**CSCI699: Theory of Machine Learning** Fall 2021

# Lecture 1: Introduction to Supervised Learning

*Instructor: Vatsal Sharan* *Scribe: Ta-Yang Wang*

## What are the goals for the course?

We will study the following questions (or a subset of them depending on interest and pace), with the main focus on the first two questions.

- *How much data is needed for learning?*
- *How much computation time needed for learning?*
- How much memory need for learning?
- Tradeoffs between these resources?
- How to be robust to worse-case perturbations/maniputations to the data?
- How to be robust to out-of-distribution data?
- How to learn models which are fair?
- How to understand some of these questions for deep neural networks?

## Goals for today's class:

- What is ML theory and why study it?
- Define supervised learning
- Define PAC learning

## What is ML? What is ML theory?

Let's look at how Leslie Valiant defines learning in his paper, A Theory of the Learnable, [Valiant'84.] (for which he was awarded the Turing award in 2010).

> *"Humans appear to be able to learn new concepts without needing to be programmed explicitly in any conventional sense. In this paper we regard learning as the phenomenon of knowledge acquisition in the absence of explicit programming."*

He then goes on to describe how a formal study of computation became possible once precise models were available, and how that theory helps explain nature, informs how computing devices should be built, and is also worth studying for its own sake.

*"Computability theory became possible once precise models became available for modeling the commonplace phenomenon of mechanical calculation. The theory that evolved has been used to explain human experience and to suggest how artificial computing devices should be built. It is also worth studying for its own sake."*

Learning seems like almost as fundamental of a task, and deserves attention for the same reason.

*"The commonplace phenomenon of learning surely merits similar attention. The problem is to discover good models that are interesting to study for their own sake and that promise to be relevant both to explaining human experience and to building devices that can learn. The models should also shed light on the limits of what can be learned, just as computability does on what can be computed."*

Let us summarize this:

- Machine Learning (ML): programs that can automate as improve based on data
- Goal of this class: understand the theory behind ML, what can be learned, how much resources does learning require.

## Why study ML theory?

All the reasons are already stated in the excerpts from Valiant's paper above, but we can talk a bit more.

- Theory can help inform practical algorithms, and can be especially helpful in designing algorithms/models for new settings. It can also help discover new frontiers, possibilities and paradigms.
  - A classic example of this is Boosting: Valiant-Kearns asked "can you boost a weak learning algorithm (which gets say 51% accuracy on binary classification) to a strong learner (which gets say 95% accuracy)?" This led to boosting, Adaboost, etc. which have been very influential in practice. Another example is many optimization algorithms for ML which are rooted/guided by theory, such as Adagrad, SGD.
  - Modern desiderata are often complex, and benefit from theoretical thinking to formalize, understand or certify requirements (such as for fairness, robustness, privacy)
- But even beyond practice, many of the questions seem to be interesting, fundamental questions on their own and help us better understand the nature of computation, and learning in nature.

With this done, we are now ready to start out with technical stuff.

## Supervised learning: A gentle introduction

We begin with formally descirbing the supervised learning setup.

- We have a training set, which is just a set of input/output pairs.
- The goal of the learner is to learn some pattern to accurately predict outputs of unseen inputs.
  - e.g. image classification (picture $\rightarrow$ dog/cat), machine translation (text in English $\rightarrow$ text in French), SAT solving (SAT instance $\rightarrow$ satisfying assignment if it exists).
- It helps to abstract this. Formally, we have,
  - An input space: $\mathcal{X} \subset \mathbb{R}^d$.
  - An output space: $\mathcal{Y} \in \{\pm 1\}$, for binary classification.
  - The goal is to come up with predictor $f(x) : \mathbb{R}^d \rightarrow \{\pm 1\}$ which predicts the output of $x$
- To reason about how well our predictor $f(x)$ is doing, we need to define a **loss function**, denoted by $\ell(f(x), y)$. The choice of the loss function depends on the domain and the goal of the learning task.
  - For a classification problem where the goal is to predict binary outputs, a reasonable choice could be the 0/1 loss: $\ell(f(x), y) = \mathbf{1}(f(x) \neq y)$.
  - For a regression problem where the goal is to predict continuous valued attributes, a common choice is the squared loss: $\ell(f(x), y) = (f(x) - y)^2$.
- We want to minimize this loss not just for one sample, or just the training set of samples, but for samples we might not have even seen yet. Formally, we care about minimizing this loss for samples drawn from some **distribution** $P$ over $(x, y)$ input/output pairs.
- This brings us to the definition of the **risk of a predictor** on a supervised learning problem:

$$R(f) = \mathbb{E}_{(x,y)\sim P}\ell(f(x), y).$$

- However, we don't actually know this distribution $P$, but we instead get some training samples. In order to ensure that the predictor learnt from the training data does well on the data distribution $P$, we need the training data to be representative of the distribution.
  - The usual way to do this is the **i.i.d. assumption:** we assume that the training set consists of $n$ samples drawn independently and identically distributed according to $P$.

With our setup formalized, we can now define an optimal predictor $f^*$, which is the predictor which minimizes the risk over all possible functions from the input to the output.

$$f^* = \operatorname*{arg\,min}_{f:\mathcal{X}\rightarrow\mathcal{Y}} \mathbb{E}_{(x,y)\sim P}\ell(f(x), y).$$

However, in practice one would never try to minimize the risk over the space of all possible functions (for the reason that it would be too statistically or computationally expensive, as we will see).

Instead, we usually commit to to a set of functions $\mathcal{F}$ beforehand. For example, this function class could comprise of all decision trees of a certain depth, all linear classifiers, or all neural networks of a certain size. And as we discussed, we do not know the distribution $P$, but are instead working with a set of $n$ samples drawn i.i.d. from $P$. This brings us to the notion of an **empirical risk minimizer (ERM)** $f_{S,\text{ERM}}$ for some function class $\mathcal{F}$, based on a training set $S = \{(x_i, y_i) : i \in [n]\}$,

$$f_{S,\text{ERM}} = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y).$$

## Error decomposition

We can decompose the risk of any function $f_S$ chosen based on the training set $S$ in the following way.

$$R(f_S) - R(f^*) = \underbrace{R(f_S) - R(f_{S,ERM})}_{\text{computation error}} + \underbrace{R(f_{S,ERM}) - \inf_{f \in \mathcal{F}} R(f)}_{\text{estimation error}} + \underbrace{\inf_{f \in \mathcal{F}} R(f) - R(f^*)}_{\text{approximation error}}$$

- Computation error: If we are just given the training set $S$, it seems the ERM is the right function to choose based on $S$, since it minimizes the risk over the training set. However, in some cases, finding the ERM might be computationally expensive or hard (we will see this later), or we may want to run a simpler/more efficient algorithm. We define the computation error as the difference between the risk of the ERM, and the risk of the predictor $f_S$.
- Estimation error: Note that the ERM was the minimizer of the training error, but our goal is to actually minimize the risk over the true (unknown) distribution $P$. The minimizer over samples from $P$ need not necessarily be good over the distribution $P$, for example, with some probability we could get a bad training set for which the ERM could be far from the true minimizer. We define the estimation error (also known as the generalization error) as the difference between the risk of the ERM, and the best possible risk achievable within the function class $\mathcal{F}$.
- Approximation error: We are choosing to work over the function class $\mathcal{F}$, but the optimal predictor $f^*$ might not lie in $\mathcal{F}$. The approximation error measures how much worse is it to work over the function class $\mathcal{F}$.

## An example: Linear regression

Let us see these concepts for the case of linear regression. We first define the setup.

- $\mathcal{X} = \{x : x \in \mathbb{R}^d, ||x||_2 \leq 1\}$ (unit ball)
- $\mathcal{Y} = \mathbb{R}$
- $\ell(f(x), y) = \frac{1}{2}(f(x) - y)^2$
- $\mathcal{F} = \{f_w(x) = \langle w, x \rangle, w \in \mathbb{R}^d\}$

Let us look at the error terms in this context.

- **Computation error**: Let us define the empirical risk $\hat{R}_S(w)$ as the risk over the training set $S$. In this case, it can be written as follows,

$$\hat{R}_S(w) = \frac{1}{2n}\sum_{i=1}^{n}(\langle w, x_i\rangle - y_i)^2$$

  - In this linear regression setting, we could bring the computation error to 0, since we could compute the ERM using a closed-form solution. To find the closed-form solution, we can differentiate the empirical risk above with respect to $w$ and set the gradient to be 0.

$$\begin{aligned} 0 = \frac{\partial \hat{R}_S(w)}{\partial w} &= \frac{1}{n}\sum_{i=1}^{n} x_i(\langle w, x_i\rangle - y_i) \\ \sum x_i y_i &= \sum_{i=1}^{n} x_i x_i^T w \\ w &= \left(\sum x_i x_i^T\right)^{-1} \sum x_i y_i \\ \implies w &= \left(XX^T\right)^{-1} Xy \end{aligned}$$

    where $X \in \mathbb{R}^{d\times n}$ is the matrix obtained by stacking all the data points $x_i$ together, and $y \in \mathbb{R}^n$ is the vector of outputs $y_i$ for all datapoints $x_i$.

    Let us analyze the **time complexity** of computing the ERM solution. An easy exercise from your algorithms class will tell you that this is $O(nd^2 + d^3)$.
    * $XX^T$: $O(nd^2)$
    * $(XX^T)^{-1}$: $O(d^3)$
    * $Xy$: $O(nd)$
    * $(XX^T)^{-1}Xy$: $O(d^2)$
  - What if we want to be more computationally efficient than this closed-form solution? One option is to use **gradient descent**. Gradient descent starts with an initialization, and then iteratively updates the estimate based on the gradient.

$$\begin{aligned} w_0 &\leftarrow 0 \\ w_{t+1} &\leftarrow w_t - n_t \nabla \hat{R}_s(w_t). \end{aligned}$$

    In this case, gradient descent updates are as follows.

$$w_{t+1} = w_t - n_t\left(\frac{1}{n}\sum_{i=1}^{n}\underbrace{x_i(\langle w_t, x_i\rangle - y_i)}_{O(d)\text{ time}}\right).$$

    What is the time complexity of this?

    * The per step complexity is $O(nd)$.
    * We can show that for this problem, in $T$ steps, gradient descent can find a $O(e^{-T})$ sub-optimal solution compared to the closed-form solution. Hence the computation error is $O(e^{-T})$ for $T$ steps of gradient descent.
  - Therefore, we're already seeing a tradefoff here. Gradient descent could be computationally more efficient, but we would pay some computation error.

- **Estimation error**: In the same setting, we can ask how much data the closed-form solution (the ERM here) needs to find a solution which has small error on the distribution?
- **Approximation error**: If the true model is not linear, we'll incur an additional penalty because we restrict ourselves to linear functions.

## PAC learning (Probably Approximately Correct Learning)

PAC learning was defined in Valiant's paper, and provides a clean and rigorous model to understand when learning is possible. For PAC learning, we are usually interested in the following setup:

- Label space: $\mathcal{X} \in \mathbb{R}^d, \mathcal{Y} = \{0, 1\}$
- Loss function: $\ell(f(x), y) = \mathbf{1}(f(x) \neq y)$

There are two crucial ideas in the PAC learning setup.

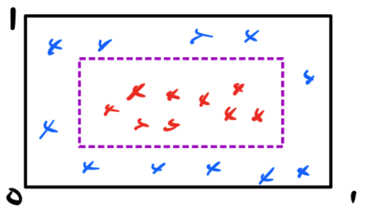
- There is no assumption on the distribution $D$ over the datapoints $\mathcal{X}$. However, note that the algorithm is also only required to do well in terms of the loss on datapoints drawn from the same distribution $D$.
- We assume that the labels $y = h(x)$ for some $h \in \mathcal{H}$. This assumption is known as the **realizability assumption**, and says that it is possible to exactly predict the labels of every datapoint using the hypothesis class $\mathcal{H}$ (we'll mostly call what we were calling the function class $\mathcal{F}$ earlier as the hypothesis class $\mathcal{H}$ now).

**Definition 1** (PAC learnability)**.** *A hypothesis class $\mathcal{H}$ is **PAC-learnable** with $m_H(\epsilon, \delta)$ samples if there exists a learning algorithm with the following property: for every $\epsilon, \delta \in (0, 1)$, every distribution $D$ over $\mathcal{X}$ and every $h \in \mathcal{H}$, when the algorithm is given $m_H(\epsilon, \delta)$ samples drawn from $D$ and labeled by $h$, the algorithm produces a hypothesis $\hat{h}$ such that with probability $1 - \delta$, $R(\hat{h}) \leq \epsilon$. (This probability is over randomness in training sets and any internal algorithm randomness)*

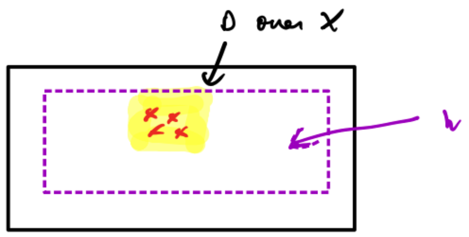
## Example: Axis-aligned rectangles (also refer to figure in whiteboard notes)

Let's consider a simple example (also see Fig. 1a).

- $\mathcal{X} = [0, 1] \times [0, 1]$
- $\mathcal{H} =$ rectangles



(a) Points inside the purple, dashed rectangle are labelled as 1, and points outside as 0.



(b) $D$ over $\mathcal{X}$ could be such that we cannot recover the true hypothesis, but prediction is easy.

Figure 1: Axis-aligned rectangles.

- $D$: uniform distribution
- Both the requirements of being only approximately correct and correct with high probability cannot be strengthened without making learning impossible:
  - Failure probability $\delta$: With some small probability we could get bad set of samples (say supported on only one corner of the cube), and cannot do anything.
  - Approximation parameter $\epsilon$: We can only do approximately correctly (cannot recover the true hypothesis $h^*$ exactly).
- Note that we can also hope to show learnability for any distribution $D$:
  - The reason is that maybe the distribution is such that we cannot learn the ground truth $h^*$, but we could still predict well on that distribution. For example, in a case where the distribution $D$ and the ground truth hypothesis $h^*$ is such that we only see examples with label 1, it may not be possible to learn any approximation to $h^*$ in terms of its parameters, but prediction is also trivial in this case (also see Fig. 1b).