Introduction to Computability and Complexity

A Glimpse into Computational Learning Theory

Ugo Dal Lago

University of Bologna, Academic Year 2024/2025

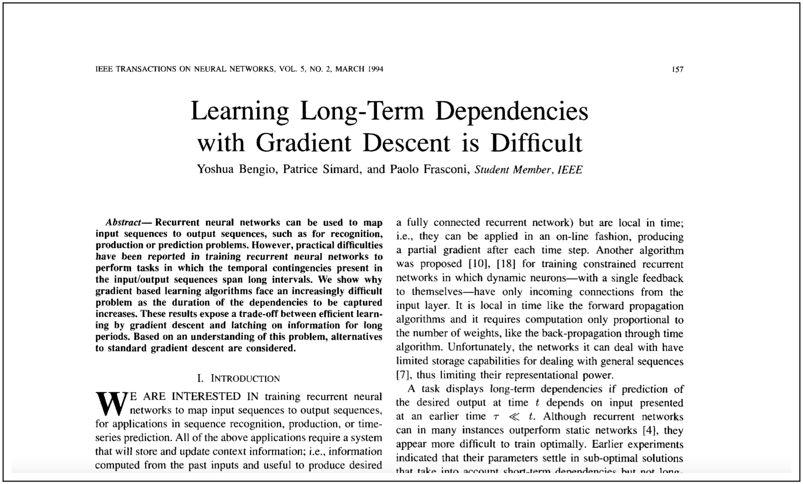
Negative Results in Learning?

- Up to now, we have been considering (positive and) negative results about **functions** as computational tasks?

Negative Results in Learning?

- Up to now, we have been considering (positive and) negative results about **functions** as computational tasks?
- Would it be possible to get negative results about the kind of tasks one typically consider in machine learning?

A Famous Example



IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 5, NO. 2, MARCH 1994 157

Learning Long-Term Dependencies with Gradient Descent is Difficult

Yoshua Bengio, Patrice Simard, and Paolo Frasconi, *Student Member, IEEE*

***Abstract*— Recurrent neural networks can be used to map input sequences to output sequences, such as for recognition, production or prediction problems. However, practical difficulties have been reported in training recurrent neural networks to perform tasks in which the temporal contingencies present in the input/output sequences span long intervals. We show why gradient based learning algorithms face an increasingly difficult problem as the duration of the dependencies to be captured increases. These results expose a trade-off between efficient learning by gradient descent and latching on information for long periods. Based on an understanding of this problem, alternatives to standard gradient descent are considered.**

I. Introduction

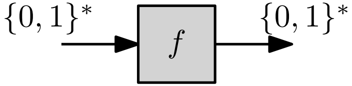
WE ARE INTERESTED IN training recurrent neural networks to map input sequences to output sequences, for applications in sequence recognition, production, or time-series prediction. All of the above applications require a system that will store and update context information; i.e., information computed from the past inputs and useful to produce desired

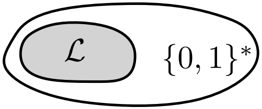
a fully connected recurrent network) but are local in time; i.e., they can be applied in an on-line fashion, producing a partial gradient after each time step. Another algorithm was proposed [10], [18] for training constrained recurrent networks in which dynamic neurons—with a single feedback to themselves—have only incoming connections from the input layer. It is local in time like the forward propagation algorithms and it requires computation only proportional to the number of weights, like the back-propagation through time algorithm. Unfortunately, the networks it can deal with have limited storage capabilities for dealing with general sequences [7], thus limiting their representational power.

A task displays long-term dependencies if prediction of the desired output at time t depends on input presented at an earlier time $\tau \ll t$. Although recurrent networks can in many instances outperform static networks [4], they appear more difficult to train optimally. Earlier experiments indicated that their parameters settle in sub-optimal solutions that take into account short-term dependencies but not long-

Learning Problems as Computational Problems

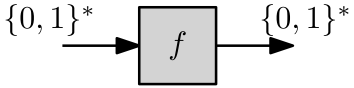
- We have so far taken functions and languages as our notions of **computational tasks**:

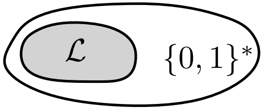




Learning Problems as Computational Problems

- We have so far taken functions and languages as our notions of **computational tasks**:

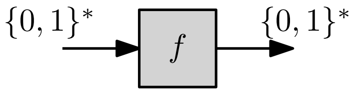


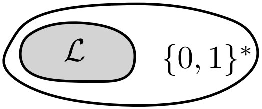


- Is it that **learning problems**, among the most crucial tasks in AI, can be seen as computational problems?

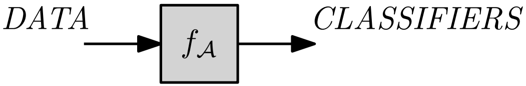
Learning Problems as Computational Problems

- We have so far taken functions and languages as our notions of **computational tasks**:



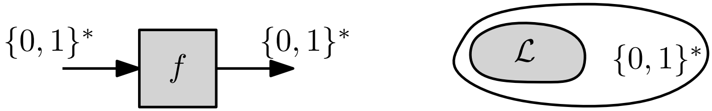


- Is it that **learning problems**, among the most crucial tasks in AI, can be seen as computational problems?
- The answer is positive: any learning algorithm $\mathcal{A}$ actually computes a function $f_{\mathcal{A}}$ whose input is a finite sequence of *labelled data* and whose output can be seen as a *classifier*:

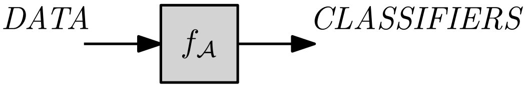


Learning Problems as Computational Problems

- We have so far taken functions and languages as our notions of **computational tasks**:

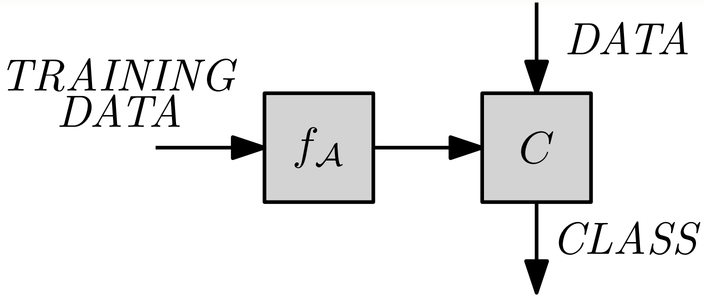


- Is it that **learning problems**, among the most crucial tasks in AI, can be seen as computational problems?
- The answer is positive: any learning algorithm $\mathcal{A}$ actually computes a function $f_{\mathcal{A}}$ whose input is a finite sequence of *labelled data* and whose output can be seen as a *classifier*:

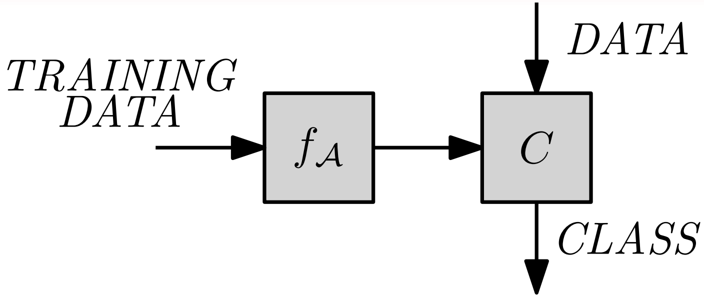


  - Could data and classifiers be encoded as strings, thus turning $f_{\mathcal{A}}$ as a function of the kind we know?
  - How could we formalize the fact that $\mathcal{A}$ correctly solves a given learning task?

Learning Problems as Computational Problems



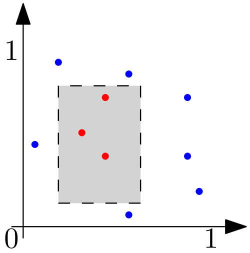
Learning Problems as Computational Problems



- Training data are **labelled**, so as to let $f_{\mathcal{A}}$ learn what the relationship exists between data and labels.
- The data C takes as input are **not labelled**, and the C's task is precisely the one of finding the appropriate label for any of them.
- Most often, C is drawn from a rather **restricted set of classifiers**, i.e. not all algorithms can be obtained in output from $f_{\mathcal{A}}$.

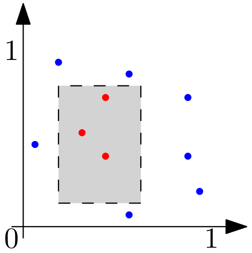
An Example Problem

Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle



An Example Problem

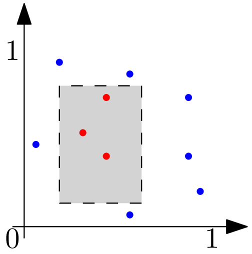
Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle



- The algorithm $\mathcal{A}$ should be able to guess a classifier, namely a *rectangle*, based on the labelled data it received in input.

An Example Problem

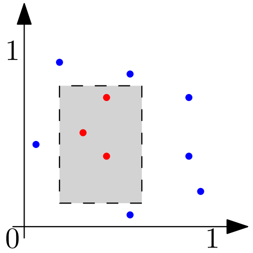
Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle



- The algorithm $\mathcal{A}$ should be able to guess a classifier, namely a *rectangle*, based on the labelled data it received in input.
- It knows that the data are labelled according to a rectangle, R but it does not know *which rectangle* is being used.

An Example Problem

Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle



- The algorithm $\mathcal{A}$ should be able to guess a classifier, namely a *rectangle*, based on the labelled data it received in input.
- It knows that the data are labelled according to a rectangle, R but it does not know *which rectangle* is being used.
- The algorithm $\mathcal{A}$ cannot guess the rectangle R with perfect accuracy if the data it receives in input are too few. As the data in D grow in number, we would expect the rectangle $f_{\mathcal{A}}(D)$ to converge to R, wouldn't we?

The Rules of the Game

- Again, $\mathcal{A}$ *knows* that the the way input data are labelled is by way of a rectangle (whose sides are parallel to the axes).
  - But it *does not know* which one!

The Rules of the Game

- Again, $\mathcal{A}$ *knows* that the the way input data are labelled is by way of a rectangle (whose sides are parallel to the axes).
  - But it *does not know* which one!
- $\mathcal{A}$ *does not know* the distribution $\mathbf{D}$ from which the points (x, y) are drawn.
  - It is supposed to “do the job” for each possible distribution $\mathbf{D}$.

The Rules of the Game

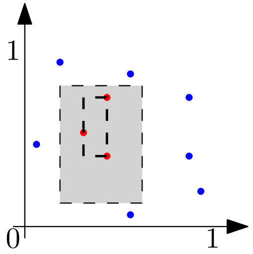
- Again, $\mathcal{A}$ *knows* that the the way input data are labelled is by way of a rectangle (whose sides are parallel to the axes).
    - But it *does not know* which one!
- $\mathcal{A}$ *does not know* the distribution **D** from which the points (x, y) are drawn.
    - It is supposed to "do the job" for each possible distribution **D**.
- $\mathcal{A}$ is an ordinary algorithm.
    - Ultimately, it can be seen as a TM.
    - We thus assume that real numbers can be appropriately approximated as binary strings.
    - In some cases, it is useful to assume $\mathcal{A}$ to have the possibility to "flip a coin", i.e., to be a randomized algorithm.

The Algorithm $\mathcal{A}_{\mathsf{BFP}}$

- We could define an Algorithm $\mathcal{A}_{\mathsf{BFP}}$ as follows:
  1. Given the data $((x_1, y_1), p_1), \ldots, ((x_n, y_n), p_n)$;
  2. Determine the smallest rectangle R including all the positive instances;
  3. Return R.

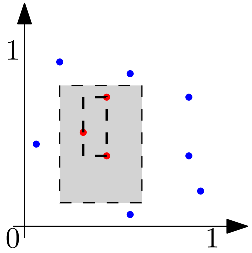
The Algorithm $\mathcal{A}_{\mathsf{BFP}}$

- We could define an Algorithm $\mathcal{A}_{\mathsf{BFP}}$ as follows:
  1. Given the data $((x_1, y_1), p_1), \ldots, ((x_n, y_n), p_n)$;
  2. Determine the smallest rectangle R including all the positive instances;
  3. Return R.
- In the probelm instance from the previous slides, one would get, when running $\mathcal{A}_{\mathsf{BFP}}$, the little bold rectangle. Of course, the result is always a sub-rectangle of the target rectangle.



The Algorithm $\mathcal{A}_{\mathsf{BFP}}$

- We could define an Algorithm $\mathcal{A}_{\mathsf{BFP}}$ as follows:
  1. Given the data $((x_1, y_1), p_1), \ldots, ((x_n, y_n), p_n)$;
  2. Determine the smallest rectangle R including all the positive instances;
  3. Return R.
- In the probelm instance from the previous slides, one would get, when running $\mathcal{A}_{\mathsf{BFP}}$, the little bold rectangle. Of course, the result is always a sub-rectangle of the target rectangle.



- The output classifier is a rectangle, which can be easily represented as a pair of coordinates.

Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
    - Is the algorithm balantantly incorrect, then?

Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
  - Is the algorithm balantantly incorrect, then?
- The answer is **negative**.

Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
  - Is the algorithm balantantly incorrect, then?
- The answer is **negative**.
- For a given rectangle R and a target rectangle T, the *probability of error* in using R as a replacement of T (when the distribution is $\mathbf{D}$) is $error_{\mathbf{D},T}(R) = \Pr_{x\sim\mathbf{D}}[x \in (R - T) \cup (T - R)]$.

Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
  - Is the algorithm balantantly incorrect, then?
- The answer is **negative**.
- For a given rectangle R and a target rectangle T, the *probability of error* in using R as a replacement of T (when the distribution is $\mathbf{D}$) is $error_{\mathbf{D},T}(R) = \Pr_{x\sim\mathbf{D}}[x \in (R-T) \cup (T-R)]$.
- As the number of samples in D grows, the result $\mathcal{A}_{\mathsf{BFP}}(D)$ does *not* necessarily approach the target rectangle, but its probability of error approaches zero.

Theorem

For every distribution $\mathbf{D}$*, for every* $0 < \varepsilon < \frac{1}{2}$ *and for every* $0 < \delta < \frac{1}{2}$*, if* $m \geq \frac{4}{\varepsilon}\ln\left(\frac{4}{\delta}\right)$*, then*

$$\Pr_{D\sim\mathbf{D}^m}[error_{\mathbf{D},T}(\mathcal{A}_{\mathsf{BFP}}(D)) < \varepsilon] > 1-\delta$$

The General Model — Terminology

- We assume to work within an **instance space** X.
  - X is the set of (encodings) of instances of objects the learner wants to classify.
  - Data from the instance spaces are generated through a distribution $\mathbf{D}$, unknown to the learner.
  - In the example, $X = \mathbb{R}^2_{[0,1]}$.

The General Model — Terminology

- We assume to work within an **instance space** X.
  - X is the set of (encodings) of instances of objects the learner wants to classify.
  - Data from the instance spaces are generated through a distribution $\mathbf{D}$, unknown to the learner.
  - In the example, $X = \mathbb{R}^2_{[0,1]}$.
- **Concepts** are subsets of X, i.e. collections of objects. These should be thought of as properties of objects.
  - In the example, concepts are arbitrary subsets of $X = \mathbb{R}^2_{[0,1]}$, i.e. arbitrary regions within $\mathbb{R}^2_{[0,1]}$.

The General Model — Terminology

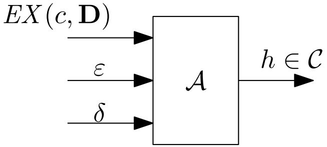
- We assume to work within an **instance space** X.
  - X is the set of (encodings) of instances of objects the learner wants to classify.
  - Data from the instance spaces are generated through a distribution $\mathbf{D}$, unknown to the learner.
  - In the example, $X = \mathbb{R}^2_{[0,1]}$.
- **Concepts** are subsets of X, i.e. collections of objects. These should be thought of as properties of objects.
  - In the example, concepts are arbitrary subsets of $X = \mathbb{R}^2_{[0,1]}$, i.e. arbitrary regions within $\mathbb{R}^2_{[0,1]}$.
- A **concept class** $\mathcal{C}$ is a collection of concepts, namely a subset of $\mathcal{P}(X)$. These are the concepts which are sufficiently simple to describe, and that algorithms can handle.
  - The concept class $\mathcal{C}$ we work with in the example is the one of rectangles whose sides are parallel to the axes.
  - The **target concept** $c \in \mathcal{C}$ is the concept the learner wants to build a classifier for.

The General Model — The Learning Algorithm $\mathcal{A}$

- Every learning algorithm is designed to learn concepts from a concept class $\mathcal{C}$ but it *does not know* the target concept $c \in \mathcal{C}$, nor the associated distribution $\mathbf{D}$.

The General Model — The Learning Algorithm $\mathcal{A}$

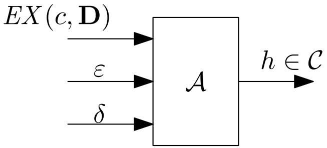
- Every learning algorithm is designed to learn concepts from a concept class $\mathcal{C}$ but it *does not know* the target concept $c \in \mathcal{C}$, nor the associated distribution $\mathbf{D}$.
- The interface of any learning algorithm $\mathcal{A}$ can be described as follows:



  where:
  - ε is **error parameter**, while δ is the **confidence parameter**;
  - $EX(c, \mathbf{D})$ should be though as an *oracle*, a procedure that $\mathcal{A}$ can call as many times she wants, and which returns an element $x \sim \mathbf{D}$ from X, labelled according to whether it is in c or not.

The General Model — The Learning Algorithm $\mathcal{A}$

- Every learning algorithm is designed to learn concepts from a concept class $\mathcal{C}$ but it *does not know* the target concept $c \in \mathcal{C}$, nor the associated distribution $\mathbf{D}$.
- The interface of any learning algorithm $\mathcal{A}$ can be described as follows:



  where:
  - ε is **error parameter**, while δ is the **confidence parameter**;
  - $EX(c, \mathbf{D})$ should be though as an *oracle*, a procedure that $\mathcal{A}$ can call as many times she wants, and which returns an element $x \sim \mathbf{D}$ from X, labelled according to whether it is in c or not.
- The **error** of any $h \in \mathcal{C}$ is defined as $error_{\mathbf{D},c}(h) = \Pr_{x\sim\mathbf{D}}[h(x) \neq c(x)]$.

The General Model — PAC Concept Classes

- Let $\mathcal{C}$ be a concept class over the instance space X. We say that $\mathcal{C}$ is **PAC learnable** iff there is an algorithm $\mathcal{A}$ such that for every $c \in \mathcal{C}$, for every distribution $\mathbf{D}$, for every $0 < \varepsilon < \frac{1}{2}$ and for every $0 < \delta < \frac{1}{2}$, is such that

$$\Pr[error_{\mathbf{D},c}(\mathcal{A}(EX(c,\mathbf{D}),\varepsilon,\delta)) < \varepsilon] > 1 - \delta$$

  where the probability is taken over the calls to $EX(c,\mathbf{D})$

The General Model — PAC Concept Classes

- Let $\mathcal{C}$ be a concept class over the instance space X. We say that $\mathcal{C}$ is **PAC learnable** iff there is an algorithm $\mathcal{A}$ such that for every $c \in \mathcal{C}$, for every distribution $\mathbf{D}$, for every $0 < \varepsilon < \frac{1}{2}$ and for every $0 < \delta < \frac{1}{2}$, is such that

$$\Pr[error_{\mathbf{D},c}(\mathcal{A}(EX(c, \mathbf{D}), \varepsilon, \delta)) < \varepsilon] > 1 - \delta$$

  where the probability is taken over the calls to $EX(c, \mathbf{D})$
- If the time complexity of $\mathcal{A}$ is bounded by a polynomial in $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$, we say that $\mathcal{C}$ is **efficiently PAC learnable**.
  - The complexity of $\mathcal{A}$ is measured taking into account the number of calls to $EX(c, \mathbf{D})$.

The General Model — PAC Concept Classes

- Let $\mathcal{C}$ be a concept class over the instance space X. We say that $\mathcal{C}$ is **PAC learnable** iff there is an algorithm $\mathcal{A}$ such that for every $c \in \mathcal{C}$, for every distribution $\mathbf{D}$, for every $0 < \varepsilon < \frac{1}{2}$ and for every $0 < \delta < \frac{1}{2}$, is such that

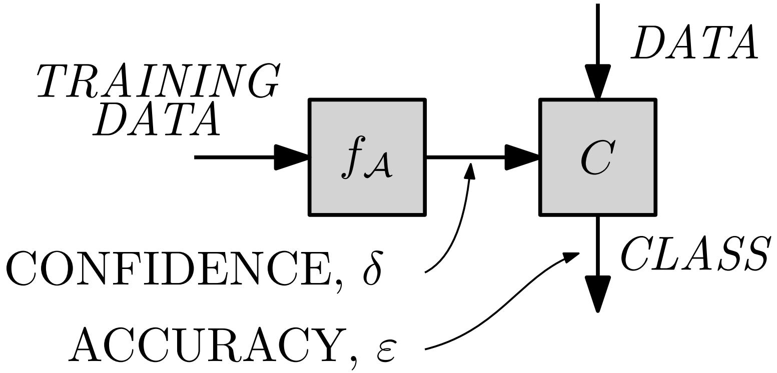
  $$\Pr[error_{\mathbf{D},c}(\mathcal{A}(EX(c,\mathbf{D}),\varepsilon,\delta)) < \varepsilon] > 1 - \delta$$

  where the probability is taken over the calls to $EX(c,\mathbf{D})$
- If the time complexity of $\mathcal{A}$ is bounded by a polynomial in $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$, we say that $\mathcal{C}$ is **efficiently PAC learnable**.
  - The complexity of $\mathcal{A}$ is measured taking into account the number of calls to $EX(c,\mathbf{D})$.

Corollary

The concept-class of axis-aligned rectangles over $\mathbb{R}^2_{[0,1]}$ is efficiently PAC-learnable.

The General Model — Two Kinds of Errors



Representation Classes

- In our definition of efficient PAC learning, the algorithm $\mathcal{A}$, having no access to the target concept $c \in \mathcal{C}$, must work in polynomial time **independently** on c. This is unrealistic in many cases. We remedy all this as follows:

Representation Classes

- In our definition of efficient PAC learning, the algorithm $\mathcal{A}$, having no access to the target concept $c \in \mathcal{C}$, must work in polynomial time **independently** on c. This is unrealistic in many cases. We remedy all this as follows:
  - We assume that elements of the instance space can be represented by binary strings.
  - We assume concepts in $\mathcal{C}$ can be represented by way of binary strings, and each concept $e \in \mathcal{C}$ requires $size(e)$ bits.

Representation Classes

- In our definition of efficient PAC learning, the algorithm $\mathcal{A}$, having no access to the target concept $c \in \mathcal{C}$, must work in polynomial time **independently** on c. This is unrealistic in many cases. We remedy all this as follows:
  - We assume that elements of the instance space can be represented by binary strings.
  - We assume concepts in $\mathcal{C}$ can be represented by way of binary strings, and each concept $e \in \mathcal{C}$ requires $size(e)$ bits.
- We then organize X as $\bigcup_n X_n$ and $\mathcal{C}$ as $\bigcup_n \mathcal{C}_n$ and any concept in $\mathcal{C}_n$ takes in input elements of X_n.
- Such a pair $X, \mathcal{C}$ is called a **representation class**.

Representation Classes

- Some examples:
  - X_n could be $\{0, 1\}^n$, the set of **boolean vectors** of of (fixed!) length n, and $\mathcal{C}_n$ is the set of all subsets of $\{0, 1\}^n$ *represented by CNFs* on precisely n variables.
  - X_n could rather be $\mathbb{R}^n$, the set of **vectors of real numbers** of length n, while $\mathcal{C}_n$ are say, the subsets of $\mathbb{R}^n$ *represented by some form of neural network* with n inputs and 1 output.

Representation Classes

- Some examples:
  - X_n could be $\{0,1\}^n$, the set of **boolean vectors** of of (fixed!) length n, and $\mathcal{C}_n$ is the set of all subsets of $\{0,1\}^n$ *represented by CNFs* on precisely n variables.
  - X_n could rather be $\mathbb{R}^n$, the set of **vectors of real numbers** of length n, while $\mathcal{C}_n$ are say, the subsets of $\mathbb{R}^n$ *represented by some form of neural network* with n inputs and 1 output.
- In many cases (e.g. SGD), one has a *single* learning algorithm that work for every value of n. In that case, we allow (in the definition of efficient PAC learning) the algorithm $\mathcal{A}$ to take time polynomial in n, $\mathit{size}(c)$, $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$.

Boolean Functions as a Representation Class

- Suppose your instance class is $X = \bigcup_n X_n$ where $X_n = \{0,1\}^n$.

Boolean Functions as a Representation Class

- Suppose your instance class is $X = \bigcup_n X_n$ where $X_n = \{0,1\}^n$.
- One **first example** of a representation class for X_n is the class $\mathcal{CL}_n$ of all *conjunctions of literals* on the variables $x_1, \ldots, x_n$.
  - As an example, the conjunction

    $$x_1 \wedge \neg x_2 \wedge x_4,$$

    defines a subset of $\{0,1\}^4$.
  - *Not all* subsets of $\{0,1\}^n$ can be captured this way.

Boolean Functions as a Representation Class

- Suppose your instance class is $X = \bigcup_n X_n$ where $X_n = \{0,1\}^n$.
- One **first example** of a representation class for X_n is the class $\mathcal{CL}_n$ of all *conjunctions of literals* on the variables $x_1, \ldots, x_n$.
  - As an example, the conjunction
    $$x_1 \wedge \neg x_2 \wedge x_4,$$
    defines a subset of $\{0,1\}^4$.
  - *Not all* subsets of $\{0,1\}^n$ can be captured this way.
- A **second example** of a representation class for X_n is a class we know, namely the class $\mathcal{CNF}_n$ of CNFs over $x_1, \ldots, x_n$, which are conjunction *of disjunctions* of literals.
  - CNFs can representf any boolean functions.
  - *All* subsets of $\{0,1\}^n$ can be captured this way.
  - We could even consider $k\mathcal{CNF}_n$ rather than arbitrary one, but this way we would lose universality.

Learning Conjuctions of Literals

- Suppose your target concept is a conjunction of literals c on n variables $x_1, \ldots, x_n$. How could a learning algorithm proceed?

Learning Conjuctions of Literals

- Suppose your target concept is a conjunction of literals c on n variables $x_1, \ldots, x_n$. How could a learning algorithm proceed?
- Data are in the form (s, b) where $s \in \{0,1\}^n$ and $b \in \{0,1\}$. The latter is a label telling us whether $s \in c$ or $s \notin c$.
- A learning algorithm could proceed by keeping a conjunction of literals h as its state, initially set to

  $$x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge \cdots \wedge x_n \wedge \neg x_n.$$

  and updating it according to positive data (while negative data are discarded).
  - If $n = 3$, the current state of h is $x_1 \wedge x_2 \wedge \neg x_2 \wedge \neg x_3$ and we receive $(101, 1)$, the hypothesis h is updated as $x_1 \wedge \neg x_2$.

Learning Conjuctions of Literals

- Suppose your target concept is a conjunction of literals c on n variables $x_1, \dots, x_n$. How could a learning algorithm proceed?
- Data are in the form (s, b) where $s \in \{0,1\}^n$ and $b \in \{0,1\}$. The latter is a label telling us whether $s \in c$ or $s \notin c$.
- A learning algorithm could proceed by keeping a conjunction of literals h as its state, initially set to

  $$x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge \cdots \wedge x_n \wedge \neg x_n.$$

  and updating it according to positive data (while negative data are discarded).
  - If $n = 3$, the current state of h is $x_1 \wedge x_2 \wedge \neg x_2 \wedge \neg x_3$ and we receive $(101, 1)$, the hypothesis h is updated as $x_1 \wedge \neg x_2$.

Theorem

The representation class of boolean conjuctions of literals is efficiently PAC-learnable.

Intractability of Learning DNFs

- We know that conjunctions of literals are efficiently learnable. But they are highly incomplete as a way to represent boolean functions.

Intractability of Learning DNFs

- We know that conjunctions of literals are efficiently learnable. But they are highly incomplete as a way to represent boolean functions.
- Let us take a look at a *slight generalization* of conjunctions of literals as a representation class.
  - A **3-term DNF formula** over n bits is a propositional formula in the form $T_1 \vee T_2 \vee T_3$, where each T_i is a conjunction of literals over $x_1, \ldots, x_n$.
  - In a sense, this class is the *dual* to 3CNFs!
  - As such, it is more expressive than conjunctions of literals, but still not universal.

Intractability of Learning DNFs

- We know that conjunctions of literals are efficiently learnable. But they are highly incomplete as a way to represent boolean functions.
- Let us take a look at a *slight generalization* of conjunctions of literals as a representation class.
  - A **3-term DNF formula** over n bits is a propositional formula in the form $T_1 \vee T_2 \vee T_3$, where each T_i is a conjunction of literals over $x_1, \ldots, x_n$.
  - In a sense, this class is the *dual* to 3CNFs!
  - As such, it is more expressive than conjunctions of literals, but still not universal.

Theorem

If $\mathbf{NP} \neq \mathbf{RP}$*, then the representation class of* 3*-term DNF formulas is not efficiently PAC learnable.*

Is This the End of the Story?

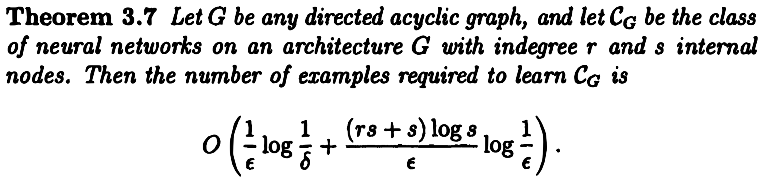
Is This the End of the Story?

- **Definitely No!** Actually, we have just *scratched the surface* of computational learning theory.

Is This the End of the Story?

- **Definitely No!** Actually, we have just *scratched the surface* of computational learning theory.
- Models and results we did not have time to talk about include:
  - The VC Dimension.
  - The Fundamental Theorem of Learning.
  - The No-Free-Lunch Theorem.
  - Occam's Razor.
  - Positive and negative results about neural networks.
  - ...
- More information can be found in of the many excellent books on CLT, e.g.
  - Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar *Foundations of Machine Learning* Second Edition. The MIT Press. 2018
  - Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: from Theory to Algorithms* Cambridge University Press. 2014.
  - Michael Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory* The MIT Press. 1994.

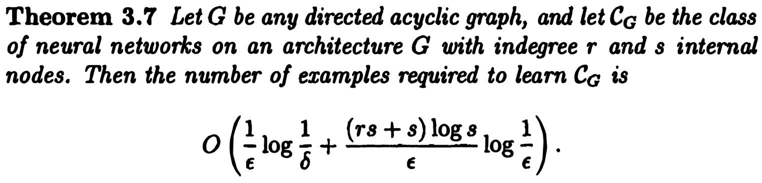
Example Results about Neural Networks (from Kearns and Vazirani's Book)



Theorem 3.7 *Let G be any directed acyclic graph, and let C_G be the class of neural networks on an architecture G with indegree r and s internal nodes. Then the number of examples required to learn C_G is*

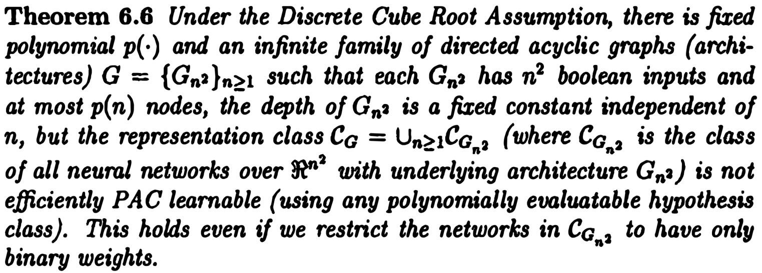
$$O\left(\frac{1}{\epsilon}\log\frac{1}{\delta}+\frac{(rs+s)\log s}{\epsilon}\log\frac{1}{\epsilon}\right).$$

Example Results about Neural Networks (from Kearns and Vazirani's Book)



Theorem 3.7 *Let G be any directed acyclic graph, and let C_G be the class of neural networks on an architecture G with indegree r and s internal nodes. Then the number of examples required to learn C_G is*

$$O\left(\frac{1}{\epsilon}\log\frac{1}{\delta} + \frac{(rs+s)\log s}{\epsilon}\log\frac{1}{\epsilon}\right).$$



Theorem 6.6 *Under the Discrete Cube Root Assumption, there is fixed polynomial $p(\cdot)$ and an infinite family of directed acyclic graphs (architectures) $G = \{G_{n^2}\}_{n\geq 1}$ such that each G_{n^2} has n^2 boolean inputs and at most $p(n)$ nodes, the depth of G_{n^2} is a fixed constant independent of n, but the representation class $C_G = \cup_{n\geq 1} C_{G_{n^2}}$ (where $C_{G_{n^2}}$ is the class of all neural networks over $\Re^{n^2}$ with underlying architecture G_{n^2}) is not efficiently PAC learnable (using any polynomially evaluatable hypothesis class). This holds even if we restrict the networks in $C_{G_{n^2}}$ to have only binary weights.*

Thank You!

Questions?