CSCI699: Machine Learning Theory Fall 2023

Lecture 5: Computational Complexity of Learning

Instructor: Vatsal Sharan

These lecture notes are based on scribe notes by Emir Ceyani, Jiahao Wen and Chandra Sekhar Mukherjee.

1 Computational Complexity of Learning

In the last few weeks, we've studied the sample complexity of learning. We saw complexity measures such as VC dimension, Rademacher complexity and algorithmic stability which allow us to analyze the number of samples need for a learning algorithm to generalize. Our discussion so far has ignored the computational costs associated with learning. That will be the focus of our investigation today.

In particular, we consider the *running time of the algorithm*, and explore what hypothesis classes can be learned in *polynomial time*.

Let us begin by recalling the definition of PAC learability:

Definition 1 (PAC learnability)**.** *A hypothesis class $\mathcal{H}$ is **PAC-learnable** if there exists a learning algorithm with the following property: $\forall \epsilon, \delta \in (0,1), \forall D \sim \mathcal{X}$ and $\forall h^* \in \mathcal{H}$, when the algorithm is given $n_{\mathcal{H}}(\epsilon, \delta)$ samples drawn from D and labelled by h^*, the algorithm produces a hypothesis $\hat{h}$ such that with probability $1-\delta$, $R(\hat{h}) \leq \epsilon$.*

Note that the probability is over randomness in training set, and any internal algorithmic randomness. This definition so far does not talk about efficiency. From your algorithms class, you probably recall that the usual notion of efficiency is that an algorithm should run in "polynomial time". But what does polynomial time mean in the learning setting, and what should the running time be polynomial with respect to?

Typically for an algorithmic task, we measure complexity with respect to the size of the problem instance. For example, for an algorithm to find shortest paths in a graph, we measure its complexity with respect to the size of the graph (the number of vertices and edges). What is the right notion of size of the problem instance in our learning setting?

A first guess could be to have the size of the training set as the size of the instance. However, the number of datapoints that the algorithm needs for learning is an important resource consideration as well, and it should be up to the algorithm to decide the minimum number of datapoints it needs.

To account for the cost of accessing training data, we can define an oracle (think of this as a black-box function) which gives a training datapoint to the algorithm. We define this as the *example oracle*

Definition 2 (Example Oracle)**.** *For any distribution D over $\mathcal{X}$ & hypothesis $h(x) : \mathcal{X} \rightarrow \mathcal{Y}$, we define* $\mathrm{Ex}(\mathcal{L}, D)$ *as the **example oracle** which executes the following steps:*

- *Draws $x \sim D$*

- *Labels* $y = h(x)$
- *Outputs* $(x, h(x))$

With this definition of an example oracle, we can define *sample complexity* as **the number of example oracle calls**. We will regard each oracle call costs unit time to the learner. Think of this oracle as a button, which the learner can press at any time to demand a labelled datapoint. The learner does not have to worry about the complexity of "implementing this button", or more formally implementing this oracle. Such an abstraction is useful because, as we will see later, we can define various other meaningful oracles as well which dictate how the learner accesses information about the data distribution.

Let us go back now to defining how we will measure the size of the problem instance. Our notion is straightforward, we just measure the instance size of each datapoint. Hence an efficient algorithm will be required to run in time polynomial in the instance size. We will typically think of the instance space as being $\mathcal{X}^d$ where $\mathcal{X}$ is $\{\pm\}^d$ or $\mathbb{R}^d$. Therefore, we want the algorithm to run in time polynomial in the feature dimension d.

The dimensionality of the data is the measure of instance size we will use, but there is one subtlety we should discuss once (and can mostly forget after that). When we talk about efficient algorithms, we need to also allow the runtime to depend polynomially on the *in representation size of the hypothesis class.*

So what is the representation size of an hypothesis class? Essentially this is just the number of bits required to write down any hypothesis in hypothesis class. For most reasonable hypothesis classes, this would be about the number of parameters which defines a function in the hypothesis class. As an example for a feed-forward neural network with instance size d (input) As an example for a feed-forward neural network,

$$\text{representation size} = \#\text{edges} \times \#\text{bits required to store each weight.}$$

However, for all hypothesis classes we consider in class,

$$\text{representation size} = \text{poly(instance size of datapoints)}.$$

Therefore for all problems that we consider,

$$\text{poly(representation size, instance size)} = \text{poly(instance size)}.$$

Therefore, we only consider **polynomial in instance size** as our notion of efficiency.

Definition 3 (Efficient PAC Learning)**.** *A hypothesis class $\mathcal{H}$ is PAC-learnable if there exists a learning algorithm A with the following property: $\forall \epsilon, \delta \in (0,1), \forall D \sim \mathcal{X}^d$(where $\mathcal{X}$ is typically $\{0,1\} or \mathcal{R}$) and $\forall h^* \in \mathcal{H}$, if A is given access to example oracle* $\mathrm{Ex}(h^*, D)$, *with probability* $1-\delta$, *it outputs a hypothesis $h \in \mathcal{H}$ with $R(\hat{h}) \leq \epsilon$. $\mathcal{H}$ is efficiently PAC learnable if running time of A is **polynomial** in $d, \frac{1}{\epsilon}, \frac{1}{\delta}$.*

As an example of a hypothesis class that can be learned efficienctly, we consider the class of conjunctions.

1.1 Conjunctions

A conjunction is an example of a Boolean function, which is a function with input domain $\mathcal{X}^d = \{0,1\}^d$. As an example consider

$$x_1 \wedge \bar{x}_3 \wedge x_4.$$

This represents the hypothesis class which is 1 if and only if $x_1 = 1, x_3 = 0,$ and $x_4 = 1$.

The class of conjunctions refers to the set of all possible conjunctions on d Boolean literals $(x_1, x_2, \dots x_d)$ and their negations $(\bar{x_1}, \bar{x_2}, \dots \bar{x_d})$. Can we design a poly$(d, 1/\epsilon, 1/\delta)$ time algorithm for learning conjuctions? The answes is yes:

Theorem 4. *The class of conjunctions on Boolean literals is efficiently PAC learnable.*

Proof. We will prove that the following algorithms is an efficient PAC learner for the class of conjunctions.

Algorithm 1 Algorithm to learn Boolean Conjunctions

```
Set $h = x_1 \vee \bar{x}_1 \vee x_2 \vee \bar{x}_2 \dots x_d \vee \bar{x}_d$
for $i = 1$ to $n$ do do
    $(a_i, y_i) \leftarrow \mathrm{EX}(h^*, D)$
    if $t_i = 1$ then
        Drop each $\bar{x}_j$ from $h$ if $(a_i)_y = 1$
        Drop each $x_j$ from $h$ if $(a_i)_y = 0$
    end if
end for
```

We will show that the above algorithm is an ERM over the class of conjunctions. This means that we need to show that the algorithm gets 0 misclassification error over training examples $(a_1, y_1), \dots, (a_n, y_n)$. We consider two cases.

- We claim that the algorithm never predicts 0 on a training datapoint which is labelled as 1. This is because after getting any new data point a_i with $y_i = 1$, h updates to predict 1 on the datapoint and it will never be updated predict 0 on a_i, as literals are only removed.
- Next, we claim that h correctly classifies all training data labelled as 0. Note that in the beginning, h is the conjunction of every literal . We remove a literal from h if it was get to 0 in an example. Such a literal cannot appear in h^*. Therefore the set of literals appearing in h at any time, contains the set of literals in target hypothesis h^*. Therefore $h(a) = 1 \implies h^*(a) = 1, \forall a \in \{0,1\}^d$. This proves that h correctly all training datapoints labelled as a 0.

Therefore we have shown that our algorithm is an ERM.

Now, we use ERM result for finite hypothesis classes as $|\mathcal{H}| \le 2^{2d}$. The result for learnability of finite hypothesis classes implies that we can learn with error ϵ with failure probability δ with $O\left(\frac{d \log(\frac{1}{\delta})}{\epsilon}\right)$ samples.

■

2 Intractability of learning 3-Term Disjunctive Normal Forms(DNFs)

We now consider the class of 3-Term Disjunctive Normal Forms(DNFs), which are only a slight generalization of the class of conjunctions.

$$\text{3-Term-DNF}_\text{d} = \{T_1 \vee T_2 \vee t_3 \ : \ T_i \text{is a conjunction on } \{x_1, \ldots x_d\}\}$$

Though conjunctions were efficiently PAC learnable, this generalization turns out to be hard to learn.

Theorem 5. *3-Term-DNF formulae* <u>*are not*</u> *efficiently PAC-learnable unless RP=NP.*

Before proving the theorem, let us recap some fundamental concepts in computational complexity.

2.1 Computational Complexity Review

The first definition we need is for the complexity class NP.

Definition 6 (NP)**.** *A decision problem C is in NP if there exists a polynomial-time algorithm A such that for every instance x of C,*

- *If x evaluates to"yes", then $\exists y,\ |y| \leq \text{poly}(|x|), A(x,y) = 1$*
- *If x evaluates to "no", then $\forall y,\ |y| \leq \text{poly}(|x|), A(x,y) = 0$*

The intuition for the definition is that A is some verifier who can verify solutions to the decision problem. y is a certificate or witness which A can check to verify the solution to the problem.

As an example, consider 3SAT on d variables

$$\begin{aligned} \text{3SAT}_\text{d} =& (x_1 \vee \bar{x}_2 \vee x_5) \\ & \wedge (x_5 \vee x_6 \vee \bar{x}_7) \\ & \vdots \\ & \wedge (x_{d-2} \vee \bar{x}_{d-1} \vee x_d). \end{aligned}$$

Here, the certificate y is just a satisfying assignment. The verifier just checks if the certificate/assignment is valid, note that it is easy to check this (it can be done in running time which is linear in the number of variables). To complete the proof that 3SAT_d is in NP, note that there always exists a certificate if the instance is satisfiable, and there does not exist any certificate if the problem is not satisfiable.

Our next complexity class is a class of randomized polynomial time algorithms.

Definition 7 (RP)**.** *A decision problem C is in RP if there exists a randomized polynomial-time algorithm A such that for every instance x of C,*

- *If x evaluates to "yes", A outputs "yes" w.p. $\geq \dfrac{2}{3}$*

- *If x evaluates to "no", A outputs "no" w.p. 1.*

The intuition for this definition is that C is in RP if there exists a polynomial-time algorithm with *one-sided error.* In particular, for the case of 3SAT, if the 3SAT instance is satisfiable, then the algorithm will output TRUE w.p. $\geq \frac{2}{3}$. If the 3SAT instance is not satisfiable, then the algorithm will always output FALSE.

It is widely believed that RP $\neq$ NP. This is a slightly stronger version of the famous P $\neq$ NP conjecture. The P $\neq$ NP conjecture states that polynomial time algorithms cannot solve all problems in NP. RP $\neq$ NP claims that even randomized polynomial time algorithms cannot solve all problems in NP. We believe that RP $\neq$ NP, because there is strong evidence that RP = P.

Our final definition is of NP-Completeness.

Definition 8 (NP-Completeness)**.** *A decision problem C is in NP-Complete if,*

- *C is in NP*
- *Every decision problem C' in NP can be reduced to C in polynomial time.*

Intuitively, if a problem is NP-Complete that solving that problem in polynomial time is sufficient for solving all problems in NP in polynomial time.

2.2 Proof of hardness of learning 3-term DNF

Let us now prove Theorem 5. Though this proof is for hardness of 3-term DNF, this recipe of showing hardness is generally quite useful.

The key idea is to reduce a NP-complete problem to the problem of learning 3-term DNF. The key property we want from mapping is that the answer to decision problem is "Yes" if and only if a set of labelled examples is *consistent* with some hypothesis $h \in \mathcal{H}$.

Definition 9. *Let $U = \{(a_1, y_1), ..., (a_n, y_n)\}$ be labelled set of instances. Let h be any hypothesis. We say that h is consistent with U if $\forall\, i \in [n], h(a_i) = y_i$.*

We now state the NP-complete problem we use to show hardness for 3-term DNF.

Definition 10 (graph 3-Coloring)**.** *Given an undirected graph $G = (V, E)$ with vertex set $V = 1, \ldots, d$, is there any assignment from every vertex $v \to \{R, B, G\}$, such that for every edge $e \in E$, the endpoints of e are assigned different colors?*

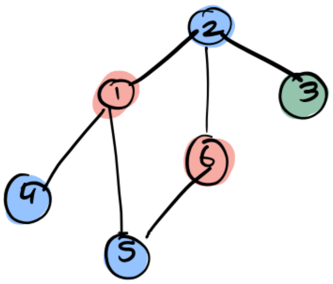
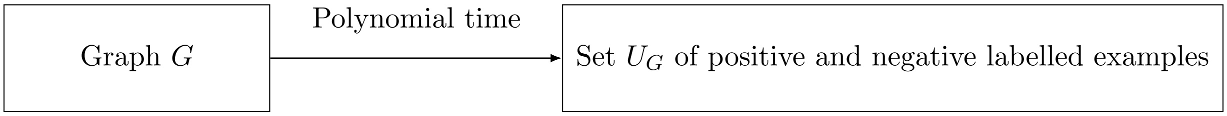


Figure 1: An example of graph 3-Coloring.

Note that graph 3-Coloring is NP-complete.

Reduction

Our goal is to show that learning 3-Term DNF is at least as hard as graph 3-coloring. Therefore we will use an algorithm for learning 3-Term DNF to solve graph 3-coloring. The first step in the reduction is to convert a graph into some set U_G of positive and negative examples, such that U_G is consistent with some $h \in \mathcal{H}$ if and only if G is 3-Colorable.



Graph G

Polynomial time

Set U_G of positive and negative labelled examples

Let us see why this is sufficient. We will use a PAC-learning algorithm for 3-Term DNF with the following parameters:

- D: uniform on U_G.
- $EX(h^*, D)$: pick a point uniformly at random from U_G.
- δ: 1/3.
- ϵ: $1/2|U_G|$.

Note that D is uniform over set of size $|U_G|$, $\epsilon = \dfrac{1}{2|U_G|}$ and the algorithm is required to output a hypothesis h with $R(h) < \epsilon$. Therefore, if the algorithm succeds then h must be consistent with set U_G, i.e. $R(h) = 0$.

All that remains is to show we can construct U_G such that there is some consistent hypothesis $h \in \mathcal{H}$ if and only if G is 3-Colorable.

Algorithm 2

```
Given instance of 3-Color, construct set $U_G$.
Use PAC-learning algorithm $A$ for 3-term DNF with $EX(h^*, D), \delta = 1/3, \epsilon = 1/2|U_G|$.
Let $h$ be the 3-term DNF returned by $A$.
if $h$ is consistent with $U_G$ then
    return "Yes".
else
    return "No".
end if
```

Constructing U_G

$$\begin{aligned} & U_G = U_G^+ \cup U_G^- \\ & U_G^+\text{: positive examples} \\ & U_G^-\text{: negative examples} \end{aligned}$$

To construct U_G^+, for every vertex i in the graph, we create a positively labelled example which is 0 at the index i and 1 everywhere else. For the example shown in Fig. 1, we construct U_G^+ as follows:

$$\begin{aligned} & |U_G^+| = |V| \\ & (v(1), +1) = ((0, 1, 1, 1, 1, 1), +1) \\ & (v(2), +1) = ((1, 0, 1, 1, 1, 1), +1) \\ & \vdots \\ & (v(6), +1) = ((1, 1, 1, 1, 1, 0), +1) \end{aligned}$$

To construct U_G^-, for every edge (i, j) in the graph, we create a negatively labelled example which is 0 at the coordinates i and j and 1 everywhere else. For Fig. 1, we have

$$\begin{aligned} & |U_G^-| = |E| \\ & (e(1, 2), -1) = ((0, 0, 1, 1, 1, 1), -1) \\ & (e(1, 4), -1) = ((0, 1, 1, 0, 1, 1), -1) \\ & \vdots \\ & (e(5, 6), -1) = ((1, 1, 1, 1, 0, 0), -1) \end{aligned}$$

We will now show that there is some consistent 3-Term DNF if and only if G is 3-Colorable. The proof is in two parts.

Part I: 3-Colorable $\implies$ there exists a consistent 3-term DNF

Consider a 3-term DNF $\phi = T_R \cup T_B \cup T_G$ where

- R: set of all vertices colored red.
- B: set of all vertices colored blue.

- G: set of all vertices colored green.

Let T_R be the conjunction of all variables whose index doesn't appear in R. $\rightarrow T_R = x_2 \cap x_3 \cap x_4 \cap x_5$. Similarly, we can get $T_B = x_1 \cap x_3 \cap x_6$ and $T_G = x_1 \cap x_2 \cap x_4 \cap x_5 \cap x_6$.

For each $i \in R$, example $v(i)$ must satisfy T_R because x_i doesn't appear in T_R.

Further, no $e(i,j) \in U_G^-$ can satisfy T_R. Both i and j cannot be colored red at the same time, one of x_i or x_j must appear in T_R. But $e(i,j)$ has 0 values for both x_i and x_j. So, T_R cannot be satisfied by $e(i,j)$. The same argument follows for T_B and T_G, and we have therefore shown that $\phi = T_R \cup T_B \cup T_G$ is consistent.

Part II: Consistent 3-term DNF $\implies$ 3-colorable

Let $\phi = T_R \cup T_B \cup T_G$ be a consistent 3-term DNF.

For a vertex i, if $v(i)$ satisfies T_R, color i red. Similar with T_B and T_G. (Break any ties arbitrarily.)

Since formula is consistent, every $v(i)$ must satisfy at least one of T_R, T_G, T_B. So every vertex is assigned a color.

Claim 11. *The coloring is a valid 3-Coloring.*

Proof. If i and $j (i \neq j)$ are assigned the same colors (say red), both $v(i)$ and $v(j)$ satisfy T_R.

$$\begin{aligned} v(i) &= (1, \ldots, 0, \ldots, 1, \ldots, 1) \\ v(j) &= (1, \ldots, 1, \ldots, 0, \ldots, 1) \\ e(i,j) &= (1, \ldots, 0, \ldots, 0, \ldots, 1) \end{aligned}$$

Since i-th bit of $v(i)$ is 0 and i-th bit of $v(j)$ is 1, we can infer that neither x_i nor $\bar{x}_i$ appears in T_R. We can see $e(i,j)$ and $v(j)$ only differs in i-th coordinate. If $v(j)$ satisfies T_R, so does $e(i,j)$. Then $e(i,j)$ should be labelled positive. So $e(i,j) \notin U_G^-$ and $(i,j) \notin E$. ■

3 Using 3-CNF formulae to avoid intractability

So far, we restricted the learning algorithm to output a hypothesis from the same class it was learning. What if we allow the algorithm to output a hypothesis from a different, more expressive class?

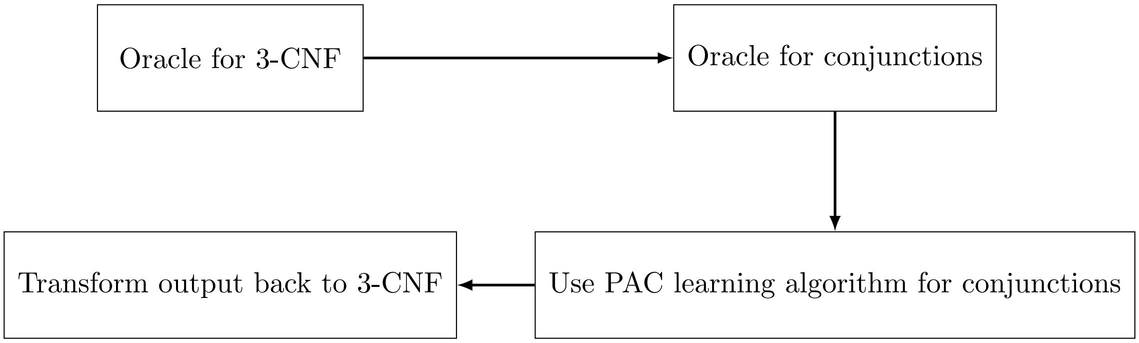
We can use the distributive law to rewrite the 3-term DNF:

$$(u \cap v) \cup (w \cap x) = (u \cup w) \cap (u \cup x) \cap (v \cup w) \cap (v \cup x)$$

We can represent any 3-term DNF_d $\phi = T_1 \cup T_2 \cup T_3$ by a 3-CNF_d $\phi = \cap_{u \in T_1, v \in T_2, w \in T_3} (u \cup v \cup w)$. This is a 3-CNF (conjunctive normal form).

Theorem 12. *The class of 3-CNF formulae is efficiently PAC-learnable.*

Proof. We will reduce the problem of PAC learning 3-CNF formulae to the problem of PAC learning conjunctions.


Oracle for 3-CNF
Oracle for conjunctions
Use PAC learning algorithm for conjunctions
Transform output back to 3-CNF

Idea: Regard 3-CNF formulae as a conjunction over a new and larger variable set.

Transformation: For every triple of literals, u, v, w over the original variable set $\{x_1, \ldots, x_d\}$. The new variable set contains a variable $y_{u,v,w} = u \cup v \cup w$. Note that when $u = v = w$, $y_{u,v,w} = u$, so all the original variable are in the new set. The number of variables is $(2d)^3$, $O(d^3)$. Note that any 3-CNF over $x_1, \ldots, x_d$ is equivalent to a conjunction over $\{y_{u,v,w}\}$. (Replacing any clause $u \cup v \cup w$ by $y_{u,v,w}$.)

Transforming oracles: For any assignment $a \in \{0,1\}^d$ to original variable, we can compute assignment to new variable set $\{y_{u,v,w}\}$ in $O(d^3)$ time.

Use conjunction learning algorithm: We can now run algorithm for conjunctions. Finally, we can transform the output h' of the algorithm back to a 3-CNF h, by expanding any occurrence of $y_{u,v,w}$ by $(u \cup v \cup w)$.

Claim 13. *If h^* and D are the target 3-CNF formula and the distribution over $\{0,1\}^d$, and $h^{*\prime}$ and D' are the corresponding conjunction over $y_{u,v,w}$ and the corresponding distribution over $y_{u,v,w}$, then if h' has errors $\leq \epsilon$ with respect to $h^{*\prime}$ and D', then h has errors $\leq \epsilon$ with respect to h^* and D.*

Proof. We simply note that our transformation of instances is one → one.

$$\text{If } a_1 \to a_1' \text{ and } a_2 \to a_2'$$
$$a_1 \neq a_2 \implies a_1' \neq a_2'.$$

Therefore, our transformation preserves the error with respect to the original and transformed instances. ■

This completes our polynomial time reduction. ■

Important takeaway: The choice of representation/hypothesis can make the difference between efficient algorithms and intractability. Going to a more expressive (richer) hypothesis classes (For example, from 3-Term DNF to 3-Term CNF) can make learning efficient. Statistically, learning over a richer hypothesis class can never help if you know the target class is in a smaller hypothesis, but when computational efficiency is brought into the picture, the picture can be very different.

To account for this, we make the following distinction:

Definition 14 (Concept and Hypothesis Class)**.**

- *The concept class C is the class to which the ground-truth hypothesis originally belongs.*
- *The hypothesis class $\mathcal{H}$ is the class from which the learner chooses its hypothesis.*

Against this backdrop, we have a revised definition for efficient PAC learning.

Definition 15 (Proper and Improper PAC Learning)**.** *If C is a concept class over the instance space $\mathcal{X}^d$ and $\mathcal{H}$ is a hypothesis class over $\mathcal{X}^d$, we say that C is (efficiently) PAC learnable using $\mathcal{H}$ if our basic definition of PAC learning is met by an algorithm which is allowed to output a hypothesis from $\mathcal{H}$. Here we implicitly assume that $\mathcal{H}$ is at least as expressive as C. (This implies that there is a representation in $\mathcal{H}$ for every function in C.)*

- *If $C = \mathcal{H}$ then the algorithm is called a proper learning algorithm.*
- *If $C \subset \mathcal{H}$ then the algorithm is called an imporper learning algorithm.*

Are there learning problems which are hard even improperly?

To answer this we look into representation independent hardness results for learning, and explore some connections between learning theory and cryptography.

4 Representation Independent Hardness Results for Learning

In some sense, cryptography and learning are two sides of the same coin. Problems which are hard to learn are possible cryptographic primitives, and cryptographic primities can be used to derive learning problems which are hard to solve. A key similarity between cryptography and learning is that both care about solving problems *on average*, when instances come from some distribution.

We currently do not know how to base cryptography on worst-case hardness, such as assuming that $P \neq NP$. Instead, cryptography must rely on stronger average-case assumptions. One widely used notion is that of *one way functions.*

Definition 16 (One Way Functions)**.** *A one-way function $f : \{0,1\}^d \to \{0,1\}^d$ is one that is easy to compute, but hard to invert.*

More formally, f can be computed in polynomial time but for any randomized polynomial time algorithm A and for any polynomial $p()$, we have

$$Pr\left[f(A(f(*)) = f(x)\right] \leq \frac{1}{p(d)}$$

where the probability is taken over x drawn uniformly from $\{0,1\}^d$, and randomness in A.

Proving that one-way functions exists is harder than showing $P \neq NP$, and we currently do not even know how to show that they exist even assuming $P \neq NP$. However, there are several functions which are good candidates and are widely believed to be one-way functions. In fact, much of internet commerce and secure communication hinges on this assumption being true.

Discrete Cube Root: A candidate one way Function

Let $N = p \cdot q$ b a product of two primes of roughly equal length. Let $f_N(x) = x^3 \bmod N$.

If one knows p and q, it can be shown that this function is invertible in polynomial time (using Euclid's GCD algorithm). However, it is widely believed that this function is hard to invert without knowledge of p and q. This forms the basis of the famous RSA cryptosystem.

For a fixed L, let F be the family of all functions:

$$F = \{f_N(x), N = p \cdot q, p \text{ and } q \text{ are primes}, \text{length}(p), \text{length}(q) \leq d\}$$

The Discrete Cube Root Assumption (DCRA) states that given N and $y = f_N(x)$ for some random $x \in \{0,1\}^d$, it is hard to compute x in polynomial time. Next lecture we will show how this assumption can be used to derive a learning problem which is hard to solve using any representation class.

5 Further Reading

The best reference for today's lecture, and the next few lectures is the book [1], which should be available using your USC account here. Sec 1.3 in Chapter 1 covers learning conjunctions (via a different proof from the one in class though), and 1.4 and 1.5 cover hardness of learning 3-term DNF and using CNFs to avoid intractability. You can read Chapter 6 of Kearns-Vazirani for more on how cryptographic assumptions give hard learning problems, including more discussion for the Discrete Cube Root problem.

References

[1] Moritz Hardt and Benjamin Recht. Patterns, predictions, and actions: A story about machine learning. *arXiv preprint arXiv:2102.05242*, 2021.