Multilayer Perceptrons with Radial Basis Functions as Value Functions in Reinforcement Learning

Victor Uc Cetina

Humboldt University of Berlin - Department of Computer Science
Unter den Linden 6, 10099 Berlin - Germany

Abstract. Using multilayer perceptrons (MLPs) to approximate the state-action value function in reinforcement learning (RL) algorithms could become a nightmare due to the constant possibility of unlearning past experiences. Moreover, since the target values in the training examples are bootstraps values, this is, estimates of other estimates, the chances to get stuck in a local minimum are increased. These problems occur very often in the mountain car task, as showed by Boyan and Moore [2]. In this paper we present empirical evidence showing that MLPs augmented with one layer of radial basis functions (RBFs) can avoid these problems. Our experimental testbeds are the mountain car task and a robot control problem.

1 Introduction

Reinforcement learning [9] is a very appealing artificial intelligence method to approach the machine learning problem. The idea of programming a computational system in such a way that it could improve its performance through several interactions with the environment is certainly attractive. In relatively small problems with discrete state and action spaces using a lookup table and algorithms like TD(λ) [8], Q-learning [12] or Sarsa [10] should be enough to get optimal results. Of course, we need to find the best set of parameters and allow for enough training episodes. The challenging part in reinforcement learning comes when we try to solve more complicated problems involving continuous spaces, and particularly high dimensional ones. Then, a lookup table is not enough to represent the value function and we need to approximate it somehow. When we get to this point, we have to decide between using a linear or a non-linear method. Linear methods like the cerebellar model articulation controllers (CMACs) [10, 5] and RBFs networks of gaussian functions [1, 3] are by far the most recommended methods for RL, primarily because they are localised function approximators and therefore they are less affected by the unlearning problem. Kretchmar and Anderson [4] studied the similarities and differences between CMACs and RBFs with Q-learning applied to the mountain car task. Another option worth mentioning is the use of regression trees like in the method proposed by Wang and Dietterich [11], although it should be noted its limited applicability for tasks where incremental learning is required.

In this paper we present experimental results showing how a non-linear function approximator like the MLP augmented with a RBFs layer could become

a good choice to represent the state-action value function in RL problems with continuous state spaces and high dimensionality. We tested this approach in the mountain car task, which is well known as a tricky control problem, especially for neural networks, as demonstrated by Boyan and Moore [2]. We also experimented with the dribbling problem in the framework of the RoboCup competitions.

The rest of this paper is organized as follows. In Section 2 we present the Sarsa algorithm and the learning structure we propose to approximate the value function. In Sections 3 and 4 we describe the experiments performed with the mountain car task and the dribbling problem respectively. Finally, we present our conclusion in Section 5 and comment about our future work.

2 Algorithm and Value Function Structure

Sarsa is an on-policy temporal difference control algorithm which continually estimates the state-action value function Q^π for the behavior policy π, and at the same time changes π toward greediness with respect to Q^π [9]. In problems with a small number of state-action pairs and discrete spaces, the Q function is stored using a lookup table. However, when the number of those pairs grows, the use of lookup tables becomes impractical, or simply impossible. We need a function approximator instead. In our case, the Q function is represented with a set of MLPs, one MLP per action. The Sarsa algorithm with the changes needed to use a set of MLPs as function approximator is presented in Algorithm 1.

Algorithm 1: Sarsa algorithm for continuous states using MLPs

```
1  initialize the weights vector W_i for all MLP_i arbitrarily
2  foreach training episode do
3  |   initialize s
4  |   choose a from s using policy derived from Q
5  |   repeat for each step of episode
6  |   |   take action a, observe r, s'
7  |   |   choose a' from s' using policy derived from Q
8  |   |   TargetQ ← MLP_a(s) + α[r + γ MLP_a'(s') − MLP_a(s)]
9  |   |   train MLP_a with example (s, TargetQ)
10 |   |   s ← s' ; a ← a'
11 |   until s is terminal
12 end
```

The use of MLPs as value function approximators in reinforcement learning is usually not recommended, given that they suffer from the unlearning problem and fall into local optima very often. However, if we add a layer of radial basis functions to the standard MLP, it is possible to create a semi-localised function approximator that can be used to obtain optimal policies in hard problems with continuous state spaces and high dimensionality. The proposed MLP has 4

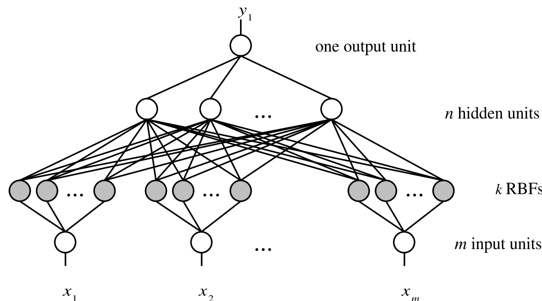

Fig. 1: Multilayer perceptron with one layer of radial basis functions

layers: 2 hidden layers plus the input and output layers (see Fig. 1). The number m of input units must equal the size of the feature vector that represents the current state of the environment. In the first hidden layer there are k RBFs. For each input variable x_i there is a set R_i of RBFs r_{ij}. The $r_{ij} \in R_i$ should be defined to cover the range of values that x_i can take. The outputs of the RBFs layer are fed into the second hidden layer that consists of n sigmoidal functions. Finally, the outputs of the second hidden layer reach the output unit. During the training stage, only the connection weights between both hidden layers, and between the second hidden layer and the output layer are learned, leaving the weights between the input and first hidden layer set to 1. Although one possibility when working with radial basis functions is the optimization of their parameters through the application of unsupervised learning methods, in the results presented here, we only experimented with the number of radial basis functions needed to learn the Q value function. We used gaussian functions of the form:

$$\text{RBF}(x_i) = \exp\left(-\frac{\|x_i - c_{i,j}\|^2}{2\sigma^2}\right)$$

The centers $c_{i,j}$ of the b_i basis functions defined for x_i are placed at a distance $dist_i$ one from the other, where

$$dist_i = \frac{\max(x_i) - \min(x_i)}{b_i} \quad \text{and} \quad \sigma_i = \frac{dist_i}{2}$$

Comprehensive introductions to radial basis functions and their training can be found in [1, 3]. The main advantage of our topology is that it can be used with high dimensional state spaces without problems of exponential grow in the number of RBFs. This is, in the case of having the same number p of RBFs for each one of the m input variables, we would need only mp RBFs, in contrast to the p^m we would use in a straightforward implementation of RBF networks. One common option to avoid the curse of dimensionality is to group the input variables in pairs, and define the number of RBFs required to cover the resulting 2-dimensional subspaces generated by each pair. However, the successful selection of the variable pairs requires some previous knowledge about

the input space of the problem, or an important amount of experimentation instead.

3 Mountain Car Problem

Our first testbed is the mountain car problem, where a car is driving along a mountain road and it must drive up a hill. However, the engine is too weak to directly go up the slope. This problem is commonly used as a testbed in reinforcement learning, and a complete description of it and its dynamics, are given by Sutton and Barto [9].

3.1 Experiments and Results

For this problem we experimented with 2, 6, 8 and 12 RBFs for each input variable, and 2 sigmoidal units in the second hidden layer. The best results were obtained with 12 RBFs and 50,000 training episodes, as it is illustrated in Fig. 2a. Each training episode was terminated either when the goal was reached, or when 100 movements were performed. The reward function penalizes the actions with -0.1 all the time, except when the last action performed allowed the car to reach the goal, in this case the reward is 0. The training policy was ϵ-greedy with a constant $\epsilon = 0.01$, $\alpha = 0.5$ and $\gamma = 0.5$. In terms of the MLPs we used $\alpha_{\mathrm{MLP}} = 0.001$ and activation functions with outputs in the interval $(-1, 1)$.

Some of our best policies were able to reach the goal in 59 steps, however in average the goal is reached in 63 steps. The quality of our solution is similar to those presented by Smart and Kaelbling [7], and more recently by Whiteson and Stone [13]. Moreover, given the great similarity between the shape of our final value function presented in Fig. 2b and the best one provided by Singh and Sutton [6, 10], we conclude that our solution is a near-optimal policy.

4 Dribbling Problem

In the RoboCup simulation league, one of the most difficult skills that the robots can perform is dribbling. Dribbling can be defined as the skill that allows a player to run on the field while keeping the ball always in its kick range. In order to accomplish this skill, the player must alternate run and kick actions. The run action is performed through the use of the command (**dash** *Power*), while the kick action is performed using the command (**kick** *Power Direction*), where $Power \in [-100, 100]$ and $Direction \in [-180, 180]$. There are three factors that make this skill a difficult one to accomplish. First, the simulator adds noise to the movement of objects, and to the parameters of commands. This is done to simulate a noisy environment and make the competition more challenging. Second, since the ball must remain close to the robot without collisioning with it, and at the same time it must be kept in the kick range, the margin for error is small. And third, the most challenging factor, the use of heterogeneous players during competitions. Using heterogeneous players means that for each game the simulator generates seven different player types at startup, and the eleven players