1

Getting Started

In the present chapter we introduce the basic notions necessary to study learning problems within the framework of statistical mechanics. We also demonstrate the efficiency of learning from examples by the numerical analysis of a very simple situation. Generalizing from this example we will formulate the basic setup of a learning problem in statistical mechanics to be discussed in numerous modifications in later chapters.

1.1 Artificial neural networks

The statistical mechanics of learning has been developed primarily for networks of so-called *formal neurons*. The aim of these networks is to model some of the essential information processing abilities of biological neural networks on the basis of artificial systems with a similar architecture. Formal neurons, the microscopic building blocks of these artificial neural networks, were introduced more than 50 years ago by McCulloch and Pitts as extremely simplified models of the biological neuron [1]. They are bistable linear threshold elements which are either *active* or *passive*, to be denoted in the following by a binary variable $S = \pm 1$. The state S_i of a given neuron i changes with time because of the signals it receives through its *synaptic couplings* J_{ij} from either the "outside world" or other neurons j.

More precisely, neuron i sums up the incoming activity of all the other neurons weighted by the corresponding synaptic coupling strengths to yield the *post-synaptic potential* $\sum_j J_{ij} S_j$ and compares the result with a threshold θ_i specific to neuron i. If the post-synaptic potential exceeds the threshold, the neuron will be active in the next time step, otherwise it will be passive

$$S_i(t+1) = \operatorname{sgn}\left(\sum_j J_{ij} S_j(t) - \theta_i \right), \tag{1.1}$$

where the sign function is defined by $\mathrm{sgn}(x) = 1$ if $x > 0$ and $\mathrm{sgn}(x) = -1$ otherwise.

The McCulloch–Pitts neuron is clearly an extreme oversimplification of its biological prototype. In fact, for every objection raised against it, it is easy to find an additional one. However, the emphasis in statistical mechanics of neural networks is on issues that are complementary to those of neurophysiology. Instead of focusing on the single neuron, the main objectives are the *collective* properties that emerge in *large* assemblies of neurons. Previous experience with complex *physical* systems such as magnets, liquid crystals and superfluids has shown that often these collective properties are surprisingly insensitive to many of the microscopic details and thus the use of extremely simplified models for the constituents is often appropriate to describe the macroscopic properties of the system. A central hypothesis in the statistical mechanics of learning is hence that learning from examples *is* such a collective emerging property and that it can be studied in large networks of McCulloch–Pitts neurons.

There are several ways of connecting formal neurons to create a network, characterized by the connectivity graph of the synaptic matrix J_{ij}. Figure 1.1 shows some simple possibilities for small systems.

A mathematical analysis is, however, possible for some extreme architectures only. Two types of connectivities will be of special interest. In the first one every neuron is connected with every other neuron, see fig. 1.1b. The dynamics (1.1) is then highly recurrent and will in general result in a chaotic sequence of different activity patterns of the neurons. For a suitably chosen set of couplings J_{ij}, however, the situation can be much simpler. Consider, e.g., the case of symmetric couplings $J_{ij} = J_{ji}$. It is then easy to show that the function

$$H(\mathbf{S}) = -\sum_{i,j} J_{ij} S_i S_j \tag{1.2}$$

with $\mathbf{S} = \{S_i\}$ denoting the complete vector of neuron activities, can never *increase* under the dynamics (1.1). Since $H(\mathbf{S})$ cannot become arbitrarily small, the system will eventually approach configurations which minimize H and then remain in these *attractor states*. In fact, by suitably choosing the couplings J_{ij} these attractors can be *prescribed* to be certain desired neuron configurations $\boldsymbol{\xi}^\mu = \{\xi_1^\mu, \ldots, \xi_N^\mu\}$, $\xi_i^\mu = \pm 1$. The index μ labels the different coexisting attractor states and runs from 1 to p. If the network is now initialized in some configuration $\mathbf{S}$ which is similar to one of these embedded patterns, $\boldsymbol{\xi}^1$ say, the dynamics will in most cases tend to this attractor and thereby *restore* the complete pattern $\boldsymbol{\xi}^1$. In this sense, attractor neural networks may function as *associative memories* which are able to retrieve a stored pattern $\boldsymbol{\xi}^1$ if initially stimulated by a noisy or incomplete variant $\mathbf{S}$ of this pattern. The parallel and distributed character

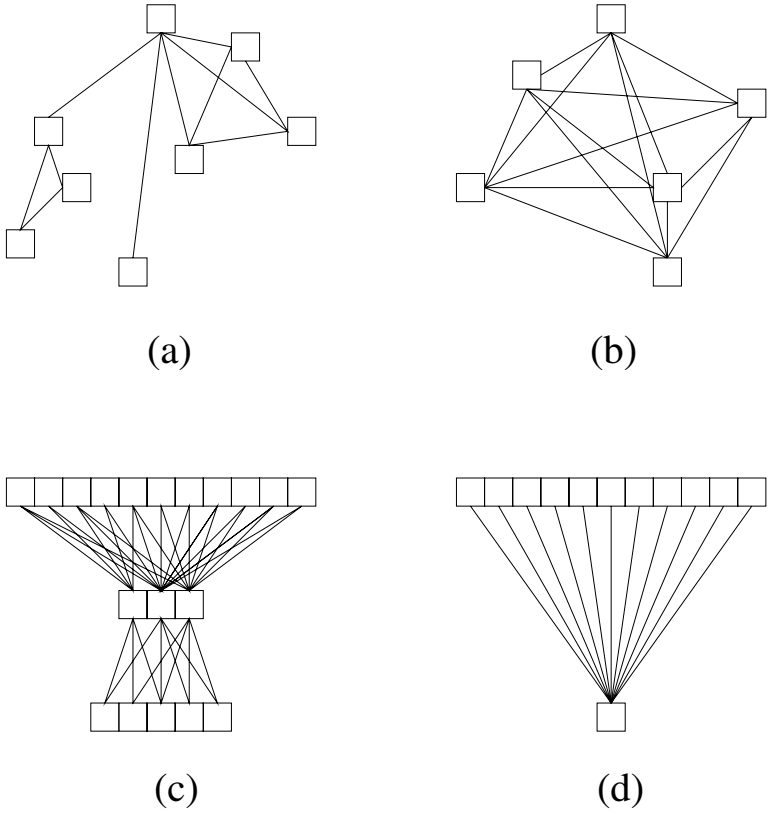

Fig. 1.1. Different types of networks of formal neurons. (a) general architecture, (b) fully connected attractor neural network, (c) feed-forward network with one hidden layer, (d) single layer perceptron.

of such a memory is rather reminiscent of the human brain and very different from present-day electronic memory devices. Central questions in the statistical mechanics of attractor neural networks concern the maximal possible number p_c of patterns $\boldsymbol{\xi}^\mu$ that can be stored, the properties of different learning rules fixing the values of the synaptic couplings as functions of the stored patterns, the typical basins of attraction quantifying the maximally admissible difference of the stimulus from the desired pattern, and the interference of different patterns in the retrieval process. There are several textbooks dealing with the statistical mechanics analysis of these systems [2, 3, 4, 5].

The other extreme type of architecture, called *feed-forward neural network*, is shown in fig. 1.1c. In such a network, the neurons can be arranged in layers $l = 1, \ldots, L$ such that every neuron in layer l only receives inputs from neurons of layer $(l-1)$ and in turn only feeds neurons in layer $(l+1)$. The first layer, $l = 1$,

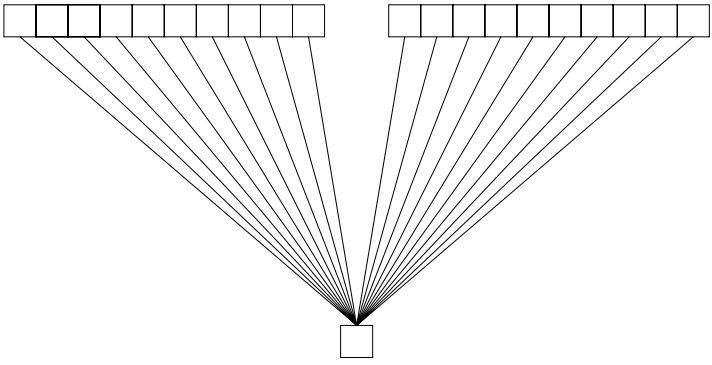

Fig. 1.2. Simple perceptron used to rank dual numbers.

is called the *input* layer; the last one, $l = L$; the *output* layer; and all layers with $1 < l < L$ are referred to as *hidden* layers.

Due to the absence of feedback loops the dynamics is very simple. The input is mapped to the output via successive time steps according to (1.1). The network therefore performs a *classification* of the input strings into classes labelled by the different configurations of the output layer. This architecture is well suited for learning from examples. In particular the simplest feed-forward neural net, the perceptron, having no hidden layers at all, as shown in fig. 1.1d, can be analysed in great detail. Its introduction by Rosenblatt in 1962 [6] initiated a first period of euphoria about artificial neural networks. The attractive features of the perceptron as a simple model for some basic cognitive abilities of the human brain also stimulated some speculations widely overestimating its relevance. So this period ended surprisingly abruptly in 1969 when some rather obvious limitations of the perceptron were clearly stated [7].

Nevertheless the perceptron is a very interesting and valuable model system. In the statistical mechanics of learning it has become a kind of "hydrogen atom" of the field and hence it will often be the focus of attention in this book. In later chapters we will discuss the application of the techniques developed for the perceptron to the more powerful *multilayer networks* having one or more hidden layers.

1.2 A simple example

It is certainly appropriate to introduce learning from examples by discussing a simple example. Consider the perceptron shown in fig. 1.2. It has $N = 20$ input units S_i each connected directly to the single output σ by real valued couplings J_i.

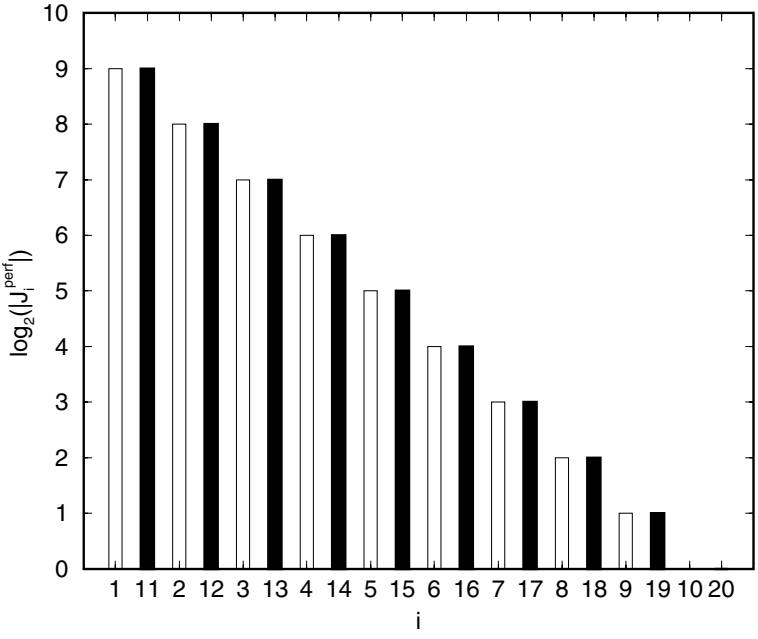

Fig. 1.3. Graphical representation of the perfect couplings for a perceptron to rank dual numbers as given by (1.4). $J_1, \ldots, J_{10}$ (white) are positive, $J_{11}, \ldots, J_{20}$ (black) are negative, cf. (1.4).

For any input vector $\mathbf{S}$ the output is determined by the rule

$$\sigma = \mathrm{sgn}\left(\sum_i J_i S_i\right), \tag{1.3}$$

which is a special case of (1.1).

We would like to use the network to rank 10-digit dual numbers.[1] To this end we require the output to be $+1$ (-1) if the dual number represented by the left ten input bits is larger (smaller) than the one given by the right ten inputs. For simplicity we ignore for the moment the possibility of the two numbers being equal.

It is easy to construct a set of couplings that does the job perfectly. Consider the coupling values

$$\begin{aligned} J_i^{\mathrm{perf}} &= 2^{10-i} \quad \text{if} \quad i = 1, \ldots, 10 \\ J_i^{\mathrm{perf}} &= -J_{i-10}^{\mathrm{perf}} \quad \text{if} \quad i = 11, \ldots, 20, \end{aligned} \tag{1.4}$$

displayed also in fig. 1.3. This choice gives, as it should, a larger weight in the superposition (1.3) to the leftmost bits in the two subfields of the input. On the other hand it ensures that less significant bits are able to tip the balance if the first

[1] A 3-digit dual number with dual code $(-1, 1, -1)$ is equal to $0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$.

bits of the two numbers coincide. The above problem is simple enough for us to guess the appropriate values of the couplings. Doing so is an example of explicit programming as used in almost all present-day computers.

However, we can apply another, more interesting procedure to solve the problem, namely learning from examples. Let us first initialize the couplings J_i at random. We then select, out of the total of $2^{20} \simeq 10^6$ different input strings, a given number p of input vectors $\boldsymbol{\xi}^\mu$, $\mu = 1, \ldots, p$ at random and for each case provide the *correct output*, which we denote by σ_T^μ. Next, we *train* the network with this set $\{\boldsymbol{\xi}^\mu, \sigma_T^\mu\}$. To this end we sequentially present each of the input vectors to the network and verify whether the resulting network output σ^μ given through (1.3) is correct, i.e. coincides with σ_T^μ. If so, which will initially happen for roughly half of the cases, we simply proceed to the next example. If however $\sigma^\mu \neq \sigma_T^\mu$ we modify the couplings in such a way that the example under consideration is less likely to be misclassified upon the next presentation. Various rules to achieve this goal will be presented in chapter 3.[2] We iterate this procedure until all examples of the training set are reproduced correctly. The fact that the procedure converges is *a priori* not obvious, but it does so for the problem under consideration: ranking numbers is a *learnable* problem for the perceptron.

The success on the training set, however, does not tell us whether the network has really learned the *rule* behind the examples. To answer this question the performance on *so far unseen* inputs has to be investigated.[3] A quantitative measure of the degree of generalization from the examples to the rule can be obtained by determining the fraction of wrong outputs when running through the *complete* set of 2^{20} different inputs. This fraction is called the generalization error ε and is one of the central quantities in the analysis of learning problems.

Figure 1.4 shows ε as a function of the size p of the training set as resulting from simulations as described above. Note that ε is a random variable which depends on the particular choice of the training set. In fig. 1.4, we have reproduced the average over 1000 random realizations of the training set.

The general behaviour is as expected. For $p = 0$ the network has no information at all about the target rule. By chance half of the examples are classified correctly, $\varepsilon = 0.5$, which is the known success rate for pure guessing. With increasing p the generalization error decreases monotonically and for $p \to \infty$ it must, of course, vanish. However, the surprising fact is that the generalization error already becomes rather small for p of the order of a few hundred, which is *much less* than the total number of different input vectors! In other words, the network is able to generalize rather well in the sense that it can approximate the desired rule on the basis of a very limited set of examples.

[2] In the simulations shown below we used the randomized perceptron learning rule discussed in section 3.2.
[3] This is also the reason why exercises are to be found at the end of most chapters of this book.

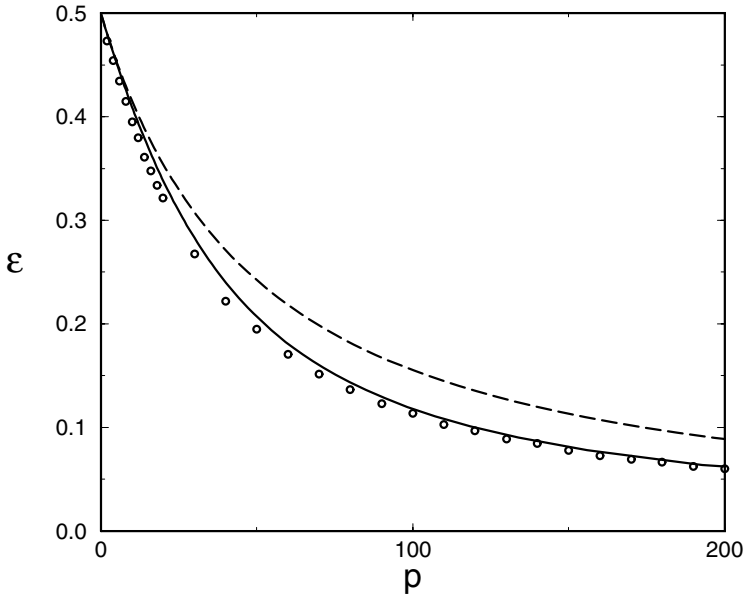

Fig. 1.4. Simulation results (circles) for the generalization error of a perceptron learning from examples to rank dual numbers. The results are averaged over 1000 realizations of the training set, and the statistical error is smaller than the symbol size. The full line gives the analytic result of the quenched calculation, the dashed line that of the annealed approximation. Both are discussed in detail in chapter 2.

In a similar way, one can show that a somewhat more complicated network made of Boolean gates is able to learn the addition of numbers from examples [8]. Another striking demonstration of learning from examples in artificial neural networks is the ability of a multilayer neural net to read English text aloud [9], and many more examples have been documented [10].

At first sight it may seem somewhat enigmatic that a system as simple as the perceptron should be "intelligent enough" to decipher a rule behind examples. Nevertheless the explanation is rather simple: the perceptron can only implement a very limited set of mappings between input and output, and the ranking of numbers happens to be one of them. Given this limitation it is therefore comparatively easy to select the proper mapping on the basis of examples. These rather vague statements will be made more precise in the following chapters.

To get a more concrete idea of how the perceptron proceeds in the above problem, it is instructive to look at the evolution of the couplings J_i as a function of the size p of the training set. In fig. 1.5 the couplings are shown for $p = 50$ and $p = 200$. In both cases we have normalized them such that $J_1 = 2^9$ in order to facilitate comparison with the target values given in (1.4) and fig. 1.3. As one easily

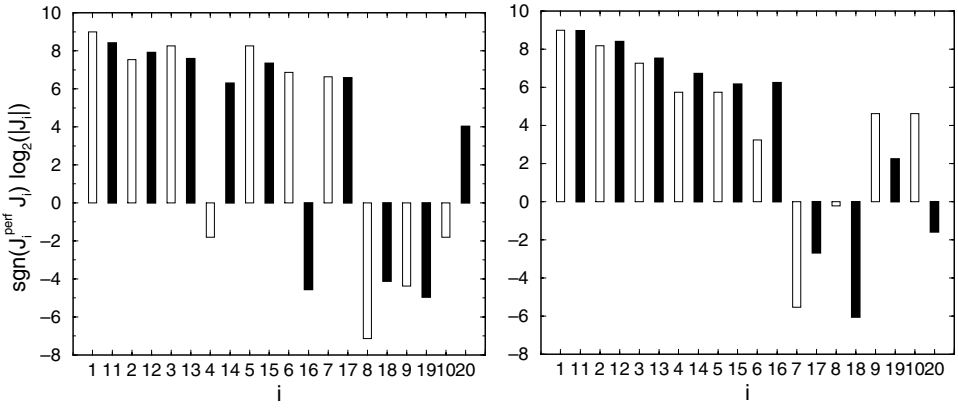

Fig. 1.5. Graphical representation of the perceptron couplings after learning 50 (left) and 200 (right) examples. The signs of the columns indicate whether the couplings have the same sign as the perfect couplings of fig. 1.3 or not.

realizes, the relation between the most important couplings J_1, J_2, J_3, J_{11}, J_{12}, J_{13} is fixed first. This is because they decide the output for the large majority of input patterns, both in the training and in the complete set. Considering that correct values for J_1, J_2, J_3, J_{11}, J_{12} and J_{13} yield a correct output for $15/16$ of all patterns already, one understands how the initial efficiency of the learning process is achieved. By the same token, one expects that inputs which give information about the couplings J_9, J_{10}, J_{19} and J_{20} are rare, with a rather slow asymptotic decay of the generalization error to zero as a result.

We have included in fig. 1.4 the generalization error as obtained from two analytic treatments of the learning problem within the framework of statistical mechanics (an approximate one called "annealed" and the exact one referred to as "quenched"). The concepts and techniques necessary to produce curves such as these are the main subject of this book.

1.3 General setup

We are now ready to formulate the basic scenario for learning problems in the statistical mechanics of artificial neural networks. To this end we consider a feed-forward network of formal neurons with N input units denoted by $S_i, i = 1, \ldots, N$ and one output σ. We restrict ourselves to networks with a single output unit merely for simplicity. Since there are no couplings between neurons in the same layer, networks with several outputs are not significantly more complex than those with a single one.

The input–output mapping performed by the network is specified by a vector $\mathbf{J} = \{J_1, \ldots, J_N\}$ of synaptic couplings. The network is required to *adapt* these couplings in order to perform a certain target mapping which will be generally referred to as the *rule*. It is often convenient to think of the target mapping as being represented by another feed-forward neural net characterized by a synaptic vector $\mathbf{T} = \{T_1, \ldots, T_N\}$. Although the *teacher* network $\mathbf{T}$ and the *student* network $\mathbf{J}$ must, of course, have the same number N of inputs and a single output they may, in general, differ in their architecture.

In his task to approximate the teacher neither the detailed architecture nor the components of the teacher vector $\mathbf{T}$ are known to the student. The only accessible information about the target rule is contained in the *training set* composed of p inputs $\boldsymbol{\xi}^\mu$, $\mu = 1, \ldots, p$ with $\boldsymbol{\xi}^\mu = \{\xi_i^\mu = \pm 1,\ i = 1, \ldots, N\}$ and their corresponding outputs $\sigma_T^\mu = \pm 1$, $\mu = 1, \ldots, p$ provided by the teacher. The prescription $\{\boldsymbol{\xi}^\mu, \sigma_T^\mu\} \mapsto \mathbf{J}$ which specifies a suitable student coupling vector $\mathbf{J}$ on the basis of the training set is called a *learning rule*. The most obvious requirement for a learning rule is to generate a student vector which approximates the teacher as well as possible. But there are also other important features such as the time needed for the determination of $\mathbf{J}$ and the flexibility with respect to the incorporation of new inputs added to the training set.

A crucial question is how the examples of the training set are selected. This is quite important since different training sets of the same size may well convey a different amount of information about the target rule. In many practical situations one cannot design the training set at will since its elements are determined by some experimental procedure. In order to model these situations one therefore assumes that the examples of the training set are selected *independently at random* according to some probability distribution $P_{\mathbf{S}}(\mathbf{S})$ defined on the input space. Throughout this book we will use simple distributions such as

$$P_{\mathbf{S}}(\mathbf{S}) = \prod_i \left[\frac{1}{2}\delta(S_i + 1) + \frac{1}{2}\delta(S_i - 1)\right], \tag{1.5}$$

which implies that the individual components S_i of the inputs are ± 1 with equal probability and independently of each other. Some properties of the δ-function $\delta(x)$ are summarized in appendix 1.

Analogously to considering a training set compiled at random it is often sensible not to concentrate on *one specific* target rule $\mathbf{T}$ but to assume that also the target is chosen at random from a *rule space* according to some probability distribution $P_{\mathbf{T}}(\mathbf{T})$. The results of the analysis will then characterize a whole class of learning problems rather than singling out the performance on one particular task.

Finally, many learning rules involve random elements in order to facilitate convergence. As a result, the training set leads not to a unique $\mathbf{J}$-vector but to

a whole distribution $P_{\mathbf{J}}(\mathbf{J})$. The learning process is then conveniently described as a reshaping of this probability distribution on the basis of the information gained from the examples [11, 12].

In conclusion there are several sources of randomness in a generic learning problem and the different ways of dealing with this probabilistic nature result in different approaches to a mathematical analysis of learning.

In order to quantify the success of the student in approximating the teacher we first need a measure of similarity or dissimilarity between two networks. Let us present the same input vector $\mathbf{S}$ to both networks and denote by σ_T and σ the corresponding output of the teacher and the student respectively. For binary outputs the quantity

$$d(\mathbf{J}; \mathbf{S}, \mathbf{T}) = \theta(-\sigma_T \sigma) \tag{1.6}$$

is an appropriate distance measure since with the θ-function defined by $\theta(x) = 1$ if $x > 0$ and $\theta(x) = 0$ otherwise (cf. (A1.17)) it is 1 in case of an error (output different from target output) and 0 otherwise.[4] We may then quantify the performance of the student on the training set by calculating the so-called *training error*

$$\varepsilon_t(\mathbf{J}; \{\boldsymbol{\xi}^\mu\}, \mathbf{T}) = \frac{1}{p} \sum_{\mu=1}^{p} d(\mathbf{J}; \boldsymbol{\xi}^\mu, \mathbf{T}) \tag{1.7}$$

which just gives the fraction of misclassified patterns of the training set. The training error is an important ingredient of most learning rules and we will see in later chapters that it is often (but not always) a good idea to choose a student vector with small training error.

On the other hand the training error is clearly not suitable for determining how well the student really approximates the target rule. This is measured by the *generalization error* $\varepsilon(\mathbf{J}; \mathbf{T})$, which is defined as the *average* of d over the *whole set* of possible inputs $\mathbf{S}$

$$\varepsilon(\mathbf{J}; \mathbf{T}) = \sum_{\{\mathbf{S}\}} P_{\mathbf{S}}(\mathbf{S})\, d(\mathbf{J}; \mathbf{S}, \mathbf{T}). \tag{1.8}$$

Equivalently we may interpret ε as the probability that the student classifies a *randomly drawn* input differently from the teacher. It is sometimes also referred to as "probability of mistake" or "expected 0–1 loss". Note that it is natural to sample the test input from the *same* probability distribution $P_{\mathbf{S}}$ as the training examples.

From the definition of ε_t and ε it is clear that the problem of learning from examples shares some features with a standard problem in mathematical statistics,

[4] For continuous outputs $d(\mathbf{J}; \mathbf{S}, \mathbf{T}) = (\sigma_T - \sigma)^2/4$ is a suitable generalization.