8 The Runtime of Learning

So far in the book we have studied the statistical perspective of learning, namely, how many samples are needed for learning. In other words, we focused on the amount of information learning requires. However, when considering automated learning, computational resources also play a major role in determining the complexity of a task: that is, how much *computation* is involved in carrying out a learning task. Once a sufficient training sample is available to the learner, there is some computation to be done to extract a hypothesis or figure out the label of a given test instance. These computational resources are crucial in any practical application of machine learning. We refer to these two types of resources as the *sample complexity* and the *computational complexity*. In this chapter, we turn our attention to the computational complexity of learning.

The computational complexity of learning should be viewed in the wider context of the computational complexity of general algorithmic tasks. This area has been extensively investigated; see, for example, (Sipser 2006). The introductory comments that follow summarize the basic ideas of that general theory that are most relevant to our discussion.

The actual runtime (in seconds) of an algorithm depends on the specific machine the algorithm is being implemented on (e.g., what the clock rate of the machine's CPU is). To avoid dependence on the specific machine, it is common to analyze the runtime of algorithms in an asymptotic sense. For example, we say that the computational complexity of the merge-sort algorithm, which sorts a list of n items, is $O(n \log(n))$. This implies that we can implement the algorithm on any machine that satisfies the requirements of some accepted abstract model of computation, and the actual runtime in seconds will satisfy the following: there exist constants c and n_0, which can depend on the actual machine, such that, for any value of $n > n_0$, the runtime in seconds of sorting any n items will be at most $c\, n \log(n)$. It is common to use the term *feasible* or *efficiently computable* for tasks that can be performed by an algorithm whose running time is $O(p(n))$ for some polynomial function p. One should note that this type of analysis depends on defining what is the input size n of any instance to which the algorithm is expected to be applied. For "purely algorithmic" tasks, as discussed in the common computational complexity literature, this input size is clearly defined; the algorithm gets an input instance, say, a list to be sorted, or an arithmetic operation to be calculated, which has a well defined size (say, the

number of bits in its representation). For machine learning tasks, the notion of an input size is not so clear. An algorithm aims to detect some pattern in a data set and can only access random samples of that data.

We start the chapter by discussing this issue and define the computational complexity of learning. For advanced students, we also provide a detailed formal definition. We then move on to consider the computational complexity of implementing the ERM rule. We first give several examples of hypothesis classes where the ERM rule can be efficiently implemented, and then consider some cases where, although the class is indeed efficiently learnable, ERM implementation is computationally hard. It follows that hardness of implementing ERM does not imply hardness of learning. Finally, we briefly discuss how one can show hardness of a given learning task, namely, that no learning algorithm can solve it efficiently.

8.1 Computational Complexity of Learning

Recall that a learning algorithm has access to a domain of examples, Z, a hypothesis class, $\mathcal{H}$, a loss function, ℓ, and a training set of examples from Z that are sampled i.i.d. according to an unknown distribution $\mathcal{D}$. Given parameters ϵ, δ, the algorithm should output a hypothesis h such that with probability of at least $1 - \delta$,

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon.$$

As mentioned before, the actual runtime of an algorithm in seconds depends on the specific machine. To allow machine independent analysis, we use the standard approach in computational complexity theory. First, we rely on a notion of an abstract machine, such as a Turing machine (or a Turing machine over the reals (Blum, Shub & Smale 1989)). Second, we analyze the runtime in an asymptotic sense, while ignoring constant factors, thus the specific machine is not important as long as it implements the abstract machine. Usually, the asymptote is with respect to the size of the input to the algorithm. For example, for the merge-sort algorithm mentioned before, we analyze the runtime as a function of the number of items that need to be sorted.

In the context of learning algorithms, there is no clear notion of "input size." One might define the input size to be the size of the training set the algorithm receives, but that would be rather pointless. If we give the algorithm a very large number of examples, much larger than the sample complexity of the learning problem, the algorithm can simply ignore the extra examples. Therefore, a larger training set does not make the learning problem more difficult, and, consequently, the runtime available for a learning algorithm should not increase as we increase the size of the training set. Just the same, we can still analyze the runtime as a function of natural parameters of the problem such as the target accuracy, the confidence of achieving that accuracy, the dimensionality of the

domain set, or some measures of the complexity of the hypothesis class with which the algorithm's output is compared.

To illustrate this, consider a learning algorithm for the task of learning axis aligned rectangles. A specific problem of learning axis aligned rectangles is derived by specifying ϵ, δ, and the dimension of the instance space. We can define a sequence of problems of the type "rectangles learning" by fixing ϵ, δ and varying the dimension to be $d = 2, 3, 4, \ldots$. We can also define another sequence of "rectangles learning" problems by fixing d, δ and varying the target accuracy to be $\epsilon = \frac{1}{2}, \frac{1}{3}, \ldots$. One can of course choose other sequences of such problems. Once a sequence of the problems is fixed, one can analyze the asymptotic runtime as a function of variables of that sequence.

Before we introduce the formal definition, there is one more subtlety we need to tackle. On the basis of the preceding, a learning algorithm can "cheat," by transferring the computational burden to the output hypothesis. For example, the algorithm can simply define the output hypothesis to be the function that stores the training set in its memory, and whenever it gets a test example x it calculates the ERM hypothesis on the training set and applies it on x. Note that in this case, our algorithm has a fixed output (namely, the function that we have just described) and can run in constant time. However, learning is still hard – the hardness is now in implementing the output classifier to obtain a label prediction. To prevent this "cheating," we shall require that the output of a learning algorithm must be applied to predict the label of a new example in time that does not exceed the runtime of training (that is, computing the output classifier from the input training sample). In the next subsection the advanced reader may find a formal definition of the computational complexity of learning.

8.1.1 Formal Definition*

The definition that follows relies on a notion of an underlying abstract machine, which is usually either a Turing machine or a Turing machine over the reals. We will measure the computational complexity of an algorithm using the number of "operations" it needs to perform, where we assume that for any machine that implements the underlying abstract machine there exists a constant c such that any such "operation" can be performed on the machine using c seconds.

DEFINITION 8.1 (The Computational Complexity of a Learning Algorithm) We define the complexity of learning in two steps. First we consider the computational complexity of a fixed learning problem (determined by a triplet $(Z, \mathcal{H}, \ell)$ – a domain set, a benchmark hypothesis class, and a loss function). Then, in the second step we consider the rate of change of that complexity along a sequence of such tasks.

1. Given a function $f : (0,1)^2 \to \mathbb{N}$, a learning task $(Z, \mathcal{H}, \ell)$, and a learning algorithm, $\mathcal{A}$, we say that $\mathcal{A}$ solves the learning task in time $O(f)$ if there exists some constant number c, such that for every probability distribution $\mathcal{D}$

over Z, and input $\epsilon,\ \delta \in (0,1)$, when $\mathcal{A}$ has access to samples generated i.i.d. by $\mathcal{D}$,

- $\mathcal{A}$ terminates after performing at most $cf(\epsilon,\ \delta)$ operations
- The output of $\mathcal{A}$, denoted $h_{\mathcal{A}}$, can be applied to predict the label of a new example while performing at most $cf(\epsilon,\ \delta)$ operations
- The output of $\mathcal{A}$ is probably approximately correct; namely, with probability of at least $1-\delta$ (over the random samples $\mathcal{A}$ receives), $L_{\mathcal{D}}(h_{\mathcal{A}}) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon$

2. Consider a sequence of learning problems, $(Z_n, \mathcal{H}_n, \ell_n)_{n=1}^{\infty}$, where problem n is defined by a domain Z_n, a hypothesis class $\mathcal{H}_n$, and a loss function ℓ_n.

   Let $\mathcal{A}$ be a learning algorithm designed for solving learning problems of this form. Given a function $g : \mathbb{N} \times (0,1)^2 \to \mathbb{N}$, we say that the runtime of $\mathcal{A}$ with respect to the preceding sequence is $O(g)$, if for all n, $\mathcal{A}$ solves the problem $(Z_n, \mathcal{H}_n, \ell_n)$ in time $O(f_n)$, where $f_n : (0,1)^2 \to \mathbb{N}$ is defined by $f_n(\epsilon, \delta) = g(n, \epsilon, \delta)$.

We say that $\mathcal{A}$ is an *efficient* algorithm with respect to a sequence $(Z_n, \mathcal{H}_n, \ell_n)$ if its runtime is $O(p(n, 1/\epsilon, 1/\delta))$ for some polynomial p.

From this definition we see that the question whether a general learning problem can be solved efficiently depends on how it can be broken into a sequence of specific learning problems. For example, consider the problem of learning a finite hypothesis class. As we showed in previous chapters, the ERM rule over $\mathcal{H}$ is guaranteed to (ϵ, δ)-learn $\mathcal{H}$ if the number of training examples is order of $m_{\mathcal{H}}(\epsilon, \delta) = \log(|\mathcal{H}|/\delta)/\epsilon^2$. Assuming that the evaluation of a hypothesis on an example takes a constant time, it is possible to implement the ERM rule in time $O(|\mathcal{H}|\, m_{\mathcal{H}}(\epsilon, \delta))$ by performing an exhaustive search over $\mathcal{H}$ with a training set of size $m_{\mathcal{H}}(\epsilon, \delta)$. For any fixed finite $\mathcal{H}$, the exhaustive search algorithm runs in polynomial time. Furthermore, if we define a sequence of problems in which $|\mathcal{H}_n| = n$, then the exhaustive search is still considered to be efficient. However, if we define a sequence of problems for which $|\mathcal{H}_n| = 2^n$, then the sample complexity is still polynomial in n but the computational complexity of the exhaustive search algorithm grows exponentially with n (thus, rendered inefficient).

8.2 Implementing the ERM Rule

Given a hypothesis class $\mathcal{H}$, the $\text{ERM}_{\mathcal{H}}$ rule is maybe the most natural learning paradigm. Furthermore, for binary classification problems we saw that if learning is at all possible, it is possible with the ERM rule. In this section we discuss the computational complexity of implementing the ERM rule for several hypothesis classes.

Given a hypothesis class, $\mathcal{H}$, a domain set Z, and a loss function ℓ, the corresponding $\text{ERM}_{\mathcal{H}}$ rule can be defined as follows:

On a finite input sample $S \in Z^m$ output some $h \in \mathcal{H}$ that minimizes the empirical loss, $L_S(h) = \frac{1}{|S|} \sum_{z \in S} \ell(h, z)$.

This section studies the runtime of implementing the ERM rule for several examples of learning tasks.

8.2.1 Finite Classes

Limiting the hypothesis class to be a finite class may be considered as a reasonably mild restriction. For example, $\mathcal{H}$ can be the set of all predictors that can be implemented by a C++ program written in at most 10000 bits of code. Other examples of useful finite classes are any hypothesis class that can be parameterized by a finite number of parameters, where we are satisfied with a representation of each of the parameters using a finite number of bits, for example, the class of axis aligned rectangles in the Euclidean space, $\mathbb{R}^d$, when the parameters defining any given rectangle are specified up to some limited precision.

As we have shown in previous chapters, the sample complexity of learning a finite class is upper bounded by $m_{\mathcal{H}}(\epsilon, \delta) = c \log(c|\mathcal{H}|/\delta)/\epsilon^c$, where $c = 1$ in the realizable case and $c = 2$ in the nonrealizable case. Therefore, the sample complexity has a mild dependence on the size of $\mathcal{H}$. In the example of C++ programs mentioned before, the number of hypotheses is $2^{10,000}$ but the sample complexity is only $c(10,000 + \log(c/\delta))/\epsilon^c$.

A straightforward approach for implementing the ERM rule over a finite hypothesis class is to perform an exhaustive search. That is, for each $h \in \mathcal{H}$ we calculate the empirical risk, $L_S(h)$, and return a hypothesis that minimizes the empirical risk. Assuming that the evaluation of $\ell(h, z)$ on a single example takes a constant amount of time, k, the runtime of this exhaustive search becomes $k|\mathcal{H}|m$, where m is the size of the training set. If we let m to be the upper bound on the sample complexity mentioned, then the runtime becomes $k|\mathcal{H}|c \log(c|\mathcal{H}|/\delta)/\epsilon^c$.

The linear dependence of the runtime on the size of $\mathcal{H}$ makes this approach inefficient (and unrealistic) for large classes. Formally, if we define a sequence of problems $(Z_n, \mathcal{H}_n, \ell_n)_{n=1}^{\infty}$ such that $\log(|\mathcal{H}_n|) = n$, then the exhaustive search approach yields an exponential runtime. In the example of C++ programs, if $\mathcal{H}_n$ is the set of functions that can be implemented by a C++ program written in at most n bits of code, then the runtime grows exponentially with n, implying that the exhaustive search approach is unrealistic for practical use. In fact, this problem is one of the reasons we are dealing with other hypothesis classes, like classes of linear predictors, which we will encounter in the next chapter, and not just focusing on finite classes.

It is important to realize that the inefficiency of one algorithmic approach (such as the exhaustive search) does not yet imply that no efficient ERM implementation exists. Indeed, we will show examples in which the ERM rule can be implemented efficiently.

8.2.2 Axis Aligned Rectangles

Let $\mathcal{H}_n$ be the class of axis aligned rectangles in $\mathbb{R}^n$, namely,

$$\mathcal{H}_n = \{h_{(a_1,\ldots,a_n,b_1,\ldots,b_n)} : \forall i, a_i \leq b_i\}$$

where

$$h_{(a_1,\ldots,a_n,b_1,\ldots,b_n)}(\mathbf{x}, y) = \begin{cases} 1 & \text{if } \forall i,\ x_i \in [a_i, b_i] \\ 0 & \text{otherwise} \end{cases} \tag{8.1}$$

Efficiently Learnable in the Realizable Case
Consider implementing the ERM rule in the realizable case. That is, we are given a training set $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ of examples, such that there exists an axis aligned rectangle, $h \in \mathcal{H}_n$, for which $h(\mathbf{x}_i) = y_i$ for all i. Our goal is to find such an axis aligned rectangle with a zero training error, namely, a rectangle that is consistent with all the labels in S.

We show later that this can be done in time $O(nm)$. Indeed, for each $i \in [n]$, set $a_i = \min\{x_i : (\mathbf{x}, 1) \in S\}$ and $b_i = \max\{x_i : (\mathbf{x}, 1) \in S\}$. In words, we take a_i to be the minimal value of the i'th coordinate of a positive example in S and b_i to be the maximal value of the i'th coordinate of a positive example in S. It is easy to verify that the resulting rectangle has zero training error and that the runtime of finding each a_i and b_i is $O(m)$. Hence, the total runtime of this procedure is $O(nm)$.

Not Efficiently Learnable in the Agnostic Case
In the agnostic case, we do not assume that some hypothesis h perfectly predicts the labels of all the examples in the training set. Our goal is therefore to find h that minimizes the number of examples for which $y_i \neq h(\mathbf{x}_i)$. It turns out that for many common hypothesis classes, including the classes of axis aligned rectangles we consider here, solving the ERM problem in the agnostic setting is NP-hard (and, in most cases, it is even NP-hard to find some $h \in \mathcal{H}$ whose error is no more than some constant $c > 1$ times that of the empirical risk minimizer in $\mathcal{H}$). That is, unless P = NP, there is no algorithm whose running time is polynomial in m and n that is guaranteed to find an ERM hypothesis for these problems (Ben-David, Eiron & Long 2003).

On the other hand, it is worthwhile noticing that, if we fix one specific hypothesis class, say, axis aligned rectangles in some fixed dimension, n, then there exist efficient learning algorithms for this class. In other words, there are successful agnostic PAC learners that run in time polynomial in $1/\epsilon$ and $1/\delta$ (but their dependence on the dimension n is not polynomial).

To see this, recall the implementation of the ERM rule we presented for the realizable case, from which it follows that an axis aligned rectangle is determined by at most $2n$ examples. Therefore, given a training set of size m, we can perform an exhaustive search over all subsets of the training set of size at most $2n$ examples and construct a rectangle from each such subset. Then, we can pick

the rectangle with the minimal training error. This procedure is guaranteed to find an ERM hypothesis, and the runtime of the procedure is $m^{O(n)}$. It follows that if n is fixed, the runtime is polynomial in the sample size. This does not contradict the aforementioned hardness result, since there we argued that unless P=NP one cannot have an algorithm whose dependence on the dimension n is polynomial as well.

8.2.3 Boolean Conjunctions

A Boolean conjunction is a mapping from $\mathcal{X} = \{0,1\}^n$ to $\mathcal{Y} = \{0,1\}$ that can be expressed as a proposition formula of the form $x_{i_1} \wedge \ldots \wedge x_{i_k} \wedge \neg x_{j_1} \wedge \ldots \wedge \neg x_{j_r}$, for some indices $i_1, \ldots, i_k, j_1, \ldots, j_r \in [n]$. The function that such a proposition formula defines is

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } x_{i_1} = \cdots = x_{i_k} = 1 \text{ and } x_{j_1} = \cdots = x_{j_r} = 0 \\ 0 & \text{otherwise} \end{cases}$$

Let $\mathcal{H}_C^n$ be the class of all Boolean conjunctions over $\{0,1\}^n$. The size of $\mathcal{H}_C^n$ is at most $3^n + 1$ (since in a conjunction formula, each element of $\mathbf{x}$ either appears, or appears with a negation sign, or does not appear at all, and we also have the all negative formula). Hence, the sample complexity of learning $\mathcal{H}_C^n$ using the ERM rule is at most $n \log(3/\delta)/\epsilon$.

Efficiently Learnable in the Realizable Case

Next, we show that it is possible to solve the ERM problem for $\mathcal{H}_C^n$ in time polynomial in n and m. The idea is to define an ERM conjunction by including in the hypothesis conjunction all the literals that do not contradict any positively labeled example. Let $\mathbf{v}_1, \ldots, \mathbf{v}_{m^+}$ be all the positively labeled instances in the input sample S. We define, by induction on $i \leq m^+$, a sequence of hypotheses (or conjunctions). Let h_0 be the conjunction of all possible literals. That is, $h_0 = x_1 \wedge \neg x_1 \wedge x_2 \wedge \ldots \wedge x_n \wedge \neg x_n$. Note that h_0 assigns the label 0 to all the elements of $\mathcal{X}$. We obtain h_{i+1} by deleting from the conjunction h_i all the literals that are not satisfied by $\mathbf{v}_{i+1}$. The algorithm outputs the hypothesis h_{m^+}. Note that h_{m^+} labels positively all the positively labeled examples in S. Furthermore, for every $i \leq m^+$, h_i is the most restrictive conjunction that labels $\mathbf{v}_1, \ldots, \mathbf{v}_i$ positively. Now, since we consider learning in the realizable setup, there exists a conjunction hypothesis, $f \in \mathcal{H}_C^n$, that is consistent with all the examples in S. Since h_{m^+} is the most restrictive conjunction that labels positively all the positively labeled members of S, any instance labeled 0 by f is also labeled 0 by h_{m^+} . It follows that h_{m^+} has zero training error (w.r.t. S), and is therefore a legal ERM hypothesis. Note that the running time of this algorithm is $O(mn)$.

Not Efficiently Learnable in the Agnostic Case
As in the case of axis aligned rectangles, unless P = NP, there is no algorithm whose running time is polynomial in m and n that guaranteed to find an ERM hypothesis for the class of Boolean conjunctions in the unrealizable case.

8.2.4 Learning 3-Term DNF

We next show that a slight generalization of the class of Boolean conjunctions leads to intractability of solving the ERM problem even in the realizable case. Consider the class of 3-term disjunctive normal form formulae (3-term DNF). The instance space is $\mathcal{X} = \{0,1\}^n$ and each hypothesis is represented by the Boolean formula of the form $h(\mathbf{x}) = A_1(\mathbf{x}) \vee A_2(\mathbf{x}) \vee A_3(\mathbf{x})$, where each $A_i(\mathbf{x})$ is a Boolean conjunction (as defined in the previous section). The output of $h(\mathbf{x})$ is 1 if either $A_1(\mathbf{x})$ or $A_2(\mathbf{x})$ or $A_3(\mathbf{x})$ outputs the label 1. If all three conjunctions output the label 0 then $h(\mathbf{x}) = 0$.

Let $\mathcal{H}^n_{3DNF}$ be the hypothesis class of all such 3-term DNF formulae. The size of $\mathcal{H}^n_{3DNF}$ is at most 3^{3n}. Hence, the sample complexity of learning $\mathcal{H}^n_{3DNF}$ using the ERM rule is at most $3n \log(3/\delta)/\epsilon$.

However, from the computational perspective, this learning problem is hard. It has been shown (see (Pitt & Valiant 1988, Kearns et al. 1994)) that unless RP = NP, there is no polynomial time algorithm that *properly* learns a sequence of 3-term DNF learning problems in which the dimension of the n'th problem is n. By "properly" we mean that the algorithm should output a hypothesis that is a 3-term DNF formula. In particular, since $\text{ERM}_{\mathcal{H}^n_{3DNF}}$ outputs a 3-term DNF formula it is a proper learner and therefore it is hard to implement it. The proof uses a reduction of the graph 3-coloring problem to the problem of PAC learning 3-term DNF. The detailed technique is given in Exercise 3. See also (Kearns & Vazirani 1994, Section 1.4).

8.3 Efficiently Learnable, but Not by a Proper ERM

In the previous section we saw that it is impossible to implement the ERM rule efficiently for the class $\mathcal{H}^n_{3DNF}$ of 3-DNF formulae. In this section we show that it is possible to learn this class efficiently, but using ERM with respect to a larger class.

Representation Independent Learning Is Not Hard
Next we show that it is possible to learn 3-term DNF formulae efficiently. There is no contradiction to the hardness result mentioned in the previous section as we now allow "representation independent" learning. That is, we allow the learning algorithm to output a hypothesis that is not a 3-term DNF formula. The basic idea is to replace the original hypothesis class of 3-term DNF formula with a larger hypothesis class so that the new class is easily learnable. The learning

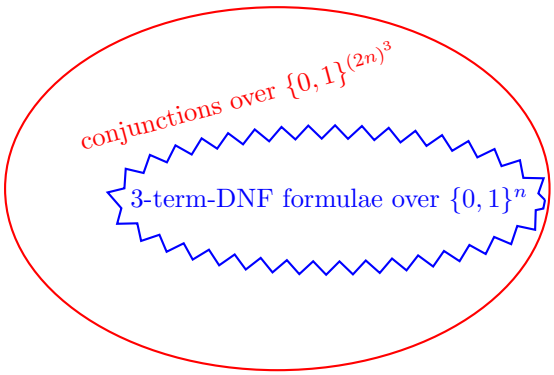
algorithm might return a hypothesis that does not belong to the original hypothesis class; hence the name "representation independent" learning. We emphasize that in most situations, returning a hypothesis with good predictive ability is what we are really interested in doing.

We start by noting that because $\vee$ distributes over $\wedge$, each 3-term DNF formula can be rewritten as

$$A_1 \vee A_2 \vee A_3 = \bigwedge_{u \in A_1, v \in A_2, w \in A_3} (u \vee v \vee w)$$

Next, let us define: $\psi : \{0,1\}^n \to \{0,1\}^{(2n)^3}$ such that for each triplet of literals u, v, w there is a variable in the range of ψ indicating if $u \vee v \vee w$ is true or false. So, for each 3-DNF formula over $\{0,1\}^n$ there is a conjunction over $\{0,1\}^{(2n)^3}$, with the same truth table. Since we assume that the data is realizable, we can solve the ERM problem with respect to the class of conjunctions over $\{0,1\}^{(2n)^3}$. Furthermore, the sample complexity of learning the class of conjunctions in the higher dimensional space is at most $n^3 \log(1/\delta)/\epsilon$. Thus, the overall runtime of this approach is polynomial in n.

Intuitively, the idea is as follows. We started with a hypothesis class for which learning is hard. We switched to another representation where the hypothesis class is larger than the original class but has more structure, which allows for a more efficient ERM search. In the new representation, solving the ERM problem is easy.



8.4 Hardness of Learning*

We have just demonstrated that the computational hardness of implementing $\text{ERM}_{\mathcal{H}}$ does not imply that such a class $\mathcal{H}$ is not learnable. How can we prove that a learning problem is computationally hard?

One approach is to rely on cryptographic assumptions. In some sense, cryptography is the opposite of learning. In learning we try to uncover some rule underlying the examples we see, whereas in cryptography, the goal is to make sure that nobody will be able to discover some secret, in spite of having access

to some partial information about it. On that high level intuitive sense, results about the cryptographic security of some system translate into results about the unlearnability of some corresponding task. Regrettably, currently one has no way of proving that a cryptographic protocol is not breakable. Even the common assumption of P $\neq$ NP does not suffice for that (although it can be shown to be necessary for most common cryptographic scenarios). The common approach for proving that cryptographic protocols are secure is to start with some *cryptographic assumptions*. The more these are used as a basis for cryptography, the stronger is our belief that they really hold (or, at least, that algorithms that will refute them are hard to come by).

We now briefly describe the basic idea of how to deduce hardness of learnability from cryptographic assumptions. Many cryptographic systems rely on the assumption that there exists a one way function. Roughly speaking, a one way function is a function $f : \{0,1\}^n \to \{0,1\}^n$ (more formally, it is a sequence of functions, one for each dimension n) that is easy to compute but is hard to invert. More formally, f can be computed in time poly(n) but for any randomized polynomial time algorithm A, and for every polynomial $p(\cdot)$,

$$\mathbb{P}[f(A(f(\mathbf{x}))) = f(\mathbf{x})] < \tfrac{1}{p(n)},$$

where the probability is taken over a random choice of $\mathbf{x}$ according to the uniform distribution over $\{0,1\}^n$ and the randomness of A.

A one way function, f, is called trapdoor one way function if, for some polynomial function p, for every n there exists a bit-string s_n (called a secret key) of length $\leq p(n)$, such that there is a polynomial time algorithm that, for every n and every $\mathbf{x} \in \{0,1\}^n$, on input $(f(\mathbf{x}), s_n)$ outputs $\mathbf{x}$. In other words, although f is hard to invert, once one has access to its secret key, inverting f becomes feasible. Such functions are parameterized by their secret key.

Now, let F_n be a family of trapdoor functions over $\{0,1\}^n$ that can be calculated by some polynomial time algorithm. That is, we fix an algorithm that given a secret key (representing one function in F_n) and an input vector, it calculates the value of the function corresponding to the secret key on the input vector in polynomial time. Consider the task of learning the class of the corresponding inverses, $H_F^n = \{f^{-1} : f \in F_n\}$. Since each function in this class can be inverted by some secret key s_n of size polynomial in n, the class H_F^n can be parameterized by these keys and its size is at most $2^{p(n)}$. Its sample complexity is therefore polynomial in n. We claim that there can be no efficient learner for this class. If there were such a learner, L, then by sampling uniformly at random a polynomial number of strings in $\{0,1\}^n$, and computing f over them, we could generate a labeled training sample of pairs $(f(\mathbf{x}), \mathbf{x})$, which should suffice for our learner to figure out an (ϵ, δ) approximation of f^{-1} (w.r.t. the uniform distribution over the range of f), which would violate the one way property of f.

A more detailed treatment, as well as a concrete example, can be found in (Kearns & Vazirani 1994, Chapter 6). Using reductions, they also show that

the class of functions that can be calculated by small Boolean circuits is not efficiently learnable, even in the realizable case.

8.5 Summary

The runtime of learning algorithms is asymptotically analyzed as a function of different parameters of the learning problem, such as the size of the hypothesis class, our measure of accuracy, our measure of confidence, or the size of the domain set. We have demonstrated cases in which the ERM rule can be implemented efficiently. For example, we derived efficient algorithms for solving the ERM problem for the class of Boolean conjunctions and the class of axis aligned rectangles, under the realizability assumption. However, implementing ERM for these classes in the agnostic case is NP-hard. Recall that from the statistical perspective, there is no difference between the realizable and agnostic cases (i.e., a class is learnable in both cases if and only if it has a finite VC-dimension). In contrast, as we saw, from the computational perspective the difference is immense. We have also shown another example, the class of 3-term DNF, where implementing ERM is hard even in the realizable case, yet the class is efficiently learnable by another algorithm.

Hardness of implementing the ERM rule for several natural hypothesis classes has motivated the development of alternative learning methods, which we will discuss in the next part of this book.

8.6 Bibliographic Remarks

Valiant (1984) introduced the efficient PAC learning model in which the runtime of the algorithm is required to be polynomial in $1/\epsilon$, $1/\delta$, and the representation size of hypotheses in the class. A detailed discussion and thorough bibliographic notes are given in Kearns & Vazirani (1994).

8.7 Exercises

1. Let $\mathcal{H}$ be the class of intervals on the line (formally equivalent to axis aligned rectangles in dimension $n = 1$). Propose an implementation of the $\text{ERM}_{\mathcal{H}}$ learning rule (in the agnostic case) that given a training set of size m, runs in time $O(m^2)$.
   Hint: Use dynamic programming.
2. Let $\mathcal{H}_1, \mathcal{H}_2, \ldots$ be a sequence of hypothesis classes for binary classification. Assume that there is a learning algorithm that implements the ERM rule in the realizable case such that the output hypothesis of the algorithm for each class $\mathcal{H}_n$ only depends on $O(n)$ examples out of the training set. Furthermore,

assume that such a hypothesis can be calculated given these $O(n)$ examples in time $O(n)$, and that the empirical risk of each such hypothesis can be evaluated in time $O(mn)$. For example, if $\mathcal{H}_n$ is the class of axis aligned rectangles in $\mathbb{R}^n$, we saw that it is possible to find an ERM hypothesis in the realizable case that is defined by at most $2n$ examples. Prove that in such cases, it is possible to find an ERM hypothesis for $\mathcal{H}_n$ in the unrealizable case in time $O(mn\, m^{O(n)})$.

3. In this exercise, we present several classes for which finding an ERM classifier is computationally hard. First, we introduce the class of n-dimensional halfspaces, HS_n, for a domain $\mathcal{X} = \mathbb{R}^n$. This is the class of all functions of the form $h_{\mathbf{w},b}(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$ where $\mathbf{w}, \mathbf{x} \in \mathbb{R}^n$, $\langle \mathbf{w}, \mathbf{x} \rangle$ is their inner product, and $b \in \mathbb{R}$. See a detailed description in Chapter 9.

   1. Show that $\text{ERM}_{\mathcal{H}}$ over the class $\mathcal{H} = HS_n$ of linear predictors is computationally hard. More precisely, we consider the sequence of problems in which the dimension n grows linearly and the number of examples m is set to be some constant times n.

      Hint: You can prove the hardness by a reduction from the following problem:

      Max FS: Given a system of linear inequalities, $A\mathbf{x} > \mathbf{b}$ with $A \in R^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ (that is, a system of m linear inequalities in n variables, $\mathbf{x} = (x_1, \ldots, x_n)$), find a subsystem containing as many inequalities as possible that has a solution (such a subsystem is called *feasible*).

      It has been shown (Sankaran 1993) that the problem Max FS is NP-hard. Show that any algorithm that finds an ERM_{HS_n} hypothesis for any training sample $S \in (\mathbb{R}^n \times \{+1, -1\})^m$ can be used to solve the Max FS problem of size m, n. *Hint*: Define a mapping that transforms linear inequalities in n variables into labeled points in $\mathbb{R}^n$, and a mapping that transforms vectors in $\mathbb{R}^n$ to halfspaces, such that a vector $\mathbf{w}$ satisfies an inequality q if and only if the labeled point that corresponds to q is classified correctly by the halfspace corresponding to $\mathbf{w}$. Conclude that the problem of empirical risk minimization for halfspaces in also NP-hard (that is, if it can be solved in time polynomial in the sample size, m, and the Euclidean dimension, n, then every problem in the class NP can be solved in polynomial time).

   2. Let $\mathcal{X} = \mathbb{R}^n$ and let $\mathcal{H}_k^n$ be the class of all intersections of k-many linear halfspaces in $\mathbb{R}^n$. In this exercise, we wish to show that $\text{ERM}_{\mathcal{H}_k^n}$ is computationally hard for every $k \geq 3$. Precisely, we consider a sequence of problems where $k \geq 3$ is a constant and n grows linearly. The training set size, m, also grows linearly with n.

      Towards this goal, consider the k-coloring problem for graphs, defined as follows:

      Given a graph $G = (V, E)$, and a number k, determine whether there exists a function $f : V \to \{1 \ldots k\}$ so that for every $(u, v) \in E$, $f(u) \neq f(v)$.

      The k-coloring problem is known to be NP-hard for every $k \geq 3$ (Karp 1972).

We wish to reduce the k-coloring problem to $ERM_{\mathcal{H}_k^n}$: that is, to prove that if there is an algorithm that solves the $ERM_{\mathcal{H}_k^n}$ problem in time polynomial in k, n, and the sample size m, then there is a polynomial time algorithm for the graph k-coloring problem.
Given a graph $G = (V, E)$, let $\{v_1 \dots v_n\}$ be the vertices in V. Construct a sample $S(G) \in (\mathbb{R}^n \times \{\pm 1\})^m$, where $m = |V| + |E|$, as follows:

- For every $v_i \in V$, construct an instance $\mathbf{e}_i$ with a negative label.
- For every edge $(v_i, v_j) \in E$, construct an instance $(\mathbf{e}_i + \mathbf{e}_j)/2$ with a positive label.

1. Prove that if there exists some $h \in \mathcal{H}_k^n$ that has zero error over $S(G)$ then G is k-colorable.
   Hint: Let $h = \bigcap_{j=1}^{k} h_j$ be an ERM classifier in $\mathcal{H}_k^n$ over S. Define a coloring of V by setting $f(v_i)$ to be the minimal j such that $h_j(\mathbf{e}_i) = -1$. Use the fact that halfspaces are convex sets to show that it cannot be true that two vertices that are connected by an edge have the same color.
2. Prove that if G is k-colorable then there exists some $h \in H_k^n$ that has zero error over $S(G)$.
   Hint: Given a coloring f of the vertices of G, we should come up with k hyperplanes, $h_1 \dots h_k$ whose intersection is a perfect classifier for $S(G)$. Let $b = 0.6$ for all of these hyperplanes and, for $t \leq k$ let the i'th weight of the t'th hyperplane, $w_{t,i}$, be -1 if $f(v_i) = t$ and 0 otherwise.
3. Based on the above, prove that for any $k \geq 3$, the $\text{ERM}_{\mathcal{H}_k^n}$ problem is NP-hard.

4. In this exercise we show that hardness of solving the ERM problem is equivalent to hardness of proper PAC learning. Recall that by "properness" of the algorithm we mean that it must output a hypothesis from the hypothesis class. To formalize this statement, we first need the following definition.

   DEFINITION 8.2 The complexity class Randomized Polynomial (RP) time is the class of all decision problems (that is, problems in which on any instance one has to find out whether the answer is YES or NO) for which there exists a probabilistic algorithm (namely, the algorithm is allowed to flip random coins while it is running) with these properties:

   - On any input instance the algorithm runs in polynomial time in the input size.
   - If the correct answer is NO, the algorithm must return NO.
   - If the correct answer is YES, the algorithm returns YES with probability $a \geq 1/2$ and returns NO with probability $1 - a$.[1]

   Clearly the class RP contains the class P. It is also known that RP is contained in the class NP. It is not known whether any equality holds among these three complexity classes, but it is widely believed that NP is strictly

[1] The constant 1/2 in the definition can be replaced by any constant in $(0, 1)$.

larger than RP. In particular, it is believed that NP-hard problems cannot be solved by a randomized polynomial time algorithm.

- Show that if a class $\mathcal{H}$ is *properly* PAC learnable by a polynomial time algorithm, then the $\text{ERM}_{\mathcal{H}}$ problem is in the class RP. In particular, this implies that whenever the $\text{ERM}_{\mathcal{H}}$ problem is NP-hard (for example, the class of intersections of halfspaces discussed in the previous exercise), then, unless NP = RP, there exists no polynomial time proper PAC learning algorithm for $\mathcal{H}$.

  Hint: Assume you have an algorithm A that properly PAC learns a class $\mathcal{H}$ in time polynomial in some class parameter n as well as in $1/\epsilon$ and $1/\delta$. Your goal is to use that algorithm as a subroutine to contract an algorithm B for solving the $\text{ERM}_{\mathcal{H}}$ problem in random polynomial time. Given a training set, $S \in (\mathcal{X} \times \{\pm 1\}^m)$, and some $h \in \mathcal{H}$ whose error on S is zero, apply the PAC learning algorithm to the uniform distribution over S and run it so that with probability ≥ 0.3 it finds a function $h \in \mathcal{H}$ that has error less than $\epsilon = 1/|S|$ (with respect to that uniform distribution). Show that the algorithm just described satisfies the requirements for being a RP solver for $\text{ERM}_{\mathcal{H}}$.