Proceedings of the 2007 IEEE Symposium on Approximate
Dynamic Programming and Reinforcement Learning (ADPRL 2007)

Fitted Q Iteration with CMACs

Stephan Timmer
Department of Computer Science
University of Osnabrueck
Osnabrueck, Germany
Email: stephan.timmer@uos.de

Martin Riedmiller
Department of Computer Science
University of Osnabrueck
Osnabrueck, Germany
Email: martin.riedmiller@uos.de

***Abstract*— A major issue in model-free reinforcement learning is how to efficiently exploit the data collected by an exploration strategy. This is especially important in case of continuous, high dimensional state spaces, since it is impossible to explore such spaces exhaustively. A simple but promising approach is to fix the number of state transitions which are sampled from the underlying markov decision process. For several kernel-based learning algorithms there exist convergence proofs and notable empirical results, if a fixed set of transition instances is used. In this article, we will analyze how function approximators similar to the CMAC-architecture can be combined with this idea. We will show both analytically and empirically the potential power of the CMAC architecture combined with an offline version of Q-learning.**

I. Introduction

Given a markov decision process $(T, S, U, P, r)^1$, many model-free reinforcement learning algorithms suffer from finding an appropriate data structure to store the Q-function $Q : S \times U \to \mathbb{R}$. For discrete and finite state spaces, the Q-learning algorithm ([1], [2]) is able to perfectly learn the Q-function only with a lookup-table. In this article, we consider the known problem of scaling Q-learning to continuous state spaces. Since we assume that no prior knowledge concerning the model (system dynamics) is available, the problem of learning the Q-function is twofold. First, an exploration procedure must collect examples of possible state transitions and their corresponding rewards. Then, a function approximator can be used to derive the Q-function from these examples. This work concentrates on two aspects of this procedure.

1) Since we want to learn a Q-function in reasonable time, every exploration strategy will necessarily leave out large parts of the continuous state space. An important question is therefore how to make efficient use of a small set of sampled state transitions
2) Powerful function approximators can help to compensate the problem of less data by generalization techniques. Unfortunately, many of these approximators are not able to preserve convergence of the Q-learning algorithm. Thus, it is necessary to search for approximator schemes for which convergence can be shown either analytically or at least empirically

The basic idea of this work is to combine the CMAC function approximator with the fitted Q iteration algorithm from [3]. The first problem of the enumeration above is addressed by the design of the fitted Q iteration algorithm. We propose two variants of the CMAC-architecture to solve the second problem. In the rest of the paper, we will first present an extension to the fitted Q iteration algorithm and then discuss important issues of combining function approximators with this algorithm. We will also present empirical evidence for typical reinforcement learning benchmarks.

II. Fitted Q Iteration and Related Work

Before approximating a Q-function, the fitted Q iteration algorithm employs a pure random exploration to sample transition instances. State transitions are represented by four tuples (s_t, u_t, r_t, s_{t+1}) with $s_t, s_{t+1} \in S, u_t \in U$ and $r_t = r(s_t, u_t)$. After a certain number of transition instances is stored, the set of four tuples is kept fixed and the actual inference of the Q-function starts. The inference procedure consists of several iterations of computing new approximations of the Q-function. As shown in Fig. 1, we extended the fitted Q iteration algorithm by interleaving the inference procedure with an ϵ-greedy exploration strategy. The original fitted Q iteration algorithm can be viewed as a special case of this algorithm by setting $\epsilon = 0$ and stopping the algorithm after the first iteration of the main loop. The ϵ-greedy strategy enables us to apply the fitted Q iteration algorithm on problems for which only very long sequences of state transitions lead to a goal state. In such a setting, pure random exploration would visit the goal state only very few times.

The power of the fitted Q iteration algorithm comes from its modularity. Since all updates are done offline^2, approximating the Q-function can be viewed as a separate, supervised learning problem. The question arises if there are function approximators especially suited for offline updating.

In [4], a special class of approximators called averagers is discussed. An averager builds a weighted sum of known output values from the training patterns to approximate an unknown output value. Since the sum of weights is always equal to one, averagers can be regarded as special kernel-based methods. The proposed algorithm, fitted value iteration, uses offline updating and is shown to converge for all approximators belonging to the averager class. Fitted value iteration is similar

^{1}Timesteps (T), states (S), actions (U), probabilities (P), reward function (r).

^{2}The term "offline" refers to the strict separation of sampling transition instances and updating the Q-function

```
Initialization
        • $N := 0$, choose discount rate $0 < \gamma < 1$
        • Set of four tuples $\mathcal{F} := \emptyset$
        • $\forall s \in S, u \in U : \hat{Q}_0(s,u) := 0$
Main Loop
        Exploration
        If $|\mathcal{F}| \le maxInstances$
                • Sample a set $\mathcal{F}_{new}$ of four tuples
                  $(s_t^l, u_t^l, r_t^l, s_{t+1}^l),\ l = 1, ..., |\mathcal{F}_{new}|$ by
                  an $\epsilon$-greedy exploration strategy and the
                  current approximation of the Q-function
                  $\hat{Q}_N$.
                • $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}_{new}$
        Inference Loop
        $M := 0$
        While M not exceeds a certain threshold
                • $N \leftarrow N+1, M \leftarrow M+1$
                • Build a set of training patterns from $\mathcal{F}$
                  $\mathcal{TF} := \{(x^l, y^l),\ l = 1, ..., |\mathcal{F}|\}$
                  – Inputs:
                    $x^l := (s_t^l, u_t^l)$
                  – Outputs:
                    $y^l := r_t^l + \gamma \max_{u \in U} \hat{Q}_{N-1}(s_{t+1}^l, u)$
                • Use a supervised learning method to in-
                  duce an approximation $\hat{Q}_N$ of the Q-
                  function from the set of training patterns
                  $\mathcal{TF}$.
        End of Inference Loop
End of Main Loop
```

Fig. 1. Fitted Q Iteration with ϵ-greedy exploration. After the number of stored state transitions exceeds the threshold $maxInstances$, the main loop is executed until the sequence of Q-functions converges.

to fitted Q iteration, except that the probability model and the reward function are assumed to be given in advance. Thus, it suffices to approximate the value function instead of the Q-function.

In [5], a similar approach to averagers is presented by aggregating states into clusters. A Q-learning variant is developed, which averages all rewards from states belonging to the same cluster.

In [6], both an algorithm and theoretical results for using kernels with model-free reinforcement learning are provided. Again, the proposed method approximates a Q-function by making offline updates on a previously sampled set of state transitions. In contrast to averagers as used in [4], the class of kernels considered in this article is exclusively designed for continuous state spaces, including gaussian kernels and every other function $\Phi : [0,1] \to \mathbb{R}^+$ satisfying the normalizing condition $\int_0^1 \Phi(x)dx = 1$. Beside the guaranteed convergence of the algorithm, the authors can show that the approximations will converge to the actual (optimal) Q-function, if the number of sampled state transitions grows to infinity and a *bandwidth* parameter is adjusted properly. Intuitively, the bandwidth controls the shape of the neighborhood of a kernel, much like the standard deviation of a gaussian process. To achieve this result, all sampled transition instances are assumed to be statistically independent. This may be a limitation for real learning tasks, since state transitions are typically collected while following a certain trajectory.

The fitted Q iteration algorithm as presented in [3], uses ensembles of regression trees to approximate the Q-function. These ensembles create partitions of the input space ($S \times U$) with constant output value for all elements of the partition. Since this procedure can also be formulated as a kernel-based method, the authors were able to prove convergence of the fitted Q iteration algorithm. A major difference to the work in [6] is that the algorithm scales to continuous action spaces and that a single kernel is used for all actions.

Unfortunately, for many popular function approximators different from kernel-based methods, it turns out to be difficult to prove convergence of Q-learning variants. In spite of this fact, there is much evidence that a variety of those approximators can be successfully applied to model-free reinforcement learning. For example, in [7], neural networks are used with offline updating to learn a Q-function. The LSPI algorithm from [8] also employs the idea of separating the sampling (exploration) phase from actually computing a policy. Indeed, Q-functions are only computed with respect to a fixed policy, because LSPI is a variant of approximate policy iteration. The approximator used in LSPI is an extension of the LSTD algorithm [9] which is a technique from linear regression. Instead of performing gradient descent, as it is typically done for CMACs, a solution is computed by solving a system of linear equations in the least square sense.

The power of the CMAC function approximator for reinforcement learning problems has already been demonstrated in combination with the sarsa(λ) algorithm ([10], [11]). The sarsa(λ) algorithm is similar to Q-learning in the sense that both algorithms learn a state-action value function. The major difference between sarsa(λ) and Q-learning is that sarsa(λ) is an on-policy method and includes eligibility traces. Although there is no proof of convergence for sarsa(λ) using a CMAC approximator, convergence could be shown empirically [10].

We believe that it is worth studying the CMAC-architecture in combination with offline updates of a Q-function as it is implemented in the fitted Q iteration algorithm. The training of the CMAC approximator consists of a (linear) gradient descent and therefore aims at minimizing the mean squared error (MSE) on the training patterns. Although there is no guarantee that a global minimum of the MSE is encountered, this procedure seems to be better than building average values. As mentioned above, there is empirical evidence that the CMAC works well in combination with temporal difference learning methods.

Since the Q-function is rarely linear with respect to the state (action) variables states, the state space is usually replaced by a feature space. As in [10], our implementation of the CMAC uses an intuitive feature representation by employing overlapping grids. The power of the CMAC can be increased by manually designing such a set of grids. However, the CMAC also works well with regular, low resolutional grids as we will demonstrate by our empirical results.

To discuss the potential of the CMAC, we will also present an averager version of the CMAC. We aim at establishing a convergence guarantee on the one hand, while keeping some favorable properties of the CMAC-architecture on the other hand. This will also allow a direct comparison between the CMAC-architecture and function approximators belonging to the averager class.

III. Two CMAC-Architectures

A. Basic functionality of the CMAC

To perform gradient descent, it is required to parametrize the Q-function by a vector of parameters $\vec{w}$. Every element of $\vec{w}$ corresponds to a feature from the feature vector $\vec{\phi}_x = (\phi_x(1), \phi_x(2), ..., \phi_x(n))$ which is built for every state-action pair $x = (s,u) \in S \times U$. The Q-function is represented by the scalar product of the feature vector with the vector of parameters

$$\hat{Q}(x) = \sum_{i=1}^{n} w(i)\phi_x(i)$$

What makes the CMAC special is the layout of the feature space. Binary features are used to partition the input space in several, possibly non-disjunctive sets. These sets are also called receptive fields. As in [10], we will consider receptive fields simply to be cells of a grid which is laid over the input space. As illustrated in Fig. 2, the feature space consists of several overlapping grids. Every cell of one of the grids

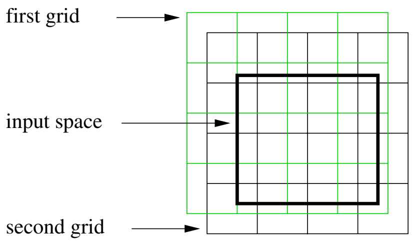

Fig. 2. Layout of the feature space. The figure is taken from [10] with minor modifications

corresponds to a binary feature $\phi(i)$, which is active for an input x ($\phi_x(i) = 1$), if and only if x lies within the cell. Otherwise, the feature is inactive ($\phi_x(i) = 0$). Since the grids are overlapping, every input lies in several cells and therefore activates several features.

To adjust the parameter vector $\vec{w}$, a set of training patterns $\mathcal{TF} = \{(x^l, y^l),\ l = 1, ..., |\mathcal{TF}|\}$ is presented to the CMAC. The mean squared errror (MSE) on the set of training patterns is defined as

$$MSE(\vec{w}) = \frac{1}{|\mathcal{TF}|} \sum_{(x^l,y^l)\in\mathcal{TF}} (y^l - \hat{Q}(x^l))^2$$

Since it holds

$$\forall i \in \{1,..,n\} : \frac{\partial \hat{Q}(x)}{\partial w(i)} = \phi_x(i),$$

the gradient of the MSE can be easily computed. To minimize the MSE, the parameter vector $\vec{w}$ is gradually updated proportional to the negative direction of the gradient of the MSE. To enlarge the set of training patterns, it is possible to duplicate individual training patterns and therefore using every pattern several times. The update of the parameter vector $\vec{w}$ is usually done for every single training pattern.

$$\vec{w}' \;=\; \vec{w} + \alpha(y^l - \hat{Q}(x^l))\vec{\phi}_{x^l} \qquad (1)$$

Here, $\vec{w}'$ denotes the new parameter vector which replaces the old vector $\vec{w}$ immediately after the update. The learning rate $0 < \alpha < 1$ is a positive step-size parameter. Since binary features are used, the update rule (1) can be simplified to

$$w(i)' = \begin{cases} w(i) + \alpha y^l - \alpha\hat{Q}(x^l), & \text{for } \phi_{x^l}(i) = 1 \quad (2) \\ w(i), & \text{for } \phi_{x^l}(i) = 0 \quad (3) \end{cases}$$

B. The averager CMAC

In [12], an MDP with six states and four binary features is presented for which the value iteration algorithm diverges if backups of the value function are approximated with a CMAC.[3]. The following argument is given to explain this undesired behavior: The CMAC possibly exaggerates the largest difference between two target functions f and g, if the two functions are replaced by the corresponding (CMAC) approximations $\hat{f}$ and $\hat{g}$. For example, let the largest difference between f and g be $d := ||f - g||_\infty$. If a CMAC is used to approximate the functions f, g, resulting in approximations $\hat{f}, \hat{g}$, it possibly holds $d < \hat{d} := ||\hat{f} - \hat{g}||_\infty$. Function approximators showing this behavior are called exaggerators. It is shown in [12] that a sequence of Q-learning updates on a function approximated by an exaggerator possibly diverges to infinity. In contrast, an approximator belonging to the averager class does not exaggerate and therefore never diverges.

We will now present an alternative update rule by which the CMAC is turned into an averager.

$$w(i)' = \begin{cases} y^l \ (initial), & \text{for } \phi_{x^l}(i) = 1 \quad (4) \\ (1-\alpha)w(i) + \alpha y^l, & \text{for } \phi_{x^l}(i) = 1 \quad (5) \\ w(i), & \text{for } \phi_{x^l}(i) = 0 \end{cases}$$

The first case is only relevant for the first, initial update of the parameter $w(i)$. A consequence of the new update rule is that the CMAC now computes an average value with respect to all activated features

$$\hat{Q}^{avr}(x) := \frac{1}{|\{j|\phi_x(j) = 1\}|} \sum_{i=1}^{n} w(i)\phi_x(i) \qquad (6)$$

We will call the new CMAC-architecture an A-CMAC or an averager CMAC. What is the difference between the CMAC

[3]The presented example considers the problem of approximating a value function instead of a Q-function as we do in this work

update rule (2) and the A-CMAC update rule (5)? Consider the second case of the A-CMAC update rule in detail: $w(i)' = w(i) + \alpha y^l - \alpha w(i)$. The only difference compared with the CMAC update rule is that the term $\alpha\hat{Q}(x^l) = \alpha\vec{\phi}_{x^l}\vec{w}$ is replaced by the term $\alpha w(i)$. The A-CMAC makes only local updates, since only $w(i)$ is incorporated into the update of the i'th component of $\vec{w}$. For example, consider a certain cell of a grid and the corresponding parameter $w(j)$. If $w(j)$ is updated according to the original CMAC update rule, the new value of $w(j)$ is influenced by parameters corresponding to other cells of other grids. Thus, the update incorporates information about other parts of the input space. This may increase the quality of the approximation, but it may also destroy convergence.

Theorem 1: Let $cmac$ be a CMAC-architecture for which every element of the input space activates the same number of binary features and all updates are performed according to the update rule of the A-CMAC. Further, let $(\hat{Q}_N^{avr})$ be the sequence of Q-functions generated by the extended fitted Q iteration algorithm if the $cmac$ architecture is used as the supervised learning method. Then, $(\hat{Q}_N^{avr})$ converges to a uniquely defined fix-point for $N \to \infty$.

Proof: The proof is adapted from [3] to the update rule used by the A-CMAC. Let n be the total number of binary features and m the constant number of features activated by an input $x = (s,u) \in (S \times U)$. We consider only iterations of the inference loop occuring after the set of transition instances $\mathcal{F}$ has been fixed ($|\mathcal{F}| \geq maxInstances$). Let the final set of training patterns be $\mathcal{TF} := \{(x^l, y^l),\ l = 1,...,T\}$. By inspecting the averager update rules (4),(5), it is easy to see that there exist constants $0 \leq \beta(i,l) < 1,\ i \in \{1,...,n\},\ l \in \{1,...,T\}$ such that $w(i) = \sum_{l=1}^{T}\beta(i,l)y^l$ with $\forall i \in \{1,...,n\} : \sum_{l=1}^{T}\beta(i,l) = 1$. Since it holds $y^l = r_t^l + \gamma \max_{u'\in U} \hat{Q}_{N-1}^{avr}(s_{t+1}^l, u')$, the approximated Q-function (6) can be written as

$$\begin{aligned}\hat{Q}_N^{avr}(s,u) &= \frac{1}{m}\sum_{i=1}^{n}\phi_{(s,u)}(i)\sum_{l=1}^{T}\beta(i,l)(r_t^l \\ &+ \gamma\max_{u'\in U}\hat{Q}_{N-1}^{avr}(s_{t+1}^l,u'))\end{aligned}$$

Let the operator $\hat{H}$ be a mapping defined on functions of the form $K : S \times U \to \mathbb{R}$

$$\begin{aligned}\hat{H}K(s,u) &:= \frac{1}{m}\sum_{i=1}^{n}\phi_{(s,u)}(i)\sum_{l=1}^{T}\beta(i,l)(r_t^l \\ &+ \gamma\max_{u'\in U}K(s_{t+1}^l,u'))\end{aligned}$$

Then it holds

$$\begin{aligned}&||\hat{H}K - \hat{H}\bar{K}||_\infty = \\ &\max_{(s,u)\in(S\times U)}[|\frac{1}{m}\sum_{i=1}^{n}\phi_{(s,u)}(i)\sum_{l=1}^{T}\beta(i,l) * \\ &\quad(\gamma\max_{u'\in U}K(s_{t+1}^l,u') - \gamma\max_{u'\in U}\bar{K}(s_{t+1}^l,u'))|] \\ &\leq \max_{(s,u)\in(S\times U)}[|\frac{1}{m}\sum_{i=1}^{n}\phi_{(s,u)}(i)\sum_{l=1}^{T}\beta(i,l) * \\ &\quad(\gamma\max_{u'\in U}[K(s_{t+1}^l,u') - \bar{K}(s_{t+1}^l,u')])|] \\ &\leq \max_{(s,u)\in(S\times U)}[|\frac{1}{m}\sum_{i=1}^{n}\phi_{(s,u)}(i)\sum_{l=1}^{T}\beta(i,l) * \\ &\quad(\gamma\max_{u'\in U,\ l'\in\{1,...,T\}}[K(s_{t+1}^{l'},u') - \bar{K}(s_{t+1}^{l'},u')])|] \\ &= \max_{(s,u)\in(S\times U)}[|\frac{1}{m}\sum_{i=1}^{n}\phi_{(s,u)}(i) * \\ &\quad(\gamma\max_{u'\in U,\ l'\in\{1,...,T\}}[K(s_{t+1}^{l'},u') - \bar{K}(s_{t+1}^{l'},u')])|] \\ &= \max_{(s,u)\in(S\times U)}[|\gamma\max_{u'\in U,\ l'\in\{1,...,T\}}[K(s_{t+1}^{l'},u') - \bar{K}(s_{t+1}^{l'},u')]|] \\ &\leq \max_{(s,u)\in(S\times U)}[|\gamma(K(s,u) - \bar{K}(s,u))|] \\ &= \gamma||K(s,u) - \bar{K}(s,u)||_\infty \\ &< ||K(s,u) - \bar{K}(s,u)||_\infty\end{aligned}$$

Since $\hat{H}$ is equal to the update operator of the averager CMAC, the sequence of Q-functions $(\hat{Q}_N^{avr})$ converges due to the (banach) fix point theorem (see for instance [13]) ■

C. Discussion of the A-CMAC

We want to discuss a certain benefit of weighted averages as presented above, compared to equally weighted averages. Examples for equally weighted averages are ensembles of regression trees as used in [3] or k-nearest neighbor without distance weighting. Both the CMAC and the A-CMAC weight an individual update by a positive learning rate α. To illustrate this fact, we perform three example-updates (A-CMAC) on a parameter w for different training patterns $(x^1, y^1), (x^2, y^2), (x^3, y^3)$. We assume that all training patterns activate the feature corresponding to parameter w. If the patterns are presented in ascending order, the final value of w is $(1-\alpha)^2y^1 + (1-\alpha)\alpha y^2 + \alpha y^3$. If additional training examples are presented, the influence of the first training patterns will decrease more and more, since y^1, y^2, y^3 will be repeatedly multiplied by $(1-\alpha)$. Thus, the A-CMAC tends to prefer training patterns which are presented last. In other words, the A-CMAC builds a prototypical output value based on the last training patterns. This effect is especially strong for high learning rates. Even if the learning rate is lowered, the effect will be still visible if a large set of training patterns is used. Trivially, an equally weighted average is equivalent to the special learning rate $\alpha = \frac{1}{|\mathcal{TF}|}$.

Consider a second example (Fig. 3) showing a small set of training patterns combined with a high learning rate. Again, we will assume that the training patterns are presented in ascending order. After performing several updates for every training pattern, parameter $w(1)$ will be most influenced by training pattern (x^2, y^2), while parameter $w(2)$ will be most influenced by training pattern (x^3, y^3). This is reasonable, since (x^2, y^2) is prototypical for the left cell, while (x^3, y^3) is prototypical

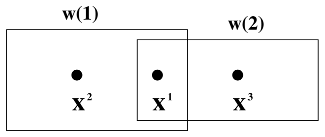

Fig. 3. Two overlapping cells corresponding to the parameters $w(1)$ and $w(2)$. The three dots denote training patterns located at different positions within the input space. The learning rate α is set to 0.8

for the right cell. Indeed, if the training patterns are presented in a different order, it may occur that training pattern (x^1, y^1) influences both cells the most. A straightforward extension to the A-CMAC update rule will be therefore to compute an optimal presentation order for the training patterns. However, we believe that even if the presentation order is suboptimal, weighting the updates with a positive learning rate is better than computing an equally weighted average. For example, consider a set of grids with low resolution, as we used them for our empirical results. Since many training patterns will fall into the same grid cell, an equally weighted average will not be meaningful. In such a setting, a prototypical representation of the input space is better suited even if the cells have suboptimal prototypes. This may also be a reason why the CMAC works well with low resolutional grids. We will show empirical results concerning this issue in the next section.

IV. Empirical Results

We performed experiments both for the mountain car plant and the acrobot plant. Every experiment consists of several runs of the (extended) fitted Q Iteration algorithm combined with the CMAC-architectures described above.

1) CMAC (original)
2) A-CMAC (averager CMAC)
3) AE-CMAC (A-CMAC with equally weighted averages)

For both benchmark problems, we used a similar layout of the feature space. Twenty (ten) overlapping grids are laid over the state-action space for the acrobot (mountain car) benchmark. All grids have a constant resolution of ten in every dimension of the state space. Since both the mountain car and the acrobot come with discrete actions, the resolution of the one-dimensional action space was designed such that every action falls into a distinct grid cell. In general, all described CMAC-architectures are able to deal with continuous action spaces. All sampled transition instances are duplicated ten times to enlarge the set of training patterns.

In the following subsection, we will empirically compare the performance of the different CMAC-architectures with respect to convergence issues as well as to the quality of the learned policies.

A. Mountain Car

The task of the mountain car benchmark is to drive a car to the top of a mountain by accelerating the car in the forward and backward direction. The system dynamics we used for simulating the mountain car can be found in [14]. The state space is two-dimensional and consists of the position (x) of the car and the velocity ($\dot{x}$) of the car. An episode ends if a goal state is reached or the current state violates the constraints $x \in [-2, 2], \dot{x} \in [-5, 5]$. The set of goal states is given by $\{(x, \dot{x}) | \ x \geq 1.0\}$. During the learning phase of the algorithm, an episode is also terminated if the length of the episode exceeds the threshold $maxSteps = 300$. Two discrete actions are available, corresponding to forces from the set $F \in \{-4, 4\}$. The discount rate was set to $\gamma = 0.98$ and the learning rate was set to $\alpha = 0.01$. We used the following reward function

$$r(s_t, u_t) = \begin{cases} 0, & \text{if } s_t \text{ is a goal state} \\ -1000 & \text{if } s_t \text{ violates constraints} \\ -1, & \text{else} \end{cases}$$

We found out that the convergence of the original CMAC heavily depends on the number of sampled episodes. We therefore performed two experiments, one with 50 sampled episodes and another with 150 sampled episodes. No additional sampling by an ϵ-greedy policy was done. Thus, every run of the algorithm consists of a single iteration of the main loop. The inference loop (inner loop) terminates after $N = 500$ iterations. All episodes start from a randomly chosen state from the set $\{(x, \dot{x}) | \ x \in [-0.5, 0.5], \dot{x} = 0\}$. The following figures show the performance of twenty averaged runs of the fitted Q Iteration algorithm after sampling 50 episodes. This corresponds approximately to 6200 sampled transition instances. The figures show that the original CMAC was not

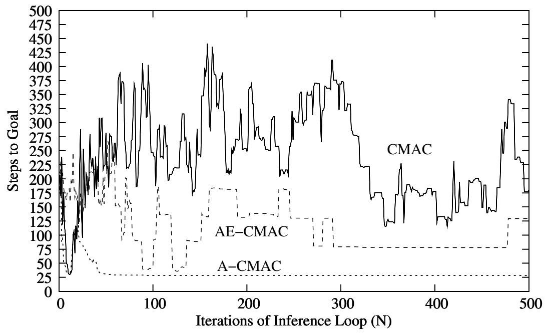

Fig. 4. Steps to the goal. The learned policies were tested by running 1000 episodes from randomly chosen starting states (50 sampled episodes)

able to converge to a stable policy. In contrast to the averager CMACs, the Bellman Residual diverged to infinity. The policy found by the A-CMAC is not optimal (which would be 17 steps) but reasonably good.

The figures 8-11 show the performance of twenty averaged runs of the fitted Q Iteration algorithm after sampling 150 episodes. This corresponds approximately to 18600 sampled transition instances. The figures show that the original CMAC is more likely to converge to a stable policy if many transition instances are sampled. If many transition instances are available, the original CMAC outperforms the averager CMACs. Unfortunately, there is no guarantee for such a behavior. Fig. 10 shows that the Bellman Residual of the

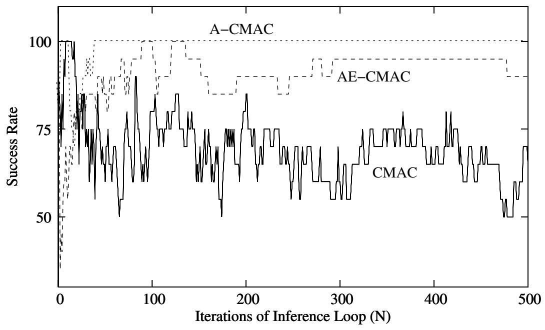

Fig. 5. Success rate. The success rate gives the percentage of episodes reaching a goal state (50 sampled episodes)

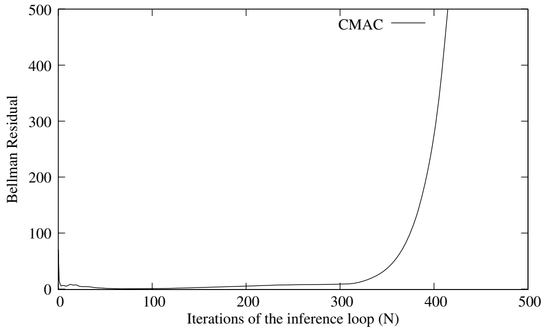

Fig. 6. Bellman Residual: $\frac{\sum_{(x^l,y^l)\in\mathcal{TF}}(\hat{Q}_N(x^l)-\hat{Q}_{N-1}(x^l))^2}{|\mathcal{TF}|}$ (50 sampled episodes)

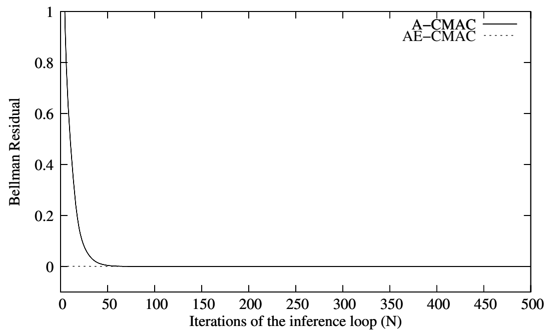

Fig. 7. Bellman Residual: $\frac{\sum_{(x^l,y^l)\in\mathcal{TF}}(\hat{Q}_N(x^l)-\hat{Q}_{N-1}(x^l))^2}{|\mathcal{TF}|}$ (50 sampled episodes)

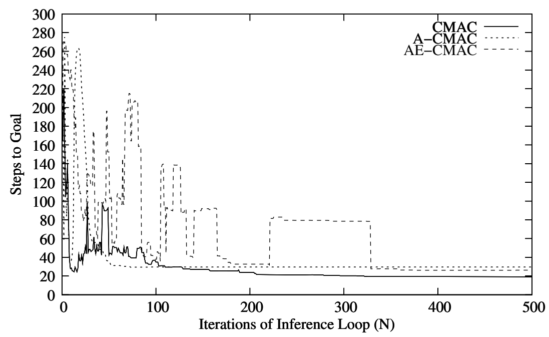

Fig. 8. Steps to the goal. The learned policies were tested by running 1000 episodes from randomly chosen starting states. (150 sampled episodes)

original CMAC may diverge even if the policy converges near to an optimum.

B. Acrobot

A detailed task description and the system dynamics of the acrobot benchmark can be found in [15]. To summarize, the goal is to swing up the second link of the acrobot above a line of certain height. The state space is four-dimensional and consists of the angles of the two links θ_1, θ_2 and their corresponding angular velocities $\dot{\theta_1}, \dot{\theta_2}$. An episode ends if a goal state is reached or the current state violates the constraints $\dot{\theta_1} \in [-4\pi, 4\pi]$ and $\dot{\theta_2} \in [-9\pi, 9\pi]$. During the learning phase of the algorithm, an episode is also terminated if the length of the episode exceeds the threshold $maxSteps = 500$. For testing a learned policy, we allow episodes to have a length of 1000. The action set consists of three discrete torques $F \in \{-1, 0, 1\}$ applied at the second joint of the acrobot. The discount rate was set to $\gamma = 0.98$, the learning rate to $\alpha = 0.01$ and the exploration rate to $\epsilon = 0.1$. The reward function is analog to the function used for the mountain car benchmark.

Every run of the algorithm consisted of six iterations of the main loop. During every iteration of the loop, one hundred new episodes were sampled by an ϵ-greedy exploration strategy. The inference loop (inner loop) always terminated after a hundred iterations. Thus, a total number of $N = 600$ approximations of the Q-function was computed. All episodes started from a randomly chosen state from the set $\{(\theta_1, \theta_2, \dot{\theta_1}, \dot{\theta_2}) | \ \theta_1 \in [-\pi, \pi], \theta_2 = \dot{\theta_1} = \dot{\theta_2} = 0\}$.

The figures 12-13 show the performance of ten averaged runs of the fitted Q iteration algorithm.

In Table I, a summary is given about the sampling process during the six iterations of the main loop. The second column gives the (cumulated) number of sampled episodes and the third column gives the number of sampled state transitions. The A-CMAC samples less transition instances, since the learning process quickly stabilizes on a policy that always reaches the goal state. Since every episode is terminated in a goal state, relatively few state transitions are collected.

Architecture	# Episodes	# State Transitions
CMAC	100	27250
CMAC	300	97658
CMAC	600	123304
A-CMAC	100	27075
A-CMAC	300	49111
A-CMAC	600	63230
AE-CMAC	100	26340
AE-CMAC	300	79333
AE-CMAC	600	157861

TABLE I
SAMPLING PROCESS

Similar to the mountain car benchmark, the bellman residual diverged for the original CMAC but converged for the averager

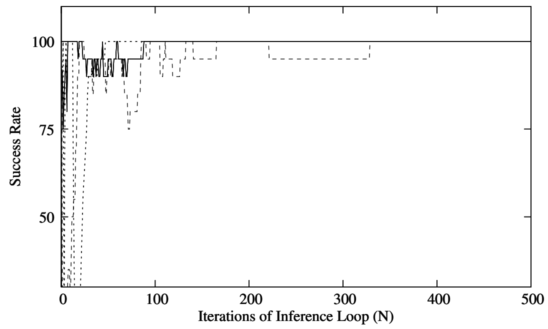

Fig. 9. Success rate. The success rate gives the percentage of episodes reaching a goal state. (150 sampled episodes)

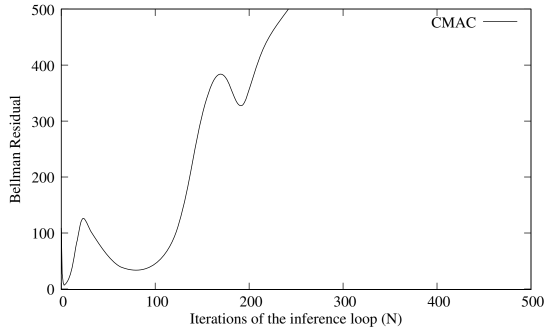

Fig. 10. Bellman Residual: $\frac{\sum_{(x^l,y^l)\in\mathcal{TF}}(\hat{Q}_N(x^l)-\hat{Q}_{N-1}(x^l))^2}{|\mathcal{TF}|}$ (150 sampled episodes)

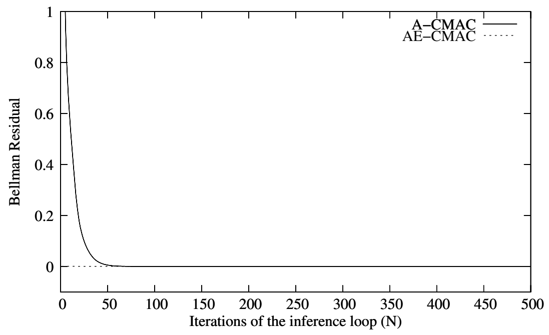

Fig. 11. Bellman Residual: $\frac{\sum_{(x^l,y^l)\in\mathcal{TF}}(\hat{Q}_N(x^l)-\hat{Q}_{N-1}(x^l))^2}{|\mathcal{TF}|}$ (150 sampled episodes)

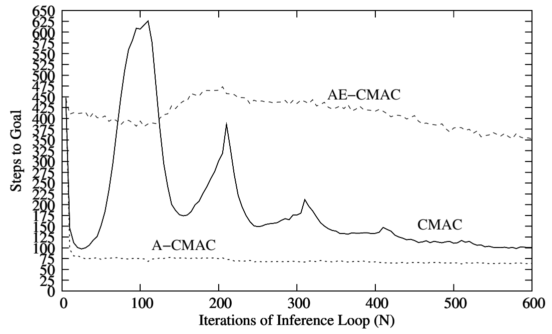

Fig. 12. Steps to the goal. The learned policies were tested by running 1000 episodes from randomly chosen starting states. All episodes have a maximal length of 1000 steps

CMACs. We omitted figures due to space constraints. Again, we made an additional experiment for the original CMAC with a larger number of sampled transition instances. The figures 14-15 show an experiment in which every run of the algorithm consists of three iterations of the main loop. During every iteration of the main loop, 500 episodes are sampled and 500 iterations of the inference loop are performed. Thus, the total number of sampled episodes is 1500. The average number of sampled transition instances was 242720. Obviously, the performance of the original CMAC is much better compared with the first experiment. In this setting, the A-CMAC is able to learn a policy which reaches a goal state after 55 steps (a figure is omitted).

Another possibility to increase the performance of the original CMAC is to reset the weights of the CMAC to the initial values after each iteration of the inference loop. Thus, for every Q iteration, we take a new CMAC. By implementing this procedure we prevent the weights of the CMAC from diverging to infinity. For a comparison of the A-CMAC with the original CMAC with *weight reset*, we made an additional experiment (with a setup equal to the first experiment) shown in Fig. 16. The figure shows that the original CMAC now clearly outperforms the A-CMAC.

V. Summary and Conclusion

The fitted Q iteration algorithm is designed to learn a Q-function on the basis of a small set of sampled state transitions. To make efficient use of the gathered data, it is desirable to combine this algorithm with powerful function approximators. We empirically showed that the CMAC approximator performs well in combination with the fitted Q iteration algorithm. Indeed, there is no guarantee that the CMAC will converge for an arbitrary chosen problem, especially if only less transition instances are available. In case of divergence, it is possible to switch to a variant of the CMAC called A-CMAC, which builds a weighted average of the output values. We showed that the A-CMAC is guaranteed to converge to a unique approximation of the Q-function. Compared to the original CMAC, using the A-CMAC may result in a loss of performance. Nevertheless, the A-CMAC inherits some basic ideas of the CMAC architecture. One of these is the layout of the feature space, another is the way of weighting the updates. We showed that non-equally weighting the updates of individual training patterns is better suited for coarse representations of the state (action) space than equally weighting the updates. We conclude that the A-CMAC is a valuable alternative to the original CMAC as long as there are no further theoretical results concerning the convergence of the CMAC.

References

[1] C. Watkins, "Learning from delayed rewards," Ph.D. dissertation, University of Cambridge, England, 1989.

[2] D. P. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*. Belmont Massachusetts: Athena Scientific, 1996.

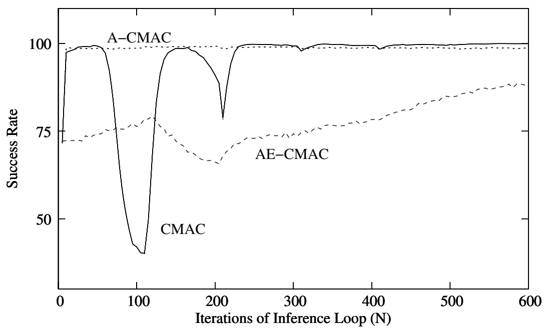

Fig. 13. Success rate. The success rate gives the percentage of episodes reaching the goal state.

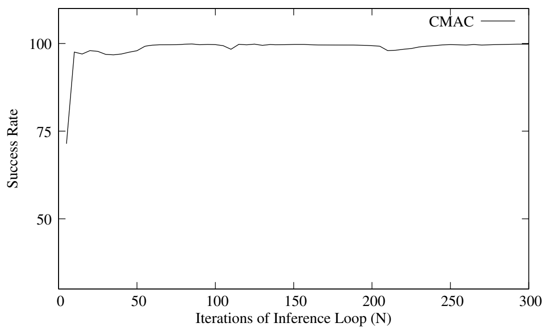

Fig. 15. Success rate. The success rate gives the percentage of episodes reaching the goal state (1500 sampled episodes)

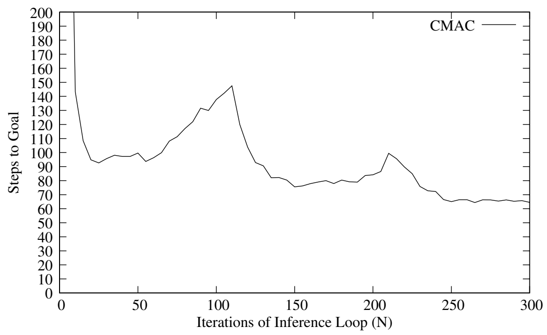

Fig. 14. Steps to the goal. The learned policies were tested by running 1000 episodes from randomly chosen starting states. All episodes have a maximal length of 1000 steps (1500 sampled episodes)

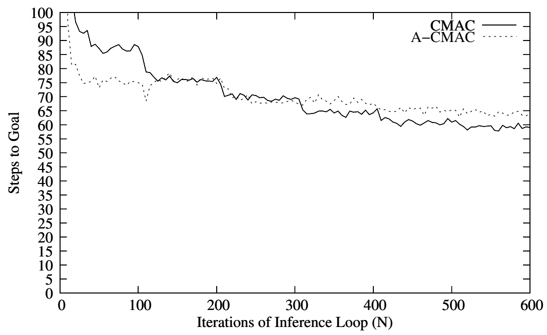

Fig. 16. Steps to the goal. The learned policies were tested by running 1000 episodes from a randomly chosen starting state. All episodes have a maximal length of 1000 steps (weight reset)

[3] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *Journal of Machine Learning Research*, vol. 6, pp. 503–556, April 2005.

[4] G. J. Gordon, "Stable function approximation in dynamic programming," in *Proceedings of the Twelfth International Conference on Machine Learning*, A. Prieditis and S. Russell, Eds. San Francisco, CA: Morgan Kaufmann, 1995, pp. 261–268.

[5] S. P. Singh, T. Jaakkola, and M. I. Jordan, "Reinforcement learning with soft state aggregation," in *Advances in Neural Information Processing Systems : Proceedings of the 1994 conference*, G. Tesauro, D. Touretzky, and T. Leen, Eds., vol. 7. The MIT Press, 1995, pp. 361–368.

[6] D. Ormoneit and S. Sen, "Kernel-based reinforcement learning," *Machine Learning*, vol. 49, no. 2-3, pp. 161–178, 2002.

[7] M. Riedmiller, "Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method," in *Proceedings of the Sixteenth European Conference on Machine Learning, Porto, Portugal*, 2005, pp. 317–328.

[8] M. G. Lagoudakis and R. Parr, "Least-squares policy iteration," *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2203a.

[9] J. A. Boyan, "Technical update: Least-squares temporal difference learning," *Machine Learning*, vol. 49, no. 2-3, pp. 233–246, 2002.

[10] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Proceedings of the International Conference on Advances in Neural Information Processing Systems*, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds., vol. 8. The MIT Press, 1996, pp. 1038–1044.

[11] S. P. Singh and R. S. Sutton, "Reinforcement learning with replacing eligibility traces," *Machine Learning*, vol. 22, no. 1-3, pp. 123–158, 1996.

[12] G. J. Gordon, "Stable function approximation in dynamic programming," Carnegie Mellon University, Pittsburgh, PA 15213, Tech. Rep. CMU-CS-95-103, January 1995.

[13] D. Luenberger, *Optimization by Vector Space Methods*. New York: Wiley, 1969.

[14] R. Schoknecht and M. Riedmiller, "Reinforcement learning on explicitly specified time-scales," *Neural Computing*, vol. 12, no. 2, pp. 61–80, 2003.

[15] R. Sutton and A. Barto, *Reinforcement Learning, An Introduction*. Cambridge, Massachusetts: The MIT Press, 1998.