Proof: We first show that the following holds:

$$\exists C > 0, \forall V, \exists \epsilon_0, \forall \epsilon < \epsilon_0, \text{pack}_{\mathcal{F}_V}(\epsilon) \geq M/\epsilon^{(C \times V)}.$$  \hspace{1cm} (14)

Eq. 14 is a version of Eq. 1 modified for considering only $\epsilon$ small; it is weaker than Eq. 1 and sufficient for our purpose.

Eq. 14 is proved as follows:

- For $x$ and $y$ in $[\frac{1}{2}, 1]^V$, the $L^1$ distance between $[0, x]$ and $[0, y]$ is lower bounded by $\Theta(||x - y||_1)$.
- Therefore, a regular grid of edge $\Theta(\epsilon)$ can be constructed in $[0, 1]^V$, yielding $\Theta(1/\epsilon^V)$ points for the sole $[1/2, 1]^V$ part. Consequently, the packing number of $\{(0, x); x \in [0, 1]^V\}$ is $\Theta(1/\epsilon^V)$ and is therefore $\omega(1/\epsilon^{V/2})$.
- This shows Eq. 14 for $C = \frac{1}{2}$.

Then, Eq. 14 classically leads to Eq. 12 (this is analogous to the proof of Eq. 2 from Eq. 1, see section 3). This is the first part of the theorem (Eq. 12). Let us now show Eq. 13, by considering the following algorithm described at iteration $n$, with $\lambda = V$:

- **generate**$_\lambda$ prepares the batch $((x_{n\lambda+1}, \ldots, x_{(n+1)\lambda})$ as follows: for each $x_{n\lambda+i}$, all coordinates $j \neq i$ are set to 0. The $i^{th}$ coordinate of $x_{n\lambda+i}$ is chosen by looking at the $n - 1$ previous points in position $i$ of each of the $n - 1$ previous batches. It is defined as the middle of the segment defined by the lowest previously-observed $i^{th}$ coordinate whose label is 0, and the highest previously observed $i^{th}$ coordinate whose label is 1. More formally: \footnote{In Eq. 15, if no point $x_{n'\lambda+i}$ has been labeled as 0, the minimum is set to 1; equivalently, if no point has been labeled as 1, the maximum is set to 0.}

  $$(x_{n\lambda+i})_i = \frac{1}{2} \left( \min_{n' \leq n} \{(x_{n'\lambda+i})_i | y_{n'\lambda+i} = 0\} + \max_{n' \leq n} \{(x_{n'\lambda+i})_i | y_{n'\lambda+i} = 1\} \right). \hspace{1cm} (15)$$

- **learn**$_\lambda$ selects any function $f_n \in \mathcal{F}_V$ which is consistent with $x_1, \ldots, x_{n\lambda}$.

At a given iteration $n$ each point $x_{n,i}$ of the batch of size $\lambda$ makes sure that the domain will be halved along the $i^{th}$ coordinate. Thus, after $N$ iterations, it is known that the target oracle/classifier is in a square of edge size $2^{-N}$. As a consequence, precision $\epsilon$ is reached in at most $\Theta(\log(1/\epsilon))$ iterations, which shows Eq. 13.

This theorem shows that, at least for $\mathcal{F}$ as above, we can have a linear speed-up until $\lambda = V$; this is the tightness of Eq. 3 for $\lambda \leq V$—similarly to the tightness of Eq. 6 (\textit{i.e.} logarithmic speed-up) shown by Eq. 10 for $\lambda$ large.
4 Experiments

We have formally proved both lower and upper bounds on batch AL. The following shows that both a simple algorithm and a more sophisticated (yet usual) AL algorithm behave as predicted by the theorems of previous sections when adapted to the batch setting.

4.1 Experiments with Naive AL

We here experiment a simple batch AL algorithm for $\mathcal{F} = \{[0, x]; x \in [0,1]^d\}$ (VC-dimension $V = d$). The new sample(s) $x_{n\lambda+1}, \ldots, x_{(n+1)\lambda}$ are $\lambda$ points randomly drawn in $v$ where

$$v = \{x \in [0,1]^d; \forall j \in [1, n\lambda]y_j = 0 \Rightarrow \neg (x_j \leq x)\} \cap \{x \in [0,1]^d; \forall j \in [1, n\lambda]y_j = 1 \Rightarrow x_j \leq x\}.$$

This means that we randomly sample the version space. Note that this parallel algorithm is straightforward to derive from its sequential counterpart that queries one random sample from the version space at each iteration. This is not true of all active learning algorithms: in many cases, it is not clear how to efficiently turn a sequential active learning into a parallel one.

The plot shows the inverse of the number of iterations for reaching precision $0.001d^2$, depending on $\lambda$; this means that what lies on the Y-axis are rates of convergence. Results are presented in Fig. 1. Each point is averaged over 33 runs.

![Graph](image-url)

**Fig. 1.** Speed-up of batch AL for a simple AL algorithm (see text)