On the Representational Capacity of Recurrent Neural Language Models

Franz Nowak[1] **Anej Svete**[1] **Li Du**[2] **Ryan Cotterell**[1]
[1]ETH Zürich [2] Johns Hopkins University
{fnowak, asvete, rcotterell}@ethz.ch leodu@cs.jhu.edu

Abstract

This work investigates the computational expressivity of language models (LMs) based on recurrent neural networks (RNNs). Siegelmann and Sontag (1992) famously showed that RNNs with rational weights and hidden states and unbounded computation time are Turing complete. However, LMs define weightings over strings in addition to just (unweighted) language membership and the analysis of the computational power of RNN LMs (RLMs) should reflect this. We extend the Turing completeness result to the probabilistic case, showing how a rationally weighted RLM with unbounded computation time can simulate any *probabilistic* Turing machine (PTM). Since, in practice, RLMs work in real-time, processing a symbol at every time step, we treat the above result as an upper bound on the expressivity of RLMs. We also provide a lower bound by showing that under the restriction to real-time computation, such models can simulate deterministic real-time rational PTMs.

https://github.com/rycolab/rnn-turing-completeness

$$\begin{tikzcd}
\mathrm{PTM} \arrow[r, phantom, "="] \arrow[r, phantom, "\text{Prop. C.1}"'] & \mathbb{Q}\mathrm{PTM} \arrow[r, phantom, "="] \arrow[r, phantom, "\text{Prop. 3.2}"'] & \mathrm{2PDA} \\
& & \Sigma\text{–}\mathrm{2PDA}{=}\varepsilon\mathrm{RLM} \arrow[u, hook, "\text{Thm. 3.1}"'] \\
\textbf{?} \arrow[uu, hook] \arrow[rr, dashed, leftrightarrow, "\textit{unbounded}", "\textit{real-time, deterministic}"'] & & \mathrm{RLM} \arrow[u, hook, "\text{Prop. 4.1}"] \\
\mathrm{RD}\text{–}\mathbb{Q}\mathrm{PTM} \arrow[u, hook] \arrow[rr, phantom, "="] \arrow[rr, phantom, "\text{Prop. 4.2}"'] & & \mathrm{RD}\text{–}\mathrm{2PDA} \arrow[u, hook, "\text{Thm. 4.1}"']
\end{tikzcd}$$

Figure 1: Roadmap through the paper showing relations between models of computation and RLMs. PTM is the classic Probabilistic Turing machine. $\mathbb{Q}$PTM is a probabilistic Turing machine with multiple rationally weighted transition functions. 2PDA is a two-tape probabilistic pushdown automaton. Σ–2PDA is a 2PDA that is deterministic in its output alphabet. RLM is a simple RNN LM. εRLM is the same but augmented with an empty symbol (ε). The prefix "RD-" denotes deterministic real-time machines.

1 Introduction

A **language model** (LM) is definitionally a semimeasure[1] over strings (Icard, 2020). Recent advances in their capabilities, leading to the widespread adoption of LMs, have sparked interest in their theoretical properties and guarantees. Previous work has characterized modern architectures such as recurrent neural networks (RNNs; Elman, 1990; Hochreiter and Schmidhuber, 1997) in terms of the formal languages they can and cannot recognize (Kleene, 1956; Minsky, 1967; Siegelmann and Sontag, 1992; Merrill et al., 2020, *inter alia*). However, characterizing LMs as formal languages is, in some sense, a category error because LMs encode semimeasures over strings instead of deciding language membership (Chen et al., 2018). In this work, we thus offer another perspective on understanding RNN LMs (RLMs) by asking: What classes of semimeasures over strings can RLMs represent, i.e., what is their computational expressivity?

The empirical capabilities of trained language models have spurred a large field of work testing their reasoning and linguistic abilities. However, our theoretical understanding of what these models are inherently capable of is still lacking (Deletang et al., 2023). Connecting an LM architecture to well-understood models of computation can help us determine whether the architecture is able to perform the sequences of computations required to carry out an algorithmic task (Pérez et al., 2019). Furthermore, connecting it to linguistic models can tell us whether the architecture is capable of correctly modeling the linguistic structure of a sentence symbolically (Linzen et al., 2016). Finally, characterizing the types of semimeasures the architecture can represent allows us to make more concrete claims about the abilities and limitations of the architecture itself.

RLMs have set many important milestones in language modeling and still hold the state of the art in some important settings of natural language processing (Qiu et al., 2020; Orvieto et al., 2023). Moreover, despite the recent trend towards the recurrence-free and, thus, parallelizable,

[1]Roughly, a semimeasure is a generalization of a probability measure that is sub-additive, meaning the measure of Σ^* may be less than one (Li and Vitányi, 2008, Ch. 4).

transformer-based LMs (Vaswani et al., 2017), elements of recurrence have found their way into recent language models and RNNs themselves have recently even been proposed as alternatives or extensions to some high-performing models (Peng et al., 2023; Orvieto et al., 2023; Zhou et al., 2023). At a high level, RNNs work by maintaining a hidden state encoding the processed string, much like how formal models of computation such as Turing machines process and store information. This sequential nature has motivated the comparison of the computational power of RNNs to that of various formal models of computation, from simple models such as finite-state automata (Kleene, 1956; Merrill et al., 2020) and counter machines, all the way up to Turing machines and related models (Minsky, 1967; Siegelmann and Sontag, 1992; Weiss et al., 2018).

Precisely where RNNs end up on the hierarchy of formal models of computation depends on the specific formalization. In this work, we characterize the computational power of RNNs in their most permissive formalization, i.e., one that allows RNNs to process and produce rational-valued vectors and perform an unbounded number of computational steps per input symbol by allowing them to emit empty tokens, ε, in between words. Siegelmann and Sontag (1992) show that such RNNs can simulate any deterministic Turing machine and are, hence, Turing complete.[2] While this sheds light on the processing power of RNNs, their result is not directly applicable to language modeling, as it does not take into account the probability assigned to the strings. By extending Siegelmann and Sontag's (1992) construction to the probabilistic case, we provide first steps towards understanding the expressive power of RLMs with rational arithmetic. We show that RLMs with rational weights and unbounded computation time can compute exactly the same semimeasures over strings as probabilistic Turing machines.

On one hand, rational arithmetic offers a reasonably faithful formalization of real-world models in that computer scientists often analyze numerical algorithms using such an idealization.[3] However, on the other hand, the assumption of unbounded computation time *does* represent a large departure from realistic models. In practice, RLMs perform a constant number of computational steps per symbol, operating in a real-time setting (Weiss et al., 2018). Therefore, we treat the above result as an *upper bound* on the computational power of recurrent RLMs. As a lower bound, we study a second type of RLMs, restricting the models to operate in real-time, which results in a more fine-grained hierarchy of specific Turing machine-like models equivalent to an RLM. We hence characterize the expressivity of RLMs in terms of classical computational models.

Our work offers a first step towards a comprehensive characterization of the expressivity of RLMs in terms of the classes of probability measures they can represent. In addition to providing insights into the computational capacity of RLMs, the work also follows the recent exploration of the measure-theoretic foundations of LMs (Welleck et al., 2020; Meister et al., 2023; Du et al., 2023), while focusing on a particular architecture. We conclude the paper by posing several open questions on the exact position of RLMs in the hierarchy of relevant computational models. Fig. 1 shows a roadmap of the paper, with the two types of RLMs of interest and their relation to different formal computational models.

[2]Note that the restriction on the Turing machine of being deterministic does not change the generated language since any non-deterministic Turing machine has an equivalent deterministic Turing machine, albeit with a potentially much longer running time for a given string (Gill, 1974).

[3]In many cases even real arithmetic is assumed, e.g., when theoretically analyzing optimization algorithms (Forst and Hoffmann, 2010, Ch. 1).

2 Preliminiaries

In this section, we build up the necessary definitions and vocabulary for the rest of the paper.

2.1 Recurrent Neural Language Models

A **formal language** L is a subset of the Kleene closure Σ^* of some finite non-empty set of symbols, i.e., an **alphabet**, Σ. An element of Σ^* is called a **string**, $\boldsymbol{y}$. Furthermore, ε denotes the empty string. We assume throughout that $\varepsilon \notin \Sigma$ and denote $\Sigma_\varepsilon \overset{\text{def}}{=} \Sigma \cup \{\varepsilon\}$. A **semimeasure** (Li and Vitányi, 2008) over Σ^* is a function $\mu\colon \Sigma^* \to [0,1]$ such that *(i)* $\mu(\varepsilon) \leq 1$ and *(ii)* for any string $\boldsymbol{y} = y_1 y_2 \ldots y_N \in \Sigma^*$, $\mu(\boldsymbol{y}) \geq \sum_{y \in \Sigma} \mu(\boldsymbol{y}y)$. If the semimeasure of all strings sums to one, i.e., $\sum_{\boldsymbol{y} \in \Sigma^*} \mu(\boldsymbol{y}) = 1$, then μ is called a **probability measure**. A **language model** (LM) p is defined as a semimeasure over Σ^*. If p is a probability measure, we call it a **tight** language model.

Most modern LMs are autoregressive, meaning they define $p(\boldsymbol{y})$ through conditional semimeasures of the next symbol given the string produced so far and the probability of ending the string, i.e.,

$$p(\boldsymbol{y}) \overset{\text{def}}{=} p(\text{EOS} \mid \boldsymbol{y}) \prod_{n=1}^{N} p(y_n \mid \boldsymbol{y}_{<n}) \qquad (1)$$

where EOS denotes the special end-of-string symbol, which specifies that the generation of a string has halted. The inclusion of EOS allows a p defined autoregressively to define a probability measure over Σ^* (Du et al., 2023). We will denote $\overline{\Sigma} \overset{\text{def}}{=} \Sigma \cup \{\text{EOS}\}$.

We will use the following definition of an RNN.

Definition 2.1. *A **simple RNN** $\mathcal{R}$ is an RNN with the following hidden state update rule:*

$$\mathbf{h}_t \overset{\text{def}}{=} f(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{V}\mathbf{r}(y_t) + \mathbf{b}) \qquad (2)$$

*where $\mathbf{h}_0$ is a vector in $\mathbb{Q}^D$, D is the dimensionality of the hidden state, $\mathbf{r}\colon \Sigma \to \mathbb{Q}^R$ is the symbol representation function, $\mathbf{b} \in \mathbb{Q}^D$, $\mathbf{U} \in \mathbb{Q}^{D\times D}$, and $\mathbf{V} \in \mathbb{Q}^{D\times R}$. The function f is the **saturated sigmoid**, defined as:*

$$f(x) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } x < 0 \\ x & \textit{if } x \in [0,1] \\ 1 & \textit{if } x > 1 \end{cases} \qquad (3)$$

Due to their sequential nature, RNNs have been linked to formal models of computation such as finite-state automata, pushdown automata (PDA), and Turing machines under various formalizations with different implications on computational power (e.g., Siegelmann and Sontag, 1992; Hao et al., 2018; Korsky and Berwick, 2019; Merrill, 2019; Merrill et al., 2020; Hewitt et al., 2020, *inter alia*). For example, if, instead of using the saturated sigmoid, we assumed that f is a function that maps to a finite set, this would result in RNNs that are at most as expressive as finite-state automata (Minsky, 1967; Svete and Cotterell, 2023). Merrill et al. (2020) study the computational power of saturated RNNs by investigating the effect of asymptotically large weights. Finally, Siegelmann and Sontag (1992) assumes rational-valued arithmetic, which is the convention we follow in this work.

An RNN specifies an LM by defining a conditional probability measure over y_t given $\boldsymbol{y}_{<t}$. Let $\mathbf{E} \in \mathbb{Q}^{|\overline{\Sigma}|\times D}$ be an output matrix and $\mathcal{R}$ an RNN. An RLM is an LM whose conditional probability measures are defined by projecting $\mathbf{E}\mathbf{h}_t$ to the probability simplex $\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$ using a projection function $\boldsymbol{\pi}\colon \mathbb{Q}^D \to \boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$:

$$p(y_t \mid \boldsymbol{y}_{<t}) \overset{\text{def}}{=} \boldsymbol{\pi}(\mathbf{E}\mathbf{h}_t)_{y_t} \qquad (4)$$

When generating from an RLM, we assume the next symbol is sampled according to the probabilities defined by $\boldsymbol{\pi}(\mathbf{E}\mathbf{h}_t)$ and is then passed as the next input symbol back into the RNN until EOS is generated.

2.2 Turing Machines

We use a reformulation of the classic definition of a probabilistic Turing machine similar to Weihrauch's (2000) Type-2 Turing machine:[4]

Definition 2.2. *A **probabilistic Turing machine** (PTM) is a two-tape machine specified by the 6-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta_1, \delta_2, q_\iota, q_\varphi)$, where*

- *Q is a finite set of states;*
- *Σ and Γ are the input and tape alphabets, and Γ includes the blank symbol $\sqcup$;*
- *$q_\iota, q_\varphi \in Q$ are initial and final states;*
- *$\delta_{\{1,2\}} : Q \times \Gamma \to Q \times \Gamma \times \Sigma_\varepsilon \times \{L, R, N\}$ are two transition functions, one of which is chosen at random at each computation step.*

The Turing machine defined above has two tapes. The first is a working tape on which symbols from the tape alphabet Γ can be read and written. The second is an append-only output tape on which $\mathcal{M}$ writes symbols of the output alphabet Σ. In the beginning, both tapes are empty, i.e., the working tape has only blank symbols $\sqcup$, and the output tape has only empty symbols ε. Starting in the initial state q_ι, at any time step t, the machine samples one of the two transition functions at random, each with probability $\frac{1}{2}$, and applies it. A given transition can be written as $(q, \gamma) \xrightarrow{y/d} (q', \gamma')$, where $q, q' \in Q, \gamma, \gamma' \in \Gamma, y \in \Sigma_\varepsilon$, and $d \in \{L, R, N\}$. The semantics of such a transition is as follows: When in state q and reading γ on the working tape, go to state q', write γ' to the working tape, write $y \in \Sigma_\varepsilon$ to the output tape, and move the head on the working tape by one symbol along the tape in the direction d, that is, left (L), right (R), or stay in place (N).[5] When $y = \varepsilon$, the machine simply does not write anything on the output tape. The machine **halts** once it reaches the final state q_φ. We call the

[4]Note that our adding an additional tape does not increase the power of the Turing machine (Sipser, 2013, Ch. 3).

[5]The stay-in-place operation is often added to make proofs simpler but does not add expressivity (Sipser, 2013, Ch. 3).

sequence of symbols $\boldsymbol{y} \in \Sigma^*$ on the output tape at that point the **output** of the machine.

Note that, once a transition function has been chosen, since it is a function, the next transition is uniquely determined by the current state q and the current tape symbol γ under the read-write head. In the following, we call a pair of $(q, \gamma) \in Q \times \Gamma$ a **configuration** of $\mathcal{M}$.

Remark 2.1. *Given a probabilistic Turing machine $\mathcal{M}$ as defined above, we can get the probability of $\mathcal{M}$ halting and outputting a specific string $\boldsymbol{y}$ by summing the probabilities of all halting paths*[6] *through the machine that result in $\boldsymbol{y}$ being written on the output tape (the probability of each path is 2^{-n}, where n is the number of computation steps).*

Remark 2.2. *The notion of halting probability as defined in remark 2.1 has a counterpart in* RLMs*, namely, the probability mass placed on all finite strings generated (Icard, 2020). For details, see Appendix A.*

This induces a semimeasure over the possible sequences $\boldsymbol{y} \in \Sigma^*$ that a PTM $\mathcal{M}$ can output, which we will call $\mathbb{P}_{\mathcal{M}}$. That is, $\mathbb{P}_{\mathcal{M}}(\boldsymbol{y})$ is the probability that $\mathcal{M}$ will halt with $\boldsymbol{y}$ as its output.

2.3 Pushdown Automata

We now move to another probabilistic computational model: The two-stack pushdown automaton.

Definition 2.3. *A* ***probabilistic two-stack pushdown automaton*** *(*2PDA*) is a two-stack-machine defined by the tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_\iota, q_\varphi)$, where*

- *Q is a finite set of states;*
- *Σ and Γ are the input and stack alphabets, and Γ includes the bottom-of-stack symbol $\bot$;*
- *$q_\iota, q_\varphi \in Q$ are the initial and final states;*
- *$\delta \colon Q \times \Gamma \times \Sigma_\varepsilon \times Q \times \Gamma_\varepsilon^4 \to \mathbb{Q}$ is a transition weighting function.*

To make the connection to Turing machines more straightforward, our definition of a 2PDA assumes[7] that its transitions depend on its current state q and the top stack symbol of only the first of the two stacks. We write transitions as $q \xrightarrow[\gamma_2 \to \gamma_4]{y, \gamma, \gamma_1 \to \gamma_3} q'$, for $q, q' \in Q$, $y \in \Sigma_\varepsilon$, $\gamma \in \Gamma, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \in \Gamma_\varepsilon$. Such a transition denotes that the 2PDA in the state q with the symbol γ on top of the first stack pops γ_1 and γ_2 from the first and second stack, pushes γ_3 and γ_4 onto the stacks and moves to state q'. At the same time, the 2PDA consumes or emits (depending on the use-case) a symbol from Σ_ε.

Again we assume the rational weighting function δ of a 2PDA is locally normalized over configurations $(q, \gamma) \in \Sigma \times \Gamma$, where γ is the symbol currently on the top of the first stack:

$$\sum_{\substack{y \in \Sigma_\varepsilon, q' \in Q, \\ \gamma_1, \gamma_2, \gamma_3, \gamma_4 \in \Gamma_\varepsilon}} \delta\left(q \xrightarrow[\gamma_2 \to \gamma_4]{y, \gamma, \gamma_1 \to \gamma_3} q'\right) = 1 \quad (5)$$

A 2PDA starts at the initial state q_ι with both stacks empty (only containing the symbol $\bot$) and then sequentially applies transitions according to their probability given by δ. The automaton halts when reaching the final state, q_φ. The sequence of the symbols output by the automaton concatenated in the order of transitions taken constitutes the output string.

A note on variants of computational models. Our version of (probabilistic) Turing machines differs from the traditional definition of mere language acceptors in that they start from a starting state q_ι and then iteratively apply probabilistic transitions to generate outputs $\boldsymbol{y} \in \Sigma^*$, where each specific $\boldsymbol{y}$ has a corresponding probability of being produced. This is to simplify the comparison to 2PDA and to be able to interpret them as language models in their own right.

Definition 2.4. *We say that two probabilistic computational models $\mathcal{M}_1$ and $\mathcal{M}_2$ are* ***weakly equivalent*** *if, for any string $\boldsymbol{y} \in \Sigma^*$, we have $\mathbb{P}_{\mathcal{M}1}(\boldsymbol{y}) = \mathbb{P}_{\mathcal{M}2}(\boldsymbol{y})$. If, furthermore, there exists a bijection between halting paths with the same output in the two models, they are called* ***strongly equivalent****.*[8]

3 An Upper Bound

In this section, we establish an upper bound on the expressive power of RLMs by extending Siegelmann and Sontag's (1992) result to the probabilistic case of language models. Because we want to upper bound the power of RLMs used in practice, we will start with a more unrealistic recurrent LM which can output empty symbols (ε), which we denote as εRLM. We first introduce a variant of probabilistic Turing machines that can have an arbitrary (finite) number of rationally valued transition

[6]We do not formally define the notion of paths here.

[7]This is without loss of generality; see Appendix B.

[8]We also lift this definition to *classes* of models (C_1, C_2), which are called strongly equivalent if $\forall \mathcal{M}_1 \in C_1 \exists \mathcal{M}_2 \in C_2$ such that $\mathcal{M}_1$ and $\mathcal{M}_2$ are strongly equivalent, and vice versa.

functions and show that they are strongly equivalent to 2PDA (Section 3.1). We then review Siegelmann and Sontag's (1992) construction for the unweighted case (Section 3.2). Finally, we extend this construction to the probabilistic case by showing how to simulate a 2PDA with an εRLM. We conclude with the observation that this results in the equivalence of PTMs and εRLMs (Section 3.3).

3.1 Rationally Weighted PTMs

This paper considers the expressive power of RNNs with *rational* weights. To make the connection to PTMs easier, it is helpful to define a more general type of a PTM which, instead of sampling between two equally probable transition functions, can have any number of possible transitions at a given computation step, each of which has a rational probability of being applied.

Definition 3.1. *A **rational-valued probabilistic Turing machine** ($\mathbb{Q}$PTM) is a PTM whose transition weighting function is of the form:*

$$\delta\colon Q \times \Gamma \times \Sigma_\varepsilon \times Q \times \Gamma \times \{L, R, N\} \to \mathbb{Q} \quad (6)$$

In other words, for any current configuration, it assigns a rational-valued probability in the interval $[0,1]$ to each available transition. We require that the probabilities are normalized over configurations, that is, for all $q \in Q, \gamma \in \Gamma$:

$$\sum_{\substack{y\in\Sigma_\varepsilon,\, q'\in Q,\\ \gamma'\in\Gamma,\, d\in\{L,R,N\}}} \delta\left((q,\gamma) \xrightarrow{y/d} (q',\gamma')\right) = 1 \quad (7)$$

The original construction by Siegelmann and Sontag (1992) uses unweighted 2PDA which are equivalent to Turing machines (Hopcroft et al., 2001). We now want to show that we can simulate a PTM with an εRLM in the same way, that is via *probabilistic* 2PDA as defined above. Therefore, we first show that PTMs and probabilistic 2PDA are also equivalent, in the following two propositions:

Proposition 3.1. PTM*s and* $\mathbb{Q}$PTM*s are weakly equivalent.*

Proof. See Appendix C for the proof sketch. ■

Proposition 3.2. $\mathbb{Q}$PTM*s and* 2PDA *are strongly equivalent.*

Proof sketch. The proof is similar to that of the unweighted case which shows that any deterministic Turing machine can be simulated by a two-stack pushdown automaton (see Fig. 2). A full proof of

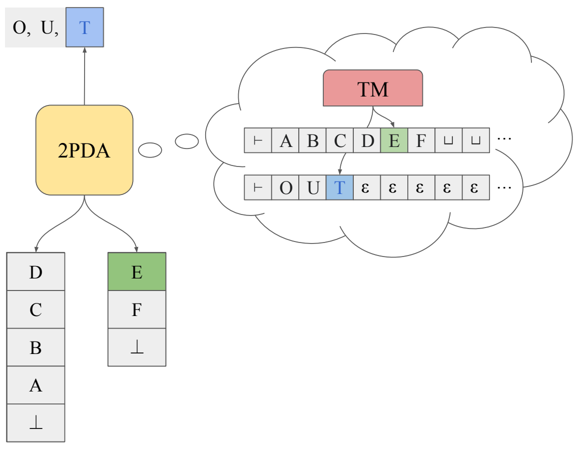

Figure 2: Simplified view of a 2PDA simulating a TM.

this is given, e.g., in Hopcroft et al. 2001, Thm. 8.13 and sketched here. The idea is for the two stacks together to simulate an infinite tape, where the first stack contains the symbols to the right of the TM's head, the top symbol on the first stack is the tape symbol under the TM's head, and the second stack contains those to the left of the head. This is easily extended to the probabilistic case using the introduced definitions of $\mathbb{Q}$PTMs and 2PDA, meaning that for any $\mathbb{Q}$PTM $\mathcal{M}$ we can construct a 2PDA $\mathcal{P}$ such that each transition of $\mathcal{M}$ corresponds exactly to one transition of $\mathcal{P}$. To simulate $\mathcal{M}$ with $\mathcal{P}$, we let the states and alphabets of $\mathcal{P}$ be the same as those of $\mathcal{M}$. Then, a transition of $\mathcal{M}$ is modeled as follows in $\mathcal{P}$:

- $\mathcal{P}$'s current state is the same as that of $\mathcal{M}$;
- The symbol scanned by $\mathcal{M}$'s head is the symbol at the top of $\mathcal{P}$'s first stack. If the top symbol of the first stack is $\perp$, $\mathcal{P}$ interprets this as $\mathcal{M}$ having just moved to $\sqcup$;
- $\mathcal{P}$ randomly chooses one of the transitions available based on the state and current top symbol on the first stack (which are the same as those available to $\mathcal{M}$);
- Say, for instance, $\mathcal{M}$ is in state q, the current symbol under the head is γ, and the randomly selected transition would cause $\mathcal{M}$ to write symbol y to the output tape, write the tape symbol γ' to the tape, and move in direction R. Then $\mathcal{P}$ would pop γ from its first stack, push it to its second stack, and emit y.

$\mathcal{P}$ halts when $\mathcal{M}$ halts, namely when reaching state q_ι. Since each transition of $\mathcal{M}$ has an equivalent transition in $\mathcal{P}$, this means that the same strings are accepted with the same corresponding probabilities by both $\mathcal{P}$ and the simulated $\mathcal{M}$. This also works in the opposite direction,[9] forming a bijection between halting paths, meaning we have strong equivalence. ■

3.2 Simulating Unweighted TMs

Before we introduce the equivalence of the models in the probabilistic case, we review the classical unweighed construction of an RNN simulating a TM first introduced by Siegelmann and Sontag (1992) and simplified by Chung and Siegelmann (2021). Specifically, Siegelmann and Sontag (1992) show that a simple RNN can encode a TM by simulating a deterministic unweighted 2PDA.[10] This 2PDA takes an input string $\boldsymbol{y}$ and maps it to the output $\mathcal{M}(\boldsymbol{y})$ given by the simulated Turing machine:

$$\mathcal{M}(\boldsymbol{y}) = \begin{cases} 1 & \textbf{if } \mathcal{M} \text{ accepts } \boldsymbol{y} \\ 0 & \textbf{if } \mathcal{M} \text{ rejects } \boldsymbol{y} \\ \texttt{undef} & \textbf{if } \mathcal{M} \text{ runs forever on } \boldsymbol{y} \end{cases} \tag{8}$$

Given a deterministic unweighted 2PDA, their construction defines an RNN that generates the encoding of the string $\mathcal{M}(\boldsymbol{y})$ on the simulated stacks before halting, or never halts if $\mathcal{M}(\boldsymbol{y}) = \texttt{undef}$. Because the construction considers (deterministic) unweighted TMs, the mapping from $\boldsymbol{y}$ to $\mathcal{M}(\boldsymbol{y})$ is meant in the sense that $\mathcal{M}(\boldsymbol{y})$ is written in the (simulated) stack at the end of the execution and *not* in the sense that this output string was generated in the language modeling sense.

The crux of the construction lies in encoding the content of a stack in a neuron.[11] Importantly, the encoding must be such that *(i)* the tops of the stacks can easily be read and *(ii)* the encoding of the stack can easily be updated upon popping off or pushing onto the stack. More precisely, this can, for example, be achieved by mapping a (binary) string $\boldsymbol{\gamma} = \gamma_1 \ldots \gamma_N$ into $\eta(\boldsymbol{\gamma}) \overset{\text{def}}{=} 0.\eta(\gamma_N)\ldots\eta(\gamma_1)$, where:[12]

$$\eta(\gamma) \overset{\text{def}}{=} \begin{cases} 1 & \textbf{if } \gamma = 0 \\ 3 & \textbf{otherwise} \end{cases} \tag{9}$$

Notice the opposite orientation of the two encodings: The top of the stack in $\boldsymbol{\gamma}$ is written on the right-hand side while it is the left-most digit in the numerical encoding which enables easy updates to the encoding; with this, popping $\gamma_N = 0$ can, for example, be performed by computing $f(10 \cdot \eta(\boldsymbol{\gamma}) - 1)$, popping $\gamma_N = 1$ by computing $f(10 \cdot \eta(\boldsymbol{\gamma}) - 3)$, pushing $\gamma = 0$ by computing $f\left(\frac{1}{10} \cdot \eta(\boldsymbol{\gamma}) + \frac{1}{10}\right)$, and pushing $\gamma = 1$ by computing $f\left(\frac{1}{10} \cdot \eta(\boldsymbol{\gamma}) + \frac{3}{10}\right)$.[13]

Similarly, the current state of a 2PDA is stored in a set of neurons keeping the one-hot encoding of the state, which is updated by simulating the transition function of the 2PDA. This can be done by intersecting the states reachable from the current configuration of the 2PDA and the states reachable by the currently read symbol, the same way as in the classical Minsky construction of a simple RNN simulating a finite-state automaton (Minsky, 1954). Because of the determinism of the transition function, this results in a single possible next state. The intersection can be implemented using conjugation, which is possible using the saturated sigmoid function.

With this, an RNN simulating a 2PDA can be constructed by keeping a hidden state vector divided into multiple sets of values, three of which will be relevant for our extension: (1) Two **stack neurons**, each representing a stack; (2) Two **readout neurons**, each encoding the symbols on top of one of the stacks; (3) $\log |Q|$ **state neurons** encoding the current state of the 2PDA. The readout neurons can be computed from the stack encodings $\eta(\boldsymbol{\gamma}_1)$ and $\eta(\boldsymbol{\gamma}_2)$ similarly to how the stack encodings are updated. See top of Fig. 3 for an illustration of how these components can be used to determine the quantities relevant to determining the next action of the 2PDA. More details of the construction can be found in Chung and Siegelmann (2021, Thm. 1).

Importantly, note that in this case, the RNN (and the 2PDA) can be fully deterministic due to the equivalence of deterministic and non-deterministic unweighted TMs. They also do not have to consider any ε-transitions or steps generating ε's since there is no generation in the sense of Section 2.3.

[9]See Appendix D for details on the backward direction.

[10]Naturally, as Prop. 3.2 suggests, *unweighted* two-stack PDAs are equivalent to unweighted TMs (Sipser, 2013).

[11]We use the term neuron as used in the machine learning literature to refer to a mathematical model of a biological neuron that takes inputs and produces an output in the form of rational values.

[12]Note that the stack encoding defined by Siegelmann and Sontag (1992) is somewhat different since it does not use the base-ten but rather base-four encoding. To keep the presentation simpler, we choose the base-ten one; the intuition remains the same.

[13]Since we are using a simple RNN, the coefficient cannot be chosen based on the current input. This is why *all* such actions are performed in parallel (into individual processing neurons). The result of the correct operation (based on the input symbol and the current stack configuration) can then be copied back into the stack neuron. This is why multiple RNN update sub-steps are required.

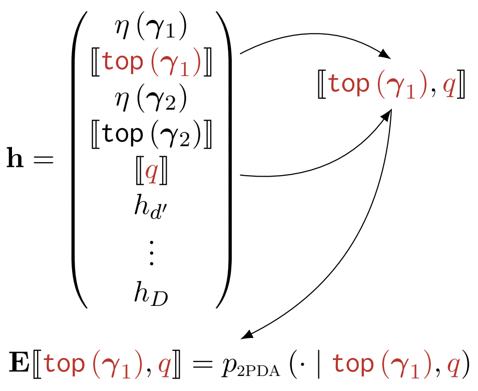
$\mathbf{h} = \left(\eta(\gamma_1), [\![\texttt{top}(\gamma_1)]\!], \eta(\gamma_2), [\![\texttt{top}(\gamma_2)]\!], [\![q]\!], h_{d'}, \vdots, h_D\right)$
$[\![\texttt{top}(\gamma_1), q]\!]$
$\mathbf{E}[\![\texttt{top}(\gamma_1), q]\!] = p_{\text{2PDA}}(\cdot \mid \texttt{top}(\gamma_1), q)$

Figure 3: A schematic illustration of how the model from Chung and Siegelmann (2021) stores the information about the configuration of the 2PDA and how it can be used to access the information needed for defining string probabilities. We denote with $[\![\cdot]\!]$ the one-hot encoding function of the input arguments. $h_{d'} \ldots h_D$ refer to the rest of the hidden state not directly relevant for determining the configuration of the 2PDA.

These two aspects will, however, require more attention in the probabilistic case, which we discuss next.

3.3 Simulating PTMs

A TM can perform an unbounded number of computational steps per output symbol. To account for this with RLMs in the language modeling setting, we extend their definition to one that allows generating ε's, effectively allowing RNNs to perform computations without affecting the output string (εRLM).

Definition 3.2. *An* RLM ***with*** ***ε-transitions*** *(εRLM) is an* RLM *that can output ε-symbols.*

More precisely, an εRLM defines a symbol representation function $\mathbf{r}\colon \Sigma_\varepsilon \to \mathbb{Q}^R$ and the output matrix $\mathbf{E} \in \mathbb{Q}^{|\overline{\Sigma}_\varepsilon| \times D}$, where D and R are parameters depending on the 2PDA (Chung and Siegelmann, 2021). Naturally, it does not actually output a symbol from $\overline{\Sigma}$ upon generating ε. Effectively, this gives an εRLM the possibility to perform an arbitrary number of computations per symbol of the string. With this additional gadget, we are able to state our main result establishing a close connection between PTMs and εRLMs.

On determinism. The construction we describe in the following theorem requires that the next transition of the 2PDA is fully specified given the current state (q, γ) and the (sampled) output symbol from Σ_ε. I.e., the non-determinism of the simulated 2PDA is constrained to the sampling step of the RLM, meaning there can only be one possible transition in the 2PDA per output symbol. We call a 2PDA or $\mathbb{Q}$PTM that has this property **Σ-deterministic**. Note that this is still a non-deterministic automaton; see Def. 4.1.

Theorem 3.1. *An ε*RLM *can simulate any Σ-deterministic probabilistic* 2PDA.

Proof sketch. Given a Σ-deterministic 2PDA, we design an εRLM that simulates that 2PDA by executing its transitions and hence defining the same semimeasure over strings. We use the LM controller from Chung and Siegelmann (2021) (with the same definitions of the parameters $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{b}$), as it conveniently models the transitions of the underlying 2PDA and exposes the parts of its configuration required to define the transition (and with it the string) probabilities. This leaves us with the task of appropriately defining the output matrix $\mathbf{E}$. Exposing the symbols on the top of the stacks, $\texttt{top}(\gamma_1)$ and $\texttt{top}(\gamma_2)$, and the current state q of the 2PDA in $\mathbf{h}_t$ (cf. Fig. 3) allow us to easily access the appropriate probabilities encoded in the output matrix $\mathbf{E}$. Note that due to the determinism of the 2PDA, a single pair of the stack symbol and the current state $(\texttt{top}(\gamma_1), q)$ determines the conditional probability measure over the next symbol.[14] More precisely, we define $\mathbf{E} \in \mathbb{Q}^{|\overline{\Sigma}_\varepsilon| \times |\Gamma_\varepsilon||Q|}$ which maps the one-hot encoding of the pair (γ, q) to a $|\overline{\Sigma}_\varepsilon|$-dimensional vector of probabilities over the next symbol.[15] To achieve that, simply let $E_{y,(\gamma,q)}$ correspond to $\delta\left(q \xrightarrow[\circ\to\circ]{y,\gamma,\circ\to\circ} \circ\right)$, where we index the output matrix directly with the elements for cleaner notation.[16] Denoting with $[\![\gamma, q]\!]$ the one-hot encoding of the tuple (γ, q), the vectors $\mathbf{E}[\![\gamma, q]\!]$ represent semimeasures over $\overline{\Sigma}_\varepsilon$, and $\boldsymbol{\pi}$ can be set to the identity function.[17] Considering that the εRLM directly simulates all possible runs

[14] Recall that the target configuration of the 2PDA depends only on the top symbol of the first stack, $\gamma \stackrel{\text{def}}{=} \texttt{top}(\gamma_1)$.

[15] One-hot encodings of the state-stack symbol pairs can be obtained by applying the RNN update (sub-)step in which the nonlinearity is used to implement conjunction. This adds another sub-step to the simulation of the full 2PDA update step.

[16] Again, due to the assumed determinism, the elements with $\circ$ are irrelevant for the weights.

[17] The identity function is generally *not* a projection function onto the probability simplex. However, since its inputs in this case already lie on the probability simplex, its use is possible. More generally, we could use the sparsemax function (Martins and Astudillo, 2016), which acts like the identity function on the probability simplex. Alternatively, we could use the more popular softmax function and set the entries of $\mathbf{E}$ to the logarithms of the original probabilities (defining $\log 0 \stackrel{\text{def}}{=} -\infty$).

of the $2\mathrm{PDA}$, it is easy to see that the $\varepsilon\mathrm{RLM}$ generates a string $\boldsymbol{y}$ with a sequence of actions if and only if the $2\mathrm{PDA}$ generates it as well. Moreover, the encoding of the probabilities in $\mathbf{E}$ means that the probabilities of the action sequences are always the same. Together, this shows that the two machines are strongly equivalent. ■

From Σ-determinism to full non-determinism. The construction outlined above allows an $\varepsilon\mathrm{RLM}$ to simulate an arbitrary Σ-deterministic $2\mathrm{PDA}$, and, by extension, a PTM. It is possible to extend this to fully non-deterministic $2\mathrm{PDA}$ by storing the direction of movement in the underlying PTM as part of the output symbol, e.g., by using two types of ε symbols, one that denotes the head moving to the left, ε_L, and one for moving to the right, ε_R; see Appendix E for details.

4 A Lower Bound

While Thm. 3.1 establishes a concrete result on the expressive power of $\varepsilon\mathrm{RLMs}$, the result follows from somewhat unrealistic assumptions, namely rationally weighted networks and unbounded computation time. We contend the first assumption is a reasonable approximation, since even for small neural networks the number of expressible states can be large; assuming double precision floating point numbers, an RNN can yield as many as $2^{64\cdot n}$ different states, where n is the number of neurons.[18] However, RLMs used in practice operate in real-time, outputting a symbol at every computation step. To make our analysis closer to this use case, in this section, we develop a lower bound on the expressivity of an RLM under the real-time restriction while still allowing rational arithmetic operations.

4.1 Real-time RLMs

Now, we switch back to studying the more common RLM with an RNN controller based on Def. 2.1. Firstly, note that the class of RLMs is a subset of the class of $\varepsilon\mathrm{RLMs}$:

Proposition 4.1. $\varepsilon\mathrm{RLM}$*s can simulate* RLM*s.*

Proof. This result follows trivially from Def. 3.2: An RLM is simply an $\varepsilon\mathrm{RLM}$ that always assigns probability 0 to outputting ε. ■

[18]To use the state space with limited precision more effectively, one could add more stack-encoding neurons or choose more efficient encodings of the stack contents.

4.2 Real-time Deterministic $2\mathrm{PDA}$

The lack of ε-transitions requires the properties of the simulated model to change: As in Thm. 3.1, the RNN construction requires that there is only one transition for every output symbol and configuration. Previously, this was done by imposing Σ-determinism, where non-determinism over symbols at a given time step can be reintroduced by delaying transitions through the use of additional ε-transitions, which is not possible here; see Appendix E. In fact, the lack of ε's and binarization means the resulting PDA has to be not just real-time, but also deterministic. We define such a $2\mathrm{PDA}$ analogously to the single stack case:[19]

Definition 4.1. *A* $2\mathrm{PDA}$ *is* ***determinisitc*** *if:*

- *For any current state $q \in Q$, current top stack symbol $\gamma \in \Gamma$, and a given output symbol $y \in \Sigma_\varepsilon$, there is at most one transition with non-zero probability.*
- *If, in a given computation step, the weight of an ε-transition is non-zero, then its weight is 1 and the weight of all other transitions is 0.*

If $\delta(q, \varepsilon, \gamma) = 0$ for all $q \in Q, \gamma \in \Gamma$ then we say it is a ***real-time deterministic probabilistic*** $2\mathrm{PDA}$ *(*$\mathrm{RD–2PDA}$*).*

Theorem 4.1. RLM*s can simulate* $\mathrm{RD–2PDA}$*s.*

Proof. This follows directly from Thm. 3.1 since an $\mathrm{RD–2PDA}$ is just a special case of the $2\mathrm{PDA}$ without ε-transitions which is exactly the restriction imposed on the RLM. ■

4.3 Real-time Deterministic $\mathbb{Q}\mathrm{PTM}$

As before, we want to connect the RLM with the better-understood PTM. To do so, we introduce a new class of rationally weighted PTMs that are deterministic and operate in real-time:

Definition 4.2. *A* $\mathbb{Q}\mathrm{PTM}$ *is* ***deterministic*** *if, for any configuration $q, \gamma \in Q \times \Gamma$, and any symbol $y \in \Sigma_\varepsilon$, there is at most one transition starting at that configuration and emitting y with non-zero probability. Furthermore, if there is a transition starting in (q, γ) outputting ε with non-zero probability, it must be the only possible transition in that configuration. If there are no ε-transitions with non-zero probability at all, then it is called* ***real-time*** *(*$\mathrm{RD–}\mathbb{Q}\mathrm{PTM}$*).*

[19]Real-time deterministic PDA have previously been investigated (Harrison and Havel, 1972; Pittl and Yehudai, 1983, *inter alia*), but to the authors' knowledge, this has not been extended to the two-stack case.

Proposition 4.2. *A* RD–$\mathbb{Q}$PTM *is exactly as powerful as a* RD–2PDA.

Proof sketch. The proof follows that of Prop. 3.2, with the only difference being that now the possible transitions for a given state and stack symbol (respectively, tape symbol) are more restricted. As there is still a one-to-one correspondence between transitions in both models, strong equivalence is maintained. ■

The resulting Turing machine that our RLM can simulate is now strictly less expressive than the original PTM. See Appendix F for the proof. Hence, our lower bound is strictly less powerful than the upper bound.

5 Open Questions

This work establishes upper and lower bounds on the expressive power of RLMs. While this shows how powerful RLMs can be, the bounds do not completely and precisely characterize the models of interest. A natural question for follow-up work is, therefore, the following.

Open Question 5.1. *What is the exact computational power of a rationally weighted* RLM*?*

While we do not answer this question definitively, we hope that the steps and framework outlined here help follow-up work to establish more precise descriptions of LMs in general, be it in the form of RNNs or other architectures.

We have introduced novel theoretical models of computation (RD–2PDA, RD–$\mathbb{Q}$PTM) that prove useful for describing RLMs in a formal setting due to the close connection between their dynamics and those of RNNs. We also provide a preliminary analysis of the concrete computational power of the novel models. For example, in Appendix F, we provide an example of a language that can be generated by a $\mathbb{Q}$PTM but not by its real-time deterministic counterpart, thereby showing that the former is more powerful than the latter. However, we leave a more precise characterization of their expressive power to future work. A concrete question of interest would be the relationship between deterministic (rationally weighted) PTMs and non-deterministic devices lower on the hierarchy of computational models, for example, non-deterministic probabilistic finite-state automata, which cannot be represented by deterministic finite-state automata (Mohri, 1997; Buchsbaum et al., 2000). Another question to be addressed is whether the εRLM introduced in Def. 3.2 is weakly equivalent to any non-deterministic PTM, without the need to introduce two different types of ε symbol to store the direction of the head in the outputs (Appendix E).

6 Discussion and Conclusion

The widespread deployment of LMs in more and more far-reaching applications motivates a precise theoretical understanding of their abilities and shortcomings. In this paper, we show that tools from formal language theory, namely, probabilistic Turing machines and their extensions, offer a fruitful means of investigating those abilities by allowing us to directly characterize the classes of (probabilistic) languages LMs can represent.

Concretely, we place two different formalizations of RLMs into the framework of probabilistic Turing machines, thus characterizing their computational power. To connect our results with the bigger picture of understanding RLMs, consider again Fig. 1. The upper part of Fig. 1 (left to right) expresses the equivalence of PTMs, their rationally valued equivalent, probabilistic two-stack PDAs, and RLMs with two types of ε-tokens, which allow the model to perform an unbounded number of computations in between outputting symbols. Those express the same class of semimeasures as general probabilistic Turing machines, i.e., all enumerable semimeasures over strings. These models upper bound the expressivity of RLMs with just one ε-symbol, which are equivalent to Σ–2PDA which are 2PDA that are deterministic in their output alphabet. The lower half of the Fig. 1 shows the results on the more realistic real-time RLMs with rational weight. We show that such models match the expressive power of real-time probabilistic Turing machines with rational weights (lower left) through their correspondence to real-time deterministic probabilistic 2-stack PDAs (lower right). These results provide a set of first insights into the modeling power of modern language models and hopefully provide a starting point for the investigation of other modern architectures, such as transformers (Vaswani et al., 2017).

Limitations

Here, we list several points of our analysis that we consider limiting. Similarly to Siegelmann and Sontag (1992), all our results assume the RLMs to have rationally valued weights and hidden states, which is not the case for RLMs implemented in practice. It remains to be shown if the bounded precision in practical implementations proves to be too restrictive for the LMs to learn to solve algorithmic problems. The upper bound result additionally assumes that computation time is unbounded, which is a departure from how RNNs function in practice. It is not clear how an RNN could be trained in a non-real-time manner, or if that would actually lead to better results on any of the standard NLP tasks.

Importantly, note that the lower bound result is likely not tight, as, we only show the ability of RLMs to simulate a specific computational model (namely, RD–$\mathbb{Q}$PTMs). There might be more expressive models that can also be simulated by RLMs. In general, the results presented are theoretical in nature and not necessarily a practically efficient way of simulating Turing machines. Moreover, we do not suggest that trained RNNs in practice actually implement such mechanisms, but only that they are *theoretically capable* of doing it; The construction thus serves the specific purpose of theoretically simulating PTMs and does not naturally extend to training and inference outside of problems specifically designed for Turing machines.

Ethics Statement

Our work sheds light on the purely theoretical capabilities of language models. To the best of the authors' knowledge, it does not pose any ethical issues.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank Abra Ganz and Clément Jambon for their thoughtful feedback and suggestions.

References

Adam L. Buchsbaum, Raffaele Giancarlo, and Jeffery R. Westbrook. 2000. On the determinization of weighted finite automata. *SIAM Journal on Computing*, 30(5):1502–1531.

W.A. Burkhard and P.P. Varaiya. 1971. Complexity problems in real time languages. *Information Sciences*, 3(1):87–100.

Yining Chen, Sorcha Gilroy, Andreas Maletti, Jonathan May, and Kevin Knight. 2018. Recurrent neural networks as weighted language recognizers. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2261–2271, New Orleans, Louisiana. Association for Computational Linguistics.

Stephen Chung and Hava Siegelmann. 2021. Turing completeness of bounded-precision recurrent neural networks. In *Advances in Neural Information Processing Systems*, volume 34, pages 28431–28441. Curran Associates, Inc.

Gregoire Deletang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A Ortega. 2023. Neural networks and the chomsky hierarchy. In *The Eleventh International Conference on Learning Representations*.

Li Du, Lucas Torroba Hennigen, Tiago Pimentel, Clara Meister, Jason Eisner, and Ryan Cotterell. 2023. A measure-theoretic characterization of tight language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9744–9770, Toronto, Canada. Association for Computational Linguistics.

Jeffrey L. Elman. 1990. Finding structure in time. *Cognitive Science*, 14(2):179–211.

Wilhelm Forst and Dieter Hoffmann. 2010. *Optimization—Theory and Practice*. Springer New York, NY.

John T. Gill. 1974. Computational complexity of probabilistic turing machines. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, page 91–95, New York, NY, USA. Association for Computing Machinery.

Yiding Hao, William Merrill, Dana Angluin, Robert Frank, Noah Amsel, Andrew Benz, and Simon Mendelsohn. 2018. Context-free transductions with neural stacks. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 306–315, Brussels, Belgium. Association for Computational Linguistics.

Michael A. Harrison and Ivan M. Havel. 1972. Real-time strict deterministic languages. *SIAM Journal on Computing*, 1(4):333–349.

John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D. Manning. 2020. RNNs can generate bounded hierarchical languages with optimal memory. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1978–2010, Online. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. *Introduction to Automata Theory, Languages, and Computation*, 3 edition. Pearson.

Thomas F. Icard. 2020. Calibrating generative models: The probabilistic chomsky–schützenberger hierarchy. *Journal of Mathematical Psychology*, 95:102308.

S. C. Kleene. 1956. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton.

Donald Ervin Knuth and Andrew Chi-Chih Yao. 1976. The complexity of nonuniform random number generation. In *Algorithms and Complexity: New Directions and Recent Results*, page 357–428, USA. Academic Press, Inc.

Samuel A. Korsky and Robert C. Berwick. 2019. On the computational power of RNNs. *CoRR*, abs/1906.06349.

Ming Li and Paul M.B. Vitányi. 2008. *An Introduction to Kolmogorov Complexity and Its Applications*, 3 edition. Springer Publishing Company, Incorporated.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. 2016. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535.

André F. T. Martins and Ramón F. Astudillo. 2016. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1614–1623.

Clara Meister, Tiago Pimentel, Gian Wiher, and Ryan Cotterell. 2023. Locally typical sampling. *Transactions of the Association for Computational Linguistics*, 11:102–121.

William Merrill. 2019. Sequential neural networks as automata. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 1–13, Florence. Association for Computational Linguistics.

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. 2020. A formal hierarchy of RNN architectures. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 443–459, Online. Association for Computational Linguistics.

Marvin L. Minsky. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA.

Marvin Lee Minsky. 1954. *Neural Nets and the brain model problem*. Ph.D. thesis, Princeton University.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. 2023. Resurrecting recurrent neural networks for long sequences.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Xiangru Tang, Bolun Wang, Johan S. Wind, Stansilaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui-Jie Zhu. 2023. RWKV: Reinventing RNNs for the transformer era.

Jan Pittl and Amiram Yehudai. 1983. Constructing a realtime deterministic pushdown automaton from a grammar. *Theoretical Computer Science*, 22(1):57–69.

Jorge Pérez, Javier Marinković, and Pablo Barceló. 2019. On the Turing completeness of modern neural network architectures. In *International Conference on Learning Representations*.

XiPeng Qiu, TianXiang Sun, YiGe Xu, YunFan Shao, Ning Dai, and XuanJing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897.

Michael O. Rabin. 1963. Real time computation. *Israel Journal of Mathematics*, 1(4):203–211.

Arnold L. Rosenberg. 1967. Real-time definable languages. *J. ACM*, 14(4):645–662.

Daniel M. Roy. 2011. *Computability, Inference and Modeling in Probabilistic Programming*. Ph.D. thesis, Massachusetts Institute of Technology, USA. AAI0823858.

Hava T. Siegelmann and Eduardo D. Sontag. 1992. On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 440–449, New York, NY, USA. Association for Computing Machinery.

Michael Sipser. 2013. *Introduction to the Theory of Computation*, 3 edition. Cengage Learning.

Anej Svete and Ryan Cotterell. 2023. Recurrent neural language models as probabilistic finite-state automata. *arXiv preprint arXiv:2310.05161*.

Kohtaro Tadaki, Tomoyuki Yamakami, and Jack C.H. Lin. 2010. Theory of one-tape linear-time turing machines. *Theoretical Computer Science*, 411(1):22–43.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Klaus Weihrauch. 2000. *Computable Analysis - An Introduction*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. On the practical computational power of finite precision RNNs for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia. Association for Computational Linguistics.

Sean Welleck, Ilia Kulikov, Jaedeok Kim, Richard Yuanzhe Pang, and Kyunghyun Cho. 2020. Consistency of a recurrent language model with respect to incomplete decoding. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5553–5568, Online. Association for Computational Linguistics.

Wangchunshu Zhou, Yuchen Eleanor Jiang, Peng Cui, Tiannan Wang, Zhenxin Xiao, Yifan Hou, Ryan Cotterell, and Mrinmaya Sachan. 2023. RecurrentGPT: Interactive generation of (arbitrarily) long text.

A Halting Probability and RLM

As discussed in remark 2.1, the halting probability of a PTM is defined as the sum of the probabilities of all halting paths (paths that end with q_φ). Note that EOS in a RLM and the final state q_φ in a PTM are similar constructs, and so we can consider a similar notion for RLM. However, a technical subtlety arises in defining such a quantity.

We first note that, while it is possible to define a probability measure over Σ^* with the autoregressive parameterization as in Eq. (1), not all semimeasures defined by Eq. (1) are probability measures over Σ^*. For example, in the definition of RLM (Section 2.1), if we pathologically choose a projection function $\boldsymbol{\pi}$ such that it always places zero probability on EOS, we would end up with a "language model" that places 0 probability on Σ^*.[20] In fact, Eq. (1) defines a probability measure over the set of finite *and infinite* strings: $\Sigma^* \cup \Sigma^\infty$. In the case of Σ^∞ having probability 0, we say that Eq. (1) defines a **tight** language model, and we can treat Eq. (1) as a probability measure over Σ^*. A similar situation exists in the case of a PTM, where the non-halting trajectories are infinite sequences that can be considered as elements of $\{0,1\}^\infty$.[21]

Defining probability measures over uncountable sets such as Σ^∞ or $\{0,1\}^\infty$ raises nontrivial difficulties. As a simple illustration, consider an infinite fair coin toss. The sample space for this semimeasure is $\{\mathsf{H},\mathsf{T}\}^\infty$. Clearly, each single infinite event (a binary string $\boldsymbol{\omega}$) has probability $(\frac{1}{2})^\infty = 0$. However, treating uncountable semimeasures carelessly would result in the following paradox:

$$1 = p(\{\mathsf{H},\mathsf{T}\}^\infty) = p\left(\bigcup_{\boldsymbol{\omega}\in\{\mathsf{H},\mathsf{T}\}^\infty} \{\boldsymbol{\omega}\}\right) = \sum_{\boldsymbol{\omega}\in\{\mathsf{H},\mathsf{T}\}^\infty} p(\{\boldsymbol{\omega}\}) = \sum_{\boldsymbol{\omega}\in\{\mathsf{H},\mathsf{T}\}^\infty} 0 \overset{?}{=} 0 \tag{10}$$

For reasons like this, a rigorous discussion of PTM (Roy, 2011) or RLM (Du et al., 2023) will involve a modicum of measure theory and typically starts with defining the appropriate σ-algebra. In this work, we find that introducing such technical machinery obscures our purposes and we therefore intentionally omitted them. For a rigorous discussion on the corresponding definition of halting probability in RLMs and more general autoregressive models, see Du et al. (2023).

B Versions of Probabilistic Two-stack Pushdown Automata

In our work, we use an adaptation of the traditional two-stack PDA whose transition function depends only on the top symbol of one of the stacks (and the current state), whereas usually the top symbols of both stacks are taken into account. This setup follows the proof by (Hopcroft et al., 2001) but warrants additional justification when applied to the probabilistic case.

Proposition B.1. *A* 2PDA *whose transition weighting function depends on the top symbol of both stacks can be simulated by a* 2PDA *whose transition function depends only on the top symbol of the first stack.*

Proof. ($\Longrightarrow$) Let $\mathcal{P}_1$ be a 2PDA whose transition function has the form $\delta : Q \times \Gamma^2 \times \Sigma_\varepsilon \times Q \times \Gamma^4 \to \mathbb{Q}$, such that:

$$\sum_{\substack{y\in\Sigma_\varepsilon, q'\in Q,\\ \gamma_1,\gamma_2,\gamma_3,\gamma_4\in\Gamma_\varepsilon}} \delta\left(q \xrightarrow[\gamma_2\to\gamma_4]{y,\gamma,\gamma',\gamma_1\to\gamma_3} q'\right) = 1 \tag{11}$$

We now show that we can construct a 2PDA $\mathcal{P}_2$ as defined in Def. 2.3, such that for any transition in $\mathcal{P}_1$, $\mathcal{P}_2$ has a finite sequence of transitions resulting in the same state and stack configurations. Let $q \xrightarrow[\gamma_2\to\gamma_4]{y,\gamma,\gamma',\gamma_1\to\gamma_3} q'$ be such a transition in $\mathcal{P}_1$, where γ is the top symbol on the first stack and γ' is the top symbol on the second stack. We can simulate this transition in $\mathcal{P}_2$ through the following chain of transitions, where we introduce a transition-specific new state q'':

$$q \xrightarrow[\gamma_2\to\varepsilon]{\varepsilon,\gamma,\gamma_1\to\gamma'} q'', q'' \xrightarrow[\varepsilon\to\gamma_4]{y,\gamma',\gamma'\to\gamma_3} q' \tag{12}$$

[20]For other examples where the autoregressive factorization results in Σ^* receiving < 1 probability mass, see Du et al. (2023).

[21]We can view each random branching as corresponding to either 0 or 1, thus resulting in an infinite binary string.

($\Longleftarrow$) The converse direction of proving that any such 2PDA $\mathcal{P}_1$ whose transitions depend on the top symbols of both stacks can simulate a specific 2PDA $\mathcal{P}_2$ whose transitions depend only on the top symbol of the first stack is trivial: For any transition in $\mathcal{P}_2$, we can just create a transition with the same semantics and weight 1 in $\mathcal{P}_1$ for each second stack top symbol γ'. ■

C Rationally Weighted Probabilistic Turing Machines

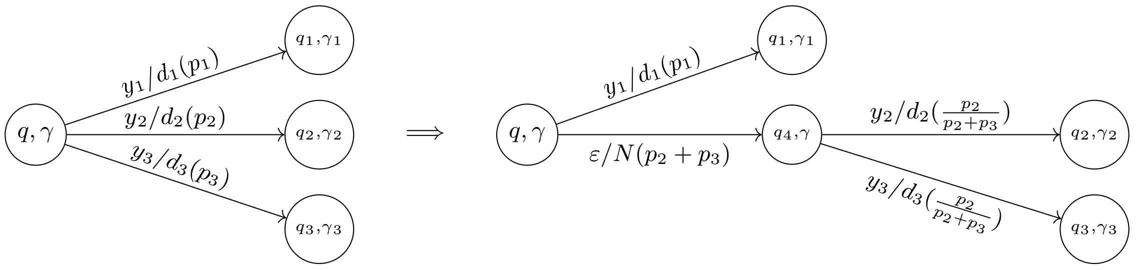

Figure 4: Binarization of an example computation graph as described in Prop. C.1.

Proposition C.1. PTM*s and* $\mathbb{Q}$PTM*s are weakly equivalent.*

Proof. ($\Longrightarrow$) The forward direction is trivial: Every PTM is a $\mathbb{Q}$PTM that has exactly two possible transitions (which might be identical) for each configuration, both with probability $\frac{1}{2}$, whereas all other symbol transitions have probability 0.

($\Longleftarrow$) We start by noting that we can transform any $\mathbb{Q}$PTM $\mathcal{M}$ into one that has exactly two possible (rational-valued) transitions at any current state and tape symbol. We do this by repeatedly applying the following transformations: For any $(q, \gamma) \in Q \times \Gamma$ that has only one possible transition, its probability is 1, so we can split it into two new identical transitions with probability $\frac{1}{2}$. For any $(q, \gamma) \in Q \times \Gamma$ that allow $k > 2$ possible transitions, we repeatedly apply the following steps:

1. We choose one of the transitions whose probability we denote by p, and leave it as it is;
2. We then create a new ε-transition with $d = N$ to a new state with probability $1-p$, leaving γ the same;
3. We then change the remaining $k-1$ transitions to start at the new state and tape symbol.

These transformations yield a $\mathbb{Q}$PTM with a completely binarized transition function. An example of this is shown in Fig. 4. Now note that any locally normalized pair of transitions with rational weights can be replaced by a sequence of transitions whose probabilities are $\frac{1}{2}$ each (Knuth and Yao, 1976; Icard, 2020). This, in conjunction with the previous transformation, allows us to convert our $\mathbb{Q}$PTM into a PTM. ■

D A PTM can simulate any 2PDA

The main idea is the same as in the forward direction of Prop. 3.2. However, there is a technicality to be considered in that the definition of the 2PDA transitions allows for transitions that have no direct equivalent in PTMs. We therefore have to make the following case distinction by types of 2PDA transitions:

(i) Any transition that pops from one stack and pushes to another has a direct equivalent in a PTM. If the $\mathcal{P}$ pops from stack one and pushes to stack two, this corresponds to a PTM transition that moves the head to the right (R). Any transition that pops from stack two and pushes to stack one can be simulated by a PTM transition moving to the left (L). Any transition that does not push or pop is a stay-in-place operation (N);

(ii) Any transition that pops from one or both stacks without pushing to the same stack(s) can be thought of as deletions that remove symbols from the PTM tape and move all the symbols on the right of the head to the left;

(iii) Transitions that push without popping correspondingly can be thought of as insertions that move all the symbols on the right of the head to the right.

To deal with cases (ii) and (iii), we need to show that any deletion or insertion can be simulated by a unique sequence of transitions in a PTM. This can be done as follows:

1. Change the current symbol under the head to a marker symbol not in the alphabet, e.g., $\downarrow$;
2. For insertions: Go to the end of the string and shift all the characters up to the marker one by one to the right.
3. Replace $\downarrow$ with the symbol to be inserted.

For deletions, step 2. changes to shifting all the symbols to the left one by one until reaching the end of the string, then moving back to the marker.

Hence, there is always a one-to-one correspondence between accepting paths in a given PTM and some $2\mathrm{PDA}$, or vice versa, meaning that the class of PTMs and $2\mathrm{PDA}$ are strongly equivalent.

E From Σ-deterministic to Fully Non-deterministic $2\mathrm{PDA}$

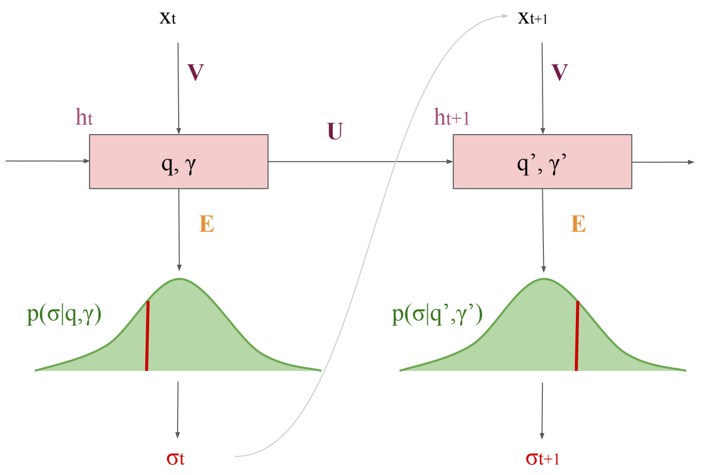

Figure 5: RLM using the controller from Chung and Siegelmann (2021).

The construction for Thm. 3.1 requires the next state and next action to be fully determined by the current configuration and the sampled output symbol, meaning all the non-determinism is contained in the sampling step (see Fig. 5). This means that we need to enforce the condition that for any configuration (q, γ), and any output symbol y (including ε), there can be at most one transition from the configuration that outputs the given symbol. We call this notion **Σ-determinism**. Note that this is not the same as determinism (Def. 4.1), where we have the additional restriction that if one ε-transition has nonzero probability, it must be the only possible transition. As shown in Appendix C, it is always possible to binarize transitions by adding ε-transitions. This allows us to enforce the single transition per output symbol restriction for any non-ε symbol, circumventing the restriction of Σ-determinism to only have one transition for each output symbol per timestep.

However, what still sets Σ-determinism apart from full non-determinism is the special case where we have multiple ε-transitions. Firstly, note that if there are more than two ε-transitions, we can utilize the binarization method (see Fig. 4). However, in the end, this still leaves us with two irreducible ε-transitions (see Fig. 6). Therefore, in order to preserve expressivity, we employ additional assumptions.

One way to address this case is to employ the following trick: When outputting ε in the $\varepsilon\mathrm{RLM}$, we can extend the output alphabet such that we include the direction of the next move together with the output, by replacing the ε-symbol with two distinguished ε-symbols, say ε_L and ε_R (ε_N can always be replaced with ε_L followed by ε_R).

See Fig. 7 for an example: Here, we have two ε-transitions that both leave the head where it is (N), but result in two different configurations. We then replace them with transitions that first move to the right (or left, respectively) with the corresponding probability, and then move back with probability 1, arriving at the desired configuration. Having added a way to distinguish these transitions when computing the new hidden state, we can keep the computation deterministic. This technique allows us to address any case where we have multiple ε-transitions, by first binarizing them and then converting them into a case where one transition goes left and the other goes right (e.g., in the case where initially both transitions move left, we can replace one of them by a ε_R followed by two consecutive ε_L transitions).

Note that the semantics of ε in the original machine stay unchanged, and we have not made a change to the operation of 2PDA, or by extension, PTMs. When reading the output of the εRLM, all ε's are still removed as before. There may be other ways of enforcing this restriction without adding a distinguished ε-output-symbol—we merely show one possible way that preserves expressivity.
We hence have the following extension to Thm. 3.1:

Proposition E.1. *An* εRLM *that can write and read in two kinds of empty tokens is weakly equivalent to the class of* PTMs*.*

Proof sketch. ($\Longrightarrow$) The forward direction is only hinted at here. The main computation works in the same way as before, but we use the two different ε-symbols to differentiate the direction of movement of the head after binarization. Then, as described above, any (non-deterministic) set of PTM transitions can be converted into a Σ-deterministic set of transitions that give the same output with the same weights, meaning we can create a weakly equivalent εRLM for any PTM.

($\Longleftarrow$) For the reverse direction, it is enough to show that for any εRLM, we can construct a PTM that computes the same semimeasure over strings. Since a PTM outputs strings in Σ^* but an εRLM outputs strings in $\Sigma_\varepsilon{}^*$, the probability of $\boldsymbol{y} = y_1 \dots y_N \in \Sigma^*$ under the desired PTM should be the sum of the probability of all strings $\boldsymbol{x} = x_1 \dots x_R \in \Sigma_\varepsilon{}^*$ under the εRLM that are equivalent to y when ε transitions are omitted. In other words, we can express the probability of a certain string $\boldsymbol{y} \in \Sigma^*$ produced by the εRLM as:

$$p(\boldsymbol{y}) = \sum_{\boldsymbol{x}\colon \mathbf{s}(\boldsymbol{x})=\boldsymbol{y}} p\left(\mathrm{EOS} \mid \boldsymbol{x}\right) \prod_{r=1}^{R} p\left(x_r \mid \boldsymbol{x}_{<r}\right) \tag{13}$$

where $\mathbf{s}\left(\boldsymbol{x}\right)$ refers to the string obtained by omitting ε's in $\boldsymbol{x}$. One readily verifies that, under a suitable choice of $\mathbf{E}$, Eq. (13) indeed defines a semimeasure over Σ^* since it is derived from the semimeasure over $\Sigma^*{}_\varepsilon$ as defined by the εRLM. Moreover, it is lower semi-computable (Def. G.2) since each $\mathbb{P}(\boldsymbol{y})$ is a

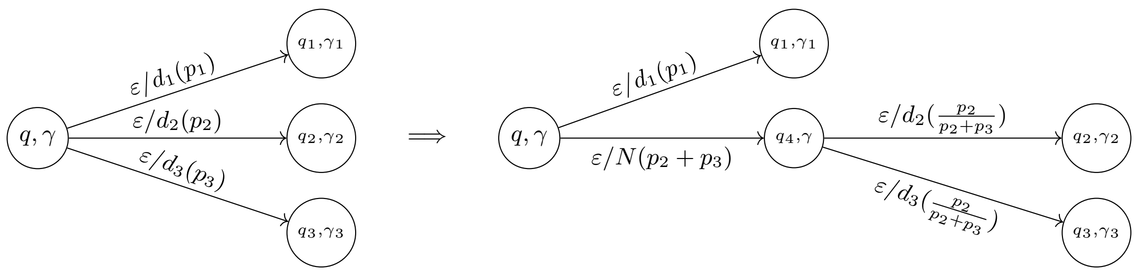

Figure 6: Binarization of multiple ε-transitions.

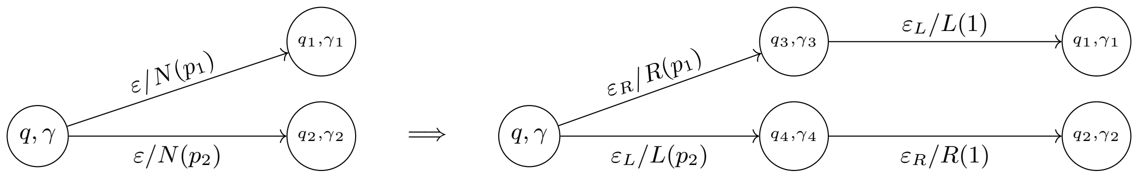

Figure 7: Resolving multiple ε-transitions with the same direction using two distinguished ε.

countable non-negative sum and hence can be approximated from below.[22] Now, we can use that by Icard (2020, Thm. 3), any enumerable (i.e., lower semi-computable) semimeasure over strings can be computed by a PTM. For completeness, we provide a proof in Appendix G. ■

F Comparison Between Different Variants of Probabilistic Turing Machines

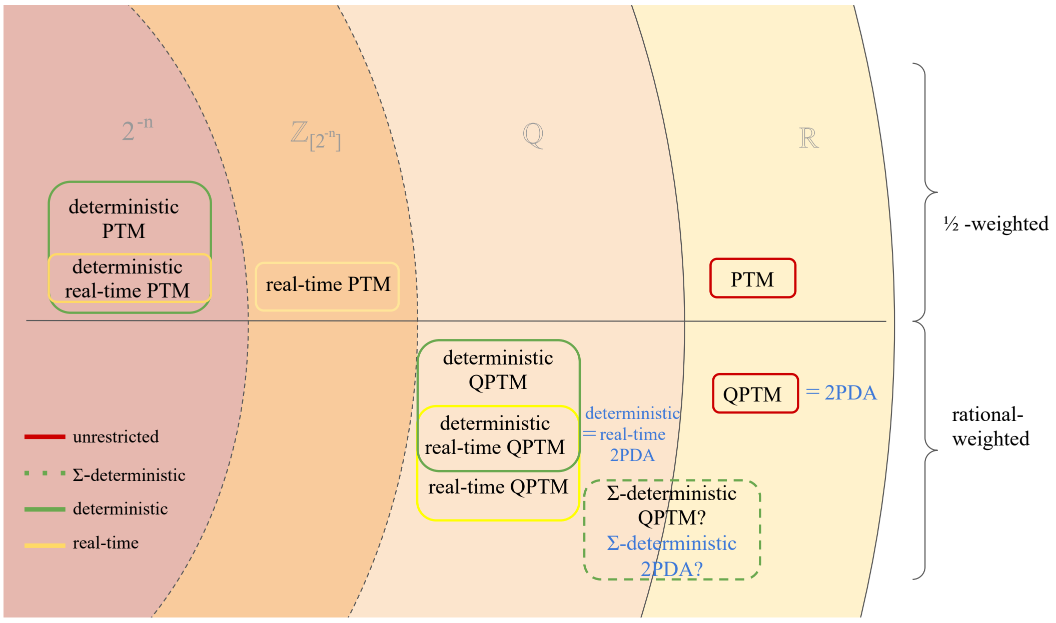

Figure 8: A schematic illustration of the different types of Turing machines and 2PDA and their corresponding place in a hierarchy of distributions. The curves differentiate different types of distributions, where 2^{-n} means every string has a binary probability, $\mathbb{Z}_{[2^{-n}]}$ refers to the dyadic distributions, and $\mathbb{Q}$ and $\mathbb{R}$ are the rational-valued and real-valued distributions, respectively. The colors of the boxes indicate which restrictions are placed on the automata, from real-time and deterministic, via Σ-deterministic, to unrestricted. Finally, the horizontal line divides formulations of machines that have just two transition functions that are uniformly distributed vs. the case of finitely many rational-values transition functions. Note that the Σ-deterministic automata are placed between the rational-valued and the real-valued distributions.

We have introduced a number of different formulations of PTMs (and 2PDA) with various additions and restrictions in the course of this work. In this section of the appendix, we compare the types of distributions that can be expressed by each of these variants of PTMs. For an illustrative overview of the distributions that can be expressed by the different models, see Fig. 8.

Definition F.1. *A* PTM *is* ***deterministic*** *if, for any configuration $q, \gamma \in Q \times \Gamma$, and any symbol $y \in \Sigma_\varepsilon$, there is at most one transition starting at that configuration and emitting y. Furthermore, if there is a transition starting in (q, γ) outputting ε, both transitions must be identical.*

Note that the definition above is more restrictive than that of a general PTM, but a superset of the class of deterministic Turing machines, $\mathcal{M}$s, which can be thought of as unweighted PTMs with the restriction that both transition functions are identical. It is a well-known result that $\mathcal{M}$s are computationally equivalent to PTMs, that is, they can recognize the same (unweighted) languages. However, in contrast to $\mathcal{M}$s, deterministic PTMs as defined above can still express a semimeasure over strings, albeit a trivial one (each string $\boldsymbol{y}$ that can be generated has a probability of $2^{-|\boldsymbol{y}|}$ of being generated).

[22]To see that Eq. (13) is a countable sum, note that when $\mathbf{s}(\boldsymbol{x}) = \boldsymbol{y}$, $\boldsymbol{x}$ can be obtained by inserting an arbitrary number of ε's in between symbols in $\boldsymbol{y}$. Hence, $\{\boldsymbol{x} \mid \mathbf{s}(\boldsymbol{x}) = \boldsymbol{y}\}$ has the same cardinality as a set of fixed-length sequences of natural numbers, which is countable. *Note that $p(\boldsymbol{y})$ may not be in $\mathbb{Q}$ even though all terms in the sum are.*

Proposition F.1. *Any deterministic* PTM *can only express distributions where each finite string has a binary probability.*

Proof sketch. This follows directly from remark 2.1. Since every string $\boldsymbol{y}$ that can be output has a unique halting path in a deterministic PTM, each such string will have a probability of the form 2^{-n}, where n is the number of non-ε computation steps. The reason for ε-transitions not counting is that by the definition of determinism, if one of the transitions is an ε-transition, then so is the other, meaning together they have probability 1. ■

Definition F.2. *A* PTM *is* ***real-time*** *if it has no ε-transitions.*

Proposition F.2. *A real-time* PTM *can express only dyadic*[23] *distributions.*

Proof sketch. The reasoning is similar to the deterministic case. By remark 2.1, every halting path has probability 2^{-n}, where n is the number of computation steps of the path. Now, in a real-time PTM, every computation step has to output a symbol, but both transition functions may output the same symbol. Hence, there may be multiple halting paths that output the same string, each of which has a binary probability, resulting in a probability that is a sum of binary fractions. In other words, every string probability in a real-time PTM is a dyadic rational. ■

A note on the language expressivity of real-time Turing machines. While deterministic and non-deterministic Turing machines can recognize or generate the same languages, it has been shown that all real-time languages are context-sensitive (Burkhard and Varaiya, 1971). In fact, there are real-time definable languages that are context-sensitive and not context-free. On the other hand, there are also context-free languages that are not real-time definable (Rosenberg, 1967). Real-time Turing machines with just one tape have been shown to only recognize regular languages (Tadaki et al., 2010). However, with just one more tape, the computational power increases dramatically (Rabin, 1963), allowing recognition of languages that are non-context-free (Rosenberg, 1967).

We now turn to the investigation of rationally weighted PTMs, i.e., $\mathbb{Q}$PTMs. First, recall that an unrestricted $\mathbb{Q}$PTM is weakly equivalent to an unrestricted PTM (Prop. C.1). Furthermore, Icard (2020, Thm. 3) showed that PTMs define exactly the enumerable semimeasures. For a reproduction of the proof, see Appendix G. This means that any enumerable real-valued semi-measure over strings can be expressed by a PTM, and, hence, a $\mathbb{Q}$PTM.

Proposition F.3. *Real-time deterministic* $\mathbb{Q}$PTM*s are strictly less expressive than general* $\mathbb{Q}$PTM*s.*

Proof. By Thm. G.1, PTMs, and hence $\mathbb{Q}$PTMs, can express real-valued semimeasures over strings. For instance, there exists a $\mathbb{Q}$PTM that generates the language Σ^* for the one-symbol alphabet $\Sigma = \{a\}$, such that the probability of a string of a certain length is given by a Poisson measure: $p(a^k) = \text{Pois}(\lambda, k) = \frac{\lambda^k e^{-\lambda}}{k!}$ for some $\lambda \in \mathbb{R}^+$. Let us choose, e.g., $\lambda = 1$. Then the probability of each string in Σ^* is an irrational number. However, an RD–$\mathbb{Q}$PTM has to output a symbol at each time step with a rational probability, and hence, there exists no RD–$\mathbb{Q}$PTM that can express the above language. ■

Note that similar arguments can be made to show that both determinism as well as real-time on their own are enough to restrict distributions from such $\mathbb{Q}$PTMs to the rationals.

Corollary F.1. *Deterministic* 2PDA *are strictly less expressive than non-deterministic* 2PDA.

Proof sketch. This follows from the one-to-one correspondence of transitions in $\mathbb{Q}$PTMs and 2PDA described in Prop. 3.2. ■

Finally, we leave the case of Σ-deterministic probabilistic automata for future work, but we hypothesize that it lies in the realm of real-valued (including irrational) distributions.

[23]A dyadic rational is a fraction whose denominator is a power of 2. We denote the set of dyadic rationals with $\mathbb{Z}_{[2^{-n}]}$.

G PTMs Can Calculate Any Enumerable Semimeasure

In this section, we restate Icard's (2020) theorem and proof showing that PTMs define exactly the set of enumerable semimeasures, albeit in our own words.

First, we recapitulate some basic notions of computation theory:

Definition G.1. *Roy (2011, Ch. II) A partial function $f : \mathbb{N} \to \mathbb{N}$ is called* ***partial computable*** *if there exists some Turing machine that halts on inputs $n \in dom(f)$ with output $f(n)$, and for inputs $n \notin dom(f)$ never halts. A function $f : \mathbb{N} \to \mathbb{N}$ is called* ***computable*** *if it is partial computable and defined everywhere, i.e., $dom(f) = \mathbb{N}$.*

We then say a set is **computably enumerable** if it is the domain of a partial computable function.

Definition G.2. *Icard (2020, Ch. 3) A real number r is* ***lower semi-computable*** *if there exists a sequence of computably enumerable rationals $\{q_n\}_{n=1}^{\infty}, q_n \in \mathbb{Q}$ that is i) monotonically increasing in n and ii) converges to r from below, i.e., $\lim_{n\to\infty} q_n = r$.*

Definition G.3. *A semimeasure μ over Σ^* is called* ***enumerable*** *if for all $\boldsymbol{y} \in \Sigma^*$ we have $\mu(\boldsymbol{y}) = r$ for some r that is lower semi-computable.*

Postulating a semimeasure μ as in Def. G.3 is equivalent to claiming that there exists a Turing machine that can associate a computable real probability with almost every element of Σ^* (Roy, 2011, Def. II.12). We now have the following theorem:

Theorem G.1. *Icard (2020, Thm. 3) Probabilistic Turing machines define exactly the enumerable semimeasures.*

Proof. We want to show that for any enumerable semimeasure $\mathbb{P}$, there exists a PTM $\mathcal{M}$ with semimeasure $\mathbb{P}_{\mathcal{M}}$ such that $\mathbb{P} = \mathbb{P}_{\mathcal{M}}$. Let $\mathbb{P}$ be an enumerable semimeasure over Σ^*. That means for any string $\boldsymbol{y} \in \Sigma^*$ there exists a computably enumerable monitonically increasing sequence of rationals $\{q_n\}_{n=1}^{\infty}$ converging to to $\mathbb{P}(\boldsymbol{y})$ from below (G.2 and G.3). Letting $q_0 = 0$, we have that $\mathbb{P}(\boldsymbol{y}) = \sum_{i=0}^{\infty}(q_{i+1} - q_i)$. Tab. 1 shows the ordered strings with their corresponding approximation sequences.

i	$\boldsymbol{y}_0$	$\boldsymbol{y}_1$	$\boldsymbol{y}_2$	$\dots$	$\boldsymbol{y}_k$	$\dots$
0	q_0^0	q_0^1	q_0^2	$\dots$	q_0^k	$\dots$
1	q_1^0	q_1^1	q_1^2	$\dots$	q_1^k	$\dots$
2	q_2^0	q_2^1	q_2^2	$\dots$	q_2^k	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	
i	q_i^0	q_i^1	q_i^2	$\dots$	q_i^k	$\dots$

Table 1: Ordered strings with their corresponding approximation sequences

Now, let $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a computable bijective pairing function. Without loss of generality, we can let π be the Cantor pairing function, defined as:

$$\pi(x, y) \overset{\text{def}}{=} \frac{1}{2}(x + y)(x + y + 1) + y \tag{14}$$

Let $\pi_1 : \mathbb{N} \to \mathbb{N}$ a projection function such that $\pi_1(n) = x$ for $n = \pi(x, y)$. Let $\{\boldsymbol{y}_k\}_{k=0}^{\infty}$ be a fixed enumeration of the strings in Σ^*, each with a fixed enumeration of rationals $\{q_i^k\}_{i=1}^{\infty}$ converging to $\mathbb{P}(\boldsymbol{y}_k)$ from below. We now define a sequence of rationals $\{r_n\}_{n=0}^{\infty}$ inductively as follows:

$$r_0 = q_0^0 \tag{15}$$

$$r_{n+1} = r_n + (q_{i+1}^k - q_i^k) \tag{16}$$

where $n + 1 = \pi(k, i)$. Note that this means that differences the increment between two rs is exactly the difference between two consecutive approximating rationals q_i^k and q_{i+1}^k, as:

$$r_{n+1} - r_n = q_{i+1}^k - q_i^k \tag{17}$$

This means the differences of rationals r are a Cantor enumeration of the differences of rationals approximating $\mathbb{P}(\boldsymbol{y})$. Note that the sequence $\boldsymbol{y}$ has its probability mass $\mathbb{P}(\boldsymbol{y})$ distributed over intervals of consecutive increasing rationals q:

$$\mathbb{P}(\boldsymbol{y}_k) = \sum_{i=0}^{\infty}(q_{i+1}^k - q_i^k) = \sum_{n:\pi_1(n+1)=k}^{\infty} (r_{n+1} - r_n) \tag{18}$$

So with every time step, we add a bit of probability mass from one of the possible strings to r. This way, r will in the limit cross the entire interval between 0 and $\sum_{\boldsymbol{y}} \mathbb{P}(\boldsymbol{y})$. To select a string $\boldsymbol{y}_k$ according to its probability, we need to randomly select a point in that interval with enough accuracy to uniquely identify the pair of approximating rationals that crossed that point when listing the sequence of rs.

We can do this as follows: By definition, our PTM $\mathcal{M}$ has access to a sequence of random bits $\{b_j\}_j = 0^\infty$. We take increasingly long substrings of this sequence as approximations to some random real number between 0 and 1, where an approximation up to the t^{th} bit is given by $\tilde{p} = \sum_{i=0}^{t} b_i 2^{-i}$.

We now let $\mathcal{M}$ read one bit b_t and calculate $\tilde{p}$, calculate the next r_t, and check whether, in the series of rs so far produced, there is a pair $r_n, r_n + 1$, such that the the following inequalities hold:

$$r_n < \sum_{i=0}^{t} b_i 2^{-i} - 2^{-t} \text{ and } r_{n+1} > \sum_{i=0}^{t} b_i^{-i} + 2^{-t} \tag{19}$$

That is, the random real number lies within the interval $(\tilde{p} - 2^{-t}, \tilde{p} + 2^{-t})$, and that interval lies between r_n and r_{n+1} meaning that regardless of the next random bit, it will not be taken out of that interval.

If there is such an interval of r_n, r_{n+1}, $\mathcal{M}$ halts and emits the corresponding string $\boldsymbol{y}_k$, where $k = \pi_1(n+1)$. The probability of $\mathcal{M}$ emitting $\boldsymbol{y}_k$ is exactly $\mathbb{P}(\boldsymbol{y}_k)$, and the probability of $\mathcal{M}$ running forever is $1 - \sum_{\boldsymbol{y}} \mathbb{P}(\boldsymbol{y})$. ■