Chapter 17
Machine Learning

Xin Yao and Yong Liu

17.1 Introduction

Machine learning is a very active sub-field of artificial intelligence concerned with the development of computational models of learning. Machine learning is inspired by the work in several disciplines: cognitive sciences, computer science, statistics, computational complexity, information theory, control theory, philosophy and biology. Simply speaking, machine learning is learning by machine. From a computational point of view, machine learning refers to the ability of a machine to improve its performance based on previous results. From a biological point of view, machine learning is the study of how to create computers that will learn from experience and modify their activity based on that learning as opposed to traditional computers whose activity will not change unless the programmer explicitly changes it.

17.1.1 Learning Models

A machine learning model has two key components: a learning element and a performance element, as shown in Fig. 17.1. The environment supplies some information to the learning element. The learning element then uses the information to modify the performance element so that it can make better decisions. The performance element selects actions to perform its task.

X. Yao (✉)
School of Computer Science, University of Birmingham, Birmingham, UK
e-mail: X.Yao@cs.bham.ac.uk

Y. Liu
University of Aizu, Aizuwakamatsu, Fukushima, Japan
e-mail: yliu@u-aizu.ac.jp

E.K. Burke and G. Kendall (eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, DOI 10.1007/978-1-4614-6940-7_17,

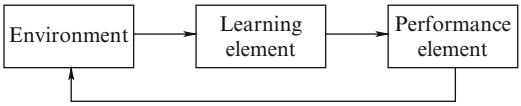

Fig. 17.1 A machine learning model

A large variety of learning elements have been proposed by researchers in the machine learning field. Based on the representation, there is symbolic and subsymbolic learning. Based on the algorithms, there are many different types of machine learning, such as decision tree, inductive logic programming, Bayesian learning, artificial neural networks, evolutionary learning and reinforcement learning. Based on the feedback available, there are three different types of machine learning: supervised, unsupervised and reinforcement learning.

The problem of supervised learning involves learning a function from a set of input–output examples. The general supervised learning model consists of two components:

1. A probability space $(\mathcal{E}, Pr)$ in which we associate each elementary event with two random variables, the input pattern $\mathbf{x}$ and the desired output y, where $\mathcal{E}$ is the event set, Pr is the probability distribution, $\mathbf{x} \in R^p$, and y is a scalar. The assumption that the output y is a scalar has been made merely to simplify exposition of ideas without loss of generality.
2. A learning machine, capable of implementing a set of functions $F(\mathbf{x}, \mathbf{w}), \mathbf{w} \in W$, where W is a set of, in general, real-valued parameters.

The purpose of supervised learning is to find the function $F(\mathbf{x}, \mathbf{w})$ so that the expected squared error

$$R(\mathbf{w}) = E[(F(\mathbf{x}, \mathbf{w}) - y)^2] \tag{17.1}$$

is minimized, where E represents the expectation value over the probability space $(\mathcal{E}, Pr)$.

In unsupervised learning, there is no specific output supplied. In the context of pattern classification, unsupervised learning learns to discover the statistical regularities of the patterns in the input, form internal representations for encoding features of the input, and thereby to create new classes automatically. In reinforcement learning, rather than being told what to do by a teacher, the learning of an input–output mapping is performed through continued interaction with the environment in order to minimize a scalar index of performance.

The environment can be either fully observable or partially observable. In the first case, the machine can observe the effects of its action and hence can use supervised learning methods to learn to predict them. In the second case, the immediate effects might be invisible so that reinforcement learning or unsupervised learning should be adopted.

17.1.2 Learning Tasks and Issues in Machine Learning

Machine learning can be applied to tasks in many domains. This section presents some important learning tasks and issues in machine learning.

17.1.2.1 Classification

A classification task in machine learning is to take each instance and assign it to a particular class. For example, in an optical character recognition task, the machine is required to scan an image of a character and output its classification. In the English language recognition, the task involves learning the classification of the digits $0 \ldots 9$ and the characters $A \ldots Z$.

17.1.2.2 Regression, Interpolation and Density Estimation

In regression, the aim is to learn some functional description of data in order to predict values for new input. An example of learning a regression function is predicting the future value of a share index in the stock market. In interpolation, the function for certain ranges of input is known. The task is to decide the function for intermediate ranges of input. In density estimation, the task is to estimate the density or probability that a member of a certain category will be found to have particular features.

17.1.2.3 Learning a Sequence of Actions

In robot learning and chess play learning, the task is to find the best strategies that can choose the optimal actions. In an example of robot navigation, a robot is assigned a task to track a colored object within a limited number of actions while avoiding obstacles and walls in an environment. There are obstacles of different shapes in the environment enclosed by the walls. To perform its task, the robot must learn the basic behavior of obstacle avoidance and moving to the target. It must also learn to co-ordinate the behavior of obstacle avoidance and the behavior of moving to the target to avoid becoming stuck due to repetition of an identical sensor–motion sequence. In chess playing, machine must decide an action based on the state of the board to move a piece in which the action will maximize its chance of winning the game.

17.1.2.4 Data Mining

The problem of data mining is of searching for interesting patterns and important regularities in large databases. Many learning methods have been developed

for determining general descriptions of concepts from examples in the form of relational data tables. Machine learning plays an important role in discovering and presenting potentially useful information from data in a form which is easily comprehensible to humans.

17.1.2.5 Issues in Machine Learning

There are many issues that need to be solved in machine learning. For example, which learning algorithm performs best for a particular learning task and representation? How many training samples are sufficient? How fast can the learning algorithms converge? When and how can prior knowledge be used in the learning process? Can a machine learn in real-time or only via offline learning? How do we choose from among multiple learning models that are all consistent with the data? Among all these issues, generalization is a key issue for any learning system. There are often two phases to design a learning system. The first phase is learning. The second phase is a generalization test. The term generalization is borrowed from psychology. In neural network learning, a model is said to generalize well when it can produce correct input–output mapping for unseen test data that have not been used in the learning phase.

17.1.3 Organization of the Chapter

The reminder of this chapter is organized as follows. Section 17.2 introduces a number of learning algorithms in order to give a breadth of coverage of machine learning. Section 17.3 addresses evolution and learning. Three levels of evolution can be introduced in neural network learning: the evolution of weight, the evolution of architectures and the evolution of learning rules. Section 17.5 points out some promising areas in machine learning. Section 17.6 provides a guideline for implementing machine learning algorithms. Section 17.7 concludes with a summary of the chapter and a few remarks.

17.2 Overview of Learning Algorithms

This section explores the basic ideas and the principles of a number of learning algorithms that are used for real-world applications.

17.2.1 Learning Decision Trees

The task of inductive learning is to find a function h that approximates f given a collection of examples of f. The function h is called a hypothesis. An example is

a pair $(x, f(x))$, where x is the input, and $f(x)$ is the output of the function applied to x. In decision-tree learning, hypotheses are represented by decision trees.

A decision tree is a diagram representing a classification system or a predictive system. The structure of the system is a tree generated based on a sequence of simple questions. The answers to these questions trace a path down the tree. As a result, a decision tree is a collection of hierarchical rules that segment the data into groups, where a decision is made for each group. The hierarchy is called a tree, and each segment is called a node. The original segment that contains the entire data set is referred to as the root node of the tree. A node with all of its successors forms a branch of the rode. The terminal nodes are called leaves that return a decision, i.e. the predicted output value for the input. The output value can be either discrete or continuous. A classification tree is used to learn a discrete-valued function, while a regression tree is used to learn a continuous function. Most decision learning algorithms are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees.

A very effective decision learning algorithm, called ID3, was developed by Quinlan (1986). In ID3, classification trees are built by starting with the set of examples and an empty tree. An attribute test is chosen for the root of the tree, and examples are partitioned into disjoint subsets depending on the outcome of the test. The learning is then applied recursively to each of these disjoint subsets. The learning process stops when all the examples within a subset belong to the same class. At this learning stage, a leaf node is created and labeled with the class.

The method used to choose the attribute test is designed to minimize the depth of the final tree. The idea is to select the attribute that can lead to an exact classification of examples as far as possible. In ID3, a statistical property, called information gain, was introduced to measure how well a given attribute separates the examples according to their target classification.

For decision-tree learning, a learned classification tree has to predict what the correct classification is for a given example. Given a training set S, containing p positive examples and n negative examples, the entropy of S to this Boolean classification is

$$E(S) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}. \tag{17.2}$$

In information theory, the entropy of S gives an estimate of the information contained in a correct answer before any of the attributes have been tested. Information theory measures information content in bits. After a test on a single attribute A, attribute A divides the training set S into subsets S_i, $i = 1, \ldots, v$, where A can have v distinct values. The information gain $G(S,A)$ of an attribute A, relative to a training set S, is defined as

$$G(S,A) = E(S) - \sum_{i=1}^{v}\frac{p_i+n_i}{p+n}E(S_i) \tag{17.3}$$

where each subset S_i has p_i positive examples and n_i negative examples. The second term in (17.3) is the expected value of entropy after S is partitioned using attribute A.

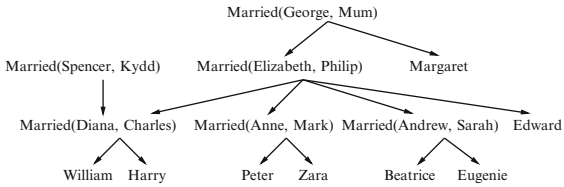

Fig. 17.2 A typical family tree

The expected entropy is the sum of entropies of each subset S_i, weighted by the fraction of examples in S_i. $G(S,A)$ is therefore the expected reduction in entropy caused by knowing the value of attribute A.

ID3 provides a simple and effective approach to decision-tree learning. However, for real-world applications, the algorithm needs to cope with problems such as a noisy data set, missing attribute values and attributes with continuous values. Dealing with these problems was studied with ID3's successor C4.5 (Quinlan 1993).

The first decision-tree learning system, called the Elementary Perceiver and Memorizer, was proposed by Feigenbaum (1961). It was studied as a cognitive-simulation model of human concept learning. The concept learning system developed by Hunt et al. (1966) used a heuristic look-ahead method to grow decision trees. ID3 (Quinlan 1986) introduced the information content as a heuristic search. The classification and regression tree system is a widely used statistical procedure for producing classification and regression (Breiman et al. 1984). Many practical issues of decision-tree induction can be found in C4.5, a decision-tree learning package by Quinlan (1993).

The advantages of decision-tree learning are its comprehensibility, fast classification and mature technology. However, by using only one attribute at each internal node, decision-tree learning can construct monothetic trees, which are limited to axis-parallel partitions of the instance space, rather than polythetic trees. Polythetic trees can use more than two attributes at each internal node, but are expensive to induce. The next section will introduce inductive logic programming, which combines inductive learning with the power of first-order representations.

17.2.2 Inductive Logic Programming

Inductive logic programming is a combination of knowledge-based inductive learning and logic programming (Russell and Norvig 2002). A general knowledge-based inductive learning is a kind of algorithm that satisfies the entailment constraint

$$\text{Background} \wedge \text{Hypothesis} \wedge \text{Descriptions} \models \text{Classifications}, \tag{17.4}$$

where *Descriptions* denote the conjunction of all the example classifications, and *Classifications* denote the conjunction of all the example classifications. Given the

Background knowledge and examples described by *Descriptions* and *Classifications*, the induction problem of knowledge-based inductive learning is to solve the entailment constraint (17.4) for the unknown *Hypothesis*.

In order to see how the background knowledge can be combined with the new hypothesis to explain examples, consider a problem of learning family relationships from examples in an extended family tree given in Fig. 17.2 (Russell and Norvig 2002).

The descriptions will be in the terms of Mother, Father, and Married relations and Male and Female properties, such as Father(Philip, Charles), Mother(Mum, Margaret), Married(Diana, Charles), Male(Philip), Female(Beatrice). Classifications depend on the target concept being learned. For learning the target concept of Grandfather, the complete set of Classifications contains $20 \times 20 = 400$ conjuncts of the form

$$\begin{aligned} &\text{Grandparent(Mum, Charles)} \ \ \text{Grandparent(Elizabeth, Beatrice)} \ \cdots \\ &\neg\text{Grandparent(Mum, Harry)} \ \ \neg\text{Grandparent(Spencer, Peter)} \ \ \cdots . \end{aligned}$$

Without the background knowledge, inductive learning can find a possible *Hypothesis*:

$$\begin{aligned} \text{Grandparent}(x,y) \Leftrightarrow\ & [\exists\ z\ \ \text{Mother}(x,z) \wedge \text{Mother}(z,y)] \\ \vee\ & [\exists\ z\ \ \text{Mother}(x,z) \wedge \text{Father}(z,y)] \\ \vee\ & [\exists\ z\ \ \text{Father}(x,z) \wedge \text{Mother}(z,y)] \\ \vee\ & [\exists\ z\ \ \text{Father}(x,z) \wedge \text{Father}(z,y)]. \end{aligned} \tag{17.5}$$

With the help of the background knowledge represented by the sentence

$$\text{Parent}(x,y) \Leftrightarrow [\text{Mother}(x,y) \vee \text{Father}(x,y)]. \tag{17.6}$$

Hypothesis can be simply defined by

$$\text{Grandparent}(x,y) \Leftrightarrow [\exists z\ \ \text{Parent}(x,z) \wedge \text{Parent}(z,y)]. \tag{17.7}$$

By using background knowledge, we can reduce the size of hypotheses greatly.

There are two basic approaches to inductive logic programming: the top-down learning of refining a very general rule and the bottom-up learning of inverting the deductive process. A top-down approach will typically begin with a general clause and search the clause by adding literals so that only positive examples are entailed. First-order inductive learning (Quinlan 1990) is such a top-down induction algorithm.

Suppose the task is to learn a definition of *Grandfather(x,y)* predicate in the family tree shown in Fig. 17.2. Examples can be divided into positive and negative ones as in decision-tree learning: 12 positive examples are

$$\langle \text{George, Charles} \rangle, \langle \text{George, Anne} \rangle, \langle \text{George, Andrew} \rangle, \ldots$$

and 388 negative examples are

$$\langle \text{George, Spencer} \rangle, \langle \text{George, Kydd} \rangle, \langle \text{George, Elizabeth} \rangle, \ldots.$$

First-order inductive learning constructs a set of clauses that must classify the positive examples while ruling out the negative examples. First-order inductive learning starts with the initial clause with Grandfather(x, y) as the head, and an empty body

$$\Rightarrow \text{Grandfather}(x, y). \tag{17.8}$$

All examples are classified as positive by this clause. To specialize it, first-order inductive learning adds literals one at a time to the clause body. Look at two clauses constructed by such addition:

$$\text{Parent}(x, z) \Rightarrow \text{Grandfather}(x, y) \tag{17.9}$$

$$\text{Father}(x, z) \Rightarrow \text{Grandfather}(x, y). \tag{17.10}$$

Although both clauses agree with all of 12 positive examples, the first allows both fathers and mothers to be grandfathers and makes larger misclassification on negative examples. The second clause is chosen to be further specialized. By adding the single literal Parent(z,y), first-order inductive learning can find

$$\text{Father}(x, z) \wedge \text{Parent}(z, y) \Rightarrow \text{Grandfather}(x, y), \tag{17.11}$$

which successfully classifies all the examples. This example gives a simple explanation how first-order inductive learning works. In real applications, first-order inductive learning generally has to search through a large number of unsuccessful clauses before finding the correct one.

Whereas first-order inductive learning (Quinlan 1990) is a top-down approach, Cigol (logic, spelled backwards), which Muggleton and Buntine (1988) developed for inductive logic programming, worked bottom-up. Cigol incorporated a slightly incomplete version of inverse resolution and was capable of generating new predicates. A hybrid (top-down and bottom-up) approach was chosen in Progol (Muggleton 1995) that inverse entailment and had been applied to a number of practical problems. A large collection of papers on inductive logic programming can be found in Lavrač and Džeroski (1994).

Inductive logic programming provides a practical approach to the general knowledge-based inductive learning problem. Its strengths lie in its firm theoretical foundations, richer hypothesis representation language, and explicit use of background knowledge. The limitations of inductive logic programming are its weak numeric representations and large search spaces.

17.2.3 Bayesian Learning

In practice, there are cases when more than one hypothesis satisfy a given task. Because it is not certain how those hypotheses perform on unseen data, it is hard to choose the best hypothesis. Bayesian learning gives a probabilistic framework for

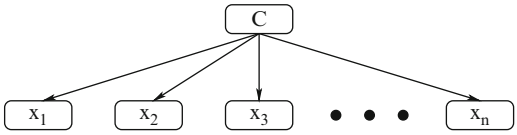

Fig. 17.3 The naive Bayes model

justification. By calculating explicit probabilities for hypotheses, Bayesian learning provides a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

Let **D** represent all the data, and H the set of all the hypotheses h_i. The probability of each hypothesis with observed **d** can be calculated by Bayes' rule:

$$P(h_i \mid \mathbf{d}) = \alpha P(\mathbf{d} \mid h_i) P(h_i), \tag{17.12}$$

where $P(h_i)$ is the prior probability, $P(\mathbf{d} \mid h_i)$ denotes the probability of observed **d** given h_i, and $P(h_i \mid \mathbf{d})$ is the posterior probability of h_i.

A practical Bayesian learning used in machine learning is the naive Bayes model shown in Fig. 17.3, where each instance **x** is described by a conjunction of attribute values $\langle x_1, x_2, \ldots, x_n \rangle$. In this model, the class variable C is the root, and the attribute values **x** are leaves.

According to (17.12), the probability of each class from a set of S is given by

$$P(C \mid x_1, x_2, \ldots, x_n) = \alpha P(x_1, x_2, \ldots, x_n \mid C) P(C). \tag{17.13}$$

In the naive Bayes model, a simplified assumption is made that the attributes are conditionally independent of each other given the class. That is, the probability of the observed conjunction $x_1, x_2, \ldots, x_n$ is just the product of probabilities for the individual attributes:

$$P(x_1, x_2, \ldots, x_n \mid C) = \prod_i P(x_i \mid C). \tag{17.14}$$

From (17.13) and (17.14), the naive Bayes model makes the prediction by choosing the most likely class:

$$C_{NB} = \text{argmax}_{C \in S} P(C) \prod_i P(x_i \mid C), \tag{17.15}$$

where C_{NB} denotes output class by the naive Bayes model.

Consider a medical diagnosis problem with three possible diagnoses (well, cold, allergy) based on three symptoms (sneeze, cough, fever). In this example, there are three attributes in which x_1 can be *sneeze* or *not sneeze*, x_2 *cough* or *not cough*, and x_3 *fever* or *not fever*, and three classes: *well*, *cold* and *allergy*. The probabilities for the three attributes and three prior class probabilities are given in Table 17.1.

Table 17.1 An example for the naive Bayes model

Diagnosis	Well	Cold	Allergy
$P(C)$	0.9	0.05	0.05
$P(\text{sneeze} \mid C)$	0.1	0.9	0.9
$P(\text{cough} \mid C)$	0.1	0.8	0.7
$P(\text{fever} \mid C)$	0.01	0.7	0.4

Given a new $\mathbf{x} = \langle$sneeze, cough, not fever$\rangle$, which class of diagnoses is it mostly like to be? First, the posterior probability $P(\text{well} \mid \text{sneeze, cough, not fever})$ of *well*, *cold* and *allergy* are calculated by the product of $P(\text{sneeze} \mid \text{well})$, $P(\text{cough} \mid \text{well})$, $P(\text{not fever} \mid \text{well})$, and $P(\text{well})$:

$$\begin{aligned} P(\text{well} \mid \text{sneeze, cough, not fever}) &= 0.1 \times 0.1 \times (1 - 0.01) \times 0.9 \\ &= 0.00891. \end{aligned} \tag{17.16}$$

Similarly we can obtain the posterior probability of *cold*:

$$P(\text{cold} \mid \text{sneeze, cough, not fever}) = 0.216 \tag{17.17}$$

and the posterior probability of *allergy*:

$$P(\text{allergy} \mid \text{sneeze, cough, not fever}) = 0.378. \tag{17.18}$$

Finally, we compare three posterior probabilities and generate output class *allergy* because the probability of allergy for the data $\mathbf{x} = \langle$sneeze, cough, not fever$\rangle$ is the largest one.

The naive Bayes model has been compared with C4.5 on 28 benchmark tasks (Domingos and Pazzani 1996). The results show that the naive Bayes model performs surprisingly well in a wide range of applications. Except for a few domains where the naive Bayes model performs poorly, it is comparable to or better than C4.5.

This section just uses the naive Bayes model to introduce the idea of Bayesian learning. Heckerman (1998) gives an excellent introduction on general learning with Bayesian networks. Bayesian learning has had successful applications in pattern recognition and information retrieval. Algorithms based on Bayesian learning won the 1997 and 2001 KDD Cup data mining competitions (Elkan 1997; Cheng et al. 2002). Experimental comparisons between Bayesian learning, decision-tree learning and other algorithms have been made on a wide range of applications (Michie et al. 1994).

17.2.4 Reinforcement Learning

Reinforcement learning concerns learning how to map situations to actions so as to maximize a numerical reward signal (Sutton and Barto 1998). Unlike supervised learning, the machine is not told which actions to take but has to discover which

actions yield the most reward by trying them. In the most practical cases, actions may affect both the immediate reward and the next situation and thus all subsequent rewards. Trial-and-error search and delayed reward are the two most important unique characteristics of reinforcement learning.

A central and novel idea of reinforcement learning is temporal-difference learning (Sutton and Barto 1998). Temporal-difference learning is a combination of Monte Carlo ideas and dynamic programming ideas. Like Monte Carlo methods, temporal-difference learning methods can learn directly from the raw experience without a model of the environment's dynamics. Like dynamic programming methods, temporal-difference learning methods update estimates based in part on other learned estimates, without waiting for a final outcome. Temporal-difference learning works because it is possible to make local improvements. At every point in the state space, the Markov property allows actions to be chosen based only on knowledge about the current state and the states reachable by taking the actions available at that state.

Temporal-difference learning methods fall into two classes: on-policy and off-policy (Sutton and Barto 1998). One of the most important breakthroughs in reinforcement learning was the development of an off-policy temporal-difference learning control algorithm known as Q-learning. The learned action-value function $Q(s,a)$ directly approximates the optimal action-value function, independent of the policy being followed. The major steps of Q-learning are (Sutton and Barto 1998):

1. Initialize $Q(s,a)$ values arbitrarily.
2. Initialize the environment.
3. Choose action a using the policy derived from $Q(s,a)$ (e.g., ε-greedy).
4. Take action a; Observe reward r and the next state s'.
5. Update the $Q(s,a)$ as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]. \tag{17.19}$$

6. Let $s \leftarrow s'$. Go to the next step if the state s is a terminal state. Otherwise, go to Step 3.
7. Repeat Steps 2–6 for a certain number of episodes.

The Sarsa learning algorithm is an on-policy temporal-difference learning method in which the action-value function Q is updated after every transition from a nonterminal state. The major steps of Sarsa learning are (Sutton and Barto 1998):

1. Initialize $Q(s,a)$ values arbitrarily.
2. Initialize the environment.
3. Choose action a using the policy derived from $Q(s,a)$ (e.g., ε-greedy).
4. Take action a; Observe reward r and the next state s'; Choose the next action a' using the policy derived from Q (e.g. ε-greedy).
5. Update the $Q(s,a)$ as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]. \tag{17.20}$$

6. Let $s \leftarrow s'$ and $a \leftarrow a'$. Go to the next step if the state s is a terminal state. Otherwise, go to Step 3.
7. Repeat Steps 2–6 for a certain number of episodes.

Sutton and Barto assessed a Sarsa learning example (Sutton and Barto 1998). The results showed that the online performance of Q-learning is worse than that of Sarsa learning.

The strengths of reinforcement learning come from its firm theoretical foundation, its ability to solve broad tasks, and its easy usage of background knowledge. Work in reinforcement learning dates back to the earliest days of machine learning when Turing proposed the reinforcement learning approach (Turing 1950), and Samuel developed his famous checkers learning program that contained most of the modern ideas of reinforcement learning, including temporal differencing and function approximation (Samuel 1959). Three threads contributed towards the modern field of reinforcement learning. The first thread is about learning by trial and error and had its origin in the psychology of animal learning, which led to the popularity of reinforcement learning in the early 1980s. The second thread arose from the problem of optimal control and its solution using value functions and dynamic programming. The third thread concerns temporal-difference methods. The survey by Kaelbling et al. (1996) provides a good starting point in the literature. The text *Reinforcement Learning: An Introduction* by Sutton and Barto, two of the field's pioneers, shows architectures and algorithms of reinforcement learning in the context of learning, planning and acting (Sutton and Barto 1998).

17.2.5 Neural Networks

Artificial neural networks, commonly referred to as neural networks, try to simulate biological brains. However, neural networks have been simplified greatly from biological brains. A neural network is a parallel computational system consisting of many processing elements connected with each other in a certain way in order to perform a task. Neural networks have gained popularity because they are adaptive, robust, fault tolerant, noise tolerant, and massively parallel.

Among the many tasks that neural networks perform, the most important one is learning. A neural network can improve its performance via learning. Perceptron learning is one of the earliest learnings developed for neural networks (Rosenblatt 1962). Perceptrons are often used to refer to feed-forward neural networks consisting of McCulloch–Pitts (MP) neurons (McCulloch and Pitts 1943):

$$y_i = \operatorname{sgn}\left(\sum_j w_{ij} x_j - \theta_i\right) \tag{17.21}$$

where the w_{ij} are called *weights* (synapses) and θ is the *threshold*. The x_j and y_i are input and output. The signum function $\operatorname{sgn}(x)$ is defined as

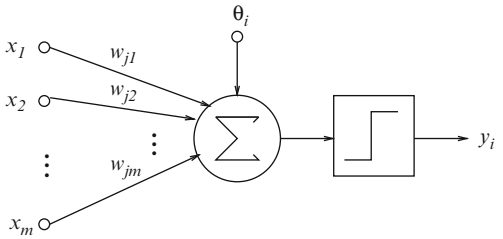

Fig. 17.4 Nonlinear model of a neuron

$$\text{sgn}(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{17.22}$$

It is also known as the threshold function or Heaviside function and is described in Fig. 17.4.

After presenting each example to a perceptron that has one layer of neurons, perceptron learning adjusts the weights until the weights converge (i.e. $\Delta w_j(t) = 0$):

$$w_j(t+1) = w_j(t) + \Delta w_j(t), \tag{17.23}$$

where

$$\Delta w_j(t) = \eta(y^p - O^p)x_j^p, \tag{17.24}$$

where η is the learning rate, x_j^p is the jth input of the pth example, y^p is the target (desired) output of the pth example, and O^p is the actual output of the pth example:

$$O^p = \text{sgn}\left(\sum_j w_j x_j^p - \theta\right). \tag{17.25}$$

The convergence theorem of perceptron learning states that if there exists a set of weights for a perceptron which solves a problem correctly, the perceptron learning rule will find them in a finite number of iterations (Rosenblatt 1962). If a problem is linearly separable, then the perceptron learning rule will find a set of weights in a finite number of iterations that solves the problem correctly. A pair of linearly separable patterns means that the patterns to be classified must be sufficiently separated from each other to ensure that the decision surface consists of a hyperplane.

The perceptron learning rule, $\Delta w_j(t) = \eta(y^p - O^p)x_j^p$, is related to the Hebbian learning rule (Hebb 1949). Hebb's postulate of learning (Hebb 1949) states that

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.

In other words, if two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

It is clear from perceptron learning that the algorithm tries to minimize the difference between the actual and desired output. We can define an error function to represent such a difference:

$$E(\mathbf{w}) = \frac{1}{2}\sum_p (y^p - O^p)^2 \tag{17.26}$$

or

$$E(\mathbf{w}) = \frac{1}{2N}\sum_p (y^p - O^p)^2, \tag{17.27}$$

where N is the number of patterns. The second error function above is called the mean square error. Learning minimizes this error by adjusting weights $\mathbf{w}$.

One advantage of introducing the error function is that it can be used for any type of transfer function, discrete or continuous. The aim of learning is to adjust $\mathbf{w}$ such that the error E is minimized, i.e. the network output is as close to the desired output as possible. There exist mathematical tools and algorithms which tell us how to minimize the error function E, such as the gradient descent algorithm which is based on partial derivatives. Given a set of training examples, $\{(x_1^p, \ldots, x_m^p; y_1^p, \ldots, y_n^p)\}_p$, the gradient descent learning algorithm can be summarized as follows:

1. Construct a neural network with m inputs and n outputs.
2. Select learning rate η and the gain parameter a.
3. Generate initial weights at random in a small range, e.g. $[-0.5, 0.5]$. Note that thresholds are regarded as weights here.
4. While the neural network has not converged do: For each training example p,
   (a) Compute O_i^p. $(O_i^p = f(u_i))$
   (b) Compute $\delta_i^p = (y_i^p - O_i^p) f'(u_i)$, where $f'(u_i) = af(u_i)(1 - f(u_i))$ if the transfer function is

$$f(u_i) = \frac{1}{1 + \exp(-au_i)}, \tag{17.28}$$

   where a is a parameter determined by the user.
   (c) Compute $\Delta w_{ij} = \eta \delta_i^p x_i^p$.
   (d) Update w_{ij} for all i, j. (All weights will be updated.)

There are two modes of the gradient descent learning algorithm. One is the sequential mode of training which is also known as online, pattern or stochastic mode. In this mode, weights are updated after the presentation of each example. The other is the batch mode of training in which weights are updated only after the complete presentation of all examples in the training set, i.e. only after each epoch.

The idea of the gradient descent learning algorithm for a single-layer neural network can be generalized to find weights for multilayer neural networks. Multilayer feedforward neural networks can solve nonlinear problems. In fact, there are mathematical theorems which show that multilayer feedforward neural networks can approximate any input–output mapping. The backpropagation algorithm can be used to train multilayer feedforward neural networks (Rumelhart et al. 1986). Its forward pass propagates the activation values from input to output. Its backward

pass propagates the errors from output to input. Backpropagation is still a gradient descent algorithm. It uses gradient information to figure out how the weights should be adjusted so that the output error can be reduced. Mathematically, backpropagation uses the chain rule to figure out how to change weights in order to minimize the error.

Consider a network with M layers $m = 1, 2, \ldots, M$ and use V_i^m to represent the output of the ith unit in the mth layer. $V_i^0 = x_i$ is the ith input. Backpropagation can be described as follows:

1. Initialize the weights to small random values.
2. Choose a pattern and apply it to the input layer so that $V_i^0 = x_i^p$.
3. Propagate the signal forwards through the network using

$$V_i^m = f(u_i^m) = f\left(\sum_j w_{ij}^m V_j^{m-1}\right) \tag{17.29}$$

   for each i and m until the final outputs V_i^M have all been calculated.
4. Compute the deltas for the output layer:

$$\delta_i^M = f'(u_i^M)(y_i^p - V_i^M). \tag{17.30}$$

5. Compute the deltas for the preceding layers by propagating errors backwards:

$$\delta_i^{m-1} = f'(u_i^{m-1}) \sum_j w_{ij}^m \delta_j^m \tag{17.31}$$

   for $m = M, M-1, \ldots, 2$.
6. Update the weights according to

$$w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ij}, \tag{17.32}$$

   where

$$\Delta w_{ij} = \eta \delta_i^m V_j^{m-1}. \tag{17.33}$$

7. Goto step 2 and repeat for the next pattern. The algorithm stops when no weight changes were made for a complete epoch or the maximum number of iterations has been reached.

The study of neural networks started with the work of MP neuron models proposed by McCulloch and Pitts (1943). The Hebbian learning rule was studied by Hebb (1949). Rosenblatt (1962) proposed perceptrons and proved the perceptron convergence theory. After Minsky and Papert's book (Minsky and Papert 1969) which showed the limitation of single-layer perceptrons, the field of neural networks was almost deserted during the 1970s. Then Hopfield published a series of papers on Hopfield networks that used the idea of an energy function to formulate a new way of understanding the computation performed by recurrent networks with symmetric synaptic connections (Hopfield 1982; Hopfield and Tank 1985). The two-volume "bible" *Parallel Distributed Processing: Explorations in*

the Microstructures of Cognition, edited by Rumelhart and McClelland attracted a great deal of attention (Rumelhart and McClelland 1986). In the mid-1980s, the field of neural networks really took off.

Neural networks have been applied to solve a wide range of problems such as pattern recognition and classification, time-series prediction, function approximation, system identification, and control. Neural network applications often include two phases. The first phase is learning. The task performed by a neural network is often represented as a set of examples. The neural network is expected to learn more general concepts from these examples. The steps involved include:

1. Select a neural network architecture, where the number of input and output nodes are determined by the task. Hidden nodes and network connectivity need to be designed mostly by trial and error.
2. Train the network using a suitable training algorithm.

The second phase is a generalization test. After the neural network is trained, it will be tested with new (never seen before) examples to see how well it generalizes.

17.2.6 Evolutionary Learning

Evolutionary learning includes many topics, such as learning classifier systems, evolutionary neural networks, evolutionary fuzzy logic systems, co-evolutionary learning and self-adaptive systems. The primary goal of evolutionary learning is the same as that of machine learning in general. Evolutionary learning can be regarded as the evolutionary computation approach to machine learning. It has been used in the framework of supervised learning, reinforcement learning and unsupervised learning, although it appears to be most promising as a reinforcement learning method. Evolutionary computation encompasses major branches, i.e. evolution strategies, evolutionary programming, genetic algorithms and genetic programming, due largely to historical reasons. At the philosophical level, they differ mainly in the level at which they simulate evolution. At the algorithmic level, they differ mainly in their representations of potential solutions and their operators used to modify the solutions. From a computational point of view, representation and search are two key issues.

Evolution strategies were first proposed by Rechenberg and Schwefel in the mid-1960s for numerical optimization. Real-valued vectors are used to represent individuals. Evolution strategies use both recombination and self-adaptive mutations. The original evolution strategy did not use populations. A population was introduced into evolution strategies later (Schwefel 1981, 1995).

Evolutionary programming was first proposed by Fogel et al. in the mid-1960s for simulating intelligence (Fogel et al. 1966). Finite-state machines were used to represent individuals, although real-valued vectors have always been used in numerical optimization. Search operators (mutations only) are applied to the phenotypic representation of individuals. There is no recombination in evolutionary programming. Tournament selection is often used in evolutionary programming.

Genetic algorithms and genetic programming are introduced in Chaps. 4 and 5 of this book, respectively. Although genetic algorithms, evolutionary programming, evolution strategies and genetic programming are different, they are all variants of population-based generate-and-test algorithms:

Generate: Mutate and/or recombine individuals in a population.
Test: Select the next generation from the parents and offsprings.

They share more similarities than differences. A better and more general term to use is evolutionary algorithms. All evolutionary algorithms have two prominent features which distinguish themselves from other search algorithms. First, they are all population-based. Secondly, there are communications and information exchange among individuals in a population. Such communications and information exchange are the result of selection and/or recombination in evolutionary algorithms. A general framework of evolutionary algorithms can be summarized as follows:

1. Generate the initial population $P(0)$ at random, and set $i \leftarrow 0$;
2. Repeat
   (a) Evaluate the fitness of each individual in $P(i)$;
   (b) Select parents from $P(i)$ based on their fitness in $P(i)$;
   (c) Generate offspring from the parents using crossover and mutation to form $P(i+1)$;
   (d) $i \leftarrow i+1$;
3. Until halting criteria are satisfied

where the search operators are also called genetic operators for genetic algorithms. They are used to generate offspring (new individuals) from parents (existing individuals).

Learning classifier systems, also known as classifier systems, are probably the oldest and best known evolutionary learning systems, although they did not work very well in their classical form. Some of the recent systems have improved this situation. Due to its historical importance, a brief introduction to the classical learning classifier systems will be introduced here.

Learning classifier systems are a particular class of message-passing, rule-based systems. They can also be regarded as a type of adaptive expert system that uses a knowledge base of production rules in a low-level syntax that can be manipulated by a genetic algorithm. In a classifier system, each low-level rule is called a classifier. A general operational cycle for the classifier system is as follows:

1. Allow the detectors (input interface) to code the current environment status and place the resulting messages on the message list.
2. Determine the set of classifiers that are matched by the current messages.
3. Resolve conflicts caused by limited message list size or contradictory actions.
4. Remove those messages which match the conditions of firing classifier from the message list.
5. Add the messages suggested by the firing messages to the list.

6. Allow the effectors (output interface) that are matched by the current message list to take actions in the environment.
7. If a payoff signal is received from the environment, assign credit to the classifiers.
8. Goto Step 1.

A genetic algorithm is used in classifier systems to discover new classifiers by crossover and mutation. The strength of a classifier updated by the credit assignment scheme is used as its fitness. A classifier's strength is based on its average usefulness in the context in which it has been tried previously. Credit assignment is a very difficult task because credit must be assigned to early-acting classifiers that set the stage for a sequence of actions leading to a favorable situation. The most well known credit assignment is the bucket brigade algorithm which uses metaphors from economics.

For a classifier called middleman, its suppliers are those classifiers that have sent messages satisfying its conditions, and its consumers are those classifiers that have conditions satisfied by its message and have won their competition in turn. When a classifier wins in competition, its bid is actually apportioned to its suppliers, increasing their strengths by the amounts apportioned to them. At the same time, because the bid is treated as a payment for the right to post a message, the strength of the winning classifier is reduced by the amount of its bid. Should the classifier bid but not win, its strength remains unchanged and its suppliers receive no payment. Winning classifiers can recoup their payments from either winning consumers or the environment payoff.

The genetic algorithm is only applied to the classifiers after certain number of operational cycles in order to approximate strengths better. There are two approaches to classifier systems; the Michigan approach and the Pitt approach. For the Michigan approach, each individual in a population is a rule. The whole population represents a complete classifier system. For the Pitt approach, each individual in a population represents a complete classifier system. The whole population includes a number of competing classifier systems.

17.3 Learning and Evolution

Learning and evolution are two fundamental forms of adaptation. There has been a great interest in combining learning and evolution with neural networks in recent years.

17.3.1 Evolutionary Neural Networks

Evolutionary neural networks refer to a special class of neural networks in which evolution is another fundamental form of adaptation in addition to learning (Yao 1991, 1993a, 1994, 1995). Evolutionary algorithms are used to perform various

tasks, such as connection weight training, architecture design, learning rule adaptation, input feature selection, connection weight initialization and rule extraction from neural networks. One distinct feature of evolutionary neural networks is their adaptability to a dynamic environment. In other words, evolutionary neural networks can adapt to an environment as well as to changes in the environment. The two forms of adaptation, i.e., evolution and learning in evolutionary neural networks, make their adaptation to a dynamic environment much more effective and efficient. In a broader sense, evolutionary neural networks can be regarded as a general framework for adaptive systems, i.e. systems that can change their architectures and learning rules appropriately without human intervention.

Evolution has been introduced into neural networks at roughly three different levels:

- Connection weights
- Architectures and
- Learning rules.

17.3.1.1 The Evolution of Connection Weights

The evolution of connection weights introduces an adaptive and global approach to training, especially in the reinforcement learning and recurrent network learning paradigm where gradient-based training algorithms often experience great difficulties.

One way to overcome gradient-descent-based training algorithms' shortcomings is to adopt evolutionary neural networks, i.e. to formulate the training process as the evolution of connection weights in the environment determined by the architecture and the learning task. Evolutionary algorithms can then be used effectively in the evolution to find a near-optimal set of connection weights globally without computing gradient information. The fitness of a neural network can be defined according to different needs. Two important factors which often appear in the fitness (or error) function are the error between target and actual outputs and the complexity of the neural network. Unlike in the gradient-descent-based case, the fitness (or error) function does not have to be differentiable or even continuous since evolutionary algorithms do not depend on gradient information. Because evolutionary algorithms can treat large, complex, nondifferentiable and multimodal spaces, which are the typical case in the real world, considerable research and application has been conducted on the evolution of connection weights.

The evolutionary approach to weight training in neural networks consists of two major phases. The first phase is to decide the representation of connection weights, i.e. whether it is to be in the form of binary strings or not. The second one is the evolutionary process simulated by an evolutionary algorithm, in which search operators such as crossover and mutation have to be decided on in conjunction with the representation scheme. Different representations and search operators can lead to

quite different training performance. A typical cycle of the evolution of connection weights is shown as follows (Yao 1999):

1. Decode each individual (genotype) in the current generation into a set of connection weights and construct a corresponding neural network with weights.
2. Evaluate each neural network by computing its total mean square error between actual and target outputs. Other error functions can also be used. The fitness of an individual is determined by the error. The higher the error, the lower the fitness. The optimal mapping from the error to the fitness is a problem dependent. A regularization term may be included in the fitness function to penalize large weights.
3. Select parents for reproduction based on their fitness.
4. Apply genetic operators, such as crossover and/or mutation, to parents to generate offspring, which form the next generation.

The evolution stops when the fitness is greater than a predefined value (i.e. the training error is smaller than a certain value) or the population has converged.

17.3.1.2 The Evolution of Architectures

The evolution of architectures enables neural networks to adapt their topologies to different tasks without human intervention and thus provides an approach to automatic neural network design as both neural network connection weights and structures can be evolved.

Architecture design is crucial in the successful application of neural networks because the architecture has significant impact on a network's information processing capabilities. Given a learning task, a neural network with only a few connections and linear nodes may not be able to perform the task at all due to its limited capability, while a neural network with a large number of connections and nonlinear nodes may overfit noise in the training data and fail to have good generalization ability.

Currently, architecture design is still very much a human expert's job. It depends heavily on expert experience and a tedious trial-and-error process. There is no systematic way to design a near-optimal architecture for a given task automatically. Design of the optimal architecture for a neural network can be formulated as a search problem in the architecture space where each point represents an architecture. Given some performance (optimality) criteria, e.g. lowest training error, lowest network complexity, etc., about architectures, the performance level of all architectures forms a discrete surface in the space. The optimal architecture design is equivalent to finding the highest point on this surface.

Similar to the evolution of connection weights, two major phases involved in the evolution of architectures are the genotype representation scheme of architectures and the evolutionary algorithm used to evolve neural network architectures. One of the key issues in encoding neural network architectures is to decide how much information about an architecture should be encoded in the chromosome. At one extreme, all the details, i.e. every connection and node of an architecture, can be specified

by the chromosome. This kind of representation scheme is called direct encoding. At the other extreme, only the most important parameters of an architecture, such as the number of hidden layers and hidden nodes in each layer, are encoded. Other details about the architecture are left to the training process to decide. This kind of representation scheme is called indirect encoding. After a representation scheme has been chosen, the evolution of architectures can progress according to the cycle as follows (Yao 1999):

1. Decode each individual in the current generation into an architecture. If the indirect encoding scheme is used, further detail of the architecture is specified by some developmental rules or a training process.
2. Train each neural network with the decoded architecture by a pre-defined learning rule (some parameters of the learning rule could be learned during training) starting from different sets of random initial weights and, if any, learning parameters.
3. Compute the fitness of each individual (encoded architecture) according to the above training result and other performance criteria such as the complexity of the architecture.
4. Select parents from the population based on their fitness.
5. Apply genetic operators to the parents and generate offspring which form the next generation.

The cycle stops when a satisfactory neural network is found.

An automatic system, EPNet (Yao and Liu 1997, 1998), based on evolutionary programming has been developed for simultaneous evolution of neural network architectures and connection weights. EPNet relies on a number of mutation operators to modify architectures and weights. Behavioral (i.e. functional) evolution, rather than genetic evolution, is emphasized in EPNet. A number of techniques were adopted to maintain the behavioral link between a parent and its offspring (Yao and Liu 1997). Figure 17.5 shows the main structure of EPNet.

EPNet uses rank-based selection (Yao 1993b) and five mutations: hybrid training, node deletion, connection deletion, connection addition and node addition (Yao and Liu 1997). EPNet uses a hybrid algorithm to train the neural network for a fixed number of epochs. Such training does not guarantee the convergence of neural network learning. Hence the training is partial. The other four mutations are used to grow and prune hidden nodes and connections.

The five mutations are attempted sequentially. If one mutation leads to a better offspring, it is regarded as successful. No further mutation will be applied. Otherwise the next mutation is attempted. The motivation behind ordering mutations is to encourage the evolution of compact neural networks without sacrificing generalization. A validation set is used in EPNet to measure the fitness of an individual, and another validation set to stop training in the final step. EPNet has been tested extensively on a number of benchmark problems, and very compact neural networks with good generalization ability have been evolved (Yao and Liu 1997).

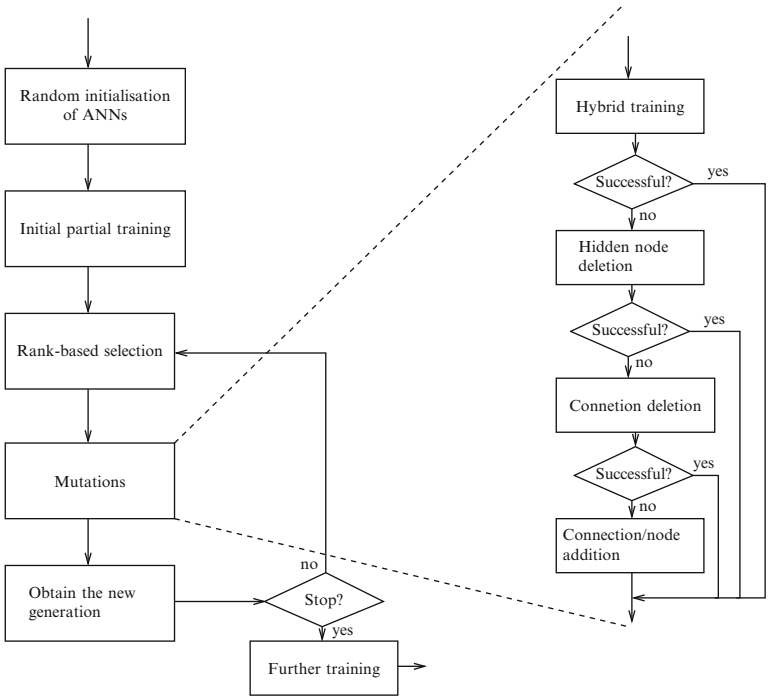

Fig. 17.5 The main structure of EPNet

17.3.2 The Evolution of Learning Rules

The evolution of learning rules can be regarded as a process of *learning to learn* in neural networks where the adaptation of learning rules is achieved through evolution. It can also be regarded as an adaptive process of automatic discovery of novel learning rules.

The relationship between evolution and learning is extremely complex. Various models have been proposed, but most of them deal with the issue of how learning can guide evolution and the relationship between the evolution of architectures and that of connection weights (Yao 1999). Research into the evolution of learning rules is still in its early stages. This research is important not only in providing an automatic way of optimizing learning rules and in modeling the relationship between learning and evolution, but also in modeling the creative process since newly evolved learning rules can deal with a complex and dynamic environment. This research will help us to understand better how creativity can emerge in artificial systems, like neural networks, and how to model the creative process in biological systems. A typical cycle of the evolution of learning rules can be described as follows (Yao 1999):

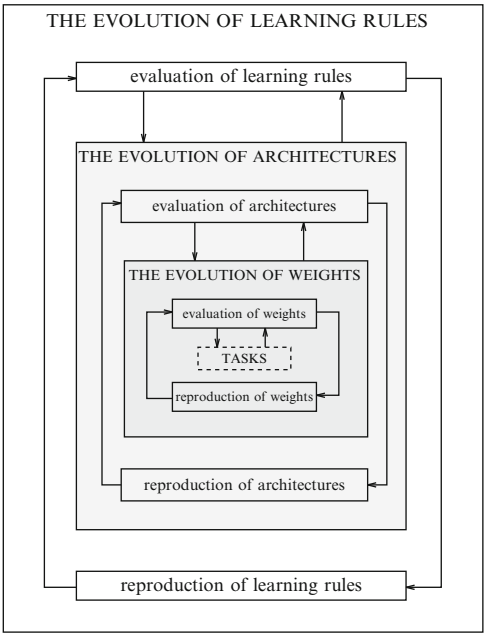

Fig. 17.6 A general framework for evolutionary neural networks

1. Decode each individual in the current generation into a learning rule.
2. Construct a set of neural networks with randomly generated architectures and initial weights, and train them using the decoded learning rules.
3. Calculate the fitness of each individual (encoded learning rule) according to the average training results.
4. Select parents from the current generation according to their fitness.
5. Apply search operators to parents to generate offspring which form the next generation.

The iteration stops when the population converges or a predefined maximum number of iterations has been reached.

17.3.3 A General Framework for Evolutionary Neural Networks

Figure 17.6 illustrates a general framework for evolutionary neural networks (Yao 1999). The evolution of connection weights proceeds at the lowest level on the fastest time scale in an environment determined by an architecture, a learning rule, and learning tasks. There are, however, two alternatives to decide the level of the evolution of architectures and that of learning rules: either the evolution of architectures is at the highest level and that of learning rules at the lower level or vice versa. The lower the level of evolution, the faster the time scale.

17.4 Ensemble Learning

17.4.1 Bias–Variance Trade-Off

In order to minimize $R(\mathbf{w})$ in Eq. (17.1) with an unknown probability distribution Pr, a training set $D = \{(\mathbf{x}(1), y(1)), \ldots, (\mathbf{x}(N), y(N))\}$ is selected and the function $F(\mathbf{x}, \mathbf{w})$ is chosen to minimize $\Sigma_{i=1}^{N} E[(F(\mathbf{x}(i), \mathbf{w}) - y(i))^2]$. Thus, the training set leads to a function $F(\mathbf{x}, \mathbf{w})$ that depends on D. To be explicit about dependence on the training set D, we rewrite the function $F(\mathbf{x}, \mathbf{w})$ as $F(\mathbf{x}, D)$.

The training set D can be chosen randomly using $(\mathcal{E}, Pr)$ by choosing N independent samples from $\mathcal{E}$. This can be described by a new probability space $(\mathcal{E}^{(N)}, Pr^{(N)})$ which consists of all the training sets D of given size N. Let E_D denote expectation over this space. Let E with no subscript denote the expectation over $(\mathcal{E}, Pr)$. Since the function $F(\mathbf{x}, D)$ is dependent on D, it differs from term to term in the sum for E_D. Consider the mean-squared error of the function $F(\mathbf{x}, D)$, which is defined by

$$E_D[(F(\mathbf{x}, D) - y)^2].$$

Taking expectations with respect to the training set D, we obtain the well-known separation of the mean-squared error (Geman et al. 1992)

$$\begin{aligned} E_D[(F(\mathbf{x}, D) - y)^2] &= E_D[F(\mathbf{x}, D)^2] - 2yE_D[F(\mathbf{x}, D)] + y^2 \\ &= (E_D[F(\mathbf{x}, D)] - y)^2 \\ &\quad + E_D[F(\mathbf{x}, D)^2] - (E_D[F(\mathbf{x}, D)])^2 \\ &= (E_D[F(\mathbf{x}, D)] - y)^2 \\ &\quad + var_D(F(\mathbf{x}, D)), \end{aligned} \tag{17.34}$$

where we have made use of the fact that y has constant expectation with respect to D, and variance definition

$$\begin{aligned} var_D(F(\mathbf{x}, D)) &= E_D[(F(\mathbf{x}, D) - E_D[F(\mathbf{x}, D)])^2] \\ &= E_D[F(\mathbf{x}, D)^2] - (E_D[F(\mathbf{x}, D)])^2. \end{aligned} \tag{17.35}$$

The first term $(E_D[F(\mathbf{x}, D)] - y)^2$ in the right-hand side of Eq. (17.34) represents the *bias* of the approximating function $F(\mathbf{x}, D)$. The bias measures how much the average function value at $\mathbf{x}$ deviates from y. The second term $var_D(F(\mathbf{x}, D))$ represents the *variance* of the approximating function $F(\mathbf{x}, D)$. The variance measures how much the function values at $\mathbf{x}$ vary from one training set to another.

Accordingly, Eq. (17.34) states that the expected mean-square value consists of the sum of two terms: bias and variance. Note that neither is negative. To achieve good performance, both the bias and the variance of the approximating function $F(\mathbf{x}, D)$ should be small.

If an allowed function $F(\mathbf{x}, D)$ is too simple, it will not be capable of capturing some of the aspects of the data. In particular, for a particular pair $(\mathbf{x}, y)$, there may be a general tendency to overestimate or a general tendency to underestimate. Both tendencies will make bias large. On the other hand, if an allowed function $F(\mathbf{x}, D)$

is too complex, it may be able to implement numerous solutions that are consistent with the training data, but most of these are likely to be poor approximations to data different from the training data. In other words, for any particular pair $(\mathbf{x}, y)$, a wide range of values of $F(\mathbf{x}, D)$, i.e. a large variance, may be obtained as the training set D varies.

There is usually a trade-off between bias and variance in the case of a training set with finite size (Geman et al. 1992): attempts to decrease bias by introducing more parameters often tend to increase variance; attempts to reduce variance by reducing parameters often tend to increase bias.

17.4.1.1 Bias–Variance–Covariance Trade-Off

There are many approaches to dealing with the bias–variance trade-off in neural network field. Given the training data set $D = \{(\mathbf{x}(1), y(1)), \ldots, (\mathbf{x}(N), y(N))\}$, this section considers estimating y by forming a neural network ensemble whose output is a simple averaging of outputs $F_i(\mathbf{x}, D)$ of a set of neural networks. All the individual networks in the ensemble are trained on the same training data set D:

$$F(\mathbf{x}, D) = \frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D), \tag{17.36}$$

where M is the number of individual networks in the neural network ensemble. Taking expectations with respect to the training set D, the expected mean-squared error of the neural network ensemble can be written in terms of individual network output:

$$E_D\left[(F(\mathbf{x}, D) - y)^2\right] = E_D\left[\left(\frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D) - y\right)^2\right]. \tag{17.37}$$

By use of Eq. (17.34), the right-hand side in (17.37) can be written as

$$E_D\left[\left(\frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D) - y\right)^2\right] = \left(E_D\left[\frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D)\right] - y\right)^2 + var_D\left(\frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D)\right), \tag{17.38}$$

where the first term in the right-hand side of (17.38) is the bias of the neural network ensemble, and the second term is the variance of the neural network ensemble. The variance of the neural network ensemble may be expressed as the sum of two terms:

$$\begin{aligned} var_D\left(\frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D)\right) &= E_D\left[\left(\frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D) - E_D\left[\frac{1}{M}\Sigma_{i=1}^{M} F_i(\mathbf{x}, D)\right]\right)^2\right] \\ &= E_D\left[\frac{1}{M^2}\left(\Sigma_{i=1}^{M}\left(F_i(\mathbf{x}, D) - E_D[F_i(\mathbf{x}, D)]\right)\right)^2\right] \end{aligned}$$

$$
\begin{aligned}
&= E_D\Big[\frac{1}{M^2}\big(\Sigma_{i=1}^{M}(F_i(\mathbf{x},D)-E_D[F_i(\mathbf{x},D)]\big)\\
&\quad \big(\Sigma_{j=1}^{M}(F_j(\mathbf{x},D)-E_D[F_j(\mathbf{x},D)]\big)\Big]\\
&= E_D\left[\frac{1}{M^2}\Sigma_{i=1}^{M}(F_i(\mathbf{x},D)-E_D[F_i(\mathbf{x},D)])^2\right]\\
&\quad +E_D\Big[\frac{1}{M^2}\Sigma_{i=1}^{M}\Sigma_{j=1,j\neq i}^{M}(F_i(\mathbf{x},D)-E_D[F_i(\mathbf{x},D)])\\
&\quad (F_j(\mathbf{x},D)-E_D[F_j(\mathbf{x},D)])\Big], \qquad (17.39)
\end{aligned}
$$

where the first term in the right-hand side of (17.39) is the weighted average of variance among the individual neural networks and the second term is the weighted average covariance among the different neural networks in the ensemble.

Similar to the bias–variance trade-off for a single neural network, there is a bias–variance–covariance trade-off for neural network ensembles. If the individual neural network F_i are highly positively correlated, for example $F_i = F, i = 1,\ldots,M$, there is no reduction in the variance of ensemble in this case. If the individual neural network F_i is uncorrelated, the weighted-average covariance among the different neural networks is reduced to zero, and the variance of ensemble can be seen to decay at $\frac{1}{M}$. Both theoretical and experimental results (Clemen and Winkler 1985; Perrone and Cooper 1993) have indicated that when individual neural networks in an ensemble are unbiased, average procedures are most effective in combining them when errors in the individual neural networks are negatively correlated and moderately effective when the errors are uncorrelated. There is little to be gained from average procedures when the errors are positively correlated.

There are a number of methods of learning neural network ensembles. To summarize, there are three approaches: independent learning, sequential learning and simultaneous learning.

17.4.1.2 Independent Learning Methods

It is clear that there is no advantage of combining a set of identical neural networks. In order to create a set of neural networks which are as uncorrelated as possible, a number of methods have been proposed to train a set of neural networks independently by varying initial random weights, the architectures, the learning algorithm used, and the data (Hansen and Salamon 1990; Sarkar 1996; Rogova 1994; Battiti and Colla 1994; Kim et al. 1995).

Experimental results have showed that neural networks obtained from a given neural network architecture for different initial random weights often correctly recognize different subsets of a given test set (Hansen and Salamon 1990; Sarkar 1996). As argued by Hansen and Salamon (1990), because each neural network makes generalization errors on different subsets of the input space, the collective decision produced by the ensemble is less likely to be in error than the decision made by any of the individual neural networks.

Currently, the commonest methods for the creation of ensembles are those which involve altering the training data, such as cross-validation (Krogh and Vedelsby 1995), bootstrapping (Raviv and Intrator 1996) and different input features (Rogova 1994; Battiti and Colla 1994).

Cross-validation is a method of estimating prediction error in its original form (Stone 1974). The procedure of m-fold cross-validation is as follows:

1. Split the data into m roughly equal-sized parts.
2. For the ith part, fit the model to the other $(m-1)$ parts of the data, and calculate the prediction error of fitted model when predicting the ith part of the data.
3. Do the above for $i = 1, \ldots, m$, and combine the m estimates of prediction error.

Cross-validation can be used to create a set of neural networks. Split the data into m roughly equal-sized parts, and train each neural network on the different parts independently. As indicated by Meir (1995), when the data set is small and noisy, such independence will help to reduce the correlation among the m neural networks more drastically than in the case where each neural network is trained on the full data.

When a larger set of independent neural networks is needed, splitting the training data into non-overlapping parts may cause each data part to be too small to train each neural network if no more data are available. In this case, data reuse methods, such as bootstrap (Efron and Tibshirani 1993), can help. Bootstrap was introduced in 1979 as a computer-based method for estimating the standard error of a statistic $s(x)$ (Efron and Tibshirani 1993). Breiman (1996) used the idea of bootstrap in bagging predictors. In bagging predictors, a training set containing N patterns is perturbed by sampling with replacement N times from the training set. The perturbed data set may contain repeats. This procedure can be repeated several times to create a number of different, although overlapping, data sets.

Although an ensemble primarily combines a set of neural networks with same architectures, it has been observed that classifiers based on different classifiers and features are frequently complementary to one another (Rogova 1994; Battiti and Colla 1994). Rogova (1994) proposed to combine several different neural network classifiers. For instance, three different neural network classifiers were used in Rogova's experiment on data of hand-printed digits (Rogova 1994).

17.4.1.3 Sequential Learning Methods

Most independent learning methods emphasize independence among individual neural networks in an ensemble. One of the disadvantages of such a method is the loss of interaction between the individual neural networks during learning. There is no consideration of whether what one individual learns has already been learned by other individuals. The errors of independently trained neural networks may still be positively correlated. It has been found that the combined results are weakened if the errors of individual neural networks are positively correlated (Clemen and Winkler 1985; Perrone and Cooper 1993). In order to decorrelate the individual neural

networks, sequential learning methods train a set of networks in a particular order (Drucker et al. 1993; Opitz and Shavlik 1996; Rosen 1996). Drucker et al. (1993) suggested training the neural networks using the boosting algorithm.

The boosting algorithm was originally proposed by Schapire (1990). Schapire proved that it is theoretically possible to convert a weak learning algorithm that performs only slightly better than random guessing into one that achieves arbitrary accuracy. The proof presented by Schapire (1990) is constructive. The construction uses filtering to modify the distribution of examples in such a way as to force the weak learning algorithm to focus on the harder-to-learn parts of the distribution.

The boosting algorithm trains a set of learning machines sequentially on data that have been filtered by the previously trained learning machines (Schapire 1990). As indicated by Drucker et al. (1994), the original boosting procedure is as follows. The first machine is trained with N_1 patterns randomly chosen from the available training data. After the first machine has been trained, a second training set with N_1 patterns is randomly selected on which the first machine would have 50% error rate. That is, there are 50% of patterns in the training set which the first machine misclassifies. Once the second machine has been trained with the second training set, another set of training patters are filtered through the first and second machines. Add the patterns on which the two machines disagree into the third training set for the third machine until there are total of N_1 patterns in it. Then the third machine is trained. During testing, each testing pattern is classified using the following voting scheme: if the first two machines agree, take their answer as the output; otherwise, assign the label as classified by the third machine.

Drucker et al. (1993) first used the idea of the boosting algorithm to improve performance of neural networks on four databases of optical character recognition problems. Drucker et al. (1994) compared the performance of the original version of boosting to that of a single neural network for an optical character recognition problem. The results showed that a single network was best for small training set size while for large training set size the original version of boosting was best. The ensemble used by Drucker et al. (1994) only consists of three individual neural networks, where the output of the ensemble is decided by adding of the outputs of the three neural networks rather than voting them.

The boosting algorithm can help to reduce the covariance between the different neural networks in an ensemble. A practical limitation of the original boosting algorithm Drucker et al. (1994) is that with a finite number of training patterns, unless the first network has very poor performance, there may not be enough patterns to generate a second or third training set. This limitation can be overcome by another boosting algorithm called AdaBoost. AdaBoost was developed by Freund and Schapire (1996) and analyzed theoretically by Schapire (1999). AdaBoost belongs to boosting by resampling that allows the training data to be reused.

Different to the boosting algorithm, Rosen (1996) proposed a decorrelation neural network training algorithm in which individual neural networks are trained not only to minimize the error between the target and its output, but also to decorrelate the errors with those from previously trained neural networks.

17.4.1.4 Simultaneous Learning Methods

Most of the independent training methods and sequential training methods follow a two-stage design process: first generating individual neural networks, and then combining them. The possible interactions between the individual neural networks cannot be exploited until the integration stage. There is no feedback from the integration stage to the individual neural network design stage. It is possible that some of the independently designed neural networks do not make much contribution to the integrated system. In order to use the feedback from the integration, simultaneous training methods train a set of neural networks together. The mixtures-of-experts (ME) architectures (Jacobs et al. 1991a,b; Jordan and Jacobs 1994) and negative correlation learning (Liu and Yao 1998a,b, 1999a,b; Chandra and Yao 2006) are two examples of simultaneous training methods.

The ME architecture is composed of multiple neural networks that combine aspects of competitive and associative learning (Jacobs et al. 1991a,b; Jordan and Jacobs 1994). The ME architecture is based on the principle of divide and conquer, in which a complex problem is decomposed into a set of simpler subproblems. It is assumed that the data can be adequately summarized by a collection of functions, each defined over a local region of the input space. The ME architecture adaptively partitions the input space into possibly overlapping regions and allocates different networks to summarize the data located in different regions. The ME architecture consists of two types of neural networks: a gating neural network and a number of expert neural networks. The ME architecture allows for all expert neural networks to look at the input and make their best guess. The gating neural network uses the normalized exponential transformation to weight the outputs of the expert neural networks to provide an overall best guess. All the parameter adjustments in the expert neural networks and gating neural network are performed simultaneously. Although the ME architecture can produce biased individual neural networks whose estimates are negatively correlated (Jacobs 1997), it does not provide a convenient way to balance the bias–variance–covariance trade-off.

17.4.1.5 Negative Correlation Learning

The idea of negative correlation learning is to introduce a correlation penalty term into the error function of each individual network so that the individual network can be trained simultaneously and interactively (Liu and Yao 1999a). Given the training data set $D = \{(\mathbf{x}(1), y(1)), \ldots, (\mathbf{x}(N), y(N))\}$, we consider estimating y by forming a neural network ensemble whose output is a simple averaging of outputs F_i of a set of neural networks. All the individual networks in the ensemble are trained on the same training data set D:

$$F(n) = \frac{1}{M}\Sigma_{i=1}^{M} F_i(n), \tag{17.40}$$

where $F_i(n)$ is the output of individual network i on the nth training pattern $\mathbf{x}(n)$, $F(n)$ is the output of the neural network ensemble on the nth training pattern, and M is the number of individual networks in the neural network ensemble.

In negative correlation learning, the error function E_i for individual i on the training data set $D = \{(\mathbf{x}(1), y(1)), \ldots, (\mathbf{x}(N), y(N))\}$ in negative correlation learning is defined by

$$\begin{aligned} E_i &= \frac{1}{N}\Sigma_{n=1}^{N} E_i(n) \\ &= \frac{1}{N}\Sigma_{n=1}^{N}\left[\frac{1}{2}(F_i(n) - y(n))^2 + \lambda p_i(n)\right], \end{aligned} \tag{17.41}$$

where N is the number of training patterns, $E_i(n)$ is the value of the error function of network i at presentation of the nth training pattern, and $y(n)$ is the desired output of the nth training pattern. The first term in the right-hand side of (17.41) is the mean-squared error of individual network i. The second term p_i is a correlation penalty function. The purpose of minimizing p_i is to negatively correlate each individual's error with errors for the rest of the ensemble. The parameter λ is used to adjust the strength of the penalty.

The penalty function p_i has the form

$$p_i(n) = -\frac{1}{2}(F_i(n) - F(n))^2. \tag{17.42}$$

The partial derivative of E_i with respect to the output of individual i on the nth training pattern is

$$\begin{aligned} \frac{\partial E_i(n)}{\partial F_i(n)} &= F_i(n) - y(n) - \lambda(F_i(n) - F(n)) \\ &= (1-\lambda)(F_i(n) - y(n)) + \lambda(F(n) - y(n)), \end{aligned} \tag{17.43}$$

where we have made use of the assumption that the output of ensemble $F(n)$ has constant value with respect to $F_i(n)$. The value of parameter λ lies inside the range $0 \leq \lambda \leq 1$ so that both $(1-\lambda)$ and λ have non-negative values.

Negative correlation learning has been analyzed in terms of mutual information on a regression task in the different noise conditions (Liu et al. 2001). Unlike independent training which creates larger mutual information among the ensemble, negative correlation learning can produce smaller mutual information among the ensemble. Through minimization of mutual information, very competitive results have been produced by negative correlation learning in comparison with independent training.

The decision boundaries and the correct response sets constructed by negative correlation learning and the independent training have been compared for two pattern classification problems (Liu et al. 2002). The experimental results show that negative correlation learning has a very good classification performance. In fact, the decision boundary formed by negative correlation learning is nearly close to the

optimum decision boundary generated by the Bayes classifier. Negative correlation learning has also been applied in online learning and class imbalance learning (Minku et al. 2010; Tang et al. 2009; Wang and Yao 2009a,b).

17.4.1.6 Evolutionary Neural Networks as Ensembles

Combining individual neural networks in a population into a neural network ensemble has a close relationship to the design of neural network ensembles. The population of neural networks can be regarded as an ensemble. The evolutionary process can be regarded as a natural and automatic way to design neural network ensembles.

Evolutionary ensembles with negative correlation learning (EENCL) were developed for automatically designing neural network ensembles (Liu et al. 2000). EENCL are studied to address the following issues: exploitation of the interaction between individual neural network design and combination and automatic determination of the number of individual neural networks. In EENCL, an evolutionary algorithm based on evolutionary programming (Fogel 1995) is used to search for a population of diverse individual neural networks that together solve a problem. To maintain a diverse population, fitness sharing (Yao et al. 1996) and negative correlation learning are used to encourage the formation of different species. In the implementation of EENCL, each neural network in the ensemble is a feedforward neural network with logistic transfer functions. The major steps of EENCL are as follows:

1. Generate an initial population of M neural networks, and set $k = 1$. The number of hidden nodes for each neural network, n_h, is specified by the user. The random initial weights are uniformly distributed inside a small range.
2. Train each neural network in the initial population on the training set for a certain number of epochs using negative correlation learning. The number of epochs, n_e, is specified by the user.
3. Randomly choose a group of n_b neural networks as parents to create n_b offspring neural networks by Gaussian mutation.
4. Add the n_b offspring neural networks to the population and train the offspring neural networks using negative correlation learning while the rest of the neural networks' weights are frozen.
5. Calculate the fitness of $M + n_b$ neural networks in the population and prune the population to the M fittest neural networks.
6. Go to the next step if the maximum number of generations has been reached. Otherwise, $k = k + 1$ and go to Step 3.
7. Form species using the k-means algorithm.
8. Combine species to form the ensembles.

There are two levels of adaptation in EENCL: negative correlation learning at the individual level and evolutionary learning based on evolutionary programming (Fogel 1995) at the population level. Negative correlation learning and fitness

sharing have been used to encourage the formation of species in the population. EENCL were tested on the Australian credit card assessment problem and the diabetes problem (Liu et al. 2000). Very competitive results have been produced by EENCL in comparison with other algorithms (Liu et al. 2000). Three combination methods have been investigated in EENCL, including simple averaging, majority voting and winner-takes-all.

Besides the *k*-means algorithm, regularized learning and constructive learning were also introduced in negative correlation learning for automatically determining the size of the ensembles (Chen and Yao 2009; Islam et al. 2003; Chen and Yao 2010).

17.5 Promising Areas for Future Application

Six recent trends and directions in machine learning are summarized by Dietterich (1997) and Langley (1996).

The first trend is in experimental studies of learning algorithms. Experimental studies of learning algorithms have shifted from the early study of idealized, handcrafted examples to realistic learning tasks that involve hundreds and thousands of cases. Besides robustness and the generality test of learning algorithms on a number of different data sets, comparisons between different learning algorithms on the same task domains need to be done. It has been realized that some explicit methods for evaluating different learning algorithms should be established, and the conditions in which a learning algorithm will perform well should be identified in order to make progress in machine learning.

The second trend is in theoretical analyses of learning processes. The main goal of theoretical analysis is to find the inductive principle with the best generalization, and then to develop learning algorithms with such inductive principle. Early studies on the convergence of learning algorithms were important but showed little insight into real learning problems. A major advance was due to the introduction of the probably approximately correct model (Vapnik 1995). For the first time, this model provided theoretical accuracy guarantees that were based on a finite number of training samples. The resulting probably approximately correct model also served the rigorous framework that addressed the concerns from real-world problems.

The third trend is in applications of machine learning. Most recent successful applications are in classification or prediction tasks. Machine learning has also been applied in the areas of configuration and layout, planning and scheduling, and execution and control. In order for machine learning to play a big role in solving problems of interest to industry and commerce, many more applications need to be undertaken.

The fourth trend is on new learning algorithms. Many new learning algorithms have been studied in the past decade. For example, a support vector machine can construct a hyperplane as the decision surface in such a way that the margin of separation between positive and negative examples is maximized. A boosting algorithm

trains a set of classifiers on data sets with entirely different distributions, and combines them in an elementary way to achieve near-optimal performance. The boosting algorithm was originally proposed by Schapire (1990). Schapire proved that it is theoretically possible to convert a weak learning algorithm that performs only slightly better than random guessing into one that achieves arbitrary accuracy. Boosting is a general method that can be used to improve the performance of any learning algorithm. Another ensemble learning called bagging combines models built on resamplings of data to yield superior models (Breiman 1996).

The fifth trend is in unified frameworks for machine learning. Machine learning has been widely studied from a variety of backgrounds. The similarities between the various approaches have often been overlooked while the differences between them were emphasized. It is important to draw distinctions among different learning algorithms. However, an ultimate goal of machine learning is to study a unified framework that can explain different learning processes in terms of common underlying mechanisms. One route to this goal is the study of hybrid learning systems that incorporate aspects of different learning algorithms.

The sixth trend is in integrated cognitive architectures. It is related to the development of integrated architectures for cognition. A common implementation in early work was to design a separate system for each new task. These systems had little of the nature of intelligent behavior, and posed limitations on work in other domains. Now research has moved to the design of integrated architectures that make strong assumptions about the control structures that can support intelligence. It is clear that learning will continue to play an important role in the development of such cognitive architectures when it is necessary to acquire knowledge from the environment for long-term adaptive behavior.

These new areas will confront researchers with many more challenge problems, and novel directions will surely emerge when the limitations of existing learning algorithms are revealed.

17.6 Tricks of the Trade

Newcomers to the field of machine learning, applying a learning algorithm to a given problem, are often not very clear about where to start to come up with a successful implementation. The following step-by-step procedures provide a guideline for implementing machine learning algorithms (Langley and Simon 1995).

17.6.1 Formulating the Problem

The first step is to formulate a given problem in terms of what can be dealt with by a particular learning algorithm. Often, some real-world problems can be transformed into simple classification tasks. For an example, the breast cancer diagnosis problem

can be formulated as a classification task that classifies a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Strategies such as divide-and-conquer can be used to decompose a complex task into a set of subproblems more amenable to the chosen learning algorithm. The strengths and limitations discussed in Sect. 17.2 provide a guideline for selecting an appropriate learning algorithm. Additionally, based on the feedback available in the problem, supervised learning can be chosen when specific output is supplied; unsupervised should be adopted when there is no specific output; reinforcement learning can be applied when the environment can be either fully observable or partially observable.

17.6.2 Choosing the Representation

The second step is to choose an appropriate representation for both the data and knowledge to be learned. The representation is referred to the attributes or features describing examples rather than the representational formalism, such as decision trees or neural networks. In some real-world problems, there might be thousands of potential features describing each input. Most learning algorithms do not scale well when there are many features. Meanwhile, examples with many irrelevant and noisy input features give little information from a statistical point of view. It is essential to choose useful and important features to feed to the learning algorithms. There are three main approaches for feature selection (Dietterich 1997). The first approach is to select a subset of the features based on some initial analysis. The second approach is to test different subsets of the features on the chosen learning algorithm and select the subsets that generate the best performance. The third approach is to automate the selection and weighting of features in the learning algorithm.

17.6.3 Collecting the Data

The third step is to collect data needed for the learning algorithm. In some applications, this process may be straightforward, but in others it can be very difficult. Generally speaking, the quantity of the data is decided by the chosen learning algorithm. Data preprocessing is often necessary in the learning process.

17.6.4 Conducting the Learning Process

Once the data are ready, the learning process can be started to find the best learning model within a set of candidate model structures according to a certain criterion. A standard tool in statistics known as cross-validation provides a good guiding criterion. First the collected data are randomly partitioned into a training set and a

test set. The training set is further divided into two disjoint subsets called estimation subset and validation subset in which the estimation subset is used to induce the learning model, and the validation subset is used to validate the model. It is possible that the learned model may end up overfitting on the validation subset. Therefore, the generalization performance of the learned model is measured on the test set which is different from the validation subset. Some learning algorithms, such as inductive logic programming, rely on the background knowledge available. How to obtain such helpful background knowledge is an important issue that will affect the outcome of those learning algorithms.

17.6.5 Analyzing and Evaluating the Learned Knowledge

Empirical comparisons have often been used to evaluate the predictive performance of the various learning methods. The experiments can be conducted on simulated data sets or a real-life data set, or both. The known best predictions on some simulated data sets make it possible to compare the learned knowledge with the known knowledge. Real-life data are helpful for evaluation of the robustness and generality of different learning algorithms. Cross-validation is a method of estimating prediction error in its original form (Stone 1974). The procedure of m-fold cross-validation is as follows:

1. Split the data into m roughly equal-sized parts.
2. For the ith part, fit the model to the other $(m-1)$ parts of the data, and calculate the prediction error of fitted model when predicting the ith part of the data.
3. Do the above for $i = 1, \ldots, m$, and combine the m estimates of prediction error.

In addition to empirical comparisons, statistical learning theory can be used to analyze the generalization ability of learning algorithms. Vapnik (1995) argued that the Vapnik–Chervonenkis dimension of the set of functions (rather than number of parameters) is responsible for the generalization ability of learning machines. This opens remarkable opportunities to overcome the "curse of dimensionality": to generalize well on the basis of a set of functions containing a huge number of parameters but possessing a small Vapnik–Chervonenkis dimension.

17.7 Conclusions

This chapter has been primarily concerned with the core learning algorithms including decision tree, inductive logic programming, Bayesian learning, neural networks, evolutionary learning and reinforcement learning. Inevitably, there are some important learning algorithms that have not been covered. One important learning algorithm dealing with imprecise and uncertain knowledge and data is fuzzy logic. Imprecision is treated based on probability in statistical learning. In contrast, fuzzy

logic is concerned with the use of fuzzy values that capture the meaning of words, human reasoning and decision making. At the heart of fuzzy logic lies the concept of a linguistic variable. The values of the linguistic variable are words rather than numbers.

Generalization is one of key issues in machine learning. In neural network learning, generalization was studied from the bias–variance trade-off point of view (Geman et al. 1992). There is usually a trade-off between bias and variance in the case of a training set with finite size: attempts to decrease bias by introducing more parameters often tend to increase variance; attempts to reduce variance by reducing parameters often tend to increase bias. Besides the generalization issue, how to scale up learning algorithms is another important issue. Dietterich (1997) reviewed learning with a large training set and learning with many features. Even though some learning techniques can solve very large problems with millions of training examples in a reasonable amount of computer time, it is unclear whether they can successfully be applied to those problems with billions of training examples.

This chapter has also been concerned with exploring the possible benefits arising from combining learning with evolution with neural networks. Different learning algorithms have their own strengths and weaknesses. Among all learning algorithms, there is no clear winner in terms of the best learning algorithm. The best one is always problem dependent. This is certainly true according to the no-free-lunch theorem (Wolpert and Macready 1997). In general, hybrid algorithms tend to perform better than others for a large number of problems.

Sources of Additional Information

The literature on machine learning is rather large, and has been growing rapidly.

- Mitchell's *Machine Learning* (Mitchell 1997) and Russell and Norvig's *Artificial Intelligence*: *A Modern Approach* (Russell and Norvig 2002) give good overviews of different types of learning algorithms.
- Machine Learning, volumes I–III, provide the early history of machine learning development (Michalski et al. 1983, 1986; Kodratoff and Michalski 1990).
- Some important papers in machine learning are collected in *Readings in Machine Learning* (Shavlik and Dietterich 1990).
- Current research in machine learning spreads out over a number of journals. Major machine learning journals include *Machine Learning*, the *Journal of Machine Learning Research*, *IEEE Transactions on Neural Networks*, *IEEE Transactions on Evolutionary Computation*, and mainstream artificial intelligence journals.
- Machine learning is also covered by a number of conferences, such as the International Conference on Machine Learning, the International Joint Conference on Neural Networks, Congress on Evolutionary Computation, the IEEE International Conference on Fuzzy Systems, and the Conference on Neural Information Processing Systems.

- Mlnet Online Information Service (http://www.mlnet.org/) funded by the European Commission is dedicated to the field of machine learning, knowledge discovery, case-based reasoning, knowledge acquisition and data mining.
- Machine learning topics can also be found at the website of the American Association for Artificial Intelligence: www.aaai.org/Pathfinder/html/machine.html.

References

Battiti R, Colla AM (1994) Democracy in neural nets: voting schemes for classification. Neural Netw 7:691–707

Breiman L (1996) Bagging predictors. Mach Learn 24:123–140

Breiman L, Friedman J, Olshen RA, Stone PJ (1984) Classification and regression trees. Wadsworth, Belmont

Chandra A, Yao X (2006) Ensemble learning using multi-objective evolutionary algorithms. J Math Model Algorithms 5:417–445

Chen H, Yao X (2009) Regularized negative correlation learning for neural network ensembles. IEEE Trans Neural Netw 20:1962–1979

Chen H, Yao X (2010) Multiobjective neural network ensembles based on regularized negative correlation Learning. IEEE Trans Knowl Data Eng 22:1738–1751

Cheng J, Greiner R, Kelly J, Bell DA, Liu W (2002) Learning Bayesian networks from data: an information-theory based approach. Artif Intell 137:43–90

Clemen RT, Winkler RL (1985) Limits for the precision and value of information from dependent sources. Oper Res 33:427–442

Dietterich TG (1997) Machine-learning research: four current directions. AI Mag 18:97–136

Domingos P, Pazzani M (1996) Beyond indpendence: conditions for the optimality of the simple Bayesian classifier. In: Saitta L (ed) Proceedings of the 13th international conference on machine learning, Bari. Morgan Kaufmann, San Mateo, pp 105–112

Drucker H, Schapire R, Simard P (1993) Improving performance in neural networks using a boosting algorithm. In: Hanson SJ et al (eds) Advances in neural information processing systems 5. Morgan Kaufmann, San Mateo, pp 42–49

Drucker H, Cortes C, Jackel LD, LeCun Y, Vapnik V (1994) Boosting and other ensemble methods. Neural Comput 6:1289–1301

Efron B, Tibshirani RJ (1993) An introduction to the bootstrap. Chapman and Hall, London

Elkan C (1997) Boosting and naive Bayesian learning. Technical report, Department of Computer Science and Engineering, University of California

Feigenbaum EA (1961) The simulation of verbal learning behavior. In: Proceedings of the western joint computer conference, Los Angeles, pp 121–131

Fogel DB (1995) Evolutionary computation: towards a new philosophy of machine intelligence. IEEE, New York

Fogel LJ, Owens AJ, Walsh MJ (1966) Artificial intelligence through simulated evolution. Wiley, New York

Freund Y, Schapire RE (1996) Experiments with a new boosting algorithm. In: Proceedings of the 13th international conference on machine learning, Bari. Morgan Kaufmann, San Mateo, pp 148–156

Geman S, Bienenstock E, Doursat R (1992) Neural networks and the bias/variance dilemma. Neural Comput 4:1–58

Hansen LK, Salamon P (1990) Neural network ensembles. IEEE Trans Pattern Anal Mach Intell 12:993–1001

Hebb DO (1949) The organization of behavior: a neurophysiological theory. Wiley, New York

Heckerman D (1998) A tutorial on learning with Bayesian networks. In: Jordan MI (ed) Learning in graphical models. Kluwer, Dordrecht

Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. Proc Nat Acad Sci USA 79:2554–2558

Hopfield JJ, Tank DW (1985) Neural computation of decisions in optimization problems. Biol Cybern 52:141–152

Hunt EB, Marin J, Stone PT (1966) Experiments in induction. Academic, New York

Islam MM, Yao X, Murase K (2003) A constructive algorithm for training cooperative neural network ensembles. IEEE Trans Neural Netw 14:820–834

Jacobs RA (1997) Bias/variance analyses of mixture-of-experts architectures. Neural Comput 9:369–383

Jacobs RA, Jordan MI, Barto AG (1991a) Task decomposition through competition in a modular connectionist architecture: the what and where vision task. Cogn Sci 15:219–250

Jacobs RA, Jordan MI, Nowlan SJ, Hinton GE (1991b) Adaptive mixtures of local experts. Neural Comput 3:79–87

Jordan MI, Jacobs RA (1994) Hierarchical mixtures-of-experts and the EM algorithm. Neural Comput 6:181–214

Kaelbling LP, Littman ML, Moore AW (1996) Reinforcement learning: a survey. J Artif Intell Res 4:237–285

Kim J, Ahn J, Cho S (1995) Ensemble competitive learning neural networks with reduced input dimensions. Int J Neural Syst 6:133–142

Kodratoff Y, Michalski RS (eds) (1990) Machine learning—an artificial intelligence approach 3. Morgan Kaufmann, San Mateo

Krogh A, Vedelsby J (1995) Neural network ensembles, cross validation, and active learning. In: Tesauro G et al (eds) Advances in neural information processing systems 7. MIT, Cambridge, pp 231–238

Langley P (1996) Elements of machine learning. Morgan Kaufmann, San Francisco

Langley P, Simon H (1995) Applications of machine learning and rule induction. Commun ACM 38:54–64

Lavrač N, Džeroski S (1994) Inductive logic programming: techniques and applications. Ellis Horwood, Chichester

Liu Y, Yao X (1998a) Negatively correlated neural networks can produce best ensembles. Aust J Intell Inf Process Syst 4:176–185

Liu Y, Yao X (1998b) A cooperative ensemble learning system. In: Proceedings of the IJCNN 1998, Anchorage. IEEE, Piscataway, pp 2202–2207
Liu Y, Yao X (1999a) Simultaneous training of negatively correlated neural networks in an ensemble. IEEE Trans Syst Man Cybern B 29:716–725
Liu Y, Yao X (1999b) Ensemble learning via negative correlation. Neural Netw 12:1399–1404
Liu Y, Yao X, Higuchi T (2000) Evolutionary ensembles with negative correlation learning. IEEE Trans Evol Comput 4:380–387
Liu Y, Yao X, Higuchi T (2001) Ensemble learning by minimizing mutual information. In: Proceedings of the 2nd international conference on software engineer, artificial intelligence, networking and parallel/distributed computing, Nagoya. International association for computer and information science, pp 457–462
Liu Y, Yao X, Zhao Q, Higuchi T (2002) An experimental comparison of neural network ensemble learning methods on decision boundaries. In: Proceedings of the IJCNN 2002, Honolulu. IEEE, Piscataway, pp 221–226
McCulloch WS, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. Bull Math Biophys 5:115–137
Meir R (1995) Bias, variance, and the combination of least squares estimators. In: Tesauro G, Touretzky DS, Leen TK (eds) Advances in neural information processing systems 7. MIT, Cambridge, pp 295–302
Michalski RS, Carbonell JG, Mitchell TM (eds) (1983) Machine learning—an artificial intelligence approach 1. Morgan Kaufmann, San Mateo
Michalski RS, Carbonell JG, Mitchell TM (eds) (1986) Machine learning—an artificial intelligence approach 2. Morgan Kaufmann, San Mateo
Michie D, Spiegelhalter DJ, Taylor CC (1994) Machine learning, neural and statistical classification. Ellis Horwood, London
Minku LL, White A, Yao X (2010) The impact of diversity on on-line ensemble learning in the presence of concept drift. IEEE Trans Knowl Data Eng 22:730–742
Minsky ML, Papert S (1969) Perceptrons: an introduction to computational geometry. MIT, Cambridge
Mitchell TM (1997) Machine learning. McGraw-Hill, New York
Muggleton SH (1995) Inverse entailment and progol. New Gener Comput 13:245–286
Muggleton SH, Buntine W (1988) Machine invention of first-order predicates by inverting resolution. In: Proceedings of the 5th international conference on machine learning, Ann Arbor. Morgan Kaufmann, San Mateo, pp 339–352
Opitz DW, Shavlik JW (1996) Actively searching for an effective neural network ensemble. Connect Sci 8:337–353
Perrone MP, Cooper LN (1993) When networks disagree: ensemble methods for hybrid neural networks. In: Mammone RJ (ed) Neural networks for speech and image processing. Chapman and Hall, London
Quinlan JR (1986) Introduction to decision tree. Mach Learn 1:81–106
Quinlan JR (1990) Learning logical definitions from relations. Mach Learn 5:239–266

Quinlan JR (1993) C4.5: programs for machine learning. Morgan Kaufmann, San Mateo

Raviv Y, Intrator N (1996) Bootstrapping with noise: an effective regularization technique. Connect Sci 8:355–372

Rogova G (1994) Combining the results of several neural networks classifiers. Neural Netw 7:777–781

Rosen BE (1996) Ensemble learning using decorrelated neural networks. Connect Sci 8:373–383

Rosenblatt F (1962) Principles of neurodynamics: perceptrons and the theory of brain mechanisms. Spartan, Chicago

Rumelhart DE, McClelland JL (ed) (1986) Parallel distributed processing: explorations in the microstructures of cognition. MIT, Cambridge

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning internal representations by error propagation. In: Rumelhart DE, McClelland JL (eds) Parallel distributed processing: explorations in the microstructures of cognition I. MIT, Cambridge, pp 318–362

Russell S, Norvig P (2002) Artificial intelligence: a modern approach. Prentice-Hall, Englewood Cliffs

Samuel AL (1959) Some studies in machine learning using the game of checkers. IBM J Res Dev 3:210–229

Sarkar D (1996) Randomness in generalization ability: a source to improve it. IEEE Trans Neural Netw 7:676–685

Schapire RE (1990) The strength of weak learnability. Mach Learn 5:197–227

Schapire RE (1999) Theoretical views of boosting and applications. In: Proceedings of the 10th international conference on algorithmic learning theory, Tokyo. Springer, Berlin, pp 13–25

Schwefel HP (1981) Numerical optimization of computer models. Wiley, Chichester

Schwefel HP (1995) Evolution and optimum seeking. Wiley, New York

Shavlik J, Dietterich T (eds) (1990) Readings in machine learning. Morgan Kaufmann, San Mateo

Stone M (1974) Cross-validatory choice and assessment of statistical predictions. J R Stat Soc 36:111–147

Sutton RS, Barto AG (1998) Reinforcement learning: an introduction. MIT, Cambridge

Tang K, Lin M, Minku FL, Yao X (2009) Selective negative correlation learning approach to incremental learning. Neurocomputing 72:2796–2805

Turing A (1950) Computing machinery and intelligence. Mind 59:433–460

Vapnik VN (1995) The nature of statistical learning theory. Springer, New York

Wang S, Yao X (2009a) Theoretical study of the relationship between diversity and single-class measures for class imbalance learning. In: Proceedings of the IEEE international conference on data mining workshops, Miami. IEEE Computer Society, Washington, DC, pp 76–81

Wang S, Yao X (2009b) Diversity exploration and negative correlation learning on imbalanced data sets. In: Proceedings of the IJCNN 2009, Atlanta, pp 3259–3266

Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 1:67–82

Yao X (1991) Evolution of connectionist networks. In: Dartnall T (ed) Preprints of the international symposium on AI, reasoning and creativity, Griffith University, Queensland, pp 49–52

Yao X (1993a) A review of evolutionary artificial neural networks. Int J Intell Syst 8:539–567

Yao X (1993b) An empirical study of genetic operators in genetic algorithms. Microprocess Microprogr 38:707–714

Yao X (1994) The evolution of connectionist networks. In: Dartnall T (ed) Artificial intelligence and creativity. Kluwer, Dordrecht, pp 233–243

Yao X (1995) Evolutionary artificial neural networks. In: Kent A, Williams JG (eds) Encyclopedia of computer science and technology 33. Dekker, New York, pp 137–170

Yao X (1999) Evolving artificial neural networks. Proc IEEE 87:1423–1447

Yao X, Liu Y (1997) A new evolutionary system for evolving artificial neural networks. IEEE Trans Neural Netw 8:694–713

Yao X, Liu Y (1998) Making use of population information in evolutionary artificial neural networks. IEEE Trans Syst Man Cybern B 28:417–425

Yao X, Liu Y, Darwen P (1996) How to make best use of evolutionary learning. In: Stocker R, Jelinek H, Durnota B (eds) Complex systems: from local interactions to global phenomena. IOS, Amsterdam, pp 229–242