Bridging Computational Complexity and Machine Learning

Meng Song

September 22, 2018

1 Introduction

Computational complexity theory is a central field of the theoretical foundations of Computer Science throughout its history, which aims to study the intrinsic complexity of computational tasks. One direction of complexity theory focuses on determining or proving the limitations on computers with finite resources. The other one explores the connections among computational problems and notions, which is considered as a high-level study of computation [2]. By pushing this line of thoughts towards a broader level, we would like to examine the connections between computational complexity which is established on rigorous mathematical analysis and a young and empirical field machine learning. We will demonstrate their connections in two directions. From the perspective of computational complexity, we will try to measure the complexity of learning algorithms along the dimension of computation and introduce the open problems in this direction. From the opposite perspective, we will review recent work on how deep learning people revisit the fundamental concept Turing Machines in computability by designing neural networks to simulate it.

2 Computational Complexity of Learning

2.1 Introduction

Most of the machine learning algorithms can be viewed as acquiring general concepts from specific training examples. For computational purposes, we can formally define machine learning algorithms as the following:

Definition 1 Machine Learning Algorithms *Given a set of instances $X = \{x_i\}$ which is randomly drawn from a fixed unknown distribution D, a set of possible target concepts $C = \{c\}$ and a set of hypotheses H. A learning algorithm (learner L) takes training samples $S = \{< x_i, c(x_i) >\}$ as input, and outputs $h \in H$ estimating c. (Note that during this training process, the learner does not know c) Then h is evaluated by its performance on a new set of test instances X_t drawn from D. [5]*

Typically in computational complexity theory, we measure the computational complexity of an algorithm as a function of its input length, which in our case is the size of training set X. However, this is wrong for learning algorithms because how much computational effort is needed for a learner to converge (with high probability) to a successful hypothesis depends not only on the training set size but also a combination of factors such as the required accuracy, confidence and more importantly the complexity of the hypothesis class. For example, if not considering the complexity of output hypothesis, a learner could simply memorize all training samples, thus transfer the computational burden from the learning process to the output hypothesis and dramatically reduce the test accuracy and its generalization ability. Therefore, an effective input size of a learning algorithm is the sample complexity.

Loosely speaking, the sample complexity of a learning problem is considered as the growth in the number of required training samples with problem size. Formally, it can be defined as following:

Definition 2 Sample Complexity *Let H be a hypothesis class of functions defined over an instance space of size d, let ϵ, δ be accuracy and confidence parameters, and let m be the number of training samples. Let $m(H, \epsilon, \delta)$ be the sample complexity of learning H with accuracy ϵ and confidence $1 - \delta$.*

By analyzing the computational complexity of learning as a function of the sample complexity, we have the following definition:

Definition 3 Computational Complexity of Learning *The computational complexity of* (ϵ, δ)*-learning* H *is said to be bounded by* $T(dm(H, \epsilon, \delta))$ *if there exists a learning algorithm that* (ϵ, δ)*-learns* H *and whose runtime is bounded by* $T(dm(H, \epsilon, \delta))$ *and such that the runtime of the hypothesis it outputs on any new instance* x *is also bounded by* $T(dm(H, \epsilon, \delta))$*.*

2.2 PAC Learning Model

2.2.1 Problem Setting

In this part, we are aiming to characterize classes of target concepts that are learnable in polynomial time. Therefore, we introduce the PAC learning which put the learner under a relaxed condition where we require only that the learner probably learn a hypothesis that is approximately correct.

To specify the problem setting that defines the PAC learning model, we further requires that in definition 1, each target concept c in C corresponds to some subset of X, or equivalently to some boolean-valued function $c : X \to \{0, 1\}$. Within this setting, we are interested in characterizing the performance of various learners L using various hypothesis spaces H, when learning individual target concepts drawn from various classes C. Because we demand that L is general enough to learn any target concept from C regardless of the distribution of training examples, we will often be interested in worst-case analyses over all possible target concepts from C and all possible instance distributions D.

Definition 4 True Error of Hypothesis *To quantify how closely the learner's output hypothesis* h *approximates the actual target concept* c*, we define that the true error of hypothesis* h *with respect to target concept* c *and distribution* D *is the probability that* h *will misclassify an instance drawn at random according to* D*.*

$$error_D(h) \equiv Pr_{x \in D}[c(x) \neq h(x)]$$

2.2.2 PAC Learnability

Definition 5 PAC Learnability *Consider a concept class* C *defined over a set of instances* X *of length* n *and a learner* L *using hypothesis space* H*.* C *is* **PAC-learnable** *by* L *using* H *if for all* $c \in C$*, distributions* D *over* X*,* ϵ *such that* $0 < \epsilon < 1/2$*, and* δ *such that* $0 < \delta < 1/2$*, learner* L *will with probability at least* $(1 - \delta)$ *output a hypothesis* $h \in H$ *such that* $error_D(h) \leq \epsilon$*, in time that is* **polynomial** *in* $1/\epsilon$*,* $1/\delta$*,* d*, and* $size(c)$*.*

This definition requires that: (1) L must output a hypothesis having arbitrarily low error ϵ with arbitrarily high probability $(1-\delta)$. (2) L must do so in time that grows at most polynomially with $1/\epsilon$ and $1/\delta$, d which is the size of instances in X, and $size(c)$ which is the encoding length of c.

2.3 Sample Complexity for Finite Hypothesis Spaces

Next, we would like to further present a general bound on the sample complexity for a very broad class of learners, called consistent learners. A learner is consistent if it outputs hypothesis h that perfectly fit the training data which is formally described as

Definition 6 *A hypothesis* h *is consistent with a set of training examples* S *of target concept* c *if and only if* $h(x) = c(x)$ *for each training example* $< x, c(x) >$ *in* S*.*

Then we would like to derive a bound on the number of training examples required by any consistent learner, independent of the specific algorithm it uses to derive a consistent hypothesis. To accomplish this, we introduce the concept version space.

Definition 7 *The version space,* $VS_{H,S}$ *with respect to hypothesis space* H *and training examples* S *is the subset of hypotheses from* H *consistent with all training examples in* S*.*

$$VS_{H,S} = \{h \in H | (\forall < x, c(x) > \in S, \quad h(x) = c(x))\}$$

With $VS_{H,S}$, to bound the number of examples needed by any consistent learner, we need only bound the number of examples needed to assure that the version space contains no unacceptable hypotheses. Therefore, we have

Definition 8 *The version space $VS_{H,S}$ is said to be ϵ-exhausted with respect to c and S if every hypothesis h in $VS_{H,S}$ has error less than ϵ with respect to c and S.*

$$\forall h \in VS_{H,S}, \quad \hat{\epsilon}_S(h) < \epsilon$$

Theorem 1 ϵ-exhausting the version space *If the hypothesis space H is finite, and S is a sequence of $m \geq 1$ independent random examples of some target concept c, then for any $0 \leq \epsilon \leq 1$, the probability that the version space with respect to H and S is not ϵ-exhausted (with respect to c) is less than or equal to $|H|e^{-\epsilon m}$.*

This means that the probability that the version space is not ϵ-exhausted after m training examples is at most $|H|e^{-\epsilon m}$. Suppose we want this probability to be at most δ

$$|H|e^{-\epsilon m} \leq \delta$$

Then the number of training examples required is at least

$$m \geq \frac{1}{\epsilon}(ln|H| + ln(1/\delta)) \tag{1}$$

Note that m grows linearly in $\frac{1}{\epsilon}$ and logarithmically in $1/\delta$ and the size of the hypothesis space H.

2.3.1 k-term DNF

In some cases of finite hypothesis space, there are concept classes that have polynomial sample complexity but nevertheless cannot be learned in polynomial time.

One interesting example is the concept class C of k-term disjunctive normal form expressions (k-term DNF). This problem has instance space $\{0,1\}^d$ and each hypothesis can be represented by the boolean formula of the form $A_1(x) \vee A_2(x) \vee \cdots \vee A_k(x)$, where each $A_i(x)$ is a conjunction of literals over the boolean variables $x_1, \ldots, x_d$.

The number of such k-term DNF is at most 3^{dk}. Assuming $H = C$, it is easy to show that $|H|$ is at most 3^{dk}. Note that this is an overestimate of H, but can still be used as an upper bound on the sample complexity. By substituting this into equation 1, we have

$$m \geq \frac{1}{\epsilon}(dkln3 + ln(1/\delta))$$

which indicates that the sample complexity of k-term DNF is polynomial in $1/\epsilon$, $ln(1/\delta)$, d and k.

Despite having polynomial sample complexity, the computational complexity of k-term DNF is not polynomial, because it has been shown that unless $RP = NP$, there is no polynomial time algorithm that properly learns a sequence of k-term DNF formula, where "properly" means that the algorithm should output a k-term DNF hypothesis. Because this learning problem can be shown to be equivalent to other problems that are known to be unsolvable in polynomial time. The proof uses a reduction of the graph coloring problem to this problem. Thus, although k-term DNF has polynomial sample complexity, it does not have polynomial computational complexity for a learner using $H = C$.

Although k-term DNF is not PAC-learnable, we can efficiently learn it by introducing improper learning [6].

2.3.2 Improper Learning

The intuitive idea is that by starting with a hypothesis class for which efficient learning is hard, we gradually switch to another representation and define a larger concept class which has more structure and is PAC-learnable. By allowing the output hypothesis to be this new class, the original concept becomes PAC learnable.

To illustrate this idea on k-term DNF, we begin with an observation that each k-term DNF formula is equivalent to a k-CNF formula, that is $A_1 \vee A_2 \vee \cdots \vee A_k = \wedge_{B1 \in A1, B2 \in A2, \ldots B_k \in A_k}(B_1 \vee$

$B_2 \vee \cdots \vee B_k$). k-CNF subsumes k-DNF because not any k-CNF can be rewritten as a k-DNF. k-CNF has both polynomial sample complexity and polynomial time complexity.

To prove that k-CNF has polynomial time complexity, we observe that for each literals $B_1 \vee B_2 \vee \cdots \vee B_k$, there are two variables indicating whether $B_1 \vee B_2 \vee \cdots \vee B_k$ is true of false. Thus there exists a Halfspace $sign(< \mathbf{w}, \psi(\mathbf{x}) > +b)$ with the same truth table as each k-DNF formula, where ψ is a mapping: $\psi : \{0,1\}^d \rightarrow \{0,1\}^{2(2d)^k}$. The VC dimension of this Halfspace is its dimension, thus the sample complexity of learning the class of Halfspace is order of $\frac{d^3}{\epsilon} \log \frac{1}{\delta}$. The overall runtime of this approach is $poly(\frac{d^3}{\epsilon} \log \frac{1}{\delta})$.

In machine learning, the class of DNF formulas is useful for learning decision trees. The exact time complexity of improperly learning DNFs is still an interesting open problem.

2.4 Sample Complexity for Infinite Hypothesis Spaces

When the hypothesis space is infinite, we cannot characterizing sample complexity in terms of $|H|$. Instead, we introduce a second measure of the complexity of H, called the Vapnik-Chervonenkis dimension of H (VC dimension, or $VC(H)$. In many cases, the sample complexity bounds based on $VC(H)$ will be tighter than those from equation 1.

The VC dimension measures the complexity of the hypothesis space H, not by the number of distinct hypotheses $|H|$, but instead but the number of distinct instances from X that can be completely discriminated using H.

To formalize this precisely, we first define the notion of shattering a set of instances.

Definition 9 *A set of instances S is* **shattered** *by hypothesis space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy.*

The ability to shatter a set of instances is closely related to the inductive bias of a hypothesis space. The VC dimension of H precisely measure this ability as following:

Definition 10 *The Vapnik-Chervonenkis dimension, $VC(H)$, of hypothesis space H defined over instance space X is the size of the largest finite subset of X shattered by H. If arbitrarily large finite sets of X can be shattered by H, then $VC(H) \equiv \infty$.*

Having VC dimension, we can derive a new tighter upper bound for sufficient number of training examples analogous to equation 1:

$$m \geq \frac{1}{\epsilon}(4 \log_2(2/\delta) + 8VC(H) \log_2(13/\epsilon))$$

Note that differ from equation 1, the number of required training examples m grows log times linear in $1/\epsilon$, rather than linearly. And also we can derive a lower bound on sample complexity.

Theorem 2 Lower bound on sample complexity *Consider any concept class C such that $VC(C) \geq 2$, any learner L, and any $0 < \epsilon < \frac{1}{8}$, and $0 < \delta < \frac{1}{100}$. Then there exists a distribution D and target concept in C such that if L observes fewer examples than*

$$max[\frac{1}{\epsilon} \log(1/\delta), \frac{VC(C)-1}{32\epsilon}]$$

Then with probability at least δ, L outputs a hypothesis h having $error_D(h) > \epsilon$

2.5 Hardness of Learning

In previous parts, most of the cases where our conclusions build on are aiming at solving the most "basic" learning algorithm ERM problem.

Definition 11 Empirical Risk Minimization Paradigm *Given a hypothesis class H, for an input training examples S, ERM is the class of learning algorithm that looks for some $h \in H$ that fits S well. Then for a new point x, the algorithm will predict a label according to its membership in h.*

$$\hat{h} = \operatorname*{argmin}_{h \in H} \hat{\epsilon}_S(h)$$

For example, linear regression is ERM when $V(z) = (f(x) - y)^2$ and H is space of linear functions $f = \mathbf{w}x + b$. For ERM to represent a "good" class of learning algorithms, the solution should generalize well, exist, be unique and especially be stable.

However, to establish hardness of learning it is not enough to show that solving the ERM problem is hard. We need to show that no representation of the problem under which the problem becomes tractable. In this part, we would like to prove that learning problems (or even properly learning) are hard by cryptographic assumptions.

Many cryptographic systems are built on the assumption that there exists a one way function $f : \{0,1\}^d \to \{0,1\}^d$ which is easy to compute but is believed to be hard to invert. More formally speaking, f can be computed in time $poly(d)$ and for every randomized polynomial time algorithm A, and for every polynomial $p(\cdot)$

$$P[f(A(f(x))) = f(x)] < \frac{1}{p(d)}$$

where the probability is taken over a random choice of x according to the uniform distribution over $\{0,1\}^d$ and the randomness of A.

A one-way function is called **trapdoor one-way function** if it is hard to invert it as long as we don't know a secret key of length $poly(d)$. Such functions are parameterized by the secret key.

A **hard bit function** (a.k.a. hard-core predicate), $b : \{0,1\}^d \to \{0,1\}$, associated with a way function f, is a boolean function that can be computed in polynomial time but such that for every polynomial algorithm A and a polynomial p we have

$$P[A(f(x)) = b(x)] < \frac{1}{2} + \frac{1}{p(d)}$$

where the probability is taken over a random choice of x according to the uniform distribution over $\{0,1\}^d$ and the randomness of A. This implies that when x is uniform over $\{0,1\}^d$, $b(x)$ is almost an unbiased coin.

Let f be a trapdoor one way function, let b be a hard bit, and consider the following distribution over training examples which are represented as pair set $\{(x, y)\}$. First, z is chosen from $\{0,1\}^d$ according to the uniform distribution, then the instance x is set to be $x = f(z)$, and the label is set to be $y = b(z)$.

Consider the hypothesis class to be the set of trapdoor one way functions parameterized by the secret key. Since the length of the secret key is $poly(d)$, it means that the sample complexity of PAC learning this class is order of $poly(\frac{d}{\epsilon}, \log(\frac{1}{\delta}))$. But learning (even improperly) this class in polynomial time implies the existence of an algorithm for which $P[A(f(z)) = b(z)] > 1 - \epsilon$, which contradicts the fact that b is a hard bit for f.

2.6 Mismatch between NP-hardness Results and the Success of Machine Learning Algorithms in Reality

In the history of machine learning, many algorithms are invented to model how human think and understand the world. There has been tremendous progress within the field on solving a wide variety of practical problems, ranging from playing Go to autonomous driving. However, most of them are proved to be NP-hard. It is shown that very simple learning problems, even involving fairly low-level perception, such as recognizing valid polyhedron scenes [Kirousis and Papadimitriou'88], turn out to be NP-complete. To illustrate this point, we will name the complexity of a few of dominant machine learning methods here. Logic based approach [McCarthy 1959] which deducing new facts from a database of facts in even very simple systems is NP-complete. Bayesian Nets reasoning is NP-hard. The recent trendy deep learning model Convolutional Neural Networks have surpassed humans on the problem of image recognition. However, complexity theory tells us that learning even a shallow net with one hidden layer is intractable. [Klivans-Sherstov'10]

In this part, we attempt to understand this mismatch between theoretical obstacle and practical success. According to the recent talk given by Sanjeev Arora [1], it is important to realize that NP-hardness only concerns difficulty of worst instance, thus only relevant if we desire an algorithm that works for every input. Hence complexity only rules out the hardness of very complicated instance. In most of real world AI applications, such as image recognition, this means very strange dataset with strange labels which rarely happen.

Therefore, the emerging lessons in computational complexity are that for most natural problems, the natural phrasing is computationally intractable (NP-hard). But this need not mean the problem is intractable in practice. Actually it may even be solvable at industrial scales. Therefore NP-hardness is not an obstacle for understanding reality.

To illustrate this clearly, one can imagine that in the instance space, machine learning tasks based on real life datasets correspond to a set of instances, whereas the mathematical phrasing of the task includes a much broader range of instances, in which many weird instances are allowed. Then people may prove that there are NP-hard instances in this mathematical phrasing. However, this does not mean that the original set of instances, or even a superset of it, is not tractable. Because the mathematical definition is a human construct, an infinite number of different such mathematical models can be constructed to explain the concept. NP hardness just refers to the intractability of this particular mathematical phrasing. An interesting research direction is to developing new theory for machine learning (especially deep learning) to overcome these intractability.

3 Neural Turing Machines

3.1 Introduction

In the past few years, deep learning methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains. These achievements result from the ability of neural networks on extracting abstract semantic representations from various perception data. However, their ability on modeling computers is unclear. This requires neural networks to have three fundamental mechanisms which are arithmetic operations, logical flow control and external memory supporting written to and read from in the course of computation. In order to push the boundaries of artificial intelligence to automatically learn the behavior of computers, some deep learning researchers have introduced the Neural Turing Machines [3] which simulate a Turing Machine but is differentiable end-to-end and can be trained with gradient descent.

Turing Machines are the central concept which provides a simplified computation model capturing the full power of general purpose computing. To some extent, computability theory is to examine what kind of problems can a turing machine solve.

Before proceeding to the design of architecture, we would like to first recall the definition of Turing Machines in computation theory.

Definition 12 Turing Machines *A turing machine has an input tape, an output tape and at least one work tape. Tapes are divided into cells, each of which can record a single symbol. The input tape is read-only, the remaining are both read and write. There is a reading head on each tape which can be moved one step in each operation. It also has a finite state which can be used to represent code, registers, etc.*

Formally, a k-tape TM M is described by a tuple (Γ, Q, δ) containing:

- *Γ is a set of the symbols that M's tapes can contain.*
- *Q is a finite set of internal states (e.g. code, location in code, registers).*
- *A function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}^k$ describing the rule M uses in performing each step. This function is called the transition function of M.*

3.2 NTM Architecture

To make an clear analogy between a Turing machine and a NTM, one can think of the Turing machine as a simplified modern computer, with the machine's tape corresponding to a computer's memory, and the transition function and register corresponding to the computer's central processing unit (CPU).

A NTM is fundamentally composed of a neural network, called the controller, and a 2D matrix called the memory bank or memory matrix (See Fig. 1). At each time step, the neural network receives some input from the outside world, and sends some output to the outside world. It also has the ability to read and write the memory matrix via a set of parallel read and write heads, where by drawing inspiration from the traditional Turing machine, the term "head" describes the specification of memory location(s).

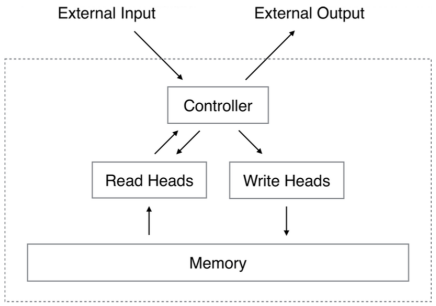

Figure 1: Neural Turing Machine Architecture.

3.3 Addressing Mechanisms

The memory matrix has N rows, each row indicates a memory location, and M is the vector size at each location. At time t, the reading or writing head emits a normalized weighting vector over the N locations. This weight vector performs as an attention focusing on a small portion of the memory.

The weights are produced by combining two addressing mechanisms. (1) Content-based addressing mechanism focuses attention on locations based on the similarity between the content at each memory location and value emitted by the controller. (2) The location-based addressing mechanism is designed to facilitate both simple iteration across the locations of the memory and random-access jumps. It does so by implementing a rotational shift of a weighting to the neighboring location. Prior to this shift, the interpolation between the content weighting and the weighting from the previous time step is produced for smoothness purpose. At the last step, to avoid leakage or dispersion of weightings over time, the weightings are further sharpened by a scalar. Fig. 2 shows the steps to produce the final weight vector w_t from previous state and controller outputs.

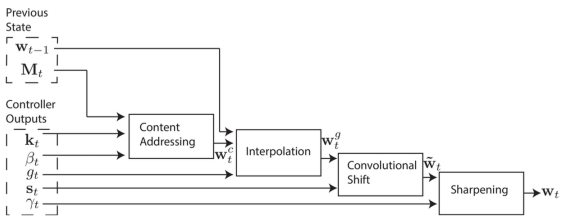

Figure 2: Flow Diagram of the Addressing Mechanism.

3.4 Reading and Writing

Differ from the counterparts in Turing Machines, the read and write operations are blurry in that they interact to a greater or lesser degree with all the elements in memory based on the weights vectors. Different from read, each write is decomposed into two parts: an erase followed by an add which are both differentiable.

3.5 Controller Network

There are two architecture choices for the controller. One is the feedforward network. The other more complicated one is recurrent controller such as LSTM (RNN). This type of controllers has its own internal memory that can complement the larger memory in the matrix. If one compares the controller to the central processing unit in a digital computer and the memory matrix to RAM, then the hidden activations of the recurrent controller are akin to the registers in the processor. They allow the controller to mix information across multiple time steps of operation. It is known that RNNs are Turing-Complete [Siegelmann and Sontag, 1995], and therefore have the capacity to simulate arbitrary procedures.

3.6 Experiments

The paper presented preliminary experiments on a set of simple algorithmic tasks such as copying, associative recall, and sorting data sequences with correct locations given as inputs. Although these tasks are simple and NTM is far from being comparable to TMs on completing complex computations, it has shown the ability to learn compact internal programs solely from data and generalize well, rather than relying on human designed programs. It turns out that these inspirations acquired from the computation concepts contribute a lot to the AI research community.

3.7 Differentiable Neural Computer

By extending NTM and coupling the system with a dynamic external memory matrix, a recent paper [4] proposes a neural network model called differentiable neural computer (DNC). Their results demonstrate that DNCs have the capacity to accomplish complex, structured tasks such as (1) Solving a moving blocks puzzle. (2) Answering synthetic questions designed to emulate reasoning and inference problems in natural language. (3) Finding the shortest path between specified points and inferring the missing links in randomly generated graphs, and then generalize these tasks to specific graphs such as transport networks and family trees.

References

[1] Sanjeev Arora. *Simons Institute Open Lecture: Does Computational Complexity Restrict Artificial Intelligence (AI) and Machine Learning?* 2017.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach.* 2009.

[3] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[4] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[5] Tom M. Mitchell. *Machine Learning.* WCB McGraw-Hill, 1997.

[6] Shai Shalev-Shwartz. *Understanding Machine Learning: From Theory to Algorithms.* 2014.