6

Inherent Unpredictability

6.1 Representation Dependent and Independent Hardness

Recall that in Chapter 1, we proved that some particular concept classes are hard to PAC learn if we place certain restrictions on the hypothesis class used by the learning algorithm. More precisely, it was shown that if $RP \neq NP$, then there is no polynomial-time algorithm for PAC learning k-term DNF using k-term DNF. However, we then went on to show that k-term DNF is efficiently PAC learnable if the algorithm is allowed to output a hypothesis from the more expressive class of kCNF formulae.

These results raise an interesting and fundamental question regarding the PAC learning model: are there classes of concepts that are hard to PAC learn, not because of hypothesis class restrictions, but because of the inherent computational difficulty of prediction — that is, regardless of the hypothesis class $\mathcal{H}$ used by a learning algorithm? More precisely, we are interested in the existence of concept classes $\mathcal{C}$ in which the VC dimension of $\mathcal{C}_n$ is polynomial in n (and thus by the results of Chapter 3, there is no *information-theoretic* barrier to fast learning — a sample of polynomial size is sufficient to determine a good hypothesis), yet $\mathcal{C}$ is not

efficiently PAC learnable using any polynomially evaluatable hypothesis class $\mathcal{H}$. In fact, by the equivalence of weak and strong learning proved in Chapter 1, we may as well strengthen this last condition and ask that $\mathcal{C}$ be not even weakly learnable using any polynomially evaluatable hypothesis class. We shall informally refer to a class $\mathcal{C}$ meeting these conditions as **inherently unpredictable**, since despite the fact that a small sample contains sufficient information to determine a good hypothesis, a polynomial time algorithm cannot even find a hypothesis beating a fair coin. Such a class would be hard to learn for different and arguably more meaningful reasons than the class of k-term DNF, for which the hardness results of Chapter 1 are essentially the consequence of a perhaps artificial syntactic restriction on the hypothesis representation.

In this chapter and the next, we will prove not only that inherently unpredictable classes exist, but furthermore that several rather natural classes of concepts are inherently unpredictable. These results will also demonstrate an interesting connection between hardness results for PAC learning and constructions in the field of public-key cryptography, where the necessary tools for our results were first developed.

6.2 The Discrete Cube Root Problem

Our proofs of inherent unpredictability will rely on some unproven computational assumptions that have become widely accepted as standard working assumptions in cryptography and computational complexity. In fact, since the $P = NP$ question is a fundamental unresolved problem in complexity theory, we cannot hope to prove inherent unpredictability for any polynomially evaluatable class without some complexity assumption. This is because for any polynomially evaluatable class $\mathcal{H}$, the problem of determining, on input any labeled sample S, whether there is a hypothesis $h \in \mathcal{H}$ consistent with S is in NP (because given any witness $h \in \mathcal{H}$ we can *verify* consistency with S in polynomial time). If $P = NP$, then such a consistent hypothesis can be computed in polynomial time, and thus by

Occam's Razor (Theorem 2.1) the concept class is PAC learnable. Thus, in the same way that the hardness of PAC learning k-term DNF using k-term DNF relied on the complexity-theoretic assumption $RP \neq NP$, and therefore on the assumed intractability of particular computational problems such as graph coloring, we must expect any theorem stating that a concept class is inherently unpredictable to rely on the assumed intractability of some specific computational problem. We now propose a candidate problem for our purposes, which will require a brief digression into number theory.

Let $N = pq$ be the product of two prime natural numbers p and q of approximately equal length. Factoring numbers of this form is widely believed to be computationally intractable, even in the case where the primes p and q are chosen randomly and we only ask the factoring algorithm to succeed with some non-negligible probability. Let $f_N(x) = x^3 \ mod \ N$, and consider the problem of inverting $f_N(x)$ — that is, the problem of computing x on inputs N and $f_N(x)$ (but not given p and q!).

In order to make this problem well-defined, we first need to arrange things so that $f_N(x)$ is in fact a bijection (permutation). Before doing so let us review some elementary number theory. The natural numbers in $\{1, \cdots, N-1\}$ that are relatively prime with N (two natural numbers are **relatively prime** if their greatest common divisor is 1) form a group under the operation of multiplication modulo N. This group is denoted by Z_N^*, and the **order** of this group, which we denote by $\varphi(N) = |Z_N^*|$, is $\varphi(N) = (p-1)(q-1)$. Returning to the question of whether $f_N(x)$ is a bijection: we claim that if 3 does not divide $\varphi(N)$, then $f_N(x)$ is a permutation of Z_N^*.

To see this, let d satisfy $3d = 1 \ mod \ \varphi(N)$. Such a d exists because the greatest common divisor of 3 and $\varphi(N)$ is 1, and so by Euclid's theorem there are integers c and d such that $\varphi(N)c + 3d = 1$. In fact, d can be efficiently computed using Euclid's extended greatest common divisor algorithm. Now we claim that the inverse function $f_N^{-1}(y)$ of $f_N(x)$ is simply the mapping $f_N^{-1}(y) = y^d \ mod \ N$: since $3d = 1 \ mod \ \varphi(N)$ means

$3d = k\varphi(N) + 1$ for some natural number k, we have

$$\begin{aligned}(f_N(x))^d \bmod N &= (x^3 \bmod N)^d \bmod N \\ &= x^{3d} \bmod N \\ &= x^{k\cdot\varphi(N)+1} \bmod N \\ &= (x^{\varphi(N)})^k x \bmod N.\end{aligned}$$

But a well-known theorem of Euler states that any element of a group raised to the order of the group is equal to the group's identity element, giving $(x^{\varphi(N)})^k = 1^k \bmod N = 1 \bmod N$ and thus $(f_N(x))^d = x \bmod N$ as desired. In the sequel we will refer to d as the **inverting exponent** for N. The existence of this inverse mapping $f_N^{-1}(y)$ establishes that $f_N(x)$ is indeed a bijection.

We can now formally define our problem.

Discrete Cube Root Problem. Two primes p and q are chosen such that 3 does not divide $\varphi(N) = (p-1)(q-1)$, where $N = p \cdot q$. Then $x \in Z_N^*$ is chosen. An algorithm for the Discrete Cube Root Problem is given as input both N and $y = f_N(x)$, and must output x.

Note that the length of the input to this problem is $O(\log N)$ — not N. So a polynomial time algorithm for this problem must run in time polynomial in $\log N$. We now discuss the computational difficulty of the Discrete Cube Root Problem, leading to a formal assumption about its intractability that is widely believed.

6.2.1 The Difficulty of Discrete Cube Roots

Notice that the Discrete Cube Root Problem would be easy to solve in polynomial time if the prime factors p and q of N were also provided as part of the input along with N and y. We could simply compute the inverting exponent d for N from p and q using Euclid's algorithm, and then compute $f_N^{-1}(y) = y^d \bmod N = x \bmod N$. We would have

to be a little careful in computing $y^d \ mod \ N$ efficiently, since we have time only polynomial in $\log N$, whereas d is of the order of N. There is a standard trick for computing $y^d \ mod \ N$ by repeatedly squaring y modulo N which we will describe in detail in Section 6.3. One consequence of this observation is that for each fixed N of length n bits there is a boolean circuit of size polynomial in n that computes cube roots modulo N — the circuit simply has the inverting exponent d for N "hard-wired", and then performs the required exponentiation on the input y.

How hard is computing cube roots when the prime factors of N are not part of the input, which is the way we have defined the Discrete Cube Root Problem? The obvious method — namely, to first factor N to obtain p and q, compute d from p and q using Euclid's algorithm, and then efficiently compute cube roots via exponentiation as outlined above — runs into the widely known and computationally difficult problem of factoring integers. Although computing cube roots has not been proved to be as hard as factoring, the security of the well-known RSA public key cryptosystem is based on the assumption that the Discrete Cube Root Problem is intractable.

We now formally state our intracability assumption for the Discrete Cube Root Problem, which is an assumption on the average-case difficulty:

The **Discrete Cube Root Assumption** states that for every polynomial $p(\cdot)$, there is no algorithm that runs in time $p(n)$, and that on input N and $y = f_N(x)$ (where N is an n-bit number that is the product of two randomly chosen primes p and q such that 3 does not divide $\varphi(N) = (p-1)(q-1)$, and x is chosen randomly in Z_N^*) outputs x with probability exceeding $1/p(n)$. The probability is taken over the random draws of p and q and x, and any internal randomization of the algorithm.

The fact that extensive efforts have not yielded any efficient algorithm or even a heuristic for computing discrete cube roots means that any PAC learning problem that is proved intractable under the Discrete Cube Root Assumption is, at least for all practical purposes, not learnable given our

current understanding of number-theoretic computation.

6.2.2 Discrete Cube Roots as a Learning Problem

Suppose we are given a random N, and $y = f_N(x)$ for a random $x \in Z_N^*$, and we want to compute x from these inputs. The Discrete Cube Root Assumption asserts that this is a difficult problem. But now suppose that in addition to these two inputs, we had access to many already solved "examples" for the given N. That is, suppose we are also given a sample

$$\langle y_1, f_N^{-1}(y_1)\rangle, \ldots, \langle y_k, f_N^{-1}(y_k)\rangle$$

where each $y_i \in Z_N^*$ is chosen randomly. Then does the problem of computing x become any easier?

The answer to this question is no, because since we are already given N, we can generate such random pairs efficiently ourselves by picking a random $x_i \in Z_N^*$ and obtaining the pair $\langle f_N(x_i), x_i\rangle$. By setting $y_i = f_N(x_i)$ (and thus $x_i = f_N^{-1}(y_i)$), and remembering that f_N is a bijection on Z_N^*, we see that these pairs have the same distribution as those $\langle y_i, f_N^{-1}(y_i)\rangle$ generated by first picking a random number $y_i \in Z_N^*$ and then computing $f_N^{-1}(y_i)$.

In our study so far, we have viewed the learning problem as that of using a training sample of random examples to find a hypothesis that has small error with respect to the target function and distribution. An equivalent view of the learning problem is that of using the training sample to predict the target function's output on a new randomly chosen input from the domain. If we choose our target function to be f_N^{-1} for some N, the input domain to be Z_N^*, the input distribution to be uniform on Z_N^*, then under the Discrete Cube Root Assumption we have a computationally hard learning problem.

Before we cast this hard learning problem in the PAC model, let us first formalize it a little further. For every natural number n, let

the class $\mathcal{F}_n$ consist of all the inverse functions f_N^{-1} for the functions $f_N(x) = x^3 \ mod \ N$, where $N = pq$ is n bits long and is the product of two primes p and q such that 3 is relatively prime with $\varphi(N) = (p-1)(q-1)$. Let $f_N^{-1} \in \mathcal{F}_n$ be the target function, and let the learning algorithm be given access to a source of random input-output pairs of f_N^{-1}, where the input distribution is uniform on Z_N^*. (Note that this hard distribution depends on the target function f_N^{-1}; in particular, it is not same as the uniform distribution on $\{0,1\}^n$.) The goal of the learning algorithm is to discover in time polynomial in n a hypothesis function h that agrees with f_N^{-1} even on only $1/p(n)$ of the distribution for some fixed polynomial $p(\cdot)$. If an algorithm A exists for this problem, it is easy to see that the Discrete Cube Root Assumption is false: given N and y as input, we first set y aside and use N to generate examples of f_N^{-1} with respect to the uniform distribution on Z_N^* as described above. We use these examples to simulate algorithm A, and then use the hypothesis h output by A to compute $f_N^{-1}(y)$. Then for a random input y, we get the correct value for $f_N^{-1}(y)$ with probability at least $1/p(n)$, thus contradicting the Discrete Cube Root Assumption.

We have already informally argued (and again, we will provide details in Section 6.3) that each function in $\mathcal{F}_n$ can be computed by a boolean circuit whose size polynomial in n. On the other hand, $\mathcal{F}$ is hard to learn in this PAC-like setting for multivalued functions (under the Discrete Cube Root Assumption). We emphasize that this negative result does not place any restriction on the form of the hypothesis h output by the learning algorithm — the only requirement is that the hypothesis can be evaluated in polynomial time. This requirement is obviously necessary in the argument just given, since the last step in using the learning algorithm to solve an instance y of the Discrete Cube Root Problem is to evaluate the hypothesis on y.

The only aspect of our learning problem that keeps it from sitting squarely in the PAC model is that our function class $\mathcal{F}$ is a class of multivalued functions, not a class of boolean functions. Indeed, it is easy to see that we could not hope for such a strong negative result for a

boolean function class, since for any boolean function we can always find a hypothesis whose error is bounded by $1/2$ with respect to any input distribution (by simply using the randomized hypothesis that flips a fair coin to predict each label), whereas the Discrete Cube Root Assumption implies that for $\mathcal{F}$, even achieving error bounded by $1 - 1/p(n)$ for any polynomial $p(\cdot)$ is intractable.

However, there is an easy fix that yields a true PAC learning problem. The idea is simple: we regard each output bit of the function f_N^{-1} as a concept (boolean function). If there is an algorithm that can be used to learn each of these output bits with high accuracy, then we can reconstruct all of the output bits with high accuracy.

More precisely, for each multivalued function f_N^{-1}, we define n boolean functions $f_{N,i}^{-1}$, $1 \leq i \leq n$, where for any $y \in Z_N^*$, $f_{N,i}^{-1}(y)$ is defined to be the ith bit of $f_N^{-1}(y)$. Now we let $\mathcal{C}_n$ be the boolean function class obtained by including $f_{N,i}^{-1}$ in $\mathcal{C}_n$ for all $f_N^{-1} \in \mathcal{F}_n$ and all $1 \leq i \leq n$.

Theorem 6.1 *Under the Discrete Cube Root Assumption, the concept class $\mathcal{C}$ is not efficiently PAC learnable (using any polynomially evaluatable hypothesis class).*

Proof: Suppose for contradiction that $\mathcal{C}$ was PAC learnable in polynomial time by algorithm A. Then given Discrete Cube Root Problem inputs N and y, as before we can efficiently generate random examples for each of the n functions $f_{N,i}^{-1}$ by choosing x' randomly from Z_N^*, setting $y' = f_N(x')$, letting the example for $f_{N,i}^{-1}$ be (y', x'_i), where x'_i denotes the ith bit of x'. We thus run n separate simulations of A, one for each $f_{N,i}^{-1}$, setting the error parameter ϵ to be $1/n^2$ in each simulation. Now we can use the n hypotheses output by A to reconstruct all the bits of $f_N^{-1}(y)$ and by the union bound, the probability that all the bits are correct is at least $1 - 1/n$, contradicting the Discrete Cube Root Assumption. □(Theorem 6.1)

6.3 Small Boolean Circuits Are Inherently Unpredictable

One of the basic goals of learning theory is to understand how the computational effort required to learn a concept class scales with the computational effort required to evaluate the functions in the concept class. Thus we are not simply interested in whether there exist inherently unpredictable concept classes (and by Theorem 6.1, we now know that under there do, at least under the Discrete Cube Root Assumption), but in how "computationally simple" such classes could be. Obviously there are limits to how simple a hard-to-learn concept class can be. For instance, we already know that if every concept in a class can be computed by a 3-term DNF formula, then that class cannot be inherently unpredictable, because we can use the hypothesis class of 3CNF, for which there is a cubic time PAC learning algorithm.

Therefore, to further understand the implications of the inherent unpredictability result for the concept class C in Theorem 6.1, we must provide an upper bound on the resources required to evaluate a concept in C. We have already argued briefly that polynomial size boolean circuits suffice, but we now describe these circuits more precisely in order to pave the way to a refined construction and a considerably stronger hardness result in the next section.

Let us first rigorously define what we mean by a boolean circuit. A boolean circuit over $\{0,1\}^n$ is a directed acyclic graph in which each vertex has indegree (or **fan-in**) either 0,1, or 2, and unbounded outdegree (or **fan-out**). Each vertex of indegree 0 is labeled with one of the input variables $x_1, \ldots, x_n$. Each vertex of indegree 1 is labeled by the symbol $\neg$, and each vertex of indegree 2 is labeled by one of the symbols $\vee$ and $\wedge$. There is a single designated **output vertex** of outdegree 0. When the n input vertices are assigned boolean values, the graph computes a boolean function on $\{0,1\}^n$ in the obvious way. When we refer to the class of **polynomial size boolean circuits**, we mean the concept class

C in which each concept $c \in C_n$ is computed by a boolean circuit with at most $p(n)$ vertices, for some fixed polynomial $p(\cdot)$. In the following analysis, we are implicitly choosing the polynomial $p(\cdot)$ large enough to perform the required computations.

The circuit to compute the multivalued function $f_N^{-1} \in \mathcal{F}_n$ (from which we can easily extract circuits for the boolean functions $f_{N,i}^{-1} \in C_n$) will have the inverting exponent d for N "hard-wired". Therefore, the circuit only needs to compute $y^d \ mod \ N$. The trick for doing this efficiently (since d may be as large as n bits long, and we have already observed that we do not have time to multiply y by itself d times), is to first generate large powers of y by repeated squaring modulo N, and then combine these to obtain $y^d \ mod \ N$.

The repeated squaring of $y \ mod \ N$ yields the sequence of $\lfloor \log d \rfloor + 1$ numbers

$$y \ mod \ N, y^2 \ mod \ N, y^4 \ mod \ N, y^8 \ mod \ N, y^{16} \ mod \ N, \ldots, y^{2^{\lfloor \log d \rfloor}} \ mod \ N$$

using $\lfloor \log d \rfloor + 1$ sequential multiplications of n bit numbers. It is important to take the result so far $mod \ N$ at each step to prevent the numbers from becoming too long.

Now the appropriate elements of this sequence — exactly those corresponding to the 1's in the binary representation of d — can be multiplied together modulo N to obtain $y^d \ mod \ N$. This takes at most an additional $\lfloor \log d \rfloor + 1$ sequential multiplications. Since the multiplication of two $O(n)$-bit numbers can be implemented using circuits whose size is polynomial in n, and we need to perform only $O(\lfloor \log d \rfloor) = O(n)$ multiplications, the entire circuit for computing $y^d \ mod \ N$ has size polynomial in n.

Since we have just shown that the class of polynomial size circuits contains our hard class C, we immediately obtain the following result.

Theorem 6.2 *Under the Discrete Cube Root Assumption, the representation class of polynomial size boolean circuits is not efficiently PAC*

learnable (using any polynomially evaluatable hypothesis class).

6.4 Reducing the Depth of Inherently Unpredictable Circuits

Theorem 6.2 gives us our first hardness result for PAC learning a natural concept class that does not rely on artificial restrictions on the learning algorithm's hypothesis class. However, there is a sense in which it is the weakest such hardness result possible — after all, we cannot really hope to learn a class more powerful than polynomial size circuits in polynomial time.

In this section we will refine our construction of circuits that are hard to PAC learn in order to show that even very simple concept classes, such as the class of all boolean functions computed by *shallow* (that is, log-depth) polynomial size circuits, are inherently unpredictable. Furthermore, in Chapter 7, we will develop a notion of reducibility among learning problems that, combined with our refined hardness result for log-depth circuits, allows us to prove the inherent unpredictability of other important concept classes, such as the class of all concepts computed by deterministic finite automata.

Let us begin by analyzing the depth of the circuit we have proposed for computing the function $f_N^{-1} \in \mathcal{F}_n$. The circuit used the trick of repeated squaring $\lfloor \log d \rfloor + 1 = \Theta(n)$ times, and therefore the depth of the circuit is $\Theta(n)$. Furthermore, no shallower circuit for computing $y^d \ mod \ N$ from the input y is known.

Our goal is to prove that even circuits whose size is polynomial in n but whose depth (longest path from an input vertex to the output vertex) is at most $O(\log n)$ are hard to learn. More precisely, the class of **log-depth, polynomial size boolean circuits** is the concept class $\mathcal{C}$ in which each concept $c \in \mathcal{C}_n$ is computed by a boolean circuit with

at most $p(n)$ vertices and depth at most $k \log n$, for some fixed constant k (independent of n) and some fixed polynomial $p(\cdot)$. In the following analysis, we are implicitly choosing the constant k and the polynomial $p(\cdot)$ large enough to perform the required computations.

While the restriction to log-depth circuits may at first seem somewhat arbitrary, it is well-known that the class of log-depth, polynomial size circuits computes essentially the same functions as the rather natural class of polynomial size **boolean formulae**. (By this we mean that there exists a polynomial $p(\cdot)$ such that every log-depth circuit of size s can be represented as a boolean formula of size at most $p(s)$, and every boolean formula of size s can be represented by a log-depth circuit of size at most $p(s)$; see Exercise 6.2.) A boolean formula over $\{0,1\}^n$ is simply a well-formed expression over a logical language containing symbols for the usual boolean connectives $\vee, \wedge, \neg$, the symbols "(" and ")" for indicating order of evaluation, and symbols for the boolean input variables $x_1, \ldots, x_n$. Such an expression computes a boolean function over $\{0,1\}^n$ in the obvious way. A convenient alternative representation for a boolean formula is a boolean circuit in which the underlying graph must be a tree: at the root (output) node of this tree, we place the outermost connective of the boolean formula; inductively, the left and right subtrees of the root are the trees for the left and right subexpressions joined by the outermost connective in the formula. Figure 6.1 shows an example formula and the corresponding tree.

When we refer to the class of polynomial size boolean formulae, we actually mean the family in which each formula over $\{0,1\}^n$ is an expression of at most $p(n)$ symbols for some fixed polynomial $p(\cdot)$, where again we will implicitly choose the polynomial $p(\cdot)$ large enough to perform the required computations.

Intuitively, one primary difference between boolean formulae (log-depth circuits) and general boolean circuits is that if the same logical subexpression $E(x_1, \ldots, x_n)$ is needed many times, in a formula we may have to duplicate the expression with each use, while in a circuit we may simply increase the fan-out of the subcircuit computing $E(x_1, \ldots, x_n)$

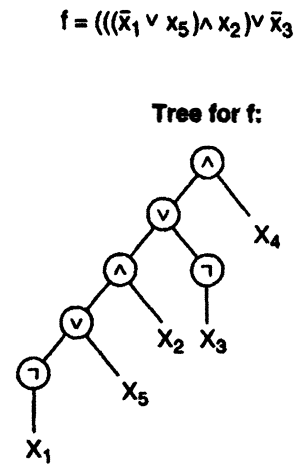

Figure 6.1: *A boolean formula and its tree circuit representation.*

and get the repetition for free. Thus, there may be some functions that can be computed by a small boolean circuit, but require a much larger boolean formula (although the existence of such functions remains an important open question), and we might wonder if it is these functions that cause the inherent unpredictability of small boolean circuits. We now show a negative answer to this question.

6.4.1 Expanding the Input

To show that shallow circuits are hard to learn, we shall modify each function $f_N^{-1} \in \mathcal{F}_n$ by providing additional inputs that make the computation of $f_N^{-1}(y) = y^d \ mod \ N$ possible using a shallow circuit, but that do not alter the difficulty of learning. In order to argue that learning remains hard, we will have to choose different hard input distributions.

The motivating idea behind the modification is actually quite simple. Suppose that knowing only the product N and a value y, we are watching

someone who also knows the decrypting exponent d for N perform the computation of $y^d \ mod \ N$ by the trick of repeated squaring of y, followed by multiplication of the appropriate square powers. If the entire computation is performed before us, then we can in fact learn the value of d from this computation, since the square powers of y multiplied together to obtain $y^d \ mod \ N$ correspond exactly to the binary representation of d, and we will have learned something we cannot obviously compute efficiently ourselves. However, if the party knowing d only computes the square powers in front of us, and then multiplies the appropriate powers together privately, then we have definitely *not* learned anything new: we could have efficiently computed these square powers of y ourselves. In this way, the party knowing d can reduce the amount of private computation to the bare minimum, without compromising the secrecy of d. In the following analysis, it is this private computation that corresponds to the circuit complexity of the target functions, which is reduced by this trick.

More precisely, for each f_N^{-1} with inverting exponent d, let us define a new function g_N^{-1} that is a mapping from $(Z_N^*)^{\lfloor \log d \rfloor + 1}$ to Z_N^*. For any $y \in Z_N^*$ we define

$$g_N^{-1}(y \ mod \ N, y^2 \ mod \ N, \ldots, y^{2^{\lfloor \log d \rfloor}} \ mod \ N) = y^d \ mod \ N = f_N^{-1}(y).$$

Thus, g_N^{-1} is essentially the same function as f_N^{-1} with one important difference: g_N^{-1} is provided with an "expanded input" in which the successive square powers of the original input y are already computed. Note that the length of the inputs to g_N^{-1} is $O(\log^2 N) = O(n^2)$ bits rather than the $O(\log N) = O(n)$ bits of input for f_N^{-1}, but is still polynomial in n. Furthermore, g_N^{-1} is simply the inverse of the vector-valued function

$$g_N(x) = (x^3 \ mod \ N, x^6 \ mod \ N, \ldots, x^{3 \cdot 2^{\lfloor \log d \rfloor}} \ mod \ N)$$

Thus, vectors in $(Z_N^*)^{\lfloor \log d \rfloor + 1}$ that are not of the successive square form are not in the range of g_N, and therefore g_N^{-1} will be defined to be the special value $*$ on such vectors.

The first important property we need of g_N^{-1} is that, like f_N^{-1}, it is hard to compute g_N^{-1} if the inverting exponent d for N is unknown. More precisely, if we let N be the product of two randomly chosen $n/2$-bit primes p and q such that 3 does not divide $\varphi(N) = (p-1)(q-1)$, and we choose y randomly in Z_N^*, then under the Discrete Cube Root Assumption it is hard to compute $g_N^{-1}(y \bmod N, y^2 \bmod N, \ldots, y^{2^{\lfloor \log d \rfloor}} \bmod N)$ on inputs N and

$$(y \bmod N, y^2 \bmod N, \ldots y^{2^{\lfloor \log d \rfloor}} \bmod N).$$

Otherwise, given inputs N and $y \in Z_N^*$ for the Discrete Cube Root Problem, a polynomial time procedure could compute the required powers of y modulo N by repeated squaring, thus obtaining the expanded input required for g_N^{-1}, and then invoke the procedure for computing g_N^{-1} to compute $f_N^{-1}(y)$. This would violate the Discrete Cube Root Assumption.

The second important property is that, *unlike* f_N^{-1},

$$g_N^{-1}(y \bmod N, y^2 \bmod N, \ldots y^{2^{\lfloor \log d \rfloor}} \bmod N)$$

can be computed by a shallow circuit that has d hard-wired. This circuit simply multiplies together the appropriate powers of y that are provided in the input sequence. Again, the numbers to be multiplied together are those powers $y^{2^i} \bmod N$ such that the ith bit of d is 1, as in the circuit for f_N^{-1}).

The problem of multiplying at most n numbers modulo N is a well-studied one and is known as the problem of **iterated products**. A naive implementation would multiply the desired numbers in pairs, and then the results in pairs, and so on, to get a circuit that is binary tree in which each internal node is a multiplication and each of the at most n leaves is one of the numbers to be multiplied. The depth of this tree is at most $\log n$. Unfortunately, since each internal node of the tree must be implemented by a multiplication circuit for two n-bit numbers, and this in itself requires circuit depth $\Omega(\log n)$, the final depth of this proposed circuit would be $O(\log^2 n)$ rather than $O(\log n)$. However, there is a sophisticated circuit construction due to Beame, Cook and Hoover (see

the Bibliographic Notes at the end of the chapter) that is beyond the scope of our investigation, but that provides circuits for iterated product of total depth only $O(\log n)$, as desired.

As with the functions f_N^{-1}, the functions g_N^{-1} are not boolean but multivalued. The definition of the associated boolean function (concept) class $\mathcal{C}'$ is completely analogous to the definition of the concept class $\mathcal{C}$ for the f_N^{-1}: for each function g_N^{-1} and each $1 \leq i \leq n$, we define the concept $g_{N,i}^{-1} \in \mathcal{C}_{n^2}$ to be the ith output bit of g_N^{-1}.

Now given Discrete Cube Root Problem inputs N and y, in the same way that a PAC learning algorithm for $\mathcal{C}$ could be used to obtain accurate approximations for all the output bits of f_N^{-1}, a PAC learning algorithm A for $\mathcal{C}'$ can be used to obtain accurate approximations for all the output bits of g_N^{-1}: we can set aside y and first generate random examples of g_N^{-1} by choosing x' randomly in Z_N^*, setting $y' = f_N(x')$, and computing the successive square powers. Setting

$$z' = (y' \bmod N, (y')^2 \bmod N, \ldots, (y')^{2^{\lfloor \log d \rfloor}} \bmod N)$$

we can compute the random example $\langle z', g_N^{-1}(z') \rangle$. The bits of x' are the boolean labels for the n functions $g_{N,i}^{-1}$ on the expanded input and can be used in n separate simulations of A. As with the argument for the $f_{N,i}^{-1}$, the n hypotheses output by A can then be used to compute $f_N^{-1}(y)$, contradicting the Discrete Cube Root Assumption. Notice that the hard distribution for the function $g_{N,i}^{-1}$ is not the uniform distribution over the input space $(Z_N^*)^{\lfloor \log d \rfloor + 1}$ but uniform over only those inputs that have the appropriate successive square form.

Since we have argued above that the concepts in $\mathcal{C}'$ are contained in the class of log-depth circuits, we have proved the following theorem:

Theorem 6.3 *Under the Discrete Cube Root Assumption, the representation class of polynomial size, log-depth boolean circuits (or equivalently, the class of polynomial size boolean formulae) is not efficiently PAC learnable (using any polynomially evaluatable hypothesis class).*

6.5 A General Method and Its Application to Neural Networks

We conclude this chapter by observing that en route to proving that the class of polynomial size boolean formulae is not efficiently PAC learnable, we in fact identified a general property of representation classes that renders them inherently unpredictable. In particular, the only special property we required of boolean formulae was their ability to efficiently compute the iterated product of a list of numbers. By formalizing this ability as a general property of representation classes, we will also be able to prove the inherent unpredictability of polynomial size neural networks.

Definition 15 *Let C be a representation class. We say that C* **computes iterated products** *if there exists a fixed polynomial $p(\cdot)$ such that for any natural number N of n bits and any $1 \leq i \leq n$, there is a concept $c \in C_{n^2}$ (thus, c has n^2 inputs) such that size$(c) \leq p(n)$, and for any $z_1, \ldots, z_n \in Z_N^*$, $c(z_1, \ldots, z_n)$ is the ith bit in the binary representation of the product $z_1 \cdots z_n$ mod N.*

Armed with this definition, by arguments identical to those used to derive Theorem 6.3, we obtain:

Theorem 6.4 *Let C be any representation class that computes iterated products. Then under the Discrete Cube Root Assumption, C is not efficiently PAC learnable (using any polynomially evaluatable hypothesis class).*

Recall that in Section 3.7 we demonstrated that the number of examples required to PAC learn any class of neural networks scaled only polynomially with the number of parameters required to specify the networks. This result ignored computational considerations, and concentrated just on the sample complexity of PAC learning. We now apply Theorem 6.4 to

show that the computational considerations are rather formidable. Our result relies on the following lemma due to J. Reif, whose proof is beyond the scope of our investigation (see the Bibliographic Notes at the end of the chapter).

Lemma 6.5 *(Reif) There is fixed polynomial $p(\cdot)$ and an infinite family of directed acyclic graphs (architectures) $G = \{G_{n^2}\}_{n \geq 1}$ such that each G_{n^2} has n^2 boolean inputs and at most $p(n)$ nodes, and for any natural number N of n bits there is an assignment of linear threshold functions to each node in G_{n^2} such that the resulting neural network computes iterated products modulo N. Furthermore, the depth of G_{n^2} is a fixed constant independent of n.*

In fact, Reif shows that Lemma 6.5 holds even when we are constrained to choose only weights in $\{0, 1\}$ for the linear threshold function at each node. From this lemma and Theorem 6.4, we immediately obtain:

Theorem 6.6 *Under the Discrete Cube Root Assumption, there is fixed polynomial $p(\cdot)$ and an infinite family of directed acyclic graphs (architectures) $G = \{G_{n^2}\}_{n \geq 1}$ such that each G_{n^2} has n^2 boolean inputs and at most $p(n)$ nodes, the depth of G_{n^2} is a fixed constant independent of n, but the representation class $\mathcal{C}_G = \cup_{n \geq 1} \mathcal{C}_{G_{n^2}}$ (where $\mathcal{C}_{G_{n^2}}$ is the class of all neural networks over $\Re^{n^2}$ with underlying architecture G_{n^2}) is not efficiently PAC learnable (using any polynomially evaluatable hypothesis class). This holds even if we restrict the networks in $\mathcal{C}_{G_{n^2}}$ to have only binary weights.*

6.6 Exercises

6.1. In this problem we consider the problem of computing discrete *square roots* rather than cube roots.

First, show that if $N = pq$ is the product of two primes p and q, and $a = x^2 \ mod \ N$ for some $x \in Z_N^*$, then there is a $y \in Z_N^*$ such that $a = y^2 \ mod \ N$ and $y \neq x \ mod \ N$ and $y \neq -x \ mod \ N$ (Hint: use the Chinese Remainder Theorem). Thus, any square a modulo N has two "different" square roots.

Now consider the **Discrete Square Root Problem**: given N that is the product of two $n/2$-bit primes, and an integer a that is the square modulo N of an element of Z_N^*, find an $x \in Z_N^*$ satisfying $a = x^2 \ mod \ N$. Show that if there is an efficient algorithm for the Discrete Square Root Problem, then there is an efficient algorithm for factoring integers, and vice-versa.

Thus the Discrete Square Root Problem is actually equivalent to factoring. With some mild additional assumptions on the numbers to be factored, this equivalence can be preserved by the techniques of this chapter to show that PAC learning the classes considered is as hard as a factoring problem; we chose to use discrete cube roots primarily for technical convenience.

6.2. Show that there is a fixed polynomial $p(\cdot)$ such that every log-depth boolean circuit of size s can be represented as a boolean formula of size $p(s)$, and every boolean formula of size s can be represented as a log-depth boolean circuit of size $p(s)$. Thus, within polynomial factors of size, these classes have equivalent computational power.

6.7 Bibliographic Notes

The first representation-independent hardness results for PAC learning follow from the influential paper of Goldreich, Goldwasser and Micali [43]. In this paper, it is shown (under a cryptographic construction) that polynomial-size boolean circuits are not efficiently PAC learnable, even if the input distribution is uniform, we only require weak learning, and membership queries are available (see Chapter 8).

The results of this chapter are due to Kearns and Valiant [60]. The Discrete Cube Root Problem was first proposed as the basis for the RSA public-key cryptosystem, named after its inventors Rivest, Shamir and Adelman [81]. The log-depth implementation of iterated products is due to Beame, Cook and Hoover [14]. Lemma 6.5 is due to Reif [76].

The Kearns and Valiant results were improved by Kharitonov [62], who showed that boolean formulae remain hard to PAC learn even if the input distribution is uniform, and membership queries are available. The Kharitonov results also apply to the class of constant-depth circuits of $\wedge$ and $\vee$ gates of unbounded fan-in. Interestingly (under an appropriately strong but still plausible assumption), these hardness results match the upper bound given for this class by an elegant learning algorithm due to Linial, Mansour and Nisan [64]. Angluin and Kharitonov [9] use cryptographic assumptions to demonstrate that membership queries cannot help for learning general DNF formulae.

The use of cryptographic tools and assumptions to obtain intractability results for learning is now fairly common in computational learning theory. In the reverse direction, a recent paper (Blum et al. [19]) demonstrates how certain assumptions on the difficulty of PAC learning problems can be used to obtain cryptographic primitives such as private-key cryptosystems and pseudo-random bit generators.