1

Introduction

Learning involves vital functions at different levels of consciousness, starting with the recognition of sensory stimuli up to the acquisition of complex notions for sophisticated abstract reasoning. Even though learning escapes precise definition there is general agreement on Langley's idea (Langley, 1986) of learning as a set of "mechanisms through which intelligent agents improve their behavior over time", which seems reasonable once a sufficiently broad view of "agent" is taken. Machine learning has its roots in several disciplines, notably statistics, pattern recognition, the cognitive sciences, and control theory. Its main goal is to help humans in constructing programs that cannot be built up manually and programs that learn from experience. Another goal of machine learning is to provide computational models for human learning, thus supporting cognitive studies of learning.

Machine learning's roots

Among the large variety of tasks that constitute the body of machine learning, one has received attention from the beginning: the acquiring of knowledge for performing *classification*. From this perspective machine learning can be described roughly as the process of discovering regularities from a set of available data and extrapolating these regularities to new data.

Classification

Over the years, machine learning has been understood in different ways. At first it was considered mainly as an algorithmic process. One of the first approaches to automated learning was proposed by Gold in his "learning in the limit" paradigm (Gold, 1967). This type of learning provides an infinite sequence of pieces of data to the learner, who generates a model that *explains* the data. At each new input the learner updates its current model (the "hypothesis"), hoping, but never knowing for sure, that it is closer to the "correct" one.

Machine learning as an algorithm

Gold's paradigm

A fundamental change in machine learning was the recognition of its nature as a *search* problem (Mitchell, 1982). Given a set of data and some language(s) for describing the data and the target knowledge, learning consists in the exploration of a hypothesis space, guided by a heuristic, until a specified termination

Machine learning as search

condition is met; as the search space is usually too large to be explored exhaustively, the learner must have a criterion to evaluate and compare hypotheses. In order to facilitate the search the hypothesis space is usually internally structured according to a *generality* relation.

Just as learning is a fundamental task in any living organism, machine learning is a fundamental task in artificial intelligence as well. It is impossible to conceive a truly intelligent agent that is not provided with the ability to extend its knowledge and improve its performance over time.

Appealing as it may be, machine learning encounters severe difficulties, which even today hinder its full exploitation. The main obstacle to be overcome is that most machine learning algorithms are very demanding in terms of computational resources, especially those that are closer to the human process of learning. This concept of *computational complexity* in learning is the core around which this book is constructed.

Computational complexity of learning

For hundreds of years the abstract nature of mathematical truths required advances through proving theorems. Existence or constructive proofs were concerned with the logical soundness of the derived results, without any attention to their concrete attainability. The same was true for algorithms: the only relevant aspect was their correctness, not their practical execution. Mathematical knowledge appeared to scientists as only limited by human skill in discovering or inventing it.

With the advent of information science, things changed radically. In fact, logician Kurt Gödel's work provided clear evidence that the discovery of some mathematical truths may be intrinsically limited (Gödel, 1931). In fact, with his famous *incompleteness theorem* he proved that Hilbert's belief in the existence of an effective procedure determining the truth or falsity of any mathematical proposition was illfounded: thus the notion of *undecidability* was born. In order to understand this fundamental notion better we have to be more precise about the concept of an *algorithm*. The word "algorithm" derives from the name of the Persian mathematician Abu Abdullah Muhammad ibn Musa al-Khwarizmi, whose work introduced Arabic numerals and algebraic concepts to the western world. He worked in Baghdad in the ninth century, when the city was a centre of scientific studies. The ancient word *algorism* originally referred only to the rules of performing arithmetic using Arabic numerals but evolved via the Latin translation of al-Khwarizmi's name into *algorithm* by the 18th century. In its more intuitive formulation, an *algorithm* is a precise and unambiguous sequence of steps that, given a problem to be solved and some input data, provides the solution thereof.[1]

Gödel's incompleteness theorem

Algorithm

[1] Actually, one may clarify the difference between *procedures* and *algorithms* by reserving the latter name for procedures that terminate. As we are concerned only with the halting case, we will use the two terms interchangeably.

In general, a particular problem to be solved is a specific instance of a *class* of problems. For example, the problem of sorting in ascending order the elements of a vector $\vec{\mathbf{x}}$ of n integer numbers belongs to a class Π of similar problems containing all such vectors, each with a different length n and different content. The notion of decidability refers to the class of problems as a whole, not to a single instance. More precisely, given a class of problems Π, we will say that the class is *decidable* if there exists an algorithm that, given as an input any instance of the problem class, provides a solution. Then, undecidability does not prevent any single instance from being solved but, rather, it limits the *generality* of the algorithm for finding the solution in any instance. In other words, for a decidable class a single algorithm is able to solve any instance of the class whereas for an undecidable class every problem instance must be solved with, in principle, a different algorithm.[2]

Decidability

In order to prove that a problem class is undecidable one has to show that no unique algorithm solves all its instances. This is usually done by reducing the problem (class) to a known undecidable problem. A basic undecidable problem is the *halting* problem, proved undecidable by Alan Turing in 1936 (Turing, 1936). The halting problem consists of writing a general algorithm that, taking as input any algorithm $\mathcal{A}$ and some input data, outputs YES or NO, depending on whether $\mathcal{A}$ halts or continues ad infinitum. Clearly, given a specific algorithm it is usually possible, with more or less ease, to decide whether it will stop for any specific input. However, there is no general algorithm that is able to provide this decision for *any* input algorithm.

Halting problem

Even though undecidability may be interesting from a philosophical point of view, in that it might be considered as a limiting factor to human knowledge, this notion is not a subject of this book, in which we are concerned only with decidable problem classes.

But, even limiting the study to decidable problems, difficulties of another nature come up. These difficulties have been again brought to our attention in recent times by computer science, which stresses a concept that was not previously considered important in mathematics. As already mentioned, mathematical results are achieved by proving theorems or by designing abstract algorithms to solve problems. In computer science this is not sufficient: the algorithm for solving a problem must be *efficient*, i.e., it must run on a computer in *reasonable* time. In order to define what "reasonable" time means, the concept of the *computational complexity* of an algorithm must be introduced.

Efficient algorithms

Given an algorithm $\mathcal{A}$, working on some data, its computational complexity is related to its run time. However, the run time depends on the programming language used to implement the algorithm and on the specific machine on which

[2] In the following we will use, for the sake of simplicity and where no ambiguity may arise, the terms "class of problems" and "problem" interchangeably.

the program is run. In order to make its definition more precise, and independent of the specific implementation, the complexity is evaluated in terms of the number of elementary steps performed by the algorithm and not in terms of the time it takes to execute them. But, even so, there are uncertainties about what has to be considered an elementary "step" in an algorithm, because this depends on the granularity of the observation. To overcome this difficulty an ideal, abstract, computer model is used, for which the notion of a "step" is precisely defined.

Turing machine

There is more than one "ideal" computer, but one of the simplest and best known is the *Turing machine*, an abstract computational device introduced by Alan Turing in the late 1930s (Turing, 1936), long before the first actual computer was built. The simplest version of the Turing machine consists of a tape, a read−write head, and a control unit. The tape, infinite in both directions, is divided into squares which contain a "blank" symbol and at least one other symbol belonging to an alphabet Σ. A square numbered 0 separates the left and right parts of the tape. The head can read or write these symbols onto the tape. The control unit of the machine specifies a finite set of states in which the machine can be; at any point in time a Turing machine is in exactly one of these states. The control unit can be thought of as a *finite state automaton*. This automaton encodes the "program". The computation proceeds in steps: at each step the head reads the content of the square in which it is positioned and, according to this content and the current state of the automaton, it writes another symbol on the same square and then moves one square to the left or to the right. At the beginning the input data are written on the right-hand part of the tape, starting at position 0, and the rest of the tape is filled with "blanks". When the computation is over the machine stops, and the output can be read on the tape.

Notwithstanding the simplicity of its mechanism, the Turing machine is believed to be able to compute any computable algorithm that one can conceive. This assertion was hypothesized by Alonzo Church (1936) through the definition of the λ-*calculus* and the introduction of the notion of effective calculability: a function is said to be effectively calculable if its values can be found by some purely mechanical process. Later, Turing showed that his computation model (the Turing machine) and the λ-calculus are equivalent, so the assertion is now known as *Church−Turing thesis*. This thesis is almost universally accepted now, even though the extent of its applicability is still a subject of debate.

Church−Turing thesis

Implementing an algorithm on a Turing machine allows the notion of computational complexity to be precisely defined. A "step" in an algorithm is a cycle <read a symbol, write a symbol, move the head> and the execution of any program is a sequence of such steps. Given a (decidable) class Π of problems, let $\mathcal{T}$ be the particular Turing machine used to find the solution. If we take an instance $\mathcal{I}$ belonging to Π, let $\mathcal{C}_{\mathcal{T}}(\mathcal{I})$ be the exact number of steps $\mathcal{T}$ uses to solve $\mathcal{I}$. Clearly, however, $\mathcal{C}_{\mathcal{T}}(\mathcal{I})$ is too detailed a measure, impossible to evaluate before

the program is executed. Thus we need a less detailed measure; to this end let n be an integer characterizing the *size* of the problem at hand. For instance, coming back to the sorting problem, n can be the length of the vector to be sorted. We can partition the class Π into subclasses Π_n, each containing only instances $\mathcal{I}_n$ of length n. Even so, running the algorithm on the $\mathcal{I}_n$ requires different numbers of steps, depending on the specific instance. As we want a safe measure of complexity, we take the pessimistic approach of attributing to the subclass Π_n the highest complexity found in the $\mathcal{I}_n$, i.e., the complexity of the *worst case*. Formally, we define the complexity as

Formal definition of computational complexity

$$\mathcal{C}_{\mathcal{T}}(n) = \underset{\mathcal{I}_n \in \Pi_n}{\text{Max}} \{\mathcal{C}_{\mathcal{T}}(\mathcal{I}_n)\}.$$

As it can be proved that the complexity $\mathcal{C}_{\mathcal{T}}(n)$ is independent of the abstract Turing machine used and that it is the same for any concrete computer, we will drop the subscript $\mathcal{T}$ from the complexity and simply write

$$\mathcal{C}_{\mathcal{T}}(n) = \mathcal{C}(n).$$

The complexity $\mathcal{C}(n)$ defined above is called the *time complexity*, because it refers to the time resource consumption of the algorithm. In a similar way, the *space complexity* can be defined as the maximum amount of memory simultaneously occupied during the program's run. In this book, the word "complexity" will always refer to the time complexity unless otherwise specified.

The introduction of the subclasses Π_n is fundamental because the very goal of computational complexity theory is to estimate how the time complexity of an algorithm *scales up* with increasing n. In order to better understand this idea, let us consider two functions f and g from the integers to the integers:

$$f: \ \mathbb{N}^+ \to \mathbb{N}^+, \qquad g: \ \mathbb{N}^+ \to \mathbb{N}^+.$$

What we are interested in is the relative *order of magnitude* of the two functions rather than their actual values. So, let us consider the ratio of $f(n)$ and $g(n)$. There are three cases:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty; \tag{1.1}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \neq 0, \infty; \tag{1.2}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \tag{1.3}$$

In the first case $f(n)$ is of an order of magnitude greater than $g(n)$; in the second case it is of the same order; and in the third case it is of a lower order of magnitude. Introducing the $\mathcal{O}$ ("big O") notation, we will say that $f(n) = \mathcal{O}(g(n))$ in

the second and third cases, namely:

$$f(n) = \mathcal{O}(g(n)) \quad \leftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = a \neq \infty. \tag{1.4}$$

From definition (1.4) we deduce that in $f(n)$ and $g(n)$ all terms of lower orders of magnitude can be omitted, as well as multiplicative constants. It is worth noting explicitly that the $\mathcal{O}$ notation tells us only that $f(n)$ is of order of magnitude *not greater* than that of $g(n)$, not that $f(n)$ and $g(n)$ have the same order in the mathematical sense. If $f(n)$ and $g(n)$ have exactly the same order of magnitude (case (1.2)), we will say that $f(n) = \Theta(g(n))$. If $f(n)$ has order of magnitude strictly greater than $g(n)$, we will write $f(n) = \Omega(g(n))$.

$\mathcal{O}$ notation

— EXAMPLE —

We give here some examples of the $\mathcal{O}$ notation:

$$\begin{aligned} f(n) &= 50n^2 + 20n + 3 = \mathcal{O}(n^2) \\ f(n) &= n^3 + 30n^2 + 100 = \mathcal{O}(n^3) \\ f(n) &= 3^n + 100^3 = \mathcal{O}(3^n) \end{aligned}$$

Given two functions $f(n)$ and $g(n)$, we can provide a formal definition of $f(n) = \mathcal{O}(g(n))$ as follows:

$$f(n) = \mathcal{O}(g(n)) \quad \leftrightarrow \quad \exists n_0 \; \exists c \; [\forall n > n_0 : f(n) \leqslant cg(n)], \tag{1.5}$$

where c is a positive constant and n_0 is a positive integer. Definition (1.5) tells that what happens for low values of n (i.e., lower than n_0) is not important; on the contrary, only the asymptotic behavior of $f(n)$ is relevant. A graphical illustration of definition (1.5) is given in Figure 1.1. We are now in a position to define precisely what a reasonable complexity is. An algorithm $\mathcal{A}$ will be said to be *efficient* if it runs with a complexity that is at most *polynomial* in the size n of the input. Actually, many problems that are relevant in practice show a much more rapid (exponential) increase in the time required to obtain a solution, when the size of the problem increases; such problems cannot be solved within acceptable time spans.

Polynomial complexity

In computer science the study of the computational complexity of algorithms takes a central place; since its beginnings, scientists have studied problems from the complexity point of view, categorizing their behavior into a well-known *complexity class hierarchy*. According to the needs of this book we show, in Figure 1.2, an oversimplified version of this hierarchy, including only three types of problem: **P**, **NP**, and **NP**-complete. The class **P** contains problems that can

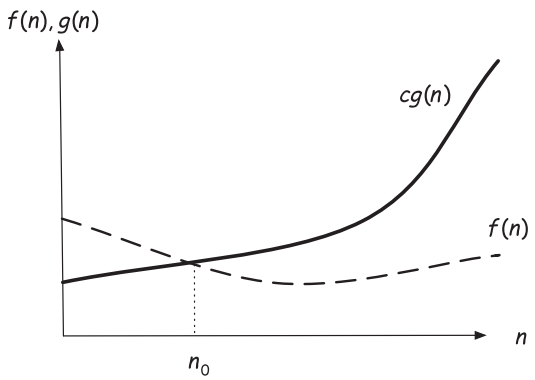

Figure 1.1 Graphical illustration of the $\mathcal{O}$ notation. If $f(n) = \mathcal{O}(g(h))$ then after a given threshold n_0, the function $f(n)$ must be smaller than $cg(n)$, where c is a positive constant. What happens for $n < n_0$ does not matter.

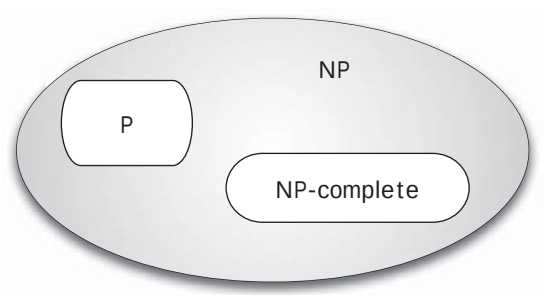

Figure 1.2 Simplified version of the complexity class hierarchy. The class **NP** includes both the class **P** and the class of **NP**-complete problems.

be solved in *polynomial* time by a Turing machine like the one described above, i.e., a *deterministic* Turing machine. In order to define the class **NP** the behavior of a deterministic Turing machine must be extended with a nondeterministic phase, to be executed at the beginning, thus becoming a *non-deterministic* Turing machine. In the non-deterministic phase, a potential solution is generated; this is then verified by the subsequent deterministic phase. The **NP** class contains those problems that can be solved in *polynomial* time by such a *non-deterministic* machine. Whereas the deterministic Turing machine captures the notion of the polynomial *solvability* of a problem class, the non-deterministic Turing machine captures the notion of the polynomial *verifiability* of a problem

class. In other words, if one is able to suggest a potential solution to a problem, the non-deterministic Turing machine can verify in polynomial time whether it is indeed a solution. It should be clear from the definition that $\mathbf{P}$ is a subclass of $\mathbf{NP}$: polynomial solvability implies polynomial verifiability.

$\mathbf{P} = \mathbf{NP}$?

Today, the question whether $\mathbf{P} = \mathbf{NP}$ is still an important open problem in computer science. On the one hand it has not been possible to prove that $\mathbf{P} = \mathbf{NP}$ and on the other hand no polynomial algorithm has been found for many problems in $\mathbf{NP}$, notwithstanding the amount of effort devoted to the task. Thus, the general opinion is that $\mathbf{P} \neq \mathbf{NP}$. This opinion is strongly supported by the existence of the subclass $\mathbf{NP}$-complete. This subclass contains (a large number of) problems that share the following property: each problem in $\mathbf{NP}$ (including $\mathbf{P}$) can be reduced in polynomial time to any problem in $\mathbf{NP}$-complete; as a consequence, it would be sufficient to solve in polynomial time just one problem in $\mathbf{NP}$-complete to prove that $\mathbf{P} = \mathbf{NP}$. Given the amount of work already devoted to this task, it seems highly unlikely that this will turn out to be the case. In this book we will assume as an underlying hypothesis that $\mathbf{P} \neq \mathbf{NP}$.

Combinatorial problems

A class of problems that are particularly prone to a dramatic increase in computational complexity with increasing problem size is the class of *combinatorial* problems, many among which are $\mathbf{NP}$-complete. Informally, a combinatorial problem is one that requires combinations of objects belonging to a set to be explored, with the goal of deciding whether a specified property holds true or of finding some optimal combination according to a specified criterion. Combinatorial problems are well represented in artificial intelligence, operational research, complex system analysis, and optimization and search. Among the large variety of existing combinatorial problems, two have received a great deal of attention, namely the *satisfiability* (SAT) problem (Cook, 1971) and the *constraint satisfaction problem* (CSP) (see for instance Kumar, 1992). In Chapters 3 and 4 these two problems will be introduced and described in detail, because they are intimately related to machine learning and to the sources of its complexity.

Typical complexity

Reconsidering the definition of computational complexity provided earlier, it is not necessarily a good idea to take the worst-case complexity as that representative of an algorithm $\mathcal{A}$. In fact, $\mathcal{A}$ might be able to provide a solution in reasonable time for the majority of the instances in Π_n, running for a very long time in only a few particular instances; this is the case, for example, for the branch-and-bound optimization algorithm (Lawler and Wood, 1966). For this reason a new paradigm has emerged, which uses the *typical* running behavior of an algorithm instead of its worst case. The notion of the typical complexity has a precise meaning. Namely, it requires two conditions:

- The typical complexity is the most probable complexity over the class of problem instances considered.

- As each problem instance has its own run time, there is a difference in complexity between that for the specific instance and the most probable complexity. When the size of the considered instances grows to infinity, the difference between their complexity and the most probable complexity must go to 0 with probability 1.

This new perspective on the complexity of algorithms was suggested by the discovery of interesting and fruitful links (previously unsuspected) between combinatorial problems and systems obeying the laws of statistical physics. It turns out that combinatorial problems share several characteristics with physical systems composed of a large number of particles and that a precise parallel can be established between such physical entities on the one hand and combinatorial functions to be optimized on the other hand.

Phase transitions

Among the many parallels that can be drawn from the link between such many-body physical systems and computational systems, one aspect is particularly relevant for this book, namely, the emergence of a *phase transition* (see for instance Hogg, 1996). Some physical systems composed of a large number of particles may exist in different *phases*. A phase is a homogeneous (with respect to some specified physical quantity) state of the system. A well-known case in everyday life is water, which can exist in solid, liquid, and gaseous phases. The phase that water is in depends on the values of the macroscopic variables describing the physical state, for instance, the temperature and pressure. In Figure 1.3 a qualitative schema of the phases in which water may exist is shown. In a phase transition we distinguish between the order and control parameters: an *order* parameter is a quantity that shows a marked difference in behavior across the transition line whereas a *control* parameter is one that determines the location of the transition. In the case of water, a possible order parameter is the density whereas a possible control parameter is the temperature. The order and control parameters characterize the phase transition.

Order and control parameters

Types of phase transitions

According to Ehrenfest's classification, there are two types of phase transition, namely *first-order* and *second-order*. A precise definition of these types will be given in Chapter 2. We just mention, here, that we are interested in first-order phase transitions; in this type of transition, in addition to the discontinuity in the order parameters, there is usually another quantity that goes to infinity when the size of the system tends to infinity as well. Moreover, at the transition point the two phases coexist. In the case of water, for example, the specific heat diverges at the transition between liquid and vapor, because heat is being supplied to the system but the temperature remains constant.

In computational problems the order parameters are usually quantities that characterize aspects of the algorithm's behavior (for instance, the probability that a solution exists) and the control parameters describe the internal structure

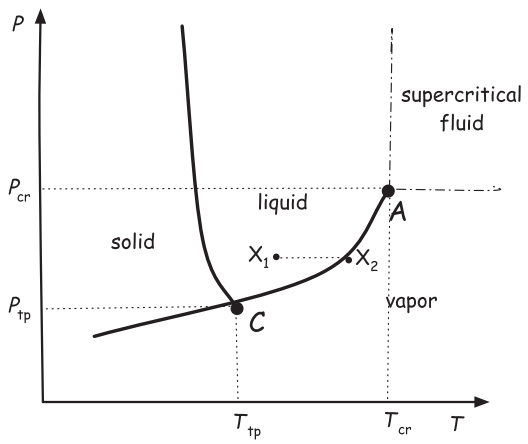

Figure 1.3 Qualitative diagrams of the phases in which water can exist. The temperature T and pressure P are the control parameters of the transition between phases. Along the separation lines two of the phases coexist. At the triple point C all three phases are present. Beyond the critical point A the water is said to be in a supercritical fluid state. In this state the molecules are too energetic and too close to each other for a clear transition between liquid and vapor to exist.

of the problem, whereas the quantity that diverges at a "phase transition" is the computational complexity of the algorithm.

There are various motivations for studying the emergence of phase transitions. First, their emergence seems to be an ubiquitous phenomenon in many-body systems, capturing some essential properties of their nature. They occur not only in physical and computational systems but also in human perception and social sciences, as will be described later in this book. Second, systems that show a phase transition exhibit, at the transition point, interesting singularities in behavior called "critical phenomena", which elucidate their real essence, in a way not evident by other means. Third, phase transitions are interesting in themselves, as they explain ensemble or macroscopic behaviors in terms of short-range microscopic interactions.

For computational systems the discovery of a phase transition in a problem class has several important consequences. The phase transition region contains the most difficult problem instances, those for which the computational complexity shows an exponential increase with the problem size. Also, the phase transition can be used as a source of "difficult" test problems for assessing the properties and the power of algorithms and for comparing them in meaningful problem instances. Moreover, very small variations in the control parameter