Ask, and it shall be given: On the Turing completeness of prompting

Ruizhong Qiu, Zhe Xu, Wenxuan Bao, Hanghang Tong {rq5,zhexu3,wbao4,htong}@illinois.edu UIUC ILLINOIS

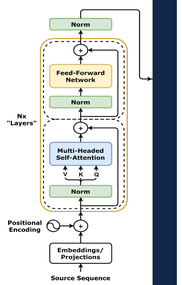
ICLR

HIGHLIGHTS

- **Our work**: the **first** theory on the LLM prompting paradigm.
  - Existing theories: the *one-model-**one**-task* paradigm;
  - *C.f.* ***LLM prompting***: the *one-model-**many**-tasks* paradigm.
- **Expressive power:** Prompting is **Turing-complete**.
  - *Not only existence*: a **simple** & **explicit** construction.
- **Complexity bounds**: For any $\mathrm{TIME}(t(n))$ function,
  - ***CoT complexity***: $\mathrm{O}(t(n)\log t(n))$ CoT steps;
  - ***Precision complexity***: $\mathrm{O}(\log(n+t(n)))$ bits.
  - **Nearly as efficient** as the **class of all Transformers**.
  - *(Not covered in this poster. See our paper for detail…)*

PRELIMINARIES

- **Background**:
  - ***Decoder-only Transformers***: mainstream architecture of LLMs.
  - ***Expressive power***: what functions a model class can represent.
- **Existing theories**: classic **one-model-one-task** paradigm.
  - The class of all hard-attention Transformers is Turing-complete;
  - Any $\mathrm{TIME}(t(n))$ function is computable in $\mathrm{O}(t(n))$ CoT steps.
- **Practice**: LLM prompting (i.e., **one-model-many-tasks**).
  - A **single** general-purpose LLM; **different** prompts for different tasks.
  - ❖ *Fundamentally, how powerful is this paradigm?*
- **Notations:**
  - Σ: an **alphabet** (i.e., the set of tokens).
    - Σ^*: the set of strings (possibly empty) over Σ.
    - Σ^+: the set of non-empty strings over Σ.
    - Σ^ω: the set of countably infinite strings over Σ.
  - $\Gamma\colon \Sigma^+ \to \Sigma$: a **decoder-only** Transformer.
    - Γ predicts the next-token via **greedy** decoding.
  - $\mathrm{generate}_\Gamma\colon \Sigma^+ \to \Sigma^+ \cup \Sigma^\omega$: autoregressive generation using Γ.
    - Here, the output of $\mathrm{generate}_\Gamma$ does not include the prompt.



TURING COMPLETENESS

- **Theorem**: There exist
  - a finite alphabet Σ, a finite-size decoder-only Transformer $\Gamma\colon \Sigma^+ \to \Sigma$,
  - and coding schemes $\mathrm{tokenize}\colon \{0,1\}^* \to \Sigma^*$ and $\mathrm{readout}\colon \Sigma^* \to \{0,1\}^*$
- with which prompting is **Turing-complete**, in the sense that
  - for every computable function $\varphi\colon \mathrm{dom}\,\varphi \to \{0,1\}^*$ with $\mathrm{dom}\,\varphi \subseteq \{0,1\}^*$,
  - there exists a **prompt** $\boldsymbol{\pi}_\varphi \in \Sigma^+$ such that for every input $\boldsymbol{x} \in \mathrm{dom}\,\varphi$,
  - $\mathrm{generate}_\Gamma\left(\boldsymbol{\pi}_\varphi \cdot \mathrm{tokenize}(\boldsymbol{x})\right)$ computes a finite CoT, and

$$\mathrm{readout}\left(\mathrm{generate}_\Gamma\left(\boldsymbol{\pi}_\varphi \cdot \mathrm{tokenize}(\boldsymbol{x})\right)\right) = \varphi(\boldsymbol{x}).$$

  - **Remarks**:
    - Σ, Γ, tokenize, and readout are independent of the function φ;
    - The prompt $\boldsymbol{\pi}_\varphi$ is independent of the input $\boldsymbol{x}$;
    - tokenize & readout run in time $\mathrm{O}(|\boldsymbol{x}|)$ & $\mathrm{O}(|\varphi(\boldsymbol{x})|)$ on a RAM, respectively.
  - ❖ *The rest of this poster is the proof sketch of this theorem…*

TWO-TAPE POST–TURING MACHINES

- **Key idea**: Define a new **imperative** model of computation.
- **Setting**: two bi-infinite tapes A & B. Each tape has
  - Infinitely many cells over the binary alphabet $\{0,1\}$ and
  - A head pointing to a cell. Input is initially written to tape A.
- **Proposed model**: two-tape Post–Turing machines (2-PTMs).
  - A 2-PTM is defined by a finite instruction sequence $\boldsymbol{\iota} = \langle \iota_0, \ldots, \iota_{|\boldsymbol{\iota}|-1} \rangle$.
  - Each instruction ι_j $(0 \le j < |\boldsymbol{\iota}|)$ is one of the following: (below, $\tau \in \{\mathtt{A}, \mathtt{B}\}$)
    - `#`: halt;
    - τ`L`: move the head of tape τ one cell left, and go to ι_{j+1};
    - τ`R`: move the head of tape τ one cell right, and go to ι_{j+1};
    - τ`0`: write `0` to the pointed cell of tape τ, and go to ι_{j+1};
    - τ`1`: write `1` to the pointed cell of tape τ, and go to ι_{j+1};
    - τ`!`$_k$ $(k \ne j)$: if the pointed cell of tape τ is `0`, go to ι_k; else go to ι_{j+1};
    - τ`?`$_k$ $(k \ne j)$: if the pointed cell of tape τ is `1`, go to ι_k; else go to ι_{j+1}.
  - **Unary encoding** of go-to's: (below, $\sigma \in \{\tau\mathtt{!}, \tau\mathtt{?}\}_{\tau \in \{\mathtt{A}, \mathtt{B}\}}$; · is concatenation)
    - If $k < j$, encode σ_k as $\sigma \cdot \mathtt{-}^{j-k} \cdot \mathtt{@}$ (i.e., repeating token "–" $j-k$ times).
    - If $k > j$, encode σ_k as $\sigma \cdot \mathtt{+}^{k-j} \cdot \mathtt{@}$ (i.e., repeating token "+" $k-j$ times).

SIMULATION via CoT STEPS

- **Key idea**: Record the execution of 2-PTMs via CoT steps.
- **Go-to instruction** $\iota_j = \sigma_k$ $(\sigma \in \{\tau\mathtt{!}, \tau\mathtt{?}\}_{\tau \in \{\mathtt{A}, \mathtt{B}\}})$:
  - If the go-to condition is not met, put token `/` in the CoT.
  - Else, put $\mathtt{=} \cdot \mathtt{-}^{j-k} \cdot \mathtt{@}$ if $k < j$ or $\mathtt{=} \cdot \mathtt{+}^{j-k} \cdot \mathtt{@}$ if $k > j$.
- **Halting & outputting**: When execution reaches instruction #,
  - Put the output between two special tokens `:` and `$` in the CoT,
  - So that readout can extract it easily.
- **Input tokenization**:
  - **Key idea**: "Write" the input to tape A via CoT steps.
  - **Trick**: Encode the input $\boldsymbol{x}$ via **Shannon's** encoding.
- ❖ *Many details omitted here… See the example below.*
- **The constructed alphabet**:

$$\Sigma = \{\#, \mathtt{AL}, \mathtt{BL}, \mathtt{AR}, \mathtt{BR}, \mathtt{A0}, \mathtt{B0}, \mathtt{A1}, \mathtt{B1}, \mathtt{A!}, \mathtt{B!}, \mathtt{A?}, \mathtt{B?}, \mathtt{-}, \mathtt{+}, \mathtt{@}, \mathtt{\^{}}, \mathtt{\$}, \mathtt{/}, \mathtt{=}, \mathtt{:}, \mathtt{0}, \mathtt{1}\}.$$

- ❖ *The construction of the Transformer is also omitted. See our paper…*

DEMONSTRATION

- **Example**: Suppose φ decides the DYCK language.
  - DYCK: balanced parenthesis sequences; "`0`" as "(", "`1`" as ")".
- **Prompt** $\boldsymbol{\pi}_\varphi$ for deciding DYCK:

```
^A?+++++++++++++@A0ALA0ALA?----@ARARA1ARBLB?++@A1#ARA?++++@
B1BRB!+++@BLB!++++@B0ARB!----------------------@ALARARA?--@
A0ALA0ALA?----@ARARA1#$
```

- The **input** `00` has **Shannon's** encoding `1010`, tokenized as:
  - tokenize(`00`) = ARARARARALALA1ALALA1=----------@
- The generated **CoT steps** for computing φ(`00`):

```
=+++++++++++++@AR/B1BR=+++@B0AR=----------------------@
=+++++++++++++@AR/B1BR=+++@B0AR=----------------------@
/A0ALA0AL=----@A0ALA0AL=----@A0ALA0AL/ARARA1ARBL=++@:0$
```

  - The final readout answer is `:0$`.
  - It correctly computes φ(`00`) = `0` (`00` $\notin$ DYCK).

❖ *More examples on GitHub!*

SCAN TO LEARN MORE