REINFORCEMENT LEARNING WITH FUNCTION APPROXIMATION: SURVEY AND PRACTICE EXPERIENCE

Yuriy Chizhov

Department of Modelling and Simulation
Riga Technical University
Kalku 1, Riga, LV-1658, Latvia
Phone: +371-26499192. E-mail: jurij.ch@gmail.com

Classical Reinforcement Learning with tabular value function form is unable to cope successfully with real world tasks which suppose continuous or large space of states and actions. Value Function Approximation and Policy Gradient allow solving the mentioned problem. In most papers the methods are described theoretically, but suffer from the lack of details of practical part. The aim of this article is to make an overview of mentioned methods and meet a lack. For this purpose some aspects of the implementing a couple of algorithms related to Value Function Approximation are shown: Tile Coding and Gradient Descent with Back-propagation Artificial Neural Network. The Mountain Car task is used to demonstrate results of experiments of Tile Coding.

Keywords: *Reinforcement Learning, Value Function Approximation, Gradient Policy, Tile Coding, Neural Network*

1. Introduction

Simple and effective idea for intelligence agents is advised by Reinforcement learning (RL) for automated exploration of unknown environment and goal achieve. As many other AI algorithms, RL should be exceedingly upgraded to cope with real world tasks. To work with continuous or large spaces of states two basic approaches were suggested: Value Function Approximation and Gradient Policy. Both algorithms are broadly investigated, but some practical details are not clear yet, for example, the influence of tiling size in Tile Coding. In one's turn, the adaptation of Neural Networks to Reinforcement Learning is not trivial task due to requirements of the problem, its properties, selecting of activation function for each hidden layer and so on. The complexity is indorsed by the words of Richard Sutton (the expert and researcher in reinforcement learning): "It is a common error to use a back-propagation neural network as the function approximator in one's first experiments with reinforcement learning, which almost always leads to an unsatisfying failure. The primary reason for the failure is that back-propagation is fairly tricky to use effectively, doubly so in an online application like reinforcement learning" [1]. Nevertheless the method was successfully applied in a series of individual works.

The paper provides a survey of value function approximation methods and describes a few technical details gained from self experience.

2. Reinforcement Learning

Reinforcement Learning is defined as the problem of an agent that learns to perform a task through trial and error interaction with an unknown environment which provides feedback in terms of numerical reward [2]. The agent and the environment interact continually (see Fig. 1) within discrete time.

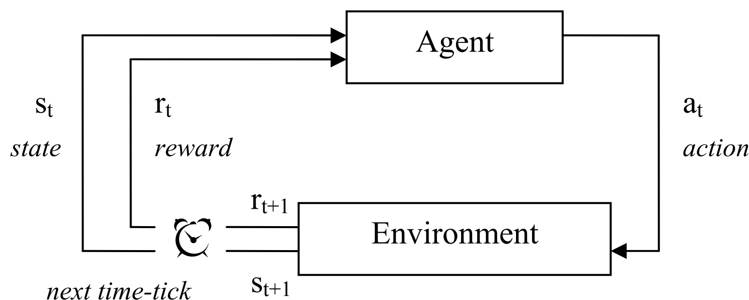

***Fig. 1**. The agent-environment interaction in reinforcement learning*

At time t the agent senses the environment to be in state s_t; based on its current sensory input s_t the agent selects an action a_t in the set A of the possible actions; then action a_t is performed in the environment. Depending on the state s_t, on the action a_t performed, and on the effects of a_t in the environment, the agent receives a scalar reward r_{t+1} and a new state s_{t+1}. The agent's goal is to maximize the amount of reward it receives from the

environment in the long run. This is usually expressed as the discounted expected payoff (or expected return as in [3]) which at time t is defined as follows:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \quad , \tag{1}$$

where γ is the discount factor ($0 \leq \gamma \leq 1$) that specifies the importance of future reward. The larger is γ, the more important and more distant future rewards.

In common case reinforcement learning algorithms use tabular functions for estimating utility (value) of current properties. Two kinds of value functions are exist in classic RL:

- functions of states – V(s) – estimates "how good" it is for the agent to be in a given state *s*;
- functions of state-actions pairs – Q(a, s) – estimates "how good" it is to perform a given action *a* in a given state *s*.

The notion of "how good" here is defined in terms of future rewards that can be expected in terms of expected return. Accordingly, value functions are defined with respect to particular policies [3]. Thus, for example, optimal value of a state is the expected infinite discounted sum of rewards which agent will gain if it starts in current state and implements optimal policy.

In reinforcement learning the agent learns how to maximize the incoming reward be developing an action-value function Q(a,s) or a state value function V(s) that maps state-action pairs or states into the corresponding expected payoff value (Equation 1).

2.1. Drawbacks of tabular RL

By the present days, researchers faced to many problems peculiar to RL (not only tabular). Most essential are the following:

- huge amount of trials – the main principle of RL require to execute certain action by agent to explore a reward for each allowable state;
- exploration and exploitation dilemma – the problem rises if the exploitation of agent is not separated from its learning. In that case the amount of exploration is another parameter including to system;
- picking up constants and parameters – usually each algorithm requires custom values of constants and parameters per each task or environment. Often it is done by expert's manual setting up;
- adopting "reality" into RL concept – obviously the problem is nature for all AI algorithms,

At last, the tabular-RL specific drawback is the "curse of dimensionality". Exactly to that problem is devoted the approximation. Classic way of value functions (state-value function or action-value function) representation in reinforcement learning is tabular form. Hence, value storing and updating is simple, intuitive and fast, in other hand, the way is only capable to cope with small number of states and actions. Simple toy tasks, like walking in grid worlds, Windy Gridworld (mentioned in [3]), Pick-and-Place Robots etc, are successfully used by researches for demonstrating principals of RL functioning. Real-world tasks, often requiring taking in account complicated physics in real time, should be neither oversimplified nor solved by other methods.

More over, it is unable to work in tasks with continuous spaces of states or actions. Due to possibly large state-action spaces, it has become clear that tabular-based reinforcement learning scales-up poorly.

For example, Q-learning's Q-table (which is $|S|\times|A|$) grows exponentially in the problem dimensions [2]. The "curse of dimensionality" implies growing of experiences required to converge to an enough estimate of the optimal V- or Q-table, and requires more memory to store the table.

2.2.2 Existing solutions

Two fundamental ways to cope with large space of states are known today wide: value function approximation and policy gradient methods. In Figure 2 the methods are shown in hierarchical structure.

First of all it is important to point out, that we can't cope with the "curse of dimensionality" simply by using local linear features. Simply because of the number of features which grows exponentially with the number of dimensions of the model state space [4].

In the function approximation technique the action-value function Q(a,s) is seen as a function that maps state-action pairs into real numbers (i.e., the expected payoff); gradient descent techniques are used to build a good approximation of function Q(a,s) from on-line experience [2]. In other words, function approximation takes examples from a desired function (in our case a value or action functions) and attempts to generalize from them to construct an approximation of the entire function [3]. So, function approximation allows represent value functions for large state spaces. We can interpret it as compressing. But the main benefit is that thanks to function approximation agent might generalize self experience from "visited" states to unknown. In [5] the authors point out some of the drawbacks of value function estimation (not including residual gradient algorithm).

Most implementations lead to deterministic policies even when the optimal policy is stochastic, meaning that probabilistic action policies are ignored even when they would produce superior performance [6]. Further, because these methods make distinctions between policy actions based on arbitrarily small value differences, tiny changes in the estimated value function can have disproportionately large effects on the policy.

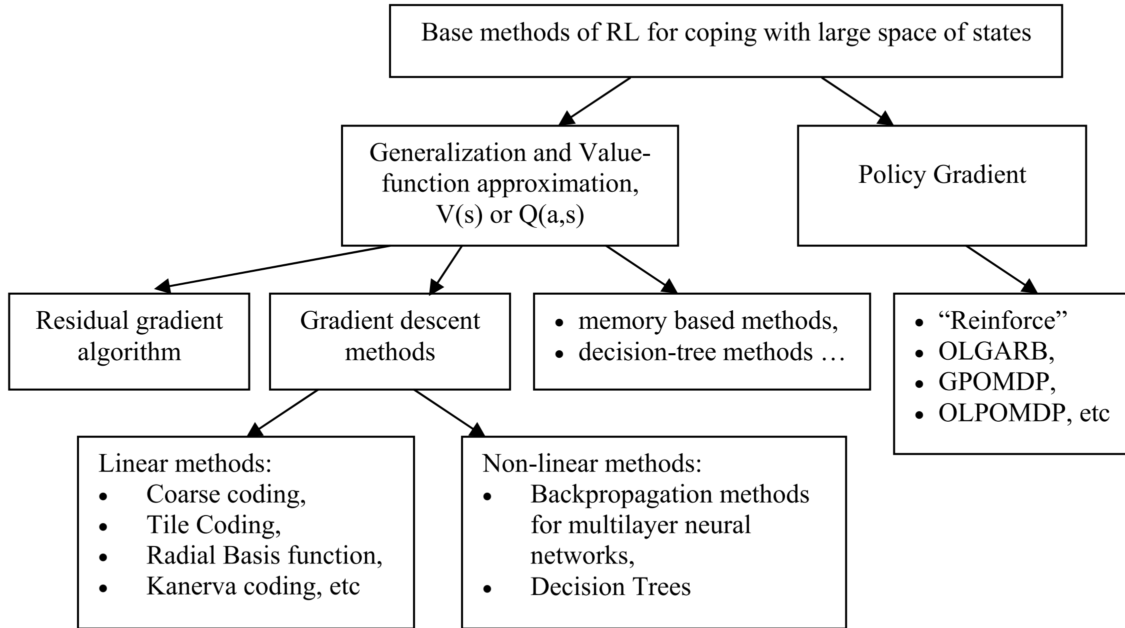

Fig. 2. *Algorithms and methods to cope with continuous or large spaces of states*

In turn a policy-gradient approach is able to bypass drawbacks of directly mentioned above. In this approach, instead of learning an approximation of the underlying value function and basing the policy on the expected reward indicated by that function, policy-gradient learning algorithms maximize the long-term expected reward by searching the policy space directly. In addition, being able to express stochastic optimal policies and being robust to small changes in the approximation, under certain conditions policy gradient algorithms are guaranteed to converge to an optimal solution [7], [8].

Interesting is that value function approximation despite some theoretical drawbacks (mentioned above), demonstrates in practice greatly better results than policy gradient. Exhaustive experiment named "*Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout*" executed by [6] demonstrates that Sarsa(λ) armed with function approximation is able perform better than OLGARB[1] in continuous, stochastic, partially-observable, competitive multi-agent environment.

The residual gradient algorithms (proposed in [9]) are a new class of algorithms, which perform gradient descent on the mean squared Bellman residual, guaranteeing convergence. It is shown, however, that they may learn very slowly in some cases.

Let's consider a special case of gradient-descent function approximation when approximate function $V_t(s)$ (which is value of state s) is a linear function of the parameter vector $\vec{\theta}_t$. The general gradient-descent method for state value prediction is as follows:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[v_t - V_t(s_t)]\nabla_{\vec{\theta}_t} V_t(s_t), \tag{2}$$

where α is a positive step-size parameter, and v_t is target output of the *t*-th training example. For details see [3]. Corresponding to every state s, there is a column vector of features $\vec{\phi}_s = (\phi_s(1), \phi_s(2), \ldots \phi_s(n))^T$. Thus, the linear approximate state-value function is given by

$$V_t(s) = \vec{\theta}_t^T \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\phi_s(i). \tag{3}$$

[1] OLGARB is initials from "On-Line GPOMDP with an Average Reward Baseline", in one's turn GPOMDP is initials from "Gradient of a Partially Observable Markov Decision Process".

Due to linear case the gradient of the approximate value function with respect to $\vec{\theta}_t$ simply is

$$\nabla_{\vec{\theta}_t} V_t(s_t) = \vec{\phi}_s . \tag{4}$$

Finally, our goal is to find the parameter vector $\vec{\theta}$. In one's turn to convert state into features representation the Tile Coding is used. Notice that for control tasks the action-value $Q_t(s_t, a_t)$ is used instead of $V_t(s_t)$.

3. Tile Coding Implementation

Tile Coding is an algorithm of generalization value function having linear representation by a set of parameters. In our case the software implements on-policy Sarsa(λ) control method using linear, gradient-descent function approximation with binary features via Tile Coding. Some parts of the software are based on [3] works.

Let's see several details how to implement value function approximation by Tile Coding algorithm. We will use Mountain Car as the experimental task due to some difficulty: gravity is stronger than the car's engine and even at full throttle the car cannot accelerate up the steep slope when starting (at zero velocity) at the bottom. The only solution is at first to move away from the goal and up the opposite slope on the left. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. The actions available to the car are full throttle forward (+1), full throttle reverse (-1) and zero throttle (0). The car moves according to a simplified physics.

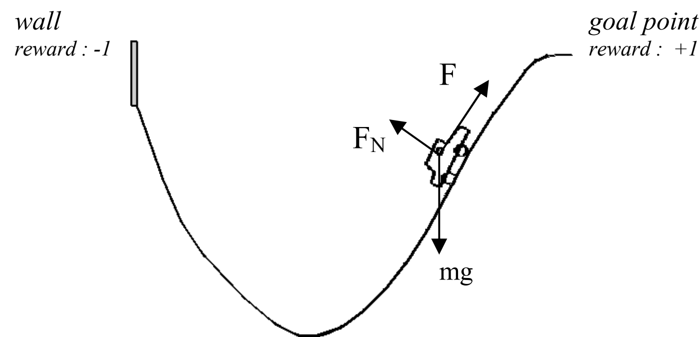

Fig. 3. *The Mountain Car task*

Its position x_t and velocity v_t are updated by:

$$x_{t+1} = bound[x_t + v_{t+1}]$$
$$v_{t+1} = bound[v_t + 0.001a_t - 0.0025\cos(3x_t)],$$

where the bound operation enforces $-1.2 \le x_{t+1} \le 0.5$ and $-0.07 \le v_{t+1} \le 0.07$. When x_{t+1} reaches the left bound it has crashed into the wall and its velocity v_{t+1} is reset to 0. When x_{t+1} reach the right boundary it has reached the goal and the episode is terminated.

The central idea of Tile Coding is that the all continuous space of search (bounded by task's parameters) is divided on pieces called tiles. In other words, each tile represents corresponding feature $\phi_s(i)$. Each tile have own weight. There might be (and should to be!) different ways of partitioning, thus each partition calls tiling. Due to it, the tiling is overlapped (see Figure 4, for example). For a given point in a search space the approximate value is sum of the weights of the tiles (one per tiling, in which it is contained). A number of equal tiling overlapped with offset (shift) usually is enough to avoid generating of different tiling. It well simplifies the implementation. The square on the search space (for 2-dimension case) bounded by each tiling calls resolution. The resolution is one of the generalization's significant properties.

It is important to point out, that high resolution is not a guarantee of best result. The explanation is given below in the text. Nevertheless the Tile Coding successfully deals with continuous variables (a proof sketch is in [10]).

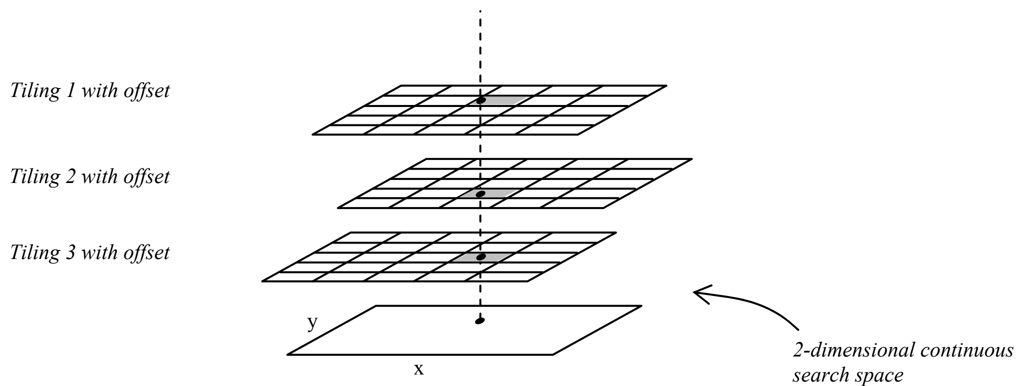

Fig. 4. *Example of overlapped tiling for 2-dimensional space*

Using built software additionally to research of [3] let's investigate dependence of convergence on tiling partitioning. The experiment supposes practically find the optimal tiling partitioning for obtaining the policy which fast (with minimal number of agent's actions) leads the agent to the goal point. Figure 5 represents last 19 observations (total 100) of 14 different partitioning (tiling).

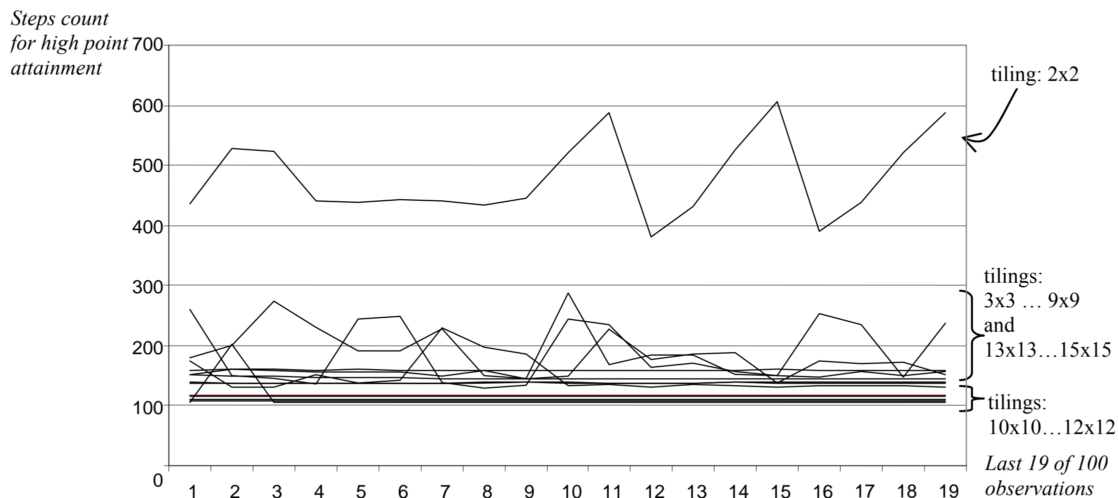

Fig. 5. *Influence of discrimination on time of convergence*

Thus, according to Figure 5, the speed of convergence is not in linear dependence on size of tiling discretization. Most optimal values are 10x10, 11x11 and 12x12. In the same time 8x8, 9x9,13x13, 14x14 unable to give best convergence. Such peculiarity occurs due to result accuracy of approximation to desired function.

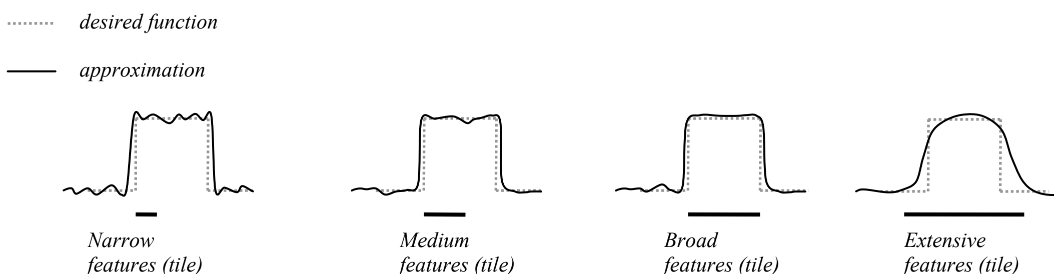

Fig. 6. *Example of features width's*

The width of feature (tile) should be selected taking in account the nature of the desired function. If it is impossible, then the work [10] purposed to automated parameter choosing will be helpful.

4. Gradient Descent with Back-propagation Neural Network

Main motivations to choose the artificial neural networks in task of approximation value function are ability to work with non-linear functions and simple adaptation of features.

Back-propagation property of neural network is used to compute the gradient of the squared TD(λ) error with respect to the network weights. According to [3] the backward view of the action-value method the following expressions take a place. The equation (in terms of RL) of multi-layered perceptron becomes the following:

$$Q(s_t, a_t^k) = g\left(\sum_{j=0}^{N} \theta_i g(\sum_{i=0}^{M} \theta_{ij} s_i) + \theta_0 b_0 \right), \tag{5}$$

where g is activation function of perceptron, a_t^k – an action *k*.

The update of gradient presented in terms of eligibility traces is the following:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \delta_t \vec{e}_t , \tag{6}$$

where δ_t is the TD-error and computing as:

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) . \tag{7}$$

Finally eligibility traces becomes to

$$\vec{e}_t = \gamma\lambda \vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t, a_t) = \gamma\lambda \vec{e}_{t-1} + \frac{\partial Q_t(s_t, a_t)}{\partial \vec{\theta}_t} , \tag{8}$$

where α – learning rate, γ – discount rate. Initial value $e_0 = 0$.

The weights of neural networks represent the vector $\vec{\theta}$ (parameter vector). By adjusting the weights, any of a wide range of different functions Q_t (or V_t) can be implemented by the network [3]. The input of the network is the state s_t. Usually the input layer size is equal to state variables count. It works only for discrete state representation. Thus, for 2-dimension task with M×N states the network should be equipped with M+N input neurons. In task with continuous state spaces this way of representation does not satisfy. Using of a gaussian distribution over the input nodes is alternative representation of continuous input state [11].

The output of network is a value of action-value function Q_t. Often output layer size is equal to number of available actions if output neurons are coded in binary mode. Thus, each output neuron is interpreting as a flag to implement or not corresponding action for a current state. Some labours advice to use the same number of networks as many the actions are possible [12] (action per network). The structure of neural network for 1 output value might be implemented as it presented in Figure 7.

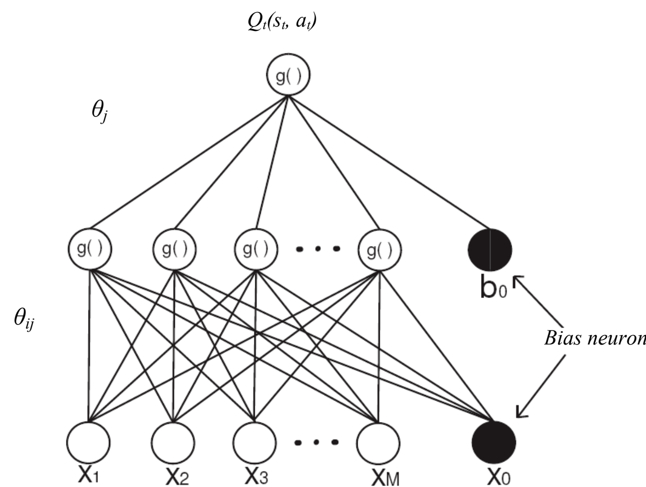

Fig. 7. *The multi-layered perceptron*

As usually, size of hidden layer is free for experiments and searching an optimal value between speed of convergence, occupied memory and quality of output value. The preferred activation function is sigmoid function:

$$g(a) = \frac{1}{1+e^{-a}}, \tag{9}$$

which is the most used activation function for back-propagation networks partly due to its simple derivative. The activation function g(a) is normally a monotonic and non-linear function. Algorithm of combining RL with function approximation is described in [3].

5. Conclusions

In this paper, we have discussed the problem of value function and action-value function approximation in Reinforcement Learning. The problems of tabular form of value function are described. To deal with them, two base methods are expounded: Tile Coding and Gradient Descent with Back-propagation Artificial Neural Network. Both methods successfully deals with continuous state space, nevertheless Tile Coding suffer of "curse of dimensionality". In one's turn binary coding of input layer neurons of neural network might be replaced by gaussian distribution over the input nodes to deal with continuous space.

In Tile Coding, the accuracy of approximating might be increased by tuning up of tiles size. Too large or too small partitioning of state space causes slack approximation (see Figure 5).

Value approximation gives opportunity to RL to be implemented in real-world tasks. In a certain sense a serious work should be done before a turn of agent's policy exploitation starts. State and action description should be transformed to corresponding method's input structure. For finding the optimal working parameters a mass of experiments should be done.

References

1. Sutton, R. *Frequently Asked Questions about Reinforcement Learning* – http://www.cs.ualberta.ca/~sutton/RL-FAQ.html. Initiated 2001.08.13. Last updated 2004.04.02. Visited 2008.04.03.
2. Butz, M.V., Goldberg, D.E., Lanzi, P.L. Gradient Descent Methods in Learning Classifier Systems: Improving CXS Performance in Multi-step Problems, *Evolutionary Computation, IEEE Transactions,* Vol. 9, Issue 5, Oct. 2005, pp: 452-473.
3. Sutton, R.S., Barto, A.G. *Reinforcement Learning. An Introduction*. Cambridge, MA: MIT Press, 1998. 342p.
4. Baxter, J., Bartlett, P.L. *Direct Gradient-Based Reinforcement Learning: I. Gradient Estimation Algorithms.* Research School of Information Sciences and Engineering, Australian National University, July 29, 1999. 24 p.
5. Sutton, R.S., McAllester, D., Singh, S., Mansour, Y. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: *Advances in Neural Information Processing Systems 12,* Cambridge, MA: MIT Press, 2000. pp. 1057-1063.
6. Beitelspacher, J., Fager, J., Henriques, G. and Amy McGovern. *Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout: Technical Report No. CS-TR-06-001*. School of Computer Science University of Oklahoma Norman, OK 73019, Feb. 2006.
7. Fager, J. *Online Policy-Gradient Reinforcement Learning using OLGARB for Space-War*. University of Oklahoma, 660 Parrington Oval, Norman, OK 73019 USA, 2006.
8. Baxter, J., Bartlett, P. L. Infinite-horizon policy-gradient estimation, *Journal of Artificial Intelligence Research*, Vol. 15, Nov. 2001, pp. 319-350.
9. Baird, L. *Residual Algorithms: Reinforcement Learning with Function Approximation*. Department of Computer Science, U.S. Air Force Academy, CO 80840-6234. 1995.
10. Sherstov, A.A., Stone, P. Function Approximation via Tile Coding: Automating Parameter Choice. In: *Symposium on Abstraction, Reformulation, and Approximation (SARA-05)*. Edinburgh, Scotland, UK, 2005, p. 12.
11. Bishop, Ch.M. *Neural Networks for Pattern Recognition.* USA: Oxford University Press, 1995, p. 504.
12. Jakša, R., Sinčák, P., Majerník, P. Back-propagation in Supervised and Reinforcement Learning for Mobile Robot Control. In: *Computational Intelligence for Modelling, Control & Automation (CIMCA'99)*. Vienna, Austria, 1999, p. 6.