

1 Evolutionary and Neural Synthesis of Intelligent Agents

Karthik Balakrishnan and Vasant Honavar

1.1 Introduction

From its very inception in the late 1950s and early 1960s, the field of Artificial Intelligence, has concerned itself with the analysis and synthesis of *intelligent agents*. Although there is no universally accepted definition of intelligence, for our purposes here, we treat any software or hardware entity as an intelligent agent, provided it demonstrates behaviors that would be characterized as *intelligent* if performed by normal human beings under similar circumstances. This generic definition of agency applies to robots, software systems for pattern classification and prediction, and recent advances in mobile software systems (*softbots*) for information gathering, data mining, and electronic commerce. The design and development of such agents is a topic of considerable ongoing research and draws on advances in a variety of disciplines including artificial intelligence, robotics, cognitive systems, design of algorithms, control systems, electrical and mechanical engineering, programming languages, computer networks, distributed computing, databases, programming languages, statistical analysis and modeling, etc.

In very simplistic terms, an agent may be defined as an entity that *perceives* its environment through *sensors* and *acts* upon it through its *effectors* [76]. However, for the agents to be useful, they must also be capable of *interpreting perceptions*, *reasoning*, and choosing actions *autonomously* and in ways suited to achieving their intended *goals*. Since the agents are often expected to operate reliably in unknown, partially known, and dynamic environments, they must also possess mechanisms to *learn* and *adapt* their behaviors through their interactions with their environments. In addition, in some cases, we may also require the agents to be *mobile* and move or access different places or parts of their operating environments. Finally, we may expect the agents to be *persistent*, *rational*, etc., and to work in groups, which requires them to *collaborate* and *communicate* [76, 87, 60].

It can be seen that this definition of agents applies to robots that occupy and operate in physical environments, software systems for evaluation and optimization, *softbots* that inhabit the electronic worlds defined by computers and computer networks, and even *animals* (or biological agents) that cohabit this planet. Robots have been used in a wide variety of application scenarios like geo, space, and underwater exploration, material handling and delivery, secu-

rity patrolling, control applications in hazardous environments like chemical plants and nuclear reactors, etc., [14, 18, 90, 62]. Intelligent software systems have been extensively used as automated tools for diagnosis, evaluation, analysis and optimization. Softbots are being increasingly used for information gathering and retrieval, data mining, e-commerce, news, mail and information filtering, etc., [7].

In general, an agent can be characterized by two elements: its *program* and its *architecture*. The agent program is a mapping that determines the actions of the agent in response to its sensory inputs, while architecture refers to the computing and physical medium on which this agent program is executed [76]. For example, a *mail filtering agent* might be programmed in a language such as C++ and executed on a computer, or a robot might be programmed to move about and collect empty soda cans using its *gripper arm*. Thus, the architecture of the agent, to a large extent, determines the *kinds* of things the agent is capable of doing, while the program determines *what* the agent does at any given instance of time. In addition to these two elements, the agent *environment* governs the kinds of tasks or behaviors that are within the scope of any agent operating in that environment. For instance, if there are no soda cans in the robot's environment, the can collecting behavior is of no practical use.

Given a particular agent environment the question then is, how can we design agents with the necessary behaviors and abilities to solve a given task (or a set of tasks)?

The field of Artificial Intelligence (AI) has long concerned itself with the design, development, and deployment of intelligent agents for a variety of practical, real-world applications [73, 86, 81, 76]. A number of tools and techniques have been developed for synthesizing such agent programs (e.g., expert systems, logical and probabilistic inference techniques, case-based reasoning systems, etc.) [73, 24, 40, 17, 16, 76].

However, designing programs using most of these approaches requires considerable knowledge of the application domain being addressed by the agent. This process of *knowledge extraction* (also called knowledge engineering) is often difficult either owing to the lack of precise knowledge of the domain (e.g., medical diagnosis) or the inability to procure knowledge owing to other limitations (e.g., detailed environmental characteristics of, say, planet Saturn). We thus require agent design mechanisms that work either with little domain-specific knowledge or allow the agent to acquire the desired information through its own experiences. This has led to considerable development of the field of *machine learning* (ML), which provides a variety of modes and

methods for *automatic synthesis* of agent programs [35, 36, 44, 74, 58].

1.2 Design of Biological Agents

Among the processes of natural adaptation responsible for the design of biological agents (animals), probably the most crucial is that of *evolution*, which excels in producing agents designed to overcome the constraints and limitations imposed by their environments. For example, *flashlight fish* (*photoblepharon palpebratus*), inhabits deep-waters where there is little surface light. However, this is not really a problem since evolution has equipped these creatures with *active vision* systems. These fishes produce and emit their own light, and use it to detect obstacles, prey, etc. This vision system is unlike any seen on surface-dwelling organisms [27].

Apart from evolution, *learning* is another biological process that allows animals to successfully adapt to their environments. For instance, animals learn to respond to specific stimuli (*conditioning*) and ignore others (*habituation*), recognize objects and places, communicate, engage in species-specific behaviors, etc., [51, 23, 27, 52]. Intelligent behavior in animals emerges as a result of interaction between the information processing structures possessed by the animal and the environments that they inhabit. For instance, it is well known that marine turtles (e.g., *Chelonia mydas*) lay their eggs on tropical beaches (e.g., Ascension Island in the Atlantic ocean). When the eggs hatch, the young automatically crawl to the water without any kind of parental supervision. They appear to be genetically programmed to interpret cues emanating from large bodies of water or patches of sky over them [27]. Animals are also capable of learning features of specific environments that they inhabit during their lifetime. For instance, rodents have the ability to learn and successfully navigate through complex mazes [82, 61].

These processes of natural adaptation, namely, evolution and learning, play a significant role in shaping the behaviors of biological agents. They differ from each other in some significant aspects including the spatio-temporal scales at which they operate. While learning operates on individuals, evolution works over entire populations (or species). Further, learning operates during the lifetime of the individual and is presumably aided by long lifespans, while evolution works over generations, well beyond an individual's effective lifespan [1].

Despite these apparent differences, evolution and learning work synergis-

tically to produce animals capable of surviving and functioning in diverse environments. While the architectures of biological agents (e.g., digestive, respiratory, nervous, immune, and reproductive systems) are shaped by evolution, the agent programs (e.g., behaviors like foraging, feeding, grooming, sleeping, escape, etc.) are affected and altered by both evolutionary and learning phenomena. In such cases, evolution produces the *innate* (or genetically programmed) behaviors which are then modified and contoured to the animal's experiences in the specific environment to which it is exposed. Thus, by equipping the agents with good designs and instincts, evolution allows them to survive sufficiently long to learn the behaviors appropriate for the environment in question.

Although the processes of evolution and learning are reasonably well detailed, there are still many gaps to be filled before a complete understanding of such processes can be claimed. Ongoing research in the neurosciences, cognitive psychology, animal behavior, genetics, etc., is providing new insights into the exact nature of the mechanisms employed by these processes: the structures they require and the functions they compute. This has led to many new hypotheses and theories of such mechanisms. *Computational modeling* efforts complement such research endeavors by providing valuable tools for *testing* theories and hypotheses in controlled, simulated environments. Such modeling efforts can potentially identify and suggest avenues of further research that can help fill in the gaps in current human understanding of the modeled processes [10].

As outlined in the previous sections, the enterprise of designing agents has to take into account three key aspects: the nature of the environment, agent architecture, and agent program. The primary focus of this book is on evolutionary synthesis of intelligent agents with a secondary focus on the use of learning mechanisms to enhance agent designs. The chapters that follow consider the design of agent programs using processes that are loosely modeled after biological evolution. They explore several alternative architectures and paradigms for intelligent agent architectures including *artificial neural networks*. They provide several examples of successful use of evolutionary and neural approaches to the synthesis of agent architectures and agent programs.

1.3 Agent Programs

As we mentioned earlier, the agent *program* is primarily responsible for the behavior of the agent or robot in a given environment. The agent program determines the sensory inputs available to the agent at any given moment in time, processes the inputs in ways suited to the goals (or functionalities) of the agent, and determines appropriate agent actions to be performed.

In order to be used in practice, these programs have to be *encoded* within the agents using an appropriate *language* and the agents must possess mechanisms to *interpret* and *execute* them. Some of the earliest programs were realized using *circuits and control systems* that directly sensed input events, processed them via appropriate *transfer functions*, and directly controlled the output behaviors of the robot [46]. Examples of such representations include systems used in the control of industrial robots, robotic arms, etc. [2]. However, these approaches require the transfer function to be known and appropriately implemented, which is often difficult in practice. In addition, these control mechanisms are rather inflexible and reprogramming the robot for a different behavior or task may entail extensive changes.

The advent of computers and their ability to be effectively *reprogrammed*, opened up entirely new possibilities in the design of agent programs for modern-day robots and software agents. In these cases, the agent programs are written in some computer language (e.g., C++, Java, LISP), and executed by the computer. The program receives the necessary inputs from the agent sensors, processes them appropriately to determine the actions to be performed, and controls the agent actuators to realize the intended behaviors. The sensors and actuators may be physical (as in the case of robots) or virtual (as in the case of most software agents). The behavior of the agent can be changed by changing the agent program associated with it.

The question then is, how can we develop agent programs that will enable the agent to exhibit the desired behaviors?

Designing Control Programs for Agent Behaviors

Many contemporary agents make use of programs that are *manually* developed. This is a daunting task, given the fact that the agent-environment interactions exhibit a host of a priori unknown or unpredictable effects. In addition, complex agent behaviors often involve tradeoffs between multiple competing alternatives. For example, suppose a robot has the task of clearing a room by pushing boxes to the walls. Let us also assume that the robot has limited sens-

ing ranges that prevent it from observing the contents of the entire room and it does not have any means to remember the positions of boxes it has observed in the past. Suppose this robot currently observes two boxes. Which one should it approach and push? This decision is critical as it directly affects the subsequent behaviors of the robot. We may program the robot to approach the closer of the two boxes, but can we be sure that such a decision made at the local level will indeed lead to any kind of globally optimal behavior? Manually designing control programs to effectively address such competing alternatives is an equally challenging proposition.

We thus need approaches to *automate* the synthesis of scalable, robust, flexible, and adaptive agent programs for a variety of practical applications. In recent years two kinds of automatic design approaches have met with much success: *discovery* and *learning*. Approaches belonging to the former category typically include some mechanism to *search* the space of agent programs in the hope of finding or discovering a good one. Each program found during this search is evaluated in environments that are representative of those that are expected to be encountered by the agent and the best ones are retained. Some discovery approaches (e.g., evolutionary search) use these evaluations to *guide* or *focus* the search procedure, making the process more efficient. The latter category includes approaches that allow the robot behaviors to be modified based on the experiences of the agent, i.e., the robot *learns* the correct behaviors based on its experience in the environment.

In the remainder of this chapter we will introduce two paradigms that aid in the automatic synthesis of agent programs. Artificial neural networks, finite state automata, rule-based production systems, Lambda Calculus, etc., offer alternative computational models for the design of agent programs. As shown by the pioneering work of Turing, Church, Post, Markov, and others, they are all essentially equivalent in terms of the class of information processing operations that they can realize [83, 56, 47]. However, in practice, each paradigm has its own advantages and disadvantages in the context of specific applications. It is therefore not at all uncommon to use hybrid models that integrate and exploit the advantages of multiple paradigms in synergistic and complementary ways [38, 80, 26]. In what follows, we focus primarily on evolutionary synthesis of neural architectures for intelligent agents. However, many of the same issues arise in the automated synthesis of agents with other information processing architectures as well (e.g., rule-based systems).

1.4 Artificial Neural Networks as Agent Programs

Artificial neural networks offer an attractive paradigm of computation for the synthesis of agent programs for a variety of reasons including their potential for *massively parallel computation*, *robustness* in the presence of noise, *resilience* to the failure of components, and amenability to *adaptation* and *learning* via the modification of computational structures, among others.

What are Artificial Neural Networks?

Artificial neural networks are models of computation that are inspired by, and loosely based on, the nervous systems in biological organisms. They are conventionally modeled as massively parallel networks of simple computing elements, called *units*, that are connected together by *adaptive links* called *weights*, as shown in Figure 1.1. Each unit in the network computes some simple function of its inputs (called the *activation function*) and propagates its outputs to other units to which it happens to be connected. A number of activation functions are used in practice, the most common ones being the *threshold*, *linear*, *sigmoid*, and *radial-basis* functions. The weights associated with a unit represent the strength of the synapses between the corresponding units, as will be explained shortly. Each unit is also assumed to be associated with a special weight, called the *threshold* or *bias*, that is assumed to be connected to a constant source of +1 (or a -1). This threshold or bias serves to modulate the firing properties of the corresponding unit and is a critical component in the design of these networks.

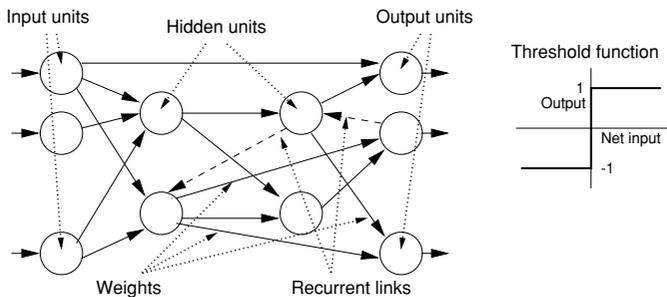


Figure 1.1
Artificial neural network (left) and the bipolar threshold activation function (right).

The input to an n -input (including the bias) unit is typically represented by a pattern vector $\mathbf{X} \in \mathcal{R}^n$ or in the case of binary patterns, by a binary vector $\mathbf{X} \in [0, 1]^n$. The weights associated with an n -input unit i are typically represented by an n -dimensional weight vector $\mathbf{W}_i \in \mathcal{R}^n$. By popular convention, the first element of the weight vector usually represents the threshold (or bias). The input activation of a unit i , represented by A_i , in response to a pattern \mathbf{X} on its input links is usually given by the vector dot product: $A_i = \mathbf{W}_i \cdot \mathbf{X}$. The output of the unit is a function of A_i and is dictated by the activation function chosen. For example, the *bipolar threshold* activation function, shown in Figure 1.1, produces: $O_i = 1$ if $A_i = \mathbf{W}_i \cdot \mathbf{X} \geq 0$ and $O_i = -1$ otherwise.

Units in the network that receive input directly from the environment are referred to as *input units*, while the units that provide the environment with the results of network computations are called *output units*. In conventional neural network terminology, the set of input and output units are said to reside in *input* and *output layers*. In addition to these kinds of units, the networks can also have other units that aid in the network computations but do not have a direct interface to or from the external environment. Such units are referred to as *hidden units*. Often, the hidden units critically determine the kinds of mappings or computations the networks are capable of performing.

In a typical neural network the activations are propagated as follows. At any given instance of time, the input *pattern* is applied to the input units of the network. These input activations are then propagated to the units that are connected to these input units, and the activations of these second layer units are computed. Now the activations of these units are propagated via their output links to other units, and this process continues until the activations reach the units in the output layer. Once the computations of the output layer units are complete (or in the case of recurrent networks the network activity stabilizes), the resulting firing pattern across the output layer units is said to be the output of the network in response to the corresponding input pattern. Thus, in a typical neural network, activations enter the input units, propagate through links and hidden units, and produce an activation in the units of the output layer.

A wide variety of artificial neural networks have been studied in the literature. Apart from differences stemming from the activation functions used, neural networks can also be distinguished based on their topological organization. For instance, networks can be single-layered or multi-layered; sparsely connected or completely connected; strictly layered or arbitrarily connected; composed of homogeneous or heterogeneous computing elements, etc. Perhaps the most important architectural (and hence functional) distinction is be-

tween networks that are simply *feed-forward* (where their connectivity graph does not contain any directed cycles) and *recurrent* (where the networks contain feedback loops). Feed-forward networks can be trained via a host of simple learning algorithms and have found widespread use in pattern recognition, function interpolation, and system modeling applications. In contrast to feed-forward networks, recurrent networks have the ability to remember and use past network activations through the use of recurrent (or feedback) links. These networks have thus found natural applications in domains involving temporal dependencies, for instance, in sequence learning, speech recognition, motion control in robots, etc. For further details regarding artificial neural networks and their rather chequered history, the reader is referred to any of a number of excellent texts [15, 32, 45, 22, 43, 31, 74].

Design of Artificial Neural Networks

As may be inferred, the input-output mapping realized by an artificial neural network is a function of the numbers of units, the functions they compute, the topology of their connectivity, the strength of their connections (weights), the control algorithm for propagating activations through the network, etc., [35]. Thus, to create a neural network with a desired input-output mapping, one has to appropriately design these different components of the network. Not surprisingly, *network synthesis* of this sort is an extremely difficult task because the different components of the network and their interactions are often very complex and hard to characterize accurately.

Much of the research on neural network synthesis has focused on algorithms that modify the weights within an otherwise fixed network architecture [22]. This essentially entails a search for a setting of the weights that endows the network with the desired input-output behavior. For example, in a network used in classification applications we desire weights that will allow the network to correctly classify all (or most of) the samples in the training set. Since this is fundamentally an *optimization* problem, a variety of optimization methods (gradient-descent, simulated annealing, etc.) can be used to determine the weights. Most of the popular learning algorithms use some form of error-guided search (e.g., changing each modifiable parameter in the direction of the negative gradient of a suitably defined error measure with respect to the parameter of interest). A number of such learning algorithms have been developed, both for *supervised* learning (where the desired outputs of the network are specified by an external teacher) and *unsupervised* learning (where the network learns to classify, categorize, or self-organize without external supervi-

sion). For details regarding these learning paradigms, the reader is referred to [22, 43, 30, 74].

Although a number of techniques have been developed to adapt the weights within a given neural network, the design of the neural architecture still poses a few problems. Conventional approaches often rely on human experience, intuition, and rules-of-thumb to determine the network architectures. In recent years, a number of *constructive* and *destructive* algorithms have been developed, that aid in the design of neural network architectures. While constructive algorithms incrementally build network architectures one unit (or one module) at a time [34, 37, 63, 89], destructive algorithms allow arbitrary networks to be pruned one unit (or one module) at a time. Thus, not only do these approaches synthesize network architectures, but also entertain the possibility of discovering *compact* (or minimal) networks. A number of such constructive and destructive learning algorithms have been developed, each offering its own characteristic bias.

In addition to these approaches, *evolutionary algorithms* (to be described shortly) have also been used to search the space of neural architectures for near-optimal designs (see [5] for a bibliography). This evolutionary approach to the design of neural network architectures forms the core of this compilation.

1.5 Evolutionary Algorithms

Evolutionary algorithms, loosely inspired by biological evolutionary processes, have gained considerable popularity as tools for searching vast, complex, deceptive, and multimodal search spaces [33, 25, 57]. Following the metaphor of biological evolution, these algorithms work with populations of individuals, where each individual represents a point in the space being searched. Viewed as a search for a *solution* to a problem, each individual then represents (or encodes) a solution to the problem on hand. As with biological evolutionary systems, each individual is characterized by a *genetic representation* or *genetic encoding*, which typically consists of an arrangement of *genes* (usually in a string form). These genes take on values called *alleles*, from a suitably defined domain of values. This genetic representation is referred to as the *genotype* in biology. The actual individual, in our case a solution, is referred to as the *phenotype*. As in biological evolutionary processes, phenotypes in artificial evolution are produced from genotypes through a process of *decoding* and *development*, as shown in Figure 1.2. Thus, while a human being

corresponds to a phenotype, his/her chromosomes correspond to the genotype. The processes of nurture, growth, learning, etc., then correspond to the decoding/developmental processes that transform the genotype into a phenotype.

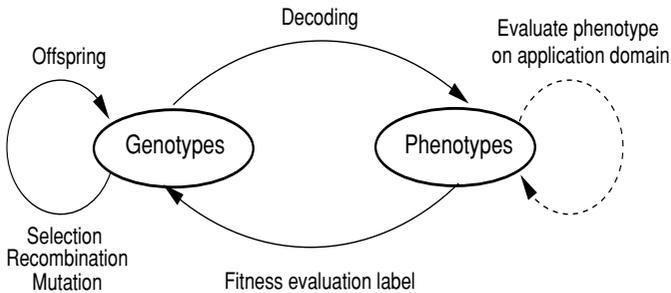


Figure 1.2

The functioning of an evolutionary algorithm.

In artificial evolution (also referred to as simulated evolution), solutions represented by the phenotypes are evaluated based on the target problem for which solutions are being sought. This evaluation of the phenotypes assigns differential fitness labels to the corresponding genotypes. Processes akin to natural selection then preferentially choose genotypes of higher fitness to participate in probabilistically more numbers of matings. These matings between chosen individuals leads to offsprings that derive their genetic material from their parents via artificial genetic operators that roughly correspond to the biological operators of *recombination* and *mutation*. These artificial genetic operators are popularly referred to as *crossover* and *mutation*. The offspring genotypes are then decoded into phenotypes and the process repeats itself. Over many generations the processes of selection, crossover, and mutation, gradually lead to populations containing genotypes that correspond to high fitness phenotypes. This general procedure, perhaps with minor variations, is at the heart of most evolutionary systems.

The literature broadly distinguishes between four different classes of evolutionary approaches: *genetic algorithms*, *genetic programming*, *evolutionary programming*, and *evolutionary strategies*. While genetic algorithms typically use *binary* (or bit) strings to represent genotypes [33, 25], genetic programming evolves *programs* in some given *language* [42]. Both these paradigms

perform evolutionary search via genetic operators of crossover and mutation. Evolutionary programming, on the other hand, allows complex structures in the genotypes but only uses a mutation operator [20]. Evolution strategies are typically used for parameter optimization [78, 3]. They employ recombination and mutation, and also permit *self-learning* (or evolutionary adaptation) of strategy parameters (e.g., variance of the Gaussian mutations). In recent years, the distinctions between these different paradigms have become rather fuzzy with researchers borrowing from the strengths of different paradigms. For instance, we use complex data structures for representing genotypes and employ both recombination as well as mutation operators to perform the evolutionary search [4]. In this regard our approach may be described as a combination of evolutionary programming and genetic algorithms. For these reasons we prefer to use the generic term, *evolutionary algorithms*, to describe our approach to the use of artificial evolution.

As each population member represents a potential solution, evolutionary algorithms effectively perform a *population-based* search in solution space. Since this is equivalent to exploring multiple regions of the space in parallel, evolutionary algorithms are efficient search tools for vast spaces. In addition, the population-based nature of evolutionary search often helps it overcome problems associated with *local maxima*, making it very suitable for searching multi-modal spaces. Further, the genetic encoding and genetic operators can be chosen to be fairly generic, requiring the user to only specify the decoding function and the *fitness* or *evaluation* function. In most cases these functions can be specified using *little* domain-specific knowledge. Thus, one does not necessarily have to understand the intricacies of the problem in order to use an evolutionary approach to solve it.

Evolutionary Synthesis of Agent Programs

As demonstrated by several of the chapters that follow, evolutionary algorithms offer a promising approach to synthesis of agent programs (in the form of artificial neural networks, LISP programs, etc.) for a wide variety of tasks [68, 19, 71, 29, 70, 64, 72, 69, 88, 13, 53, 85, 41, 12, 48, 54, 59, 8, 50].

In such cases, evolutionary search operates in the space of agent programs, with each member of the population representing an agent *behavior*. By evaluating these behaviors on the target agent task and performing fitness-proportionate reproduction, evolution discovers agent programs that exhibit the desired behaviors.

1.6 Evolutionary Synthesis of Neural Systems

In an earlier section we alluded to the difficulty of synthesizing artificial neural networks that possess specific input-output mappings. Owing to the many properties of evolutionary algorithms, primarily their ability to search vast, complex, and multimodal search spaces using little domain-specific knowledge, they have found natural applications in the automatic synthesis of artificial neural networks. Several researchers have recently begun to investigate evolutionary techniques for designing such neural architectures (see [5] for a bibliography).

Probably the distinguishing feature of an evolutionary approach to network synthesis is that unlike neural network learning algorithms that typically determine weights within *a priori* fixed architectures and constructive/destructive algorithms that simply design network architectures without directly adapting the weights, evolutionary algorithms permit *co-evolution* or *co-design* of the network architecture as well as the weights. Evolutionary algorithm may be easily extended to automatically adapt other parameters such as rates of mutation [79] and learning rate [77], and even the learning algorithm [9]. In addition, by appropriately modifying the fitness function, the same evolutionary system can be used to synthesize vastly different neural networks, each satisfying different task-specific performance measures (e.g., accuracy, speed, robustness, etc.) or user-specified design constraints (e.g., compactness, numbers of units, links and layers, fan-in/fan-out constraints, power consumption, heat dissipation, area/volume when implemented in hardware, etc.). Evolutionary algorithms also allow these networks to be optimized along multiple dimensions either *implicitly* [4] or *explicitly* via the use of different multi-objective optimization approaches [21, 67, 66, 39].

An Example of Evolutionary Synthesis of Neural Networks

A number of researchers have designed evolutionary systems to synthesize neural networks for a variety of applications. Here we will present the approach adopted by Miller et al. (1989).

In their system, Miller et al., *encode* the topology of an N unit neural network by a *connectivity constraint matrix* C , of dimension $N \times (N + 1)$, as shown in Figure 1.3. Here, the first N columns specify the constraints on the connections between the N units, and the final column codes for the connection that corresponds to the *threshold or bias* of each unit. Each entry C_{ij} , of the connectivity constraint matrix indicates the nature of the constraint

on the connection from unit j to unit i (or the constraint on the threshold bias of unit i if $j = N + 1$). While $C_{ij} = 0$ indicates the *absence* of a *trainable* connection between units j and i , a value of 1 signals the *presence* of such a trainable link. The rows of the matrix are concatenated to yield a bit-string of length $N \times (N + 1)$. This is the genotype in their evolutionary system.

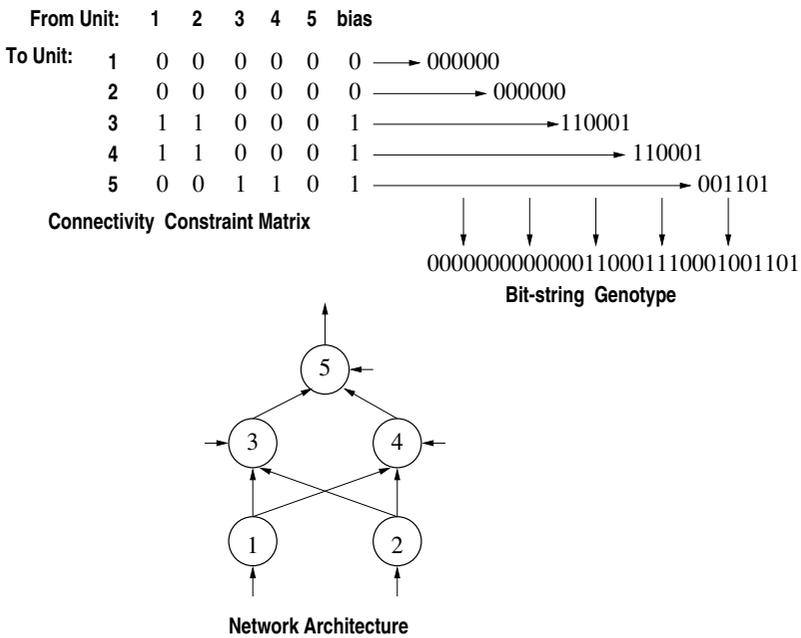


Figure 1.3
An example of an evolutionary approach to the synthesis of neural networks.

The fitness of the genotype is evaluated as follows. First, the genotype is *decoded* into the corresponding neural network (or the phenotype). This decoded network has connections (or weights) between units that have a 1 in the corresponding position in the connectivity constraint matrix (or the genotype), as explained earlier. Even though feedback connections can be specified in the genotype, they are ignored by the decoding mechanism. The system thus evolves purely feed-forward networks. Next, all the connections in the network are set to small random values and trained for a fixed number

of epochs on a given set of training examples, using the *back-propagation* algorithm [75]. The *total sum squared error* (\mathcal{E}) of the network, at the end of the training phase, is used as the *fitness* measure, with *low* values of \mathcal{E} corresponding to better performance and hence a higher fitness label for the corresponding genotype.

The system maintains a population of such genotypes (bit-strings), and uses a *fitness-proportionate* selection scheme for choosing parents for reproduction. The genetic operator *crossover* swaps rows between parents while *mutation* randomly flips bits in the genotype with some low, pre-specified probability. The researchers used this evolutionary system to design neural networks for the *XOR*, *four-quadrant* and *pattern-copying* problems [55].

1.7 Genetic Representations of Neural Architectures

Central to any evolutionary design system is the choice of the *genetic representation*, i.e., the encoding of genotypes and the mechanism for decoding them into phenotypes. The choice of the representation is critical since it not only dictates the kinds of phenotypes (and hence solutions) that can be generated by the system, but also determines the amount of *resources* (e.g., time, space, etc.) expended in this effort. Further, the genetic operators for the system are also defined based largely on the representation chosen. These factors contribute directly to the *efficiency* (e.g., time, space, etc.) and the *efficacy* (e.g., quality of solution found, etc.) of the evolutionary search procedure [66, 65]. Thus, a careful characterization of the properties of genetic representations as they relate to the performance of evolutionary systems, is a necessary and useful venture.

Some authors have attempted to characterize *properties* of genetic representations of neural architectures [11, 28]. However, most such characterizations have been restricted to a specification of the properties of the encoding scheme without considering in detail the associated decoding process. This is an important oversight because the decoding process not only determines the phenotypes that emerge from a given genotype, but also critically influences the resources required in the bargain. For instance, a cryptic encoding scheme might be compact from a storage perspective but might entail a decoding process that is rather involved (both in terms of time and storage space). If we were blind to the effects of the decoding process, we might be enamoured by this encoding scheme and use it as our genetic representation. This would have

severe consequences on the performance of the evolutionary system.

For these reasons it is imperative that we consider the encoding and the decoding process as a closely related pair while characterizing different properties of genetic representations. Given this motivation, we have identified and precisely defined a number of key properties of genetic representations of neural architectures, taking both the encoding and the decoding processes into account [6]. However, it must be pointed out that this is only a preliminary characterization and we expect these definitions to get more refined as we examine a large variety of evolutionary systems more closely. The following section describes these properties.

Properties of Genetic Representations of Neural Architectures

In order to develop formal descriptions of properties of genetic representations, we need the following definitions.

- $\mathcal{G}_{\mathcal{R}}$ is the space of genotypes representable in the chosen genetic representation scheme \mathcal{R} . $\mathcal{G}_{\mathcal{R}}$ may be explicitly enumerated or implicitly specified using a grammar Γ whose language $L(\Gamma) = \mathcal{G}_{\mathcal{R}}$.
- $p = \mathcal{D}(g, \mathcal{E}_D)$, where \mathcal{D} is the decoding function that produces the phenotype p corresponding to the genotype g possibly under the influence of the environment \mathcal{E}_D (e.g., the environment may set the parameters of the decoding function). A value of λ for \mathcal{E}_D denotes the lack of direct interaction between the decoding process and the environment. It should be borne in mind that \mathcal{D} may be *stochastic*, with an underlying probability distribution over the space of phenotypes.
- $p_2 = \mathcal{L}(p_1, \mathcal{E}_L)$, where the learning procedure \mathcal{L} generates phenotype p_2 from phenotype p_1 under the influence of the environment \mathcal{E}_L . The environment may provide the training examples, set the free parameters (e.g., the learning rate used by the algorithm) etc. We will use $\mathcal{L} = \lambda$ to denote the absence of any form of learning in the system. In the following discussion we will use the term *decoding function* to refer to both \mathcal{D} and \mathcal{L} . This slight abuse of notation allows the following properties to apply to genetic representations in general, even though they are presented in the context of evolutionary design of neural architectures.
- $\mathcal{P}_{\mathcal{R}}$ is the space of all phenotypes that can be constructed (in principle) given a particular genetic representation scheme \mathcal{R} . Mathematically, $\mathcal{P}_{\mathcal{R}} = \{p/\exists g \in \mathcal{G}_{\mathcal{R}}[(p_1 = \mathcal{D}(g, \mathcal{E}_D)) \wedge (p = \mathcal{L}(p_1, \mathcal{E}_L))]\}$

- \mathcal{S} is the set of *solution networks*, i.e., neural architectures or phenotypes that satisfy the desired performance criterion (as measured by the fitness function π) in a given environment \mathcal{E}_π . If an evolutionary system with a particular representation \mathcal{R} is to successfully find solutions (even in principle), $\mathcal{S} \subseteq \mathcal{P}_\mathcal{R}$, or, at the very least, $\mathcal{S} \cap \mathcal{P}_\mathcal{R} \neq \emptyset$. In other words, there must be at least one solution network that can be constructed given the chosen representation \mathcal{R} .
- \mathcal{A} is the set of *acceptable* or *valid* neural architectures. For instance, a network may be deemed invalid or unacceptable if it does not have any paths from the inputs to the outputs. In general, \mathcal{A} may be different from $\mathcal{P}_\mathcal{R}$. However, it must be the case that $\mathcal{A} \cap \mathcal{S} \neq \emptyset$ if a particular evolutionary system is to be useful in practice.

We now identify some properties of genetic representations of neural architectures. Unless otherwise specified, we will assume the following definitions are with respect to an *a priori* fixed choice of \mathcal{E}_D , \mathcal{L} , and \mathcal{E}_L .

1. **Completeness:** A representation \mathcal{R} is *complete* if every neural architecture in the solution set can be constructed (in principle) by the system. Formally, the following two statements are equivalent definitions of completeness.

- $(\forall s \in \mathcal{S})(\exists g \in \mathcal{G}_\mathcal{R})[(p_1 = \mathcal{D}(g, \mathcal{E}_D)) \wedge (s = \mathcal{L}(p_1, \mathcal{E}_L))]$
- $\mathcal{S} \subseteq \mathcal{P}_\mathcal{R}$

Thus, completeness demands that the representation be capable of producing all possible solutions to the problem. Often, this may be hard to satisfy and one may have to choose between *partially complete* representations. In such cases, another figure of merit called *solution density*, denoted by $\frac{|\mathcal{S} \cap \mathcal{P}_\mathcal{R}|}{|\mathcal{P}_\mathcal{R}|}$, might be useful. One would then choose representations that correspond to higher solution densities, since this implies a higher likelihood of finding solutions. It should be noted that if the solution density is very high, even a *random search* procedure will yield good solutions and one may not have much use for an evolutionary approach.

2. **Closure:** A representation \mathcal{R} is *completely closed* if every genotype decodes to an acceptable phenotype. The following two assertions are both equivalent definitions of closure.

- $(\forall g \in \mathcal{G}_\mathcal{R})[(p_1 = \mathcal{D}(g, \mathcal{E}_D)) \wedge (\mathcal{L}(p_1, \mathcal{E}_L) \in \mathcal{A})]$
- $\mathcal{P}_\mathcal{R} \subseteq \mathcal{A}$

A representation that is not closed can be transformed into a closed system by *constraining* the decoding function, thereby preventing it from generating invalid phenotypes. Additionally, if the genetic operators are designed to have

the property of closure, then one can envision *constrained closure* wherein all genotypes do not correspond to acceptable phenotypes, however, closure is guaranteed since the system *never* generates the invalid genotypes. Closure has bearings on the *efficiency* of the evolutionary procedure as it determines the amount of effort (space, time, etc.) wasted in generating unacceptable phenotypes.

3. **Compactness:** Suppose two genotypes g_1 and g_2 both decode to the same phenotype p , then g_1 is said to be more *compact* than g_2 if g_1 occupies less *space* than g_2 :

$$\bullet (p_1 = \mathcal{D}(g_1, \mathcal{E}_D)) \wedge (p = \mathcal{L}(p_1, \mathcal{E}_L)) \wedge (p_2 = \mathcal{D}(g_2, \mathcal{E}_D)) \wedge (p = \mathcal{L}(p_2, \mathcal{E}_L)) \wedge |g_1| < |g_2|$$

where $|g|$ denotes the size of storage for genotype g .

This definition corresponds to *topological-compactness* defined by

Gruau (1994). His definition of *functional-compactness* – which compares the genotype sizes of two phenotypes that exhibit the same behavior, can be expressed in our framework (for solution networks) as

$$\bullet (p_1 = \mathcal{D}(g_1, \mathcal{E}_D)) \wedge (\mathcal{L}(p_1, \mathcal{E}_L) \in \mathcal{S}) \wedge (p_2 = \mathcal{D}(g_2, \mathcal{E}_D)) \wedge (\mathcal{L}(p_2, \mathcal{E}_L) \in \mathcal{S}) \wedge |g_1| < |g_2|$$

Compactness is a useful property as it allows us to choose genetic representations that use space more efficiently. However, compact and cryptic representations often require considerable decoding effort to produce the corresponding phenotype. This is the classic space-time tradeoff inherent in algorithm design. Hence, the benefits offered by a compact representation must be evaluated in light of the increased decoding effort before one representation can be declared preferable over another.

4. **Scalability:** Several notions of scalability are of interest. For the time being we will restrict our attention to the *change* in the *size* of the phenotype, measured in terms of the numbers of units, connections, or modules. This change in the size of the phenotype manifests itself as a change in the *size* of the encoding (space needed to store the genotype), and a corresponding change in *decoding time*. We can characterize the relationship in terms of the *asymptotic order of growth* notation commonly used in analyzing computer algorithms — $O(\cdot)$.

For instance, let $n_{N,C} \in \mathcal{A}$ be a network (phenotype) with N units and C connections (the actual connectivity pattern does not really matter in this example). We say that the representation is $O(K)$ -*size-scalable with respect to units* if the addition of *one* unit to the phenotype $n_{N,C}$ requires an increase

in the size of the corresponding genotype by $O(K)$, where K is some function of N and C . For instance, if a given representation is $O(N^2)$ size-scalable with respect to units,

then the addition of one unit to the phenotype increases the size of the genotype by $O(N^2)$. Size-scalability of encodings with respect to connections, modules, etc., can be similarly defined.

The representation is said to be $O(K)$ -time-scalable with respect to units if the time taken for decoding the genotype for $n_{N+1,C}$ exceeds that used for $n_{N,C}$ by no more than $O(K)$. Similarly, time-scalability with respect to the number of connections, modules, etc., can also be defined.

Scalability is central to understanding the space-time consequences of using a particular genetic representation scheme in different contexts. In conjunction with completeness and compactness, scalability can be effectively used to characterize genetic representations.

5. Multiplicity: A representation \mathcal{R} is said to exhibit *genotypic multiplicity* if multiple genotypes decode to an identical phenotype. In other words, the decoding function is a many to one mapping from the space of genotypes to the corresponding phenotypic space.

$$\bullet (\exists n \in \mathcal{P}_{\mathcal{R}}) (|\{g \in \mathcal{G}_{\mathcal{R}} / (p = \mathcal{D}(g, \mathcal{E}_D)) \wedge (n = \mathcal{L}(p, \mathcal{E}_L))\}| > 1)$$

Genotypic multiplicity may result from a variety of sources including the encoding and decoding mechanisms. If a genetic representation has the property of genotypic multiplicity, it is possible that multiple genotypes decode to the same *solution* phenotype. In such cases, if the density of solutions is also high, then a large fraction of the genotypic space corresponds to potential solutions. This will make the evolutionary search procedure very effective.

A representation \mathcal{R} is said to exhibit *phenotypic multiplicity* if different instances of the same genotype can decode to different phenotypes. In other words, the decoding function is a one to many mapping of genotypes into phenotypes.

$$\bullet (\exists g_1, g_2 \in \mathcal{G}_{\mathcal{R}})[(p_1 = \mathcal{D}(g_1, \mathcal{E}_D)) \wedge (n_1 = \mathcal{L}(p_1, \mathcal{E}_L))] \wedge (p_2 = \mathcal{D}(g_2, \mathcal{E}_D)) \wedge (n_2 = \mathcal{L}(p_2, \mathcal{E}_L)) \wedge (g_1 = g_2) \wedge (n_1 \neq n_2)]$$

Phenotypic multiplicity may result from several factors including the effects of the environment, learning, or stochastic aspects of the decoding process. If the density of solutions is low, then the property of phenotypic multiplicity increases the possibility of decoding to a solution phenotype.

6. Ontogenetic Plasticity: A representation \mathcal{R} exhibits *ontogenetic plasticity* if the determination of the phenotype corresponding to a given geno-

type is influenced by the environment. This may happen as a result of either environment-sensitive developmental processes (in which case $\mathcal{E}_D \neq \lambda$), or learning processes (in which case $\mathcal{L} \neq \lambda$).

Ontogenetic plasticity is a useful property for constraining or modifying the decoding process based on the dictates of the application domain. For instance, if one is evolving networks for a pattern classification problem, the search for a solution network can be dramatically enhanced by utilizing a *supervised* learning algorithm for training individual phenotypes in the population. However, if such training examples are not available to permit supervised learning, one will have to be content with a purely evolutionary search.

7. Modularity: Gruau’s notion of *modularity* [28] is as follows: Suppose a network n_1 includes several instances of a subnetwork n_2 then the encoding (genotype) of n_1 is *modular* if it codes for n_2 only once, with instructions to copy it that would be understood by the decoding process. Modularity is closely tied to the existence of *organized structure* or regularity in the phenotype that can be concisely expressed in the genotype in a form that can be used by the decoding process. Other notions of modularity dealing with *functional* modules, *recursively-defined* modules etc., are also worth exploring. It can be observed that the property of modularity automatically results in more compact genetic encodings and a potential *lack of redundancy* (described below). In modular representations any change in the genotypic encoding of a module, either due to genetic influences or errors, affects all instances of the module in the phenotype. Non-modular representations, on the other hand, are resistive to such complete alterations. It is hard to decide *a priori* which scenario is better, since modular representations benefit from benign changes while non-modular representations are more robust to deleterious ones.

8. Redundancy: Redundancy can manifest itself at various levels and in different forms in an evolutionary system. Redundancy often contributes to the robustness of the system in the face of failure of components or processes. For instance, if the reproduction and/or decoding processes are error-prone, an evolutionary system can benefit from *genotypic redundancy* (e.g., the genotype contains redundant genes) or *decoding redundancy* (e.g., the decoding process reads the genotype more than once). If the phenotype is prone to failure of components (e.g., units, connections, sub-networks, etc.), the system can benefit from *phenotypic redundancy*. Phenotypic redundancy can be either *topological* (e.g., multiple identical units, connections, etc.) or *functional* (e.g., dissimilar units, connections, etc., that somehow impart the same function).

It is worth noting that genotypic redundancy does not necessarily imply phenotypic redundancy and vice versa (depending on the nature of the decoding process). This simply reiterates the importance of examining the entire representation (encoding as well as decoding) when defining properties of evolutionary systems. Also note that there are many ways to realize both genotypic as well as phenotypic redundancy: by replication of identical components (structural redundancy) or by replication of functionally identical units, or by building in modules or processes that can dynamically restructure themselves when faced with failure of components etc. [84].

9. Complexity: Complexity is perhaps one of the most important properties of any evolutionary system. However, it is rather difficult to characterize satisfactorily using any single definition. It is probably best to think of complexity using several different notions including: *structural complexity* of genotypes, *decoding complexity*, *computational (space/time) complexity* of each of the components of the system (including decoding of genotypes, fitness evaluation, reproduction, etc.), and perhaps even other measures inspired by *information theory* (e.g., entropy, Kolmogorov complexity, etc.) [49].

Although it is clear that one would like to use a genetic representation that leads to lower system complexities, the many interacting elements of the evolutionary system, genetic representations and their properties, and the existence of many different kinds of complexities, make it hard to arrive at *one* scalar measure that would satisfy all. Needless to say, characterization of complexity remains a subjective measure of the user's preferences and the dictates of the application problem.

This list of properties, although by no means complete, is nevertheless relevant in an operationally useful characterization of evolutionary systems in general, and the design of neural architectures in particular. Table 1.1 illustrates a characterization of the evolutionary system proposed by Miller et al., that was described in Section 1.6.

1.8 Summary

In this chapter we have introduced the evolutionary approach to the synthesis of agent programs in general, and artificial neural networks in particular. That evolution is a powerful, and more importantly, an aptly suited design approach for this undertaking, will be amply demonstrated in the chapters to follow.

Since the efficiency and efficacy of any evolutionary design system is crit-

Table 1.1

Properties of the genetic representation used by Miller et al.

Property	Satisfied?	Comments
Completeness	✓	With respect to feed-forward networks.
Closure	×	Invalid networks can result.
Topological Compactness	✓	Determined by back-propagation.
Functional Compactness	✓	Also possible.
Space Scalability	✓	$O(N)$ with respect to units.
Time Scalability	✓	$O(N)$ with respect to units.
Genotypic Multiplicity	×	No genotypic multiplicity.
Phenotypic Multiplicity	✓	Dictated by back-propagation.
Ontogenetic Plasticity	✓	Back-propagation used for training.
Modularity	×	Genotype only specifies connections.
Genotypic/Decoding Redundancy	×	One gene for each connection.
Phenotypic Redundancy	✓	Units and modules, but not connections.
Space Complexity	✓	Dictated by genotype size.
Time Complexity	✓	Dictated by GA and back-propagation.

ically governed by the encoding mechanism chosen for specifying the genotypes and the decoding mechanism for transforming them into phenotypes, extreme care must be taken to ensure that these two mechanisms are designed with the application problem in mind. To aid this process, in this chapter, we identified and formalized a number of properties of such genetic representations. To the extent possible, we have tried to characterize each property in precise terms. This characterization of properties of genetic representations will hopefully help in the rational choice of genetic representations for different applications.

For instance, suppose we need to design neurocontrollers for robots that have to operate in hazardous and *a priori* unknown environments. Examples of such applications include exploration of unknown terrains, nuclear waste cleanup, space exploration, etc. Since robots in such environments are required to plan and execute sequences of actions (where each action in a sequence may be dependent on previous actions performed as well as the sensory inputs), a recurrent neural network is probably needed. Further, if the system is to be used to design robots capable of functioning in different, *a priori* unknown environments, the robot controllers must have ontogenetic plasticity, i.e., the robots must be capable of learning from their experiences in the environment. The hazardous nature (e.g., in nuclear waste cleanup) or remoteness of the environment (e.g., in the case of robots used to explore distant planets) makes it desirable that the controllers operate robustly in the face of component failures etc., which calls for phenotypic redundancy of some form (e.g., duplication of

units, links, or modules of the neurocontroller). In addition, implementation technology and cost considerations might impose additional constraints on the design of the controller. For instance, hardware realization using current VLSI technology would benefit from locally connected, modular networks built from simple processors. Also extended periods of autonomous operation might require designs that are efficient in terms of power consumption, etc.

In order to design a robot controller satisfying these multiple performance constraints, one might resort to an evolutionary design approach. In such cases, a number of these constraints translate into properties that we have identified in Section 1.7. Using these, one can choose an appropriate genetic representation that can be used to evolve appropriate robot behaviors. Elsewhere we have demonstrated an evolutionary approach to the synthesis of robot behaviors for a box-pushing task, where we choose a genetic representation based on the properties we have identified in this chapter [4].

The remaining chapters in this volume address fundamental concerns and demonstrate successful applications of evolutionary search in the synthesis of intelligent agent designs and behaviors. The chapters are authored by prominent researchers, each an authority in his/her area of expertise. In this sense, this compilation presents a unique snapshot of cutting-edge research from leading researchers around the world.

References

- [1]D. H. Ackley and M. L. Littman. Interactions between learning and evolution. In *Proceedings of the Second International Conference on Artificial Life*, pages 487–509, 1991.
- [2]D. Anand and R. Zmood. *Introduction to Control Systems*. Butterworth-Heinemann, Oxford, 1995.
- [3]T. Bäck, G. Rudolph, and H.-P. Schwefel. Evolutionary programming and evolution strategies: Similarities and differences. In *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 11–22. 1993.
- [4]K. Balakrishnan. *Biologically Inspired Computational Structures and Processes for Autonomous Agents and Robots*. PhD thesis, Department Of Computer Science, Iowa State University, Ames, IA, 1998.
- [5]K. Balakrishnan and V. Honavar. Evolutionary design of neural architectures — a preliminary taxonomy and guide to literature. Technical Report CS TR 95-01, Department of Computer Science, Iowa State University, Ames, IA, 1995.
- [6]K. Balakrishnan and V. Honavar. Properties of genetic representations of neural architectures. In *Proceedings of the World Congress on Neural Networks*, pages 807–813, 1995.
- [7]J. Bradshaw, editor. *Software Agents*. MIT Press, Cambridge, MA, 1997.
- [8]F. Cecconi, F. Menczer, and R. Belew. Maturation and evolution of imitative learning in artificial organisms. *Adaptive Behavior*, 4(1):179–198, 1995.
- [9]D. J. Chalmers. The evolution of learning: An experiment in genetic connectionism. In

- Proceedings of the 1990 Connectionist Models Summer School*, pages 81–90, 1990.
- [10]P. Churchland and T. Sejnowski. *The Computational Brain*. MIT Press, Cambridge, MA, 1992.
- [11]R. Collins and D. Jefferson. An artificial neural network representation for artificial organisms. In *Proceedings of the Conference on Parallel Problem Solving from Nature*, pages 259–263, 1990.
- [12]R. Collins and D. Jefferson. Antfarm: Towards simulated evolution. In *Proceedings of the Second International Conference on Artificial Life*, pages 579–601, 1991.
- [13]M. Colombetti and M. Dorigo. Learning to control an autonomous robot by distributed genetic algorithms. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, 1992.
- [14]I. Cox and G. Wilfong, editors. *Autonomous Robot Vehicles*. Springer-Verlag, New York, NY, 1990.
- [15]J. Dayhoff. *Neural Network Architectures: An Introduction*. Van Nostrand Reinhold, New York, 1990.
- [16]T. Dean, J. Allen, and Y. Aloimonos. *Artificial Intelligence - Theory and Practice*. Benjamin Cummings, Redwood City, CA, 1995.
- [17]J. Durkin. *Expert Systems – Design and Development*. Macmillan, New York, NY, 1994.
- [18]H. Everett. *Sensors for Mobile Robots: Theory and Application*. A. K. Peters Ltd, Wellesley, MA, 1995.
- [19]D. Floreano and F. Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 421–430, 1994.
- [20]D. Fogel. Asymptotic convergence properties of genetic algorithms and evolutionary programming: Analysis and experiments. *Cybernetics and Systems: An International Journal*, 25:389–407, 1994.
- [21]C. Fonseca and P. Fleming. An overview of evolutionary algorithms in multi-objective optimization. *Evolutionary Computation*, 3(1):1–16, 1995.
- [22]S. Gallant. *Neural Network Learning and Expert Systems*. MIT Press, Cambridge, MA, 1993.
- [23]C. Gallistel. *The Organization of Learning*. MIT Press, Cambridge, MA, 1990.
- [24]M. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1993.
- [25]D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, Reading, MA, 1989.
- [26]S. Goonatilake and S. Khebbal, editors. *Intelligent Hybrid Systems*. John Wiley, West Sussex, UK, 1995.
- [27]J. Grier and T. Burk. *Biology of Animal Behavior*. Mosley-Year Book, New York, NY, 2 edition, 1992.
- [28]F. Gruau. Genetic micro programming of neural networks. In K. Kinnear, editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA, 1994.
- [29]I. Harvey, P. Husband, and D. Cliff. Seeing the light: Artificial evolution, real vision. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, 1994.
- [30]M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, 1995.
- [31]S. Haykin. *Neural Networks*. Macmillan, New York, NY, 1994.
- [32]J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation*. Addison Wesley, Redwood City, CA, 1991.
- [33]J. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press,

Ann Arbor, 1975.

[34]V. Honavar. *Generative Learning Structures and Processes for Generalized Connectionist Networks*. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, WI, 1990.

[35]V. Honavar. Toward learning systems that integrate different strategies and representations. In V. Honavar and L. Uhr, editors, *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, pages 561–580. Academic Press, San Diego, CA, 1994.

[36]V. Honavar. Intelligent agents. In J. Williams and K. Sochats, editors, *Encyclopedia of Information Technology*. Marcel Dekker, New York, NY, 1998.

[37]V. Honavar and L. Uhr. Generative learning structures and processes for generalized connectionist networks. *Information Sciences*, 70:75–108, 1993.

[38]V. Honavar and L. Uhr, editors. *Artificial Intelligence and Neural Networks – Steps toward Principled Integration*. Academic Press, San Diego, CA, 1994.

[39]J. Horn and N. Nafpliotis. Multiobjective optimization using the niched pareto genetic algorithm. Illigal technical report 93005, University of Illinois, Urbana-Champaign, IL, 1993.

[40]J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann, San Mateo, CA, 1993.

[41]J. Koza. Genetic evolution and co-evolution of computer programs. In *Proceedings of the Second International Conference on Artificial Life*, pages 603–629, 1991.

[42]J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[43]S. Y. Kung. *Digital Neural Networks*. Prentice Hall, New York, NY, 1993.

[44]P. Langley. *Elements of Machine Learning*. Morgan Kauffman, San Mateo, CA, 1995.

[45]D. Levine. *Introduction to Neural and Cognitive Modeling*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1991.

[46]F. Lewis, C. Abdallah, and D. Dawson, editors. *Control of Robot Manipulators*. Macmillan, New York, NY, 1993.

[47]H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[48]M. Lewis, A. Fagg, and A. Sodium. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1992.

[49]M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, New York, NY, 1997.

[50]H. Lund, J. Hallam, and W.-P. Lee. Evolving robot morphology. In *Proceedings of IEEE Fourth International Conference on Evolutionary Computation*, 1997.

[51]N. Mackintosh. *Conditioning and Associative Learning*. Clarendon, New York, NY, 1983.

[52]D. McFarland. *Animal Behavior*. Longman Scientific and Technical, Essex, England, 1993.

[53]F. Menczer and R. Belew. Evolving sensors in environments of controlled complexity. In *Proceedings of the Fourth International Conference on Artificial Life*, 1994.

[54]O. Miglino, K. Nafasi, and C. Taylor. Selection for wandering behavior in a small robot. *Artificial Life*, 2(1):101–116, 1994.

[55]G. Miller, P. Todd, and S. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384, 1989.

[56]M. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.

[57]M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[58]T. Mitchell. *Machine Learning*. McGraw Hill, New York, NY, 1997.

[59]S. Nolfi, J. Elman, and D. Parisi. Learning and evolution in neural networks. *Adaptive*

Behavior, 3(1):5–28, 1994.

[60]H. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3), 1996.

[61]J. O’Keefe and L. Nadel. *The Hippocampus as a Cognitive Map*. Clarendon, Oxford, UK, 1978.

[62]O. Omidvar and P. van der Smagt, editors. *Neural Systems for Robotics*. Academic Press, San Diego, CA, 1997.

[63]R. Parekh. *Constructive learning: Inducing grammars and neural networks*. PhD thesis, Department of Computer Science, Iowa State University, Ames, IA, 1998.

[64]M. Patel. Concept formation: A complex adaptive approach. *Theoria*, (20):89–108, 1994.

[65]M. Patel. Situation assessment and adaptive learning: Theoretical and experimental issues. In *Proceedings of the Second International Round-Table on Abstract Intelligent Agents*, 1994.

[66]M. Patel. Constraints on task and search complexity in ga+nn models of learning and adaptive behaviour. In T. Fogarty, editor, *Evolutionary Computing 2*, pages 200–224. Springer-Verlag, Berlin, 1995.

[67]M. Patel. Heuristic constraints on search complexity for multi-modal non-optimal models. In *IEEE International Conference on Evolutionary Computation*, 1995.

[68]M. Patel, M. Colombetti, and M. Dorigo. Evolutionary learning for intelligent automation: A case study. *Journal of Intelligent Automation and Soft Computing*, 1(1):29–42, 1995.

[69]M. Patel and M. Dorigo. Adaptive learning of a robot arm. In *Selected Papers from AISB Workshop on Evolutionary Computation*, pages 180–194, 1994.

[70]M. Patel and V. Maniezzo. Nn’s and ga’s: Evolving co-operative behaviour in adaptive learning agents. In *IEEE World Congress on Computational Intelligence*, 1994.

[71]C. Reynolds. Evolution of corridor following behavior in a noisy world. In *From Animals to Animals 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, 1994.

[72]C. Reynolds. Evolution of obstacle avoidance behavior: Using noise to promote robust solutions. In K. Kinnear, editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA, 1994.

[73]E. Rich and K. Knight. *Artificial Intelligence*. McGraw Hill, New York, NY, 1991.

[74]B. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, New York, NY, 1996.

[75]D. E. Rumelhart and J. L. McClelland, editors. *Parallel Distributed Processing, Vol I-II*. MIT Press, Cambridge, MA, 1986.

[76]S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[77]R. Salomon. Improved convergence rate of back-propagation with dynamic adaptation of the learning rate. In *Proceedings of the First International Conference on Parallel Problem Solving from Nature*, pages 269–273, 1991.

[78]H.-P. Schwefel, editor. *Numerical Optimization of Computer Models*. John Wiley, Chichester, UK, 1981.

[79]H.-P. Schwefel. Collective phenomena in evolutionary systems. In *Proceedings of 31st Annual Meeting of the International Society for General System Research*, pages 1025–1033, 1987.

[80]R. Sun and L. Bookman, editors. *Computational Architectures Integrating Symbolic and Neural Processes*. Kluwer Academic, New York, NY, 1994.

[81]S. L. Tanimoto. *Elements of Artificial Intelligence Using Common Lisp*. Computer Science Press, New York, NY, 1995.

[82]E. Tolman. Cognitive maps in rats and men. *Psychological Review*, 55:189–208, 1948.

[83]L. Uhr and V. Honavar. Introduction. In V. Honavar and L. Uhr, editors, *Artificial*

Intelligence and Neural Networks: Steps Toward Principled Integration. Academic Press, San Diego, CA, 1994.

[84]J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, Princeton, NJ, 1956.

[85]J. Walker. Evolution of simple virtual robots using genetic algorithms. Master's thesis, Department of Mechanical Engineering, Iowa State University, Ames, IA, 1995.

[86]P. Winston. *Artificial Intelligence*. Addison Wesley, New York, NY, 1992.

[87]M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[88]B. Yamauchi and R. Beer. Integrating reactive, sequential, and learning behavior using dynamical neural networks. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 382–391, 1994.

[89]J. Yang, R. Parekh, and V. Honavar. DistAl: An inter-pattern distance-based constructive learning algorithm. In *Proceedings of the International Joint Conference on Neural Networks*, Anchorage, Alaska, 1998. To appear.

[90]A. Zalzala and A. Morris, editors. *Neural Networks for Robotic Control: Theory and Applications*. Ellis Horwood, New York, NY, 1996.