Least-Squares Temporal Difference Learning based on Extreme Learning Machine

Pablo Escandell-Montero, José M. Martínez-Martínez, José D. Martín-Guerrero, Emilio Soria-Olivas, Juan Gómez-Sanchis *

IDAL, Intelligent Data Analysis Laboratory, University of Valencia Av. de la Universidad, s/n, 46100, Burjassot, Valencia - Spain

Abstract. This paper proposes a least-squares temporal difference (LSTD) algorithm based on extreme learning machine that uses a singlehidden layer feedforward network to approximate the value function. While LSTD is typically combined with local function approximators, the proposed approach uses a global approximator that allows better scalability properties. The results of the experiments carried out on four Markov decision processes show the usefulness of the proposed approach.

1 Introduction

Reinforcement learning (RL) is a machine learning method for solving decisionmaking problems where decisions are made in stages [1]. This class of problems are usually modelled as Markov decision processes (MDPs) [2]. Value prediction is an important subproblem of several RL algorithms that consists in learning the value function V^{π} of a given policy π ; a widely used algorithm for value prediction is least-squares temporal-difference (LSTD) learning [3, 4]. LSTD has been successfully applied to a number of problems, especially after the development of the least-squares policy iteration algorithm [5], which extends LSTD to control by using it in the policy evaluation step of policy iteration.

LSTD assumes that value functions are represented with linear architectures, $V^{\pi}(s)$ is approximated by first mapping the state s to a feature vector $\phi(s) \in \mathbb{R}^k$, and then computing a linear combination of those features: $\phi(s)^{\top}\beta$, where β is the coefficients vector. The selection of an appropriate feature space is a critical element of LSTD. When a deep knowledge of the problem is available, it can be used to select a suitable ad-hoc set of features. However, in general the state space is divided in a regular set of features using for example state aggregation methods or radial basis function (RBF) networks with fixed bases. Most of these techniques are local approximators, i.e., a change in the input space only affects a localized region of the output space. One of the major potential limitations of local approximators is that the number of required features grows exponentially with the input space dimensionality [6].

This paper studies the use of extreme learning machine (ELM) together with LSTD. ELM is an algorithm recently proposed in [7] for training single-hidden layer feed forward networks (SLFNs). ELM works by assigning randomly the weights of the hidden layer and optimizing only the weights of the output layer

^{*}This work was partially financed by MINECO and FEDER funds in the framework of the project EFIS: Un Sistema Inteligente Adaptativo para la Gestión Eficiente de Energía en Grandes Edificios, with reference IPT-2011-0962-920000.

through least-squares. This procedure can be seen as a mapping of the inputs to a feature space, where features are defined by the hidden nodes, and then computing the weights that linearly combine the features. Therefore, ELM can be combined with LSTD to solve value prediction problems. The main advantage of this approach is its good scalability to high dimensional problems due to the global nature of SLFNs.

2 Extreme Learning Machine

Extreme learning machine (ELM) is an algorithm for training SLFNs where the weights of the hidden layer can be initialized randomly, thus being only necessary the optimization of the weights of the output layer by means of standard least-square methods [7].

Let us consider a set of N patterns, $\mathcal{D} = (\mathbf{x}_i, \mathbf{o}_i); i = 1 \dots N$, where $\{\mathbf{x}_i\} \in \mathbb{R}^{d_1}$ and $\{\mathbf{o}_i\} \in \mathbb{R}^{d_2}$, so that the goal is to find a relationship between $\{\mathbf{x}_i\}$ and $\{\mathbf{o}_i\}$. If there are M nodes in the hidden layer, the SLFN's output for the *j*-th pattern is:

$$\mathbf{y}_{\mathbf{j}} = \sum_{k=1}^{M} h_k \cdot f\left(\mathbf{w}_k, \mathbf{x}_j\right) \tag{1}$$

where $1 \leq j \leq N$, \mathbf{w}_k stands for the parameters of the k-th element of the hidden layer (weights and biases), h_k is the weight that connects the k-th hidden element with the output layer and f is the function that gives the output of the hidden layer; in the case of MLP, f is an activation function applied to the scalar product of the input vector and the hidden weights. Equation (1) can be expressed in matrix notation as $\mathbf{y} = \mathbf{G} \cdot \mathbf{h}$, where **h** is the vector of weights of the output layer, **y** is the output vector and **G** is given by:

$$\mathbf{G} = \begin{pmatrix} f(\mathbf{w}_1, \mathbf{x}_1) & \dots & f(\mathbf{w}_M, \mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ f(\mathbf{w}_1, \mathbf{x}_N) & \dots & f(\mathbf{w}_M, \mathbf{x}_N) \end{pmatrix}$$
(2)

Then, the weights of the output layer can be computed as $\mathbf{h} = \mathbf{G}^{-1} \cdot \mathbf{o}$ using the Penrose-Moore pseudoinverse to invert \mathbf{G} robustly.

3 Least-squares temporal-difference learning

Temporal difference (TD) is probably the most popular value prediction algorithm [1]; well-known control algorithms like Q-learning or SARSA are based on TD. It uses bootstrapping: predictions are used as targets during the course of learning. Let us assume that the value of state s, $V^{\pi}(s)$, is estimated by first mapping s to a feature vector $\phi(s)$, and then combining linearly these features using a coefficients vector β , denoted by $V^{\pi}_{\beta}(s)$. Then, for each state on each observed trajectory, TD incrementally adjusts the coefficients of V^{π}_{β} toward new target values. Let $V^{\pi}_{\beta_t}(s)$ denote the value estimate of state s at time t, in the t^{th} step TD performs the following calculations [1]:

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t + \alpha_t (r_t + \gamma V_{\boldsymbol{\beta}_t}^{\pi}(s_{t+1}) - V_{\boldsymbol{\beta}_t}^{\pi}(s_t))$$
(3)

ESANN 2013 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 24-26 April 2013, i6doc.com publ., ISBN 978-2-87419-081-0. Available from http://www.i6doc.com/en/livre/?GCOI=28001100131010.

where r_{t+1} is the observed reward, γ is the discount factor, and α_t is the learning step-size sequence. TD has been shown to converge to a good approximation of V^{π} under some technical conditions [1]. It can be observed that, after an observed trajectory (s_0, s_1, \ldots, s_L) , the changes made by the update rule of Equation (3), have the form $\beta = \beta + \alpha_n (\mathbf{d} + \mathbf{C}\beta + \boldsymbol{\omega})$, where

$$\mathbf{d} = \mathbf{E} \left\{ \sum_{i=0}^{L} \phi(s_i) r_i \right\}; \quad \mathbf{C} = \mathbf{E} \left\{ \sum_{i=0}^{L} \phi(s_i) (\gamma \phi(s_{i+1}) - \phi(s_i))^\top \right\}; \qquad (4)$$

and $\boldsymbol{\omega}$ = zero-mean noise [4]. It has been shown in [8] that $\boldsymbol{\beta}$ converges to a fixed point $\boldsymbol{\beta}_{\text{TD}}$ satisfying $\mathbf{d} + \mathbf{C}\boldsymbol{\beta}_{\text{TD}} = \mathbf{0}$.

The least-squares temporal difference (LSTD) algorithm also converges to the same coefficients β_{TD} , but instead of performing some kind of gradient descent, LSTD builds explicit estimates of a constant multiples of the **C** matrix and **d** vector. Then, it solves $\mathbf{d} + \mathbf{C}\beta_{\text{TD}} = \mathbf{0}$ directly. LSTD uses the following data structures to build from experience the matrix **A** (of dimension $K \times K$, where K is the number of features) and the vector **b** (of dimension K):

$$\mathbf{b} = \sum_{i=0}^{L} \phi(s_i) r_i; \qquad \mathbf{A} = \sum_{i=0}^{L} \phi(s_i) (\gamma \phi(s_{i+1}) - \phi(s_i))^{\top}$$
(5)

After *n* independent trajectories, **b** and **A** are unbiased estimates of *n***d** and $-n\mathbf{C}$ respectively [4]. Therefore, $\boldsymbol{\beta}_{\text{TD}}$ can be computed as $\mathbf{A}^{-1}\mathbf{b}$.

In comparison with TD, LSTD improves data efficiency and, in addition, eliminates the learning step-size parameter α_t [4].

3.1 LSTD algorithm based on ELM

An important requirement of LSTD is that the method used to approximate the value function should compute its output as a linear combination of a set of fixed features. Although there are many methods that fulfil this requirement, other powerful methods cannot be combined with LSTD.

SLFN is a well-known function approximator that has been successfully applied in many machine learning tasks and that, in principle, cannot be used together with LSTD algorithm. However, when ELM algorithm is employed to train SLFNs, the training process is equivalent to map the input space into a set of fixed features through a non-linear transformation defined by the hidden nodes, and then computing the weights of the output layer through least squares. Thus, a SLFN can be employed to learn value functions in a LSTD scheme when it is trained using ELM. The pseudocode of the proposed method is shown in Algorithm 1.

4 Experiments and results

In this section, the performance of the proposed LSTD algorithm based on ELM is compared with the classical LSTD combined with RBF features. Experiments

Algorithm 1 LSTD learning based on ELM

Input: Policy π to be evaluated, discount factor γ 1: Initialize randomly \mathbf{w} (weights and biases corresponding to the hidden nodes of the SLFN) 2: Let $\phi(\mathbf{x}) : \mathbf{x} \to \mathbf{f}(\mathbf{w}, \mathbf{x})$ denote the mapping from an input \mathbf{x} to the output of the SLFN's hidden layer 3: Set $\mathbf{A} = 0$; $\mathbf{b} = 0$; t = 04: repeat 5: Select a start state s_t 6: while $s_t \neq s_{\text{END}}$ do 7Apply policy π to the system, producing a reward r_t and next state s_{t+1} $\mathbf{A} = \mathbf{A} + \phi(s_t)(\gamma\phi(s_{t+1}) - \phi(s_t))$ 8: 9: $\mathbf{b} = \mathbf{b} + \phi(s_t)r_t$ 10: t=t+1end while 11: 12: until reaching the desired number of episodes 13: $\boldsymbol{\beta} = \mathbf{A}^{-1}\mathbf{b}$ 14: output $V^{\pi}(s) \approx \phi(s)^{\top} \beta$

are carried out in a modified version of the MDP used in [4], pictured in Fig. 1.a. In its original form, the MDP contains m states, where state 0 is the initial state for each trajectory and state m is the absorbing state. Each non-absorbing state has two possible actions; both actions bring the current state closer to the absorbing state, but the step made by each action is different (see Fig. 1.a). All state transitions have reward -1 except the transition from state m - 1 to state m, which has a reward of -2/3.

The state space of the original MDP is defined by one dimension, but we are interested in evaluating the proposed algorithm in MDPs with different dimensionality. Therefore, a generalization of the original MDP to a *d*-dimensional space is proposed. For each new dimension, two more possible actions are added to the action set. The initial state is located at one extreme of the state space, while the absorbing state is in the opposite extreme. As in its original form, the reward is -1 for all state transition except for the transitions that reach directly the absorbing state. Fig. 1.b shows the resulting MDP for the case of d = 2. Besides generalization to *d*-dimensions, the discrete MDP is transformed into continuous. In the continuous version, each state variable can take values into the range [0, 1], thus, the state space is not discrete anymore. Similar to the



Fig. 1: Discrete MDP for dimensionality 1 (a) and 2 (b).

discrete case, for each dimension there are two possible actions that bring the current state closer to the absorbing state in a quantity *step* or $2 \times step$, where *step* is a parameter defined by the user. Furthermore, each action is perturbed by independent and identically distributed Gaussian noise; the noise amplitude was $0.2 \times step$. The initial and absorbing states are also in the two extremes of the state space; e.g., for dimensionality d = 3, the initial state is defined by $s_0 = [0, 0, 0]$ and the absorbing state by $s_{\text{END}} = [1, 1, 1]$.

Both methods, LSTD-ELM and LSTD-RBF are used to evaluate a policy π in a total of 4 MDPs whose dimensionality varies from 1 to 4, where π consists in selecting all possible actions with the same probability. In all MDPs the discount factor was set to $\gamma = 0.85$. The performance of each method is measured in terms of the mean absolute error (MAE). Similar to [4], MAE has been measured against a "gold standard" value function, $V_{\rm MC}$, built using Monte Carlo simulation [1] on a representative set of discrete states.

In the LSTD-RBF, it is necessary to select the parameters of the Gaussian kernels (centres and widths). In the general case, when an RBF network is used to approximate a function, the centres and widths can be selected according to the distribution of the data in the input space [6]. However, in LSTD-RBF the input data is unknown in advance. The solution commonly adopted is to distribute the centres uniformly along all the input space. To this end, the kmeans clustering algorithm is employed using as input an equidistant grid of points over the state space. Regarding Gaussian widths, a common approach is to fix the same widths for all Gaussian kernels using some heuristic. For example, in [6], the widths are fixed as $\sigma = d_{\text{max}}/\sqrt{2M}$, where M is the number of centroids and d_{\max} is the maximum distance between any pair of them. Another possible heuristic to find the function widths is given in [9]; it consists in, after selecting the centres, finding the maximum distance between centres and set σ equal to half of this distance, or one-third in a more conservative case. In our experiments, RBFs with these three values of widths, denoted by σ_{Hav} , σ_{Alp1} and σ_{Alp2} respectively, were tested. In addition, the half and twice of these three heuristics were also tested, i.e., a total of 9 values of σ .

Fig. 2 shows the MAE for both methods versus the number of features. For the sake of simplicity, Fig. 2 only shows the MAE error of the three best values of σ for LSTD-RBF. Experiments with LSTD-ELM were repeated 20 times, and the presented results show the worst case. For LSTD-RBF, it can be observed that using the same number of features MAE generally increases with the dimensionality of the MPD, whereas for LSTD-ELM it remains almost constant. Thus, despite the fact that in both cases MAE tends to approximately the same value, the number of features required by LSTD-ELM to achieve good performance is notably lower.

5 Conclusions

This paper has presented an LSTD algorithm that uses SLFNs trained with ELM to approximate the value function. Typically, LSTD has been combined with local function approximators, whose main drawback is poor scalability when

ESANN 2013 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 24-26 April 2013, i6doc.com publ., ISBN 978-2-87419-081-0. Available from http://www.i6doc.com/en/livre/?GCOI=28001100131010.



Fig. 2: MAE versus number of features. For d = 1, it has been used $n_{epi} = 2000$ episodes, MAE was computed in $n_{\text{MAE}} = 30$ equidistant discrete points, V_{MC} was computed using $n_{\text{MC}} = 4.8 \cdot 10^5$ episodes, and step = 0.033. Similarly, for d = 2: $n_{epi} = 3000$, $n_{\text{MAE}} = 64$, $n_{\text{MC}} = 1.024 \cdot 10^6$ and step = 0.125. For d = 3: $n_{epi} = 4000$, $n_{\text{MAE}} = 343$, $n_{\text{MC}} = 5.488 \cdot 10^6$ and step = 0.143. And for d = 4: $n_{epi} = 5000$, $n_{\text{MAE}} = 1296$, $n_{\text{MC}} = 2.0736 \cdot 10^7$ and step = 0.167.

the number of dimension grows. In contrast, the proposed method uses a global approximator that can deal with high dimensional problems. The performance of the proposed algorithm has been compared with a classical LSTD based on RBF in four MDPs, whose dimensionality varies from 1 to 4. The obtained results have shown that the proposed approach can be scaled to high dimensional problems better than LSTD-RBF. Future research includes studying its performance in more complex problems and extending the approach to a control RL algorithm.

References

- Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. The MIT Press, March 1998.
- [2] Martin L. Puterman. Markov Decision Processes. Wiley-Interscience, March 2005.
- [3] Steven J. Bradtke and Andrew G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1-3):33–57, 1996.
- [4] Justin A. Boyan. Technical update: Least-squares temporal difference learning. Machine Learning, 49(2-3):233-246, November 2002.
- [5] Michail G Lagoudakis, Ronald Parr, and L. Bartlett. Least-squares policy iteration. Journal of Machine Learning Research, 4:2003, 2003.
- [6] Simon O. Haykin. Neural Networks and Learning Machines. Prentice Hall, 2008.
- [7] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1-3):489–501, December 2006.
- [8] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1 edition, May 1996.
- [9] Ethem Alpaydin. Introduction to Machine Learning. The MIT Press, October 2009.