1 Introduction

This lecture is an introduction to wavelets focusing on Haar wavelets in 1D. In section 2, we first give the definition and observe some of its basic properties. In section 3, we show the Fast Haar Transform. In section 4, we discuss how Haar basis is useful for working with histograms.

2 Haar wavelets

Haar wavelets form an orthogonal basis for $\mathbb{R}^n$. In general, different wavelets are obtained by taking a “mother wavelet” and applying shifts and dilations.

The “mother wavelet” for Haar wavelets is shown by the following basis for $n = 2$:

$$H_2 = \begin{pmatrix} +1 & -1 \\ +1 & +1 \end{pmatrix}$$

For $n = 2^k$, we shift the mother wavelet across the entire domain, then repeat after dilating the wavelet.

For example, the Haar wavelets for $n = 8$ are as follows:

$$H_8 = \begin{pmatrix} +1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & +1 & -1 \\ +1 & +1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & +1 & -1 & -1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \\ +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \end{pmatrix}$$

Each scale of dilation corresponds to a level. The first four rows correspond to the first level, the next two rows to the second level, and so on. Naturally, we can visualize the Haar basis as a binary tree on these vectors, where a node $z$ is a parent of nodes $x$ and $y$ if $\text{supp}(z) = \text{supp}(x) \cup \text{supp}(y)$; we will refer to this representation as the wavelet tree.

2.1 Orthogonality

To see why the rows of the above matrix are orthogonal, it is enough to observe to following:

- if the vectors at the same level, they have disjoint supports
- if the vectors are at different levels, the vector at the higher level is constant on the support of the vector at the lower level
2.2 Extension to 2D

It might be desirable to extend the above wavelets to higher dimensions, for example to represent 2D signals such as images.

One natural way to extend the 1D Haar wavelets to 2D is to take the tensor product. You can also view this as taking the tensor product of two binary trees. Another option is to construct a quad tree by recursing into four subgrids.

3 Fast Haar Transform

How efficiently can we compute the Haar transform of \( x \), i.e. \( y = H_n x \)? The “dumb” way using standard matrix vector multiplication takes \( O(n^2) \). By observing that the matrix \( H_n \) has only \( O(n \log n) \) non-zero entries (since there are exactly \( n \) non-zero entries across all the vectors in each level), we can reduce this to \( O(n \log n) \). We can do even better though. The following method runs in linear runtime:

Algorithm 1 Recursive method for computing Haar transform in \( O(n) \)

1: for \( i = 0, \ldots, \frac{n}{2} - 1 \) do
2: \( y_i = x_{2i} - x_{2i+1} \)
3: for \( i = \frac{n}{2}, \ldots, n - 1 \) do
4: \( y_i = H_{\frac{n}{2}} x'_{i-n} \) \( \triangleright \) recurse
5: where \( x'_j = x_{2j} + x_{2j+1} \)

The algorithm takes advantage of the fact that if we collapse the bottom half of matrix \( H_n \) horizontally in half, we just get \( H_{n/2} \). We can alternatively view the above algorithm as computing the sum of each subtree bottom up (where at each node, we subtract the value of right subtree from that left subtree to compute a new entry of \( y \), and we propagate their sum).

As Ludwig pointed out in class, we can avoid recursion: since each entry in \( y \) is the difference of two partial sums of \( x \), by pre-computing all the prefix sums of \( x \) in linear time, we can compute each entry of \( y \) in constant time.

4 Histograms

What kinds of signals can be sparsely represented in the Haar basis? Notice that each Haar wavelet has small support and is very local, so it is reasonable to expect that signals that are local in nature are well suited for representation in the Haar basis. One class of such signals is piece-wise constant functions, or histograms.\(^1\) We give a formal definition below.

**Definition 1.** A \( k \)-histogram is a function \( h : \{0, \ldots, n-1\} \rightarrow \mathbb{R} \) defined by \( h(i) = v_j \) if \( I_j \ni i \), where \( I_1, \ldots, I_k \) are disjoint intervals covering \( \{0, \ldots, n-1\} \) and each associated with a value \( v_i \). Alternatively, \( h = \sum_{j=1}^{k} v_j t_j \), where each \( t_j \) is a threshold function, i.e. \( t_j(i) = \begin{cases} 0 & \text{if } i < T_j \\ 1 & \text{if otherwise} \end{cases} \) for some \( T_j \).

**Claim 2.** Any threshold function \( t \) is \( 1 + \log n \) sparse in Haar basis.

\(^1\)For practical motivation, histogram can compactly represent distributions of values, which are useful for query optimization for databases.
Proof. Let $T$ be the threshold of $t$. At each level of the wavelet tree, there is at most one basis element $b$ such that $t \cdot b$ is non-zero (select $b$ that has $T$ in its support). Since there are $1 + \log n$ levels, our claim follows.

Corollary 3. Any $k$-histogram is $O(k \log n)$-sparse in the Haar basis.

We have seen that histograms are sparse in the Haar basis.

4.1 Approximation by histograms

Given $x$, how efficiently can we find a histogram approximating $x$? We can approximate an $x$ by a $k$-histogram by finding $h^*$ s.t.

$$||h^* - x||_2 = \min_{k\text{-histogram } h} ||h - x||_2$$

The fastest known algorithm for finding $h^*$ uses dynamic programming and takes $O(n^2 k)$.

However, we can do much better if we accept a small blow up in sparsity. In just $O(n)$ time we can find $g$ s.t. $g$ is $k' = O(k \log n)$-sparse in Haar basis and $||g - x||_2 \leq ||h^* - x||_2$ using the following algorithm. Recall from Lecture 1 that the thresholding operator $H_{k'}(x)$ returns the vector containing only the $k$ largest (in absolute value) coordinates of $x$, and sets all other coordinates to 0.

Algorithm 2 $O(k \log n)$-sparse approximation algorithm

1: Let $H'$ be the orthonormalized $H_n$
2: Compute $g = H'^T(H_{k'}(H'x))$ $\triangleright$ note $H'^{-1} = H'^T$

Analysis

Proof of approximation guarantee:

$$||g - x||_2 = ||H'^T(H_{k'}(H'x)) - x||_2$$

(1)

$$= ||H_{k'}(H'x) - H'x||_2$$

(2)

$$\leq ||H'h^* - H'x||_2$$

(3)

$$= ||h^* - x||_2$$

(4)

(2), (4) since unitary transformations preserve the 2-norm.

(3) since $h^*$ is $k'$-sparse in the Haar basis, and among all $k'$-sparse vectors in Haar basis, $H_{k'}(H'x)$ best approximates $x$ by definition.

Runtime is $O(n)$ since computing Haar transform or its inverse\(^2\) takes $O(n)$ and we can compute $H_{k'}$ in $O(n)$ by using a selection algorithm.

\(^2\)Just as the Fast Haar Transform could be visualized as working bottom-up on the wavelet tree given values at the leaves in order to compute partial sums at each internal node, the inverse Haar Transform can be seen as working top-down, using partial sums at internal nodes to recover the values at the leaves.
4.2 Computing $k$-sparse approximation with less space

If one wants to directly compute the $k$-sparse Haar transform, we can save on space using the following method from [1]. In a streaming model where we get the next coordinate of $x$ at each time step, we only need $O(k)$ space to keep track of the top $k$ entries so far (ones that have been computed using only the coordinates $x$ seen so far). Now, visualize our computation $H'x$ as before as traversing the wavelet tree. Note that at any given time, only $\log n$ of the nodes are “active” (corresponding to the path from the root to the current $x$ coordinate in the wavelet tree), meaning only their partial sums will be updated in the next time step. So we need only $O(\log n)$ space to keep track of active nodes rather than the entire tree, and total of $O(k + \log n)$ space is needed at any given time.

To update after seeing a new $x$ coordinate, we update all the partial sums along the active path, delete the nodes that are “dead” now and update the top-$k$ elements if necessary, and go down a new active path to reach the next leaf. Since the length of a path is $\log n$, and selection of top $k$ items takes $O(1)$ amortized per item, each update takes $O(\log n)$. Note that the overall time has increased to $O(n \log n)$ at the cost of smaller space usage.

References