NLU: Lecture 14
LLMs as Formal Machines

Shay Cohen
(based on slides from Butoi et al.)

February 12, 2025

School of informatics

Why formal study of LLMs?

- Opening the black box of LLMs: what are the underlying computations they do?
- We can better understand the limitations of LLMs
- Find connections to previous models may help deriving new methodology for LLMs

What are language models?

- Let $y = y_1 \cdots y_n$
- An autoregressive language model defines $p(y)$

$$p(y) = p(\text{EOS} \mid y) \prod_{t=1}^{n} p(y_t \mid y_{<t})$$

Nothing new! Any distribution over strings can be written that way

Food for thought: Why is that the case?

What are language models?

- Let $y = y_1 \cdots y_n$
- An autoregressive language model defines $p(y)$

$$p(y) = p(\text{EOS} \mid y) \prod_{t=1}^{n} p(y_t \mid y_{<t})$$

Nothing new! Any distribution over strings can be written that way

Food for thought: Why is that the case? The chain rule

Representation-based language model

Remember the conditional probability model over words given a representation, which is roughly:

$$p(y_t \mid y_1 \cdots y_{t-1}) = \text{softmax}(Eh(y_1 \cdots y_{t-1}))_{y_t}$$

Recurrent Neural Networks

We are given a sequence of $x_1, \ldots x_n$.

The hidden state:

$$h_{t+1} = f(h_t, x_t)$$

f is a function that maps the current state and symbol into a new state, so $f : \mathbb{R}^d \times \Sigma \rightarrow \mathbb{R}^d$

Elman RNNs

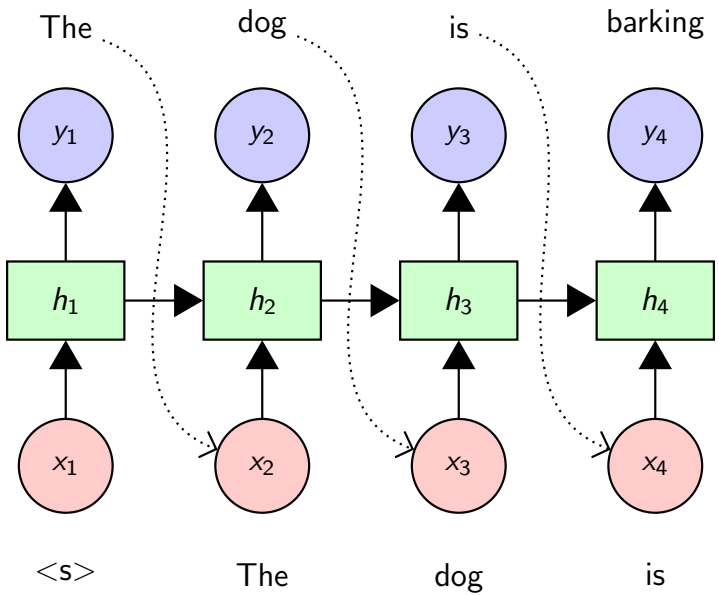
We will focus on so-called Elman RNNs, an older model that has been studied:

$$h_{t+1} = f(h_t, x_t) = \sigma(Uh_t + Vr(x_t) + b)$$

where r is an embedding function, U and V are matrices and b is a vector

RNNs as language models

If we want to turn an RNN into a language model, we add another layer that takes as input h_t and returns a probability distribution over the vocabulary



Every y_t is fed as an x_{t+1} in the next step

RNNs as language models

Food for thought: What is the connection to n-grams?

RNNs as language models

Food for thought: What is the connection to n-grams?

Both approaches provide a probability distribution over sentences

> *We multiply the probability of the n-grams together,*
> *OR*
> *We multiply the probability of the words emitted by the RNN,*

to get the probability over the string $p(y_1 \cdots y_n)$.

Food for thought: Why is it important that the mapping f is deterministic?

RNNs as language models

Food for thought: What is the connection to n-grams?

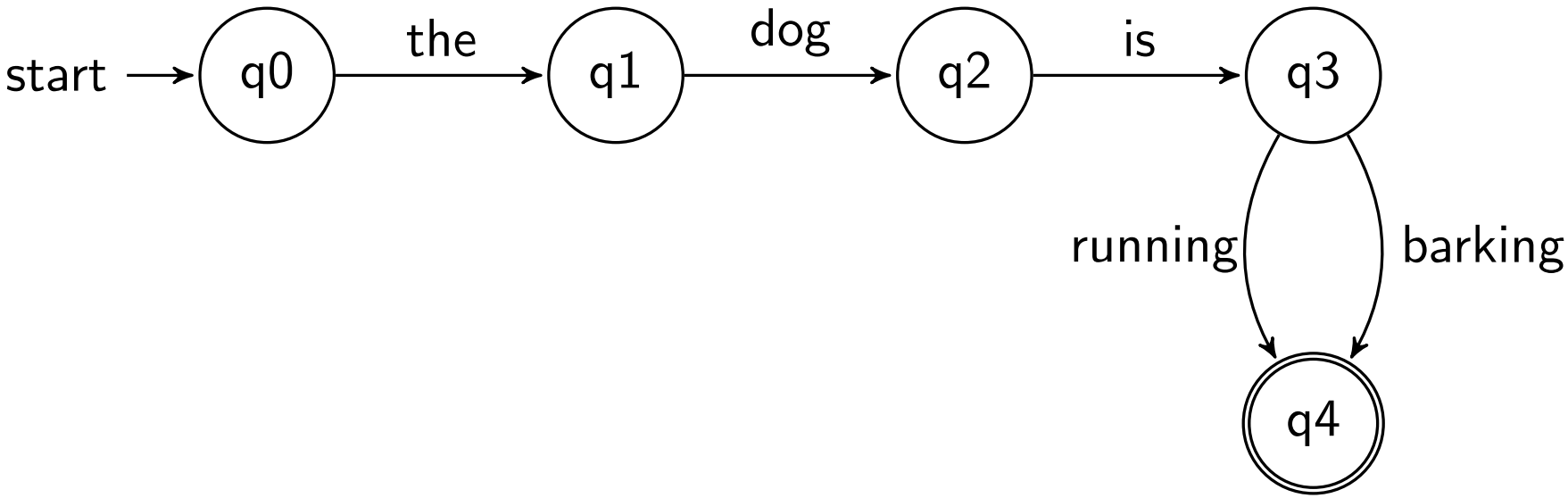
Both approaches provide a probability distribution over sentences

> *We multiply the probability of the n-grams together,*
> *OR*
> *We multiply the probability of the words emitted by the RNN,*

to get the probability over the string $p(y_1 \cdots y_n)$.

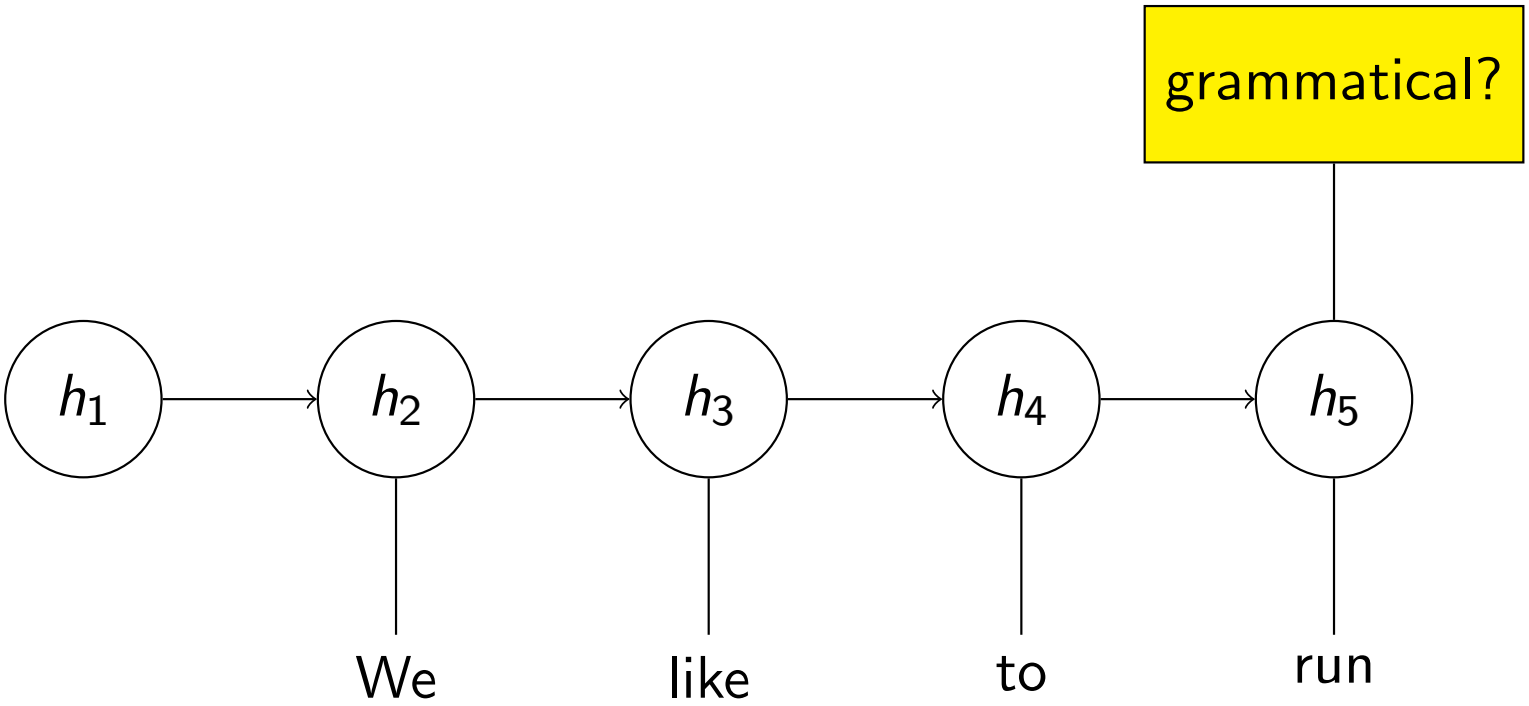
Food for thought: Why is it important that the mapping f is deterministic? We do not need to "sum out" h!

Reminder: Finite State Machines



A set of states Q with q_0 the initial state, an alphabet Σ, a transition function $\delta(q, y) = q'$ and a set of final states $F \subseteq Q$.

RNNs as Recognisers



- Apply a softmax or some other activation unit to binarise the state after encoding into *grammatical* or *not grammatical*
- Given an RNN, its language is defined as the set of all strings it classifies as "grammatical"

RNNs Have States Like Automata

- RNNs have many similarities to automata
- At each step, consume an input symbol, and calculate a new state
- Rather than using a look-up table like transition, use a mathematical formula to calculate a state *vector*

Food for thought: Does that change anything?

RNNs Have States Like Automata

- RNNs have many similarities to automata
- At each step, consume an input symbol, and calculate a new state
- Rather than using a look-up table like transition, use a mathematical formula to calculate a state *vector*

Food for thought: Does that change anything?

Answer: It depends!

Characterising a formalism through its languages

Let $\mathcal{F}$ be a "formalism", whether all RNNs as acceptors or FSMs

Each specific instance in $A \in \mathcal{F}$ is a single RNN or an FSM

It defines a language $L(A)$. We can characterise the formalism by the set $L(\mathcal{F}) = \{L(A) \mid A \in \mathcal{F}\}$.

Two formalisms $\mathcal{F}, \mathcal{F}'$ are equivalent if $L(\mathcal{F}) = L(\mathcal{F}')$.

Do you remember? Which languages do FSMs accept?

Characterising a formalism through its languages

Let $\mathcal{F}$ be a "formalism", whether all RNNs as acceptors or FSMs

Each specific instance in $A \in \mathcal{F}$ is a single RNN or an FSM

It defines a language $L(A)$. We can characterise the formalism by the set $L(\mathcal{F}) = \{L(A) \mid A \in \mathcal{F}\}$.

Two formalisms $\mathcal{F}, \mathcal{F}'$ are equivalent if $L(\mathcal{F}) = L(\mathcal{F}')$.

Do you remember? Which languages do FSMs accept?
Regular languages
Indeed, the regular languages fully characterise FSMs.
$L(\text{FSM}) = \textit{regular languages}$

State Space in RNNs

- In principle, the state space for RNNs is infinite, not only infinite, but continuous

- Can we use this fact somehow to simulate really complex algorithms?

We will test RNNs in two cases: the case of unbounded precision and computation and the case of bounded computation and finite precision

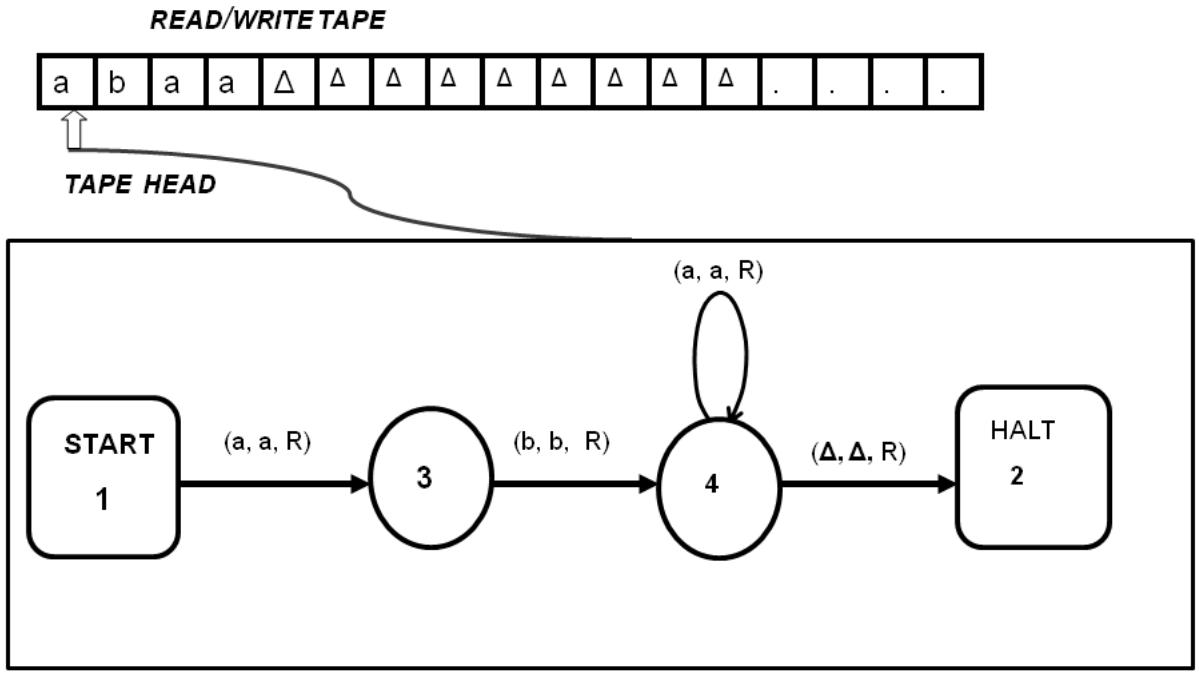
Food for thought: What do you think the outcome will be?

RNNs Power

Depending on the amount of power we provide to the latent states of RNNs, they can either:

- Implement any computable function
- Be significantly limited to finite state languages

Turing Machines



A Turing Machine for aba*

Church-Turing thesis: any reasonably computable function can be implemented using a Turing machine!

Two-stack Pushdown Automaton

An equivalent formalism to Turing machines

Rather than having an infinite tape, we have two stacks. A two-stack PD automaton has:

- Q - a finite set of states
- Σ - a finite alphabet
- Γ - a finite stack alphabet (for both stacks)
- δ - a transition function in $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \Gamma \rightarrow Q \times \Gamma^* \times \Gamma^*$.
  The transition outputs a new state and strings to replace the top of the two stacks based on the current input symbol and the top symbols of the two stacks
- $q_0 \in Q$ - an initial state
- $q_A \in Q$ - an accepting state
- $q_R \in Q$ - a reject state

RNNs Power

Depending on the amount of power we provide to the latent states of RNNs, they can either:

- Implement any computable function

  We will start here and show how a 2-PDA can be **simulated** using an RNN. This means we need to bridge the continuous approach of RNNs with the discrete manner PDAs work

- Be significantly limited to finite state languages

RNNs Power

Depending on the amount of power we provide to the latent states of RNNs, they can either:

- Implement any computable function

  We will start here and show how a 2-PDA can be **simulated** using an RNN. This means we need to bridge the continuous approach of RNNs with the discrete manner PDAs work

- Be significantly limited to finite state languages

Class exercise: Spend a few minutes to try and construct this simulation

Encoding a Stack

- We will assume that our alphabet is $\Gamma = \{0, 1\}$ in the stack $y_1 \cdots y_n$. (This is enough, why?)

Encoding a Stack

- We will assume that our alphabet is $\Gamma = \{0, 1\}$ in the stack $y_1 \cdots y_n$. (This is enough, why?)
- A stack can be represented as a number: $0.\gamma_N \gamma_{N-1} \ldots \gamma_1$ where $\gamma_i = \begin{cases} 1 & \text{if } y_i = 0, \\ 3 & \text{otherwise.} \end{cases}$
- We need operations on the stack: mostly pushing and popping

Food for thought: How do we simulate pushing and popping on a number?

Performing Stack Operations

Stack operations can be performed by manipulating a real number that represents the whole stack:

- Popping $y_N = 0 : 10 \times 0.\gamma_N\gamma_{N-1}\ldots\gamma_1 - 1$
- Popping $y_N = 1 : 10 \times 0.\gamma_N\gamma_{N-1}\ldots\gamma_1 - 3$
- Pushing $y = 0 : \frac{1}{10} \times 0.\gamma_N\gamma_{N-1}\ldots\gamma_1 + \frac{1}{10}$
- Pushing $y = 1 : \frac{1}{10} \times 0.\gamma_N\gamma_{N-1}\ldots\gamma_1 + \frac{3}{10}$

Note: Automata transition between configurations

A configuration is a "state of the world" for the two-stack automaton

It includes:

- The current state of the two stacks
- The current state the machine is in

The two-stack automaton can be thought of as transitioning between configurations based on the input!

Configurations

Given a 2-stack PDA, and an input x, there is a sequence of configurations $c_1, \ldots, c_n$ that we transition to

$$c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_n$$

If we know how to move between configurations, we can simulate any machine

We will show how RNNs can move between configurations, represented by their latent state

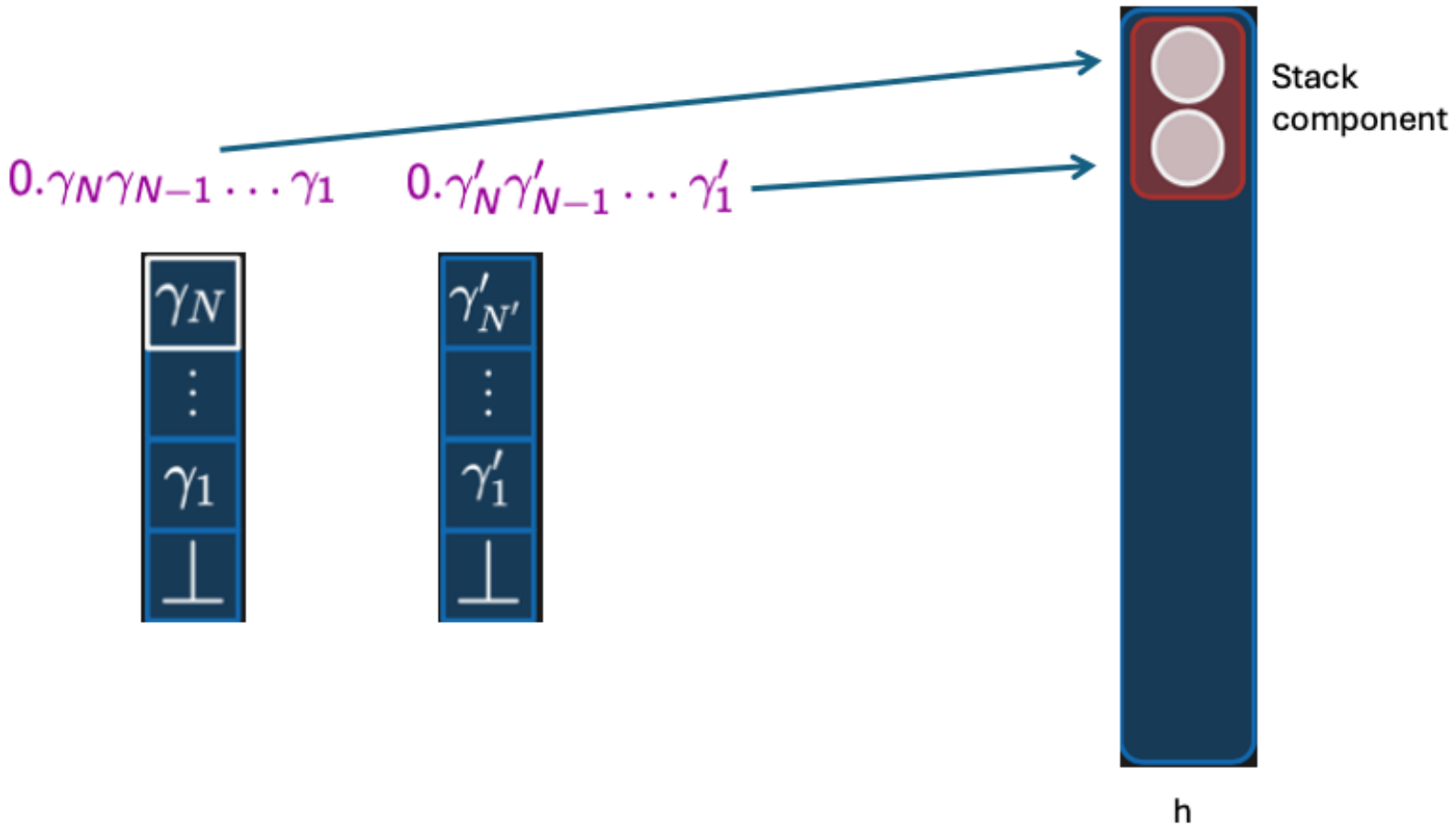
How to encode a "configuration" of a 2-stack PDA machine?

- A vector in the RNN will be in $\mathbb{R}^d$ where $d = 2 + 3 + 3 + N$ where N is the number of states in the Turing machine

- How to encode a configuration of a Turing machine using a vector in the RNN h?
  - h_1 is the first stack encoded as before
  - h_2 is the second stack encoded as before
  - h_3, h_4 and h_5 encode the top element of the first stack (remember the alphabet is $\{0, 1\}$ and we need one more slot for denoting an empty stack!)
  - h_6, h_7 and h_8 encode the top element of the second stack
  - The rest of the vector (positions 9 to $8 + N$) indicate which state the Turing machine is currently in

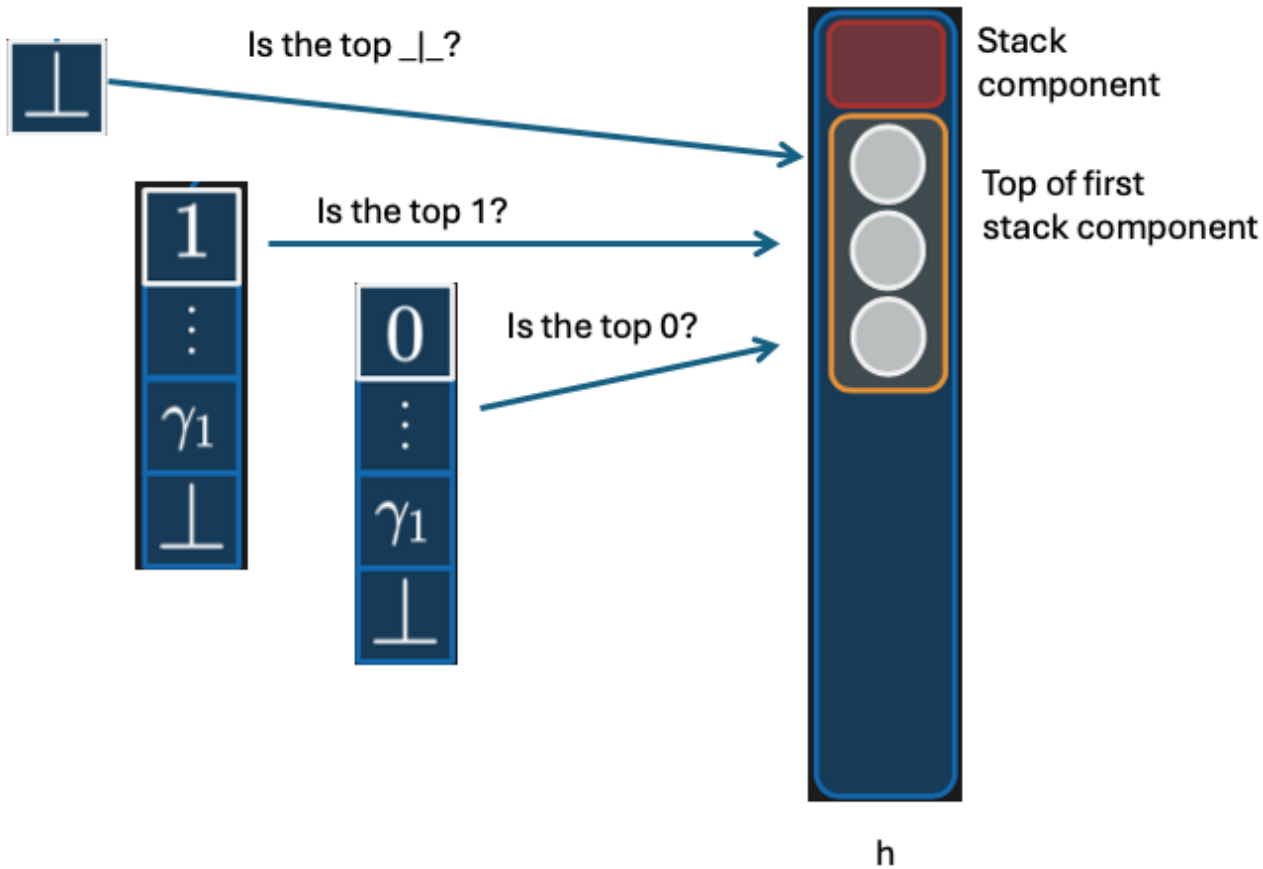
The Construction (adapted from Butoi et al.)

- Encode the contents of the two stacks



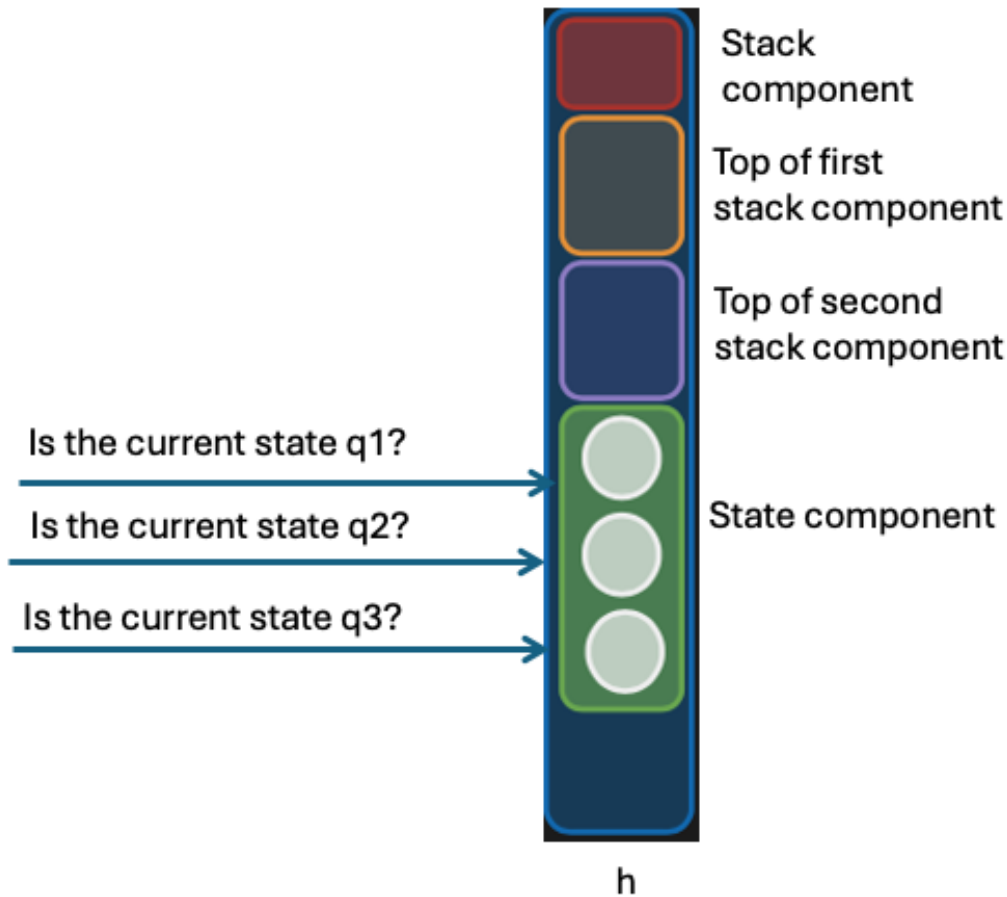
The Construction (adapted from Butoi et al.)

- Encode the contents of the two stacks
- Encode the top of the two stacks



The Construction (adapted from Butoi et al.)

- Encode the contents of the two stacks
- Encode the top of the two stacks
- Encode the current state



How to simulate a 2-stack PDA?

We can set the parameters of the RNN U, V, b such that it will transition between configurations of the Turing machine according to its transition rules

For more details, see Siegelmann and Songtag (1992)

Note that we are not doing **any** training here! We figure out the parameters in a constructive manner, not learning from data

Summary: Turing Machines and RNNs

- Turing machines can be simulated using two-stack automata
- Two-stack automata can be simulated using RNNs with unbounded precision and computation
- The state in the RNN encodes the two stacks and the state in the two-stack automaton (a "configuration")
- Therefore: **RNNs with unbounded precision and computation are as powerful as Turing machines!**

Back to RNNs Power

Depending on the amount of power we provide to the latent states of RNNs, they can either:

- Implement any computable function
- Be significantly limited to finite state languages

  If RNNs are bounded in precision and computation, they are equivalent to FSMs!

Back to RNNs Power

Depending on the amount of power we provide to the latent states of RNNs, they can either:

- Implement any computable function
- Be significantly limited to finite state languages

  If RNNs are bounded in precision and computation, they are equivalent to FSMs!

Bounded precision and computation

- An RNN with the update rule

$$h_{t+1} = H(Uh_t + Vr(y_t) + b)$$

can simulate any deterministic automaton (where $H(x) = 0$ if $x \leq 0$ and 1 otherwise; The Heaviside function)

Bounded precision and computation

- An RNN with the update rule

$$h_{t+1} = H(Uh_t + Vr(y_t) + b)$$

  can simulate any deterministic automaton (where $H(x) = 0$ if $x \leq 0$ and 1 otherwise; The Heaviside function)

- An RNN with bounded precision is no more expressive than a finite state automaton

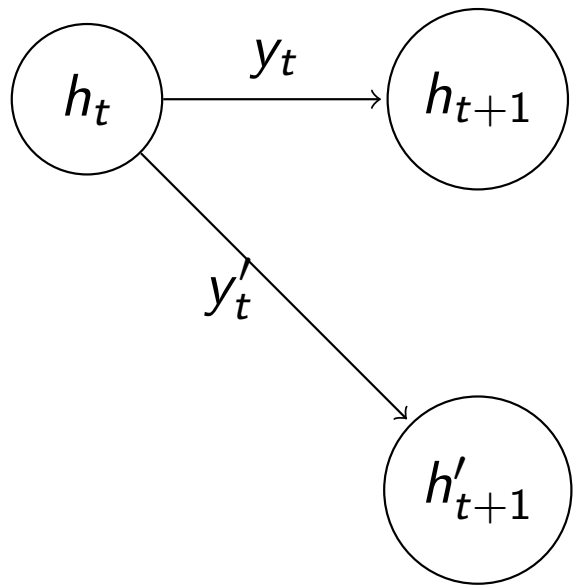
  Class exercise: Try to construct the latter argument

Any RNN can be an FSM

(Note we assume that the activation leads to binary states!)

$$h_{t+1} = f(h_t, y_t)$$

$$h'_{t+1} = f(h_t, y'_t)$$



RNNs can simulate FSMs

- The construction takes any **deterministic** FSM and transforms it into an RNN
  Food for thought: What about non-deterministic FSMs?

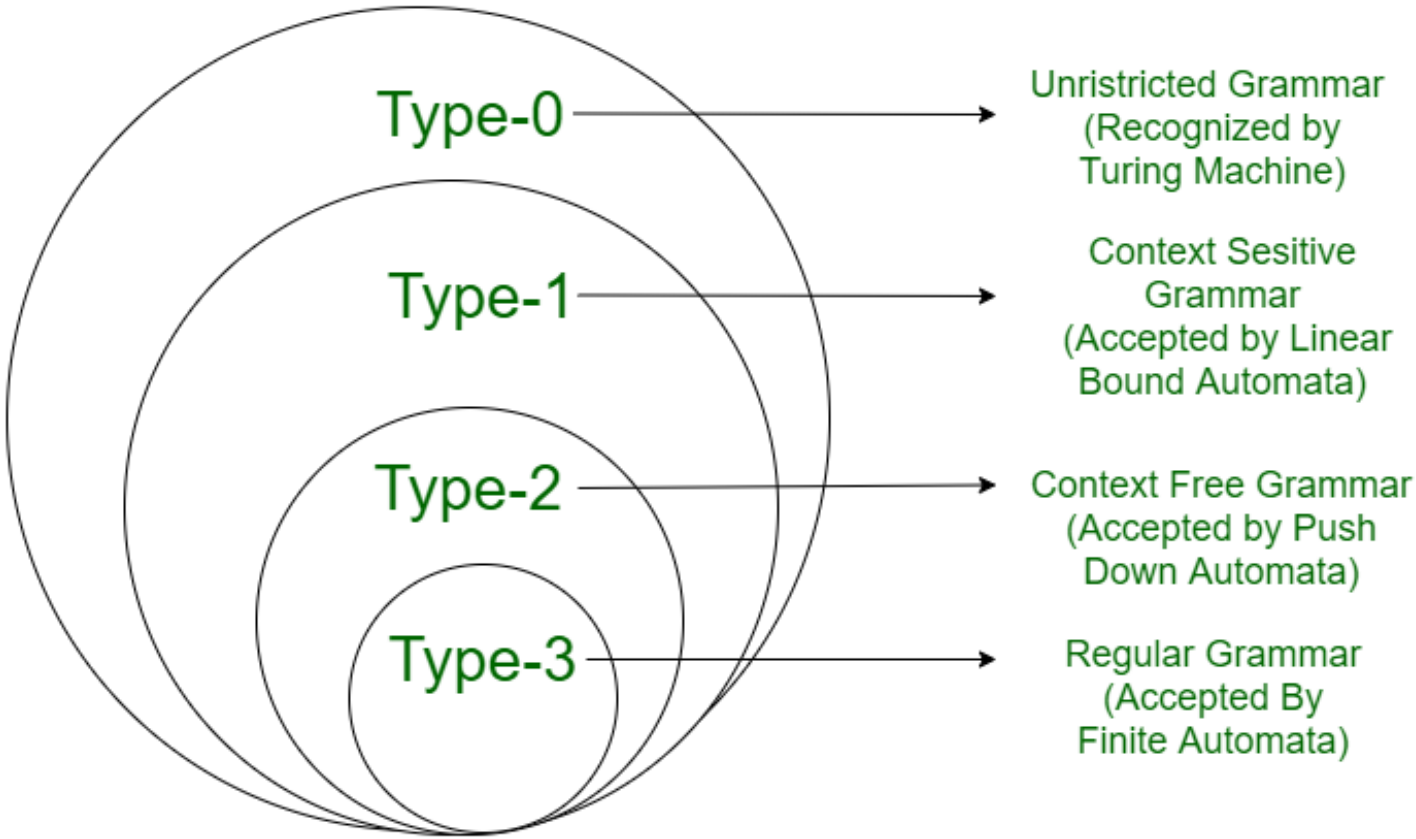
RNNs can simulate FSMs

- The construction takes any **deterministic** FSM and transforms it into an RNN
  Food for thought: What about non-deterministic FSMs? Any non-deterministic FSM can be determinised
- The latent state h_t will be indexed by a pair of state and symbol. The state provides the current state in the simulated machine, and the symbol provides the symbol we are being fed
- $h_{t+1} = H(Uh_t + Vr(y_t) + b)$. We will set U to give 0.5 for all elements in the vector (q, a) ranging over a such that transition from q in h_t provides q'; We will set V to add another 0.5 to all elements in the vector that correspond to y_t
- The H function now will zero out all those that are just 0.5 and keep only one element that is 1.0. We get from that h_{t+1}.

The devil is in the details

- To make the construction work we need more attention to details, but that's the construction at a high level

A stark difference in the Chomsky hierarchy



RNNs with finite precision and bounded computation are reduced from Type-0 to Type-3.

Summary

- Studying LLMs in formal terms is a fertile ground to understand them better

References

A. Butoi, R. Chan, R. Cotterell, W. Merrill, F. Nowak, C. Pasti, L. Strobl, A. Svete, Language Models and Formal Languages, tutorial in ACL 2024, `https://acl2024.ivia.ch/`

H. T. Siegelman and E. D. Sontag. Neural networks with real weights: analog computational complexity. COLT 1992.