- Learning to recognize spoken words.

  All of the most successful speech recognition systems employ machine learning in some form. For example, the SPHINX system (e.g., Lee 1989) learns speaker-specific strategies for recognizing the primitive sounds (phonemes) and words from the observed speech signal. Neural network learning methods (e.g., Waibel et al. 1989) and methods for learning hidden Markov models (e.g., Lee 1989) are effective for automatically customizing to individual speakers, vocabularies, microphone characteristics, background noise, etc. Similar techniques have potential applications in many signal-interpretation problems.
- Learning to drive an autonomous vehicle.

  Machine learning methods have been used to train computer-controlled vehicles to steer correctly when driving on a variety of road types. For example, the ALVINN system (Pomerleau 1989) has used its learned strategies to drive unassisted at 70 miles per hour for 90 miles on public highways among other cars. Similar techniques have possible applications in many sensor-based control problems.
- Learning to classify new astronomical structures.

  Machine learning methods have been applied to a variety of large databases to learn general regularities implicit in the data. For example, decision tree learning algorithms have been used by NASA to learn how to classify celestial objects from the second Palomar Observatory Sky Survey (Fayyad et al. 1995). This system is now used to automatically classify all objects in the Sky Survey, which consists of three terrabytes of image data.
- Learning to play world-class backgammon.

  The most successful computer programs for playing games such as backgammon are based on machine learning algorithms. For example, the world's top computer program for backgammon, TD-GAMMON (Tesauro 1992, 1995), learned its strategy by playing over one million practice games against itself. It now plays at a level competitive with the human world champion. Similar techniques have applications in many practical problems where very large search spaces must be examined efficiently.

TABLE 1.1
Some successful applications of machine learning.

three features: the class of tasks, the measure of performance to be improved, and the source of experience.

A checkers learning problem:

- Task T: playing checkers
- Performance measure P: percent of games won against opponents
- Training experience E: playing practice games against itself

We can specify many learning problems in this fashion, such as learning to recognize handwritten words, or learning to drive a robotic automobile autonomously.

A handwriting recognition learning problem:

- Task T: recognizing and classifying handwritten words within images
- Performance measure P: percent of words correctly classified

- Artificial intelligence

  Learning symbolic representations of concepts. Machine learning as a search problem. Learning as an approach to improving problem solving. Using prior knowledge together with training data to guide learning.
- Bayesian methods

  Bayes' theorem as the basis for calculating probabilities of hypotheses. The naive Bayes classifier. Algorithms for estimating values of unobserved variables.
- Computational complexity theory

  Theoretical bounds on the inherent complexity of different learning tasks, measured in terms of the computational effort, number of training examples, number of mistakes, etc. required in order to learn.
- Control theory

  Procedures that learn to control processes in order to optimize predefined objectives and that learn to predict the next state of the process they are controlling.
- Information theory

  Measures of entropy and information content. Minimum description length approaches to learning. Optimal codes and their relationship to optimal training sequences for encoding a hypothesis.
- Philosophy

  Occam's razor, suggesting that the simplest hypothesis is the best. Analysis of the justification for generalizing beyond observed data.
- Psychology and neurobiology

  The power law of practice, which states that over a very broad range of learning problems, people's response time improves with practice according to a power law. Neurobiological studies motivating artificial neural network models of learning.
- Statistics

  Characterization of errors (e.g., bias and variance) that occur when estimating the accuracy of a hypothesis based on a limited sample of data. Confidence intervals, statistical tests.

TABLE 1.2
Some disciplines and examples of their influence on machine learning.

- Training experience E: a database of handwritten words with given classifications

A robot driving learning problem:

- Task T: driving on public four-lane highways using vision sensors
- Performance measure P: average distance traveled before an error (as judged by human overseer)
- Training experience E: a sequence of images and steering commands recorded while observing a human driver

Our definition of learning is broad enough to include most tasks that we would conventionally call "learning" tasks, as we use the word in everyday language. It is also broad enough to encompass computer programs that improve from experience in quite straightforward ways. For example, a database system that allows users to update data entries would fit our definition of a learning system: it improves its performance at answering database queries, based on the experience gained from database updates. Rather than worry about whether this type of activity falls under the usual informal conversational meaning of the word "learning," we will simply adopt our technical definition of the class of programs that improve through experience. Within this class we will find many types of problems that require more or less sophisticated solutions. Our concern here is not to analyze the meaning of the English word "learning" as it is used in everyday language. Instead, our goal is to define precisely a class of problems that encompasses interesting forms of learning, to explore algorithms that solve such problems, and to understand the fundamental structure of learning problems and processes.

1.2 DESIGNING A LEARNING SYSTEM

In order to illustrate some of the basic design issues and approaches to machine learning, let us consider designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament. We adopt the obvious performance measure: the percent of games it wins in this world tournament.

1.2.1 Choosing the Training Experience

The first design choice we face is to choose the type of training experience from which our system will learn. The type of training experience available can have a significant impact on success or failure of the learner. One key attribute is whether the training experience provides direct or indirect feedback regarding the choices made by the performance system. For example, in learning to play checkers, the system might learn from *direct* training examples consisting of individual checkers board states and the correct move for each. Alternatively, it might have available only *indirect* information consisting of the move sequences and final outcomes of various games played. In this later case, information about the correctness of specific moves early in the game must be inferred indirectly from the fact that the game was eventually won or lost. Here the learner faces an additional problem of *credit assignment*, or determining the degree to which each move in the sequence deserves credit or blame for the final outcome. Credit assignment can be a particularly difficult problem because the game can be lost even when early moves are optimal, if these are followed later by poor moves. Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

A second important attribute of the training experience is the degree to which the learner controls the sequence of training examples. For example, the learner might rely on the teacher to select informative board states and to provide the correct move for each. Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move. Or the learner may have complete control over both the board states and (indirect) training classifications, as it does when it learns by playing against itself with no teacher present. Notice in this last case the learner may choose between experimenting with novel board states that it has not yet considered, or honing its skill by playing minor variations of lines of play it currently finds most promising. Subsequent chapters consider a number of settings for learning, including settings in which training experience is provided by a random process outside the learner's control, settings in which the learner may pose various types of queries to an expert teacher, and settings in which the learner collects training examples by autonomously exploring its environment.

A third important attribute of the training experience is how well it represents the distribution of examples over which the final system performance P must be measured. In general, learning is most reliable when the training examples follow a distribution similar to that of future test examples. In our checkers learning scenario, the performance metric P is the percent of games the system wins in the world tournament. If its training experience E consists only of games played against itself, there is an obvious danger that this training experience might not be fully representative of the distribution of situations over which it will later be tested. For example, the learner might never encounter certain crucial board states that are very likely to be played by the human checkers champion. In practice, it is often necessary to learn from a distribution of examples that is somewhat different from those on which the final system will be evaluated (e.g., the world checkers champion might not be interested in teaching the program!). Such situations are problematic because mastery of one distribution of examples will not necessary lead to strong performance over some other distribution. We shall see that most current theory of machine learning rests on the crucial assumption that the distribution of training examples is identical to the distribution of test examples. Despite our need to make this assumption in order to obtain theoretical results, it is important to keep in mind that this assumption must often be violated in practice.

To proceed with our design, let us decide that our system will train by playing games against itself. This has the advantage that no external trainer need be present, and it therefore allows the system to generate as much training data as time permits. We now have a fully specified learning task.

A checkers learning problem:

- Task T: playing checkers
- Performance measure P: percent of games won in the world tournament
- Training experience E: games played against itself

In order to complete the design of the learning system, we must now choose

1. the exact type of knowledge to be learned
2. a representation for this target knowledge
3. a learning mechanism

1.2.2 Choosing the Target Function

The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program. Let us begin with a checkers-playing program that can generate the *legal* moves from any board state. The program needs only to learn how to choose the *best* move from among these legal moves. This learning task is representative of a large class of tasks for which the legal moves that define some large search space are known a priori, but for which the best search strategy is not known. Many optimization problems fall into this class, such as the problems of scheduling and controlling manufacturing processes where the available manufacturing steps are well understood, but the best strategy for sequencing them is not.

Given this setting where we must learn to choose among the legal moves, the most obvious choice for the type of information to be learned is a program, or function, that chooses the best move for any given board state. Let us call this function $ChooseMove$ and use the notation $ChooseMove : B \rightarrow M$ to indicate that this function accepts as input any board from the set of legal board states B and produces as output some move from the set of legal moves M. Throughout our discussion of machine learning we will find it useful to reduce the problem of improving performance P at task T to the problem of learning some particular *target function* such as $ChooseMove$. The choice of the target function will therefore be a key design choice.

Although $ChooseMove$ is an obvious choice for the target function in our example, this function will turn out to be very difficult to learn given the kind of indirect training experience available to our system. An alternative target function—and one that will turn out to be easier to learn in this setting—is an evaluation function that assigns a numerical score to any given board state. Let us call this target function V and again use the notation $V : B \rightarrow \Re$ to denote that V maps any legal board state from the set B to some real value (we use $\Re$ to denote the set of real numbers). We intend for this target function V to assign higher scores to better board states. If the system can successfully learn such a target function V, then it can easily use it to select the best move from any current board position. This can be accomplished by generating the successor board state produced by every legal move, then using V to choose the best successor state and therefore the best legal move.

What exactly should be the value of the target function V for any given board state? Of course any evaluation function that assigns higher scores to better board states will do. Nevertheless, we will find it useful to define one particular target function V among the many that produce optimal play. As we shall see, this will make it easier to design a training algorithm. Let us therefore define the target value $V(b)$ for an arbitrary board state b in B, as follows:

1. if b is a final board state that is won, then $V(b) = 100$
2. if b is a final board state that is lost, then $V(b) = -100$
3. if b is a final board state that is drawn, then $V(b) = 0$

4. if b is a not a final state in the game, then $V(b) = V(b')$, where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game (assuming the opponent plays optimally, as well).

While this recursive definition specifies a value of $V(b)$ for every board state b, this definition is not usable by our checkers player because it is not efficiently computable. Except for the trivial cases (cases 1–3) in which the game has already ended, determining the value of $V(b)$ for a particular board state requires (case 4) searching ahead for the optimal line of play, all the way to the end of the game! Because this definition is not efficiently computable by our checkers playing program, we say that it is a *nonoperational* definition. The goal of learning in this case is to discover an *operational* description of V; that is, a description that can be used by the checkers-playing program to evaluate states and select moves within realistic time bounds.

Thus, we have reduced the learning task in this case to the problem of discovering an *operational description of the ideal target function* V. It may be very difficult in general to learn such an operational form of V perfectly. In fact, we often expect learning algorithms to acquire only some *approximation* to the target function, and for this reason the process of learning the target function is often called *function approximation*. In the current discussion we will use the symbol $\hat{V}$ to refer to the function that is actually learned by our program, to distinguish it from the ideal target function V.

1.2.3 Choosing a Representation for the Target Function

Now that we have specified the ideal target function V, we must choose a representation that the learning program will use to describe the function $\hat{V}$ that it will learn. As with earlier design choices, we again have many options. We could, for example, allow the program to represent $\hat{V}$ using a large table with a distinct entry specifying the value for each distinct board state. Or we could allow it to represent $\hat{V}$ using a collection of rules that match against features of the board state, or a quadratic polynomial function of predefined board features, or an artificial neural network. In general, this choice of representation involves a crucial tradeoff. On one hand, we wish to pick a very expressive representation to allow representing as close an approximation as possible to the ideal target function V. On the other hand, the more expressive the representation, the more training data the program will require in order to choose among the alternative hypotheses it can represent. To keep the discussion brief, let us choose a simple representation: for any given board state, the function $\hat{V}$ will be calculated as a linear combination of the following board features:

- x_1: the number of black pieces on the board
- x_2: the number of red pieces on the board
- x_3: the number of black kings on the board
- x_4: the number of red kings on the board

- x_5: the number of black pieces threatened by red (i.e., which can be captured on red's next turn)
- x_6: the number of red pieces threatened by black

Thus, our learning program will represent $\hat{V}(b)$ as a linear function of the form

$$\hat{V}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

where w_0 through w_6 are numerical coefficients, or weights, to be chosen by the learning algorithm. Learned values for the weights w_1 through w_6 will determine the relative importance of the various board features in determining the value of the board, whereas the weight w_0 will provide an additive constant to the board value.

To summarize our design choices thus far, we have elaborated the original formulation of the learning problem by choosing a type of training experience, a target function to be learned, and a representation for this target function. Our elaborated learning task is now

Partial design of a checkers learning program:

- Task T: playing checkers
- Performance measure P: percent of games won in the world tournament
- Training experience E: games played against itself
- *Target function:* $V{:}Board \rightarrow \Re$
- *Target function representation*

$$\hat{V}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

The first three items above correspond to the specification of the learning task, whereas the final two items constitute design choices for the implementation of the learning program. Notice the net effect of this set of design choices is to reduce the problem of learning a checkers strategy to the problem of learning values for the coefficients w_0 through w_6 in the target function representation.

1.2.4 Choosing a Function Approximation Algorithm

In order to learn the target function $\hat{V}$ we require a set of training examples, each describing a specific board state b and the training value $V_{train}(b)$ for b. In other words, each training example is an ordered pair of the form $\langle b, V_{train}(b)\rangle$. For instance, the following training example describes a board state b in which black has won the game (note $x_2 = 0$ indicates that red has no remaining pieces) and for which the target function value $V_{train}(b)$ is therefore $+100$.

$$\langle\langle x_1 = 3, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 0\rangle, +100\rangle$$

Below we describe a procedure that first derives such training examples from the indirect training experience available to the learner, then adjusts the weights w_i to best fit these training examples.

1.2.4.1 ESTIMATING TRAINING VALUES

Recall that according to our formulation of the learning problem, the only training information available to our learner is whether the game was eventually won or lost. On the other hand, we require training examples that assign specific scores to specific board states. While it is easy to assign a value to board states that correspond to the end of the game, it is less obvious how to assign training values to the more numerous *intermediate* board states that occur before the game's end. Of course the fact that the game was eventually won or lost does not necessarily indicate that *every* board state along the game path was necessarily good or bad. For example, even if the program loses the game, it may still be the case that board states occurring early in the game should be rated very highly and that the cause of the loss was a subsequent poor move.

Despite the ambiguity inherent in estimating training values for intermediate board states, one simple approach has been found to be surprisingly successful. This approach is to assign the training value of $V_{train}(b)$ for any intermediate board state b to be $\hat{V}(Successor(b))$, where $\hat{V}$ is the learner's current approximation to V and where $Successor(b)$ denotes the next board state following b for which it is again the program's turn to move (i.e., the board state following the program's move and the opponent's response). This rule for estimating training values can be summarized as

Rule for estimating training values.

$$V_{train}(b) \leftarrow \hat{V}(Successor(b)) \tag{1.1}$$

While it may seem strange to use the current version of $\hat{V}$ to estimate training values that will be used to refine this very same function, notice that we are using estimates of the value of the $Successor(b)$ to estimate the value of board state b. Intuitively, we can see this will make sense if $\hat{V}$ tends to be more accurate for board states closer to game's end. In fact, under certain conditions (discussed in Chapter 13) the approach of iteratively estimating training values based on estimates of successor state values can be proven to converge toward perfect estimates of V_{train}.

1.2.4.2 ADJUSTING THE WEIGHTS

All that remains is to specify the learning algorithm for choosing the weights w_i to best fit the set of training examples $\{\langle b, V_{train}(b)\rangle\}$. As a first step we must define what we mean by the *best fit* to the training data. One common approach is to define the best hypothesis, or set of weights, as that which minimizes the squared error E between the training values and the values predicted by the hypothesis $\hat{V}$.

$$E \equiv \sum_{\langle b, V_{train}(b)\rangle \in \, training\ examples} (V_{train}(b) - \hat{V}(b))^2$$

Thus, we seek the weights, or equivalently the $\hat{V}$, that minimize E for the observed training examples. Chapter 6 discusses settings in which minimizing the sum of squared errors is equivalent to finding the most probable hypothesis given the observed training data.

Several algorithms are known for finding weights of a linear function that minimize E defined in this way. In our case, we require an algorithm that will incrementally refine the weights as new training examples become available and that will be robust to errors in these estimated training values. One such algorithm is called the least mean squares, or LMS training rule. For each observed training example it adjusts the weights a small amount in the direction that reduces the error on this training example. As discussed in Chapter 4, this algorithm can be viewed as performing a stochastic gradient-descent search through the space of possible hypotheses (weight values) to minimize the squared error E. The LMS algorithm is defined as follows:

LMS weight update rule.

For each training example $\langle b, V_{train}(b)\rangle$

- Use the current weights to calculate $\hat{V}(b)$
- For each weight w_i, update it as

$$w_i \leftarrow w_i + \eta\ (V_{train}(b) - \hat{V}(b))\ x_i$$

Here η is a small constant (e.g., 0.1) that moderates the size of the weight update. To get an intuitive understanding for why this weight update rule works, notice that when the error $(V_{train}(b) - \hat{V}(b))$ is zero, no weights are changed. When $(V_{train}(b) - \hat{V}(b))$ is positive (i.e., when $\hat{V}(b)$ is too low), then each weight is increased in proportion to the value of its corresponding feature. This will raise the value of $\hat{V}(b)$, reducing the error. Notice that if the value of some feature x_i is zero, then its weight is not altered regardless of the error, so that the only weights updated are those whose features actually occur on the training example board. Surprisingly, in certain settings this simple weight-tuning method can be proven to converge to the least squared error approximation to the V_{train} values (as discussed in Chapter 4).

1.2.5 The Final Design

The final design of our checkers learning system can be naturally described by four distinct program modules that represent the central components in many learning systems. These four modules, summarized in Figure 1.1, are as follows:

- The **Performance System** is the module that must solve the given performance task, in this case playing checkers, by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output. In our case, the