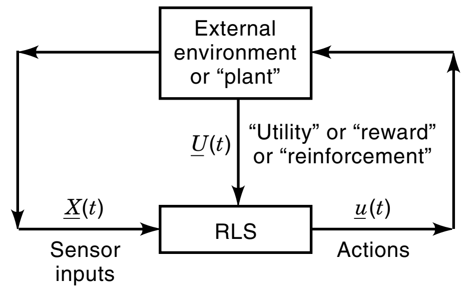

Figure 1. Reinforcement Learning Systems (RLS).

NEUROCONTROLLERS

NEUROCONTROL: AN OVERVIEW FOR THE PRACTITIONER*

What Is Neurocontrol?: The General Approach

Neurocontrol is a new branch of engineering practice and research, which first came together as an organized field in 1988 (1).

Actually, the philosophy behind neurocontrol dates back much earlier than 1988. Norbert Wiener (2) originally defined "cybernetics" as a kind of unified theory of control and communication in the animal and the machine. Neurocontrol tries to implement Wiener's original vision, by building control systems or decision-making systems which can *learn* to improve their performance over time, and can use a parallel distributed kind of computing hardware similar to what the brain uses.

For a long time, many engineers have been intrigued by the idea of developing an "all-purpose black box controller," which could be plugged into any application. The box would have wires going out to the actuators and other controllers, wires coming in from sensors, and a special wire coming in from the *utility module*—a system which monitors and measures the overall success of the controller, based on criteria which must be supplied by the user. The engineer using the box would have to worry about providing the right inputs and outputs, but the black box would figure out all the rest, based on learning. It would learn by itself how to maximize utility over future time, even if that requires developing a complex strategy in order to reach a desirable state. Many people now define *reinforcement learning* as the problem of designing this kind of black box (1,3), illustrated in Fig. 1.

Reinforcement learning systems (RLS) do exist today—but they vary greatly in quality, and they all have notable limitations. Still, there is a pathway now defined for future research which does seem to point the way, in concrete terms, to the development of future reinforcement learning systems which really could replicate the high level of intelligence and flexibility that exists in the brains of mammals. [Actually, performance is usually better with reinforcement learning designs in which the utility function is a known differentiable function of the other inputs. Such modified designs may even be more plausible as models of the brain. (4)] There has been considerable research demonstrating links between such RLS designs and the brains and behavior of various mammals (e.g., see Refs. (5) and (4,6–7); however, new partnerships between engineers and biologists will be crucial to a deeper understanding of these links.

As a practical matter, most control tasks today do not require full-fledged brains to perform them. There is a complex "ladder" of designs available in neurocontrol, rising up from simple designs of limited power through to very complex, more brainlike designs. Roughly speaking, they range in scope from designs which "clone" the observed behavior of an expert, through to designs for tracking setpoints or desired trajectories, through to full-scale designs to optimize goal satisfaction over time. Effective engineering groups usually start out by implementing the simpler designs, in general-purpose software, and then systematically climb up the ladder, one step at a time, to expand their capabilities and to reduce their costs in coping with ever more difficult applications. The key to effectiveness, then, is to know where one is on the ladder at any time, and to know what the choices and benefits are for the next step up. This requires making some effort to map out, decode, and unify a rather complex, fragmented literature, drawing from many different disciplines which use different terminology. This chapter will try to help the reader in this task.

Furthermore, in practical terms, real-time learning or "learning on the fly" is not always the most effective way to solve a control problem. We usually have three kinds of information about the plant available to us when we try to solve a control problem: (1) true prior information, such as a physical model of the plant to be controlled; (2) a database of datastreams for the sensors and actuators in the plant, datastreams which could be based on physical recordings or on simulation; and (3) the actual current stream of data from the plant which we are now controlling in real time. Statistical principles (8) suggest that the best controller will always be the one which combines all three sources of information in an optimal manner.

Roughly speaking, traditional control uses only the first source of knowledge in designing the controller. (Of course, even most traditional controllers will respond to sensor input *after* they have been designed and put into operation.) *Offline learning* in neurocontrol uses the second source of knowledge. *Real-time* learning in neurocontrol [and adaptive control (9,10)] uses the third. The challenge is to develop all three capabilities, and then find ways to blend (or select) all three across a range of applications. The simplest applications really do involve the *control* of a physical plant, like a furnace. The more complex applications may really involve making *de-*

* The views expressed herein are those of the author, not those of his employers, although the work was written on government time.

cisions so as to optimize or influence a much more complicated environment, like a factory or an entire economy.

Relations With Other Forms of Control

In the 1980s, neural network people and control theorists often expressed extreme emotional judgments about the strengths and weaknesses of neurocontrol versus conventional control. For example, some neural network people argued that neural networks could solve problems that mere mathematical approaches could not. Some control theorists argued that the reverse was true, and that all users of neural networks must be black box black magicians. Rhetoric like this has lingered on in some quarters, but a more concrete and reliable understanding has begun to emerge in the mainstreams of both fields.

The convergence actually began in 1988, in the National Science Foundation (NSF) workshop which first brought together people from different parts of this field, and injected the term "neurocontrol" (1). In that workshop, it became clear that the major designs being used in neurocontrol can actually be considered as special cases of more general learning designs within the field of control theory. (Unfortunately, some people associate control theory only with the simplest form of control, like thermostats; control theory in the broadest sense is really the theory of decision and control, including simple systems like thermostats, but also including nonlinear optimal control, stochastic control, and so on.)

To understand the concrete implications of this situation, consider the following analogy. The general learning control designs used in neurocontrol can be compared to circuit boards performing a higher-level function, containing some empty sockets where something has to be plugged in. For example, most of these learning control designs contain sockets where you must plug in some kind of general-purpose system which can learn to approximate nonlinear functions. Most people simply plug in some sort of artificial neural network (ANN) into these sockets. But you could just as well plug in an elastic fuzzy logic module (11), a Taylor series module, a soft gain scheduler, or a differentiable system of user-specified equations or transfer functions (3,12) into any one of these sockets if you know how to plug in all the associated information required (see chapter 8 of Ref. 10 or chapter 10 of Ref. 3.)

The learning control design itself—the circuit board—does not really contain any neural networks. Therefore, it may be slightly misleading to call these higher-level designs neurocontrollers. It would seem more precise to call them learning control designs or intelligent control designs. However, the terms intelligent control and learning control have been used in the past to refer to a wide variety of other designs, of varying degrees of real intelligence. In this article, the term *learning control* will refer to the specific types of generalized learning control design which are used in the neural network community; however, this is not standard terminology.

In practical applications, then, the design process here actually works at three levels. On the lowest level, we must decide which subsystems to plug in—the specific ANNs or other modules to perform function approximation or prediction or whatever. At the middle level, we must choose one or more higher-level learning designs, to perform general higher-level tasks like cloning a human expert, tracking a desired trajectory, or optimizing a performance measure. At the highest level, we must map these higher-level tasks into a real application, which often requires a variety of tasks to be performed, in parallel or in sequence.

Unfortunately, many neurocontrol papers talk at length about their choices on one of these levels, without doing justice to the other levels. Some papers, when evaluating neurocontrol, confuse the costs and benefits of the learning design with the costs and benefits of particular subsystems; in fact, in some cases, they restrict the analysis to only one type of ANN. One of the worst common mistakes is to confuse the pros and cons of backpropagation—a very general technique for calculating derivatives (12) and adapting networks—with the pros and cons of a particular class of ANN, which is properly called the Multilayer Perceptron (MLP) but often improperly called a "backpropagation network."

Benefits and Capabilities of Learning Control Designs

Because the design process operates at these three levels, it does not make sense to ask what the benefits of neurocontrol are as such. Instead, we can ask what the benefits are for using these learning control designs, in general, at the middle level of the design process. Then, when we need to fill in the sockets, we can ask what the benefits are of using specific types of ANNs instead of other possible subsystems. In many applications, at some stages of development, it makes sense to use a mixture of subsystems, including some ANNs and some other types of subsystem.

The benefits of learning control in general are fairly straightforward. In traditional design approaches, controllers are usually developed based on models before the plant is even built. Then, once a prototype is built, and the control does not actually work as intended, there is a long and laborious period of testing, remodelling, and tweaking. In the aircraft and manufacturing robotics areas, many experts estimate that 80% of the costs of the entire system development effort come from this tweaking stage. If one could replace tweaking by a more automated process of learning, one could reduce these costs substantially, and accelerate the development schedule. In some cases—as with novel high-performance aircraft—the reduction in tweaking could also mean a reduction in the need to crash a few prototypes during the development process; such crashes were once a normal and unavoidable part of the development process, but are becoming less and less acceptable today.

Learning control can be used in two major ways to reduce the need for tweaking after the development of traditional controllers:

1. *Given a flexible enough control structure,* initialized to something like a traditional controller, one can train the controller to optimize performance over a wide range of possible assumptions, in offline learning. For example, one can use a simulation model to generate multiple streams of training data, but with different parameters or couplings assumed in each stream of data. When first proposed in 1990 (13), this general approach was called "learning offline to be adaptive online." Since then, Ford Research has extended the approach in a wide range of applications (e.g., 14,15), and called it *multistream learning*. The success of this approach de-

pends heavily on the flexibility of the control structure and on how well it is initialized. Controllers developed in this way are now the only affordable mechanism which have demonstrated it can achieve ultralow emissions on road-tested cars from a U.S. manufacturer.

2. True real-time learning permits adaptation to the actual real-world plant. For example, in aviation one might use real-time learning much as a human pilot does, by gradually extending the envelope of safe operation from low speeds to higher speeds, while always monitoring how close the system is to the edge of that envelope. In this case, the trick is to move out far enough that one is learning something, but not so far that one is in danger.

Actually, there are many variants of these approaches, and a very complex connection to formal control theory. For example, in automotive control, one could pool actual data from a number of real cars, for use in offline multistream learning. Likewise, there is room for more systematic efforts in deciding how to generate the multistream training data. In one version, one could even train an "adversary neural network" to control the random disturbances and parameter uncertainties, and to try to destabilize the proposed controller (neural or nonneural); this would be a way of implementing the notion of "robust stability as a differential game", which is fundamental in modern control theory (16). One way to evaluate the actual stability of competing controllers and competing engineers may be to offer them each the challenge of destabilizing each other's controllers, subject to various limitations on how much disturbance they are allowed to introduce. Alternatively, the offline learning techniques developed in neurocontrol can be used as an efficient numerical technique for finding the solutions to a nonlinear stochastic optimization problem—which is also an important element of robust control (16). Finally, it should be possible in principle to link these kinds of learning-based designs to actual Computer-Aided Design (CAD) tools and simulations, in order to permit something like "design for controllability" or "design for dynamical performance"; this could someday be extremely useful in reducing the number of generations required for certain kinds of design processes (e.g., for fuel cell automobiles), but little work has been done so far along these lines.

The discussion so far has described the benefits of these learning control designs in general. But there is a wide variety of designs available, intended to perform different tasks, with a variety of different applications and benefits. These designs will be described in more detail later on. For now, in summary, there are three broad classes of designs forming a kind of ladder from simplest to most complex:

1. *Cloning designs.* These designs permit you to emulate the actual behavior of an existing expert controller, such as a human being or an accurate but expensive automatic controller. Unlike the usual expert systems, these designs imitate what the human does rather than what he or she says.
2. *Tracking designs.* These designs permit you to track a desired setpoint (like desired temperature, in a thermostat) or a desired reference trajectory (e.g., desired movement of a robot arm) or a reference model whose function is to output a desired trajectory.
3. Designs to perform *multiperiod optimization,* explicitly or implicitly. The explicit designs tend to be simpler to implement, and more exact, but computationally more expensive and less capable of coping with random disturbances. The implicit designs are sometimes called *reinforcement learning,* and have strong connections to what we see in the brain (5). In some applications, it is best to use a hybrid, where the explicit methods provide a short-period look-ahead and the implicit methods account for payoffs or results beyond that period (17).

Sometimes it is very difficult to find the best controller simply by using an optimization method, starting from neural networks initialized with random weights. It is often best to take a step-by-step learning approach. In this approach, one first trains a controller to solve a relatively simple task. The final version of that controller, after training, is then used as the initial version of a controller trained to perform a more difficult task. In any kind of learning system, the initial structure of the controller—the starting point—plays an important role in deciding what can be learned.

For example, Accurate Automation Corporation (AAC) (18) visited Wright Patterson Air Force Base a few years ago, to propose the use of optimizing neurocontrol to help solve some of the pervasive weight problems expected with the National Aerospace Plane, NASP. (NASP was intended to be a prototype of an airplane which could reach earth orbit, as an airplane, at airplanelike costs.) But they were told that it took millions of dollars even to develop a controller able to stabilize the craft—let alone optimize it—following their nonlinear model of the vehicle.

AAC then created a video game to run on Silicon Graphics, to simulate the NASP model, but at a speed slow enough that a human being would have some hope of stabilizing it. Many humans played the game, but only a few were able to stabilize and land the craft consistently. AAC recorded the behavior of those successful humans in the game, and simply developed a neural network clone of their behavior patterns. This neural net could then stabilize the aircraft, in all their tests, and—unlike the humans—could run at electronic speed. The resulting network could then be used, in principle, as the initial value of the controller for a neural optimization scheme.

Based in part on this very rapid success, AAC—a small neural network company—became a prime contractor on the NASP program, and then went on to play the lead role in the follow-ons to NASP, the LoFlyte program and the HyperX, where neurocontrol is planned to play a central role.

Of course, step by step learning is not the only way to define the starting point for a learning controller. For the first task to be learned, one may start out by using neural networks with random weights, or weights selected on an intuitive basis. Or one may define the total control system to equal a previously designed traditional controller plus a simple neural network. Or one may use a neural network clone of a pre-existing traditional controller. Or one may use a set of fuzzy IF-THEN rules encoded into an elastic fuzzy logic module (11). [Fuzzy IF-THEN rules (11) are rules like, "If the engine is very hot and the pressure is rising, turn down the fuel intake fairly quickly."] The choice really depends on what kind of information is readily available, and on the requirements of the particular application. One can never guarantee that a nonlinear learning system of significant size will find the globally optimal strategy of action; however, one can generally expect it to improve upon the best of what is tried when initializing the system.

The difficulty of finding the global optimum, and the value of careful initialization, vary greatly from application to application. Unfortunately, the conventional wisdoms about these issues often reflect past traditions and habits rather than the real needs of particular applications.

Learning Versus Feedback Versus Adaptation

The previous section discussed the benefits and capabilities of learning control in general. The section after next will discuss the benefits of neural networks versus other subsystems, within the framework of learning control. But first, this section will begin to round out the discussion of learning control in general, by discussing the relation between learning, feedback, and adaptation, which tends to be confusing even to researchers in the field.

In control, in general, there is a ladder of five major categories of design, in order:

1. *Static controllers.* For example, the valve controller on an ordinary gas stove is a static controller. After you turn the dial, the system simply injects gas at a fixed rate. Some people put timers on top of static controllers (as in some gas ovens), but the basic principle remains the same: the control action is specified completely in advance, without any use of sensor input other than a clock.
2. *Feedforward controllers.* In a chemical plant, the controller for one valve may actually respond to the flow of other gasses or liquids coming into the reactor. In a feedforward controller, the control action at any time, $\boldsymbol{u}(t)$, may depend on some sensor inputs—but not on inputs which measure how well the controller is performing. Static and feedforward controllers, together, are often referred to as "open-loop control."
3. *Fixed feedback controllers.* Feedback control, as a practical tool in engineering, dates back at least to James Watt's flywheel controller, which was crucial to the successful operation of the steam engine and which in turn was crucial to the Industrial Revolution. The modern, more mathematical view of feedback emerged much later, particularly in the seminal work of Norbert Wiener (2). To explain this concept, Wiener discussed several simple examples such as the everyday thermostat. In the thermostat, there is a feedback from a thermometer to the controller which turns the furnace on and off. When the temperature is too high, the furnace is turned off. In other words, there is a sensor which measures the actual value of the variable (temperature) which we are trying to control. The control action is specified as a function of that sensory reading. In fixed feedback control, the controller has no memory; its behavior is fully specified in advance as a function of all the sensory inputs (and perhaps of its own past actions) at specified times. For example, the function may depend on sensor readings at the present time, t, and on readings at times $t - 1, \ldots, t - k$, for some k.
4. *Adaptive controllers.* Adaptation changes the behavior of a controller, so as to account for changing conditions in the plant being controlled. For example, a good human driver knows how to adapt when roads become slippery due to rain. Even if the driver cannot see how slippery the road is in different places, he can sense how the behavior of his car is changing, and adapt accordingly. Later, when the rain dries out or he enters a dry patch, he has no trouble in returning to his old pattern of driving. In engineering, there are many situations which require adaptation, such as changes in the mass and location of a load to be carried (19), changes in the friction of a moving part, changes in the atmosphere, wear and tear, and so on. In formal terms, adaptation tries to adjust the control rule so as to account for variations in the plant which cannot be observed directly, and which typically (but not always) vary slowly with time.
5. *Learning controllers.* Learning control tries to build systems which, like the brain, accumulate knowledge over time about the dynamics of the plant to be controlled—or, more generally, about the environment which the controller lives in, and about strategies for coping with these dynamics. For example, an inexperienced driver may not know how to change his driving behavior during a rainstorm. This causes many crashes. Over time, a driver may learn how to sense and respond to such changing road conditions. He or she learns to become adaptive. Notice that drivers can respond much faster and much better to conditions which they have learned to adapt to than they do to conditions which they are learning about for the first time.

Adaptation refers to the driver's ability to respond to *current conditions*. Learning refers to the longer-term, cumulative process of building up a skill. In formal terms, learning tries to adjust the control system so as to account for parameters or structure in the plant which are initially unknown but are not expected to change; this may include learning the dynamics or probability distributions for changes in mass, friction, etc.—thus, learning how to adapt to changes in these specific parameters.

These distinctions are of great importance, but they can become very fuzzy at times. For example, the distinction between feedforward control and feedback control depends on our making a distinction between "goal variable" sensors and other sensors. This may not always be clear. Likewise, one may use a mathematical design derived from a learning formulation, in order to build a controller intended for use in adaptation. In fact, that approach has been central to the modern field of adaptive control (9,10). Note that the four types of learning control discussed previously (cloning, tracking, explicit optimization, implicit optimization) are all subcategories of the learning control category here.

The term *feedback control* is normally used in a very broad sense, including fixed feedback control, adaptive control, and learning control.

The term *classical control* is used in very different ways, by different researchers. Most often it refers to classical designs based on Laplace transforms for dealing with single-input single-output (SISO) linear controllers such as Proportional Integro-Differential (PID) controllers. Modern control typically refers to a collection of more recent approaches, most of which involve the sophisticated design of Multiple-Input Multiple-Output (MIMO) fixed feedback controllers; however, traditional adaptive control (9,10) is usually included as well.

The term *robust control* has also been used in two different ways. In a broad sense, robust control refers to the development of control designs which are expected to remain stable, even if the parameters or states of the plant may be different from what one expects initially. In a narrow sense, robust control refers to specific techniques which have been developed to design fixed feedback controllers which remain stable over a wide range of possible values for the parameters of the plant. Some engineers prefer systems which are robust in the narrow sense, because it can be difficult to analyze the stability of systems with adaptive characteristics. However, there are many cases where it is impossible to find a fixed controller powerful enough to stabilize a plant over the entire normal operating range (19).

The multistream approach described in the previous section fits within the broad definition of robust control, but does not fit within the narrow definition. It does not fit within the narrow definition because it requires the use of a control structure general enough to permit adaptive behavior. In other words, the controller must have some kind of internal *memory* which implicitly keeps track of the road friction or mass or other time-varying parameters of the plant. Much of the best research into formal robust control also fails to meet the narrow definition, because it includes the use of *observers* or *state estimators* which contain this kind of memory (16). In other words, they are not fixed feedback controllers as defined previously. Narendra and Annaswamy have argued (9) that traditional adaptive control may be thought of as a form of robust control in the broader definition.

Stability, Performance, Chaos and Verification

In choosing between competing control designs, the engineer must usually trade off three different criteria: (1) the actual degree of stability expected from the system; (2) the actual degree of performance; (3) the degree of formal confirmation available, to confirm that the system will always have the expected degree of stability and performance, across different possible conditions. In tracking problems, the issue of performance is often discussed in terms of steady state accuracy, transient response, and disturbance rejection (20). Stability is often discussed in terms of margins for error and the allowable range of variation for the (unknown) parameters of the plant to be controlled.

The distinction between *actual stability* and *stability proofs* is especially important in many practical applications. As we climb up the ladder of control designs, from static controllers up to nonlinear learning controllers, the behavior of the controller becomes more and more complex. This makes it more and more difficult to prove theorems about stability. However, *if* learning control is used appropriately, the more complex designs make it possible to achieve greater stability in a more robust way, over a wider range of possible conditions.

In a 1990 workshop (3), Narendra expressed this point in a very graphic way. He described some simulations of an ordinary sort of nonlinear tracking problem, which could be addressed by use of adaptive control. First, he linearized the plant in the usual fashion, and implemented the usual linear adaptive control designs for which he and others had proven many, many stability theorems (9). Despite the theorems, the controller blew up regularly in simulation. Then, he used a neural network tracking design, which essentially just replaced a matrix with an ANN in an ordinary adaptive control design. For that design, he could prove no theorems at that time, but the design remained stable across a wide range of simulations.

Since then, Narendra and others have in fact generated dozens of theorems for various forms of neural adaptive control or tracking control. But the lessons from this example still remain valid. The first lesson is that many stability proofs make strong, simplifying assumptions about the nature of the plant or of the environment. We can make stronger proofs by assuming that the controller and the plant are both made up of independent linear systems, but if the plant is not actually linear, then the proofs become little more than empty window-dressing. (There are other critical assumptions as well in these theorems.) The second lesson is that we actually can develop proofs for more general controllers in time, but it may take time. Prior to the development of formal stability proofs, we must often start out by understanding the sources of instability in a more practical sense, and developing those more stable designs which we later prove theorems about. The third lesson is that the officials responsible for verification and validation in different sectors may simply have no absolute, valid mathematical guarantee available to them for any kind of controller in the real world.

The practical development of verification and validation techniques is a complex art, for which the needs vary greatly from application to application. For example, many people claim that the control of manned aircraft must be one of the strictest areas of application, where firm mathematical proofs of stability are always essential, because of the safety issues with many human lives and expensive vehicles at risk. However, with conventional aircraft, when there is some sort of large unexpected damage—such as a wing shot off, or a hydraulics failure—then the assumptions behind the proofs fly out the window. For severe faults, the current control designs have almost a 100% probability of failure, which is to say a fatal crash.

Back in 1992, White and Sufge (3), working with Urnes of McDonnell Douglas, developed a model-free reinforcement learning scheme which, in simulation, could relearn the control of an F-15 in two seconds in half of the cases of severe damage. The McDonnell Douglas simulation model of the F-15 was a relatively realistic model, central to the development of that vehicle. Thus, simulations suggested that the rate of crashes could be cut in half by using a real-time learning scheme in this application. One hundred percent success was absolutely not required, because it was impossible; simply to reduce the crashes from 100% to 50% would be a great accomplishment. This has large implications both for aviation safety and for the balance of power in aerial warfare.

Based on the success of this early simulation work, the NASA Ames Research Center awarded a large contract to McDonnell Douglas to translate this work into a working system. An entire cottage industry of reconfigurable flight control has sprung up, with a link to the emerging world of thrust vectoring (control by changing where the engines point, rather than moving flaps on the wings and such). There are many parallel efforts going on, each with its own ladder of designs intended to reduce the crash rate further and further. It is argued (21) that a multistream training approach using implicit multiperiod optimization methods could be very useful in this application.

Charles Jorgensen of NASA Ames has reported that the first stage of neural-based reconfigurable flight control has been totally successful. More precisely, it has been used to land a full, manned MD-11 jumbo jet with all flight surfaces locked up, to simulate a total loss by hydraulics. (See http://ccf.asrc.nasa.gov/dx/basket/storiesetc/96_39.html.) The verification and validation required for this experiment, involving collaboration between NASA Ames and NASA Dryden, probably contains many lessons of general interest. (See www.nasa.gov for a discussion of the relation between these major components of NASA.) In general, the development of practical techniques for verification and validation is similar in spirit to the development of control software; it may be slow and laborious, but as time goes on, it permits a gradual rise in the level of intelligence which can be used in our control systems.

Strictly speaking, the difference between seeking stability and seeking high performance is not so great as one might imagine. For example, in multiperiod optimization, one can simply construct a utility function (or cost function) which penalizes the system whenever it enters certain forbidden zones. By minimizing the expected value of such a cost function, one minimizes the probability of entering these zones. One maximizes stability. Usually, when the random disturbances are Gaussian (which permits very large disturbances on rare occasions), the probability of entering the danger zone can never be reduced to zero. In that case, stochastic optimization may indeed be the safest choice available, even though it does not permit zero risk. For a truly realistic and complete description of the control problem, one cannot really expect risk to equal zero, no matter what the control strategy.

In practice, users of optimization methods usually do not define a utility function based solely on stability (i.e., minimizing risk). By adding terms to represent energy use, pollution, jerkiness of motion, actuator constraints, and so on, one can develop a controller based on a reasonable tradeoff between various aspects of performance and stability, weighted according to the needs of the specific application. Some users explore a variety of utility functions in order to get a feeling for what the choices and tradeoffs are.

The optimization-based approach to stability may also open the door to a new approach called *chaos control* (22). Leaders in the chaos field have argued that traditional control too often focuses on trying to stabilize systems at a fixed point, even when this is both expensive and unnecessary. By designing highly sensitive plants which can even become chaotic, and by accepting low-energy controls which only try to keep the plant within an acceptable region, we may be able to save energy and increase performance. One might even argue that the SR-71 aircraft already provides an example of this kind of tradeoff. The main difficulty with this idea of chaos control lies in actually designing plants and controllers which embody the idea. This difficulty could be overcome simply by using learning control systems based on multiperiod optimization (presumably implicit optimization) with utility functions that keep the plant within an acceptable region of operation. In fact, the use of such methods during the design process would make it possible to tune the physical design parameters, together with the control parameters, so as to maximize some kind of combination of stability and performance together.

Once we accept that real-world plants are in fact highly nonlinear, the most rigorous, formal approach to achieving stability is fully nonlinear robust control. (There also exists a far less general nonlinear theory, based on feedback linearization.) The key results of this theory, mentioned previously, are that the development of a robust control system is equivalent to the solution of a differential game or of a stochastic optimization problem (16). Thus, for example, Professor Michael Athans of MIT, a major leader of the mathematically rigorous group in aerospace control, has argued that there is a critical need to develop general computer software to solve the Hamilton-Jacobi-Bellman (HJB) equation for larger-scale, nonlinear systems in order to implement this approach. The HJB equation is the foundation of multiperiod optimization (23), to be discussed in greater detail toward the end of this article.

Unfortunately, for all but the smallest systems, it is basically impossible to solve the HJB equation exactly (except in a few very special cases, such as the linear case). Numerical methods or approximation methods are required. All solutions of the HJB equation in such cases are in fact approximations, regardless of whether they are called numerical solutions or approximate solutions. At present, the safest procedure is to use the most accurate available approximation methods, which include some of the implicit learning control methods discussed here. In the long term, it will be crucial to develop more formal tools to analyze the numerical approximation errors and their implications for stability.

As this article goes to press, Richard Sacks of AAC and Daniel Prokhorov (24) of Ford Research Laboratories have each reported stability results for the MBAC designs to be discussed later. Johan Suykens of the Catholic University of Leuven (Belgium) has discussed the application of existing stability theorems for nonlinear Model-Predictive Control to the case of neurocontrol (25,26). See Ref. (27) for some additional examples of practical applications of neurocontrol.

Benefits and Costs of Using Alternative Neural Networks

Once we have decided to use a learning control design, when should we use neural networks to provide the required subsystems? What kinds of artificial neural networks (ANNs) should we use? Should we implement learning in all of the subsystems?

In many applications, the best strategy is to use a mix of ANNs and other structures, at different stages of development. When the first stage of controller development is based on some kind of physical plant model, for example, it often makes sense to use that model directly, instead of taking the time to train a neural network to approximate that model. On the other hand, some learning control designs do not even require a model of the plant. Others require the development of dual subroutines (12,3) which require some effort to program and debug (28). In the future, when software becomes available to generate these dual subroutines automatically, starting from user-supplied models expressed in some standard format, it will become much easier to use physical models directly.

Aside from neural networks and first-principles models, there are a host of methods used in the past in engineering to approximate nonlinear functions—gain-scheduling schemes, Taylor series, fuzzy logic, interpolation tables, and so on. Yet almost every useful general-purpose approximation scheme has been repackaged by someone as a kind of neural network, and trained by use of neural network methods! Corresponding to Taylor series are higher order neural networks or Ivanenko designs, among others. Elastic fuzzy logic (8) has been formulated as a kind of ANN, with interesting potential properties. Many local neural networks serve, in effect, as glorified lookup tables, with varying degrees of interpolation or soft switching. (For example, RBF and CMAC will be discussed in the next few paragraphs.) Various kinds of mixture of experts networks (29,30) provide something like gain scheduling, except that the soft switching is trained to give optimal results, and nonlinear relations can be estimated within each region. (Such designs can be especially useful when you are worried about the system forgetting what it learns in rare but important types of conditions.) Because of all these parallels, the decision to use neural networks is really just a decision to perform the learning function in a systematic way; all the same structures used in the past are still available, in effect, as ANNs.

In summary, the practical choice is between using specialized nonlinear structures, based on prior knowledge, such as a physical plant model, versus the use of some kind of neural network to provide a general-purpose ability to learn any nonlinear function. For maximum accuracy, one would want to combine prior knowledge and learning-based knowledge (5); however, this is not always worth the effort involved, and the best approach to combining the two sets of information will depend on the particular application. It will depend especially upon the accuracy and completeness of the prior information, and on the availability of training data. Even after you decide to use a neural network, the choice of which neural network to use can often make or break your application.

In the past (31), four general advantages have been cited for neural networks here: universal approximation ability, ease of use, availability of chips and PC boards, and links to the brain. (See also the article on NEURAL-NETWORK ARCHITECTURES.)

Almost every major variety of ANN used in engineering has some variety of universal approximation theorem attached to it, proving that it can approximate a smooth function arbitrarily well, if given enough neurons. Andrew Barron (32) has gone further, by proving that the most popular form of ANN—the multilayer perceptron (MLP)—can approximate smooth functions *of many arguments* with less additional complexity (i.e., fewer parameters) than are required for "linear basis function approximators." Linear basis function approximators include most of the common alternatives, such as the usual local networks and Taylor series. Sontag has pointed out that there are a few classical approximators—such as rational functions (ratios of polynomials)—which can do as well, in theory; however, MLPs tend to be far more manageable than rational functions in ordinary engineering applications involving multiple inputs. There is another class of ANN—the Simultaneous Recurrent Network with an MLP core (33,34)—which can also approximate certain types of nonsmooth function which the MLP cannot handle.

These results from Barron and Sontag confirm the existence of a very unpleasant tradeoff, which has long been appreciated in intuitive terms by practical neurocontrol engineers. There is one class of ANN design—the MLP and its extensions—which can approximate functions in a parsimonious way, and therefore do a better job of generalizing or extrapolating from a limited amount of data. There is a different class of ANN designs—local designs like the Radial Basis Function (RBF) and the CMAC (see CEREBELLAR MODEL ARITHMETIC COMPUTERS)—which permit very rapid real-time learning and easier mathematical analysis. The present generation of off-the-shelf ANNs do not provide the *combination* of good generalization ability and real-time learning that the neurons of the brain provide! In the long term, advanced research should make it possible to achieve more brain-like capabilities (35), and there are some practical tricks available (15,30,36,3). For the time being, however, this tradeoff between learning speed and generalization ability tends to favor a greater use of offline learning than we would want in the long term. In many practical applications, it is currently best to start off with an approach based on offline learning, and then add elements of real-time learning in a cautious, step-by-step manner. Ease of use has probably been the most dominant factor in the widespread use of ANNs in learning control. After all, if an ANN from off the shelf can approximate a nonlinear function to any desired degree of accuracy anyway, then why bother with all the complexity of representing the function in other ways?

Availability of special purpose chips and PC boards has also been a major factor. It was crucial, for example, to the Ford (15) and AAC applications mentioned previously. In ordinary computing or supercomputing, one can normally fit only a small number of independent processors on a chip (usually just one). This is because one must accommodate a large instruction set, digital logic, and so on. However, for distributed ANN chips, it is good enough to perform the same arithmetic operation over and over again in each processor. Neural chips now on the market already contain up to thousands of processors per chip. Computers based on such chips have demonstrated hundreds of times more throughput per dollar for what they do than conventional computers.

There are many applications where traditional controllers have enough accuracy to do the job, but are too large or expensive to implement. For example, one cannot afford to put a large Cray into every airplane, car, or manufacturing cell. In such cases, neural network clones of the traditional controller can be very useful. Many neural chip manufacturers also supply software subroutines to simulate their chips, so that neurocontrol experts can develop realistic designs which are easily migrated into hardware. Naturally, the availability and throughput of chips is greater for some types of ANNs than for others.

Finally, the link to the brain itself has also been a major motivation behind the development and use of neural network designs. Because the brain itself is a neurocontroller (5), it provides both an existence proof for the ultimate potential power of neurocontrol and a source of clues for how to achieve that power.

Model-Based Designs Versus Model-Free Designs

A number of engineers have reported that neurocontrol has worked better than classical control in their applications because it does not depend on the quality of models available for the physical plant to be controlled.

For example, White and Sofge reported great success in applying reinforcement learning to the continuous, low-cost manufacturing of high quality carbon composite parts (3). Other approaches to this problem had already been studied extensively by McDonnell Douglas, because of the large economic implications. A key reason why the earlier approaches did not work was that the manufacturing process was so complex that the first-principles models available were not very accurate. Unfortunately, after this technology had already proven itself out in actual production, its commercialization was stalled by contracting issues unrelated to the technology as such.

The Air Force recently held an in-depth workshop, inviting the lead engineers involved in controlling high-powered optical instruments and associated space structures (37). Those speakers reported, on the whole, that the use of modern control theory had produced little if any improvement over classical control in these applications, and had also been extremely expensive. The problem, they claimed, was the need for very detailed, accurate plant models. On the other hand, neural network tracking approaches—particularly the work of David Hyland, one of the experienced people in these applications—led to significant improvements, at relatively low cost.

These benefits have been quite real, but one must be careful to understand what they really tell us. In actuality, the real choice is not between model-based designs and model-free designs. The practical choice is between *five* different alternatives, all quite common in neurocontrol:

1. Truly model-free learning designs, which include cloning designs, direct inverse tracking designs, and smaller-scale reinforcement learning designs;
2. Implicitly model-based designs, such as the DRAL architecture discussed in the article on NEURAL NETWORKS FOR FEEDBACK CONTROL in this encyclopedia;
3. Designs which require us to train an ANN or some other learning-based system to predict or emulate the plant;
4. Designs which use expert first-principles models of the usual sort;
5. Designs which depend on *multistream models*—stochastic descriptions of the plant which include an expression of uncertainty about plant parameters, coupling, and possible defects, in addition to random disturbances.

The White and Sofge example was based on alternative number one—the use of a reinforcement learning system which pushed the envelope on how large a task can be handled in a truly model-free design. The Hyland system was based on alternative number three—the use of a neural model, which in turn depended critically on advanced prior work developing ways to train neural models (38). [See (3, Ch. 10) for related theoretical work.]

The DRAL system, and some of the work by Berenji (39), has exploited the assumption that there is a single action variable $u(t)$, whose impact on the plant always has the same sign. It is mathematically equivalent to the use of a model-based design in which the model is simply $x(t) = ku(t)$, for some positive constant k. The stability and success of these systems helps show how some of the neural model-based designs can in fact be very robust with respect to the precise details of the model.

In practical system development work, it is often critical to develop the best possible initial controller based on prior information, before the physical plant has actually been built. Even if real-time learning will be used, this initial controller provides a starting point for further learning. By definition, this initial controller must be based on some kind of model, even if it is only a simulation model used to generate training data! At this stage of development, true-model independence is impossible; the best one can do is to reduce the degree of dependence by using a multistream model instead of a conventional fixed, deterministic model. However, as explained previously, success in training a controller to perform well on such multistream data requires the use of a controller capable of *memory* or of adaptive behavior. Neural networks embodying "time-lagged recurrence" (3,12,34) provide that capability.

It has been argued that the brain itself relies heavily on large-scale reinforcement learning designs which require the use of neural models (40). Perhaps it may use hybrid designs, which make the results relatively robust with respect to errors in those models; however, without exploiting some knowledge about cause-and-effect relationships, and without an ability to form expectations about the results of actions, the brain could never handle the complexity of the decisions that it must make in everyday life.

NEUROCONTROL: DETAILED OVERVIEW OF THE DESIGNS

This section will provide additional technical detail for the four broad classes of neurocontrol discussed previously—cloning, tracking, explicit multiperiod optimization, and implicit multiperiod optimization. First, however, it will define some notation and describe some common ANN subsystems which can be used when building up a larger control system.

Notation

This section will assume that the controller sees a vector $\boldsymbol{X}(\mathrm{t})$ of m observables $(X_1(t), \ldots, X_m(t))$ at each time t, and that it will then output a vector $\boldsymbol{u}(t)$ of control actions. In effect, $\boldsymbol{X}(t)$ represents the input from the sensors, and $\boldsymbol{u}(t)$ the output to the actuators. Frequently there will be an additional vector $\boldsymbol{r}(t)$ which represents the estimated state of the plant. There may be a reinforcement signal, $U(t)$, or a utility function, $U(\boldsymbol{X})$ or $U(\boldsymbol{r})$, which the control system tries to maximize over time. This notation is slightly different from the traditional notation of control theory, but it has a number of practical advantages related to the use of neural networks and the links to other related disciplines. In mnemonic terms, the X relates to eXternal data, the r to Representation of Reality (usually through Recurrent neurons), and U represents utility. Strictly speaking, the estimated state vector $\boldsymbol{r}(t)$ is often composed of the combination of $\boldsymbol{X}(t)$ and $\boldsymbol{R}(t)$, where $\boldsymbol{R}(t)$ represents the output of some (time-lagged) recurrent neurons in one of the subsystems of the controller. In some designs it is assumed that the plant to be controlled is completely observable, in which case $\boldsymbol{X}(t)$ and $\boldsymbol{r}(t)$ will be the same.

Common Subsystems

Most ANN designs used in engineering can be built up in tinker-toy fashion by linking together *static neural networks*. A static neural network receives a vector of inputs $\boldsymbol{X}$ and generates a vector of outputs $\boldsymbol{Y}$. It contains an array of weights or parameters W. Learning usually involves the adjustment of the weights, W, although it often involves some changes in the connections in the network as well. The operation of a static neural network can always be represented as:

$$\boldsymbol{Y} = \boldsymbol{f}(\boldsymbol{X}, W)$$

where $\boldsymbol{f}$ is some function. To be precise, $\boldsymbol{f}$ is sometimes called a vector-valued function of a vector, or simply a mapping.

When we use designs that let us use any parametrized static mapping, then of course we are not limited to neural networks as such. When we actually use a static neural network, we may choose to insert inputs from various different sources; therefore, the inputs and outputs will usually be labeled as something else besides $\boldsymbol{X}$ and $\boldsymbol{Y}$.

The construction of larger systems by linking together static neural networks is not just a useful mathematical fiction. It is also a useful approach to building up models and flexible software to implement learning control. This approach makes it easier to switch neural and nonneural components in and out of a general learning design.

Learning control designs are usually not built up directly from static neural networks. They are built up from larger subsystems which in turn may be made up of static neural networks or other parametrized static mappings. The three most common types of subsystems today are: (1) supervised learning systems (SLS); (2) systems trained on the basis of gradient feedback; and (3) system identification subsystems.

Supervised learning systems (SLS) try to learn the functional relationship between one observed vector $\boldsymbol{X}(t)$ and another $\boldsymbol{Y}(t)$, based on seeing examples of $\boldsymbol{X}(t)$ and $\boldsymbol{Y}(t)$.

For real-time learning, we usually assume that the SLS starts out with an initial set of weights W at each time t. Then, after it observes $\boldsymbol{X}(t)$, it makes a prediction for $\boldsymbol{Y}(t)$. Then, after observing the actual value of $\boldsymbol{Y}(t)$, it goes back and adjusts the weights W. In advanced research, this common procedure is sometimes called weight-based real-time learning. There are alternative approaches to real-time learning, still at the research stage, called memory-based learning or syncretism (35, Ch. 13).

For *offline* learning, we often assume that there is a database or training set of examples, which may be labeled as $\boldsymbol{X}(t)$ and $\boldsymbol{Y}(t)$ for $t = 1$ to T. We often use the real-time learning approach, cycling through the observations one by one, in multiple passes through the entire database. (These passes are often called epochs.) Many SLS designs also provide an option for batch learning, where the weights are adjusted only after some kind of analysis of entire training set. In fact, most model estimation methods taken from the field of statistics may be thought of as batch learning designs.

The most common forms of SLS are based on some sort of error feedback, which may be written:

$$\boldsymbol{Y}(t) = \boldsymbol{f}(\boldsymbol{X}(t), W) \tag{1}$$

$$E(t) = E(\boldsymbol{Y})(t), \boldsymbol{Y}(t) \tag{2}$$

$$F\_\hat{\boldsymbol{Y}} = \nabla_{\boldsymbol{Y}} E(\boldsymbol{Y})(t), \boldsymbol{Y}(t)) \tag{3}$$

$$F\_\hat{Y}_i(t) = \frac{\partial}{\partial \hat{Y}_i} E(\boldsymbol{Y}(t), \boldsymbol{Y}(t)) \tag{4}$$

where E is some kind of error function. (See the articles on ARTIFICIAL INTELLIGENCE, GENERALIZATION and FEEDFORWARD NEURAL NETS.) Equation (1) simply states that the outputs of the neural network will be used as a prediction of $\boldsymbol{Y}(t)$. Equation (2) states that we calculate error as some function of the actual value of $\boldsymbol{Y}(t)$ and of the predictions. To measure error, most people simply use square error—that is, the squared length of the difference between the two vectors; however, there are some applications (especially in pattern classification) where other error measures can work better. Finally, Eqs. (3) and (4) are two equivalent ways of expressing the same idea, using different notation. In both cases, we use the derivatives (i.e., gradient vector) of error as a feedback signal, which will then be used in training the ANN.

After we know the derivatives of error with respect to the outputs of the ANN, we can then go on to compute the derivatives of error with respect to the weights,and then adjust the weights accordingly. The backpropagation algorithm, in its original form from 1974 (12), permits us to calculate all the derivatives of error with respect to the weights at low cost, for virtually any nonlinear differentiable structure, not just ANNs! Equations 1 through 4 are used most often with Multilayer Perceptrons. (See Chapter 8 of Ref. 12 for the most general form of MLP.) However, many other ANN learning procedures can be expressed in this form as well.

In supervised learning, the vector $\boldsymbol{Y}(t)$ is sometimes called the vector of desired outputs or desired responses or targets. Because this vector is known to us, we can use a variety of nearest-neighbor prediction methods or associative memory designs, instead of derivative-based learning. However, this only applies to subsystems which perform supervised learning. Sometimes, as part of a control design, we need to adapt a static neural network $\boldsymbol{f}(\boldsymbol{X}(t), W)$ without access to a vector of targets $\boldsymbol{Y}(t)$. Typically, the larger design tells us how to calculate the vector $F\_\hat{\boldsymbol{Y}}(t)$, based on information elsewhere. Subsystems of this sort must be trained on the basis of derivative feedback, which in turn requires some use of backpropagation.

In other words, for true supervised learning tasks, we have a choice between derivative-based learning methods and other sorts of methods. For certain other learning tasks, derivative-based learning is the only possible alternative.

Finally, in control applications, we often need to use subsystems which learn to predict the plant to be modeled. Conceptually, we might describe these systems as:

$$\hat{\boldsymbol{Y}}(t) = \boldsymbol{f}(W, \boldsymbol{X}(t), \boldsymbol{X}(t-1), \boldsymbol{Y}(t-1), \ldots, \boldsymbol{X}(t-k), \boldsymbol{Y}(t-k), \ldots) \tag{5}$$

where t represents physical time in the plant, assuming some kind of fixed sampling rate for the sensors and actuators in the plant. Systems of this general sort are called neuroidentification systems. There is a ladder of designs available for neuroidentification, similar to the ladder of designs in control.

In the simplest neuroidentification designs, there is no actual use of inputs before some fixed time interval k. The prediction problem is actually treated as a supervised learning problem, with an expanded list of inputs. Networks of this sort are called Time Delay Neural Networks (TDNN). They are similar to Finite Impulse Response (FIR) systems in signal processing, and to nonlinear autoregressive (NAR(k), or, more precisely, NARX(k)) models in statistics (12,41). Unfortunately, these models are often called NARMA models in the literature of adaptive control. This usage has become so widespread in some areas that some people even consider it a convention rather than an error; however, the original concept of ARMA modeling is so important and fundamental in statistics (12,41) that the original usage should be preferred, even in control theory. In statistics, ARMA refers to mixed Autoregressive Moving-Average processes—stochastic systems which contain patterns in the disturbance terms which AR models cannot represent in a parsimonious way; such patterns result whenever there is "observation error," (i.e., error in sensing or measuring the state of the plant to be controlled).

More powerful designs for neuroidentification result from adding one or both of two additional features: (1) time-lagged recurrence; (2) dynamic robust training.

Time-lagged recurrent networks (TLRNs) essentially contain a kind of internal memory or short-term memory, as required for *adaptive behavior*. They provide a generalization of true ARMA modeling capability, which is also similar in spirit to Infinite Impulse Response (IIR) systems in signal processing, and to Extended Kalman Filtering (EKF) in conventional control. James Lo (42) has argued that TLRNs perform better than EKF in these applications. [The reader should be warned, however, that there is another completely different application of EKF methods in neurocontrol, involving the acceleration of learning rates. For example, the Ford group has used several generations of such acceleration methods (14,15).] TLRNs are harder to train than TDNNs; however, with an effective use of the Adaptive Learning Rate algorithm (3, Ch. 3) and appropriate initial values, they can sometimes learn more quickly than TDNNs.

As an example, the key successes of Ford Research in neurocontrol depend very heavily on the use of TLRNs (14,15), trained by the use of backpropagation through time (BTT). Likewise, the recent success of Jose Principe in speech recognition has relied heavily on the various forms of TLRN he has used, also trained using BTT. BTT was first implemented in 1974, on a classical multivariate ARMA estimation problem (12); see (12, Ch. 8) for a more modern tutorial, emphasizing the use of TLRNs. BTT is not strictly speaking a real-time learning method, because it requires calculations which operate backwards through time; however, it can be used in practice in a real-time mode, in engineering applications which make use of fast electronic hardware (14,15). Unfortunately, the most popular true real-time methods for adapting TLRNs have severe disadvantages. See (34) for a survey of these alternatives, including the new Error Critic design which, in my view, is the only alternative which is plausible as a model of what goes on in the brain.

Dynamic robust estimation can be applied both to TDNNs and to TLRNs, in order to improve the quality of the resulting predictions. The key idea is to minimize errors in multiperiod prediction directly. This idea has been used in various forms for a long time (12), but there is a substantial need for more research to understand the deep theoretical principles involved, and to develop designs which better reflect that understanding (3, Ch. 10). The reader should be warned that parallel identification as used in adaptive control is only the first step up this very high ladder (3), and often performs worse than simple conventional training.

Finally, in some applications, such as stock market trading, a simple predictive model of the plant or environment may not be good enough. In some applications, it is desirable to climb one step further up the ladder, to train true generalized stochastic models of the plant or environment. Among the relevant tools are the Stochastic Encoder/Decoder/Predictor (SEDP) (3, Ch. 13) and, for smaller-scale problems, the Self-Organizing Map (SOM) (43). Here, instead of trying to output the most likely prediction for $\boldsymbol{Y}(t)$, we try to build a kind of simulation model for $\boldsymbol{Y}(t)$. We try to train a network which outputs possible values for $\boldsymbol{Y}(t)$, in a stochastic way, such that probability of outputting any particular value for $\boldsymbol{Y}(t)$ matches the true probability of that value coming from the actual plant. (More precisely, it should match the conditional probability of that value, given the information from times $t - 1$, etc.) These designs have led to a few successful implementations related to control, but there is a need for considerably more research in this area. For example, no one has yet tried to prove universal stochastic process approximation theorems here that are analogous to the theorem which Barron and Suntag have proven for the deterministic case.

Cloning

The very first neurocontroller ever implemented was a cloning controller developed by Widrow and Smith (44).

At that time, no one used the words neurocontrol or cloning in this sense. Even in the 1980s, many researchers thought of ANNs simply as supervised learning systems, without allowing for other types of ANN design. In order to develop a neurocontroller, they would follow two steps: (1) build up a database of training examples of sensor inputs $\boldsymbol{X}(t)$ and correct control actions $\boldsymbol{u}(t)$; (2) use supervised learning to learn the mapping from $\boldsymbol{X}(t)$ to $\boldsymbol{u}(t)$.

At first glance, this kind of exercise seems purely circular. If we already know what the correct control actions are, for a wide variety of possible situations $\boldsymbol{X}(t)$, then why bother to train a neural net? Why not simply use the pre-existing controller or algorithm which tells us what the correct control actions are? The answer is that the pre-existing controller may actually be a special human being, or a very expensive computer program, which may be too scarce, too expensive, or too slow to use in all the applications of interest. Therefore, this approach can be quite useful at times as a way of cloning the behavior of that pre-existing controller.

It is very unfortunate that many early papers using this approach did not adequately explain where their database of correct control actions came from.

Even within the area of cloning, we again face a ladder of designs. In all cases, we begin by recording examples of $\boldsymbol{X}(t)$ and $\boldsymbol{u}(t)$ from a human expert of pre-existing controller. In the simplest designs, we use supervised learning to learn the mapping from $\boldsymbol{X}(t)$ to $\boldsymbol{u}(t)$. In more sophisticated designs, we use neuroidentification methods to predict the desired $\boldsymbol{u}(t)$ as a function of $\boldsymbol{X}(t)$ and of earlier information. Whenever the human expert or pre-existing controller need to have some kind of memory of earlier time periods, as when they need to exhibit adaptive behavior, the sophisticated designs should be used.

For historical reasons, there is no really standard terminology in this area. In the chemical industry, sophisticated cloning techniques are sometimes called operator modeling (3, Ch. 10). In the robotics industry, Hirzinger's group has used cloning to copy specific skilled movements of human operators, and called this skill learning. [Hirzinger's group is perhaps the most advanced group in the world today applying a broad spectrum of learning-based intelligent controllers to practical real-world applications, including space robots, flexible high-throughput manufacturing robots, medical robots, and others (45).] The neural aerospace company mentioned previously (AAC) has actually occasionally used the word cloning.

Some roboticists may ask what the connection is between cloning as described here, and the older pendant-based methods of training robots. Very simply, the older methods yield a static controller, as previously defined, while the cloning methods yield an ability to respond to sensor inputs $\boldsymbol{X}$; in other words, they can be used to train feedforward, feedback, or even adaptive controllers.

In many applications, cloning approaches are a good place to start, even if the ultimate goal is to develop an optimizing controller. For example, even before developing any automatic controller, one may try to develop a telerobotic interface, to permit a human being to directly control a robot designed for the application. If the human cannot learn to control this robot, one may reconsider the physical robot design. If the human can control it, one can then clone the human behavior, and use the result as the starting point for a more sophisticated learning controller.

Tracking

Tracking controllers are defined as controllers which try to make the plant stay at a desired setpoint, or follow (track) a desired trajectory over time. More precisely, the control actions $\boldsymbol{u}(t)$ are chosen so as to make the actual observed state $\boldsymbol{X}(t)$ match a desired reference trajectory, $\boldsymbol{X}^*(t)$ or $\boldsymbol{X}_r(t)$, supplied by the user. (The setpoint case, also called homeostatic control, is the case where the desired states $\boldsymbol{X}^*(t)$ do not change over time—except when the user changes the setpoint.)

Both in neurocontrol and in classical control, the majority of academic papers published today focus on tracking control. As a result, the literature is extremely complex and somewhat difficult to summarize accurately. Many neural tracking designs are essentially just conventional tracking designs, or adaptive control designs (9,10), with matrices replaced by neural networks. Unfortunately, many researchers have made the mistake of assuming that tracking problems are the only problems of interest to control theory.

Roughly speaking, there is once again a ladder of learning-based designs available:

1. Direct inverse control
2. Model-based or indirect adaptive control, based on the short-term minimization of a simple general-purpose measure of tracking error (usually just square error)
3. Model-based or indirect adaptive control in the short-term minimization of a special purpose, application-specific Liapunov function
4. Hybrid designs, which combine one or more of the previous three, together with the use of a pre-existing fixed feedback controller
5. Designs which convert the tracking problem into a task in multiperiod optimization

As discussed previously, the models required in these designs are sometimes replaced by simple implicit relations like y = kx, where k is a positive scalar. The term *"direct"* is sometimes used to describe implicitly model-based designs of this sort.

True direct inverse control (DIC) was once the most popular form of neurocontrol. DIC was applied most often to robot control (1,46), or to biological models of hand and eye movements (46,47). In DIC, we usually assume that there is a simple relation between the control variables and the position of the robot arm, which can be expressed as $\boldsymbol{X} = \boldsymbol{g}(\boldsymbol{u})$. For example, if $\boldsymbol{u}$ consists of three variables, each controlling the angle of one of the three joints in a robot arm, then $\boldsymbol{g}$ is the function which determines where the hand will be located in spatial coordinates. If the function $\boldsymbol{g}$ happens to be invertible, then there will be a unique solution for $\boldsymbol{u}$, for any vector $\boldsymbol{X}$:

$$\boldsymbol{u}' = \boldsymbol{g}^{-1}(\boldsymbol{X}) \tag{6}$$

In DIC, one tries to learn the function $\boldsymbol{g}^{-1}$, simply by observing pairs of $\boldsymbol{u}(t)$ and $\boldsymbol{X}(t)$ and using supervised learning. Then, to control the arm, one simply sets:

$$\boldsymbol{u}(t) = \boldsymbol{g}^{-1}(\boldsymbol{X}^*(t)) \tag{7}$$

When the mapping from $\boldsymbol{X}$ to $\boldsymbol{u}$ is learned in this simple static way (39), the errors tend to be about 3%—too large for realistic robotic applications. However, when the neural network is also given inputs from past times, very accurate tracking becomes possible (1,47). Miller has shown videos of a system based on this approach which could learn to push an unstable cart around a figure 8 track with very high accuracy, and then readapt (with real-time learning) within three loops around the track after a sudden change in the mass on the cart. Miller also developed a VLSI control board for use in a conventional robot, but the U.S. robotics company involved underwent a reorganization before the product could become widely used (See the article on NEURAL NETWORK ARCHITECTURES for a discussion of neural VLSI design.)

DIC does require the assumption that the function $\boldsymbol{g}$ be invertible. If the vector $\boldsymbol{u}$ has more degrees of freedom than the vector $\boldsymbol{X}$, then this is clearly impossible. Some ANNs have been developed which effectively throw away the extra degrees of freedom in $\boldsymbol{u}$. But most control engineers have moved on to model-based designs, which are usually considered to be more powerful and more general, and which permit a systematic exploitation of the extra control power of any extra degrees of freedom. [See (6) for a discussion of direct versus indirect adaptive control.]

Model-based adaptive control is the dominant form of neurocontrol today in academic publications. The papers by Narendra and coauthors, starting from (48), have played a leading role in this development. [See also his papers in (1,3,11).] A wide variety of designs have been considered, a wide variety of theorems proven, and a wide variety of simulations studied. There have certainly been some real-world applications as well. But again, it is difficult to summarize the literature accurately in a brief overview. In this overview, I will assume a conventional sampled-time approach.

(Narendra usually uses a differential equation formulation, which is more or less equivalent.) Neurocontrollers of this sort had already been implemented by 1987, by Jordan and Rumelhart and by Psaltis et al. (49), but the later more rigorous analysis has been crucial to the use of these methods.

On the whole, most of these designs effectively involve the effort to minimize tracking error at time t + 1, the very next time period:

$$U(t+1) = (\boldsymbol{X}^*(t+1) - \boldsymbol{X}(t+1))^2 \tag{8}$$

(Narendra uses the letter e instead of U. The letter U emphasizes the link to optimization methods, and reminds us that this error is actually a kind of physical cost rather than something like a prediction error.) These designs require the use of an Action network and a Model network, both of which can be adapted in real time. The model network learns to predict $\boldsymbol{X}(t+1)$ as a function of $\boldsymbol{X}(t)$ and $\mathbf{u}(t)$ (and perhaps of earlier information); it is adapted by neuroidentification methods. The Action network inputs $\boldsymbol{X}(t)$ (and earlier information) and outputs $\boldsymbol{u}$(t). The Action network is trained on the basis of derivative feedback, which may be calculated as follows:

$$F\_u_i(t) = \sum_j \frac{\partial X_j(t+1)}{\partial u_i(t)} \cdot \frac{\partial U(t_1)}{\partial X_j(t+1)} \tag{9}$$

In actuality, this calculation may be performed more economically by backpropagating through the Model network; in other words, one may use the dual subroutine for the Model network, in order to reduce the computational costs (12, Ch. 8; 3, Ch. 10). In order to ensure stability, it is important to limit the overall speed of learning in these networks.

Numerous general stability theorems have been proven for this class of design, very similar to the theorems which exist for adaptive control in general. Nevertheless, all of these theorems (both neural and classical) do require some very stringent conditions. In some applications, like certain forms of vibration control, one may expect certain instabilities to be damped out automatically, so that these stringent conditions will be met. Great success has been reported in some applications (30). But in many application domains—like chemical plants and aerospace vehicles—there are major barriers to the use of any standard adaptive control techniques, neural or classical, because of some bad historical experience with instabilities.

There are many plants where actions which appear stabilizing in the short-term (at time $t + 1$) will have the opposite effect in the long-term. Consider, for example, the bioreactor benchmark problem in (1). Lyle Ungar has shown how all kinds of neural and classical adaptive control designs still tend to go unstable when used on that simulated plant. However, when engineers have used multiperiod optimization designs (which account for long-term effects), they have had great success in controlling that plant (50).

In ordinary control engineering, there are actually two standard ways to overcome these potential instabilities. One is to treat the tracking problem as a multiperiod optimization problem. The other is to replace the function U in Eqs. (8) and (9) by an application-specific Liapunov function, which meets some other stringent requirements, related to the dynamics of the plant, which must be known. For certain application-specific areas of nonlinear control, such as stiff robot arms, this has been a very useful approach. Unfortunately, it places great demands on human ingenuity to *find* the Liapunov functions which meet all the requirements, in any complex application. The need to use a simple preordained model of the plant will tend to force the use of a restrictive class of physical plants, as in robotics. Hirzinger's group (38) has shown that substantial improvements in performance are possible, if one explores a wider class of physical plants (like light-weight flexible arms), which then require a more powerful control design.

In any event, neural learning designs need not be an alternative to Liapunov-based adaptive control. Instead, the Critic networks in some reinforcement learning designs (to be described later) may be used as a constructive technique to actually find the Liapunov functions for difficult, complex applications (51). In fact, many of the special-purpose Liapunov functions used in practice actually came from an analytical solution of a multiperiod optimization problem. (See for example the work of Sanner at the University of Maryland, using neural adaptive control for a variety of space robots, including robots built at the university to be controlled from the university after launch.) The neural optimization methods simply offer a numerical solution for the same class of problems, when the analytical solution becomes too complex.

The hybrid neural/classical designs mentioned above are largely beyond the scope of this article. Particularly interesting examples are some of the methods described by Frank Lewis elsewhere in this encyclopedia, the Feedback Error Learning design of Kawato et al. (1), and the Seraji-like Neural Adaptive Controller as described by Richard Saeks of AAC at many conferences. All of these designs use the traditional feedback controller to insure stability even before learning begins, but also exploit real-time learning in order to improve performance or stability over time.

Finally, to convert a tracking problem into a multiperiod optimization problem, one need only minimize U (as defined in Eq. (8) over future time periods. In principle, one tries to pick $\boldsymbol{u}(t)$ so as to minimize (or maximize):

$$\sum_{\tau=t+1}^{\infty} U(\tau) \tag{10}$$

In practice, one can then add additional terms to the utility (or cost) function, so as to minimize some combination of tracking error, energy consumption, jerkiness, depreciation, and so on. This class of designs has very strong stability properties. For example, Model-Predictive Control (MPC), a method in this class, has received wide acceptance in the chemical industry, where conventional forms of adaptive control are usually considered too unstable to be trusted.

Explicit Multiperiod Optimization

Until recently, explicit multiperiod optimization was the method of choice for very difficult, realistic challenges in neurocontrol. Because the method is very straightforward and exact, it still deserves a place in virtually every serious toolbox for neurocontrol.

In the simplest version of the method, the user must supply a deterministic Model of the plant to be controlled (a Model which could be based on neuroidentification) and a utility function $U(\boldsymbol{X})$. The goal is to train an Action network, which inputs $\boldsymbol{X}(t)$ and outputs $\boldsymbol{u}(t)$, so as to maximize (or minimize) the sum of U over time.

In each iteration, we start out at time $t = 1$. We use the Model network and the initial version of the Action network to generate a stream of predictions for $\boldsymbol{X}(t)$ from time $t = 1$ up to some final time $t = T$. We then use BTT to calculate the complete gradient of U_{total} with respect to the weights in the Action network. (U_{total} is just the sum of $U(t)$ from $t = 1$ to $t = T$.) We adjust the weights in response to that gradient, and then start a new iteration or quit.

This is more or less equivalent to the classical multiperiod optimization methods called the calculus of variations (20) and differential dynamic programming (52). The main novelty is that BTT allows a faster calculation of derivatives, and the use of neural networks allows a general function approximation capability. Complete pseudocode for the approach may be found in (9, Ch. 8).

This simple version was used in Widrow's classic truck backer-upper (1) and Jordan's robot arm controller (53), both discussed in the 1988 NSF workshop on neurocontrol (1). Sometimes (as in Widorw's case) the time T is actually the time when a control task is completed. Sometimes [as in most of the work by Ford (14,15) and by McAvoy et al. (3, Ch. 10)] there is a fixed look-ahead into an ongoing process; this is sometimes called a receding horizon approach (as in some recent work by Theresa Long on engine control (54) and other work by Acar). In giving talks on this approach, the Ford group has frequently stressed the need to calculate complete gradients accurately—an issue which is often badly confused in the existing literature.

A slight variant of this approach is to adapt a schedule of actions from time $t = 1$ to $t = T$, instead of an Action network. That approach was used in the official DOE/EIA model of the natural gas industry, which I developed circa 1986 (21). It was also used in the cascade phase two design for robot arm control by Uno, Kawato et al. (1), and in the chemical plant controller of McAvoy et al. (3, Ch. 10).

Instead of simply minimizing or maximizing U_{total} in an unconstrained manner, one sometimes needs to minimize it subject to constraints. In that case, we can combine the approach described previously with more classical methods designed to combine gradient information and constraint information, to arrive at a schedule of actions. In fact, the work of McAvoy et al. takes this approach, which may be seen as a useful special case of a more conventional method—nonlinear Model-Predictive Control.

In the robotics area, Hirzinger has also applied a variant of these approaches very successfully in his outer loop optimization (45). Hrycej of Daimler-Benz has also reported a number of successful applications (55).

Note that all of this work assumes that a Model of the plant is available, and proceeds as if the Model were perfectly exact. One could account for random disturbances or errors, in principle, by using the methods of Differential Dynamic Programming (52). However, there is reason to question the efficiency of these methods in accounting for such effects, relative to the implicit optimization methods described in the next section.

There are other ways to perform explicit multiperiod optimization, without using BTT. Some of these methods involve less accurate methods of computing gradients, or more expensive ways of computing the same derivatives (34). Others are totally derivative-free methods, like evolutionary computing (EC), which includes genetic algorithms. ES works by simulating entire populations of possible control designs and selecting out those with the best overall observed performance. EC can be very useful for small enough control problems, in off-line learning, and it can also be used to provide *initial values* for a gradient-based system. (See the work of Tariq Samad of Honeywell (56) and of Krishnakumar (57) for some practical control work using ECs.) They can provide an alternative to step-by-step learning, in avoiding local minimum problems. In the long-term, however, a brainlike approach would have to involve a totally different sort of stochastic search method for real-time learning in order to enable the solution of larger problems (4,58).

Disadvantages of the explicit approach relative to the implicit approach are: (1) the assumption that the Model is exact; (2) the inability to account for payoffs or costs beyond time T, in the receding horizon approach; (3) the computational cost of simulating T time periods in every cycle of adaptation. It is possible to eliminate the second disadvantage by using a hybrid design, in which a Critic network supplies the derivatives which start up the gradient calculations for $t = T$ (17).

Implicit Multiperiod Optimization

This class of designs is the most complex, sophisticated and brainlike class of designs in neurocontrol. They learn to maximize the sum of future utility without ever developing an explicit schedule or plan for what will happen in the future. In effect, they can solve problems in "planning" without an explicit plan.

More concretely, these designs try to maximize the sum of future utility, in situations where there does not exist an exact, deterministic model of the plant or environment. There may exist a stochastic model, which can be used to simulate the environment, but not a deterministic model. In formal terms, these are learning designs which try to solve general problems in nonlinear stochastic optimization over time. Of course, these designs can still be applied to the special case where the plant happens to be deterministic.

In control theory, there is only one family of algorithms which can find the exact solution to such problems, in a computationally efficient manner: *dynamic programming*. Designs which learn to approximate or converge to the dynamic programming solution are sometimes called *approximate dynamic programming* (ADP) (3) or *neurodynamic programming* (59). As an alternative, these designs are sometimes called *reinforcement learning* designs. [The connections between reinforcement learning, approximate dynamic programming, backwards feedback and neural networks were first discussed—albeit it in very crude form—in 1968 (60).] The most precise label for these designs, favored by experts in this field in their most technical discussions, is the term adaptive critic. The phrase adaptive critic was coined by Bernard Widrow, who implemented the first working neurocontroller in this class (61). This is the most precise label because there are other ways to approximate dynamic programming which do not involve learning, because the designs in this class do not always require neural networks, and because the term reinforcement learning has been used in the past to refer to a very wide range of concepts beyond the scope of this encyclopedia.

In dynamic programming, the user normally supplies a utility function $U(\boldsymbol{X}, \boldsymbol{u})$, and a stochastic model of the environment, which may be written:

$$\boldsymbol{X}(t+1) = \boldsymbol{f}(\boldsymbol{X}(t), \boldsymbol{u}(t), \boldsymbol{e}(t), W_{\mathrm{f}}) \tag{11}$$

where W_{f} represents the parameters or weights of the model and $\boldsymbol{e}(t)$ is a vector of random numbers representing random disturbances. The problem is to find a strategy of action, $\boldsymbol{u}(\boldsymbol{X})$, so as to maximize:

$$\left\langle \sum_{r=0}^{T} U(\boldsymbol{X}(t+i), \boldsymbol{u}(t+i))/(1+r)^{i} \right\rangle \tag{12}$$

where r is a user-supplied parameter corresponding exactly to the idea of an interest rate or discount factor in economics, and where the angle brackets denote the expectation value of this sum. In many applications, T is chosen to be infinity or r is chosen to be zero or both. In some papers, the term $1/(1 + r)$ is called γ. $U(\boldsymbol{X}, \boldsymbol{u})$ often depends only on $\boldsymbol{X}$; however, I include u here for the sake of generality.

In dynamic programming, one solves this problem by solving the Bellman equation, which may be written as:

$$\begin{aligned} J(\boldsymbol{X}(t)) &= \max_{\boldsymbol{u}(t)} \left[U(\boldsymbol{X}(t), \boldsymbol{u}(t)) + \frac{\langle J(\boldsymbol{X}(t+1)) \rangle}{(1+r)} \right] \\ &= \max_{\boldsymbol{u}(t)} \left[U(\boldsymbol{X}(t), \boldsymbol{u}(t)) + \frac{\langle J(\boldsymbol{f}(\boldsymbol{X}(t), \boldsymbol{u}(t), \boldsymbol{e}(t), W_{\mathrm{f}})) \rangle}{(1+r)} \right] \end{aligned} \tag{13}$$

Solving the Bellman equation means finding the function $J(\boldsymbol{X})$ which satisfies this equation. After we have found that function, we simply pick $\boldsymbol{u}(t)$ at all times so as to maximize the right-hand side of this equation. After we know $J(\boldsymbol{X})$, the selection of $\boldsymbol{u}$ is a problem in short-term maximization. In other words, dynamic programming converts a difficult problem in long-term maximization or planning into a more straightforward problem in short-term maximization.

In theory, dynamic programming could be used to solve *all* problems in planning and control, exactly. In practice, the sheer computational cost of solving the Bellman equation becomes prohibitive even for many very small control problems. The cost rises exponentially with the number of variables in the plant or environment. Plants governed by a single state variable are usually manageable, but plants based on ten are usually far too complex.

Adaptive critic designs *approximate* dynamic programming, by learning an approximation to the function $J(\boldsymbol{X})$ (or to its gradient or to something very similar.). The neural network (or other approximator) which approximates the J function (or gradient . . .) is called a Critic. An adaptive critic system is defined as a control system which contains a Critic network, adapted over time through some kind of generalized learning procedure.

The adaptive critic family of designs is extremely large and extremely diverse. It includes some very simple designs, like the Critic/Actor lookup-table system of Barto, Sutton and Anderson (BSA) (62), which has become extremely popular in computer science. As an example, an advanced version of this system has been used to play backgammon. It has achieved master class performance in that game, proving that adaptive critic designs can in fact achieve something like real intelligence. (63). The adaptive critic family also includes more complex, more brainlike designs (3,40,64,65,66), combining a Critic network, an Action network, and a Model network. These more complex designs have demonstrated the ability to handle a variety of difficult test problems in engineering, more effectively than alternative designs, both neural and nonneural. The family also includes a special form of Error Critic first proposed in order to explain certain features of the cerebellum (3, Ch. 13). This form of critic has apparently been extremely successful in some practical but proprietary applications in the automotive sector. Finally, the adaptive critic family also includes two brain and three brain designs which, in my view, should be rich enough and powerful to capture the essence of the higher-level intelligence which exists in the brains of mammals (5,67).

Space does not permit a complete description of these four subfamilies in this article. However, a few general observations may be of use to the reader, to supplement the citations mentioned above.

In the Barto-style family of methods, the most popular methods are the original BSA design (62) and Q-learning of Watkins (68). In the BSA design, the Critic learns to approximate the function $J(\boldsymbol{X})$. Thus the Critic inputs a vector of observed sensor data $\boldsymbol{X}$, and outputs a scalar estimate of J. The Critic is trained by a *temporal difference method,* which is a special case of Heuristic Dynamic Programming (HDP) (69) first published in 1977).

In HDP, one trains the Critic by use of supervised learning. At each time t, the input vector is simply $\boldsymbol{X}(t)$. The target vector is the scalar $U(t) + (J(t + 1)/(1 + r))$, the right-hand side of the Bellman equation, using the Critic itself to estimate $J(t + 1)$. There are two counterintuitive aspects to this design: (1) the training for time t cannot be carried out until after the data for $t + 1$ are known (or simulated); (2) the weights in the Critic are adapted as if the target is constant, even though we know that a change in those weights would change the estimate of $J(t + 1)$! Many researchers have responded to the second aspect by reinventing a "new" method, which adjusts the weights W_J so as to minimize:

$$\left(\hat{J}(\ddot{\boldsymbol{X}}(t), W_J) - (U(t) + \frac{\hat{J}(\boldsymbol{X}(t+1), W_J)}{1+r} \right)^2 \tag{14}$$

where W_J are the weights in the Critic network, and J-hat refers to the Critic network. Unfortunately, this sensible-looking procedure leads to incorrect results almost always, at least in the linear-quadratic case (69). HDP proper always converges to the correct results in that case.

The Barto-style family can sometimes be very robust and very efficient, when the action vector $\boldsymbol{u}(t)$ actually consists of a small number of discrete choices. When the action variables are truly continuous, or when there are many action variables, the methods are still robust, but extremely slow to converge. Furthermore, the validity of these designs requires the assumption that $\boldsymbol{X}(t) = \boldsymbol{r}(t)$, that is, that the plant being controlled is completely observable. In practice, we can overcome that limitation by estimating the state vector $\boldsymbol{r}(t)$, and providing the entire state vector as input to the Critic; however, the estimation of the state vector tends to require something like a neuroidentification component.

As we climb up the ladder of designs, the next method beyond the Barto-style methods is ADAC—the Action-Dependent Adaptive Critic, closely related to Q-learning, and developed independently in 1989 (3). (In fact, many authors have reinvented ADAC and claimed a new method for modified Q-learning.) This design was applied successfully by McDonnell-Douglas in several real-world problems, including the manufacture of high-quality carbon-carbon composite parts and simulated control of damaged F-15s (3). ADAC can handle larger problems than the Barto-style family, but it still has limits on scale, and problems related to persistence of excitation. In a strict sense, there are actually three designs in the ADAC group the McDonnell-Douglas work used the simplest of the three, which is called Action-Dependent HDP (ADHDP).

Next up the ladder are several designs which I have called Model-Based Adaptive Critics (MBAC) or Brain-Like Intelligent Control (40,64,65,66). These designs all require the use of three core components; a Critic, an Action network, and a Model. In the simplest design, the Critic is again trained by HDP. (Some authors now use the term HDP to refer to this entire design.) The Model is typically trained by some sort of neuroidentification procedure. The Action network is trained based on the derivatives of $J(t + 1)$ with respect to the weights in the Action network; these derivatives are calculated by backpropagating through the Critic, the Model, and the Action network, in that order. [See Refs. (3) and (9, Ch. 8) for critical implementation details.] The Model plays a crucial role here, in distributing the feedback from the Critic to specific Action variables; this kind of rational distribution or credit assignment allows these designs to handle larger-scale problems than the simple two-network designs can. This subfamily also contains two more advanced designs, Dual Heuristic Programming (DHP) and Globalized DHP (GDHP), proposed before 1981 (12, Ch. 7), and their action-dependent variants. These more advanced designs use the Model in training the Critic itself, so as to improve credit assignment still further. Research in neuroscience has shown that certain parts of the brain do indeed seem to learn to predict as a Model network would (5). Grossberg has pointed out many times that a large part of the results from animal learning experiments require the existence of an expectations system in the brain.

In formal terms, DHP is a learning-based approximation to the stochastic generalization of the classical Pontryagin principle, which was given in Ref. 3 (Ch. 13). Some additional recent implementations are cited in Ref. 4. (See also Ref. 70.)

Unfortunately, the term *model-based reinforcement learning* has sometimes been broadened to include designs of the Barto subfamily. In true, full-fledged brainlike systems, one would expect a Model network to be used to perform at least three functions: (1) the credit assignment function, discussed in the previous paragraph; (2) the estimation of the state vector $r(t)$ to be input to the Critic and Action network; (3) simulations or dreams (71, 1, Ch. 8) of possible future $r(t)$, for use in training the Critic and Action network. Only the first function really changes the core design of the adaptive critic proper; the others simply alter the flow of inputs into that core.

Finally, some new designs have recently begun to emerge in theory which try to bridge the gap between brainlike designs and the major features of higher-order intelligence in the mammalian brain. The GDHP design (71) does actually meet certain basic tests (40) which a credible, first-order model of intelligence in the brainshould meet. However, if the sockets in that design are filled in with conventional feedforward or Hebbian neural networks, the system is unable to learn to solve certain basic problems in spatial navigation (33,34) which a truly brainlike system should be able to handle. This difficulty could be solved fairly easily, in principle, by coupling together two entire "brains" (adaptive critic systems) in one design—a higher-order master system built out of SRN networks (33,34) and a fast low-order slave system based on feedforward networks (15). However, there is still a need for a kind of middle brain as well, in order to explain phenomena such as task learning, temporal chunking, spatial chunking, the basal ganglia, and so on. The pathway to developing such neurocontrollers now seems fairly clear, but considerable research remains to be done (5,67,4,58). Furthermore, many researchers would argue that there exists another gap, between the intelligence one observes in the ordinary mammalian brain and the higher-order intelligence or consciousness in the human mind (72,12, Ch. 10); however, one may still expect that an understanding of the former should contribute to a greater possibility of understanding the latter. Difficult testbed problems in engineering and computer science will play a crucial role in permitting the development of mathematical concepts necessary to both sorts of understanding.

BIBLIOGRAPHY

1. W. T. Miller, R. Sutton, and P. Werbos (eds.), *Neural Networks for Control,* Cambridge, MA: MIT Press, 1990, now in paperback. Warning: pseudo in Ch. 2. contains errors; see Refs. 3 and 17. Also, the discussion of recurrent networks is somewhat dated.
2. N. Wiener, *Cybernetics, or Control and Communications in the Animal and the Machine,* 2nd ed., Cambridge, MA: MIT Press, 1961.
3. D. White and D. Sofge (eds.), *Handbook of Intelligent Control,* New York: Van Nostrand, 1992.
4. P. Werbos, Values, Goals and Utility in an Engineering-Based Theory of Mammalian Intelligence, in K. H. Pribram (ed.), *Brain and Values,* Hillsdale, NJ: Erlbaum, 1998.
5. P. Werbos, Learning in the brain: An engineering interpretation. In K. Pribram (ed.), *Learning as self-organization* Mahwah, NJ: Erlbaum 1996.
6. J. Houk, J. Davis, and D. Beiser (eds.), *Models of Information Processing in the Basal Ganglia,* Cambridge, MA: MIT Press, 1995.
7. J. C. Houk, J. Keifer, and A. Barto, Distributed motor commands in the limb premotor network, *Trends Neurosci.,* **16**: 27–33, 1933.
8. P. Werbos, Econometric techniques: Theory versus practice, *Energy,* **15** (3/4): 1990.
9. K. Narendra and A. Annaswamy, *Stable Adaptive Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1989.
10. K. J. Atrom and B. Wittenmark, *Adaptive Control,* New York: Addison-Wesley, 1989.
11. M. M. Gupta and N. K. Sinha (eds.), *Intelligent Control Systems,* Piscataway, NJ: IEEE Press, 1996, Chap. 13.
12. P. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting,* New York: Wiley, 1994.

13. P. Werbos, Neurocontrol in A. Maren (ed.), *Handbook of Neural Computing Applications,* San Diego, CA: Academic, 1990.
14. L. Feldkamp et al., Enabling concepts for applications of neurocontrol, in K. S. Narendra (ed.), *Proc. 8th Yale Workshop Adaptive Learning Systems,* New Haven, CT: Prof. Narendra, Dept. of Electrical Eng., Yale U., 1994. See also Unravelling dynamics with recurrent networks: application to engine diagnostics, in *Proc. 9th Yale Workshop Adaptive Learning Systems*, New Haven, CT: Prof. Narendra, Dept. Electrical Eng., Yale U., 1996.
15. G. V. Puskorius, L. A. Feldkamp, and L. I. Davis, Dynamic neural network methods applied to on-vehicle idle speed control, *Proc. Int. Conf. Neural Netw.* IEEE, 1996.
16. J. S. Baras and N. S. Patel, Information state for robust control of set valued discrete time systems, *Proc. 34th Conf. Decision and Control (CDC),* IEEE Press, 1995, p. 2302.
17. P. Werbos, Optimization methods for brain-like intelligent control, *Proc. IEEE Conf. CDC,* IEEE, 1995. Also see Ref. 40.
18. R. M. Pap, *Design of neurocontroller to operate active flight surfaces,* Technical report to NSF grant ECS-9147774. Chattanooga, TN: Accurate Automation Corp., April 2, 1992.
19. G. Sachs et al., *Robust control concept for a hypersonic test vehicle, AIAA-95-6061,* Washington, DC: Am. Inst. Aero and Astro, 1995.
20. B. Widrow and E. Walach, *Adaptive Inverse Control,* Englewood Cliffs, NJ: Prentice-Hall, 1994.
21. M. L. Padgett and T. Lindblad (eds.), *Virtual Intelligence,* SPIE Proc. Series **2878**, SPIE—Int. Soc. Opt. Eng., Bellingham, Wash., 1996.
22. I. Peterson, Ribbon of chaos, *Science News,* **139** (4): Jan. 26, 1991.
23. A. E. Bryson and Y.-C. Ho, *Applied Optimal Control,* New York: Hemisphere Publishing, 1975.
24. D. Prokhorov, *Adaptive Critic Designs and Their Applications,* Ph.D. thesis, Electr. Eng. Dept., Texas Tech U., Lubbock, TX: Dec. 1997.
25. J. A. Suykens, B. DeMoor, and J. Vandewalle, Nlq theory: A neural control framework with global asymptotic stability criteria, *Neural Netw.,* **10**: 615–637, 1997.
26. J. A. Suykens, J. Vandewalle, and B. DeMoor, Lur'e systems with multilayer perceptron and recurrent neural networks: Absolute stability and dissipativity, *IEEE Trans. Autom. Control,* in press.
27. P. Simpson (ed.), *Neural Network Applications,* Piscataway, NJ: IEEE Press, 1996, chap. 1–6.
28. P. Werbos, Maximizing long-term gas industry profits in two minutes in Lotus using neural network methods, *IEEE Trans. Syst. Man. Cybern.,* **19**: 315–333, 1989.
29. R. A. Jacobs et al., Adaptive mixtures of local experts, *Neural Computation,* **3**: 79–87, 1991.
30. T. W. Long, A learning controller for decentralized nonlinear systems, *Amer. Control Conf.,* IEEE Press, 1993.
31. E. Fiesler and R. Beale (eds.), *Handbook of Neural Computation,* New York: Oxford Univ. Press, 1996, chap. A.2.
32. A. R. Barron, Universal approximation bounds for superpositions of a sigmoidal function, *IEEE Trans. Inf. Theory* **39**: 930–945, 1993.
33. P. Werbos and X. Z. Pang, Generalized maze navigation: SRN critics solve what feedforward or Hebbian nets cannot. *Proc. Conf. Syst., Man Cybern. (SMC)* (Beijing), IEEE, 1996. (An earlier version appeared in WCNN96 and Yale96(11).)
34. X. Z. Pang and P. Werbos, Neural network design for J function approximation in dynamic programming, *Math. Modelling Sci. Comput.* **5** (21): 1996. Available also as adap=org 9806001 from xxx.lanl.gov/form, setting "Other groups" to "nlin-sys."
35. V. Roychowdhury, K. Siu, and A. Orlitsky (eds.), *Theoretical Advances in Neural Computation and Learning,* Boston: Kluwer, 1994.
36. J. T. Lo, Adaptive system identification by nonadaptively trained neural networks, *Proc. Int. Conf. Neural Netw.,* IEEE, 1996, pp. 2066–2071.
37. M. Obal and A. Das, *Proc. Workshop Neural Decision and Control Technol. Aerospace Syst.* Phillips Laboratory (AFMC), Kirtland Air Force Base, Albuquerque, NM, Feb. 1997.
38. *Neural Network System Identification: Final Report,* Contract NAS1-18225 Task 9, for NASA Langley, Harris Corp. Gov't Aerospace Sys. Div., Melbourne, FL, 1993.
39. H. Berenji, A reinforcement learning-based algorithm for fuzzy logic control, *Int. J. Approx. Reasoning,* **6** (2): Feb. 1992.
40. P. Werbos, Optimal neurocontrol: Practical benefits, new results and biological evidence, *Proc. World Congr. Neural Netw. (WCNN95),* Erlbaum, 1995. This and several other papers cited here may be obtained from links on www.eng.nsf.gov/ecs/werbos.htm
41. G. E. P. Box and G. M. Jenkins, *Time-Series Analysis: Forecasting and Control,* San Francisco: Holden-Day, 1970.
42. J. T. Lo, Synthetic approach to optimal filtering. *IEEE Trans. Neural Networks,* **5**: pp. 803–811, September 1994. See also J. T. Lo, Adaptive optimal filtering by pretrained neural networks, *Proc. World Congr. Neural Netw.,* Mahwah, NJ: Erlbaum, 1995, pp. 611–615.
43. T. Kohonen, The self-organizing map, *Proc. IEEE* **78** (9): Sept. 1990. See also more recent book which elaborates on the issue of probability densities: H. Ritter, T. Martinez, and K. Schulten, *Neural Computation and Self-Organizing Maps,* Reading, MA: Addison-Wesley, 1992.
44. B. Widrow and F. W. Smith, Pattern-recognizing control systems, *Computer Inf. Sci. (COINS) Proc.,* Spartan, 1964.
45. G. Hirzinger et al., Neural Perception and manipulation in robotics, in M. van der Meer and R. Schmidt (eds.), *Kunstliche Intellkigenz, Neuroinformatik, und Intelligente Systeme,* DLR, Berlin, 1996.
46. M. Kuperstein, INFANT neural controller for adaptive sensory-motor coordination, *Neural Netw.,* **4** (2): 1991.
47. P. Gaudiano and S. Grossberg, Vector associative maps: Unsupervised real-time error-based learning and control of movement trajectories, *Neural Netw.,* **4**: 147–183, 1991.
48. K. S. Narendra and K. Parasarathy, Identification and control of dynamical systems using neural networks, *IEEE Trans. Neural Networks,* **1**: 4–27, 1990.
49. D. Psaltis, A. Sideris, and A. Tamamura, Neural controllers, in *Proc. Int. Conf. Neural Netw.,* IEEE, 1987, pp. IV-551–558.
50. F. Yuan et al., A simple solution to the bioreactor benchmark problem by application of Q-learning, *Proc. World Congr. Neural Netw.,* Mahwah, NJ: Erlbaum, 1995.
51. P. Werbos, New methods for the automatic construction of Liapunov functions. In P. Pribram (ed.), *Origins: Brain and Self-Organization,* Mahwah, NJ: Erlbaum, 1994, pp. 46–52.
52. D. Jacobson and D. Mayne, *Differential Dynamic Programming,* New York: American Elsevier, 1970.
53. M. Jordan, Generic constraints on underspecified target trajectories. In *Proc. IJCNN,* IEEE, June 1989.
54. D. L. Simon and T. W. Long, Adaptive optimization of aircraft engine performance using neural networks, *AGARD Conf. Proc.* 572 (conference held September 25–29, 1995, Seattle). NATO Advisory Group for Aerospace Research & Development, pp. 34-1–34-14. For related material, contact Neurodyne at Cambridge, Mass.

55. T. Hrycej, Model-based training method for neural controllers. In I. Aleksander and J. Taylor (eds.), *Artificial Neural Networks* 2, Amsterdam: North Holland, 1992, pp. 455–458.

56. T. Samad and W. Foslien, Parametrized neurocontrollers, *Proc. IEEE Int. Symp. Intell. Control,* Piscataway, NJ: IEEE, 1993.

57. K. Krishnakumar and J. C. Neidhofer, Immunized artificial systems—concepts and applications, in *Genetic Algorithms in Computers and Engineering,* New York, NY: Wiley, 1997.

58. P. Werbos, A Brain-Like Design To Learn Optimal Decision Strategies in Complex Environments, in M. Karny, K. Warwick, and V. Kurkova (eds.), *Dealing with Complexity: A Neural Networks Approach,* London: Springer, 1998. Also in S. Amari and N. Kasabov, *Brain-Like Computing and Intelligent Information Systems,* London: Springer, 1998. See also international patent application #WO 97/46929, filed June 1997, published Dec. 11, 1997.

59. D. P. Bertsekas and J. N. Tsiskilis, *Neuro-dynamic Programming,* Belmont, MA: Athena Scientific, 1996.

60. P. Werbos, The elements of intelligence. *Cybernetica* (Namur), **3**: 1968.

61. B. Widrow, N. Gupta, and S. Maitra, Punish/reward: learning with a Critic in adpative threshold systems, *IEEE Trans. Syst. Man. Cybern.,* **5**: 455–465, 1973.

62. A. Barto, R. Sutton, and C. Anderson, Neuronlike adaptive elements that can solve difficult learning control problems, *IEEE Trans. Syst. Man. Cybern.,* **13**: 834–846, 1983.

63. G. J. Tesauro, Practical issues in temporal difference learning. *Machine Learning,* **8**: 257–277, 1992.

64. D. Prokhorov and D. Wunsch, Adaptive critic designs, *IEEE Trans. Neural Networks,* **8** (5): 997–1007, 1997.

65. P. Eaton, D. Prokhorov, and D. Wunsch, Neurocontrollers for ball-and-beam system, in *Proc. Artificial Neural Netw. Eng. (ANNIE),* ASME Press, 1996.

66. N. Visnevski and D. Prokhorov, Control of a nonlinear multivariable system with adaptive critic designs, in *Proc. Artificial Neural Netw. Eng. (ANNIE),* ASME Press, 1996.

67. P. Werbos, A hyrid hierarchical neural-AI model of mammal-like intelligence, *Proc. Syst. Man. Cybern. 1997,* Piscataway, NJ: IEEE Press, 1997.

68. C. J. C. H. Watkins, *Learning From Delayed Rewards,* Ph.D. thesis, University of Cambridge, England, 1989. See also Watkins and Dayan, Technical note: Q-learning, *Machine Learning,* **8** (3/4): 279–292, 1992.

69. P. Werbos, Consistency of HDP applied to a simple reinforcement learning problem, *Neural Netw.,* **3** (2): 179–189, 1990.

70. L. Dolmatova and P. Werbos, Traps and tricks in standard benchmark problems for neurocontrol, in A. Meystel (ed.), *Proc. 1997 Int. Conf. Intell. Syst. Semiotics,* NIST Special publication 918, Washington, DC: U.S. Government Printing Office, 1997.

71. P. Werbos, Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research, *IEEE Trans. Syst. Man. Cybern.,* **17**: 7–20, 1987.

72. P. Werbos, Optimization: A Foundation for understanding consciousness. In D. Levine and W. Elsberry (eds.), *Optimality in Biological and Artificial Networks?,* Mahwah, NJ: Erlbaum, 1996.

PAUL J. WERBOS
National Science Foundation

NEUROFUZZY SYSTEMS. See FUZZY NEURAL NETS.
NEUROSCIENCE. See NEUROTECHNOLOGY.