Reinforcement Learning with RBF-Networks

Scientific Project
University of Applied Sciences Weingarten

by
Markus Schneider

Feb. 2008

Abstract

As far as we leave the domain of small state and action spaces, we cannot represent the value function as a table anymore. Generalization is needed. Radial Basis Functions are often used in reinforcement learning as linear function approximation to learn a value function. These functions have very good convergence guarantees and producing very smooth approximations.

This work will give a short introduction to reinforcement learning and function approximation. Standard Radial Basis Functions and Gaussian Softmax Basis Function will be compared. The performance of the discussed algorithms will be applied on two examples: The Mountain Car Task and the Crawling Robot Task.

Contents

1 The Reinforcement Learning Framework

At the beginning an agent is in an initial state. At each time step the agent performs an action, the environment returns a new state and a reward. The environment could be stochastic, that means it is possible to do the same action in the same state twice and reach a different next state or a different reward. This could be caused by noisy sensor data, inaccuracy in motor control or a changing world. The goal for the agent is to find a sequence of actions that maximizes the cumulative rewards over time. The reinforcement learning framework allows the agent to learn this sequence through interaction with the environment.

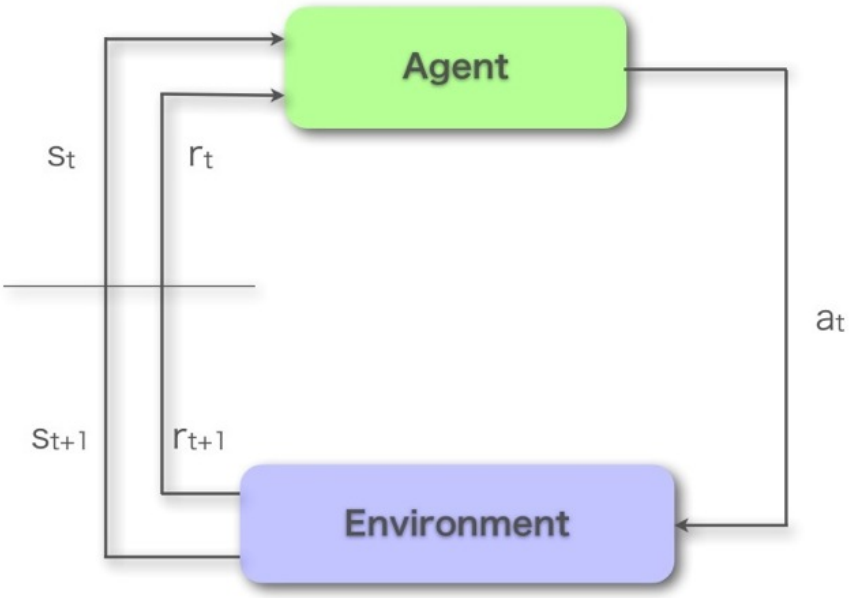


Figure 1: The agent-environment interaction [Sutton and Barto, 1998]

1.1 States and Actions

More mathematically the reinforcement learning framework could be described as following: At a given time t the agent is in a state $s_t \in \mathcal{S}$ and can apply an action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{S}$ is the set of all states and $\mathcal{A}(s_t)$ is a set of all actions possible in state s_t. As a result the agent transfers to a next state s_{t+1} and receives a numerical reward[1] $r \in \mathbb{R}$. The received next state and reward are not necessarily correct. Therefore it could happen that due to noisy sensor data a wrong state is assumed.

[1] A negative reward is also oftern called costs

1.2 The Policy

The mapping from states to action is done by the agent's *policy* and is denoted by $\pi_t(s, a)$. The policy could be interpreted as the probability of choosing action a in state s.

$$\pi(s, a) \quad = \quad Pr\{a = a_i \mid s = s_t\} \tag{1}$$

1.3 The Markovian System

In reinforcement learning, we generally assume that reaching a successor state s_{t+1} only depends on the current state s_t and the action a_t, chosen in this state. This assumption is true for most games like chess or backgammon. All relevant information is given by the positions of figures on the board, no matter how these positions had been reached. Such a system is called *Markovian system* and satisfies the following equation:

$$Pr\{s_{t+1}, r_{t+1} \mid T(s_{0:t}, a_{0:t})\} \tag{2}$$
$$= \quad Pr\{s_{t+1}, r_{t+1} \mid s_t, a_t\} \tag{3}$$

where $T(s_{0:t}, a_{0:t})$ denotes the "history" of all previous states $s_{0:t}$ and actions $a_{0:t}$.

The *Markovian system* could be fully described by the set of states $\mathcal{S}$, the set of actions available in s, $\mathcal{A}(s)$ and the transition probability between two states

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} \mid s_t, a_t\} \tag{4}$$

together with the expected value of the next reward

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} \mid s_t, a_t, s_{t+1}.\} \tag{5}$$

1.4 Returns

As described before, the agent's goal is to maximize the reward in the long run. In this case it is not sufficient to choose the action that returns the highest reward in the next step. Also action and rewards more far in the future must be considered.

There are two classes of learning problems: If there is a final terminal state s_T we talk about an *episodic task*. After reaching a terminal state the episode ends and

the system resets to a starting state. In this case we can simply sum up all future rewards. The expected long term return R_t at time t is given by:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{6}$$

This approach is impossible for the second class of learning problems - the *continuing tasks*. It is not possible to break these tasks into episodes and in contrast to *episodic tasks*, there is no terminal state. Rewards in the far future must be discounted to avoid an infinite sum.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{7}$$

where, the parameter γ, $0 \leq \gamma \leq 1$, is called the *discount rate*. Future rewards could be weighted with this parameter.

We can combine both definitions to following equation:

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{8}$$

where $T = \infty$ for *continuing tasks* and $\gamma = 1$ for *episodic tasks*.

1.5 Value Functions

Most reinforcement learning algorithms try to estimate how good a given state is, or how good it is to perform a given action in a certain state. These estimates are called *state-value functions* or *action-value functions* respectively. Normally the value of a state or action is the expected future reward like described before. Of course the future reward depends on the agent's actions. Therefore, the value functions are defined with respect to a policy. The *state-value function* can be defined as

$$V^{\pi}(s_t) = E_{\pi}\{R_t \mid s_t\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t\right\} \tag{9}$$

here, $E_{\pi}\{\}$ denotes the expected value, being in state s_t and following the policy π. The *action-value functions* look's similar:

$$\begin{aligned} Q^{\pi}(s_t, a_t) &= E_{\pi}\{R_t \mid s_t, a_t\} && (10) \\ &= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t, a_t\right\} && (11) \end{aligned}$$

A very important property of value functions is the relation between a state and its possible successor states. This relation is described by the *Bellman equation* for state-value functions[2]:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s') \right] \tag{12}$$

and for action-value functions:

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a') \right] \tag{13}$$

The variables $\mathcal{P}^{a}_{ss'}$, $\mathcal{R}^{a}_{ss'}$ and π averages over all possibilities and weight the values by its probability of occurrence.

1.6 Temporal Difference Learning

One of the most successful algorithms in the last few years was *temporal difference learning* (TD). This algorithm tries to calculate the value function with the temporal error in the current estimation:

$$\delta^{\pi} = r_{t+1} + \gamma \hat{V}^{\pi}(s_{t+1}) - \hat{V}^{\pi}(s_t) \tag{14}$$

This can now used as an update for the state-value functions and action-value functions. Here $\hat{V}^{\pi}$ denotes that this is an estimate, not the true value function. TD learning need to wait until the timestep $t+1$ to get the reward r_{t+1} and the next state s_{t+1} and then performs the update for timestep t.

```
1 Initialize V(s) arbitrarily, π to the policy
2 Repeat (for each episode)
3   Init s
4   Repeat (for each step of episode):
5     a ← action given by π for s
6     Take action a, observe r, s'
7     V(s) ← V(s) + α[r + γV(s') − V(s)]
8     s ← s'
9   until s is terminal
```

Listing 1: Tabular TD(0)

[2]See [Sutton and Barto, 1998, Chapter 3.7] for the full proof

The parameter $0 < \alpha <= 1$ is used to weight the current observation or to define how reliable the sensor-information is. This is called the *learning rate* and could also depend on time.

This algorithm could be easily reformulated for action-value functions. For a detailed explanation of *SARSA* and *Q-learning* see [Sutton and Barto, 1998, Chapter 6.4 and 6.5].

2 Function Approximation

For continuous or large discrete state spaces it is no longer possible to store the value function in a table. Function approximation is needed to generalize and interpolate over states. From now on the value function V_t is an approximation with parameter vector $\vec{\theta}_t$. That means V_t is totally depending on $\vec{\theta}_t$. The advantage is that the number of components of $\vec{\theta}_t$ is typically much less than the number of states.

Normally we try to minimize the mean-squared error (MSE) between the approximated function V_t and the true value function V^π.

$$MSE(\vec{\theta}_t) = \sum_{s \in S} P(s) \Big[V^\pi(s) - V_t(s) \Big]^2 \tag{15}$$

where P is a distribution weighting the errors of different states. This is necessary because it is usually impossible (or undesired) to reduce the error to 0 at all states. The objective is to find a parameter vector $\vec{\theta}^\star$ for which $MSE(\vec{\theta}^\star) \leq MSE(\vec{\theta})$ for all possible $\vec{\theta}$. This is the *global optimum* and not always possible. But we can try to converge to a *local optimum* where the above equation is not true for all $\vec{\theta}$, but for those in some neighborhood of $\vec{\theta}^\star$.

2.1 Gradient-Descent Methods

One general mathematical approach to minimize the function $MSE(\vec{\theta}_t)$ is gradient descent. After each observed example gradient descent adjusts the parameter vector a small amount in the direction to the negative gradient. We can only do small adjustments of the parameters, because big steps could affect the value function in an unstable way. This is especially in the topic of robotics extremely dangerous.

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2} \alpha \nabla_{\vec{\theta}_t} \Big[V^\pi(s_t) - V_t(s_t) \Big]^2 & (16) \\ &= \vec{\theta}_t + \alpha \Big[V^\pi(s_t) - V_t(s_t) \Big] \nabla_{\vec{\theta}_t} V_t(s_t) & (17) \end{aligned}$$

This algorithm is proven to converge to a *local optimum* if the learning rate α satisfies the following properties depending on time t:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \qquad \sum_{t=0}^{\infty} \alpha_t^2 < \infty \tag{18}$$

2.2 Linear Function Approximation

A very common way in reinforcement learning is the use of *linear function approximations*. These functions are very easy to learn and can generalize over states. It is called linear, because the approximate function V_t is a linear function of the parameter vector $\vec{\theta}_t$. That means for every state s, there is a *feature vector* $\phi(s)$ of the same length as θ. The value of the function is then calculated with:

$$V_t(s) = \vec{\theta}_t^T \cdot \vec{\phi}(s) = \sum_{i=1}^{N} \theta_i \cdot \phi_i(s) \tag{19}$$

where $\phi_i(s)$ is called the *activation function* and θ_i is the weight of the feature i.

For gradient-decent on linear function approximation we can easily calculate the gradient with respect to $\vec{\theta}_t$ as

$$\nabla_{\vec{\theta}_t} V_t(s) = \vec{\phi}(s). \tag{20}$$

In the linear case there is only one optimum $\vec{\theta}^\star$ and this method is guaranteed to converge to or near the global optimum. Therefore these techniques are very favorable in the learning task. Some popular methods are *Coarse Coding*, *Tile Coding*, *RBF-Networks* and *Gaussian Softmax Basis Function Networks*. For a detailed discussion of the last two approaches see section 3. We can also see tables as a special case of linear function approximation where θ_i is a single index representing state i and $\phi(s)$ is a vector where only the i^{th} element is 1.0 and all others are 0.0.

2.3 TD-Learning with Function Approximation

We can easily reformulate the standard temporal difference learning algorithms with the help of Equation 16. Now not the value-function itself will be updated, but the elements of the parameter vector $\vec{\theta}$ according to the gradient of the error function.

In all algorithm the vector $\vec{e}$ is used as a *eligibility trace*[3] for active features. The variable λ is the *trace-decay parameter* to shift the e-trace between the one-step TD update ($\lambda = 0$) and the full Monte Carlo[4] update ($\lambda = 1$). The variable γ is the normal *discount rate* like mentioned before and α is the *learning rate*.

2.3.1 Gradient Descent TD(λ)

```
1  Initialize θ⃗ arbitrarily
2  Repeat (for each episode)
3    e⃗ = 0⃗
4    s ← initial state of episode
5    Repeat (for each step of episode):
6      a ← action given by π for s
7      Take action a, observe r, s'
8      δ ← r + γV(s') − V(s)
9      e⃗ ← γλe⃗ + ∇_θ⃗ V(s)
10     θ⃗ ← θ⃗ + αδe⃗
11     s ← s'
12   until s is terminal
```

Listing 2: Gradient Descent TD(λ)

[3]For an detailed explanation of eligibility traces see [Sutton and Barto, 1998, Chapter 7.2]

[4]Monte Carlo Methods are based on averaging a lot of samples to solve the reinforcement learning problem. They are also introduced in [Sutton and Barto, 1998, Chapter 5]

2.3.2 Gradient Descent SARSA(λ)

```
Initialize θ⃗ arbitrarily
e⃗ = 0⃗
s, a ← initial state and action
Repeat (for each episode)
    Take action a, observe r, s′
    Choose a′ from π(s′)
    δ ← r + γQ(s′, a′) − Q(s, a)
    e⃗ ← γλe⃗ + ∇_θ⃗ Q(s, a)
    θ⃗ ← θ⃗ + αδe⃗
    s, a ← s′, a′
  until s is terminal
```

Listing 3: Gradient Descent SARSA(λ)

2.3.3 Gradient Descent Watkins's Q(λ)

```
Initialize θ⃗ arbitrarily
e⃗ = 0⃗
s, a ← initial state and action
Repeat (for each episode)
    Take action a, observe r, s′
    Choose a′ from π(s′)
    a* ← argmax_b Q(s′, b)
    δ ← r + γQ(s′, a*) − Q(s, a)
    e⃗ ← γλe⃗ + ∇_θ⃗ Q(s, a)
    θ⃗ ← θ⃗ + αδe⃗
    if a′ not a*, then e⃗ = 0⃗
    s, a ← s′, a′
  until s is terminal
```

Listing 4: Gradient Descent Watkins's Q(λ)

3 RBF-Networks

A Radial Basis Function network is a simple 3-layer network as shown in Figure 2. The input layer is a simple fan-out layer and does no processing. The second layer consists of Gaussian kernels, weighting the distance between the input and the center of the kernel, typically called the *activation function* of the kernel. The final layer is the weighted sum of its inputs with a linear output.

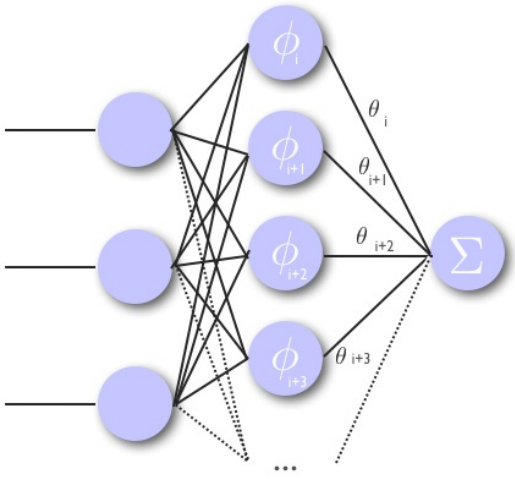


Figure 2: A typical Radial Basis Function network

The advantages of RBF-Networks are, that the single features are no longer limited to be 1 or 0. Now each feature can be any value in the interval $[0, 1]$. Usually there is more than 1 feature active, unlike in the tabular case[5]. Doya used in [Doya and Morimoto] modified RBF networks to learn a stand-up task for a robot.

The activation function of the i^{th} Gaussian kernel given some input vector $\vec{x}$ could be calculated by

$$\phi_i(\vec{x}) = exp\left(-\sum_{j=0}^{n} \frac{(x_j - \mu_{ij})^2}{2\sigma_{ij}^2}\right). \tag{21}$$

$\vec{\mu}_i$ is the center of the i^{th} kernel location and $\vec{\sigma}$ is the width of the i^{th} kernel. In Figure 3 is an example for 3 one dimensional Gaussian kernels, all with $\sigma = 0.5$ and centers on 0, 1, 2 respectively drawn with a dotted line. The weights for the kernels are 0.4, 0.9 and 0.9. The black solid line denotes the weighted sum of all 3 activation functions. If σ becomes higher, then the shape of the kernel becomes wider.

[5]To work proper, there should be at least two active features per state

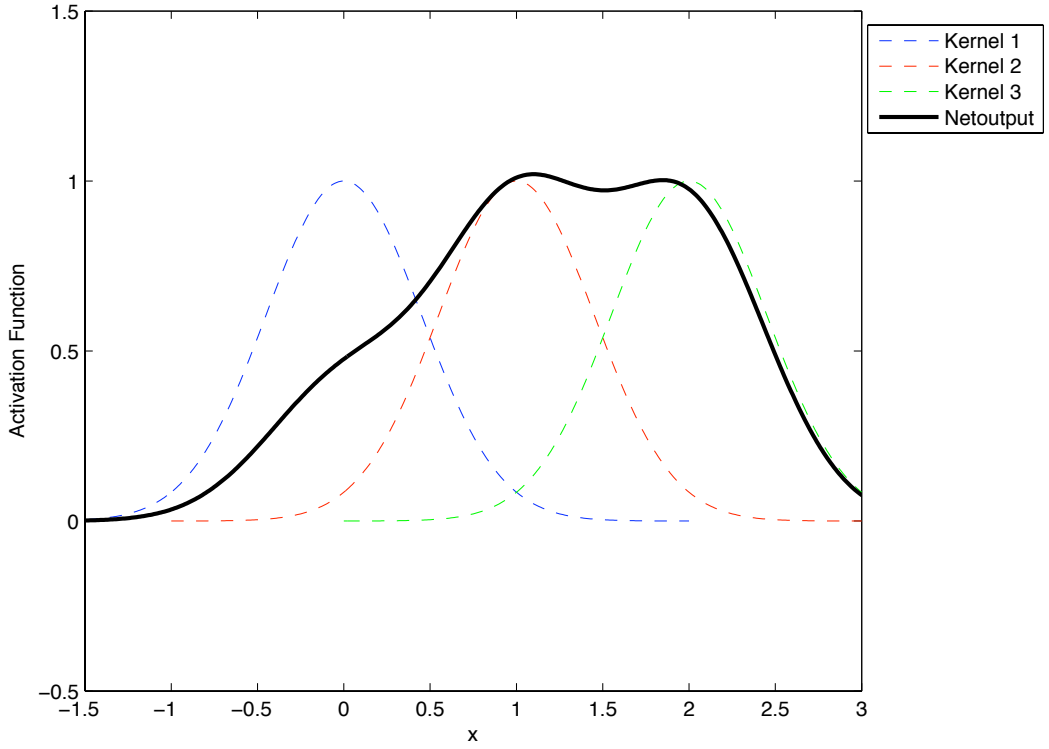


Figure 3: Three one dimensional Gaussian kernels. The weights for the kernels are 0.4, 0.9 and 0.9. The black solid line denotes the weighted sum of all.

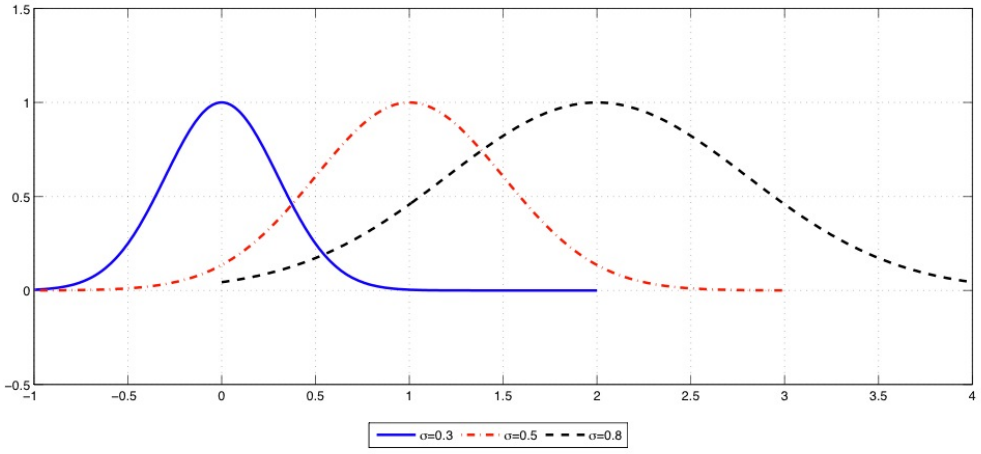


Figure 4: Adjusting the width of a kernel with parameter σ. 0.3, 0.5, 0.8 from left to right.

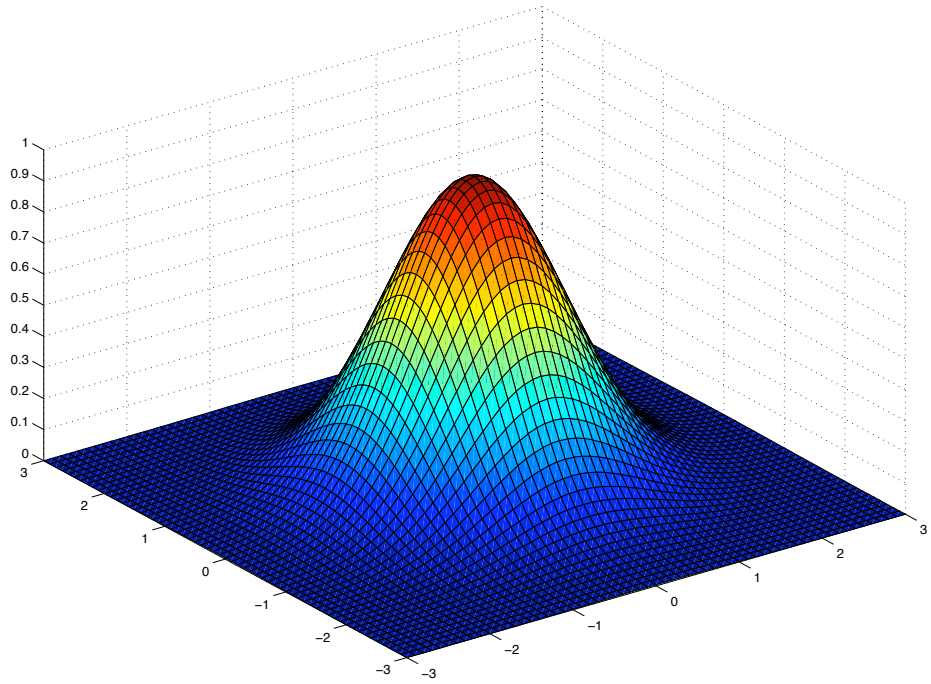


Figure 5: A two dimensional Gaussian kernel

3.1 Gaussian Softmax Basis Function Networks (GSBFN)

GSBFNs or Normalized Radial Basis Functions are slightly modified RBFs, where the sum all features is normalized to 1.0. This results in a better interpolation in regions where fewer kernels located as shown in Figure 6. We can modify the standard activation function calculation in Equation 23 to

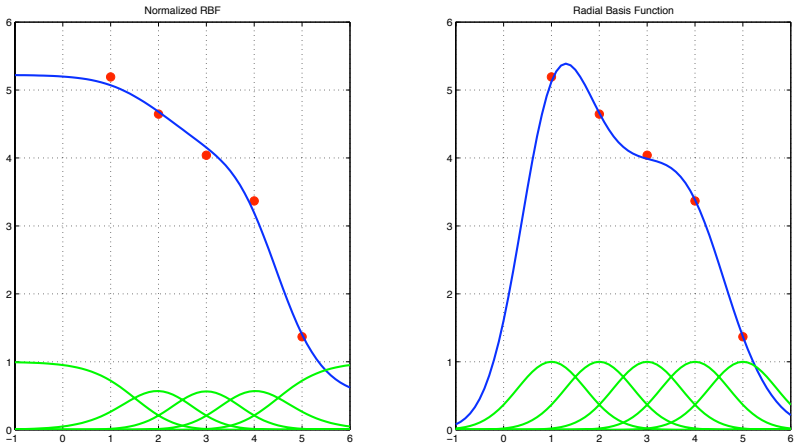
$$\phi_i(\vec{x}) = \frac{\Psi_i(\vec{x})}{\sum_{j=1}^{n} \Psi_j(\vec{x})} \tag{22}$$

with

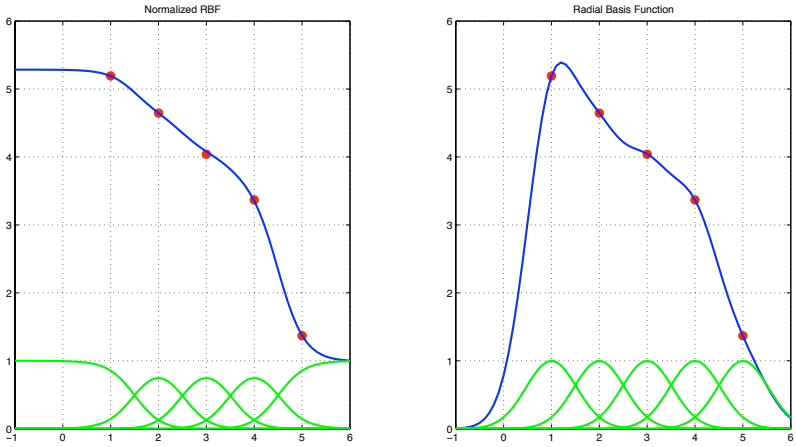
$$\Psi_i(\vec{x}) = exp\left(-\sum_{j=0}^{n} \frac{(x_j - \mu_{ij})^2}{2\sigma_{ij}^2}\right). \tag{23}$$

A GSBFN is still a linear function approximation, there the value function can still be calculated as described in section 2 by

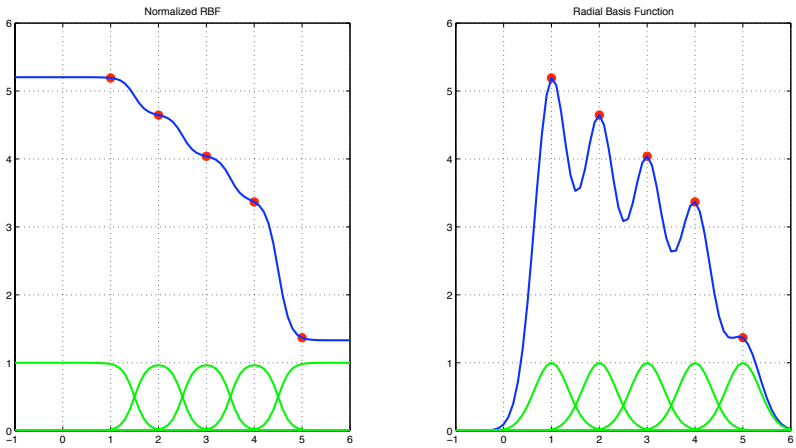
$$V_t(s) = \vec{\theta}_t^T \cdot \vec{\phi}(s) = \sum_{i=1}^{N} \theta_i \cdot \phi_i(s) \text{ and gradient } \nabla_{\vec{\theta}_t} V_t(s) = \vec{\phi}(s)$$



(a) $\sigma = 1$



(b) $\sigma = 0.75$



(c) $\sigma = 0.5$

Figure 6: Examples of GSBFNs (left) and RBFs (right) and the resulting value functions. The red circles are samples of the desired function. Green are the single RBF functions and blue is the approximation.

4 Experimental Results

4.1 The Mountain Car Task

The Mountain Car Task was first introduces by [Moore, 1990] and the goal is to drive an underpowered car up a hill. The problem is, that the force of gravity is stronger than the car's engine. Even at full throttle the car cannot accelerate up the slope. Therefore the agent must first drive in the opposite direction and up the slope on the left. Then, applying full throttle, the agent can reach enough velocity to drive up to the goal. There are three possible actions: full throttle forward (+1), full throttle reverse (−1) and zero throttle (0). The reward is −1 everywhere until the car moves past the goal position. The agent's state, $s = [x, \dot{x}]^T$, is described by two continuous variables: The position and velocity of the car. These variables are updated by

$$x_{t+1} = x_t + \dot{x}_{t+1} \tag{24}$$
$$\dot{x}_{t+1} = \dot{x}_t + 0.001 a_t - 0.0025 \cos(3 x_t) \tag{25}$$

where the position and velocity are limited to $-1.2 \leq x_{t+1} \leq 0.5$ and $-0.07 \leq \dot{x}_{t+1} \leq 0.07$ respectively. If the car hits the wall on the left, its velocity is set to 0. See Figure 7 for a schemantic of the Mountain Car Task.

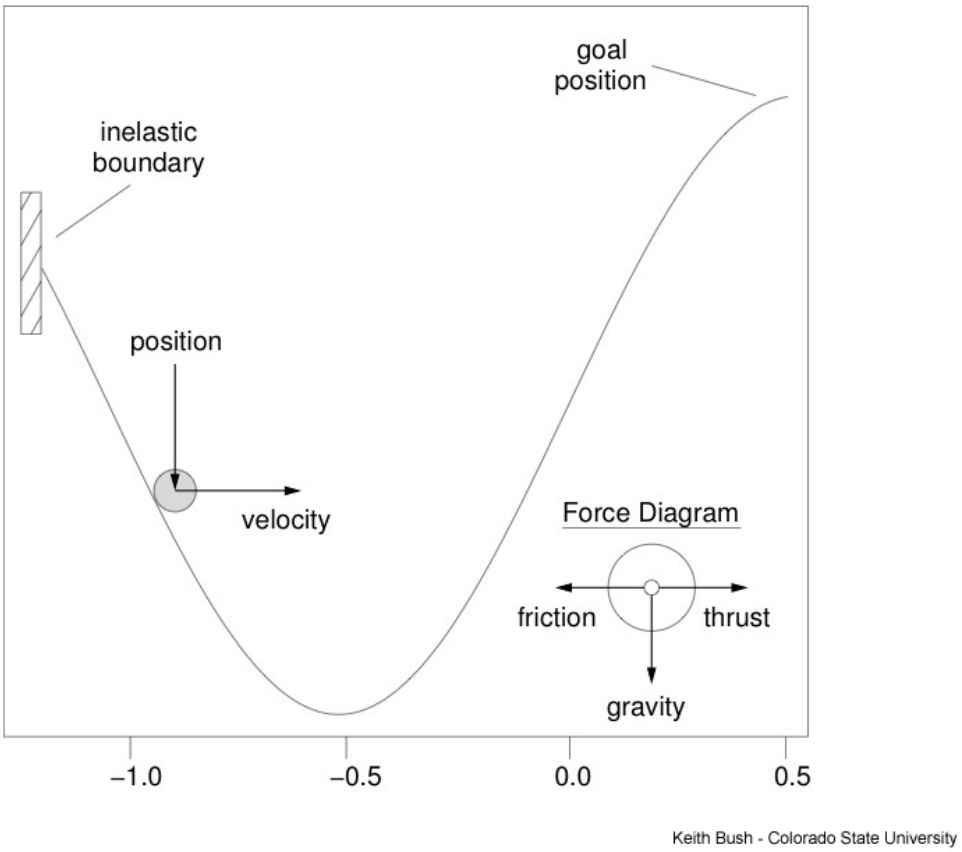


Figure 7: Schematic of the Mountain Car Task

4.1.1 The Solution

To solve this problem, Gradient Descent Q(λ) was used. The eligibility updates were done according to Watkins learning algorithm. Gaussian Softmax Basis Function Networks were used as a linear function approximation like described before. 100 NRBFs were uniform distributed over the state space with a width of 0.5. The learning rate α was set to 0.5 and the temporal credit assignment parameter, λ, was set to 0.95. Exploration can be encouraged by setting the exploration and the initial value function to zero.

Figure 8 shows the number of steps the agent needs to reach the goal, averaged over 30 episodes.

Figure 9 shows the negative of $\max_a Q(s, a)$. This function expresses the *cost-to-go* from each state.

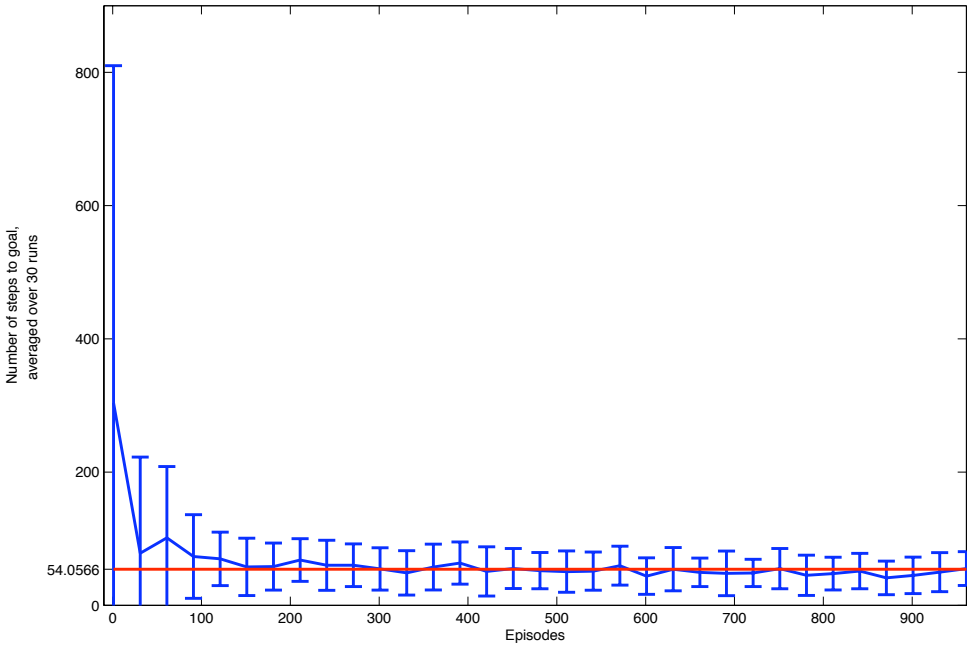


Figure 8: The Learning-curve for Q(λ)

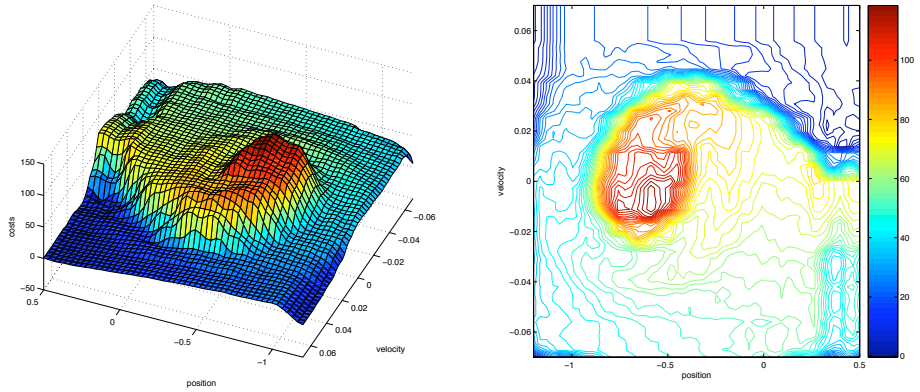


Figure 9: The cost-to-go function for the MC-Task

4.2 A simple Crawling Robot

The second benchmark is a small crawling robot first introduced in [Kimura et al., 1997]. The robot has 2 joints, one to move the arm horizontal, and the other to move it vertical. The agent's goal is to move the body forward. Putting the arm to the ground and moving it to the body could do this. This will result in a positive reward for the agent. A movement in the opposite direction will punish the agent with a negative reward.

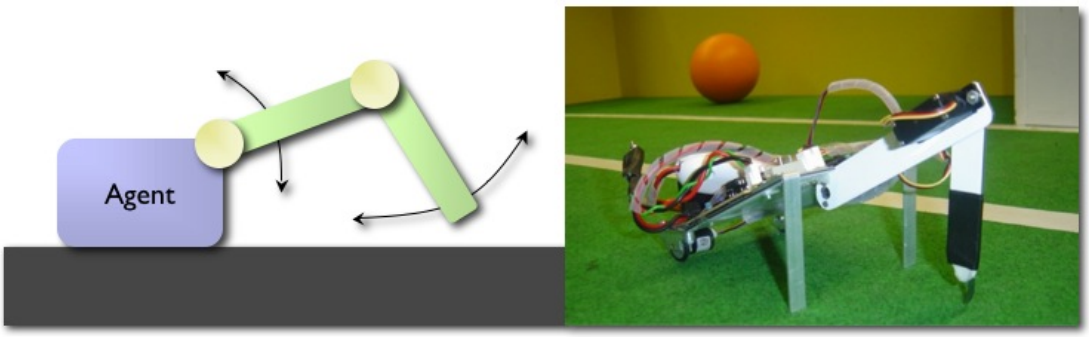


Figure 10: A small crawling robot

The state, $s = [x, y]^T$, represent the position of the tip of the arm. There are 4 possible actions for the robot in each state: right ($[1, 0]^T$), left ($[-1, 0]^T$), up ($[0, 1]^T$) and down ($[0, -1]^T$). The Robot can move its arm from 1 to 5 in both axes. The robot receives a reward of $+1$ for a movement to the left and a reward of -1 for a movement to the right. The pit hits the ground at $Y - Position < 2.5$. If the robot tries to move its arm outside the valid range, the arm will remain in the current position and a reward of -1 will be the result.

The state update is done by

$$s_{t+1} = s_t + a_t + w_t \tag{26}$$

were w_t is a noise vector to simulate inaccuracy in motor control and measurement. The noise is normally distributed with zero mean and a specified sigma value: $N(0, \sigma)$

4.2.1 The Solution

For this simple task it was sufficient to use only 30 uniform distributed NRBs with a width of 0.75 to achieve a good result.

The value function for each of the 4 actions (right, left, up and down) are drawn in Figure 11. At each position the action with the highest value is taken. For instance, at position $[5, 1]^T$ the robot will move its arm up, at position $[5, 3]^T$ the

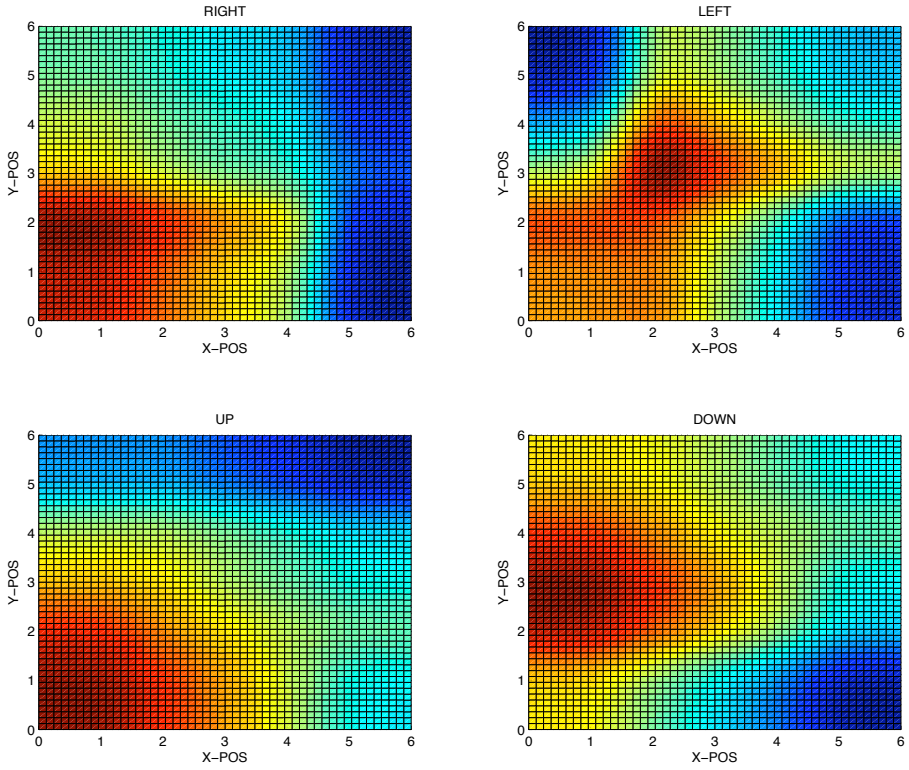


Figure 11: The 4 value functions for the Crawling Task.

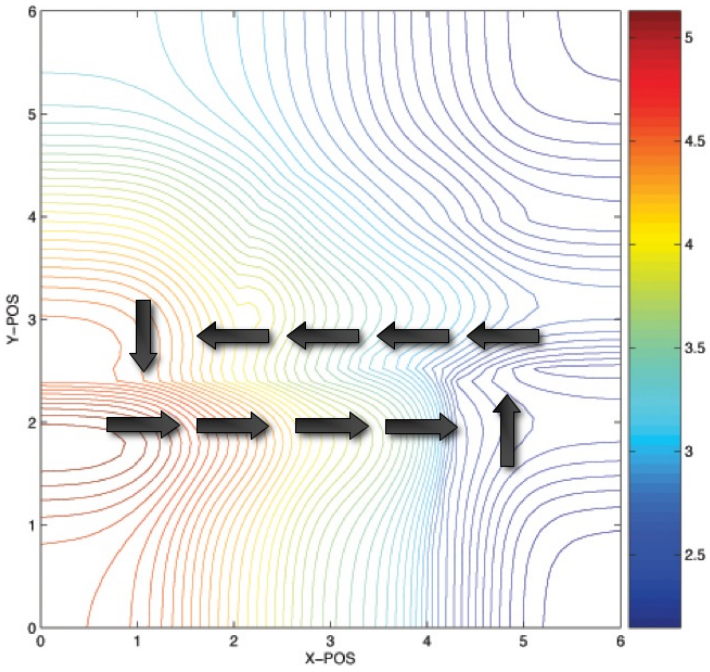


Figure 12: A typical trajectory

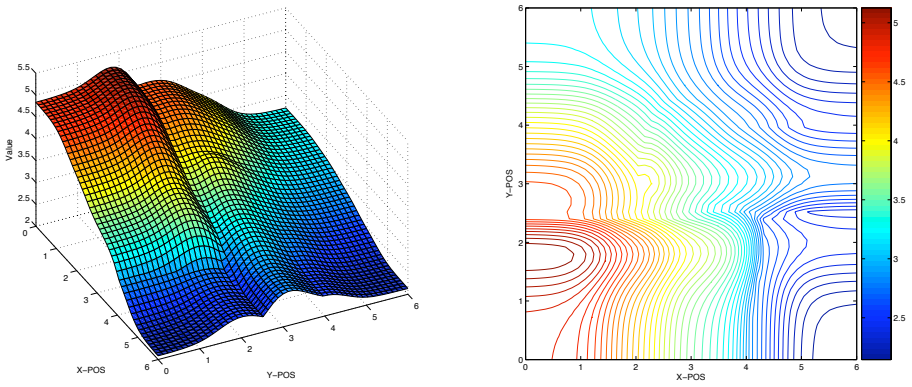


Figure 13: The $\max_a Q(s, a)$ function for the crawling robot

robot will move the arm to the left etc. Figure 12 shows a typical trajectory. We can also see the punishment for the invalid actions. This is most obvious for the movements to the right at positions $[5, *]^T$.

Figure 13 shows the value function for the optimal policy. The gab between 2 and 3 is very clear. This is due to the fact, that the robot will receive no reward if the arm is higher than 2.4.

A References

Rémi Coulom. Feedforward neural networks in reinforcement learning applied to high-dimensional motor control. In Masayuki Numao Nicoló Cesa-Bianchi and Ruediger Reischuk, editors, *Proceedings of the 13th International Conference on Algorithmic Learning Theory*, pages 402–413. Springer, 2002. URL citeseer.ist.psu.edu/coulom02feedforward.html.

Kenji Doya and Jun Morimoto. Reinforcement learning of dynamic motor sequences.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. URL citeseer.ist.psu.edu/article/kaelbling96reinforcement.html.

H. Kimura, K. Miyazaki, and S. Kobayashi. Reinforcement learning in pomdps with function approximation. 1997.

R. Kretchmar and C. Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning, 1997. URL citeseer.ist.psu.edu/article/kretchmar97comparison.html.

A. W. Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge, 1990.

Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. Technical report, College of Engineering, University of California, Riverside, CA 92521, May 1991.

Doina Precup, Richard S. Sutton, and Satinder Singh. Eligibility traces for off-policy policy evaluation. In *Proc. 17th International Conf. on Machine Learning*, pages 759–766. Morgan Kaufmann, San Francisco, CA, 2000. URL citeseer.ist.psu.edu/precup00eligibility.html.

Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1–3):123–158, 1996. URL citeseer.ist.psu.edu/singh96reinforcement.html.

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press, 1996. URL citeseer.ist.psu.edu/sutton96generalization.html.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning. An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

Christopher Watkins and Peter Dayan. Q-learning, 1992.

B List of Figures