CMPUT 653: Foundations of Reasoning in LLMs Lecture Notes

Siddhartha Chitrakar

October 7, 2025

Contents

Preface

Symbols and Remark on Notation

Insert Preface.

Acknowledgments

Insert Acknowledgments.

CMPUT 653: Foundations of Reasoning in LLMs **Fall 2025**

Lecture 1 Video, Lecture 2 Video

Lecturer: Csaba Szepesvári *Sep. 3 and Sep. 5 2025* *Scribe: Siddhartha Chitrakar*

Lecture 1 and 2: Modelling Sequences

Note: Lecture 1 contains errors due to reusing the notation p for different meanings. Lecture 2 corrected this; [*timestamp] used to link second lecture.

We use **probabilistic sequence models** to model sequences. Exercise 1.1, 1.2 intuitively explain why we use this, but first we introduce some notation:

- Σ - Alphabet Characters (set of all 1 length sequences)
  - English lowercase alphabet: $\Sigma = \{a, b, c, d, \ldots, y, z\}$
  - DNA alphabet $\Sigma = \{A, C, G, T\}$
  - LLM alphabet usually has at least 30,000 unique characters
- Concatenation - An operation (denoted by $\oplus$) that joins two sequences
  - $a \oplus b = ab$

Thus, we define n length sequences by concatenation:

- $\Sigma^2 := \{a_1 a_2 : a_1, a_2 \in \Sigma\}$, (set of all 2 length sequences)
  - If $\Sigma = \{a, b\}$ then, $\Sigma^2 = \{aa, ab, ba, bb\}$
- $\Sigma^n := \{a_1 a_2 \ldots a_n : a_1, a_2, \ldots a_n \in \Sigma\}$, (set of all n length sequences)
  - $w = (w_1 w_2 \ldots w_n) \in \Sigma^n$ (sequence of length n in order from w_1 to w_n)
- $\Sigma^0 := \emptyset := \{\perp\}$, ($\perp$ denotes the empty length sequence)

We can find the size (cardinality) of these n length sequences sets:

- $|\Sigma| = N$
- $|\Sigma^2| = N^2$
- $|\Sigma^n| = N^n$

Remark 1.1. Keep note that the number of sequences of length n grows exponential since $|\Sigma|^n = N^n$

Definition 1.1. We define the set of **all possible sequences** as

$$\Sigma^* := \bigcup_{n=0}^{\infty} \Sigma^n$$

Exercise 1.1. Let $\mathcal{L} \subseteq \Sigma^*$ be a language. We deterministically define a recognizer function r such that $r : \Sigma^* \to \{\text{yes}, \text{no}\}$. **Why is this problematic?**

Exercise 1.2. Consequently to Exercise 1.1, what are some advantages of using probabilistic sequence models?

1.1 Defining a probability distribution [32:24]

We define a probability distribution p over the set of all possible sequences Σ^*.

$$p : \Sigma^* \to [0, 1]$$

such that the following conditions are met:

1. $0 \leq p(w) \leq 1$ for all sequences $w \in \Sigma^*$.
2. $\sum_{w \in \Sigma^*} p(w) = 1$

With this distribution, we perform text completion tasks. Let $w \in \Sigma^*$ be a sequence, called **prefix/prompt**. Let $U \in \Sigma^*$ be a **random completion**.

We define a completed sequence W' as the concatenation of the prompt and the random completion:

$$W' := w \oplus U$$

We consider the probabilities for all possible completions, $\forall u \in \Sigma^*$:

$$P(W' = w \oplus u) = p(w \oplus u)$$

Example 1.1. Consider prefix $w =$ "The weather is...". The random completion, U, could be $u_1 =$"hot", $u_2 =$"tornado", u_3="biking". Based on what our model is trained on it would likely output decreasing probabilities respectively.

Remark 1.2. This is how LLMs works. When you give a prompt w, it samples from all possible completions u. However, Σ^* is countably infinte, so how do we learn a probability for every sequence?

1.2 Decomposition by Chain Rule [*00:29]

This section deals with the problem of Σ^* being countably infinite.

Definition 1.2 (Chain Rule)**.** Let $w = (w_1, w_2, \ldots, w_n)$ (sequence of length n. The joint probability of the sequence, $p(w)$, can be decomposed:

$$p(w) = p(w_1) \cdot p(w_2|w_1) \cdot p(w_3|w_1, w_2) \cdots p(w_n|w_1, \ldots, w_{n-1})$$

We will **exploit** what is the next character given what we have though Chain Rule. First, we ask how can a probabilistic model know when to stop?

Definition 1.3 (STOP Symbol)**.** A character $\langle STOP \rangle$ that signals the model when to stop.

Intuition: To define conditional probabilities we...

1. Start from the distribution of all possible sequences. (Section 1.1)
2. Now define conditional probability from this (Section 1.2.1)

Remark 1.3 (Bottom-up approach)**.** We could have started from the conditional distribution, then use chain rule to get the distribution of all sequences.

1.2.1 Defining the conditional distribution [*1:45]

Let $W \sim p$, be a random variable sample of a random string. Formally,

$$\forall w \in \Sigma^*, \mathbb{P}(W_n = w_n) = p(w)$$

Goal: Derive a conditional distribution of the next character from known info:

$$\mathbb{P}(W_n = w_n | W_{1:n-1} = w_{1:n-1})$$

Exercise 1.3. What is the problem about this "Goal"? Why would we need to extend $\langle STOP \rangle$ to W_n to infinite sequences?

Definition 1.4 (Extend Stop to Infinite Sequences)**.** Define $\hat{\Sigma} \coloneqq \Sigma \cup \langle STOP \rangle$. Additionally, pad the random variable, W, to an infinite sequence:

$$\hat{W} = W \langle STOP \rangle \langle STOP \rangle \langle STOP \rangle ...$$

Theorem 1.1 (Conditional Distribution of p)**.** Let $\hat{p} : \bigcup_{n\geq 1} \Sigma^{n-1} \times \hat{\Sigma} \to [0,1]$,

$$\begin{aligned}\hat{p}(w_n|w_{1:n-1}) &\coloneqq \mathbb{P}(\hat{W}_n = w_n|\hat{W}_{1:n-1} = w_{1:n-1}) \\ &= \begin{cases} \frac{\mathbb{P}(\hat{W}_{1:n}=w_{1:n})}{\mathbb{P}(\hat{W}_{1:n-1}=w_{1:n-1})} & \text{if } \mathbb{P}(\hat{W}_{1:n-1} = w_{1:n-1}) \neq 0 \\ p_0(w_n) & \text{if } \mathbb{P}(\hat{W}_{1:n-1} = w_{1:n-1}) = 0 \end{cases}\end{aligned}$$

by Chain Rule Definition 1.2 and let p_0 be any distribution.

Exercise 1.4. Prove Theorem 1.1 is a probability distribution

Remark 1.4. Theorem 1.1, $\tilde{p}$, suffices to model completion tasks like Definition 1.5 (Autoregressive models). We generate a character by character completion sequence until the model samples $\langle$STOP$\rangle$.

Definition 1.5 (**Autoregressive Model**)**.** Autoregressive models predict the next value in a sequence based on the values that came before it.

Lecture 1 and 2 Exercises Solutions (Tentative, will be updated)

Solution 1.1. This forces a yes/no decision, and languages are too complex to be deterministically decided like this. For example, language can be interpreted or explained in many different ways.

Solution 1.2. Compared to Exercise 1.1, this relaxes the deterministic clause and instead we ask how likely is a sequence part of a language. This allows us to navigate the complexities and nuances of languages better.

Solution 1.3. First, we look at **finite sequences**, $\Sigma^{n*} \coloneqq \bigcup_{i=0}^{n} \Sigma^i$.

Consider a conditional distribution on finite sequences, $\hat{p} : \Sigma^{n*} \times \Sigma \to [0, 1]$,

$$\hat{p}(w_1 w_2 \dots w_{n+1}) \coloneqq p(w_{n+1}|w) = \frac{p(w_1 w_2 \dots w_{n+1})}{p(w_1 w_2 \dots w_n)}$$

Claim: For any sequence w $\in \Sigma^n$, we must extend $\langle \text{STOP} \rangle$ so $|w| = n$.

Proof. Contradiction: Assume we don't extend $\langle \text{STOP} \rangle$. Start with $n = 2$, then, the outcomes are $\{\emptyset, a, b, ab, ba, aa, ba\}$. Suppose $p(b) = \frac{1}{3}$ and $p(ba) = \frac{2}{3}$, then $p(ba|b) = \frac{p(ba)}{p(b)} = \frac{2/3}{1/3} = 2$!? So what went wrong?

$p(b)$ is exactly b. But, it should be the prob. that the first letter seen is b.

Formally, we want $p(b)$ as $p(b) + p(ba) + p(bb)$. To do this, we extend $\langle \text{STOP} \rangle$ so $p(b) = p(b \oplus \langle \text{STOP} \rangle) + p(ba) + p(bb)$. If we want to know the prob. that the sequence is exactly b, it is now $p(b \oplus \langle \text{STOP} \rangle)$

We can inductively show that $\forall n \geq 2$, we must extend $\langle \text{STOP} \rangle$. $\square$

To end, Σ^* is countably infinite, thus we must extend $\langle \text{STOP} \rangle$ to infinite sequences. Importantly, this makes the conditional dependencies compatible!

Intuition explanation: Consider $w =$ "The weather is" and $w_{n+1} =$ "hot". Then, the sentence, "The weather is", is not valid and rarely seen.

$p(\text{"The weather is"}) < p(\text{"The weather is hot"}) \implies \tilde{p}(\text{"The weather is hot"}) > 1$. Clearly, without extending, the conditional dependencies is not compatible.

Solution 1.4 (Prove this and the compatibility yourself...)**.**

CMPUT 653: Foundations of Reasoning in LLMs **Fall 2025**

Lecture 2 Video [27:10], Lecture 3 Video

Lecturer: Csaba Szepesvári *Sep. 5 and Sep. 9 2025* *Scribe: Siddhartha Chitrakar*

Lecture 2 and 3: Transformer Architecture

Note: Lecture 3 recapped and continued the discussion of Transformer Architecture; [*timestamp] used to link Lecture 3.

Assumption: From now on and for convenience $\Sigma \coloneqq \hat{\Sigma} = \Sigma \cup \langle STOP \rangle$

Exercise 2.1. Read over this lecture: Provide the intuition and re-define the transformer architecture to the specific well-known euclidean space $\mathbb{R}^d$.

Transformers map strings to strings, and we generalize them to abstract vector spaces for alternatives to Euclidean space. Why are vector spaces important? Because transformers act in the vector space, $\mathcal{W}$.

Definition 2.1 (Workspace)**.** Let $\mathcal{W} \neq \emptyset$ be a **vector space** over scalars $\{0, 1\}$ called a **workspace**. The elements $\mathbf{x} \in \mathcal{W}$ are called vectors.

Recall vector spaces are equipped with two operations, addition $(+)$ and scalar multiplication $(\cdot)$, which for $\forall \mathbf{x}, \mathbf{y} \in \mathcal{W}$:

1. **Closure under Addition:** $\mathbf{x} + \mathbf{y} \in W$.
2. **Scalar Multiplication:** $1 \cdot \mathbf{x} = \mathbf{x}$ and $0 \cdot \mathbf{x} = \mathbf{0}$

Definition 2.2 (Embedding Map)**.** Let $e : \Sigma \to \mathcal{W}$ be an embedding map that maps each $x \in \Sigma$ to a vector representation $\mathbf{w} \in \mathcal{W}$.

$$e(x) = \mathbf{w}$$

Definition 2.3 (Unembedding Map)**.** Let $u : \mathcal{W} \to \Sigma$ be an unembedding map that maps each vector $\mathbf{w} \in \mathcal{W}$ back to a character $x \in \Sigma$

$$u(\mathbf{w}) = x$$

Remark 2.1. The embedding/unembedding maps aren't inverses. Sometimes they reverse the same info, but these are not bijections (different dimensions).

Before we define transformers, here is **notation** for component map extension

- If given $(a_1, a_2, \ldots, a_n) \in A^n$, then the component map extension is
  - $s^{[n]}((a_1, a_2, \ldots, a_n)) = (s(a_1), s(a_2), \ldots, s(a_n))$
  - $s(a_1, a_2, ..., a_n) \coloneqq s^{[n]}((a_1, a_2, \ldots, a_n))$ (Drop the [n] for brevity)

2.1 Intuition of Transformer Architecture

Intuition: [Talk by 3B1B] is a good source to visualize the intuition

Transformers, f, maps an input $\vec{w} \in \Sigma^*$, to an output $\vec{w} \in \Sigma^*$ by refining the representation of $\vec{w}$. Goal: "Enrich" the vector with information. The "enriching" process is a sequential composition of functions:

$$f = u^{[n]} \circ f_L^{(n)} \circ \cdots \circ f_1^{(n)} \circ e^{[n]}$$

Step 1. Embedding ($e^{[n]}$)

First, $e^{[n]} : \Sigma^n \to \mathcal{W}^n$, embeds a input sequence $\vec{w} \coloneqq (w_1, ..., w_n) \in \Sigma^*$ component wise to a vector sequence $(e(w_1), ..., e(w_n)) = (\mathbf{x_1}, ..., \mathbf{x_n}) \in \mathcal{W}^n$.

Step 2. Transformer Layers ($f_\ell^{(n)}$)

The vector sequence $(\mathbf{x_1}, ..., \mathbf{x_n})$ is then processed through finitely many L layers. Each layer, $f_\ell^{(n)}$, refines every vector by two sub-steps:

Step 2.1 Attention ($A_\ell^{(n)}$): Updates vectors $\mathbf{x}_i$ by aggregating "useful" info from the entire sequence. How do you determine the "useful" info:

Query (q_ℓ): Asks current vector $\mathbf{x}_i$, "What info do I need?"

Key (k_ℓ): Answers from other vector $\mathbf{x}_j$, "This is the info I represent."

Attention Score (a_ℓ): Compares Query ($\mathbf{x}_i$), with Key ($\mathbf{x}_j$) for relevance.

Value (v_ℓ): The actual info of vector $\mathbf{x}_j$ if deemed relevant (key/query matches)

Step 2.2 Multilayer perceptron ($m_\ell^{[n]}$): The multilayer perceptron $m_\ell^{[n]}$ performs a non-linear transformation on each vector independently.

Step 3. Unembedding ($u^{[n]}$)

After L layers, the "enriched" vectors is unembeded by $u^{[n]} : \mathcal{W}^n \to \Sigma^n$. This maps the last vectors to the vocab space Σ to produce an output sequence.

2.2 Definition of Transformer Architecture [37:39], [*6:13]

Definition 2.4 (Transformer Map)**.** $f : \Sigma^* \to \Sigma^*$ is a regular **transformer map** over the sets of functions $\mathcal{M}, \mathcal{A}, \mathcal{Q}, \mathcal{K}, \mathcal{V}$ if $\forall n \geq 1$,

$$f|_{\Sigma^n} = u^{[n]} \circ f_L^{(n)} \circ \cdots \circ f_1^{(n)} \circ e^{[n]}$$

such that for each layer $\ell \in \{1, \ldots, L\}$:

1. The layer function $f_\ell^{(n)}$ is a composition $f_\ell^{(n)} = m_\ell^{[n]} \circ A_\ell^{(n)}$, s.t $m_\ell^{[n]} \in \mathcal{M}$.
   - The compositions are a multilayer perceptron and attention map.
2. The attention map $A_\ell^{(n)} : \mathcal{W}^n \to \mathcal{W}^n$ is defined $\forall \mathbf{x} \in \mathcal{W}^n$,

$$(A_\ell^{(n)}(\vec{x}))_i = \mathbf{x}_i + \sum_{j=1}^{n} a_\ell(q_\ell(\mathbf{x}_i), k_\ell(\mathbf{x}_j))v_\ell(\mathbf{x}_j)$$

   where $a_\ell \in \mathcal{A}$, $q_\ell \in \mathcal{Q}$, $k_\ell \in \mathcal{K}$, and $v_\ell \in \mathcal{V}$.

   2.1. Function $a_\ell \in \mathcal{A}$ is the attention pattern which determines whether information from vector j (via $v_\ell(\mathbf{x}_j)$) is passed to vector i:

$$a_\ell : \mathcal{W} \times \mathcal{W} \to \{0, 1\}$$

   2.2. In practice, the summation is **normalized**

$$\frac{\sum_{j=1}^{n} a_\ell(q_\ell(\mathbf{x}_i), k_\ell(\mathbf{x}_j))v_\ell(\mathbf{x}_j)}{\sum_{j=1}^{n} a_\ell(q_\ell(\mathbf{x}_i), k_\ell(\mathbf{x}_k))}$$

   2.3 However, we need a **probabilistic unembedding** to predict the next sequence defined in Theorem 1.1. The unembedding (u) maps the final vector representation, $\mathbf{x}_i \in \mathcal{W}$, to a probability distribution $\pi_i := \frac{\exp(\mathbf{w}_j^T \mathbf{z}_i)}{\sum_{j \in \Sigma} \exp(\mathbf{w}_g jT \mathbf{z}_i)}$ over the alphabet Σ. [*19:44]

Note: Summation range can be modified, $j < i$ for **causal attention** or $j \neq i$
.

2.2.1 Discussion About Transformer Architecture [*25:43]

Note: The discussion is to introduce different aspects about transformers.

Sparse Attention [*25:43]: The attention normalization in Definition 2.4 creates a global dependency that's recalculated for every update, leading to $O(n^2)$ complexity and poor caching. This motivates **sparse attention** where each vector only attends to a limited subset of vectors.

Positional Embeddings [*29:23]: The attention summation is permutation-invariant. However, in langauge, word order absolutely matters!

- Initial solution adds a position vector $\mathbf{p}_1, ...\mathbf{p_n} \in \mathcal{W}$ to each input $\mathbf{x}_i$. The query and key become $q_\ell(\mathbf{x}_i + \mathbf{p}_i)$ and $k_\ell(\mathbf{x}_j + \mathbf{p}_j)$. This struggles with infinite sequences $\mathbf{p}_1, \mathbf{p}_2, ...$; thus, we now use rotational embeddings
- **Rotational Positional Embeddings [*1:13:17]:** Positional dependent matrix, R_i acting on queries and keys.

  $$\mathbf{q}'_i = R_i q_\ell(\mathbf{x}_i) \quad \text{and} \quad \mathbf{k}'_j = R_j k_\ell(\mathbf{x}_j)$$

  The attention score depends on the relative distance between vectors, due to the rotation matrix property ($R_i^T R_j = R_{i-j}$):

  $$\langle \mathbf{q}'_i, \mathbf{k}'_j \rangle = (R_i q_\ell(\mathbf{x}_i))^T (R_j k_\ell(\mathbf{x}_j)) = q_\ell(\mathbf{x}_i)^T R_{i-j} k_\ell(\mathbf{x}_j)$$

  This allows the model to generalize far better to unseen sequence lengths!

Multi-headed Attention [*41:30]: This addresses different types of information simultaneously by running attention patterns in parallel.

Mixtures of Experts [*1:06:50]: Analogous to multi-headed attention where we have a set of k parallel "expert" MLPs, denoted as $m_i(x)$, and a trainable gating network p. For each input, the gating network selects a subset of the experts.

- **Expert Networks (m_i):** A collection of k parallel MLPs.
- **Gating Network (p):** A trainable network that outputs a probability distribution over the experts. The weight for the i-th expert is $p_i(x)$.
- For sequence x, y is the weighted sum of the outputs from all experts.

$$y = \sum_{i=1}^{k} p_i(x) \cdot m_i(x)$$

- Load balancing prevents the gating network from using the same few favorite experts. For a fixed context sequence $x_1, ..., x_T$, the cumulative weight to each expert is roughly equal:

$$\sum_{t=1}^{T} p_i(x_t) \approx \frac{T}{k}$$

Other Discussions:

- Decoding Strategies [*50:55]
- Complexity of Matrices [*1:01:22]
- Precision [*1:03:53]
- Layered Norm [**17:31] From Lec. 4, Lec. 3 explanation is incorrect

Lecture 2 and 3 Exercises Solutions (Tentative, will be updated)

Solution 2.1. Why in practice do we use $\mathbb{R}^d$? So we can encode the sequences to vectors and provide some meaning to it with numbers.

Redefining Transformers Now the abstract workspace $\mathcal{W} = \mathbb{R}^d$

Embedding Map: $e^{[n]} : \Sigma^n \to \mathbb{R}^d$ encodes sequences to vectors

Unembedding Map: $u^{[n]} : \mathbb{R}^d \to \Sigma^n$ encodes vectors back to sequences.

- In practice, there is a **softmax function** to get a probability distribution.

Query, Key, Value (q_ℓ, k_ℓ, v_ℓ These are linear transformations meaning matrix multiplication! $q_\ell(\mathbf{x}_i) = W_Q\mathbf{x}_i$, $k_\ell(\mathbf{x}_j) = W_K\mathbf{x}_j$, and $v_\ell(\mathbf{x}_j) = W_V\mathbf{x}_j$, where W_Q, W_K, W_V are learned weight matrices.

Attention Score (a_ℓ**)**: scaled dot-product to measure similarity by direction

$$a_\ell(q, k) = \text{softmax}\left(\frac{q^T k}{\sqrt{d_k}}\right)$$

Measures similarity between the query and key vectors

CMPUT 653: Foundations of Reasoning in LLMs **Fall 2025**

Lecture 4 Video

Lecturer: Csaba Szepesvári *Sep. 11, 2025* *Scribe: Siddhartha Chitrakar*

Lecture 4: Thinking Like Transformers

Note: This section explores the capabilities of transformers, specifically the class of functions they can compute.

4.1 RASP: Restricted Access Sequence Processing [5:21]

From "Thinking like Transformers" by Weiss et al. [1], RASP language allows us to think the types of functions transformers can compute by defining primitive functions analogous to a transformer. RASP-L is a variant of this with more restrictions like limiting functions to 8-bit integers.

In RASP we ignore the vector space structure by instead working with a sequence of integers. A function f is defined as $f : \mathbb{Z}(\text{int-8}) \to \mathbb{Z}(\text{int-8})$, on a sequence $S \coloneqq [s_0, s_1, \ldots, s_{n-1}]$ where each $s_i \in \mathbb{Z}$.

RASP Primitive Functions: RASP uses the following primitive functions:

1. **Token-wise Map**: Applies function f to each element of a sequence.

$$\texttt{tokmap}(S, f) \implies [f(S[i]) \text{ for } i \in \{0, \ldots, n-1\}]$$

2. **Sequence-wise Map**: Applies function f to two sequences, S_1 and S_2.

$$\texttt{seqmap}(S_1, S_2, f) \implies [f(S_1[i], S_2[i]) \text{ for } i \in \{0, \ldots, n-1\}]$$

3. **Attention Map**: Models the attention mechanism.

$$\texttt{kqv}(Q, K, V, \texttt{pred}, \texttt{agg}) :$$

   (a) **Attention Matrix Creation**: $\forall (i, j)$, the predicate function creates a boolean attention matrix A from the *ith* query and *jth* key.

$$\forall (i, j), A[i, j] = \texttt{pred}(Q[i], K[j])$$

   (b) **Value Selection**: $\forall i$, a list of values $V[i]$ is formed from the values

$V[j]$ where the attention matrix entry is true.

$$\forall i, V[i] = [V[j] \text{ for j in [n] if } A[i,j] = \text{True}]$$

(c) **Aggregation**: Applied to $V[i]$ to produce the final output. The common aggregation functions are `mean`, `max`, or `min`.

$$\texttt{out}[i] = \texttt{agg}(V[i])$$

The function then returns the output sequence `out`.

4. **Indices**: Returns the indices sequence of integers from 0 to $n-1$

$$\texttt{indices}() \implies [0, 1, \ldots, n-1]$$

4.2 RASP Exercises

Exercise 4.1. [32:10] Using the RASP primitives, write the function:

$$\texttt{replace}(S, v_{in}, v_{out})$$

where S is a sequence of integers, and v_{in}, v_{out} are integers. The function should return a sequence such that the output is:

$$[v_{out} \texttt{ if } S[i] == v_{in} \texttt{ , else } S[i] \texttt{ for i in [n]}]$$

Exercise 4.2. [54:08] Now, what if we don't have access to v_{in}, v_{out}. Using the RASP primitives, write the function:

$$\texttt{replace}(S)$$

where S is a sequence of integers. For this function, let v_{in}, be the first element ($S[0]$), and v_{out}, be the second element ($S[1]$).

The function should return a sequence such that the output is:

$$[S[1] \texttt{ if } S[i] == S[0]\texttt{, else } S[i] \texttt{ for i in [n]}]$$

Exercise 4.3. [1:05:26] Can you reverse a string in-place with causal attention and/or without? If you can't reverse, is there a trick to not do it in-place?

CMPUT 653: Foundations of Reasoning in LLMs **Fall 2025**

Lecture 5 Video

Lecturer: Vlad Tkachuk *Sep. 16, 2025* *Scribe: Siddhartha Chitrakar*

Lecture 5: Transformer Representational Power

Note: Last lecture gave an intuition on what functions can transformers compute; this lecture formally studies what functions transformers can represent

Example 5.1 (Parity Function, XOR)**.** Consider the **parity function** as $P : \{0,1\}^n \to \{0,1\}$ such that $P(x) = (\sum_{i=1}^n x_i) \pmod 2$.

Let function class $\mathcal{N}_n(s(n)) \coloneqq \{f : \{0,1\}^n \to \{0,1\} \mid \exists \theta \text{ s.t. } NN_\theta = f\}$ with:

- A single hidden layer of size $O(s(n))$.
- Sign activation functions.

$$\text{If } s(n) \in o(2^n), \text{ then Parity} \notin \mathcal{N}_n(s(n))$$

This example motivates an expressive power limitations. They can't represent a parity function, which motivates what functions can Transformers represent. How does changing architecture features like depth or width change things...

Remark 5.1. Recall Transformer Definition 2.4. We assume:

- $\mathcal{W} = \mathbb{R}^d$ (i.e Exercise 2.1)
- $e(w) = \underbrace{\theta_{TE} w}_{\text{Token Embedding}} + \underbrace{\theta_{PE}}_{\text{Positional Embedding}}$
- $m_\ell(\mathbf{z}) = \mathbf{z} + W_2^{(\ell)} \mathrm{ReLU}(W_1^{(\ell)} \mathbf{z} + \mathbf{b}_1^{(\ell)}) + \mathbf{b}_2^{(\ell)}$
  - where $\theta_{MLP}^{(\ell)} = (W_1^{(\ell)}, W_2^{(\ell)}, \mathbf{b}_1^{(\ell)}, \mathbf{b}_2^{(\ell)})$.
- $(A_\ell^{(n)}(\mathbf{x}))_i = \mathbf{x}_i + \sum_{j=1}^i a_\ell(W_Q^{(\ell)} \mathbf{x}_i, W_K^{(\ell)} \mathbf{x}_j) W_V^{(\ell)} \mathbf{x}_j$
  - where $\theta_{ATTN}^{(\ell)} = (W_Q^{(\ell)}, W_K^{(\ell)}, W_V^{(\ell)})$.
  - Denote the sum up to i as causal and $i-1$ as strict causal attention.
- $u(\mathbf{z}) = \underset{j \in \Sigma}{\mathrm{argmax}}\, (\theta_U \mathbf{z})_j$

Definition 5.1 (Transformer Parameter Set)**.** A Transformer in Remark 5.1 is parameterized by a set of learnable parameters, denoted by Θ. [33:24]

$$\Theta = \left(\theta_{TE}, \theta_{PE}, \left(\theta^{(\ell)}_{MLP}, \theta^{(\ell)}_{ATTN}\right)_{\ell \in [L]}, \theta_U\right)$$

where:

- $\theta_{TE} \in \mathbb{R}^{d \times |\Sigma|}$ is the token embedding matrix.
- $\theta_{PE} \in \mathbb{R}^{n \times d}$ is positional embedding parameters.
- $\left(\theta^{(\ell)}_{MLP}, \theta^{(\ell)}_{ATTN}\right)_{\ell \in [L]}$ is the MLP and attention parameters for $\ell \in \{1, 2, \ldots, L\}$ layers such that each matrix is $\mathbb{R}^{d \times d}$
- $\theta_U \in \mathbb{R}^{|\Sigma| \times d}$ is the parameter matrix for the final unembedding layer.

What Are Some Architectural Differences Can We Have? [36:00]

- **Depth (L):** The number of layers (constant) in the Transformer
- **Width (d):** The embedding size, $d \log(n)$
- **Precision:** The numerical precision as practical implementations are limited by finite-precision arithmetic (e.g., float, int).
- **Layer Norm:** Layer norms vary affecting stability and performance.
- **Multi-head Attention:** This doesn't vary too drastically as you can simulate multi-head attention with single.

Definition 5.2 (Transformer Function Class)**.** Let width $d(n)$ and precision $p(n)$ be functions of the input size n. We define the class of functions computable by a Transformer, $\mathcal{T}(d(n), p(n))$, as follows:

$$\mathcal{T}(d(n), p(n)) = \left\{ f : \Sigma^* \to \Sigma^* \mid \forall n \in \mathbb{N}, \exists \theta \text{ s.t. } TF_\theta|_{\bigcup_{m=1}^{n} \Sigma^m} = f|_{\bigcup_{m=1}^{n} \Sigma^m} \right\}$$

where the Transformer TF_θ is constrained by:

- Width: The embedding dimension is $\mathcal{O}(d(n))$.
- Precision: The parameters θ are $\mathcal{O}(p(n))$.

Now, what functions should we care about? [49:49]

We consider "reasoning problems". Problems with steps like a graph problem that output 1 if there is a path and 0 otherwise. We redefine our class:

Definition 5.3 (Transformer Function Class for "Reasoning Problems")**.**

$$\mathcal{T}(d(n), p(n)) = \left\{ f : \mathbf{\{0,1\}}^* \to \mathbf{\{0,1\}}^* \mid \forall n \in \mathbb{N}, \exists \theta \text{ s.t. } \; TF_\theta|_{\bigcup_{m=1}^{n} \Sigma^m} = f|_{\bigcup_{m=1}^{n} \Sigma^m} \right\}$$

The lecture then **discusses** the **intuition** and **overview** of what next:

Turing Machine (TM): [56:00] We can define function classes to how much resources we give to a TM (finite state machine with unbounded memory).

- Resources are time [58:08] (class below) and space (not discussed)
- $DTime(t(n)) = \{f : \{0,1\}^* \to \{0,1\} \mid \exists$ a TM that computes f in $O(t(n))$ steps$\}$
- Note: the class "P" often in P/NP formulations is $P = \bigcup_{i \geq 1} DTime(n^i)$

The **main takeaway of this lecture** **[1:11:37]** is what can chain-of-thought function class, $COT(t(n), d(n), p(n)$ improve? Formally:

$$DTime(t(n)) \subseteq COT(t(n), 1, \log(n)) \subseteq DTime(t^2(n) + n^2)$$

- Notice $\bigcup_{i \geq 0} COT(n^i, \dots) = P$
- **If** $t(n)$ **is unbounded then** $COT(t(n), \dots)$ **is Turing Complete**
- [1:16:04] Briefly mentions current/future work like loop transformers [2]

CMPUT 653: Foundations of Reasoning in LLMs **Fall 2025**

Lecture 6 Video

Lecturer: Dale Schuurmans *Sep. 18, 2025* *Scribe: Siddhartha Chitrakar*

Lecture 6: Autoregressive LLMs Are Computationally Universal

Note: This lecture discussed the paper shown below [3]; thus, I will link the sections talked about the paper but read the paper for more info. [Paper Link]

Autoregressive Large Language Models
are Computationally Universal

Dale Schuurmans[1,2], Hanjun Dai[1], and Francesco Zanini[2]

[1]Google DeepMind
[2]University of Alberta

Abstract

We show that autoregressive decoding of a transformer-based language model can realize universal computation, without external intervention or modification of the model's weights. Establishing this result requires understanding how a language model can process arbitrarily long inputs using a bounded context. For this purpose, we consider a generalization of autoregressive decoding where, given a long input, emitted tokens are appended to the end of the sequence as the context window advances. We first show that the resulting system corresponds to a classical model of computation, a *Lag system*, that has long been known to be computationally universal. By leveraging a new proof, we show that a universal Turing machine can be simulated by a Lag system with 2027 production rules. We then investigate whether an existing large language model can simulate the behaviour of such a universal Lag system. We give an affirmative answer by showing that a single system-prompt can be developed for `gemini-1.5-pro-001` that drives the model, under deterministic (greedy) decoding, to correctly apply each of the 2027 production rules. We conclude that, by the Church-Turing thesis, prompted `gemini-1.5-pro-001` with extended autoregressive (greedy) decoding is a general purpose computer.

Figure 6.0.1: The paper this lecture is based on [3]. Read for more info.

- Introduction [5:28]
  - Recall Theorem 1.1, Chain Rule of Probability [6:19]
  - The bounded context window defines a K-Markov model [8:27]
  - Challenge: Figure how to operate a K-Markov model on an input string larger than K
- Extend Chain Rule of Prob. to Extended Autoregressive Decoding [10:10]
  - Idea: Extension generates and append two symbols instead of one
- Introduction and Examples of LAG Systems [21:11]

Definition 6.1 (Lag System)**.** A Lag system consists of a finite set of rules $x_1...x_N \to y$, where N is the length of the context, $x_1...x_N \in \Sigma^N$ denotes a sequence of symbols to be matched, and $y \in \Sigma^*$ is a corresponding output [3].

For a deterministic Lag system, each pattern $x_1...x_N$ is unique, hence the Lag system defines a partial function $L : \Sigma^N \to \Sigma^*$ that maps a pattern $x_1...x_N$ to a corresponding output y [3].

Lag system computation is defined by operating over a memory string, where in each iteration, a rule is matched to the prefix of the memory string, then the result appended to the string, after which the first symbol is deleted [3].

For interesting behavior, we have at least a rule that appends more than 1 symbol. We also introduce a Halt symbol H to terminate whenever generated. Computation also terminates when there is no rule for the current context, which includes when the current memory string is shorter than N.

- Lag System Operates Memory as a Queue Machine [29:45]
  - Thus, we can rotate the memory even if the memory is bigger than the context size!
  - Challenge: The queue machine has no external controller
  - Challenge: The queue machine can only operate in one direction
  - **To simulate a TM, we need to track the current machine state and be able to rotate the memory in both directions!**
- Lag System can track the current state and rotate both directions [37:11]
  - Explanation and Simulations [42:55 - 55:10]
- Lag Systems simulates a Turing Machine w/ a cubic time overhead [55:10]
  - Examples and Simulations [1:03:51]
- Autoregressive LLM can simulate a Universal LAG System [1:10:07]
  - Brute force: Entire set of rules as a system prompt to LLM [1:12:34]
  - Note: Gemini 1.5 Pro was used due to the large context window.
- Questions/Current/Future Work [1:15:38]
  - Proof is now done for random LLMs

Appendix A: Extra Proofs and Results

More appendix content here.

Appendix B: Supplementary Figures

More appendix content here.

References

[1] G. Weiss, Y. Goldberg, and E. Yahav, "Thinking like transformers," in *International Conference on Machine Learning*, PMLR, 2021, pp. 11 080–11 090.

[2] W. Merrill and A. Sabharwal, "Exact expressive power of transformers with padding," *arXiv preprint arXiv:2505.18948*, 2025.

[3] D. Schuurmans, H. Dai, and F. Zanini, "Autoregressive large language models are computationally universal," *arXiv preprint arXiv:2410.03170*, 2024.