Auto-Regressive Next-Token Predictors are Universal Learners

Peroni Jacopo, Rahbari Aryan

Abstract

The following report summarizes the results of the paper "Auto-Regressive Next-Token Predictors are Universal Learners" by Eran Malach. It discusses the potential of auto-regressive learning combined with chain-of-thought data. Our report is also about the implementation of an auto-regressive linear predictor and its performance.

1 Theory

Large language models (LLMs) play an important role in modern machine learning. Models, like GPT-4 and LaMDA, are capable of achieving remarkable performances. They are trained with vast amounts of text data. Because of the large scales of these models, the natural question occurs if the performance of LLMs is achieved by memorizing the data and replicating data as an autocomplete system or if these models perform novel logical reasoning. The paper "Auto-regressive Next-Token Predictors are Universal Learners" by Eran Malach argues, both theoretically and experimentally, that the success of LLMs is based on the auto-regressive next-token scheme and not on a particular architecture. Even simple predictors working with linear models are able to perform well. The paper also introduces a new notion of complexity, namely *length complexity*, which provides information about the amount of data necessary in order to solve a given problem.

1.1 Auto-regressive next-token predictors

First, we will formally define, what auto-regressive learning is, how it relates to classical supervised learning and discuss the performance of linear auto-regressive models. Let $\mathbb{D}$ be the set of tokens, which is assumed to be finite. The set of contexts consisting of n tokens is denoted as $X = \mathbb{D}^n$ and the set of arbitrary long sequences of tokens is $Z = \mathbb{D}^*$. If we fix t, the set of strings of length t is $Z_t = \mathbb{D}^t$. In this setting an auto-regressive (abreviated as AR) function is a map $h : X \times Z \longrightarrow \mathbb{D}$, whose output is interpreted as the next token for a string of tokens with a given context. We can decompose Z into a direct union of Z_t and get a restriction $h_t : X \times Z_t \longrightarrow \mathbb{D}$. Therefore, if H is a class of certain auto-regressive functions, it can be decomposed into a direct product of H_t, where H_t is a class of AR-functions defined on $X \times Z_t$. Knowing that, we can come up with a formal definition of learnability in the framework of auto-regressive learning. Often, the length of strings is limited. By this reason, we often have $H = H_1 \times H_2 \times \cdots \times H_T$ for a given T.
Let $\hat{D}$ be a probability distribution defined on $X \times Z_T$.

Definition 1 *We say $\hat{D}$ is realizable by H if there exists a map $h \in H$ such that $h_{t-1}(x, z_{<t}) = z_t$ for for every almost every tuple $(x, z) \in \hat{D}$ given a $t \leq T$, where $z_{<t}$ is the substring of z from the first token until the $(t-1)$th token and z_t is the token in z at the t-th position.*

Now we can define AR-Learnability of H.

Definition 2 *The hypothesis class H is learnable if there exists an algorithm, such that for every ϵ, δ and $\hat{D}$, which is realizable by H, returns with a probability of at least $1-\delta$ a function $\hat{h}$ such that*

$$\mathbb{P}(\{t \leq T s.t. \hat{h}(x, z_{<t}) \neq z_t\}) \leq \epsilon$$

given a sample of size $m(\epsilon, \delta)$ by $\hat{D}$.

We say that H is efficiently learnable if $\hat{h}$ can be computed in polynomial time.

The following theorem stated in the paper relates PAC and AR learnability with each other.

Theorem 1 *Given a hypothesis class $H = H_1 \times \cdots \times H_T$. If each H_t is (efficiently) PAC learnable with sample complexity $m(\epsilon, \delta)$, then H is (efficiently) AR learnable with sample complexity $m(\epsilon/T, \delta/T)$.*

1.2 Linear Decoders

An important example of a class H which is efficiently AR learnable is the linear class $H^{lin} = H_1^{lin} \times \cdots \times H_T^{lin}$ for a fixed T. The elements of each H_t are linear classifiers of the form

$$h_W(x, z) = argmax_{D \in \mathbb{D}} \langle W_D, \psi(x, z) \rangle$$

with the $W = (W_D)_{D \in \mathbb{D}} \in \mathbb{R}^{\mathbb{D} \times d \times (n+t)}$ and an embedding $\psi : \mathbb{D} \longrightarrow \mathbb{R}^d$. Therefore, every element of H is a tuple of T linear classifiers. Each classifier can be trained given a convex loss function with the stochastic gradient descent (SGD) algorithm in an efficient manner. That a class H is efficiently learnable in the context of AR if each H_t is efficiently learnable in the context of PAC is a fact which is stated by the theorem above.
The paper shows that a linear auto-regressive model already is quite powerful and can compute a large class of functions. Now it is important to clarify, what we mean with "computing".
We start with a context x of n tokens. Given an AR hypothesis $h \in H$, we compute the next token by

$$h^{(1)}(x) = h_1(x, \emptyset),$$

then we include the token $h^{(1)}(x)$ in the next input and get our second token

$$h^{(2)}(x) = h_2(x, (h^{(1)}(x))).$$

We proceed with this scheme until we have

$$h^{(T)}(x) = h_T(x, (h^{(1)}(x), \ldots, h^{(T-1)}(x))).$$

We say that a function $f : \mathbb{D}^n \longrightarrow \mathbb{D}$ is computed by h, if we start with x as a context and finally get $h^{(T)}(x) = f(x)$. We can make an interesting observation. Although, every step is computed by a linear classifier, the auto-regressive scheme gives us a nonlinear result. It is the case that every Turing computable function can be approximated by a linear hypothesis provided the suitable data. Indeed, we have:

Theorem 2 *Any $f : \mathbb{D}^n \longrightarrow \mathbb{D}$, that is Turing computable in time $T(n)$ and any distribution $\hat{D}$ over inputs of size n, there is a dataset of strings of tokens of length $poly(T(n))$, such that training a linear AR model over this dataset leads to a function that computes f with respect to $\hat{D}$.*

That is also the weakness of next-token predictors. We need chains of tokens of the length T in order to train a linear predictor. Such data, which is called Chain-of-Thought data in this context, is not always available and the length T can be quite large. It might seem to be the case that we reduce complexity by training linear classifiers but the number T can be quite large. By this reason, it makes sense to introduce a new notion of complexity.

1.3 Length complexity

An interesting result of this paper is that any Turing computable function can be approximated by linear predictors, which can be trained efficiently. But we can not state that the complexity of the problem disappeared. In order to train the predictor we need strings of tokens, which represent

an internal "chain-of-thought". In this sense, we have a trade-off in complexity, namely between computational and length complexity.
We shall formally define length complexity. Let F be a class of functions mapping from $\mathbb{D}^n$ to $\mathbb{D}$ and H some AR hypothesis class.

Definition 3 *We say, that H computes F with length complexity T if*

$$\forall f \in F \exists h \in H : h^{(T)}(x) = f(x).$$

Basically, the length complexity T tells us how many individual predictors $h_1, \ldots, h_T$ we need, in order to compute f. If every "step of thought" t can be computed/approximated in H_t, we will need T such steps beginning from the context x in order to reach the final output $f(x)$.

Definition 4 *We say that H ϵ-approximates F with length complexity T if*

$$\forall f \in F : \exists h \in H : \mathbb{P}(h^{(T)}(x) \neq f(x)) \leq \epsilon.$$

Now we can apply length complexity as a tool for analyzing the following problem.
We will analyze a generalization of the XOR-problem, namely the problem of parities. We work with the dictionary $\mathbb{D} = \{0, 1\}$. Our input are n bits, denoted as $x = (x_1, \ldots, x_n)$, and our output has the form

$$\xi_A(x) = \sum_{i \in A} x_i \bmod 2$$

where A is a subset of $[n] = \{1, \ldots, n\}$.
It is possible to compute ξ_A by a circuit of linear threshold functions with $\mathcal{O}(\log n)$ gates. Since every gate can be computed by a linear classifier, there is an AR-function $h^{(T)}$ with $T = \mathcal{O}(\log n)$ which computes ξ_A.
We can conclude, that the parity function with n inputs has length complexity of

$$\mathcal{O}(\log n)$$

with respect to the hypothesis class H^{lin}, which is efficiently learnable by the usage of SGD as described in the previous subsection. With the classical supervised setting, algorithms such as the Statistical Query algorithms require a computational complexity of $\Omega(2^n)$. On the other hand, given appropriate chain-of-thought data consisting of $\mathcal{O}(\log n)$ tokens, the parity problem becomes an efficiently solvable in the setting of auto-regressive learning.
These theoretical findings motivate training auto-regressive next-token predictors on chain-of-thought data in order to solve tasks such as text generation or arithmetic multiplication. The paper experimentally demonstrates that AR achieves remarkable results even with the usage of simple models such as linear classifiers or shallow neural networks.

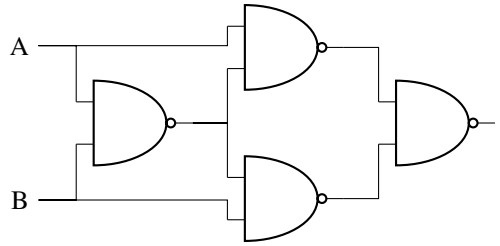
2 Experiment

2.1 Description & Implementation

The function that we tried to approximate is the following, given $\mathbf{x} \in \{0, 1\}^n$

$$f(\mathbf{x}) = \bigotimes_{i=1}^{n} \mathbf{x}_i$$

where $A \otimes B$ is the XOR operation. To do that we started from the reformulation of one XOR operation in NAND gates, that is the following



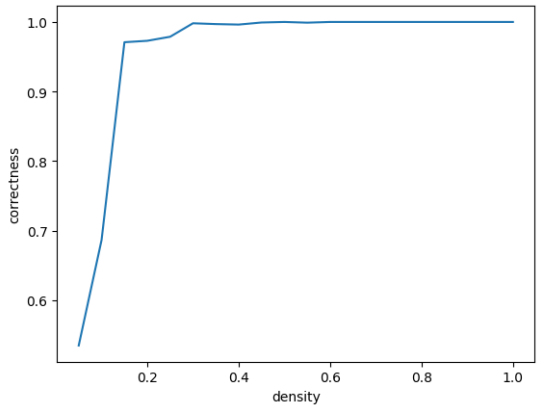


Figure 1: Case $n = 10$, 20 possible densities, 50 tests per density

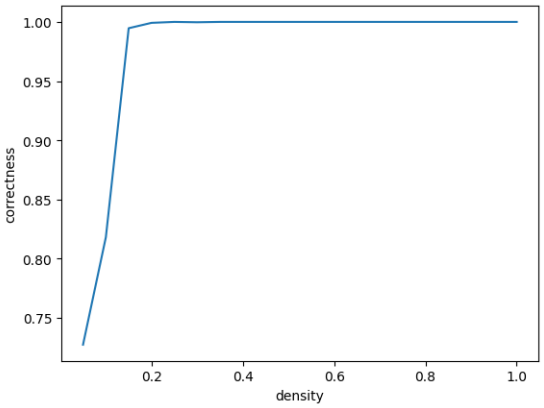


Figure 2: Case $n = 12$, 20 possible densities, 50 tests per density

with 4 gates, this tells us that the length complexity of the model will be $T = 4$.
The function f is a sequence of circuits like this one, so if we start with a sequence of n elements, the length complexity of the model will be $T = 4(n - 1)$.
We trained a chain of linear decoders, where each of them has the role of learning one gate, this gives us at the end one function $h^{(T)} : \mathbb{D}^n \longrightarrow \mathbb{D}$ that acts like f.

2.2 Result

Fixed n the possible inputs of this problem are 2^n, thus for small values of n it is easy to train on all possible cases, this gives always an equivalence between $h^{(T)}$ and f on every input.
The interesting case is when we train the model giving a smaller amount of data, we graphed the exactness of the model (computed as the number of correct guesses on a test set with a size of $0.5 * 2^n$) with respect to the amount of data given as training set (expressed as a number in $[0, 1]$ that is the percentage used of all the possible inputs).
We ran the experiment for $n = 10$ and $n = 12$, it was not possible to do for more than this amount due to computational limits, but the results suggest that this model would work anyway.

As we can see from Figure 1 and Figure 2 already at 20% of all possible inputs the model is very reliable.

3 Conclusion

From theory, we know that any Turing computable (TC) function can be expressed by a linear threshold circuit, this tells us that, with enough intermediate labels, any TC function can be learned

from an AR next token predictor, where each step trains a linear classifier.
This result gives the main problem related to this model, which is the computational cost of generating the labels in the chain of thoughts. In fact, if the number of gates (and so the number of labels) is too big, the problem could become infeasible.
This is exactly the problem faced by us in the experiments where the time required to make those labels was too much for the available computational power.
To conclude, there are two important aspects that we want to stress

- Even with a "small" amount of input data the model still acts well, giving us a faster way to compute the solution compared to the generator of the data we used to train the model.
- If the labels are given for "free" (e.g. scraping on the internet) this learning process becomes significantly faster than usual ML methods.