6.895 Randomness and Computation March 19, 2008

Lecture 13

Lecturer: Ronitt Rubinfeld *Scribe: Edwin Chen*

1 Overview

1.1 Last Lecture: Boosting Weak Learners Into Strong Learners

In the last two lectures, we showed that if any concept class can be weakly learned, then it can also be strongly learned. Our proof used a Boosting Algorithm that could boost any weak PAC-learner (an algorithm that could learn to do slightly better than average) into a strong PAC-learner (an algorithm that could learn to do a lot better than average).

1.2 This Lecture: Hard Functions, Uniformity, and Derandomization

The Boosting Algorithm we used was based on what was originally a complexity theory result. (It took the Machine Learning theorists a while to realize that it gave them a boosting algorithm.)

In this lecture, we mention the original complexity result (on the existence of a "hard" set of inputs). We use it to prove Yao's XOR Lemma, an analogous theorem for boosting hardness of functions: from any function that is *slightly*-hard-to-compute using small circuits, we can produce a *very*-hard-to-compute function.

Afterwards, we discuss the non-uniformity property of circuits that allows us to "cheat" to prove this result. We then look at randomized complexity classes and consider the question of whether randomness truly helps, thus entering the derandomization section of this course.

2 Yao's XOR Lemma

2.1 Definition of Hard

We would like to construct monstrously-hard functions from slightly-hard functions, because this will allow us to show certain complexity classes are hard-on-average if they are even slightly-hard. But first, what do we mean for a function to be *hard* to compute?

Definition 1 *A function $f : \{\pm 1\}^n \to \{\pm 1\}$ is* δ-hard on distribution D for size g *if for any Boolean circuit*[1] *C with at most g gates*[2]*, $\Pr_{x \in_D \{0,1\}^n}[C(x) = f(x)] \leq 1 - \delta$.*

In other words, f is δ-hard if there is some δ-fraction of the inputs that is hard to compute.

For example, if $\delta = 1 - 2^{-n}$, then there is no circuit of size g that can exactly compute f (there is always some input on which f is wrong); if $\delta = \frac{1}{2}$, then no circuit of size g does better than random guessing.

Our goal is to construct a function that is hard to compute for close to $\frac{1}{2}$ of the inputs; that is, we want to find a function f such that $\Pr_{x \in_D \{0,1\}^n}[C(x) = f(x)] \leq \frac{1}{2} + \epsilon$.

Definition 2 *Let M be a measure.*[3] *If $\mathrm{Adv}_C(M)$*[4] *$< \epsilon |M|$*[5] *for any circuit C of size g, we say that f is* ϵ-hard-core on M for size g.

[1]We assume that our circuits are of bounded fan-in.

[2]Recall that we can describe any function with a circuit of at most an exponential number of gates. Thus, in order to say anything interesting, we need to assume a bound on the size of our circuits

[3]Recall that the "measure" of an example x is the probability that the filter from our Boosting Algorithm keeps example x.

[4]$\mathrm{Adv}_C(M) = \sum_x R_C(x)M(x)$ is the advantage of C on M, where $R_C(x) = +1$ if $f(x) = C(x)$ and -1 otherwise. In other words, the advantage is the expected value of the number of examples x on which f is correct minus the number of examples on which f is wrong

[5]$|M| = \sum_x M(x)$ is the total "mass" of all examples according to M, or the expected number of examples that we keep.

When M is the characteristic function of a set S, we make the following, perhaps easier-to-understand, definition.

Definition 3 *Let S be a set. We call f* ϵ-hard-core on S for size g *if for any circuit C with at most g gates,* $\Pr_{x \in_U S}[C(x) = f(x)] \leq \frac{1}{2} + \frac{\epsilon}{2}$.

In other words, f is ϵ-hard-core on S if every circuit can do at most $\frac{\epsilon}{2}$-better than randomly guessing the value of f on S; note that the bigger ϵ is, the easier f is to compute.

We will show below that for every function f, there is a hard-core set on which f's hardness is concentrated. That is, a function's hardness is not equally spread out everywhere; some inputs are harder to compute than others.

2.2 Yao's XOR Lemma and Some Motivation

We now state Yao's XOR Lemma, which says that for any hard function f, the XOR of many copies of f is even harder. Intuitively, this is true because computing XOR requires knowing either the value of f at each input or how many times we are wrong.

Definition 4 *For any function f, define a new function*

$$f^{\oplus k}(x_1, \ldots, x_k) = f(x_1)f(x_2)\cdots f(x_k).$$

We can think of $f^{\oplus k}$ as a parity function computing the number of $f(x_i)$ that take on the value -1.[6]

Theorem 5 (Yao's XOR Lemma) *If a function f is δ-hard for size g, then $f^{\oplus k}$ is $2(\epsilon' + (1-\delta)^k)$-hard-core over $\{\pm 1\}^n$ for size $g' = \Theta(\epsilon'^2\delta^2 g)$. (Equivalently, $f^{\oplus k}$ is $\frac{1}{2} - (\epsilon' + (1-\delta)^k)$-hard.)*

Before proving this theorem, we give some further motivation why this result should hold.

Suppose we have a δ-biased coin that lands heads with probability δ and tails with probability $1-\delta$. What is the best algorithm for predicting the outcome of the coin? If $\delta \geq \frac{1}{2}$, we should simply output "heads" each time; otherwise, we should output "tails". The probability that we correctly predict the outcome of one toss is then $\max\{\delta, 1-\delta\}$.

What if we want to predict the XOR of k tosses? In this case, we can show that under the best possible algorithm,

$$\Pr[\text{correctly predict XOR of } k \text{ tosses}] = \frac{1}{2} + (1-2\delta)^k.$$

In other words, while we can do pretty well at predicting the outcome of one coin toss, we can only do slightly better than random at predicting the XOR of k tosses.

Also, note the similarity between the $\frac{1}{2} - (\epsilon' + (1-\delta)^k)$ term in Theorem 5 and the $\frac{1}{2} + (1-2\delta)^k$ term in this equation. (We can think of the ϵ' term in the first expression as noise.)

2.3 A Proof of Yao's XOR Lemma

To prove Yao's XOR Lemma[7], we will assume the following results.

Theorem 6 (Impagliazzo Hard-Core Set Theorem) *Let f be ϵ-hard for size g on the uniform distribution on n-bit strings, and let $0 < \delta < 1$. Then there is a measure M with $\mu(M) \geq \epsilon$ such that f is δ-hard-core on M for size $g' = \frac{\epsilon^2\delta^2 g}{4}$.*

[6] This is in the $\{\pm 1\}$ basis; in the $\{0,1\}$ basis, $f^{\oplus k}$ is still a parity function, but looks instead like $f^{\oplus k}(x_1, \ldots, x_k) = f(x_1) \oplus f(x_2) \oplus \cdots \oplus f(x_k)$.

[7] This proof of Yao's XOR Lemma is simply one with a close connection to learning theory (in particular, boosting) and complexity theory. There are heaps upon heaps of others!

Lemma 7 *Let M be a measure such that $\mu(M) \geq \delta$. If f is $\frac{\epsilon}{2}$-hard-core on M for size $2n < g < \frac{1}{\delta}\frac{2^n(\epsilon\delta)^2}{8\delta n}$, then f is ϵ-hard-core for size g on some set S such that $|S| \geq \delta 2^n$.*

Corollary 8 *If f is δ-hard for size g, then f has a 2ϵ-hard-core set S for size $g' = \frac{\epsilon^2\delta^2 g}{4}$, and $|S| \geq \delta 2^n$.*

In other words, these results say that for any hard function f, there is always a fairly large core on which f's hardness is concentrated. Theorem 6 and Lemma 7 are proved in Homework 3. Corollary 8 easily follows from them. The proofs should be fairly short and simple (if not, you are doing them wrong!), using ideas from the Boosting Algorithm we went over in class.

It is easy to see that these hard-core results, together with the following lemma, prove our theorem.

Lemma 9 (Missing Lemma) *Assume f is $2\epsilon'$-hard-core for size g' on a set H such that $|H| \geq \delta 2^n$. Then $f^{\oplus k}$ is $2(\epsilon' + (1-\delta)^k)$-hard-core over $\{\pm 1\}^n$ for size g'.*

That is, the Missing Lemma says that no circuit of size at most g' correctly computes $f^{\oplus k}$ on more than $\frac{1}{2} + \epsilon' + (1-\delta)^k$ of the inputs.

To complete the proof of Yao's XOR Lemma, it remains to show the Missing Lemma.

Proof of Missing Lemma Suppose that f is $2\epsilon'$-hard-core on a set H, $|H| \geq \delta 2^n$, for size g'. For contradiction, assume also that there exists a circuit C of size at most g' which satisfies

$$\Pr_{x_1,\ldots,x_k \in \{\pm 1\}^n}[C(x_1,\ldots,x_k) = f^{\oplus k}(x_1,\ldots,x_k)] \geq \frac{1}{2} + \epsilon' + (1-\delta)^k. \tag{1}$$

Our plan is to use C to produce a new circuit C' with at most g' gates such that

$$\Pr_{x \in H}[C'(x) = f(x)] \geq \frac{1}{2} + \epsilon',$$

contradicting our assumption that f is $2\epsilon'$-hard-core.

Let A_m denote the event that exactly m of $x_1,\ldots,x_k$ are in H. (Exactly m of the x_i are in the "hard part" and exactly $k-m$ are in the "easy part".) Then the probability that none of the x_i are in H is at most

$$\Pr[A_0] = \Pr[\text{none of the } x_i \text{ are in } H] \leq (1-\delta)^k,$$

and combining this with our assumption in (1), we can lower-bound the probability that C correctly computes $f^{\oplus k}$ given that at least one of the x_i landed in H:

$$\Pr_{x_1,\ldots,x_k}[C(x_1,\ldots,x_k) = f^{\oplus k}(x_1,\ldots,x_k) | A_m \text{ for some } m > 0] \geq \frac{1}{2} + \epsilon'. \tag{2}$$

By an averaging argument, this means there must exists a specific $m > 0$ such that

$$\Pr_{x_1,\ldots,x_k}[C(x_1,\ldots,x_k) = f^{\oplus k}(x_1,\ldots,x_k) | A_m] \geq \frac{1}{2} + \epsilon'.$$

Now suppose we are given some random $x \in H$, and we want to compute $f(x)$. Using our result in (2), for a random selection of

$$x_1,\ldots,x_{m-1} \in_R H$$

$$y_{m+1},\ldots,y_k \in_R H,$$

and a random permutation π on k elements, we have

$$\Pr_{x,x_i\text{'s},y_j\text{'s},\pi}[C(\pi(x_1,\ldots,x_{m-1},x,y_{m+1},\ldots,y_k)) = f^{\oplus k}(x_1,\ldots,x_{m-1},x,y_{m+1},\ldots,y_k)] \geq \frac{1}{2} + \epsilon'$$

Using an averaging argument once more, this means there is a choice of $x_1, \ldots, x_{m-1}, y_{m+1}, \ldots, y_k, \pi$ such that

$$\Pr_x[C(\pi(x_1, \ldots, x_{m-1}, x, y_{m+1}, \ldots, y_k)) = f^{\oplus k}(x_1, \ldots, x_{m-1}, x, y_{m+1}, \ldots, y_k)] \geq \frac{1}{2} + \epsilon' \tag{3}$$

Notice that $f^{\oplus k}(x_1, \ldots, x_{m-1}, x, y_{m+1}, \ldots, y_k)] = f(x) + [\bigoplus f(x_i) + \bigoplus f(y_j)]$, and that $(\bigoplus f(x_i) + \bigoplus f(y_j))$ is fixed for all x. Thus, any circuit satisfying (3) that correctly computes this bit (we can think of this bit, along with the x_i's, y_j's, π as being hardcoded into the circuit) can be used to correctly compute $f(x)$ at least $\frac{1}{2} + \epsilon'$ of the time, and such a circuit of size at most g' exists.[8]

Thus, we have a circuit C' of size at most g' such that $\Pr_x[C'(x) = f(x)] \geq \frac{1}{2} + \epsilon'$, and this contradiction is what we wanted to show. ■

One important aspect of our definition of hardness is that functions are hard-core against *circuit sizes*, rather than the running times of a program; this allowed us to perform the key step in the proof of "hardcoding" our desired bit. In the next section, we discuss further this issue of *non-uniformity.*

3 Uniformity

3.1 Models of Computation

What kinds of models of computation do we have?

- Turing machines
- Circuits
- Decision Trees

Turing Machines are a *uniform* model of computation: a program has the same description for all input sizes. Circuits and decision trees, on the other hand, are a *non-uniform* model of computation: an algorithm is allowed to have a different description for each input size.

One question we can ask about non-uniform models of computation is: for each input size i, how do we construct our algorithm $\mathcal{A}_i$?

3.2 A Non-Uniform Complexity Class

To talk about the power and limits of non-uniformity, it will be useful to define a complexity class.

Definition 10 *Let C be a class of languages, and take any function $a : \mathbb{N} \to \mathbb{N}$. (We can think of a as a length function.) The class C* with advice a, *C/a, is defined as the set of languages L such that $L \in C/a$ if and only if there is some $L' \in C$ such that for all n, there exists $\alpha_n \in \{0,1\}^*$ with $|\alpha_n| \leq a(n)$, and $x \in L$ if and only if $(x, \alpha_{|x|}) \in L'$.*

Definition 11 $\mathrm{P/poly} = \bigcup \mathrm{P}/n^c$ *is the class of languages recognizable in polynomial time with polynomial-sized advice.*

How are these complexity classes connected to non-uniformity? It can be shown that P/poly is also the set of languages computable by a polynomial-sized (non-uniform) circuit. (The circuits correspond exactly to the advice.)

[8]It is important to note that we are *not* constructing this circuit. We have bazillions of circuits, one for each choice of the x_i's, y_j's, π, and we are simply saying that there is at least *one* of them that is correct on computing $f(x)$ at least $\frac{1}{2} + \epsilon'$ of the time.

Now that we have defined a non-uniform complexity class, we can use it to ask: how powerful is non-uniformity? Since, for any input of length n, we are allowed to use the fastest possible algorithm for inputs of that length, it seems that non-uniformity is extremely powerful. Indeed, as we will see in the next section, non-uniformity is as least as powerful as randomness.

Moreover, even just one bit of advice gives an extraordinary amount of power – the nth bit of advice can encode whether the nth Turing Machine halts, so there are uncomputable languages in $P/1$!

4 Derandomization

As above, in order to investigate the power and limits of randomness, we first define some complexity classes.

Definition 12 RP *(randomized polynomial time) is the class of languages L for which there exists a polynomial-time algorithm $\mathcal{A}$ such that if $x \in L$, then* $\Pr[\mathcal{A} \text{ accepts } x] \geq \frac{1}{2}$*, and if $x \notin L$, then* $\Pr[\mathcal{A} \text{ accepts } x] = 0$.

In other words, we can think of RP as the class of languages with *one-sided error*: if the algorithm $\mathcal{A}$ accepts an input x, then x is in L with certainty; if $\mathcal{A}$ rejects x, then there is a chance that the algorithm made a mistake.

If we allow *two-sided error*, the possibility that the algorithm makes a mistake when both accepting and rejecting, we have the language BPP.

Definition 13 BPP *(bounded-error probabilistic polynomial time) is the class of languages L for which there exists a polynomial-time algorithm $\mathcal{A}$ such that* $\Pr[\mathcal{A} \text{ accepts } x] \geq \frac{2}{3}$ *if $x \in L$, and* $\Pr[\mathcal{A} \text{ accepts } x] \leq \frac{1}{3}$ *if $x \notin L$.*

Note that there is nothing special about our choice of $\frac{2}{3}$ and $\frac{1}{3}$; any choice of *constants* bounded away from $\frac{1}{2}$ works.

Since non-uniformity is extremely powerful already, one question is whether random coins make it even better. It turns out that they do not: RP $\subseteq$ P/poly and BPP $\subseteq$ P/poly.[9] In other words, there is no point in derandomizing circuits; maybe we can get a smaller circuit if we allow randomization, but if we cannot compute something in polynomial-time without randomization, then we cannot compute it in polynomial-time with randomization either.

What about uniform models of computation (“normal” Turing machines)? Does P = RP? While this is still an open question, many researchers believe it to be true. We will consider questions such as this for the rest of the semester.

[9]It is easy to see that RP $\subseteq$ BPP, so in fact RP $\subseteq$ BPP $\subseteq$ P/poly.