A Least-Square approach to Reinforcement Learning

Mathieu Tanneau
mathieu.tanneau@polymtl.ca

Abstract

Several algorithms used in Reinforcement Learning, such as temporal difference (TD) methods, aim at approximately solving a Bellman equation. The convergence of such methods relies on iteratively computing a fixed point, which may not always hold. here, we take a different approach, writing the solution of the Bellman equation as the solution of a convex, unconstrained program. We implement optimization techniques which display superlinear convergence for that problem. We then propose improvements over existing gradient-based TD methods.

1 Introduction

In all that follows, we assume the reader to have some familiarities with concepts and methods from the field of Reinforcement Learning (RL), including Dynamic Programming, Temporal Difference methods, function approximation, and related algorithms. We refer to [2] for the corresponding material.

As stated in [2], Markov Decision Processes (MDPs) are the theoretical foundation for Reinforcement Learning (RL) methods.

In the tabular case, given a policy π, the associated value function v_π is the unique solution of the system of linear equations, derived from the Bellman equation:

$$(I - \gamma P_\pi) v_\pi = r_\pi$$

When the state space is too large, one may use some function approximator. In the present work, we focus on linear function approximation. One can show (see [2]) that the optimal parameter vector is the unique solution to the following system of linear equations:

$$A\theta = b \tag{1}$$

with A, b defined below. We denote d the stationnary distribution over states, D the corresponding diagonal matrix, and Φ the feature matrix.

$$A = \Phi^T D (I - \gamma P_\pi) \Phi$$
$$b = \Phi^T D r_\pi$$

Therefore, many RL techniques aim at computing the solution to 1, either explicitly (by Dynamic Programming) or approximately (eg by Temporal Difference). Most of the time, they do so by iteratively computing the fixed point of a Bellman operator.

In the present work, we explore a different approach: instead of viewing the solution of 1 as a fixed point, we write it as the solution of a convex, unconstrained minimization problem, refered to as *Least Square approach*.

First, we derive new methods to compute the solution of the Bellman equation, and compare their performance to that of Dynamic Programming. Then, we try to improve existing gradient-based TD methods, by implementing momentum and adaptive learning rate techniques. Finally, we derive a limited-memory version of the LSTD algorithm.

2 Least Square approach

In this section, we consider the tabular case. We want to solve the following system:

$$Av = b \tag{2}$$

with $A = I - \gamma P_\pi$ and $b = r_\pi$. As mentionned earlier, this system can be solved using DP, and convergence is linear at rate γ.

Another way to solve 2 is by Least-Square approach. Indeed, multiplying 2 by A^T yields the following *normal equations*:

$$A^T Av = A^T b \tag{3}$$

which correspond to the first order condition of a *Least-Square* minimization problem:

$$\min_v \frac{1}{2} v^T A^T Av - (A^T b)^T v - \frac{1}{2} b^T b \tag{4}$$

This approach presents two main advantages compared to DP:

- Provided that A is non-singular, $A^T A$ is definite positive, regardless of the eigenvalues of A. Therefore, 4 is convex, so any v that satisfies the normal equations 3 is a global minizer for 4, and the reciprocal is also true.
- 4 can be solved using second order-based methods such as approximate Newton, and conjugate gradient. Contrarily to DP, these methods enjoy superlinear convergence.

The purpose of this section is to study efficient techniques to solve 4, and compare their performances to that of DP. We focus our analysis on *Conjugate Gradient*, which is the method of choice for solving unconstrained convex quadratic programs.

2.1 Conjugate gradient for Least Square

In this section, we now focus on convergence properties and numerical performance of Conjugate Gradient. We refer to [1] for the algorithm and its implementation, which we will not detail here. We shall only point out that the matrix product $A^T A$ need not be computed explicitly, so that each iteration can requires only two matrix-vector products.

2.1.1 Convergence rate

We begin by stating some convergence results regaring CG. The detailed theorems and proof are found in [1].

- First, in exact arithmetic (ie, if no rounding ever occured in the computations), CG terminates after exactly n iterations. This ensures that CG will converge in finite time, although n iterations mean an overall cost of $O(n^3)$.
- Second, if the eigenvalues of $A^T A$ occur in r distinct clusters, then CG *approximately* solves the problem after r steps. Therefore, the more clustered the eigenvalues are, the faster CG is.
- Finally, let $\kappa(M)$ denote the condition number of M. The sequence of CG iterates converges *at least* linearly with rate $\frac{\kappa(A)-1}{\kappa(A)+1}$:

$$\|v_k - v^*\|^2_{A^T A} \leq \left(\frac{\sqrt{\kappa(A^T A)} - 1}{\sqrt{\kappa(A^T A)} + 1} \right)^{2k} \|v_0 - v^*\|^2_{A^T A}$$

where, for a symetric positive definite matrix M, $\|\cdot\|_M$ is the euclidean norm induced by M.

In our case, recall that $A = I - \gamma P_\pi$, and assume all eigenvalues of P_π are real.
Since $Sp(P_\pi) \subset [-1, 1]$, then $Sp(A) \subset [1-\gamma, 1+\gamma]$. Therefore:

$$\kappa(A) \leq \frac{1+\gamma}{1-\gamma}$$

and recall $\kappa(A^T A) = \kappa(A)^2$, which yields:

$$\frac{\kappa(A)-1}{\kappa(A)+1} \leq \frac{\frac{1+\gamma}{1-\gamma}-1}{\frac{1+\gamma}{1-\gamma}+1} = \gamma$$

As a consequence, factoring norm equivalence and cost per iteration, the behaviour of CG is *at worst* similar to that of DP.

2.1.2 Preconditionning

The performance of CG can be further improved by preconditionning. When using a preconditionner, the linear system 3 becomes:

$$C^{-T} A^T A C^{-1} \hat{v} = C^{-T} A^T b \tag{5}$$

with $\hat{v} = Cv$.

One may choose C so that the condition number of $C^{-T} A^T A C^{-1}$ is smaller than that of $A^T A$, or to cluster the eigenvalues of $C^{-T} A^T A C^{-1}$, thus improving the behaviour of CG. Various preconditionners are discussed in [1].

By definition, $A = I - \gamma P_\pi$, so a natural preconditionner is to set $C^{-1} = I + \gamma P_\pi$. Then,

$$AC^{-1} = (I - \gamma P_\pi)(I + \gamma P_\pi) = I - (\gamma P_\pi)^2$$

In our experiments, this preconditionner reduces the number of iterations by a factor of 2. However, each iteration now costs twice as more, so there is no overall benefit in computing time.

Sparse preconditionners, may offer a way out of this extra computationnal cost. For diagonal preconditionners, the product $C^{-1}v$ only requires n operations, and this additional computional cost is not significant. One possibility is to extract the diagonal of $I + \gamma P_\pi$, for which we observed erratic improvements in performance. Finding good preconditionners is extremely problem-specific, and generally remains an art.

2.2 Numerical results

We now compare the numerical performance of various Least-Square techniques and Dynamic Programming. We ran our experiments on the random-walk problem, and implemented the following algorithms:

- Dynamic Programming
- First-order LS methods: gradient descent, gradient descent with momentum, gradient descent with adaptive learning rates (AdaGrad, RMSProp and ADAM)
- Second-order LS methods: conjugate gradient and Newton method

For ease of reading, we only display results for DP, CG, ADAM, and gradient descent with momentum. The other methods performed poorly.

2.2.1 Learning curves

The learning curves displayed in figure 1 were obtained on a $1000-$state random walk, with a discount factor $\gamma = 0.9$. The policy was to go left or right with 0.5 probability, and take 1 to 100 steps in that direction.

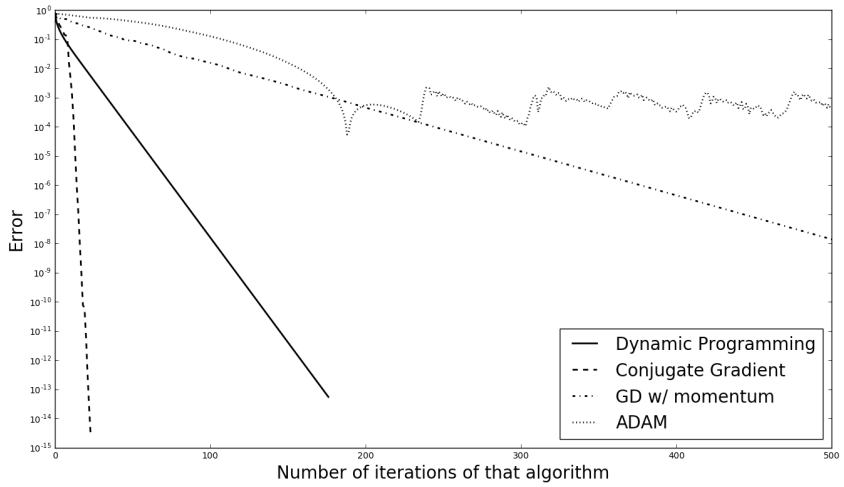

Figure 1: Learning curves

In this experiment, Conjugate Gradient significantly outperforms other methods, including Dynamic Programming. For the first few iterations, CG and DP exhibit similar performance, which is in line with the error bound for CG discussed in 2.1.1. After a few iterations, however, the CG error drops sharply, quicly reaching the stopping criterion. Gradient descent eventually reaches termination (in about 1000 iterations, not displayed), while ADAM seems trapped in some sort of cycling, after only a couple hundred iterations.

2.2.2 Influence of discount factor

We now study the inlfuence of the discount factor on numerical performance of DP and CG. We already know that the convergence rate of DP is γ, thus larger values of γ lead to poorer performance for DP. We also mentionned that the convergence of CG is at worst linear, with convergence rate γ. Therefore, one could expect CG to also display poorer performance when γ increases.

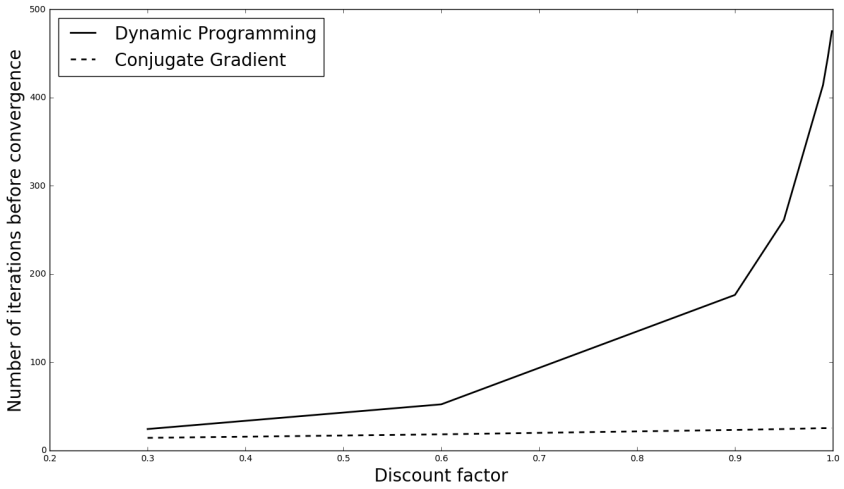

Figure 2: Influence of discount factor on numerical performance

However, this is not the case, as displayed by figure 2: the performance of CG is unaffected by the discount factor. Indeed, the behaviour of CG depends on how the eigenvalues of A^TA are distributed: if there are r clusters of eigenvalues, then CG terminates in about r iterations. Since $A = I - \gamma P_\pi$, changing γ does not affect the clustering of the eigenvalues of A. The same may be happening for the eigenvalues of A^TA.

3 Improving Gradient-based TD Learning

3.1 Gradient-Based TD

In order to overcome convergence issues in the case of off-policy learning with function approximation, gradient-based TD methods were introduced in [3] and [4]. Gradient-based TD algorithms, such as GTD, GTD-2 and TDC seek to minimize an error function, by approximating a gradient descent.

The error function that GTD-2 and TDC seek ot minimize is the *Mean Squarred Projected Bellman Error* (MSPBE), which is defined as :

$$MSPBE(\theta) = E[\delta\phi]^T E[\phi\phi^T]^{-1} E[\delta\phi]$$

Another set of paramters, w, is introduced to estimate the gradient of the error function. In what follows, we focus on TDC, which updates are given by:

$$\theta \leftarrow \theta + \alpha\delta\phi - \alpha\gamma\phi'\phi^T w$$
$$w \leftarrow w + \beta(\delta - \phi^T w)$$

and the gradient estimate is given by:

$$-\frac{1}{2}\nabla_\theta MSPBE(\theta) \simeq \delta\phi - \gamma\phi'\phi^T w$$

Convergence conditions on α and β are found in [4].

3.2 Numerical results

Since TDC provides an estimate of the gradient, we may use that estimate to implement more efficient techniques than gradient descent. Therefore, we implemented TDC with momentum and adaptive learning rate. We compared the performance of these methods to TD(0) and LSTD algorithms. Numerical results are displayed in figure 3

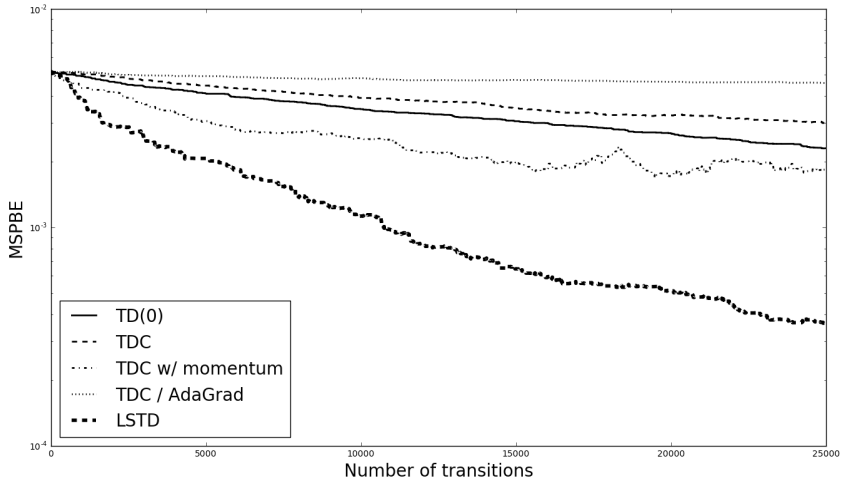

Figure 3: Learning curves

Overall, as illustrated in figure 3, TDC with adaptive learning rate (here, AdaGrad) performs quite poorly. This might be caused by poor estimates of the gradient, or by the fact that α is automatically scaled while β is not. On the other hand, adding momentum seems to improve performance, especially for early learning.

Obviously, different behaviours may be observe by carefully tuning the hyperparameters. However, hyperparameter tuning is problem-specific. Our goal is rather to exhibit the potential of improvement over gradient-based TD.

4 Limited-memory LSTD

4.1 LSTD

The idea behing the LSTD algorithm is to construct an estimators A^{-1} and b, and compute θ directly from the product $A^{-1}b$. This approach is more data-efficient than TD methods, yet its computational cost grows as $O(n^2)$, where n is the number of features. Therefore, LSTD is often impractical when dealing with a large number of features.

The estimator of A^{-1} is updated at each iteration, using the Sherman-Morrison formula :

$$\hat{A}^{-1} \leftarrow \hat{A}^{-1} - \frac{\hat{A}^{-1}\phi(\phi - \gamma\phi')^T\hat{A}^{-1}}{1 + (\phi - \gamma\phi')^T\hat{A}^{-1}\phi}$$

4.2 Limited-memory LSTD

Let us define $\hat{B}_t = \hat{A}_t^{-1}$. Given a vector v, the product $\hat{B}_t v$ can be written as:

$$\hat{B}_t \cdot v = \hat{B}_{t-1}v - \frac{\hat{B}_{t-1}\phi_t(\phi_t - \gamma\phi_{t+1})^T\hat{B}_{t-1}v}{1 + (\phi_t - \gamma\phi_{t+1})^T\hat{B}_{t-1}\phi_t}$$

This matrix-vector product can be computed in time $O(n)$, assuming $\hat{B}_{t-1}v$ and $\hat{B}_{t-1}\phi_t$ are known. From this remark, we can derive a limited-memory version of LSTD, which we denote L-LSTD.

Instead of keeping track of all past data, L-LSTD only performs an update based on the last m observations. Such an update can then be computed in time $O(m^2n)$, which is faster than $O(n^2)$ when $m \ll n$.

4.2.1 Numerical results

L-LSTD appears to perform very badly in practice. Learning either stops very quickly, or explodes. One explanation could be that consecutive observations are highly correlated. Therefore, we added a larger batch memory, from which minibatches of size m were sampled at each iteration. This did not improve performance.

References

[1] Nocedal Jorge and J Wright Stephen. *Numerical optimization*. Springer, 1999.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 2017. Second Edition - Draft.

[3] Richard S Sutton, Hamid R Maei, and Csaba Szepesvári. A convergent $o(n)$ temporal-difference algorithm for off-policy learning with linear function approximation. In *Advances in neural information processing systems*, pages 1609–1616, 2009.

[4] Richard S Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 993–1000. ACM, 2009.