

# Learning for stochastic dynamic programming

Sylvain Gelly and Jérémie Mary and Olivier Teytaud

\*

IA-TAO, Lri, Bât. 490,  
Université Paris-Sud, 91405 Orsay Cedex, France

**Abstract.** We present experimental results about learning function values (i.e. Bellman values) in stochastic dynamic programming (SDP). All results come from openDP ([opendp.sourceforge.net](http://opendp.sourceforge.net)), a freely available source code, and therefore can be reproduced. The goal is an independent comparison of learning methods in the framework of SDP.

## 1 What is stochastic dynamic programming (SDP) ?

We here very roughly introduce stochastic dynamic programming. The interested reader is referred to [1] for more details. Consider a dynamical system that stochastically evolves in time depending upon your decisions. Assume that time is discrete and has finitely many time steps. Assume that the total cost of your decisions is the sum of instantaneous costs. Precisely:  $cost = c_1 + c_2 + \dots + c_T$ ,  $c_i = c(i, x_i, d_i)$ ,  $x_i = f(x_{i-1}, d_{i-1}, \omega_i)$ ,  $d_{i-1} = strategy(x_{i-1}, \omega_i)$  where  $x_i$  is the state at time step  $i$ , the  $\omega_i$  are a random process,  $cost$  is to be minimized, and  $strategy$  is the decision function that has to be optimized. Stochastic dynamic programming is a control problem : the element to be optimized is a function. Stochastic dynamic programming is based on the following principle : Take the decision at time step  $t$  such that the sum "cost at time step  $t$  due to your decision" plus "expected cost from time steps  $t + 1$  to  $T$  from the state resulting from your decision" is minimal. Bellman's optimality principle states that this strategy is optimal. Unfortunately, it can only be applied if the expected cost from time steps  $t + 1$  to  $T$  can be guessed, depending on the current state of the system and the decision. Bellman's optimality principle reduces the control problem to the computation of this function. If  $x_t$  can be computed from  $x_{t-1}$  and  $d_{t-1}$  (i.e., if  $f$  is known) then this is reduced to the computation of

$$V(t, x_t) = E c(t, x_t, d_t) + c(t + 1, x_{t+1}, d_{t+1}) + \dots + c(T, x_T, d_T)$$

Note that this function depends on the strategy. We consider this expectation for any optimal strategy (even if many strategies are optimal,  $V$  is uniquely determined). Stochastic dynamic programming is the computation of  $V$  backwards in time, thanks to the following equation :  $V(t, x_t) = \inf_{d_t} c(t, x_t, d_t) + V(t + 1, x_{t+1})$ , or equivalently

$$V(t, x_t) = \inf_{d_t} [c(t, x_t, d_t) + E_{\omega_{t+1}} V(t + 1, f(x_t, d_t, \omega_{t+1}))] \quad (\text{main equation})$$

Thanks to Bellman's optimality principle, the computation of  $V$  is sufficient to define an optimal strategy by  $d_{i-1} = \arg \min c_{i-1} + V(i, x_i) =$

---

\*This work was supported in part by the Pascal Network of Excellence.

$\arg \min [c(i-1, x_{i-1}, d_{i-1}) + E_{\omega_i} V(i, f(x_{i-1}, d_{i-1}, \omega_i))]$ . This is a well known, robust solution, applied in many areas including the famous problem of power supply management, which is a stock management problem. A general introduction, including learning, is [1]. Combined with learning, SDP can lead to positive results in spite of large dimensions. It can also be combined with various techniques for restrictions the domain ([2],[3]). Indeed, this equation can not be applied to all possible  $x_t$  (the domain is usually infinite). The usual solution is based on learning :  $V(t, x)$  is evaluated for a finite number of  $x$ , and a full function  $V(t, \cdot)$  is obtained by learning. The sampling method refers to the way of choosing the  $x$ 's for a learning. The regression refers to the way  $V(t, \cdot)$  is built from examples. The optimizer refers to the method used for solving the main equation above in  $d_t$  for one particular value of  $t, x_t$ ,  $V(t+1, \cdot)$  being already approximated. Using regression (instead of interpolation or discretization) is in particular usual in the area of reinforcement learning, which is a generalization of dynamic programming ([2, 3]). We here consider an approach based on backwards (in time) induction of the whole function value. The pros of this method are (i) robustness wrt the initial state (ii) stable convergence by decomposition among time steps (no fixed point iterations) (iii) learning is performed on noise-free points, within the precision of the optimization and of the previously computed  $V(t, \cdot)$ .

## 2 Why this article provides interesting results

Many results have been reported in the literature, in particular in cases in which dynamic optimization without learning would be untractable. Learning was sometimes based on neural networks ([4, 5]), adaptive discretization ([6]), CMAC ([7],[8]), EM-learning of a sum of gaussians ([9]), or various forms of local units ([10, 11, 12]). But as pointed out in [13], learning-based dynamic optimization has not reached the industry ; this is at least partially due to the lack of clear comparison between the different learning-techniques in this framework, each author implementing one algorithm and comparing it to existing results on one problem, sometimes with different values of parameters, different loss functions, different time steps, different action spaces (many algorithms being in particular unable to deal with continuous and/or multi-dimensional action spaces). In particular, the lack of standard format for describing problems in continuous state spaces with no prior knowledge on the transition function makes comparisons difficult ; also, such a format is very difficult to define, as Turing-computability is nearly the only common basis for various problems (and some authors even only work on logs of simulations). We therefore implemented a full learning-base dynamic programming tool, freely available, in which any author can add an optimization tool, a sampling method or a learning method. For this first work based on this tool, we mainly integrated the full Weka suite, which allows the comparison of many learning methods, and some www-available learning tools. We compare in the same framework (same problems, same optimizers, same sampling methods, same fitnesses), many classical learning tools. We believe

that the comparisons are very different from standard learning benchmarks, as in the SDP case (i) robustness is much more important, (ii) the existence of false local minima in the approximation of the function value might be misleading for optimizers, and (iii) the decay of contrasts through time is an important trouble (iv) the exact loss function ( $L^2$  norm,  $L^p$  norm, something else ?) that should be used during learning is not clearly known even in theory (see [14] for some elements).

### 3 Methods

We compare the following learning methods. A **linear regression on the kernel (LRK) matrix**  $K$ , where  $K(i, j) = (1 + xy)^d$  (polynomial case) or  $K(i, j) = \exp(-\|x - y\|^2/\sigma^2)$  (gaussian case). I.e., with  $n$  examples  $(x_i, y_i)_{i \in [1, n]}$ , this method builds  $K$  and performs a linear regression (regularized by the weights) on examples  $(K(i, 1), K(i, 2), \dots, K(i, n)), y_i$ . Using Weka [15] (the descriptions below are slightly adapted from the Weka www-site), we also use : **meta.AdditiveRegression** (AR): Meta classifier that enhances the performance of a regression base classifier. Each iteration fits a model to the residuals left by the classifier on the previous iteration. Prediction is accomplished by adding the predictions of each classifier. Smoothing is accomplished through varying the shrinkage (learning rate) parameter. The base classifier is here decision stump (see below). **rules.ConjunctiveRule** : this class implements a single conjunctive rule learner. A rule consists of a conjunction of antecedents and the consequent (class value) for the regression. In this case, the consequent is the distribution of the available classes (or numeric value) in the dataset. If the test instance is not covered by this rule, then it's predicted using the default class distributions/value of the data not covered by the rule in the training data. This learner selects an antecedent by computing the Information Gain of each antecedent and prunes the generated rule using Reduced Error Pruning (REP). For regression, the Information is the weighted average of the mean-squared errors of both the data covered and not covered by the rule. In pruning, the weighted average of the mean-squared errors of the pruning data is used for regression. **trees.DecisionStump** : decision stump algorithm, i.e. one-node decision trees (with one-variable-comparison-to-real in the node). **rules.DecisionTable** : building and using a simple decision table majority classifier. For more information see [16]. **lazy.IBk** :  $k$ -nearest neighbours algorithm. **lazy.KStar** :  $K^*$  is an instance-based classifier, that is the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. The underlying assumption of instance-based classifiers such as  $K^*$ , IB1, PEBLS, etc, is that similar instances will have similar classes. For more information on  $K^*$ , see [17]. **functions.LeastMedSq** : implements a least median squared linear regression using the robust regression and outlier regression ([18]). **functions.LinearRegression** : linear regression based on the Akaike criterion for model selection. **trees.lmt.LogisticBase** : LogitBoost algorithm (see [19]). **functions.MultilayerPerceptron** (MLP)

: the multilayer perceptron of weka. **functions.RBFNetwork** : implements a normalized Gaussian radial basis function network. It uses the k-means clustering algorithm to provide the basis functions and learns a linear regression (numeric class problems) on top of that. Symmetric multivariate Gaussians are fit to the data from each cluster. It standardizes all numeric attributes to zero mean and unit variance. **trees.REPTree** : fast decision tree learner. Builds a regression tree using variance reduction and prunes it using reduced-error pruning (with backfitting). Only sorts values for numeric attributes once. **functions.SimpleLinearRegression** (SLR) : a simple linear regression model. **functions.SMOreg** : implements A.J.Smola and B. Scholkopf sequential minimal optimization algorithm for training a support vector regression using polynomial or RBF kernels ([20, 21]). This implementation normalizes all attributes. **rules.ZeroR** : simply predicts the mean ; only for comparison.

## 4 Results

All our benchmarks have been designed in a manner that make greedy optimization unefficient ; for most problems, the rewards are provided only at the time at which the goal is reached (for robotic problems), or problems are designed in such a way that greedy algorithms are known inefficient (stock management ; here, as in many interesting cases of stock management, the greedy algorithm is worst than the algorithm that randomly chooses a decision). We however always compare our results to the greedy one, that decides only by instantaneous minimization of the loss, and to the so-called random one (that randomly chooses one decision consistent with constraints). The problems are presented below :

Problem	Stock manag.	Simp. bot	Bot	Many bots	Arm	Away
State space	dim 4	2	2	8	2	2
Random process	dim 1	0	0	0	2	2
Action space	dim 4	1	1	4	2	2
Nb timesteps	30	20	20	20	30	40
Nb scenarii	9	0	0	0	3	2

Rough description : stock management: use your stocks optimally to satisfy a demand ; simple bot : avoid an obstacle and reach a target ; bot : avoid many obstacles and reach a target ; many bots : get together and reach a target ; arm : follow a moving target with the hand ; away : your hand must avoid a bug. In problems of bots, the only choice is the angle of the direction (the speed is constant). All action spaces are continuous. All state variables are continuous. In some cases, the random process is not exactly markovian (but almost). Learning methods were compared on various (very different) problems and can be reproduced by the (linux-equipped) reader thanks to the freely available source-code. Hyper-parameters where the default ones in the source codes that have been used, except for (i) the  $\Gamma$  parameter of SMOReg, the inverse of the variance, is set to 10, (ii) there are 15 neurons in the hidden layer of the MLP, (iii) in *IBk*, the are  $k = 5$  neighbours and the weighting scheme is the inverse-distance. The architecture of the code is designed for an easy introduction of new software and the authors are available for helping people interested in introducing/comparing new methods. We only report the results for the seven best methods and for the

best among "greedy" and "stupid". All local optimizations have been performed by the naive genetic algorithm provided in the source code, allowed to use 70 function evaluations. 300 points were sampling in a quasi-random manner (the Niederreiter sequence ; see e.g. [22]).

Away		Arm	
SMOreg	2.6 ± 0.2	SMOreg	278 ± 45.3431
IBk	2.7 ± 0.331662	IBk	357 ± 44.2832
KStar	2.7 ± 0.244949	LRK(poly)	473 ± 90.89
LRK(Gaussian)	3.05 ± 0.15	LeastMedSq	503 ± 77.9808
SLR	3.1 ± 0.2	MLP	522 ± 61.2862
AR	3.3 ± 0.458258	LRK(gaus)	530 ± 73.6206
REPTree	3.5 ± 0.316228	LRK(linear)	567 ± 59.8415
Stupid	6 ± 1.41421	Greedy	600 ± 0.0

  

Arm		Stock Management	
SMOreg	6.90001 ± 0.111998	MLP	1863.63 ± 7.06183
MLP	8.11303 ± 0.567178	SMOreg	2196.03 ± 11.9168
LeastMedSq	8.52682 ± 0.723674	LinearRegression	2333.77 ± 1.39077
KStar	8.97947 ± 0.0615761	IBk	2377 ± 15.5075
IBk	8.98368 ± 0.0387504	Stupid	2562.94 ± 2514.3
AR	8.99936 ± 0.00191481	KStar	2589.71 ± 25.9275
LRK(gaus)	9 ± 3.16228e-16	LRK(gaus)	2719.04 ± 165.774
Greedy	9 ± 0.0	LRK(linear)	2747.29 ± 23.5923

  

Simple Bot		Bot	
SMOreg	550 ± 3.16228e-16	LRK(poly)	60 ± 3.16228e-16
IBk	550 ± 3.16228e-16	SMOreg	60.5 ± 1.5
KStar	650 ± 3.16228e-16	MLP	61 ± 2
DecisionTable	700 ± 74.162	IBk	65 ± 3.16228e-16
REPTree	875 ± 130.863	REPTree	91 ± 8.88819
LRK(gauss)	1000 ± 3.16228e-16	AR	92.5 ± 6.80074
LRK(linear)	1000 ± 3.16228e-16	LRK(gaus)	100 ± 3.16228e-16
Greedy/Stupid	1000 ± 0.0	Greedy	100 ± 0.0

## 5 Conclusion

We compared various learning algorithms in the case of dynamic programming. The focus was on a limited number of sample points per time step (300). SVM with the SMO algorithm and gaussian kernel was the best algorithm in 4 cases, and the second best in the two remaining cases. One can note that sigmoids look very natural for function values in stock management (classical curves, when they follow the law of increasing marginal cost, are strictly convex and "look like" sigmoidal functions). If one is interested in symbolic controllers, one can note the good overall performance of REPTree, that was always better than the greedy/stupid method, whereas many learners were not. Note that our results might strongly rely on the fact that we restrict our attention to a limited number of sample points per time step. All the experiments can be performed with <http://opendp.sourceforge.net>.

## References

- [1] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-dynamic programming*, athena scientific. 1996.
- [2] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. Technical Report UM-CS-1993-002, , 1993.
- [3] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*. MIT Press., Cambridge, MA, 1998.
- [4] R. Coulom. High-accuracy value-function approximation with neural networks. In *Esann 2004*.
- [5] Rémi Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.
- [6] R. Munos and A. Moore. Variable resolution discretization in optimal control. Technical report, 1999.
- [7] R.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.
- [8] D.Precup and R.S. Sutton. Exponentiated gradient methods for reinforcement learning. In *Proc. 14th International Conference on Machine Learning*, pages 272–277. Morgan Kaufmann, 1997.
- [9] J. Yoshimoto, S. Ishii, and M. Sato. Application of reinforcement learning to balancing of acrobot, 1999.
- [10] C.W. Anderson. Q-learning with hidden-unit restarting. In *Advances in Neural Information Processing Systems 5*, pages 81–88, 1993.
- [11] R. Kretchmar and C. Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning, 1997.
- [12] B. Ratitch and D. Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *ECML 2004: 347-358*, 2004.
- [13] S. Keerthi and B. Ravindran. A tutorial survey of reinforcement learning, 1995.
- [14] R. Munos. Error bounds for approximate value iteration. In *Proceedings of AAAI 2005*, 2005.
- [15] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2005.
- [16] R. Kohavi. The power of decision tables. In Nada Lavrac and Stefan Wrobel, editors, *Proceedings of the European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 914, pages 174–189, Berlin, Heidelberg, New York, 1995. Springer Verlag.
- [17] John G. Cleary and Leonard E. Trigg. K\*: an instance-based learner using an entropic distance measure. In *Proc. 12th International Conference on Machine Learning*, pages 108–114. Morgan Kaufmann, 1995.
- [18] P. Rousseeuw and A. Leroy. *Robust Regression and Outlier Detection*. John Wiley and Sons, 1987.
- [19] J. Otero and L. Sanchez. Induction of descriptive fuzzy classifiers with the logitboost algorithm. *soft computing* 2005. 2005.
- [20] A.J. Smola and B. Scholkopf. A tutorial on support vector regression. 1998.
- [21] S.K. Shevade, S.S. Keerthi, C. Bhattacharyya, and K.R.K. Murthy. Improvements to smo algorithm for svm regression. 1999.
- [22] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, 1992.