Neuro-Dynamic Programming: An Overview

Dimitri P. Bertsekas
bertsekas@lids.mit.edu

Laboratory for Information and Decision Systems
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

Neuro-dynamic programming (NDP for short) is a relatively new class of dynamic programming methods for control and sequential decision making under uncertainty. These methods have the potential of dealing with problems that for a long time were thought to be intractable due to either a large state space or the lack of an accurate model. They combine ideas from the fields of neural networks, artificial intelligence, cognitive science, simulation, and approximation theory. We will delineate the major conceptual issues, survey a number of recent developments, describe some computational experience, and address a number of open questions.

We consider systems where decisions are made in stages. The outcome of each decision is not fully predictable but can be anticipated to some extent before the next decision is made. Each decision results in some immediate cost but also affects the context in which future decisions are to be made and therefore affects the cost incurred in future stages. Dynamic programming (DP for short) provides a mathematical formalization of the tradeoff between immediate and future costs.

Generally, in DP formulations there is a discrete-time dynamic system whose state evolves according to given transition probabilities that depend on a decision/control u. In particular, if we are in state i and we choose decision u, we move to state j with given probability $p_{ij}(u)$. Simultaneously with this transition, we incur a cost $g(i, u, j)$. In comparing, however, the available decisions u, it is not enough to look at the magnitude of the cost $g(i, u, j)$; we must also take into account how desirable the next state j is. We thus need a way to rank or rate states j. This is done by using the optimal cost (over all remaining stages) starting from state j, which is denoted by $J^*(j)$. These costs can be shown to

satisfy some form of Bellman's equation

$$J^*(i) = \min_u E\{g(i,u,j) + J^*(j) \mid i,u\}, \qquad \text{for all } i,$$

where j is the state subsequent to i, and $E\{\cdot \mid i,u\}$ denoted expected value with respect to j, given i and u. Generally, at each state i, it is optimal to use a control u that attains the minimum above. Thus, decisions are ranked based on the sum of the expected cost of the present period, and the optimal expected cost of all subsequent periods.

The objective of DP is to calculate numerically the optimal cost function J^*. This computation can be done off-line, i.e., before the real system starts operating. An optimal policy, that is, an optimal choice of u for each i, is computed either simultaneously with J^*, or in real time by minimizing in the right-hand side of Bellman's equation. It is well known, however, that for many important problems the computational requirements of DP are overwhelming, mainly because of a very large number of states and controls (Bellman's "curse of dimensionality"). In such situations a suboptimal solution is required.

Cost Approximations in Dynamic Programming

NDP methods are suboptimal methods that center around the approximate evaluation of the optimal cost function J^*, possibly through the use of neural networks and/or simulation. In particular, we replace the optimal cost $J^*(j)$ with a suitable approximation $\tilde{J}(j,r)$, where r is a vector of parameters, and we use at state i the (suboptimal) control $\tilde{\mu}(i)$ that attains the minimum in the (approximate) right-hand side of Bellman's equation

$$\tilde{\mu}(i) = \arg\min_u E\{g(i,u,j) + \tilde{J}(j,r) \mid i,u\}.$$

The function $\tilde{J}$ will be called the *scoring function*, and the value $\tilde{J}(j,r)$ will be called the *score* of state j. The general form of $\tilde{J}$ is known and is such that once the parameter vector r is determined, the evaluation of $\tilde{J}(j,r)$ of any state j is fairly simple.

We note that in some problems the minimization over u of the expression

$$E\{g(i,u,j) + \tilde{J}(j,r) \mid i,u\}$$

may be too complicated or too time-consuming for making decisions in real-time, even if the scores $\tilde{J}(j,r)$ are simply calculated. In such problems we may use a related technique, whereby we approximate the expression minimized in Bellman's equation,

$$Q(i,u) = E\{g(i,u,j) + J^*(j) \mid i,u\},$$

which is known as the *Q-factor corresponding to* (i,u). In particular, we replace $Q(i,u)$ with a suitable approximation $\tilde{Q}(i,u,r)$, where r is a vector of

parameters. We then use at state i the (suboptimal) control that minimizes the approximate Q-factor corresponding to i:

$$\tilde{\mu}(i) = \arg\min_u \tilde{Q}(i, u, r).$$

Much of what will be said about approximation of the optimal cost function also applies to approximation of Q-factors. In fact, we will see later that the Q-factors can also be viewed as optimal costs of a related problem. We thus focus primarily on approximation of the optimal cost function J^*.

We are interested in problems with a large number of states and in scoring functions $\tilde{J}$ that can be described with relatively few numbers (a vector r of small dimension). Scoring functions involving few parameters are called *compact representations*, while the tabular description of J^* are called the *lookup table representation*. Thus, in a lookup table representation, the values $J^*(j)$ are stored in a table for all states j. In a typical compact representation, only the vector r and the general structure of the scoring function $\tilde{J}(\cdot, r)$ are stored; the scores $\tilde{J}(j, r)$ are generated only when needed. For example, $\tilde{J}(j, r)$ may be the output of some neural network in response to the input j, and r is the associated vector of weights or parameters of the neural network; or $\tilde{J}(j, r)$ may involve a lower dimensional description of the state j in terms of its "significant features", and r is the associated vector of relative weights of the features. Thus determining the scoring function $\tilde{J}(j, r)$ involves two complementary issues: (1) deciding on the general structure of the function $\tilde{J}(j, r)$, and (2) calculating the parameter vector r so as to minimize in some sense the error between the functions $J^*(\cdot)$ and $\tilde{J}(\cdot, r)$.

Approximations of the optimal cost function have been used in the past in a variety of DP contexts. Chess playing programs represent a successful example. A key idea in these programs is to use a *position evaluator* to rank different chess positions and to select at each turn a move that results in the position with the best rank. The position evaluator assigns a numerical value to each position, according to a heuristic formula that includes weights for the various features of the position (material balance, piece mobility, king safety, and other factors). Thus, the position evaluator corresponds to the scoring function $\tilde{J}(j, r)$ above, while the weights of the features correspond to the parameter vector r. Usually, some general structure of position evaluator is selected (this is largely an art that has evolved over many years, based on experimentation and human knowledge about chess), and the numerical weights are chosen by trial and error or (as in the case of the champion program Deep Thought) by "training" using a large number of sample grandmaster games.

As the chess program paradigm suggests, intuition about the problem, heuristics, and trial and error are all important ingredients for constructing cost approximations in DP. However, it is important to supplement heuristics and intuition with more systematic techniques that are broadly applicable and retain as much as possible the nonheuristic aspects of DP.

NDP aims to develop a methodological foundation for combining dynamic programming, compact representations, and simulation to provide the basis for a rational approach to complex stochastic decision problems.

Approximation Architectures

An important issue in function approximation is the *selection of architecture*, that is, the choice of a parametric class of functions $\tilde{J}(\cdot, r)$ or $\tilde{Q}(\cdot, \cdot, r)$ that suits the problem at hand. One possibility is to use a neural network architecture of some type. We should emphasize here that in this article we use the term "neural network" in a very broad sense, essentially as a synonym to "approximating architecture." In particular, we do not restrict ourselves to the classical multilayer perceptron structure with sigmoidal nonlinearities. Any type of universal approximator of nonlinear mappings could be used in our context. The nature of the approximating structure is left open in our discussion, and it could involve, for example, radial basis functions, wavelets, polynomials, splines, etc.

Cost approximation can often be significantly enhanced through the use of *feature extraction*, a process that maps the state i into some vector $f(i)$, called the *feature vector* associated with the state i. Feature vectors summarize, in a heuristic sense, what are considered to be important characteristics of the state, and they are very useful in incorporating the designer's prior knowledge or intuition about the problem and about the structure of the optimal controller. For example in a queueing system involving several queues, a feature vector may involve for each queue a three-value indicator, that specifies whether the queue is "nearly empty", "moderately busy", or "nearly full". In many cases, analysis can complement intuition to suggest the right features for the problem at hand.

Feature vectors are particularly useful when they can capture the "dominant nonlinearities" in the optimal cost function J^*. By this we mean that $J^*(i)$ can be approximated well by a "relatively smooth" function $\hat{J}(f(i))$; this happens for example, if through a change of variables from states to features, the function J^* becomes a (nearly) linear or low-order polynomial function of the features. When a feature vector can be chosen to have this property, one may consider approximation architectures where both features and (relatively simple) neural networks are used together. In particular, the state is mapped to a feature vector, which is then used as input to a neural network that produces the score of the state. More generally, it is possible that both the state and the feature vector are provided as inputs to the neural network.

A simple method to obtain more sophisticated approximations, is to partition the state space into several subsets and construct a separate cost function approximation in each subset. For example, by using a linear or quadratic polynomial approximation in each subset of the partition, one can construct piecewise linear or piecewise quadratic approximations over the entire state space. An important issue here is the choice of the method for partitioning the state space. Regular partitions (e.g., grid partitions) may be used, but they often lead to a large number of subsets and very time-consuming computations. Generally

speaking, each subset of the partition should contain "similar" states so that the variation of the optimal cost over the states of the subset is relatively smooth and can be approximated with smooth functions. An interesting possibility is to use features as the basis for partition. In particular, one may use a more or less regular discretization of the space of features, which induces a possibly irregular partition of the original state space. In this way, each subset of the irregular partition contains states with "similar features."

Simulation and Training

Some of the most successful applications of neural networks are in the areas of pattern recognition, nonlinear regression, and nonlinear system identification. In these applications the neural network is used as a universal approximator: the input-output mapping of the neural network is matched to an unknown nonlinear mapping F of interest using a least-squares optimization. This optimization is known as *training the network*. To perform training, one must have some training data, that is, a set of pairs $(i, F(i))$, which is representative of the mapping F that is approximated.

It is important to note that in contrast with these neural network applications, in the DP context there is no readily available training set of input-output pairs $(i, J^*(i))$, which can be used to approximate J^* with a least squares fit. The only possibility is to evaluate (exactly or approximately) by simulation the cost functions of given (suboptimal) policies, and to try to iteratively improve these policies based on the simulation outcomes. This creates analytical and computational difficulties that do not arise in classical neural network training contexts. Indeed the use of simulation to evaluate approximately the optimal cost function is a key new idea, that distinguishes the methodology of this article from earlier approximation methods in DP.

Using simulation offers another major advantage: it allows the methods of this article to be used for systems that are hard to model but easy to simulate; that is, in problems where an explicit model is not available, and the system can only be observed, either as it operates in real time or through a software simulator. For such problems, the traditional DP techniques are inapplicable, and estimation of the transition probabilities to construct a detailed mathematical model is often cumbersome or impossible.

There is a third potential advantage of simulation: it can implicitly identify the "most important" or "most representative" states of the system. It appears plausible that if these states are the ones most often visited during the simulation, the scoring function will tend to approximate better the optimal cost for these states, and the suboptimal policy obtained will perform better.

Neuro-Dynamic Programming

The name *neuro-dynamic programming* expresses the reliance of the methods of this article on both DP and neural network concepts. In the artificial intelligence community, where the methods originated, the name *reinforcement*

learning is also used. In common artificial intelligence terms, the methods allow systems to "learn how to make good decisions by observing their own behavior, and use built-in mechanisms for improving their actions through a reinforcement mechanism." In more mathematical terms, "observing their own behavior" relates to simulation, and "improving their actions through a reinforcement mechanism" relates to iterative schemes for improving the quality of approximation of the optimal cost function, or the Q-factors, or the optimal policy. There has been a gradual realization that reinforcement learning techniques can be fruitfully motivated and interpreted in terms of classical DP concepts such as value and policy iteration; see the nice survey by Barto, Bradtke, and Singh [BBS93], and the book by Sutton and Barto [SuB98], which point out the connections between the artificial intelligence/reinforcement learning viewpoint and the control theory/DP viewpoint, and give many references.

Two fundamental DP algorithms, policy iteration and value iteration, are the starting points for the NDP methodology. The most straightforward adaptation of the policy iteration method operates as follows: we start with a given policy (some rule for choosing a decision u at each possible state i), and we approximately evaluate the cost of that policy (as a function of the current state) by least-squares-fitting a scoring function $\tilde{J}(\cdot, r)$ to the results of many simulated system trajectories using that policy. A new policy is then defined by minimization in Bellman's equation, where the optimal cost is replaced by the calculated scoring function, and the process is repeated. This type of algorithm typically generates a sequence of policies that eventually oscillates in a neighborhood of an optimal policy. The resulting deviation from optimality depends on a variety of factors, principal among which is the ability of the architecture $\tilde{J}(\cdot, r)$ to accurately approximate the cost functions of various policies (the book by Bertsekas and Tsitsiklis [BeT96] makes this point more precise).

The approximate policy iteration method described above calculates many simulated sample trajectories before changing the parameter vector r of the scoring function $\tilde{J}(j, r)$. Another popular NDP methodology adjusts the parameter vector r more frequently, as it produces sample state trajectories

$$(i_0, i_1, \ldots, i_k, i_{k+1}, \ldots,).$$

These trajectories correspond to either a fixed policy, or to a "greedy" policy that applies, at state i, the control u that minimizes the expression

$$E\{g(i, u, j) + \tilde{J}(j, r) \mid i, u\},$$

where r is the current parameter vector. A central notion here is the notion of a *temporal difference*, defined by

$$d_k = g(i_k, u_k, i_{k+1}) + \tilde{J}(i_{k+1}, r) - \tilde{J}(i_k, r),$$

and expressing the difference between our expected cost estimate $\tilde{J}(i_k, r)$ at state i_k and the predicted cost estimate $g(i_k, u_k, i_{k+1}) + \tilde{J}(i_{k+1}, r)$ based on the

outcome of the simulation. If the cost approximations were exact, the average temporal difference would be zero by Bellman's equation. Thus, roughly speaking, the values of the temporal differences can be used to make incremental adjustments to r so as to bring about an approximate equality (on the average) between expected and predicted cost estimates along the simulated trajectories. This viewpoint, formalized by Sutton in [Sut88], can be implemented through the use of gradient descent/stochastic approximation methodology. Sutton proposed a family of methods of this type, called TD(λ), and parameterized by a scalar $\lambda \in [0, 1]$. One extreme, TD(1), is related to policy iteration and least-squares parameter estimation, while the other extreme, TD(0), is related to value iteration and stochastic approximation. A related method is Q-learning, introduced by Watkins [Wat89], which is a stochastic approximation-like method that iterates on the Q-factors. While there is convergence analysis of TD(λ) and Q-learning for the case of lookup table representations (see Tsitsiklis [Tsi94]), the situation is much less clear in the case of compact representations.

A simpler type of NDP method, called *rollout*, is to approximate the optimal cost-to-go by the cost of some reasonably good suboptimal policy, called the *base policy*. Depending on the context, the cost of the base policy may be calculated either analytically, or more commonly by simulation. In a variant of the method, the cost of the base policy is approximated by using some approximation architecture. It is possible to view this method as a single step of a (possibly approximate) policy iteration method. The rollout approach is particularly simple to implement, and is also well-suited for on-line replanning, in situations where the problem parameters change over time. The rollout approach may also be combined with rolling horizon approaximations, and in some variations is related to *model predictive control*, and *receding horizon control*; see Keerthi and Gilbert [KeG88], the surveys by Morari and Lee [MoL99], and Mayne et. al. [MRR00], and the references quoted there. Despite being less ambitious than the approximate policy iteration and TD methods mentioned earlier, rollout algorithms have performed surprisingly well in a variety of studies and applications, often achieving a spectacular improvement over the base policy.

While the theoretical support for some of the NDP methodology has only recently emerged, there have been quite a few reports of successes with problems too large and complex to be treated in any other way. A particularly impressive success is the development of a backgammon playing program by Tesauro [Tes92]. Here a neural network provided a compact representation of the optimal cost function of the game of backgammon by using simulation and TD(λ). The training was performed by letting the program play against itself. After training for several months, the program nearly defeated the human world champion. Variations of the method used by Tesauro have been used in a variety of applications.

The recent experience of researchers, involving several engineering applications, has confirmed that NDP methods can be impressively effective in problems

where traditional DP methods would be hardly applicable and other heuristic methods would have a limited chance of success. We note, however, that the practical application of NDP is computationally very intensive, and often requires a considerable amount of trial and error. Fortunately, all the computation and experimentation with different approaches can be done off-line. Once the approximation is obtained off-line, it can be used to generate decisions fast enough for use in real time. In this context, we mention that in the machine learning literature, reinforcement learning is often viewed as an "on-line" method, whereby the cost approximation is improved as the system operates in real time. This is reminiscent of the methods of traditional adaptive control. We will not discuss this viewpoint, as we prefer to focus on applications involving a large and complex system. A lot of training data is required for such a system. These data typically cannot be obtained in sufficient volume as the system is operating; even if they can, the corresponding processing requirements are typically too large for effective use in real time.

Extensive references for the material of this article are the research monographs by Bertsekas and Tsitsiklis [BeT96], and by Sutton and Barto [SuB98]. A more limited textbook discussion is given in the DP textbook by Bertsekas [Ber95]. The 2nd edition of the first volume of this DP text [Ber00] contains a detailed discussion of rollout algorithms. The extensive survey by Barto, Bradtke, and Singh [BBS95], and the overviews by Werbös [Wer92a], [Wer92b], and other papers in the edited volume by White and Sofge [WhS92] point out the connections between the artificial intelligence/reinforcement learning viewpoint and the control theory/DP viewpoint, and give many references.

REFERENCES

[BBS95] Barto, A. G., Bradtke, S. J., and Singh, S. P., 1995. "Real-Time Learning and Control Using Asynchronous Dynamic Programming," Artificial Intelligence, Vol. 72, 1995, pp. 81-138.

[BeT96] Bertsekas, D. P., and Tsitsiklis, J. N., 1996. Neuro-Dynamic Programming, Athena Scientific, Belmont, MA.

[Ber95] Bertsekas, D. P., 1995. Dynamic Programming and Optimal Control, Vol. II, Athena Scientific, Belmont, MA.

[Ber00] Bertsekas, D. P., 2000. Dynamic Programming and Optimal Control, Vol. I, 2nd Edition, Athena Scientific, Belmont, MA.

[KeG88] Keerthi, S. S., and Gilbert, E. G., 1988. "Optimal, Infinite Horizon Feedback Laws for a General Class of Constrained Discete Time Systems: Stability and Moving-Horizon Approximations," J. Optimization Theory Appl., Vo. 57, pp. 265-293.

[MRR00] Mayne, D. Q., Rawlings, J. B., Rao, C. V., and Scokaert, P. O. M., 2000. "Constrained Model Predictive Control: Stability and Optimality," Automatica, Vol. 36, pp. 789-814.

[MoL99] Morari, M., and Lee, J. H., 1999. "Model Predictive Control: Past, Present, and Future," Computers and Chemical Engineering, Vol. 23, pp. 667-682.

[SuB98] Sutton, R. S., and Barto, A. G., 1988. Reinforcement Learning, MIT Press, Cambridge, MA.

[Sut88] Sutton, R. S., 1988. "Learning to Predict by the Methods of Temporal Differences," Machine Learning, Vol. 3, pp. 9-44.

[Tes92] Tesauro, G., 1992. "Practical Issues in Temporal Difference Learning," Machine Learning, Vol. 8, pp. 257-277.

[Tsi94] Tsitsiklis, J. N., 1994. "Asynchronous Stochastic Approximation and Q-Learning," Machine Learning, Vol. 16, pp. 185-202.

[Wat89] Watkins, C. J. C. H., "Learning from Delayed Rewards," Ph.D. Thesis, Cambridge Univ., England.

[Wer92a] Werbös, P. J, 1992. "Approximate Dynamic Programming for Real-Time Control and Neural Modeling," in D. A. White and D. A. Sofge, (eds.), Handbook of Intelligent Control, Van Nostrand, N. Y.

[Wer92b] Werbös, P. J, 1992. "Neurocontrol and Supervised Learning: an Overview and Valuation," in D. A. White and D. A. Sofge, (eds.), Handbook of Intelligent Control, Van Nostrand, N. Y.

[WhS92] White, D. A., and Sofge, D. A., (eds.), 1992. Handbook of Intelligent Control, Van Nostrand, N. Y.