Feature Discovery in Approximate Dynamic Programming

Philippe Preux and Sertan Girgin and Manuel Loth

Abstract— **Feature discovery aims at finding the best representation of data. This is a very important topic in machine learning, and in reinforcement learning in particular. Based on our recent work on feature discovery in the context of reinforcement learning to discover a good, if not the best, representation of states, we report here on the use of the same kind of approach in the context of approximate dynamic programming. The striking difference with the usual approach is that we use a non parametric function approximator to represent the value function, instead of a parametric one. We also argue that the problem of discovering the best state representation and the problem of the value function approximation are just the two faces of the same coin, and that using a non parametric approach provides an elegant solution to both problems at once.**

I. Introduction

In computer science, *feature discovery*, also known as feature construction, stands as a topic of outstanding importance, in particular in the fields of artificial intelligence, optimization, and machine learning; it considers the problem of finding the best, or at least, a good, or not too bad, representation of data [1]. Two reasons can be stated for employing feature discovery. First, in some cases, we have to represent some objects, and we have to represent them by a set of features that hopefully describe the object well enough so that the algorithm handling it will also perform well: this is the case where we do not know exactly how to represent the object so as to perform the task, within a reasonable amount of CPU-time; this is the case in many games for instance, such as Go, or card games, or even video games. Second, in other cases, we want to solve a problem for which we have formal properties that guarantee that we have enough information to effectively solve the problem; but, we also know that adding more information will make the actual resolution faster; this is the case when solving Markov decision problems in which we know for sure that the state is fully described, and that an optimal policy may be obtained; however, experimental evidences show us that better policies may be obtained faster if the state is enriched with extra information. For example, consider the problems that deal with the control of a dynamical system following Newton's law: we know that its state is given by its current position and velocity, and that an action provided as an acceleration, perfectly defines the next state; this information is enough to learn an optimal policy. For instance, let us consider the cart-pole problem in which a pole is fixed to a moving cart and it can revolve around this fixed point. A horizontal force may be applied to the cart in either direction; starting from a given position (x, θ) and velocity $(\dot{x}, \dot{\theta})$, the goal is to get the pole in upright position, in an equilibrium state (see Fig. 1); though the state $(x, \dot{x}, \theta, \dot{\theta})$ is enough to compute, or learn, an optimal policy, it is well-known that enriching the state with $\cos(\theta)$ and $\sin(\theta)$ greatly accelerates learning a good policy (see [2]).

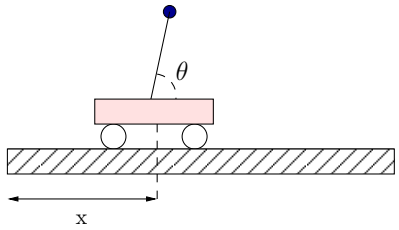

Fig. 1. The cart-pole problem.

So, we are left with finding this useful information to enrich the state representation. Before going further, we would like to emphasize that for us, what really matters here are "practical" reasons, at least in the case of MDPs for which we know we have adequate information in the state to learn, and represent an optimal policy. Indeed, for this kind of problems, we know for sure that we can compute an optimal policy, and we have different classes of algorithms to achieve this goal. Our point here is that we aim at obtaining the same kind of policy, that is, policies with identical quality, but using fewer computational resources, in particular CPU time. So, we will be dealing with optimizing the learning curve of an algorithm, not making possible the acquisition of an optimal solution. This aim would surely benefit from a relevant formalization of learning curves, but there is no such thing in the current state of the art; it would indeed be a great achievement to have one. In the meantime, less formal approaches are still worthy.

While finding the best representation of states can be done by a human expert, we think it worthwhile to study how this can be automated for various reasons: first, it is not easy, even for an expert, to find a good representation as the complexity of the problem increases, and an automatic tool may, to the least, help him, or her; second, pursuing the goal of having a software to learn optimal policies to be used by non experts, the ability to discover a good representation of states would definitely be an appreciated feature of this software.

Now, we would like to detail what we mean by this information we want to add to the state representation.

Philippe Preux and Manuel Loth are affiliated to the Université de Lille, the Laboratoire d'Informatique Fondamentale de Lille (Computer Science Lab., associated to the CNRS), and the INRIA; Sertan Girgin is on leave from INRIA (email: {philippe.preux,manuel.loth}@inria.fr, sertan@ceng.metu.edu.tr).

The authors acknowledge the support from INRIA during this research. Manuel Loth also gratefully acknowledges the support *Région Nord - Pas-de-Calais, in France*.

Assuming a Markov decision problem (MDP)–Newtonian systems are good examples of the kind of systems we wish to deal with here and, to make our presentation clearer, we will assume that kind of systems, though our work applies to other kinds of MDPs–, a state can be assumed to be a vector of P real components ($P = 4$ for the cart-pole problem mentioned above). Then, we want to add components to the state variables; obviously, assuming we have no other source of information on the state except for these P state variables, the information we can add may only be non linear mappings of these state variables[1]. Such a non linear mapping is called a *feature*. So, in some way, given a set of state variables, we can define all those features, and choose the useful ones. Facing such an infinite amount of potential features is obviously not possible, so we have to restrict our search to some classes of such non linear mappings.

To tackle the problem of learning a value function on a continuous domain, it is inevitable to use a function approximator. Many kinds of such function approximators have been proposed since the seminal works by mathematicians in the XIX$^{\text{th}}$ century. Today, in machine learning, neural networks, and kernel methods are very popular: provided an input (the representation of a state in this paper), the function approximator outputs a certain real value; the function approximator has to be trained, or fit, so that the output is the expected one, given an input, and that, for all possible inputs in a certain domain. We may categorize function approximators in various ways; for our purpose here, the relevant categorization is that of parametric *vs.* non parametric function approximators. Let us precise what we mean by these words, since these are used with various meanings. A *parametric* function approximation has a fixed architecture and the training phase aims at finding real weights; this fixed architecture does not depend on the data that are actually used during training, only the weights depend on them. A *non parametric* function approximator does not have a fixed architecture: training involves finding weights and the architecture itself: this architecture evolves according to the data that are actually observed. For instance, a radial basis function grid (RBF-grid), or a traditional multi-layer perceptron (MLP) is a parametric function approximation, whereas a cascade-correlation network (CCN) [3] is a non parametric one.

Strictly speaking, using a parametric function approximation makes sense only if we have a good idea of what the function to represent looks like, or more formally, if we have a good idea of its regularities, and singularities. Indeed, with a fixed architecture, the set of functions that can be well approximated up to a certain accuracy is restricted; while the set of all MLPs do have the universal approximation property, once we have chosen a single MLP to work with, it is not able to approximate with an arbitrary accuracy, any arbitrary function; the common practice is then to choose an MLP such that its architecture is complex enough to be able to represent the (expected, or assumed) complexity of the function to be learned, opening the way to over-fitting among other problems.

To the opposite, a non parametric function approximator is ideal when the function to be learned is not known beforehand. The learning algorithm is then capable to make the architecture more complex, as required during learning, to cope with the singularities of the function under learning. The idea is very simple: starting with the most simple architecture, when unable to fit the data up to the expected degree of accuracy, the algorithm makes the architecture a little more complex to cope with this lack of precision[2].

In mathematical terms, in the parametric approach, we search for the best value of (scalar) parameters of a given estimator function, whereas in a non parametric estimator, along with the same parameters optimization, we allow ourselves to add new terms to the function. While the mathematics of parametric function approximation is well developed, those of non parametric function approximations is far less developed; this is clearly a drawback of this approach, since we can not hope (yet) to take advantage of interesting formal properties.

Adding a new term to a non parametric function approximator turns out to add a non linear mapping of the input variables. So, that is exactly what we named a feature above, and so, we can consider a non parametric function approximator as a way to enrich the representation of data: when the approximation algorithm finds it suitable, *e.g.* when the precision is not as high as expected, it adds a new feature; thus, the features that have been added along a learning process may be considered as the useful information we wish to discover. So, we see that when we consider non parametric function approximation, the approximation of the learned function, and the acquisition of a good representation of the data are really the two faces of the same coin.

In the sequel, we will thus present this approach. Section II essentially introduces our notation, and a very synthetic background on approximate dynamic programming. Then, in Section III, the two non parametric function approximators we use in our work are presented. The "Equi-Correlated Network" (ECON) belongs to the family of kernel methods; the other one is the cascade-correlation networks. Then, Section IV describes our approach, aiming at embedding a non parametric function approximator into an adaptive dynamic programming algorithm. Section V presents some experimental results. Section VI concludes and discusses future work.

II. Background

In this section, we provide background material on Markov decision problems, and approximate dynamic programming. The section on MDPs serves only as a mean to define our notations. The section on approximate dynamic programming

[1]In most cases, linear mappings will be redundant due to the algorithms and/or function approximators being used.

[2]Note that, this process cannot repeat indefinitely as it will hinder generalization and lead to over-fitting.

is very synthetic. If necessary, the reader is kindly invited to check one of these following references [4], [5], [6].

A. Markov decision problems

We consider MDPs defined on a discrete or continuous state space, with a discrete action space, and in discrete time. We only consider MDPs in which the goal is to optimize the sum of discounted rewards.

- T denotes the set of instants, $t \in T$ its elements,
- $\mathcal{X}$ denotes the state space, $x \in \mathcal{X}$ its elements; in this paper, $\mathcal{X}$ is a subset of $\mathbb{R}^P$ for some P which we name the *dimension* of the state space,
- $\mathcal{A}$ denotes the action space, which is assumed to be discrete in this paper, $a \in \mathcal{A}$ its elements,
- $\mathcal{P}$ denotes the transition function: $\mathcal{X} \times \mathcal{A} \times \mathcal{X} \mapsto [0, 1]$, $\mathcal{P}(x, a, x') = Pr[x_{t+1} = x' | x_t = x, a_t = a]$,
- $\mathcal{R}$ denotes the reward, or cost, function: $\mathcal{X} \times \mathcal{A} \times \mathcal{X} \mapsto \mathbb{R}$, $\mathcal{R}(x, a, x') = \mathbb{E}[x_{t+1} = x' | x_t = x, a_t = a]$, where $\mathbb{E}$ denotes the expectation operator,
- r_t denotes the particular reward, or return, or consequence, or outcome, ... that has been received at instant t: r is a realization of a stochastic process which mean, for each (x, a, x'), is defined by $\mathcal{R}$,
- $\gamma \in [0, 1)$ is the discount factor,
- a policy is denoted by π, and we consider either deterministic policies $\pi(x)$, stochastic policies $\pi(x, a)$,
- the objective function to optimize is denoted by J: here, we deal only with $J(x) \equiv \sum_{t=0} \gamma^t r_t | x_0 = x$.

We can consider the problem with all these information available: this is known as a *planning* problem, and dynamic programming methods are the celebrated way to solve it. We can also consider settings in which the transition function, or the reward function, is unknown: this is known as the *learning* problem, and we have currently two main approaches, namely, temporal difference (TD) learning, and direct policy search (DPS). These two methods are basically actor-critic approaches, the focus being put on the critic in TD learning, whereas being put on the actor in the DPS. In these cases, since no model of the environment is available, learning takes place by means of interactions with the environment (or a simulator of it).

All these approaches either learn, or compute, a value function, or directly a policy. To face continuous, or large discrete, state spaces, these approaches rely on a function approximator to represent these functions. To meet with the introductory discussion of parametric and non parametric function approximators, most works to date have dealt with parametric function approximators, though interesting non parametric approaches have also been published. We have yet tackled the case of the learning problem–either value-based [7], or by direct policy search [8]–, so we focus on approximate dynamic programming approaches in this paper.

B. Approximate dynamic programming

Approximate dynamic programming (ADP) is a well-known dynamic programming approach to deal with large, either discrete, or continuous, state spaces. ADP basically comes in two flavors, either relying on the value iteration algorithm, or the policy iteration approach. In either case, the 'A' in ADP involves using a function approximator to represent the value function, and derive a policy from it.

1) Approximate Policy Iteration: (API) is the policy iteration algorithm in which a function approximator is used to represent the value function. So, instead of having an exact representation of the value function of the current policy, we only have an approximate representation of it. So, starting with an initial policy π_0, API alternates the two phases:

- Approximate policy evaluation: approximation of the value function of π_k, V^{π_k}, which yields $\hat{V^{\pi_k}}$
- Policy improvement: based on V^{π_k}, compute a greedy policy π_{k+1}.

The state space being assumed possibly infinite, the evaluation of V^{π_k} relies on a set of sampled states. Variants are possible, and we refer the reader to relevant references, such as [5], [6] for more details.

For the same reason, the policy improvement can not be performed explicitly, and thoroughly. Instead of that, this step may be made implicit by computing the optimal action in states for which it is required, yielding some implicit π_{k+1}. This optimal action is the greedy action, based on V^{π_k}.

2) Approximate Value Iteration: (AVI) is the value iteration algorithm in which the value functions are represented using a function approximator. So, given a current estimation of the value function $\hat{V_k}$, we estimate the result of applying the Bellman operator on this function on a set of samples; from these estimations, we perform a regression to obtain a new estimate $\hat{V_{k+1}}$.

3) Feature discovery in ADP: The use of features instead of raw variables have been advocated for very long. The use of a compact representation of states by way of a set of features has been studied from a theoretical point of view by [9]. However, this work does not deal with the problem of feature discovery itself.

As us, some authors have proposed that feature discovery and function approximation are very closely related. [10] deals with two problems defined on a discrete state space (tic-tac-toe, and the 8-puzzle). [11] deals with the supervised learning setting.

More closely related to our work, [12] studied automatic basis function construction for value function approximation. Given a set of trajectories and starting from an initial approximation, they iteratively use neighborhood component analysis to find a mapping from the state space to a low-dimensional space based on the estimation of the Bellman error, and then by discretizing this space aggregate states and use the resulting aggregation matrix to derive additional basis functions. This tends to aggregate states that are close to each other with respect to the Bellman error, leading to a better approximation by incorporating the corresponding basis functions.

[13] considers feature selection as a supervised classification problem, within the context of approximate value

iteration. Their approach is thus also very different from ours.

III. Non parametric function approximation

At this point, we discuss the non parametric function approximators. Among the many variants that have been published, a particularly important class is that of kernel methods. An other non parametric function approximator is the cascade-correlation networks. We shortly describe both approaches below.

A. Kernel methods

Kernel methods are indeed very old ideas from statistics. Nearest neighbors, and Parzen windows are closely related, and kernel methods are actually descendants of those techniques. Given a function to be learned $V : \mathcal{X} \mapsto \mathbb{R}$ from a set of N examples, that is N couples (x_i, y_i), a kernel method aims at finding an approximator $\hat{V}$ of the form $\hat{V}(x) = \sum_j \omega_j k(x, x_j)$, where the set of x_j is a subset of the example set, the ω_j's are real parameters, and k, a kernel function, that is merely a function defined as $\mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$. Such a function can best be considered as a distance, or a dissimilarity function; we may impose certain conditions on k[3] such as symmetry, and positive definitiveness[4], but we do not make this kind of assumptions here. Now, the non parametric aspect comes into play as the sum of terms defining $\hat{V}$ is changing over time; initially very simple, maybe a mere constant, new terms are added, and possibly removed, from the sum as examples are acquired. To avoid having as many terms in the sum as the number of examples, regularization techniques may be employed to balance the number of terms in the sum, and the accuracy of the prediction made by the approximator.

Stated with words, such kernel methods can be seen as one hidden layer perceptrons, the input of the perceptron being the state variable, the hidden units being these kernel functions, and the output being linear. Starting from a network with no hidden unit, new hidden units are added incrementally, as needed. Excellent performance have been obtained in supervised learning, classification, and regression [14], [15], as well as in reinforcement learning (see *e.g.* [16], [17] as representators of two different approaches of this same idea, in the context of reinforcement learning). This can also be seen as an RBF-based algorithm, where the number, the center, and the hyper-parameters of the basis are not set *a priori*, but evolves during learning.

In the next section, we very briefly describe an algorithm we have introduced, namely the Equi-Correlated Network, which is a descendant of, and a significant improvement over, the Equi-Gradient descent algorithm [17].

[3]Certain authors do impose these conditions on the name "kernel" method, but we do not share this point of view.

[4]Meeting these conditions for k opens the door to inheriting very nice mathematical properties, which in turn, may be turned into, more or less, efficient algorithm, in particular, optimizing algorithm that converges towards an optimum guaranteed globally optimal.

1) The Equi-Correlated Network: The Equi-Gradient Algorithm is a supervised learning algorithm based on the idea of the LARS algorithm [18]. Originally proposed in a supervised learning setting, this algorithm has been later applied to feature discovery, and kernelized. Then, it has been cast to the reinforcement learning framework, leading to the Equi-Gradient Descent algorithm [17]; since then, it has been enhanced in order to automatically tune the kernel function (hyper-)parameters [19].

The basic idea is a strict application of the principle of kernel methods stated above: given a set of examples, an estimator is built incrementally, in the form $\sum_j \omega_j k_j$, the kernels being added, or removed, one at a time. The objective function is composed of two terms: one is the l_2 error measure on examples, while the second is an l_1 regularization on the ω's; these two terms are combined by way of a regularization coefficient λ which gives more or less importance to either of these two criteria. To avoid choosing this parameter *a priori*, the LARS algorithm has been proposed which, very efficiently, computes the whole regularization path, that is, the set of all solutions (λ, Ω), where Ω denotes the set of the coefficients ω of the estimator matched to λ. This set is actually finite. Using an l_1 regularization, we obtain very sparse solutions, that is, a function approximator that has a rather small number of terms (one property of the l_1 regularization with regards to l_2 is that, instead of keeping ω's as almost 0, most of the parameters are actually zero-ed). These non null terms are associated to features, the associated kernel function, and so, the LARS ends-up selecting the most relevant features to approximate a certain set of examples.

Typically Gaussian, the kernels have their covariance matrix to be tuned. In [19], the Equi-Corrleation Network (ECON) algorithm is proposed, which performs a global optimization to tune it automatically, using a global optimizing algorithm named Direct [20]. So, ECON provides an estimator in which the number of terms is minimized, and the hyper-parameters of the kernel are optimized.

B. Cascade-correlation networks

An other kind of non parametric function approximator is the cascade-correlation network architecture, introduced by [3]. In a CCN, the hidden units are again added one at a time, as required to meet an expected accuracy; however, instead of forming a mere sum, the hidden units are cascaded (see Fig. 2).

The growth of a CCN may be briefly summarized as follows:

0) Create P input nodes, 1 output node (with linear activation function) and connect the inputs to the output.
1) All connections leading to output neurons are trained on a sample set and corresponding weights (*i.e.*, only the input weights of output neurons) are determined by using an ordinary learning algorithm (such as "delta" rule, here we use the RPROP) until the error of the network no longer decreases. Note that, only the input weights of the output neuron are being trained, therefore there is no back-propagation.

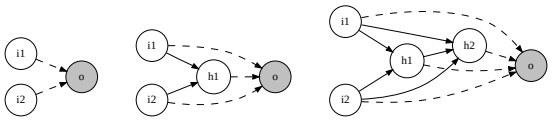

Fig. 2. Illustration of the growth of a cascade-correlation network. Initially (a), the CCN has only an input layer (here made of two inputs i1, and i2) and an output o (in gray). Both input, and output units may be either linear, or non linear. Then, if the accuracy of this network is below a given threshold, a hidden unit is added according to the procedure detailed in the text (b). Iteratively, as long as the expected accuracy is not reached, one hidden unit is added, one after the other (c). At each iteration of the growth, only the weight of the connections to the output node is trained (dashed connections), while the others are frozen once for all (solid connections). We see on this sketch that only the connections to the output node are trained, so that no back-propagation is required.

2) If the accuracy of the network is above a given threshold then the process terminates.
3) Otherwise, a set of *candidate units* is created. These units typically have non-linear activation functions, such as sigmoid or Gaussian. Every candidate unit is connected with all input neurons and with all existing hidden neurons (which is initially empty); the weights of these connections are initialized randomly. At this stage the candidate units are not connected to the output neuron, and therefore they are not actually active in the network. Let s denote a training sample. The connections leading to a candidate unit are trained with the goal of maximizing $\mathcal{S}$, the correlation between the candidate units value denoted by v_s, and the residual error observed at the output o denoted by $e_{s,o}$. S is defined as $S = |\sum_s (v_s - v)(e_{s,o} - e_o)|$ where v and e_o are the values of v_s and $e_{s,o}$ averaged over all samples, respectively. As in step 1, learning takes place with an ordinary learning algorithm by performing gradient ascent with respect to each of the candidate units incoming weights $\partial S/\partial w_i = \sum_{s,o}(e_{s,o} - e_o)\sigma_o f'_s I_{i,s}$ where σ_o is the sign of the correlation between the candidates value and output o, f'_s is the derivative for sample s of the candidate units activation function with respect to the sum of its inputs, and $I_{i,s}$ is the input the candidate unit received from neuron i for sample s[5]. The learning of candidate unit connections stops when the correlation scores no longer improve or after a certain number of passes over the training set. Now, the candidate unit with the maximum correlation is chosen, its incoming weights are frozen (i.e. they are not updated in the subsequent steps) and it is added permanently to the network by connecting it to the output neuron (Fig. 2b and c, solid lines show frozen weights). All other candidate units are discarded.
4) Return back to step 1.

Until the desired accuracy is achieved at step 2, or the number of neurons reaches a given maximum limit, a CCN completely self-organizes itself. By adding hidden neurons one at a time and freezing their input weights, training of both the input weights of the output neuron (step 1) and the input weights of candidate units (step 3) reduce to one step learning problems. By training candidate nodes with different activation functions and choosing the best among them, it is possible to build a more compact network that better fits the training data[6]. In this setting, each hidden node becomes a feature and the network itself acts as a linear function approximator based on these features.

[21] advocates for the use of CCN to solve the reinforcement learning problem on-line. Dealing with a bit more elaborated CCN than us, it is shown that a CCN using much less hidden units than a multi-layer perceptron, and even much less units that a RAN, often provides better policies.

C. Conclusion on non parametric function approximators

We have presented two different non parametric function approximators. Let us add that both are stochastic algorithms. Despite their differences, the crucial point we wish to make here is that the hidden units being added, or removed, corresponding to terms being added, or removed from $\hat{V}$, can be seen as features induced by the approximator. These features are added when the complexity of the examples can not be met by the current architecture of the approximator. In turn, in (some of) the kernel methods, features are automatically removed as soon as they become unnecessary.

This entire process is well-matched to our goal of determining a set of good basis functions for function approximation in ADP.

One last remark: both CCN and ECON yield a linear function approximator, which is more stable than a non linear approximator, and easier to train. Yet, using non linear functions as basis functions, we keep the ability to represent highly non linear functions.

IV. Non parametric function approximation and Approximate Dynamic Programming

In this section, we discuss the embedding of a non parametric function approximator into an approximate value iteration algorithm. This discussion will be quite short since it simply amounts to tie things together, *i.e.* the non parametric function approximators, and the approximate dynamic programming algorithms together.

A. Non parametric function approximation and Approximate Value Iteration

The AVI may be sketched as follows:

- $k \leftarrow 0$

[5]Note that, since only the input weights of candidate units are being trained there is again no need for back-propagation. Besides, it is also possible to train candidate units in parallel since they are not connected to each other. By training multiple candidate units instead of a single one, different parts of the weight space can be explored simultaneously. This consequently increases the probability of finding neurons that are highly correlated with the residual error.

[6]Furthermore, with deterministic activation functions the output of a neuron stays constant for a given sample input; hence, the number of calculations in the network can be reduced by storing the output values of neurons, improving the efficiency compared to traditional networks.

- initialize a regressor: this implicitly initialize a first estimate of $\hat{V}_k$, at random,
- build a set of sample states $\mathcal{S}$,
- while some condition unfulfilled, iterate:
  - $k \leftarrow k+1$
  - for each $x \in \mathcal{S}$, compute

$$W(x) \leftarrow \max_a \sum_{x'} \mathcal{P}(x,a,x')[\mathcal{R}(x,a,x')+\gamma \hat{V}(x')]$$

  - regress data $(\mathcal{S}, W)$: this yields the new $\hat{V}_k$.

In the sequel, we name the while loop the ADP iterations, to distinguish them from the iterates made during the regression.

At the end of this algorithm, we get the greedy policy based on the last estimate of the value of the optimal policy, $\hat{V}_k$.

The determination of the set of sample states is a delicate point. It may be done very easily by using the states located at the intersection of a regular grid on the state space. However, it is clear that a non regular location would in general be much better, so that sample states are located in "important" areas of the value to compute. Nonetheless, this requires some knowledge on this function that we do not know, and it turns out that the determination of the best location of the sample states is as difficult as computing the function itself. In the sequel, we thus resign ourselves to use a set of samples located at the intersection of a regular grid.

V. Experimental assessment

In this section, we report on the application of the proposed approach on a continuous Markov decision problem, namely the inverted pendulum, which is closely related to the cart-pole problem discussed in the introduction. This part only contains preliminary results, but we think that they are quite interesting, and significant.

A. Settings

For the inverted pendulum problem (see Fig. 3), the state space is $\mathcal{X} \equiv \{(\theta, \dot{\theta}) \in [-\pi, \pi] \times [-3\pi, 3\pi]\}$, 3π being considered as a maximum angular velocity. There are two actions, $\mathcal{A} = \{-5, +5\}$, in ISO units. The system is deterministic; the reward function is defined as $\mathcal{R}(x,a) = \cos(\theta)$. The time is discretized into steps of 0.03 s. We follow exactly the definition of the problem used in [22], the only difference being that we consider a discrete set of instants, that is $T \subset \mathbb{N}$. In all experiments, the discount factor is set to 0.9.

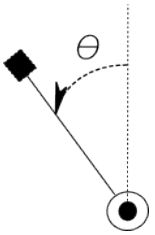

Fig. 3. The inverted-pendulum problem.

We run the algorithm, and assess the result by measuring the performance of a policy which is greedy for the current estimate of the value function, on a trajectory of 250 steps. Each test trajectory begins with the pendulum in a downward position, with null velocity (*i.e.*, $\theta = \pi, \dot{\theta} = 0$). We report the sum of the rewards, undiscounted to better distinguish between good and bad trajectories: indeed, even with an optimal policy, it takes some iterations to reach the equilibrium position so that even if a perfect equilibrium follows, the fact that the reward would be discounted by a very small number would make it difficult to interpret results. With this measure, 250 is obviously a upper bound, since it does not take into account the sub-optimal states that have to be visited between the initial position, and the equilibrium position. -250 is the lower bound, which could be reached if the pendulum remained at its initial position.

During ADP, the estimation of the value function is made by computing the value of a set of states, located on a grid which $\Delta_\theta = \Delta_{\dot{\theta}} = \pi/10$, that is 1400 sample states. We perform 50 ADP iterations. Regarding the AVI+CCN, each iteration is made of 15 phases of growth, each phase selecting the best out of 15 candidate nodes. So, this means that each value function is represented by a CCN with 15 hidden nodes. Regarding the AVI+ECON version, at each iteration, the ECON regressor is run until it can no longer improve its accuracy, 100 steps to the most. This means that the value function is represented by at most 100 terms.

B. Results obtained with the AVI+CCN algorithm

For the AVI+CCN version, Figure 4 shows some typical results. The performance of the greedy policy with regards to the current estimate of the value function is plotted against the iteration. This shows four things:

- after just a few iterations (5 on the plot), a very good performance of 216 is reached,
- extremely sharp changes in performance between two subsequent ADP iterations,
- the performance basically oscillates,
- the performance is never below 0.

So, we keep in mind that a quite very good approximation is obtained very quickly, and that this approximation only requires 5 hidden units. The oscillations are very likely due to the fact that we are using a function approximator (whether non parametric or parametric does not matter here). This behavior has to be investigated further. It may also be due to the small number of candidate units (only 15).

We have tried to lessen the number of sample points that are used to regress the value function of the current policy. With only 250 samples, 30 ADP iterations of 5 iterations each, choosing the best candidate among 6 also yields good performance very fast: after 11 such ADP iterations, so using 11 hidden units, a performance of 216 is obtained.

Finally, it is interesting to compare these results obtained on the planning task, with the results obtained on exactly the same task, but in a learning setting. For the learning

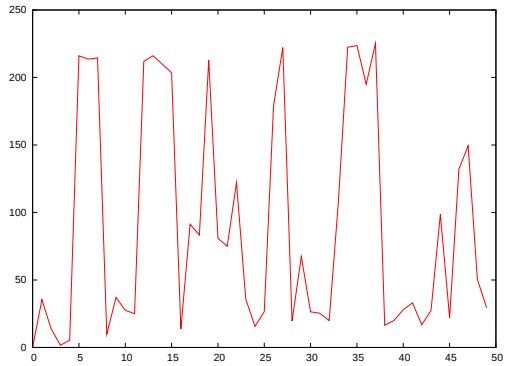

Fig. 4. Performance of the ADP+CCN algorithm on the inverted pendulum task. The x-axis shows the ADP iteration, that is also, the number of hidden units in the CCN; the y-axis is the performance of the resulting greedy policy, where the maximum that may be obtained is 250. Two striking points are: policies performing very well are obtained very quickly, hence with small CCN; the performance is rather unstable along ADP iterations. See text for more explanations, and comments.

task, a CCN represents the current value function, embedded into LSPI (see [7] for more details). We also compare the performance obtained with a parametric approximator embedded into LSPI, namely a regular grid, made of 514 RBF. Using 5000 samples, the learning with a regular RBF grid (parametric) representation of the value function led to a performance of 91; using the same 5000 samples, the LSPI with CCN embedded led to a performance of approximately 120, with 40 hidden units. These figures are thus to be compared with the aforementioned performance of 216, obtained by a CCN with 15 hidden units, after 5 iterations, in the planning setting.

C. Results obtained on the AVI+ECON algorithm

In the ECON, we use multi-dimensional Gaussian kernels only, so that the (hyper-)parameters are made of the entries of the covariance matrix. We run different variants of the ECON algorithm:

- BASIC ECON: which does not perform kernel function parameter tuning: we have to provide a parameter σ and the covariance matrix of all the kernels is set to σI, where I is the identity matrix,
- ISO ECON: which tunes the parameters of each Gaussian kernel, keeping its shape symetric: so the covariance matrix is again a multiple of the identity matrix, σI, but the coefficient σ is tuned automatically,
- DIAG ECON: in which the Gaussian kernels are no longer restricted to symetric ones; the covariance matrix is still diagonal, but no longer a multiple of the identity matrix, hence of the form $\text{diag}(\sigma_1, \sigma_2, ...\sigma_P)$.

We expect that the versions tuning the parameters perform better, the non symetric one (DIAG) even better than ISO.

That is exactly what we obtained experimentally. Figure 5 shows that the BASIC version performs quite bad, ISO better, and DIAG performs the best. If we focus on the DIAG variant which performs the best, we see that the performance of the greedy policy with regards to the current value function estimate basically steadily performs better and better; this is a striking difference with what we observed when using the CCN, where larger, and more irregular, oscillations were observed; with the ECON, the more ADP iterates, the better the policies. In particular, during the second half ADP iterates, a majority of iterates yield reasonably well-performing policies (accumulated rewards, along a trajectory of 250 steps, lie in the range 100-150). However, the best policies found by the ECON are not as good as those found with the CCN; as can be seen by comparing Figures 4 and 5, ECON has not been able to yield policies performing better than 127, while it often occurs that the policy found with the CCN is above 200. We have not yet any explanation about that; maybe ECON needs more ADP iterations than CCN to yield such good policies; maybe the function approximated by the CCN better fits the value function of the inverted pendulum, than the one provided by a ECON. We also noticed that the best policies are obtained by ECON using 100 kernel functions (the limit number we had set); this is also in striking difference with the CCN which yields much better performing policies with only 5 hidden units. To investigate this issue, we have run both CCN and ECON algorithms as mere regressors, on a very good approximation of the optimal value function of this problem. After each iteration of the algorithm, that is after adding one hidden unit to the CCN, or one kernel to the ECON, we measure the mean squared error. The result is displayed at Figure 6: it is striking how the CCN obtains a very accurate approximation of the function after just a few iterations, whereas ECON never reaches the same accuracy, even after 300 iterations.

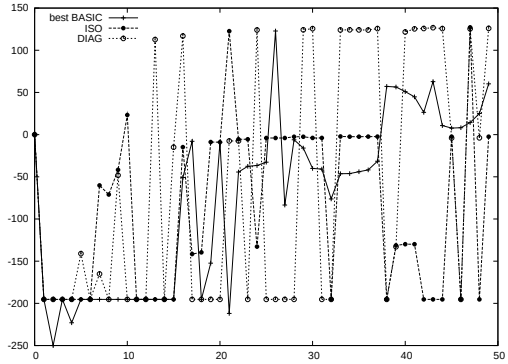

Fig. 5. Performance of three variants of the AVI+ECON algorithm on the inverted pendulum task. As in figure 4, the x-axis shows the ADP iteration, and the y-axis is the performance of the resulting greedy policy, where the maximum that may be obtained is 250. 3 variants of ECON are plotted. The best variant (DIAG) shows more stability than the CCN version; however, the performance is less here than with the CCN version. See text for more explanations, and comments.

D. General remarks about the experimental results

As features are discovered, it would be interesting that they make sense for a human being. As it is clear from the form of the estimator that yield the CCN, and the ECON, the hidden units of the CCN are extremely difficult to interpret. The situation is very different with the ECON; indeed, as a

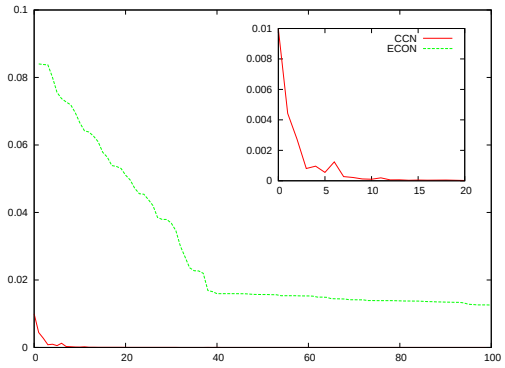

Fig. 6. We consider the task of regressing the inverted pendulum optimal value function with the CCN, and the ECON and we plot the MSE against the iteration of the algorithm. CCN yields a high accuracy after a very small number of iterations, and this accuracy is consistently much higher than with the ECON algorithm. See text for more details.

mere sum of Gaussians, the kernel location and shape may be understood quite easily; these are very well correlated with the flatness and ridges of the function being regressed.

One last word about computational time: for both algorithm, it is rather fast, 50 AVI iterations being run in about one minute on a standard 1.6 GHz laptop, except for the ISO and DIAG which are an order of magnitude slower.

VI. Conclusion and Future Work

In this paper, we first argued that the function approximation problem, and the feature discovery problem may best be seen as being the two faces of the same coin. Based on that, we put forward the non parametric function approximators which, despite the lack of strong theoretical basis, have demonstrated excellent performance in various areas of machine learning. Following our work on the reinforcement learning problem, we discussed how such an approximator may be embedded into approximate dynamic programming. Then, some experimental results were provided, obtained by an approximate value iteration algorithm, either with a cascade-correlation network embedded in it, or a kernel method we designed, named ECON; in each case, the non parametric function approximator represents the value function by selecting useful features. These experiments show that this approach is able to obtain very fast good policies, using a very sparse representation, made of a few hidden units.

Various tracks will be followed in the near future. We will first complete the experimental assessment of our propositions, by making it more thorough, and on different problems. We will also study different variants of the approximate dynamic programming algorithm, only the simplest one having been studied here. In particular, we would like to tackle larger state spaces. One aspect of feature discovery is to provide features that may be understood by human beings: it is clear that the cascaded hidden units of a CCN do not meet this expectation. In a kernel method, the features are merely weighted and added up to make the estimation, not cascaded, so that it is possible to get some information from their location in the state space, and their shape.

References

[1] I. Guyon and A. Elisseeff, *An introduction to feature extraction*. Springer, 2006.

[2] S. Girgin and P. Preux, "Feature discovery in reinforcement learning using genetic programming," in *Proc. 11^{th} European Conference on Genetic Programming (EuroGP)*, ser. Lecture Notes in Computer Science, M. O'Neill, L. Vanneschi, S. Gustafson, A. Esparcia-Alcázar, I. D. Falco, A. D. Cioppa, and E. Tarantino, Eds., vol. 4971. Springer, 2008, pp. 218–229.

[3] S. E. Fahlman and C. Lebière, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems*, D. S. Touretzky, Ed., vol. 2. Denver 1989: Morgan Kaufmann, San Mateo, 1990, pp. 524–532.

[4] M. Puterman, *Markov Decision Processes — Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[5] D. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific, 1996.

[6] W. Powell, *Approximate Dynamic Programming*. Wiley, 2007.

[7] S. Girgin and P. Preux, "Incremental basis function expansion in reinforcement learning using cascade-correlation networks," in *Proc. of the 8^{th} International Conference on Machine Learning and Applications (ICML-A)*. La Jolla, USA: IEEE Press, Dec. 2008.

[8] ——, "Basis expansion in natural actor critic methods," in *Recent Advances in Reinforcement Learning*, ser. Lecture Notes in Artificial Intelligence (LNAI), S. Girgin, M. Loth, R. Munos, P. Preux, and D. Ryabko, Eds., vol. 5323. Springer, 2008, pp. 111–124.

[9] J. N. Tsitsiklis and B. V. Roy, "Feature-based methods for large scale dynamic programming," *Machine Learning*, vol. 22, pp. 59–94, 1996.

[10] P. Utgoff and D. Precup, "Constructive function approximation," in *Feature extraction, construction, and selection: A data-mining perspective*. Kluwer, 1998, pp. 219–235.

[11] S. Perkins, K. Lacker, and J. Theiler, "Grafting: Fast, incremental feature selection by gradient descent in function space," *Journal of Machine Learning Research*, vol. 3, pp. 1333–1356, 2003.

[12] P. Keller, S. Mannor, and D. Precup, "Automatic basis function construction for approximate dynamic programming and reinforcement learning," in *Proc. of the 23^{rd} International Conference on Machine Learning (ICML)*. NY, USA: ACM, 2006, pp. 449–456.

[13] J.-H. Wu and R. Givan, "Feature-discovering approximate value iteration methods," in *Abstraction, Reformulation and Approximation*, ser. Lecture Notes in Computer Science, vol. 3607. Springer, 2005, pp. 321–331.

[14] J. Platt, "A resource-allocating n?etwork for function interpolation," *Neural Computation*, vol. 3, no. 2, pp. 213–225, 1991.

[15] G.-B. Huang, P. Saratchandran, and N. Sundararajan, "A generalized growing and pruning RBF (GGAP-RBF) neural netowrk for function approximation," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 57–67, Jan. 2005.

[16] S. Vijayakumar, A. D'Souza, and S. Schaal, "Incremental online learning in high dimensions," *Neural Computation*, vol. 17, no. 12, pp. 2602–2632, 2002.

[17] M. Loth, M. Davy, and P. Preux, "Sparse temporal difference learning using LASSO," in *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE Press, Apr. 2007, pp. 352–359.

[18] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least-angle regression," *Annals of statistics*, vol. 32, no. 2, pp. 407–499, 2004.

[19] M. Loth and P. Preux, "The equi-correlation network: A new kernelized-lars with automatic kernel parameters tuning," 2008, (submitted).

[20] D. Jones, C. Perttunen, and B. Stuckman, "Lipschitzian optimization without the lipschitz constant," *Journal of Optimization Theory and Aplications*, vol. 79, no. 1, pp. 157–181, Oct. 1993.

[21] P. Vamplew and R. Ollington, "Global versus local constructive function approximation for on-line reinforcement learning," in *AI 2005: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, vol. 3809. Springer, 2005, pp. 113–122.

[22] R. Coulom, "Reinforcement learning using neural networks with applications to motor control," Ph.D. dissertation, Institut National Polytechnique de Grenoble, France, 2002.