Decision trees, boosted decision trees, branching programs, neural nets intro.

Administrative:

- Enrollment should be good now.
- Next homework will go out next Wednesday or Friday.
- Recap of where we are in course, and interplay between representation, optimization, and generalization; usual story is that as representation power improves, optimization and generalization become more painful.
- Many basic latex errors in assignments; please consult lshort for an easy reference/tutorial.
- Please keep giving feedback.

Here's the lemma from last time which we'll need today:

Lemma 1. Let continuous $g : \mathbb{R}^d \to \mathbb{R}$ and any $\epsilon > 0$ be given.

a. There exists a partition of $[0,1]^d$ into rectangles $(R_1, \ldots, R_N)$ and a function $h(x) := \sum_{i=1}^N g_i \mathbf{1}[x \in R_i]$ where $g_i := g(x)$ for some $x \in R_i$ such that $\|g - h\|_1 \leq \epsilon$. (Without loss of generality, this partition can be the uniform gridding of $[0,1]^d$ into cubes of equal volume.)

b. Let a basis $\mathcal{B}$ given so that for any $\tau > 0$ and rectangle R, there exists $f_R \in \text{span}(\mathcal{B})$ with $\|f_R - \mathbf{1}_R\|_1 \leq \tau$. Then there exists $f \in \text{span}(\mathcal{B})$ with $\|g - f\|_1 \leq \epsilon$.

Decision trees and branching programs

The class $\text{DT}(\mathcal{H})$ of **decision trees** with splitting functions $\mathcal{H}$ is the set of binary trees which evaluate an input x as follows.

- Internal nodes have an associated $h \in \mathcal{H}$; if $h(x) = 0$, evaluation continues with the left subtree, otherwise it continues with the right.
- Leaf nodes have an associated constant which they output when reached.

(Tree drawn in class.)

When $\mathcal{H}$ is not specified, let DT denote **standard decision trees**, which have the choice

$$\mathcal{H} := \left\{ x \mapsto \mathbf{1}[x_i \geq r] : i \in [d], r \in \mathbb{R} \right\}.$$

Often these are called "decision trees with axis-aligned splits".

Theorem. Given any continuous $g : \mathbb{R}^d \to \mathbb{R}$ and any $\epsilon > 0$, there exists $f \in \text{DT}$ with $\|f - g\|_1 \leq \epsilon$.

Proof. Let $h(x) := \sum_{i=1}^N g_i \mathbf{1}[x \in R_i]$ be the gridding function as in the lemma above, moreover with $(R_1, \ldots, R_N)$ being a uniform grid, and $N := k^d$ for some integer k. Note that $[0,1]$ can be partitioned into k intervals of length $1/k$ with a decision tree of size $\lceil \lg(k) \rceil$: *(picture drawn in class)* the root has predicate $\mathbf{1}[x \geq \lceil k/2 \rceil / k]$, the two children have predicates $\mathbf{1}[x \geq \lceil k/4 \rceil / k]$ and $\mathbf{1}[x \geq \lceil 3k/4 \rceil / k]$, and so on. Doing this

first for dimension 1, and then recursing on each leaf but for the next dimension yields a tree with depth $d\lceil \lg(k) \rceil$ and $\mathcal{O}(k^{d+1})$ leaves *(picture drawn in class!)*, finally outputting the appropriate g_i in the final leaves, gives an exact duplicate of h (up to measure zero sets). The result follows since $\|h - g\|_1 \leq \epsilon$. $\square$

Remarks.

- It also possible to construct a less balanced tree which for dimension i uses a sequence of predicates $\mathbf{1}[x_i \leq 1/k], \mathbf{1}[x_i \leq 2/k], \ldots, \mathbf{1}[x_i \leq (k-1)/k] \ldots$
- $\ldots$ either way, the size of the approximating tree is exponential in dimension!
- Decision trees will probably not be mentioned outside this part of the course! It's a good time to mention an **open problem** for them: learning decision trees is a disaster. The best result I know is due to Kearns and Mansour (1999), which basically says boosting is better, the issue being that data is partitioned amongst leaves, so with k leaves, most are trained with less than $1/k$ fraction of the data.

Along these lines, notice that decision trees can also easily represent indicators on single rectangles, which by the earlier lemma means a linear combination of trees can fit any function.

Theorem. Let any continuous $g : \mathbb{R}^d \to \mathbb{R}$ and any $\epsilon > 0$ be given. Let $\mathcal{B}$ denote standard decision trees with at most $2d$ nodes. Then there exists $f \in \text{span}(\mathcal{B})$ such that $\|f - g\|_1 \leq \epsilon$.

Note: the popular methods "boosted decision trees" and "random forests" build linear combinations of size-bounded decision trees, though each has additional quirks arising from the learning algorithm.

Lemma. Given any rectangle $R := [a_1, b_1] \times \cdots \times [a_d, b_d]$, the function $x \mapsto \mathbf{1}[x \in R]$ can be represented as a decision tree with $2d$ nodes.

Proof. It suffices to construct a decision tree with a single branch with predicates of the form $\mathbf{1}[x_i \geq a_i]$ or $\mathbf{1}[x_i \leq b_i]$, where the leaf corresponding to the logical and of all predicates outputs 1, and all other leaves output 0. $\square$

Proof (of theorem)**.** The preceding lemma indicates $\mathcal{B}$ contains indicators on all rectangles, which by Lemma 1 implies the approximation bound. $\square$

Remarks.

1. The size of the tree is relevant in two ways. First, shallow trees take less time to construct. Second, they have better generalization properties (this will make more sense in the third part of the course).
2. It is a disaster to learn even these trees! In fact, note that these trees are a single branch, and thus called "decision lists", which are hard to learn agnosticially Feldman et al. (2009).

   Open problem: find some way to circumvent the hardness barrier for instance via "improper learning".

There is a *very* interesting modification of decision trees that has small sizes for certain functions. Specifically, a **Branching Program** is like a decision tree, except two internal nodes may share a child!

Example (Kalai 2004, Figure 1)**.** (In class, the following picture is drawn. There are i nodes at depth i, each with predicate $\mathbf{1}[x_i \geq 1/2]$; all adajacent nodes share a child. There are $d+1$ leaves; leaf i outputs i/d. Together, the branching program computes $f(x) = \sum_{i=1}^n x_i/d$ over the boolean hypercube $\{0,1\}^d$.

Theorem (Kalai 2004, Figure 1)**.** The branching program in the preceding example has $\mathcal{O}(d^2)$ nodes, whereas any standard decision tree with no error on this problem has $\Omega(2^d)$ nodes.

Proof. Consider any standard decision tree f. Due to the axis-aligned split and boolean inputs, each edge in the tree constrains the inputs which can pass along that edge to have some fixed value in at most 1 bit. Consequently, for any fixed leaf which is some distance k from the root, it is reached by exactly those strings which have specific value for $\leq k$ of the bits, and any values on the remaining bits. But if any bits are unconstrained, then there exists two bit strings which differ on only one bit, both of which can reach this leaf; these two strings do not have the same value for $\sum_i x_i/d$, meaning the decision tree is wrong on at least one. Consequently, any correct tree will have exactly one bit string reach each leaf (no unconstrained bits per leaf), meaning there are 2^d leaves and 2^{d+1} total nodes. $\square$

Remark. Branching programs are somewhat forgotten. The paper by Kalai is beautiful and highly recommended, make sure to get the long version. Possible project idea!

Neural networks

Let's take a step back before defining the usual types of neural networks. Specifically, let's look at all of our function classes in a more unified way, as forms of programming languages.

The class $\text{span}(\mathcal{B})$ of linear representations over basis $\mathcal{B}$ is equivalently the set of functions $\mathcal{S}_1$ whose elements may be generated by the following rules:

$$\begin{aligned} f \in \mathcal{B} &\implies f \in \mathcal{S}_1, \\ \alpha, \beta \in \mathbb{R}, f, g \in \mathcal{S}_1 &\implies \alpha f + \beta g \in \mathcal{S}_1. \end{aligned}$$

Decision trees with splitting class $\mathcal{H}$ are built from the following rules:

$$\begin{aligned} \alpha \in \mathbb{R} &\implies (x \mapsto \mathbb{R}) \in \mathcal{S}_2, \\ h \in \mathcal{H}, l \in \mathcal{S}_2, r \in \mathcal{S}_2 &\implies (x \mapsto \mathbf{1}_h(x) l(x) + (1 - 1_h(x)) r(x)) \in \mathcal{S}_2. \end{aligned}$$

In both cases, the rule for combining elements of $\mathcal{S}_i$ into larger ones is fairly restrictive. It is more natural to consider a general sort of composition, namely applying elements of the base class directly:

$$\begin{aligned} f \in \mathcal{B} &\implies f \in \mathcal{S}_3, \\ r \in \mathbb{Z}_+, (f_1, \ldots, f_r, g) \subseteq \mathcal{S}_3^{r+1}, g : \mathbb{R}^r \to \mathbb{R} &\implies g(f_1, \ldots, f_r) \in \mathcal{S}_3. \end{aligned}$$

Classically, neural networks are identified by graphs, where nodes correspond to the functions $\mathcal{B}$ above; they take in real numbers, output real numbers. The edges of the graph connect outputs from nodes to inputs of other nodes. These nodes are typically associated in layers, where layer i contains those nodes whose longest path to the input is i. It is convenient to create a layer 0 denoting inputs themselves.

(Picture drawn in class.)

Classically, the nodes $\mathcal{B}$ have form $x \mapsto \sigma(a^\top x + b)$, where σ is nonlinear; the most common choice used to be the *sigmoid* $z \mapsto (1 + \exp(-z))^{-1}$, but now everyone uses the *ReLU* (rectified linear unit) $z \mapsto \max\{z, 0\}$.

Typically, the graph layout of a network is fixed, along with the choice of function at each node; what is allowed to vary are the real numbers parameterizing each node, for instance (a, b) in the preceding. Note that weights may be shared in different nodes.

Here are a few other choices that are popular lately.

- Maximization nodes: an r-dimensional input x is mapped to its maximum: $x \mapsto \max_i x_i$.
- "Convolution" operations: a fixed set of weights are dot-producted with k different subsets of nodes in previous layers. Often, these subsets are organized spatially in some way, and the name "convolution" comes from the thought of sliding a single "filter" along a previous layer. Usually a few "filters" are used, and some maximization nodes are also interspersed to "downsample".

The next lecture will establish that 2 layers are necessary and sufficient for neural networks to approximate continuous functions arbitrarily finely. A representation result for 3 layers will also be included since its proof is more intuitive.

References

Feldman, Vitali, Venkatesan Guruswami, Prasad Raghavendra, and Yi Wu. 2009. "Agnostic Learning of Monomials by Halfspaces Is Hard."

Kalai, A.T. 2004. "Learning Monotonic Linear Functions." In *COLT*. http://research.microsoft.com/en-us/

um/people/adum/publications/2004-Learning_Monotonic_Linear_Functions-long_version.pdf.

Kearns, Michael, and Yishay Mansour. 1999. "On the Boosting Ability of Top-down Decision Tree Learning Algorithms." *Journal of Computer and Systems Sciences* 1 (58): 109–28.