced in the past, an associated value Q_i that generalizes the Q-value across all the actions at that state, and the eligibility e_i. Additionally, the case stores a set of a set of N_i different actions u_{ij}, their associated Q-values Q_{ij}, and their eligibilities e_{ij} ($j = 1, \cdots, N_i$). That is, $C_i = (x_i, Q_i, e_i, \{u_{ij}\}, \{Q_{ij}\}, \{e_{ij}\} \mid j = 1, \cdots, N_i\ )$.
  - **Nearest neighbors selection**: A similarity function associated with the state space is used to compute the distance d_i^x from the query state x_q to the state x_i of each case C_i in memory. Then, the set of nearest neighbors to the query state, NN_q, is determined using the following rule:

$$NN_q = \{C_i \in \text{Memory} \mid d_i^x \leq \tau_k\} \tag{17}$$

    Thus, in the case-based version, only the similarity across the state space is considered to find nearest neighbors. As a consequence, a case may be selected as a near neighbor even when the query action is very different than the one stored in the case. However, each case can contribute with many different values to determine the Q-value of the query action. This produces an increase in both, the generalization and the resolution of the function approximator because the Q-value of an state-action pair may be estimated approximately even when the agent has tried only a few number of actions at similar states and, at the same time, there are several elements within each case to represent the Q-value with higher accuracy for different actions at the similar states.
  - **Function evaluation**: The evaluation of the function approximator at query point (x_q, u_q) requires two kernel functions: $K^x(\cdot)$ and $K^u(\cdot)$. The former one is used to determine the relative contribution of each case C_i to the value of the query state x_q. The later

[4]Equation 16 corresponds to replace of eligibilities.

one is used to determine the relative contribution of each action u_{ij} within each case to the value of the query action u_q.

The evaluation is performed in two phases: during the first phase a similarity function associated with the action space is used to compute the distance d^u_{ij} from the query action u_q to the action u_{ij} for each action in case C_i in the nearest-neighbor set. Then, the Q-value for the query action, $Q_i(u_q)$, is determined for each case using the following weighted average:

$$Q_i(u_q) = (1-\rho)Q_i + \rho\left(\sum_{\forall\, u_{ij}\in C_i} \frac{K^u(d^u_{ij})}{\sum_j K^u(d^u_{ij})} Q_{ij}\right) \quad \forall C_i \in NN_q \tag{18}$$

where ρ is a parameter that controls the blending between the Q-value associated with the state, Q_i, and the Q-values associated with each action, Q_{ij}. Thus, $Q_i(u_q)$ represents the Q-value of the query point (x_q, u_q) according to case C_i, which is a blend between the Q-value associated with the state and the weighted average of the Q-values associated with the actions.

During the second phase the distances associated with state space d^x_i's and the kernel function $K^x(\cdot)$ are used to determined the relative contribution of each case C_i and merge the $Q_i(u_q)$'s into a weighted average. The following equation shows the computation.

$$\hat{Q}(x_q, u_q) = \sum_{\forall\, C_i \in NN_q} \frac{K^x(d^x_i)}{\sum_j K^x(d^x_j)} Q_i(u_q) \tag{19}$$

- **TD(λ) update**: The learning update is performed to each case in memory and each action within a case.

$$\Delta Q_i = \alpha(r_{t+1} + \gamma\hat{Q}_{t+1} - \hat{Q}_t)\, e_i \quad \forall C_i \in \text{Memory} \tag{20}$$

$$\Delta Q_{ij} = \alpha(r_{t+1} + \gamma\hat{Q}_{t+1} - \hat{Q}_t)\, e_{ij} \quad \forall u_{ij} \in C_i \tag{21}$$

where $\hat{Q}_t$ and $\hat{Q}_{t+1}$ are the values of state-action pairs at (x_t, u_t) and (x_{t+1}, u_{t+1}) respectively.

- **Eligibility update**: The eligibilities of all the cases in memory and all the actions within a case are updated according to their relative contributions.[5]

$$e_i \leftarrow \begin{cases} (1-\rho)\frac{K^x(d^x_i)}{\sum_j K^x(d^x_j)} & \text{if } C_i \in NN(x,u) \\ \lambda\gamma\, e_i & \text{otherwise} \end{cases} \tag{22}$$

$$e_{ij} \leftarrow \begin{cases} \rho\frac{K^u(d^u_i)}{\sum_j K^u(d^u_j)} & \text{if } C_i \in NN(x,u) \\ \lambda\gamma\, e_{ij} & \text{otherwise} \end{cases} \tag{23}$$

$$\tag{24}$$

- **Case addition criterion**: A new case is added to the memory when the distance to the closest neighbor, $d^x_{\min} = \min\{d^x_i\}$, is larger than the threshold parameter τ_d.

Memory-based function approximators are more memory efficient than CMACs but require more computation to calculate the Q-value for a given state-action pair and to perform the TD(λ) updates. However, the main advantage of memory-based function approximators is their ability to dynamically allocate resources in regions of the state-action space that require them the most.

[5] Equations 22 and 23 corresponds to replace of eligibilities.

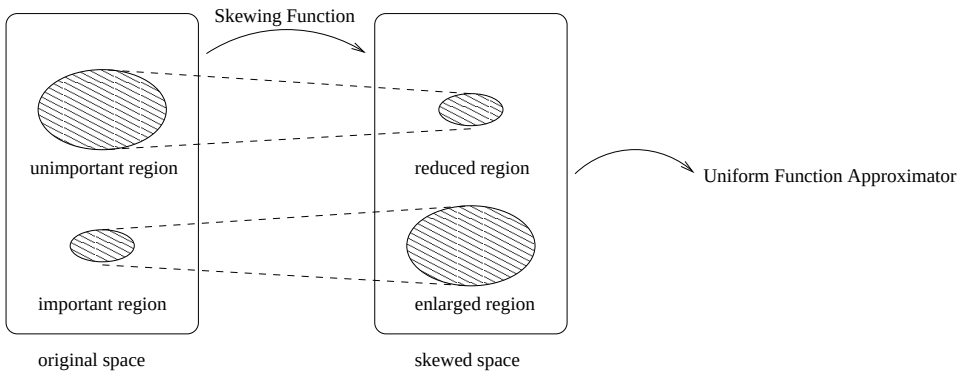

Figure 5: Preallocating resources non-uniformly through the use of a skewing function. The function expands important regions and compresses unimportant ones. The skewed space) is the input to the uniform function approximator.

4 Non-uniform Preallocation of Resources

Function approximators may not have enough resolution to represent the Q-function in the entire state-action space. However, each class of function approximator provide different ways of distributing resources non-uniformly across the state-action space so that some regions will have more resolution than others. For example, CMACs may use tilings with non-uniform tile sizes; while memory-based function approximators may use different density thresholds in different regions of the state-action space. However, designing non-uniform function approximators more complex than designing their uniform counterparts. Moreover, non-uniform designs often hinder the computational efficiency and memory storage of the function approximator. The idea of distributing the resources non-uniformly is inspired by quantization techniques widely used in digital communication theory. The objective is to approximate an analog signal with a quantized version so that the error between the two is minimized. For that purpose, more quantization levels are allocated to those regions where the amplitude of the signal is more frequently used, which reduces the average error performed in the quantization process (see, for example, [19]).

We proposed a technique that takes advantages of the simplicity and efficiency of the uniform versions of function approximators and yet are equivalent to the more elaborate, but more accurate non-uniform counterparts. The technique consists in distorting the state-action space by applying a *skewing function* and then use the *skewed state-action space* as the input to the uniform function approximator. In this way, the skewing function can be designed to expand or compress different regions of the state-action space. Expanded regions cover more area in the skewed state-action space; hence, they will use more resources of the function approximator and, consequently, will have better resolution to map the value function. Conversely, compressed regions cover less area, use fewer resources of the function approximator, and have less resolution to map the value function. There are no restrictions on the skewing function besides being able to map each point of the state-action space into some point on the skewed state-action space. Figure 5 demonstrates the graphically the process of preallocating resources non-uniformly.

The design principle we will follow in this research to specify the skewing function is inspired by optimization techniques used in digital communication theory. In that field, the skewing function is determined so that it optimize some design criterion such as mean square quantization error given the probability distribution of the input signal. The two main factors that influence the mean square

Input: state x_t, weights w_t, and skewing function $s(\cdot,\cdot)$
Output: best action $\hat{u}^*$
Algorithm:

1. For every action u_i from $u_{\min}$ to $u_{\max}$ at step Δ_u:
   (a) **Compute skewed state-action pair**: $(x^s, u^s) \leftarrow s(x_t, u_i)$.
   (b) Evaluate $\hat{Q}_{w_t}(x^s, u^s)$.
   (c) Keep the action $\hat{u}^* = u_i$ that produces the maximum value.
2. Return $\hat{u}^*$

Figure 6: One-Step Search Algorithm with Skewing Function.

quantization error are the difference between the quantized and real value at some input point and the the number of times such input point is evaluated. The mean quantization error is proportional to the product of both, the actual error at some input point and the number of times that input point is used. Thus, the idea behind the optimization of the mean squared quantization error is to reduce the error more heavily on the input points that are more frequently used.

Following this principle, we can empirically design the skewing function such that the function approximator have more resolution on the regions of the state-action space that are more frequently used. The increased resolution in those regions will allow the function approximator to be closer to the real value (given enough experience) and incur to less error, which is desired. A priori knowledge of the system can guide the design of the skewing function. Although the only knowledge that is required is about the expected frequency of use of regions in the state-action space. The skewing function should be designed in such a way that expands frequently used regions and compresses those that are used infrequently. Alternatively, the function approximator could be design to dynamically allocate more resources in those regions of the state-action space that are use more frequently. In this way, the resolution of the function approximator increases automatically without the need of prior knowledge.

4.1 Implementation of Non-Uniform Functions Approximators

The one-step search and the SARSA algorithms require little modification to accommodate the use of non-uniform function approximators. In both cases, the standard implementation of the uniform function approximator is used over the skewed state-action pair to accomplish the effect of non-uniform resource preallocation. Figures 6 and 7 show the new versions of the one-step search and the SARSA algorithms (additions shown in bold).

5 Results

This section describes the results of using uniform and non-uniform versions of CMAC, instance-based, and case-based function approximators in problems with continuous state and actions spaces. The problems studied are the double integrator and the pendulum swing up. The first one is a linear

Input: initial set of weights w_0, and skewing function $s(\cdot, \cdot)$
Algorithm:

1. Initialization:
   (a) perceive current state: x_{curr}
   (b) select action: $u_{\text{curr}} \leftarrow \epsilon$-*greedy-policy*$(x_{\text{curr}})$
2. **Compute skewed state-action pair**: $(x^s_{\text{curr}}, u^s_{\text{curr}}) \leftarrow s(x_{\text{curr}}, u_{\text{curr}})$.
3. Accumulate gradient:
   $\mathbf{W}_t \leftarrow \nabla_w \hat{Q}(x^s_{\text{curr}}, u^s_{\text{curr}}) + \lambda\gamma\mathbf{W}_{t-1}$
4. Perform action:
   Execute u_{curr}, observe resultant reward, r, and next state, x_{next}.
5. Select next action:
   $u_{\text{next}} \leftarrow \epsilon$-*greedy-policy*$(x_{\text{next}})$
6. **Compute skewed state-action pair**: $(x^s_{\text{next}}, u^s_{\text{next}}) \leftarrow s(x_{\text{next}}, u_{\text{next}})$.
7. Learn:
   $W_{t+1} \leftarrow w_t + \alpha(r + \gamma\hat{Q}(x^s_{\text{next}}, u^s_{\text{next}}) - \hat{Q}(x^s_{\text{curr}}, u^s_{\text{curr}}))\mathbf{W}_t$
8. Loop:
   $x_{\text{curr}} \leftarrow x_{\text{next}}$
   $u_{\text{curr}} \leftarrow u_{\text{next}}$
   If x_{curr} is a goal state then terminate else goto 2.

Figure 7: Gradient Descent version of SARSA Algorithm with Skewing Function.

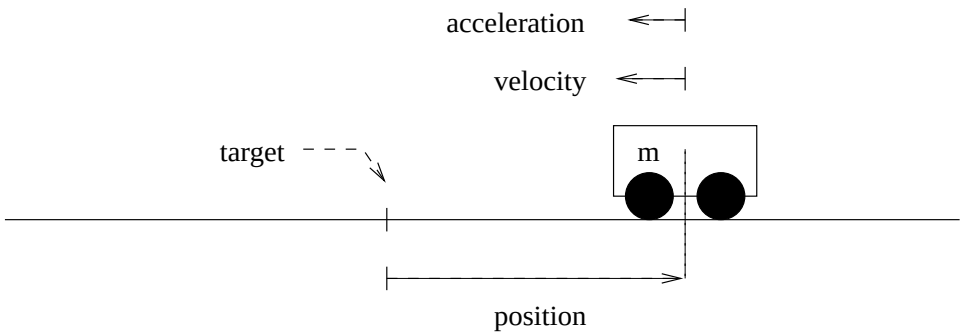

Figure 8: Double integrator. A car moving in a flat terrain subject to a the application of a single force.

dynamics system and the second one is a non-linear one, both with quadratic costs (i.e., negative quadratic rewards) that depend on the state and action values. The function approximators used in the double integrator are CMAC and instance-based and the ones used in the pendulum swing up are CMAC and case-based. The source code used to perform all the experiments is in a compressed tar file available through anonymous ftp at the following URL:

`ftp://ftp.cc.gatech.edu/pub/ai/students/carlos/RLI/experiments.tar.gz`

The source code is in `C++` and follows the standard software interface for reinforcement learning problems developed by Richard S. Sutton and Juan C. Santamaría.[6]

5.1 Double Integrator

The double integrator is a system with linear dynamics and bidimensional state. It represents a car of unit mass moving in a flat terrain and subject to the application of a single force (see Figure 8). The state of the system consists of the current position, p, and velocity v of the car, or $\mathbf{x} = [v\ p]^T$ in vectorial representation. The action is the acceleration, a, applied to the system, or $\mathbf{u} = [a]$ in vectorial representation. The objective is to move the car from a given starting state to the origin (i.e., $\mathbf{x_d} = [0\ 0]^T$) such that the sum of the rewards is maximized. The one-step reward function is a negative quadratic function of the difference between the current and desired position and the acceleration applied, $r_{t+1} = -(p_t^2 + a_t^2)$ or in vectorial representation, $r_{t+1} = -((\mathbf{x}_t - \mathbf{x_d})^T Q(\mathbf{x}_t - \mathbf{x_d}) + \mathbf{u}_t^T R \mathbf{u}_t)$, where Q and R are the positive definite 2×2 and 1×1 matrices respectively defined as $Q = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ and $R = [1]$. The one-step reward function penalizes the agent more heavily when the distance between the current and desired states is large and also when the action applied is large. This type of reward function is widely used in robotic applications because it specifies policies that drive the system to the desired state quickly while keeping the size of the driving control small. The tradeoffs between the two can be specified by appropriate selection of the Q and R matrices. This formulation is standard in optimal control theory ([21, 13]), in which the objective is to minimize costs instead of maximize rewards. However, both formulations are mathematically equivalent.

[6]The documentation for the standard software interface can be found in URL:
`http://envy.cs.umass.edu/People/sutton/RLinterface/RLinterface.html`.

The dynamics of the double integrator is very simple and it is described by the following equations,

$$\frac{dp}{dt} = v \tag{25}$$

$$\frac{dv}{dt} = a \tag{26}$$

The acceleration is bounded to be in the range between the minimum and maximum acceleration values (i.e., $a \in [a_{\min}\ a_{\max}]$, where $a_{\min} = -1$ and $a_{\max} = 1$). The double integrator is an instance of a more general class of linear dynamic systems and can be expressed using the more convenient linear matrix equation,

$$\mathbf{x}_{t+1} = \begin{bmatrix} v_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} v_t \\ p_t \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v_t \\ p_t \end{bmatrix} \Delta t + \begin{bmatrix} 1 \\ 0 \end{bmatrix} [a]\Delta t = \mathbf{x}_t + A\mathbf{x}_t\Delta t + B\mathbf{u}_t\Delta t \tag{27}$$

The simulation used a time step of $\Delta t = 0.05$ seconds and new control actions were selected every four time steps. A trial consists of starting the system at position $p = 1$ with velocity $v = 0$ and running the system until either 200 decision steps had elapsed (i.e. 40 simulated seconds) or the state gets out of bounds (i.e., when $|p| > 1$ or $|v| > 1$), whichever comes first. In the latter case, the agent receives a negative reward (i.e., a punishment) of 50 units to discourage it from going out of bounds. An experiment consists of 36 replications of 50 trials each, measuring the number of time steps and cumulative cost (i.e., negative sum of rewards) for each replication after each trial. The average number of time steps and cumulative cost across replications are used as measures of performance.

5.1.1 Optimal Solution

Systems with linear dynamics and quadratic cost functions have a simple closed-form solution for the optimal policy known as *Linear-Quadratic Regulator (LQR)*. The derivation follows from the solution to the Hamilton-Bellman-Jacobi partial differential equation (see, for example, [21] or [5] for details). Given a dynamical system with an n-dimensional state vector, $\mathbf{x}$, a p-dimensional action vector, $\mathbf{u}$, a linear dynamic function $\dot{\mathbf{x}} = A(\mathbf{x} - \mathbf{x_d}) + B(\mathbf{u} - \mathbf{u_d})$, where A and B are $n \times n$ and $n \times p$ matrices respectively, and a one-step quadratic cost (i.e., negative reward) function of the form $r = -((\mathbf{x} - \mathbf{x_d})^T Q(\mathbf{x} - \mathbf{x_d}) + (\mathbf{u} - \mathbf{u_d})^T R(\mathbf{u} - \mathbf{u_d}))$, where Q and R are positive definite $n \times n$ and $p \times p$ matrices respectively.[7] Then, the optimal policy is given by $\mathbf{u}^* = -K(\mathbf{x} - \mathbf{x_d}) + \mathbf{u_d}$ with an associated optimal value function given by $V(\mathbf{x}) = (\mathbf{x} - \mathbf{x_d})^T P(\mathbf{x} - \mathbf{x_d})$, where K and P are $p \times n$ and $n \times n$ matrices given by Equations 29 and 29 respectively. The last equation is known as the *continuous-time Ricatti equation* and it is used to find the unknown matrix P, which then is used in Equation 29 to find K also called the *gain* matrix [5, 17].[8]

$$K = R^{-1}B^T P \tag{28}$$

$$\dot{P} = -Q - A^T P - PA + PBR^{-1}B^T P \tag{29}$$

[7] The constant $\mathbf{x}_{\mathrm{d}}$ corresponds to the desired goal state and the constant $\mathbf{u}_{\mathrm{d}}$ corresponds to the required force to maintain the system at the desired state (i.e., once the system is in $\mathbf{x}_{\mathrm{d}}$, applying $\mathbf{u}_{\mathrm{d}}$ results in no change).

[8] The continuous-time Ricatti equation can be simplified further for time-invariant policies. This is the case when $t \to \infty$ and $\dot{P} \to \mathbf{0}$. Then the equation simplifies to, $Q + A^T P + AP = PBR^{-1}B^T P$. The resulting optimal controller is referred to as the Linear-Quadratic Regulator of the system.

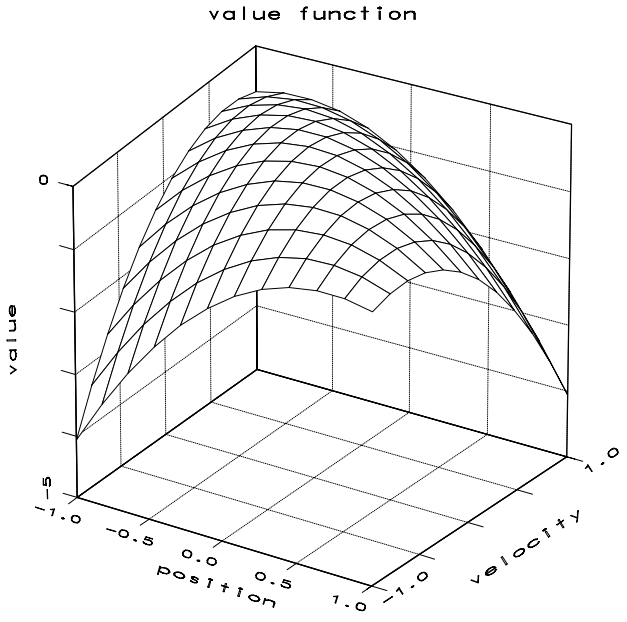

Figure 9: Optimal value function for the double integrator. The value function (vertical axis) is plotted against position and velocity (horizontal plane). States near the origin have values closer to zero than states far from the origin because it takes more time and cost to drive the system towards the origin.

In the double integrator, the values for A, B, Q, and R were described previously.[9] The solution of P in the time-invariant case is $P = \begin{bmatrix} \sqrt{2} & 1 \\ 1 & \sqrt{2} \end{bmatrix}$, which produces an equation for the optimal actions as $u^* = -Kx = -(\sqrt{2}\,v + p)$, where v and p are the instantaneous velocity and position respectively.

Figures 9 and 10 show the optimal value function and the optimal trajectory generated using the optimal policy. In the simulator the optimal controller achieves a performance of $-1.4293 \approx V(\mathbf{x_0}) = \sqrt{2}$ when the initial state is $\mathbf{x}_0 = [0\ 1]^T$. The discrepancy between the actual sum of rewards and the one predicted by the value function is due to the discretization across time and it is not considered significant. Figure 11 shows the optimal trajectory and the cumulative reward of the controller at different points of the trajectory.

5.1.2 Uniform CMAC

In this configuration, the agent used a CMAC to approximate the Q-function. There are three dimensions of interest: position, p, velocity, v, and acceleration, a. The CMAC used 36 tilings. Each of the three dimensions were divided into 12 intervals. 12 tilings depended on all three variables. 12 other tilings depended on position and velocity pairs only, and the remaining finally 12 tilings depended on one dimension only (6 for position and 6 for velocity). The tilings sharing the same dimensions were uniformly offseted across each dimension. Other CMAC configurations were tried

[9] In the double integrator $\mathbf{x}_\mathrm{d} = [0\ 0]^T$ and $\mathbf{u}_\mathrm{d} = [0]$

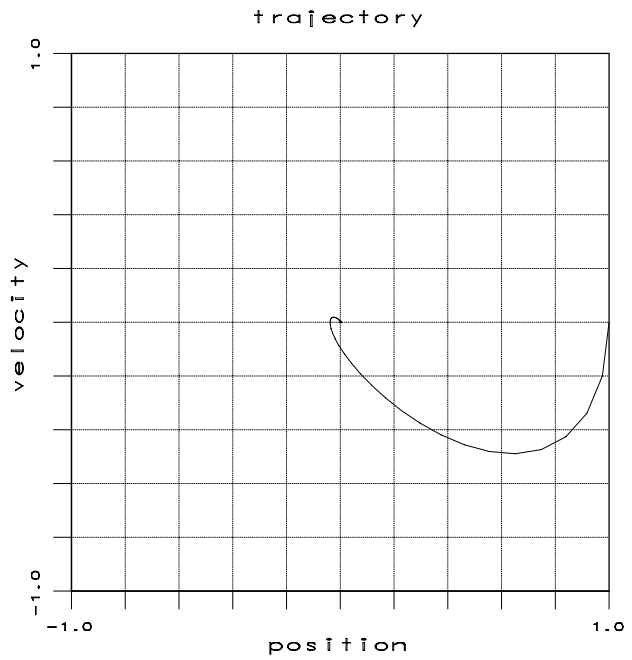

Figure 10: Optimal trajectory. The starting point is $\mathbf{x}_0 = [0\ 1]^T$ (center right) and goal point is $\mathbf{x_d} = [0\ 0]^T$ (center).

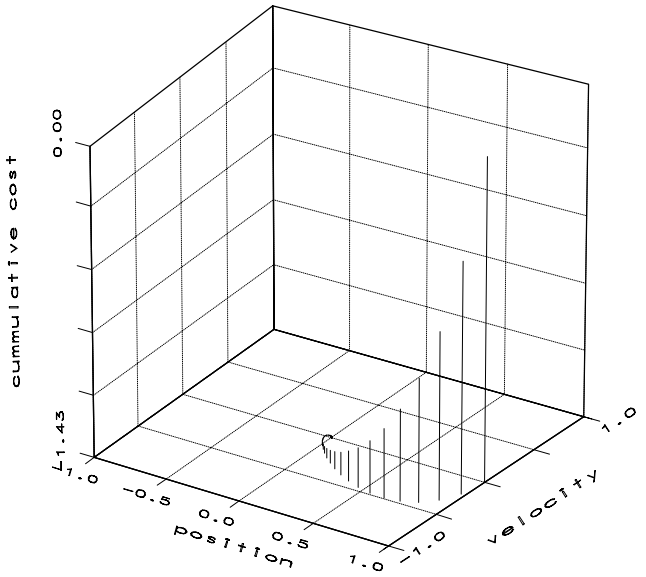

Figure 11: Optimal trajectory and discounted sum of rewards. The discounted sum of rewards (vertical axis) decreases faster when the system is far from the origin than when it is closer to the origin.

Table 1: Summary of the design of the agent using uniform CMAC function approximator for the double integrator.

Factor	Description
Variables	3: position: $p \in [-1\ 1]$; 12 intervals velocity: $v \in [-1\ 1]$; 12 intervals acceleration: $a \in [-1\ 1]$; 12 intervals
Tilings	36: 12 based on p, v, and a; uniformly offseted. 12 based on p and v; uniformly offseted. 6 based on p; uniformly offseted. 6 based on v; uniformly offseted.
One-step search	ϵ-greedy policy (Figure 1): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Learning	SARSA algorithm (Figure 2): $\gamma = 0.99$ $\alpha = 0.1$ $\lambda = 0.7$

and compared empirically; the configuration described here is the one that produced best results. The SARSA algorithm in Figure 2 was used for updating the weights of the CMAC using the traces indicated by Equation 12. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.7$, $\alpha = 0.1$, and $\epsilon = 0$. The one-step search was performed using 25 equally spaced values for the acceleration between the minimum value, $a_{\min} = -1$, and the maximum value, $a_{\max} = +1$ (i.e., $25\Delta_u = a_{\max} - a_{\min}$). Table 1 summarizes the agent's design.

Figures 12 and 13 show the average number of time steps and accumulated cost per trial respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

5.1.3 Non-uniform CMAC

In this configuration, the agent used the same CMAC configuration as in the previous experiment. However, the CMAC used skewed versions of the three variables of interest: position, velocity, and acceleration. The skewing functions were as follows: $p^s = \sqrt[3]{p}$, $v^s = \sqrt{v}$, and $a^s = \sqrt{a}$. Figure 14 shows the effect of the skewing functions of the class $s(x) = \sqrt[k]{x}$ for $k = 2, 3$. The function expands the region near the origin so that the CMAC can use more tiles to represent the value for the Q-function more accurately in that region. The larger the value of k the larger the expansion. Thus, the dimension corresponding to the position is skewed more heavily than the velocity and acceleration. We chose this skewing function design because we knew the system will spend more time near the origin orbiting around the goal state in order to reduce punishments. Thus, we inferred that the region near the origin of the state-action space will require more resolution of the function approximator. Additionally, the one-step reward function depends only with the position, which results in a Q-function that is more sensitive to the position than to the velocity (refer to the Q

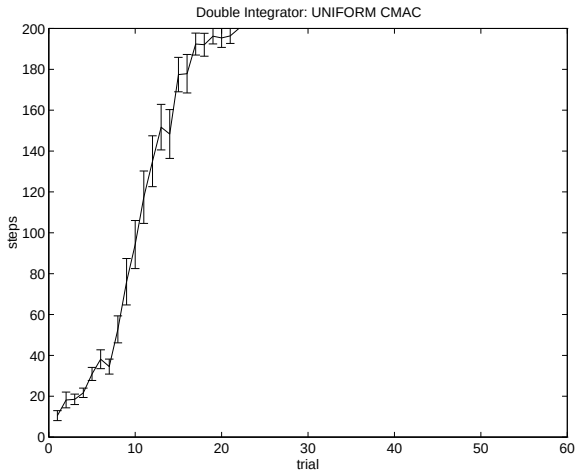

Figure 12: Average steps per trial for uniform CMAC.

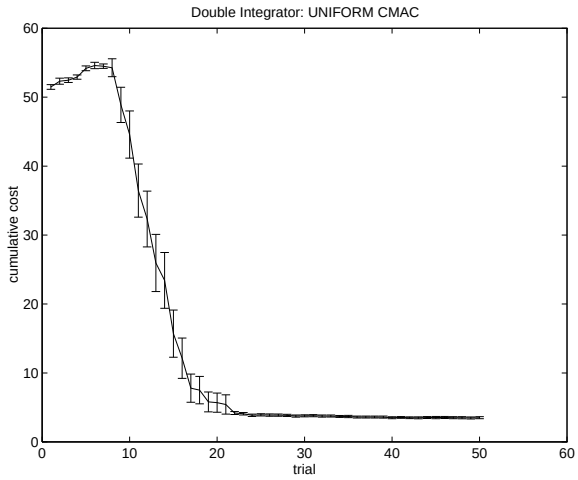

Figure 13: Average cumulative cost per trial for uniform CMAC.

matrix). The learning algorithm and the free constants were the same as with uniform CMAC. Table 2 summarizes the agent's design.

Figures 15 and 16 show the average number of time steps and accumulated cost per trial respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

5.1.4 Uniform Instance-Based

In this experiment the agent used a instance-based representation of the Q-function. Each case represents a point in the state-action space and holds its associated value. The density threshold and the smoothing parameters where set to $\tau_d = 0.06$ and $\tau_k = 0.06$ respectively. The similarity metric was the Euclidean distance and the kernel function was the Gaussian. This produces cases with spherical receptive fields and blending of the Q-function using a small number of cases. The SARSA algorithm in Figure 2 was used to update the values of each case using the eligibility traces indicated by Equation 16. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.8$, $\alpha = 0.5$, and $\epsilon = 0$. The one-step search was performed using 25 equally spaced values for the acceleration between the minimum value, $a_{\min} = -1$, and the maximum value, $a_{\max} = +1$ (i.e., $25\Delta_u = a_{\max} - a_{\min}$). Table 3 summarizes the agent's design.

Figures 17 and 18 show the average number of time steps and accumulated cost per trial respectively. The error bars represent the ± 1 the standard deviations of the mean across 36 replications.

5.1.5 Non-uniform Instance-Based

The agent used the same instance-based configuration as in the previous experiment but using the skewed version of the three variables of interest. The skewing functions are the same as the one used for the non-uniform CMAC (i.e., $p^s = \sqrt[3]{p}$, $v^s = \sqrt{v}$, and $a^s = \sqrt{a}$). The effect of using the instance-based function approximator on the skewed state-action space is similar to having non-constant density and smoothing parameters. Thus, the density of cases increases near the origin because that region gets expanded by the skewing function. Similarly, the smoothing parameter

Table 2: Summary of the design of the agent using non-uniform CMAC function approximator for the double integrator.

Factor	Description
Variables	3: position: $p \in [-1\ 1]$; 12 intervals velocity: $v \in [-1\ 1]$; 12 intervals acceleration: $a \in [-1\ 1]$; 12 intervals
Tilings	36: 12 based on p, v, and a; uniformly offseted. 12 based on p and v; uniformly offseted. 6 based on p; uniformly offseted. 6 based on v; uniformly offseted.
Skewing functions	$p^s = \sqrt[3]{p}$ $v^s = \sqrt{v}$ $a^s = \sqrt{a}$
One-step search	ϵ-greedy policy (Figure 6): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Learning	SARSA algorithm (Figure 7): $\gamma = 0.99$ $\alpha = 0.1$ $\lambda = 0.7$

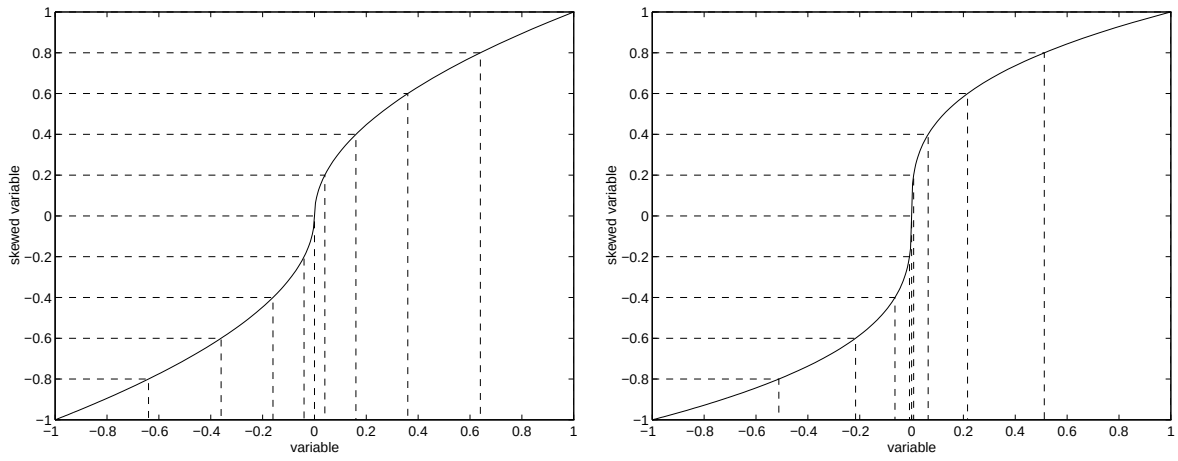

Figure 14: Skewing functions.
Left: skewing function $s(x) = \sqrt{x}$. Right: skewing function $s(x) = \sqrt[3]{x}$. Uniform intervals of the skewed variable (vertical axis) correspond to non-uniform intervals for the variable (horizontal axis). The expansion near the origin is stronger as the value of k increases.

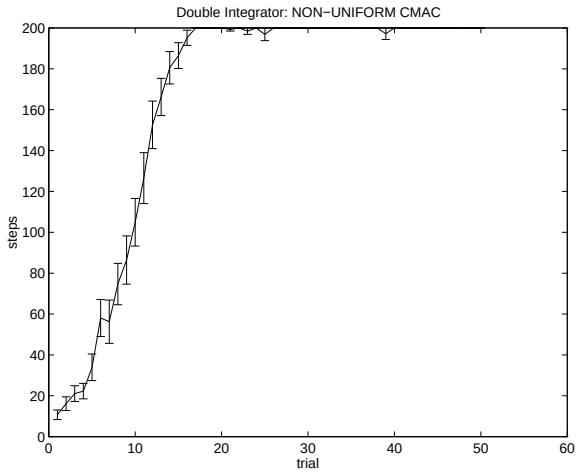

Figure 15: Average steps per trial for non-uniform CMAC.

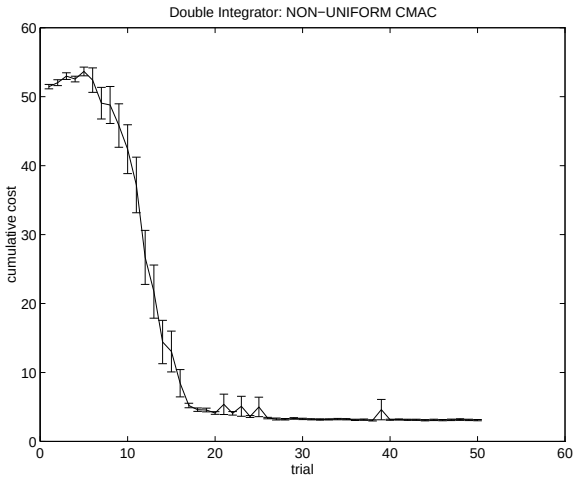

Figure 16: Average cumulative cost per trial for non-uniform CMAC.

Table 3: Summary of the design of the agent using uniform instance-based function approximator for the double integrator.

Factor	Description
Variables	3: position: $p \in [-1\ 1]$ velocity: $v \in [-1\ 1]$ acceleration: $a \in [-1\ 1]$
Distance function	Euclidean: $d(x_i, x_q) = \sqrt{(p_i - p_q)^2 + (v_i - v_q)^2 + (a_i - q_q)^2}$ density threshold: $\tau_d = 0.06$
Kernel function	Gaussian: $K(d_i) = \exp\,(\frac{d_i}{\tau_k})^2$ smoothing parameter: $\tau_k = 0.06$
One-step search	ϵ-greedy policy (Figure 1): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Learning	SARSA algorithm (Figure 2): $\gamma = 0.99$ $\alpha = 0.5$ $\lambda = 0.8$

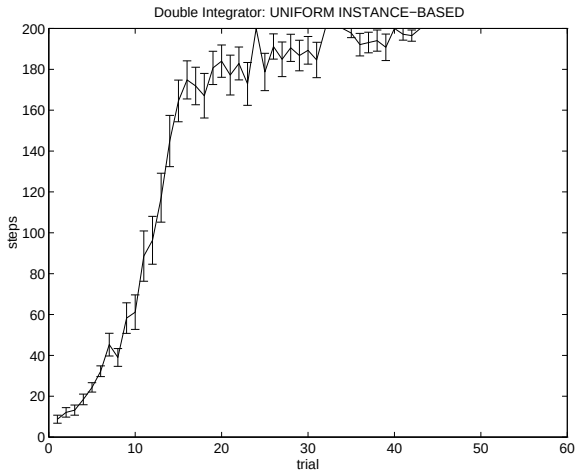

Figure 17: Average steps per trial for uniform instance-based.

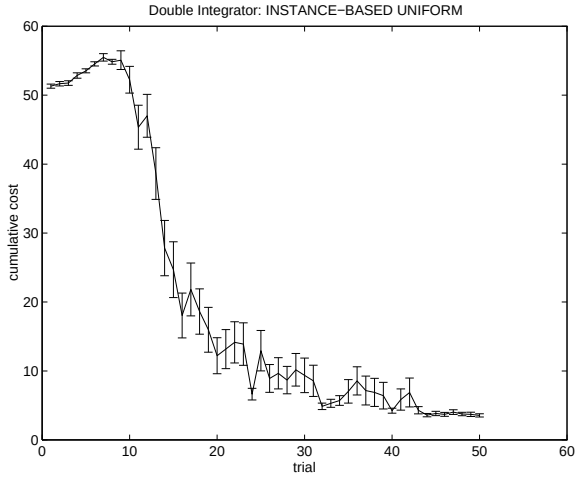

Figure 18: Average cumulative cost per trial for uniform instance-based.

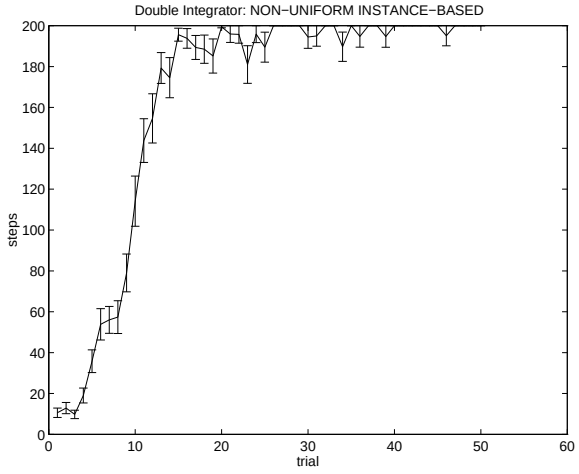

Figure 19: Average steps per trial for non-uniform instance-based.

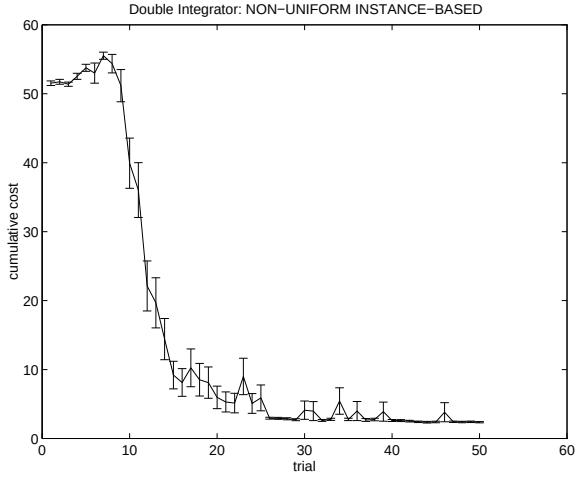

Figure 20: Average cumulative cost per trial for non-uniform instance-based.

adjusts accordingly so that the increased density of cases near the origin does not produce too much blending of their values. Table 4 summarizes the agent's design.

Figures 19 and 20 show the average number of time steps and accumulated cost per trial respectively. The error bars represent the ± 1 the standard deviations of the mean across 36 replications.

5.2 Pendulum Swing Up

The pendulum is a non-linear dynamics system with a bidimensional state. It represents a single bar held by one extremum and that can swing in a vertical plane. The bar is actuated by a motor that applies a torque at the hanging point (see Figure21). The state of the system consists of the current angle, θ, and angular velocity ω of the pendulum, or $\mathbf{x} = [\theta\ \omega]^T$ in vectorial representation. The action is the angular acceleration, α, applied to the system, or $\mathbf{u} = [\alpha]$ in vectorial representation. The objective is to move the pendulum from its rest position (i.e., hanging down with no velocity)

Table 4: Summary of the design of the agent using non-uniform instance-based function approximator for the double integrator.

Factor	Description
Variables	3: position: $p \in [-1\ 1]$ velocity: $v \in [-1\ 1]$ acceleration: $a \in [-1\ 1]$
Distance function	Euclidean: $d(x_i, x_q) = \sqrt{(p_i - p_q)^2 + (v_i - v_q)^2 + (a_i - q_q)^2}$ density threshold: $\tau_d = 0.06$
Kernel function	Gaussian: $K(d_i) = \exp\,(\frac{d_i}{\tau_k})^2$ smoothing parameter: $\tau_k = 0.06$
Skewing functions	$p^s = \sqrt[3]{p}$ $v^s = \sqrt{v}$ $a^s = \sqrt{a}$
One-step search	ϵ-greedy policy (Figure 6): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Learning	SARSA algorithm (Figure 7): $\gamma = 0.99$ $\alpha = 0.5$ $\lambda = 0.8$

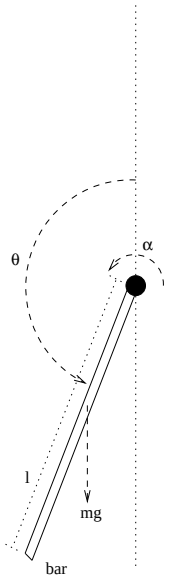

Figure 21: Pendulum. A bar hanging from one extremum and subject to gravity and the torque applied by a motor.

to the origin of the state space (i.e., bar facing up with no velocity, $\mathbf{x_d} = [0\ 0]^T$) such that the sum of the rewards is maximized. The one-step reward function is of the same class used for the double integrator. That is a negative quadratic function of the difference between the current and desired angular position and angular velocity,[10] and the angular acceleration applied, $r_{t+1} = -((\Delta_{2\pi}\theta_t)^2 + \omega^2 + \alpha_t^2)$ or in vectorial representation, $r_{t+1} = -(\Delta\mathbf{x}_t^T Q \Delta\mathbf{x}_t + \mathbf{u}_t^T R \mathbf{u}_t)$, where Q and R are the positive definite 2×2 and 1×1 matrices respectively defined as $Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $R = [1]$.

The dynamics of the pendulum is given by the following equations,

$$\begin{aligned} \frac{d\omega}{dt} &= \frac{3}{4}\frac{1}{ml^2}(\alpha + mlg\sin(\theta)) \\ \frac{d\theta}{dt} &= \omega \end{aligned} \tag{30}$$

where $m = 1/3$ and $l = 3/2$ are the mass and length of the bar respectively, and $g = 9.8$ is the gravity. The angular acceleration is bounded to be in the range between the minimum and maximum angular acceleration values (i.e., $\alpha \in [\alpha_{\min}\ \alpha_{\max}]$, where $\alpha_{\min} = -3$ and $\alpha_{\max} = 3$). This system is more difficult than the ones with linear dynamics. Unlike the double integrator, no closed-form analytical solution exist for the optimal solution and complex numerical methods are required to compute it. Moreover, the maximum and minimum angular acceleration values are not strong enough to move the pendulum straight up from the starting state without first creating angular momentum. Thus, the optimal solution sometimes requires to apply an action that moves the system in a direction oppositive to the goal state first in order to build up enough momentum to be able to swing the bar up to the top. Additionally, the equilibrium point at the goal state is unstable, which means that the agent not only needs to manage to swing up the bar but actively balance it at the goal position as well.

[10] The difference between actual and desired angle positions must be performed using modulo 2π and is denoted by $\Delta_{2\pi}\theta$.

Table 5: Summary of the design of the agent using uniform CMAC for the function approximator for the pendulum swing up.

Factor	Description
Variables	3: position: $\theta \in [-\pi\ \pi]$; 12 intervals velocity: $\omega \in [-2\pi\ 2\pi]$; 12 intervals acceleration: $\alpha \in [-3\ 3]$; 12 intervals
Tilings	36: 12 based on θ, ω, and α; uniformly offseted. 12 based on θ and ω; uniformly offseted. 6 based on θ; uniformly offseted. 6 based on ω; uniformly offseted.
One-step search	ϵ-greedy policy (Figure 1): $\epsilon = 0$ $\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$
Learning	SARSA algorithm (Figure 2): $\gamma = 0.99$ $\alpha = 0.1$ $\lambda = 0.5$

As in the double integrator, the simulation used a time step of $\Delta_t = 0.05$ seconds and new control actions were selected every four time steps. A trial consists of starting the system at position $\theta = -\pi$ with angular velocity $\omega = 0$ and running the system until 200 decision steps has elapsed (i.e. 40 simulated seconds) or the angular velocity of the pendulum gets out of bounds (i.e., $\omega > 2\pi$). Thus, unlike the double integrator, the angle of the link can wrap around without causing the trial to terminate. An experiment consists of 36 replications of 100 trials each and measuring the number of time steps and cumulative cost (i.e., negative sum of rewards) for each replication after each trial. The average number of time steps and cumulative cost across replications is used as a measure of performance.

5.2.1 Uniform CMAC

As in the double integrator configuration, the agent used a uniform CMAC to approximate the Q-function. The CMAC consisted of 36 tilings using the three state-action variables arranged in the same configuration as with the double integrator: 12 intervals along each dimension, 12 tiling depending on all three variables, 12 other tilings depended on position and velocity pairs only, and the remaining 12 tilings depended on one dimension only (6 for position and 6 for velocity). The tilings sharing the same dimensions were uniformly offseted across each dimension. The SARSA algorithm was used for updating the weights. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.5$, $\alpha = 0.1$, and $\epsilon = 0$. The one-step search was performed using 24 equally spaced values for the angular acceleration between minimum value, $\alpha_{\min} = -3$, and the maximum value, $\alpha_{\max} = +3$ (i.e., $24\Delta_u = \alpha_{\max} - \alpha_{\min}$). Table 5 summarizes the agent's design.

Figures 22 and 23 show the average number of time steps and accumulated cost per trial respec-

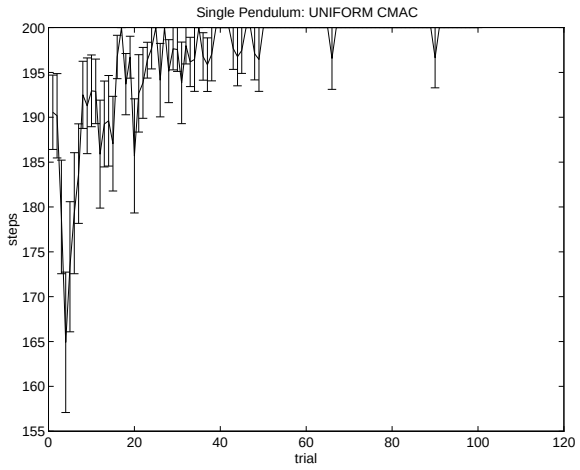

Figure 22: Average steps per trial for uniform CMAC.

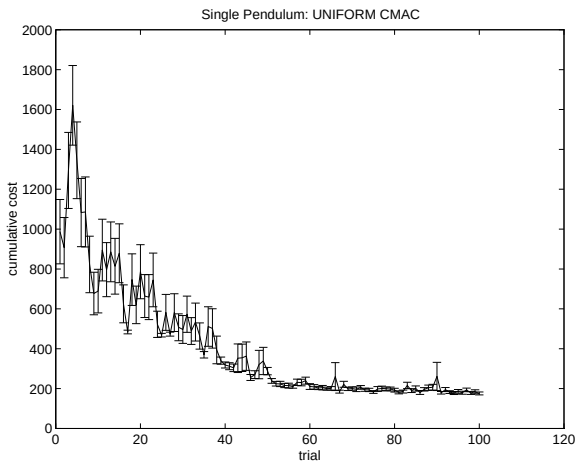

Figure 23: Average cumulative cost per trial for uniform CMAC.

tively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

5.2.2 Non-uniform CMAC

In this experiment, the agent used a non-uniform CMAC with the same configuration as in the previous experiment. The skewing functions for each of the three variables were as follows: $\theta^s = \sqrt{\theta}$, $\omega^s = \sqrt{\omega}$, and $\alpha^s = \sqrt{\alpha}$. As in the double integrator, the skewing functions expands the regions near the origin so that the CMAC can use more tiles to represent the value for the Q-function with more resolution. We chose this design because we know the system will spend more time balancing the bar once it is up and this corresponds to the region near the origin. Unlike in the double integrator, the same skewing function is used for the position and velocity because the one-step reward function depends equally on both variables (refer to the Q matrix). The learning algorithm and the free constants were the same as with the uniform CMAC. Table 6 summarizes the agent's design.

Figure 24 and 25 show the average number of time steps and accumulated cost per trial across 36 replications. The error bars represent the ± 1 standard deviations of the mean across replications.

5.2.3 Uniform Case-Based

In this experiment the agent used a case-based representation of the Q-function. We performed several experiments using different configurations of the instance-based function approximator but the agent was not able to successfully learn to maintain the bar balanced once it reached the goal. It appears that the instance-based function approximator does not have enough resolution to effectively represent the value function due to the complexity of the non-linear dynamics of the pendulum. The case-based function approximator uses more resources and has better resolution than the instance-based. Each case in the case-based function approximator represents a point in the state space and the value of 6 equally spaced actions in the interval $[\alpha_{\min}\ \alpha_{\max}]$. The blending factor was set to $\rho = 0.6$, which means that during the Q-value computations each case contributes with 40% according to the Q-value associated to the state and 60% according to the Q-values associated with the actions. The density threshold was set to $\tau_d = 0.05$. The smoothing parameters for the state

Table 6: Summary of the design of the agent using non-uniform CMAC for the function approximator for the pendulum swing up.

Factor	Description
Variables	3: position: $\theta \in [-\pi\ \pi]$; 12 intervals velocity: $\omega \in [-2\pi\ 2\pi]$; 12 intervals acceleration: $\alpha \in [-3\ 3]$; 12 intervals
Tilings	36: 12 based on θ, ω, and α; uniformly offseted. 12 based on θ and ω; uniformly offseted. 6 based on θ; uniformly offseted. 6 based on ω; uniformly offseted.
Skewing functions	$\theta^s = \sqrt{\theta}$ $\omega^s = \sqrt{\omega}$ $\alpha^s = \sqrt{\alpha}$
One-step search	ϵ-greedy policy (Figure 6): $\epsilon = 0$ $\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$
Learning	SARSA algorithm (Figure 7): $\gamma = 0.99$ $\alpha = 0.1$ $\lambda = 0.5$

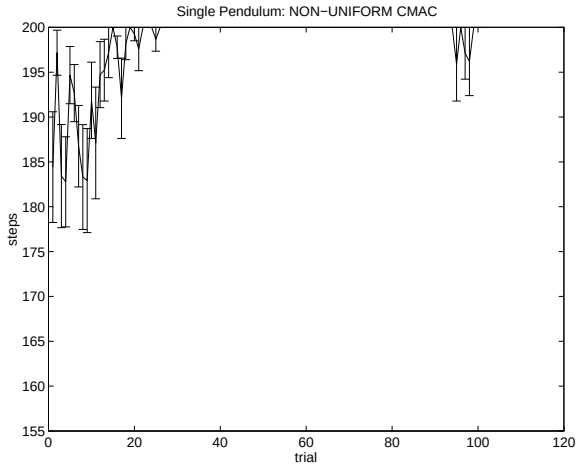

Figure 24: Average steps per trial for non-uniform CMAC.

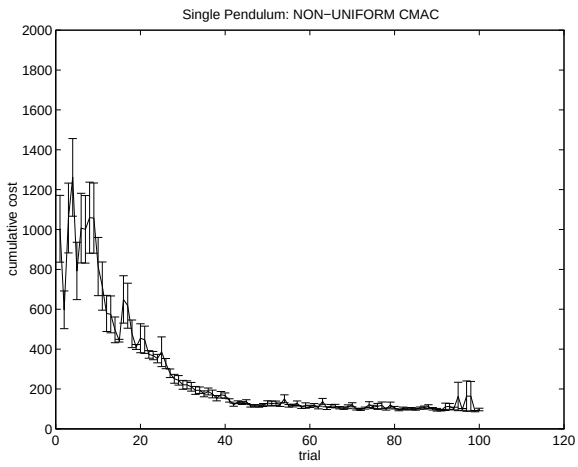

Figure 25: Average cumulative cost per trial for non-uniform CMAC.

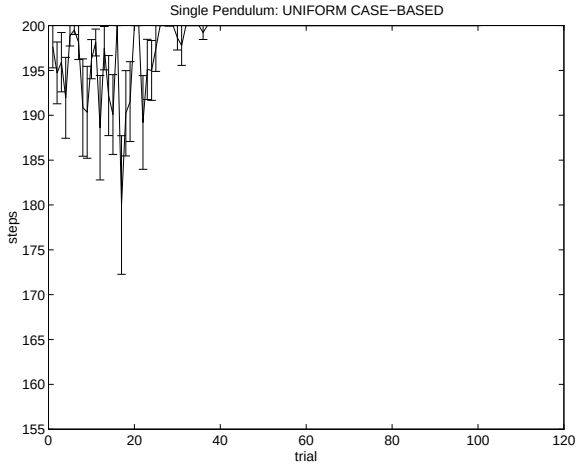

Figure 26: Average steps per trial for uniform case-based.

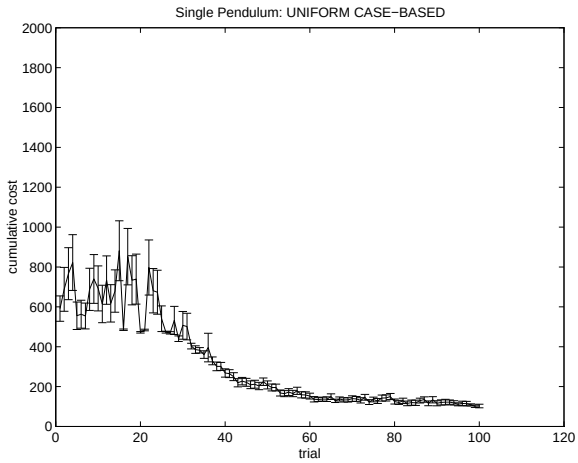

Figure 27: Average cumulative cost per trial for uniform case-based.

space and the action space were set to $\tau_k^x = 0.05$ and $\tau_k^u = 0.2$ respectively. The similarity metric for the input space was a weighted Euclidean distance and the kernel function was the Gaussian. This produces cases with elliptical receptive fields with the major axis oriented diagonally in the state space, which bias the function approximator to generalize more in regions in which the angle and the velocity of the pendulum have the same sign. This kind of bias proved to be very useful once the bar reached the goal position and the active balancing began. The similarity metric for the output space was the Euclidean distance. The SARSA algorithm in Figure 2 was used to update the values of each case using the eligibility traces indicated by Equations 22 and 23. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.7$, $\alpha = 0.4$, and $\epsilon = 0$. The one-step search was performed using 24 equally spaced values for the acceleration between the minimum value, $\alpha_{\min} = -3$, and the maximum value, $\alpha_{\max} = +3$ (i.e., $24\Delta_u = \alpha_{\max} - \alpha_{\min}$). Table 7 summarizes the agent's design.

Figures 26 and 27 show the average number of time steps and accumulated cost per trial respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

5.2.4 Non-uniform Case-Based

The agent used the same case-based configuration as in the previous experiment but using the skewed version of the three variables of interest. The skewing functions are the same as the one used for the non-uniform CMAC (i.e., $p^s = \sqrt{p}$, $v^s = \sqrt{v}$, and $a^s = \sqrt{a}$). The effect of using the case-based function approximator on the skewed state-action space is similar to having non-constant density and smoothing parameters. Thus, the density of cases increases near the origin because that region gets expanded by the skewing function. Similarly, the smoothing parameter adjusts accordingly so that the increased density of cases near the origin does not produce too much blending of their values. Table 8 summarizes the agent's design.

Figures 28 and 29 show the average number of time steps and accumulated cost per trial respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

Table 7: Summary of the design of the agent using uniform case-based function approximator for the pendulum swing up.

Factor	Description
Variables	2 inputs: position: $\theta \in [-\pi\ \pi]$ velocity: $\omega \in [-2\pi\ 2\pi]$
	1 output: acceleration: $\alpha \in [-3\ 3]$
Case structure	Number of actions: 6, equally spaced in $[-3\ 3]$ Blending factor: $\rho = 60\%$
Distance functions	Input space: Weighted Euclidean: $d_M(x_i, x_q) = \sqrt{(x_i - x_q)^T M^T M (x_i - x_q)}$ $M = \begin{bmatrix} 1 & 1 \\ -1.75 & 1.75 \end{bmatrix}$ density threshold: $\tau_d = 0.06$
	Output space: Euclidean: $d_M(u_i, u_q) = \lvert\alpha_i - \alpha_q\rvert$
Kernel functions	Input space: Gaussian: $K^x(d_i^x) = \exp\left(\frac{d_i^x}{\tau_k^x}\right)^2$ smoothing parameter: $\tau_k^x = 0.05$
	Output space: $K^u(d_i^u) = \exp\left(\frac{d_i^u}{\tau_k^u}\right)^2$ smoothing parameter: $\tau_k^u = 0.2$
One-step search	ϵ-greedy policy (Figure 1): $\epsilon = 0$ $\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$
Learning	SARSA algorithm (Figure 2): $\gamma = 0.99$ $\alpha = 0.4$ $\lambda = 0.7$

Table 8: Summary of the design of the agent using uniform case-based function approximator for the pendulum swing up.

Factor	Description
Variables	2 inputs: position: $\theta \in [-\pi\ \pi]$ velocity: $\omega \in [-2\pi\ 2\pi]$
	1 output: acceleration: $\alpha \in [-3\ 3]$
Case structure	Number of actions: 6, equally spaced in $[-3\ 3]$ Blending factor: $\rho = 60\%$
Distance functions	Input space: Weighted Euclidean: $d_M(x_i, x_q) = \sqrt{(x_i - x_q)^T M^T M (x_i - x_q)}$ $M = \begin{bmatrix} 1 & 1 \\ -1.75 & 1.75 \end{bmatrix}$ density threshold: $\tau_d = 0.06$
	Output space: Euclidean: $d_M(u_i, u_q) = \lvert\alpha_i - \alpha_q\rvert$
Kernel functions	Input space: Gaussian: $K^x(d_i^x) = \exp\left(\frac{d_i^x}{\tau_k^x}\right)^2$ smoothing parameter: $\tau_k^x = 0.05$
	Output space: $K^u(d_i^u) = \exp\left(\frac{d_i^u}{\tau_k^u}\right)^2$ smoothing parameter: $\tau_k^u = 0.2$
Skewing functions	$\theta^s = \sqrt{\theta}$ $\omega^s = \sqrt{\omega}$ $\alpha^s = \sqrt{\alpha}$
One-step search	ϵ-greedy policy (Figure 6): $\epsilon = 0$ $\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$
Learning	SARSA algorithm (Figure 7): $\gamma = 0.99$ $\alpha = 0.4$ $\lambda = 0.7$

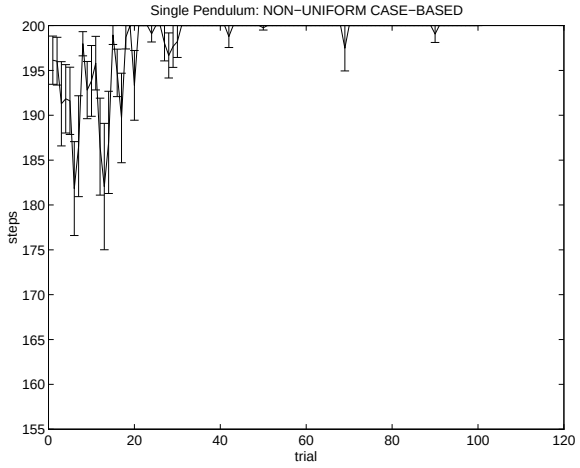

Figure 28: Average steps per trial for non-uniform case-based.

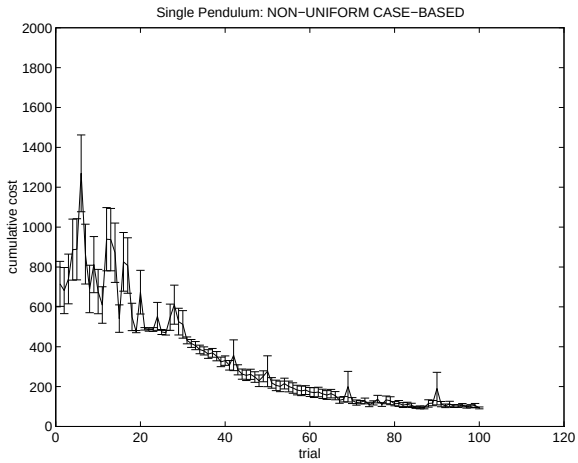

Figure 29: Average cumulative cost per trial for non-uniform case-based.

6 Discussion

This section discusses in detail the results obtained in the double integrator and the swing up pendulum problems.

6.1 Double Integrator

Table 9 shows the average and standard deviation of the cumulative cost and steps for CMAC and instance-based function approximators at trial 50.

The effect of preallocating resources through the use of skewing functions is clear with both, CMAC and instance-based function approximators. In the CMAC, there is statistical evidence at the 2% confidence level at trial 50 that the cumulative cost achieved by the agent using the non-uniform version is smaller than the one achieved with the uniform version ($t_0 = 2.429$, P-value $= 0.018$).[11] Additionally, the learning performance of the agent using non-uniform CMAC stabilized faster (approximately trial 17) than one using the uniform CMAC (approximately trial 22) (see Figures 13 and 16). Similarly, the first time the agent was able to stay in-bounds for the whole trial occurred at trial 17 when using the non-uniform CMAC. The same event occured at trial 22 for the agent using the uniform CMAC (see Figures 12 and 15). Also, after 50 trials, the agent using

[11] All test of hypotheses assume unknown and different variances for $\bar{X}_1$ and $\bar{X}_2$. The statistic

$$t_0^* = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

can be used for testing $H_0 : \mu_1 = \mu_2$ in this case and it is distributed approximately as t with degrees of freedom given by

$$\nu = \frac{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^2}{\frac{(S_1^2/n_1)^2}{n_1+1} + \frac{(S_2^2/n_2)^2}{n_2+1}} - 2$$

if the null hypothesis is true.

the non-uniform CMAC was more consistent than the agent using uniform CMAC because of the smaller variance on the performance (non-uniform: $\hat{\sigma} = 0.0909$, uniform: $\hat{\sigma} = 0.1485$). Summarizing, the main effects of preallocating CMAC tiles more heavily near the origin of the state-action space are improvement in learning performance, faster learning, and better consistency. This is expected because the system spends most of the time around the origin trying to maximize the rewards. This requires very small adjustments using small changes in acceleration and the ability to represent the Q-function with enough resolution in that region of the state space. The skewing functions used to represent the non-uniform CMAC were designed for that purpose.

The effect of resource preallocation is also significant with the agent using the instance-based function approximator. At trial 50, there is statistical evidence at the 1% confidence level to reject the hypothesis that both versions, uniform and non-uniform, perform similarly ($t_0 = 4.653$, P-value $= 0.000$). Additionally, there is an improvement in the learning performance achieved with the agent using the non-uniform version over the agent using the uniform one because the former was able to stabilize the performance around trial 26 while the later took until trial 43 (see Figures 18 and 20). Also, the agent using the non-uniform version learned to stay in-bounds for the whole trial at trail 21. The same event occurs at trial 26 when using the uniform version (see Figures 17 and 19). As with the CMAC version, there is a significant difference in the consistency of the performance since the variance of the non-uniform version is smaller than the variance of the uniform version (non-uniform: $\hat{\sigma} = 0.0944$, uniform: $\hat{\sigma} = 0.2375$). In summary, the effect of resource preallocation in the instance-based function approximator is similar than the one observed in the CMAC function approximator. Although the instance-based version allocates resources dynamically by creating new cases when exploring new regions of the state-action space, the effect of the skewing functions appears to show more heavily than the advantage of dynamically allocating resources in this task.

There appears to be no statistical significant difference between the performance achieved with the uniform versions of the CMAC and instance-based function approximators ($t_0 = -0.161$, P-value $= 0.872$). However, there is a statistical significant difference between the performance achieved with the non-uniform versions; the instance-based version appears to be an improvement over the CMAC ($t_0 = 5.502$, P-value $= 0.000$). This difference may be dependent on the design decisions involved in both types of function approximators and the sensibility of such decisions on the learning behavior of the agent; further research is needed into this issue. Additionally, contrasting the performance curves of the agents using CMACs with the agents using the instance-based function approximator, it appears that the agents using the former are more consistent than the ones using the later. The increased consistency achieved with the CMAC is, presumably, due to its capacity to smoothly generalize values to new regions of the state-action space. This does not occur with the instance-based version because there is an abrupt change or "peak" in the value function every time a new case is added to the library.

6.2 Pendulum Swing Up

Table 10 shows the average and standard deviation of the cumulative cost and steps for CMAC and case-based function approximators at trial 100. The agent using the instance-based function approximator was not able to successfully learn to maintain the bar balanced once it reached the goal.

The effect of preallocating resources through the use of skewing functions shows more significantly in the CMAC than the case-based function approximators. In the CMAC, there is statistical

Table 9: Summary of results for the double integrator at trial 50.

Function approximator	Skewing	Cumulative cost [a]		Steps[b]	
		mean	std. dev.	mean	std. dev
Optimal	N/A	1.4293	0	200	0
CMAC	uniform	3.5100	0.1485	200	0
	non-uniform	3.0870	0.0909	200	0
Instance-Based	uniform	3.5552	0.2375	200	0
	non-uniform	2.3660	0.0944	200	0

[a] Cumulative cost is the sum of all the immediate cost the agent received at every step during the trial. The lower values represent better solutions.

[b] Steps is the number of steps the agent took during the trial. The mean value 200 with 0 standard deviation means that the agent completed all 36 replications successfully.

evidence at trial 100 that the cumulative cost achieved with the non-uniform version is better than the one achieved with the uniform version ($t_0 = 8.079$, P-value $= 0.000$). Additionally, the learning performance of the agent using non-uniform CMAC stabilized faster (approximately at trial 41) than the one using uniform CMAC (approximately at trial 52) (see Figures 23 and 25). Similarly, the first time the agent was able to stay in-bounds for the whole trial across the 36 replications occured at trial 35 for the agent using the non-uniform CMAC. The same event occured at trial 50 for the agent using non-uniform CMAC (see Figures 12 and 15). At trial 100, the agents using the uniform and non-uniform CMAC achieved similar level of consistency across replication (uniform: $\hat{\sigma} = 7.3259$, non-uniform: $\hat{\sigma} = 6.3807$). Summarizing, the main effects of preallocating CMAC tiles more heavily near the origin of the state-action space are improvement on learning performance and faster learning. These results are similar to the ones achieved in the double integrator problem even though the pendulum swing up problem is much more difficult. The improvement in performance in the cumulative cost from 175.79.26 for the agent using uniform CMAC to 97.3036 for the agent using non-uniform CMAC reveals the importance using a function approximator with higher resolution in the region near the goal state. The pendulum is highly unstable at this region and the agent is required to perform very small adjustments using small changes in acceleration in order to keep the bar balanced. Moreover, any mistake that causes the agent to loose control in the balance of the bar produces a lot of cost because it makes the bar to fall and oscillate.

The effect of resource preallocation in the case-based function approximators does not show as clearly as with the CMAC function approximators. At trial 100, there is not enough statistical evidence to reject the hypothesis that both versions, uniform and non-uniform, perform similarly ($t_0 = -0.945$, P-value $= 0.349$). Also, there is not noticeable difference in the learning performance in both versions since they both take around 60 trials to stabilize (see Figures 27 and 29) and 35 trials to learn to stay in-bounds (see Figures 26 and 28). There is, however, a significant difference in the consistency of the performance since at trial 100 the variance of the non-uniform version is smaller than the variance of the uniform version (non-uniform: $\hat{\sigma} = 4.7070$, uniform: $\hat{\sigma} = 9.1446$). However, the agent using the uniform version is able to stay in-bounds across all 36 replications after trial 37 whereas the agent using non-uniform version has some replications that achieve less than 200 steps after trial 37 (see Figures 26 and 28). Presumably, this is because the skewing functions focus the resources in excess in the region near the goal state, which produces insufficient generalization

Table 10: Summary of results for the pendulum swing up at trial 100.

Function approximator	Skewing	Cumulative cost [a]		Steps[b]	
		mean	std. dev.	mean	std. dev
CMAC	uniform	175.7926	7.3259	200	0
	non-uniform	97.3036	6.3807	200	0
Instance-Based	uniform	N/A	N/A	N/A	N/A
	non-uniform	N/A	N/A	N/A	N/A
Case-Based	uniform	102.7438	9.1446	200	0
	non-uniform	93.0292	4.7070	200	0

[a] Cumulative cost is the sum of all the immediate cost the agent received at every step during the trial. The lower values represent better solutions.

[b] Steps is the number of steps the agent took during the trial. The mean value 200 with 0 standard deviation means that the agent completed all 36 replications successfully.

for the agent to regain control of the bar once it looses the balance and force the system to go out-of-bounds. In summary, the effect of resource preallocation in the case-based function approximator is less notorious than in the CMAC function approximator. This is as expected because the case-based version allocates resources dynamically by creating a new case when exploring new regions of the state-action space. Thus, this type of function approximator is less sensitives to the effects of skewed variables of the state-action space.

There appears to be no strong differences in the performance at trial 100 of the agents using the non-uniform CMAC, uniform case-based, and non-uniform case-based function approximators. The cumulative cost achieved by the agents using these three types of function approximators is around 100 units whereas the one achieved by the agent using the uniform CMAC is around 175 units (see Table 10). In the context of learning rate and consistency of performance across replications, the agents using the case-based versions appear to take a few more trials to stabilize the performance and the agent using the CMAC version appears to incur in greater costs and less consistency at the earlier trials of the learning curve. However, these differences are not as drastic as the ones obtained when contrasting the performance of these agents with the performance of the agent using the uniform CMAC. In summary, the learning performance achieved with the agents using the case-based versions is as good as the learning performance achieved with the one using the non-uniform CMAC. The effect of resource preallocation shows more clearly in the agents using CMACs because that type of function approximator cannot allocate resources dynamically in regions of the state-action space that require them the most.

6.3 Summary

Two different types of function approximators, CMAC and memory-based, were tested in the double integrator and the pendulum swing up problems. Both problems are characterized by dynamical systems having continuous state and action spaces and the results obtained in these experiments support the feasibility of using a function approximator to represent the Q-function in the state-action space and generalizing the value of individual experiences across states and actions.

Additionally, the results obtained demonstrate the feasibility of the modular implementation of non-uniform function approximators using skewing functions. The proposed approach appears to benefit the efficiency of function approximators that use static resource allocation such as the CMAC. On the other hand, the proposed approach does not appears to affect the efficiency of function approximators that use dynamic resource allocation such as the case-based. In particular, the effect of non-uniform resource preallocation resulted in faster learning, better performance, and improved consistency. These results showed more significantly in the pendulum swing up than in the double integrator presumably because it is more difficult to actively balance the pendulum than the car at the goal state. Thus, the effect of a function approximator with high resolution at the region containing the goal state shows more significantly in the pendulum swing up than the double integrator.

Two memory-based function approximator were evaluated: instance-based and case-based. The instance-based function approximator uses only the state-action pairs the agent has actually experienced to represent the Q-function and new state-action pairs are incorporated when the distance to the nearest neighbor as measured by some metric function is larger than the density threshold. The case-based function approximator uses memory elements more complex than simply state-action pairs. Every case represents a state the agent has visited in the past and a set of actions along with their associated Q-values. The actions within a case are actions that the agent may or may have not experienced before. Nevertheless, this enhanced representation better support credit assignment and generalization because the agent is able to update the associated Q-values of actions it has never experienced before based on state-action pairs that are similar according to the distance function. The case-based function approximator is able to solve problems that the instance-based function approximator may not solve as is the case of the pendulum swing up.

7 Conclusions

In this research, we explored two important issues related to reinforcement learning applied to systems described by continuous state and action spaces. The first question asked about possible ways to generalize the outcome of experiences involving states and actions in continuous spaces. We described some of the techniques researchers have used to represent the value function and proposed an alternative method to represent and generalize the value of states-actions pairs using the outcome of individual experiences. The idea consists of using only one function approximator to represent the Q-function on the combined state-action space and use the already familiar SARSA learning algorithm to adapt the parameters of the function approximator using TD(λ) backups. The disadvantage of this approach compared to other simpler ones is that the one-step search is computationally expensive. However, more efficient implementations of the one-step search may be possible by exploiting the class of function approximator being used. Additionally, there is no theoretical guarantee that this approach will converge to the correct Q-function (the function approximator may not even be able to represent the exact value function across the entire state-action space), but the results obtained on two different kinds of sparse coarse-coded function approximators on two different classes of continuous systems are encouraging. These results support the idea of using function approximators to solve reinforcement learning problems in systems with continuous state and actions. Function approximators are able to generalize the value for unseen regions of the state-action space based on the estimates of other regions of the state-action space. Also, they provide a compact representation for the Q-function and can be trained incrementally as the agent collects more data.

The second question asked about the effect of designing function approximators in which resource are preallocated across the state-action space. This is an important design decision because it directly affects the resolution capabilities of the function approximator in different regions of the state-action space. Usually, function approximators that uniformly distribute resources are more efficient and easier to implement than their non-uniform counterparts. We hypothesized that the need for non-uniform function approximators is due to the better use of the resources that can be obtained by increasing the resolution of the function approximator in important regions of the state-action space. The results obtained tend to support this hypothesis. In addition, we presented a technique that combines simplicity and efficiency of uniform function approximators with the benefits of the non-uniform ones. In this technique, a skewing function is used to transform the original state-action space into a deformed version, which is then used as the domain of the uniform function approximator. The skewing function is designed in advance to expand important regions of the state-action space so that the function approximator can use more resources to estimate the corresponding Q-values with more resolution. The use of the skewing function adds little computational complexity to procedure.

We also explored the feasibility of using memory-based function approximators in reinforcement learning problems with continuous state and action spaces. Memory-based function approximators belong to the class of sparse coarse-coded function approximators and can dynamically allocate resources with experiences. The results obtained in this research showed that the memory-based function approximator was less sensible to preallocation of resources than the CMAC function approximator. We conjecture that this effect may be due to the capability of allocating more resources in those regions of the state-action space that are used more often, which allows them to dynamically adjust the resolution in those regions. This may serve as an advantage when designers do not know in advance which regions of the state-action space the function approximator should represent with higher resolution.

In the experiments presented in this paper, we used predesigned skewing functions that remained constant throughout the experiment. However, it is also possible to use adaptive skewing functions so that the agent can dynamically increase the resolution of the function approximator as it collects more data. Future research will address this issue.

References

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

[2] C. G. Atkeson. Memory-based learning control. In *Proceedings of the 1991 American Control Conference*, volume 3, pages 2131–2136, Boston, MA, 1991.

[3] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.

[4] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[5] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA, 1995.

[6] P. Cishosz. Truncating temporal differences: on the efficient implementation of td(λ) for reinforcement learning. *Journal of Artificial Intelligence Research*, 2:287–318, 1996.

[7] P. Kanerva. Sparse distributed memory and related models. In M. H. Hassoun, editor, *Associative Neural Memories: Theory and Implementation*, chapter 3. Oxford University Press, New York, NY, 1993.

[8] D. Kibler and D. W. Aha. Instance-based prediction of real-valued attributes. *Computational Intelligence*, 5(2):51–57, 1989.

[9] L. J. Lin. Self-improving reactive agents based on reinforcement learning. *Machine Learning*, 8(3-4):293–321, 1992.

[10] S. Mahadevan and J. Connell. Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Proceedings of the Eight International Workshop on Machine Learning*, volume 1, pages 328–332. Morgan Kaufmann, 1991.

[11] R. A. McCallum, G. Tesauro, D. Touretzky, and T. Leen. Instance-based state identification for reinforcement learning. *Advances in Neural Information Processing Systems*, 7, 1995.

[12] A. W. Moore and C. G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233, 1995.

[13] K. S. Narendra and A. M. Annaswamy. *Stable Adaptive Systems.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[14] J. Peng. *Efficient Dynamic Programming-Based Learning for Control.* PhD thesis, Department of Computer Science, Northeastern University, 1993.

[15] J. Peng and R. J. Williams. Incremental multi-step q-learning. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 189–195, Aberdeen, Scotland, 1994.

[16] A. Ram and J. C. Santamaría. Continuous case-based reasoning. *Artificial Intelligence*, 90(1-2):25–77, 1997.

[17] R. J. Richards. *An Introduction to Dynamics and Control.* Longman, New York, NY, 1979.

[18] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical Report CUED/F-INFEG/TR66, Cambridge University Department, 1994.

[19] K. S. Shanmugam. *Digital and Analog Communication Systems.* Longman, New York, NY, 1979.

[20] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

[21] R. F. Stengel. *Optimal Control and Estimation.* Dover Publications, Mineola, NY, 1994.

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

[23] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8, 1996.

[24] C. L. Tham. Reinforcement learning of multiple tasks using a hierarchical cmac architecture. *Robotics and Autonomous Systems*, 15(4):247–274, 1995.

[25] J. N. Tsitsiklis and B. Van Roy. Analysis of temporal-difference learning with function approximation. *Advances in Neural Information Processing Systems*, 9, 1996.

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